IE 678 Deep Learning

01 – Introduction

Prof. Dr. Rainer Gemulla

Universität Mannheim

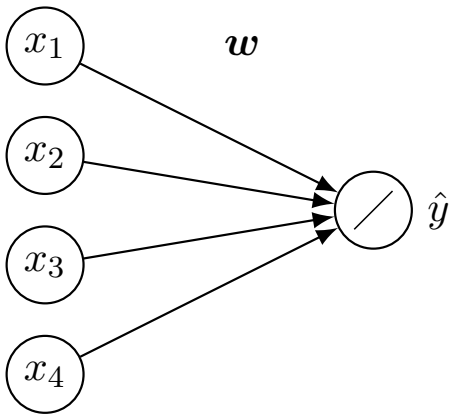
Version: 2025-1

Artificial Neural Network (ANN)

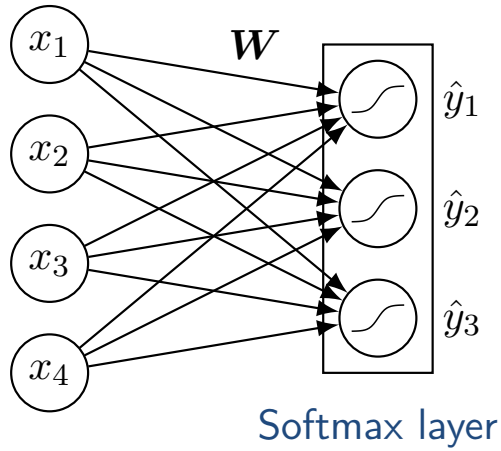
- A powerful family of models inspired by biological neural networks
  - Hope: should be good at what humans are good at
  - Many relationships to what we learned so far (in the ML course)
- Studied to
  - Understand how the brain works (**connectionism**, not covered here)
  - Build learning machines
- Usually, ANNs have an "input layer" and an "output layer"; they are used for a variety of learning tasks
  - Classification, regression, prediction, clustering, ...
  - Feature generation / dimensionality reduction
  - Supervised, semi-supervised, few-shot, unsupervised
  - Generative, discriminative
- Deep learning = ANN with multiple "hidden" layers
  - Deep does not always mean many hidden layers
  - Tremendous success across many applications in recent years

Feedforward neural networks

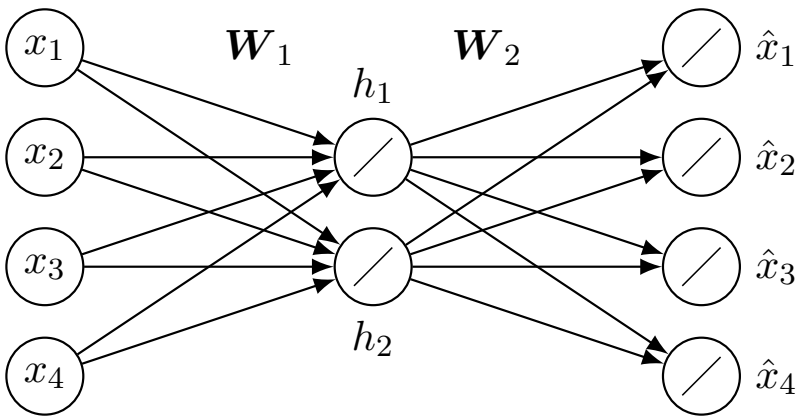
We start with simple FNN architectures and their relationship to models we know:



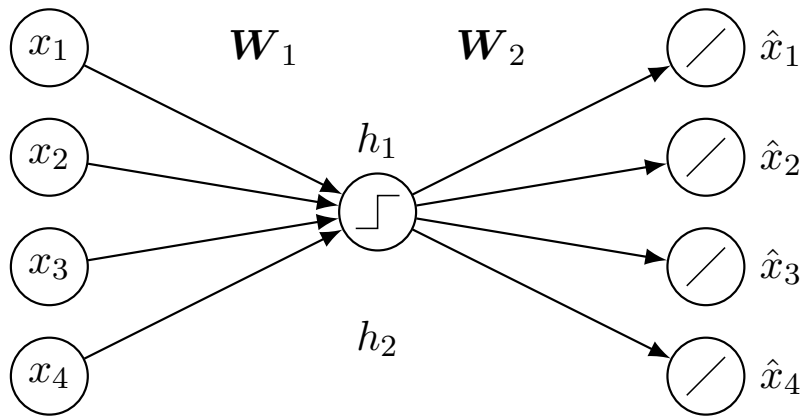
Linear regression



Softmax regression



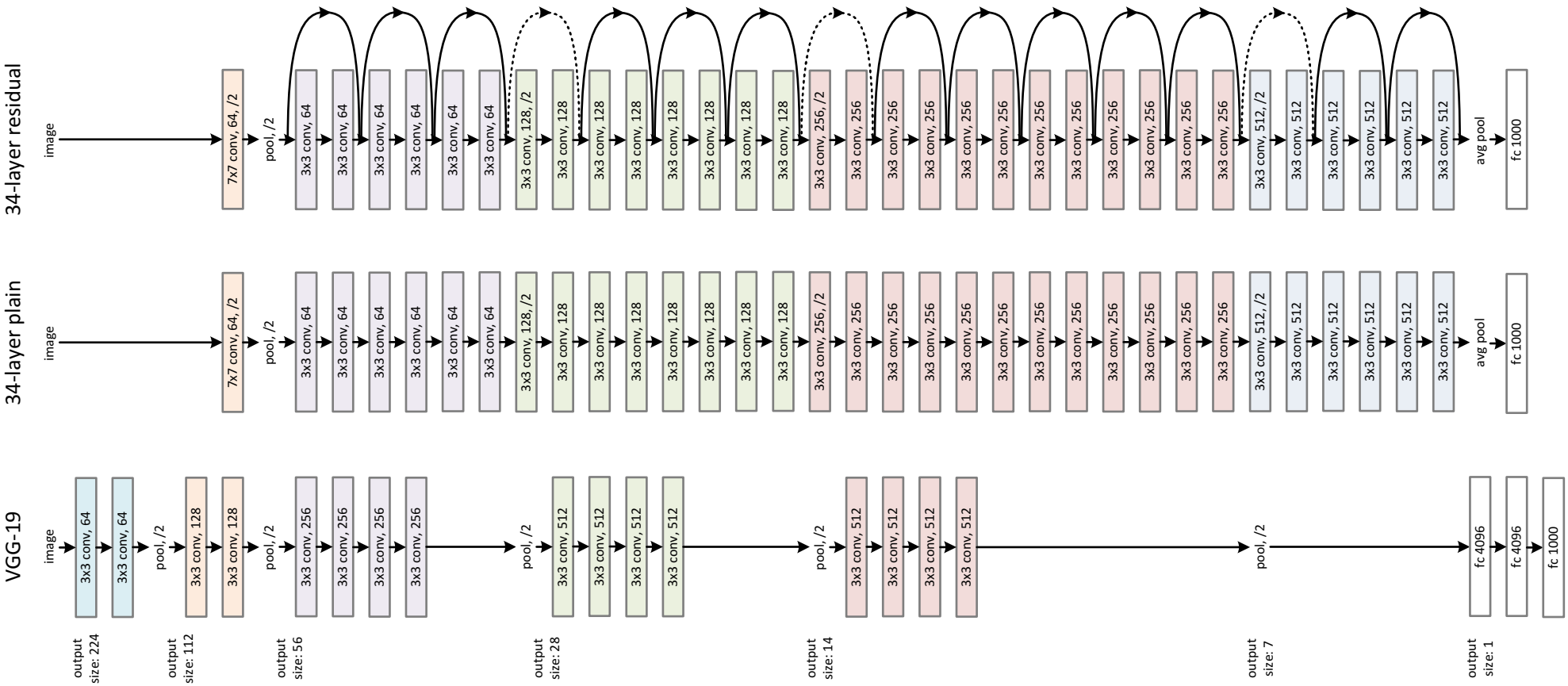
Singular value decomposition



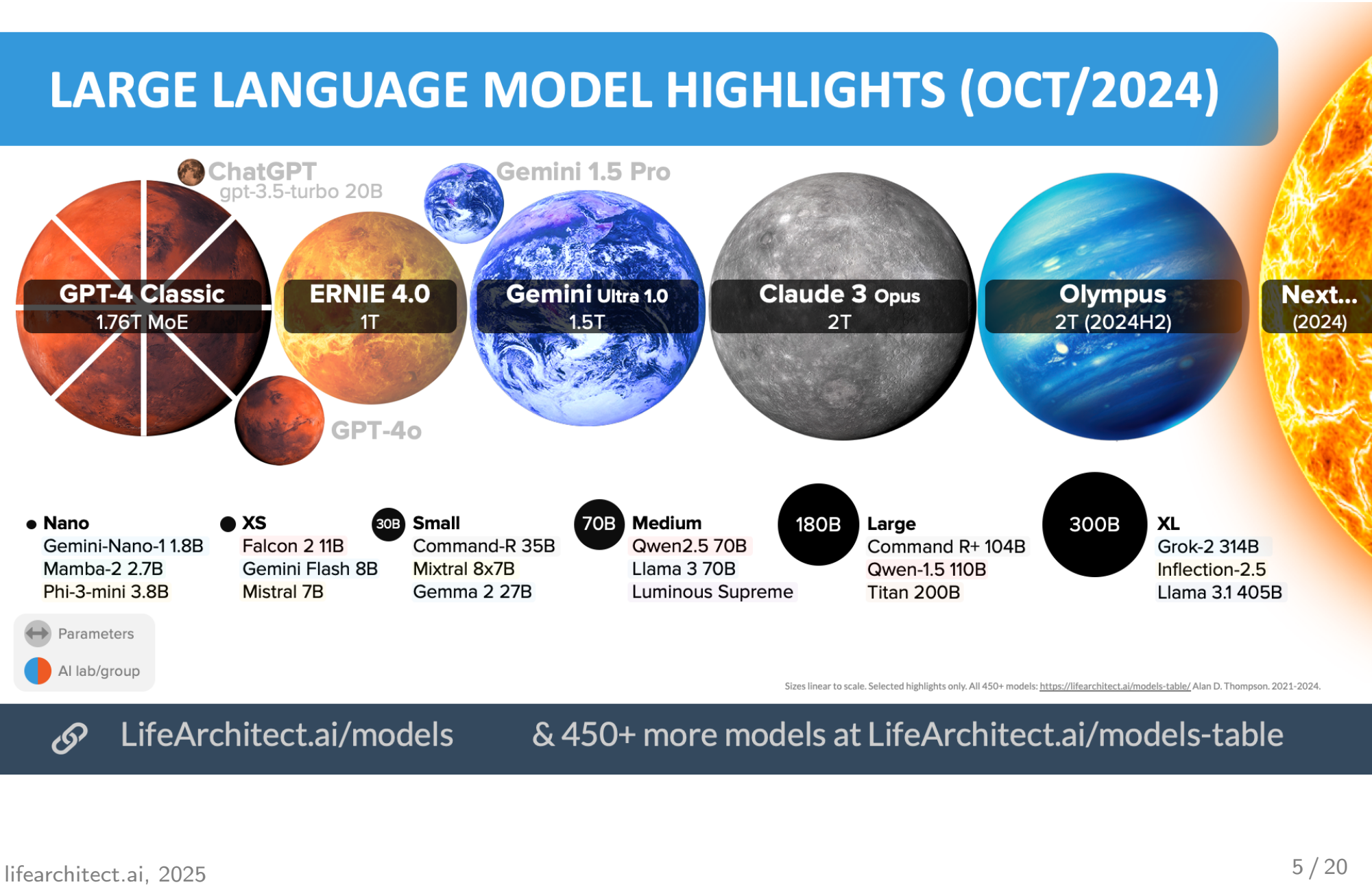
K-means clustering

FNNs can get much more complex...

Example architectures for image classification tasks



. . . and larger. . .


LARGE LANGUAGE MODEL HIGHLIGHTS (OCT/2024)
ChatGPT
gpt-3.5-turbo 20B
Gemini 1.5 Pro
GPT-4 Classic
1.76T MoE
ERNIE 4.0
1T
Gemini Ultra 1.0
1.5T
Claude 3 Opus
2T
Olympus
2T (2024H2)
Next...
(2024)
GPT-4o
Nano
Gemini-Nano-1 1.8B
Mamba-2 2.7B
Phi-3-mini 3.8B
XS
Falcon 2 11B
Gemini Flash 8B
Mistral 7B
30B
Small
Command-R 35B
Mixtral 8x7B
Gemma 2 27B
70B
Medium
Qwen2.5 70B
Llama 3 70B
Luminous Supreme
180B
Large
Command R+ 104B
Qwen-1.5 110B
Titan 200B
300B
XL
Grok-2 314B
Inflection-2.5
Llama 3.1 405B
Parameters
AI lab/group
Sizes linear to scale. Selected highlights only. All 450+ models: https://lifearchitect.ai/models-table/ Alan D. Thompson. 2021-2024.
LifeArchitect.ai/models
& 450+ more models at LifeArchitect.ai/models-table

. . . and more powerful

text-to-image (DALL-E 1/2)

an espresso machine that makes coffee from human souls, artstation

panda mad scientist mixing sparkling chemicals, artstation

a corgi's head depicted as an explosion of a nebula

text-to-text, few-shot learning (GPT-4)

Benchmark	**GPT-4** Evaluated few-shot	**GPT-3.5** Evaluated few-shot
MMLU Multiple-choice questions in 57 subjects (professional & academic)	86.4% 5-shot	70.0% 5-shot
HellaSwag Commonsense reasoning around everyday events	95.3% 10-shot	85.5% 10-shot
AI2 Reasoning Challenge (ARC) Grade-school multiple choice science questions. Challenge-set.	96.3% 25-shot	85.2% 25-shot
WinoGrande Commonsense reasoning around pronoun resolution	87.5% 5-shot	81.6% 5-shot
HumanEval Python coding tasks	67.0% 0-shot	48.1% 0-shot
DROP (f1 score) Reading comprehension & arithmetic.	80.9 3-shot	64.1 3-shot

. . . and more expensive

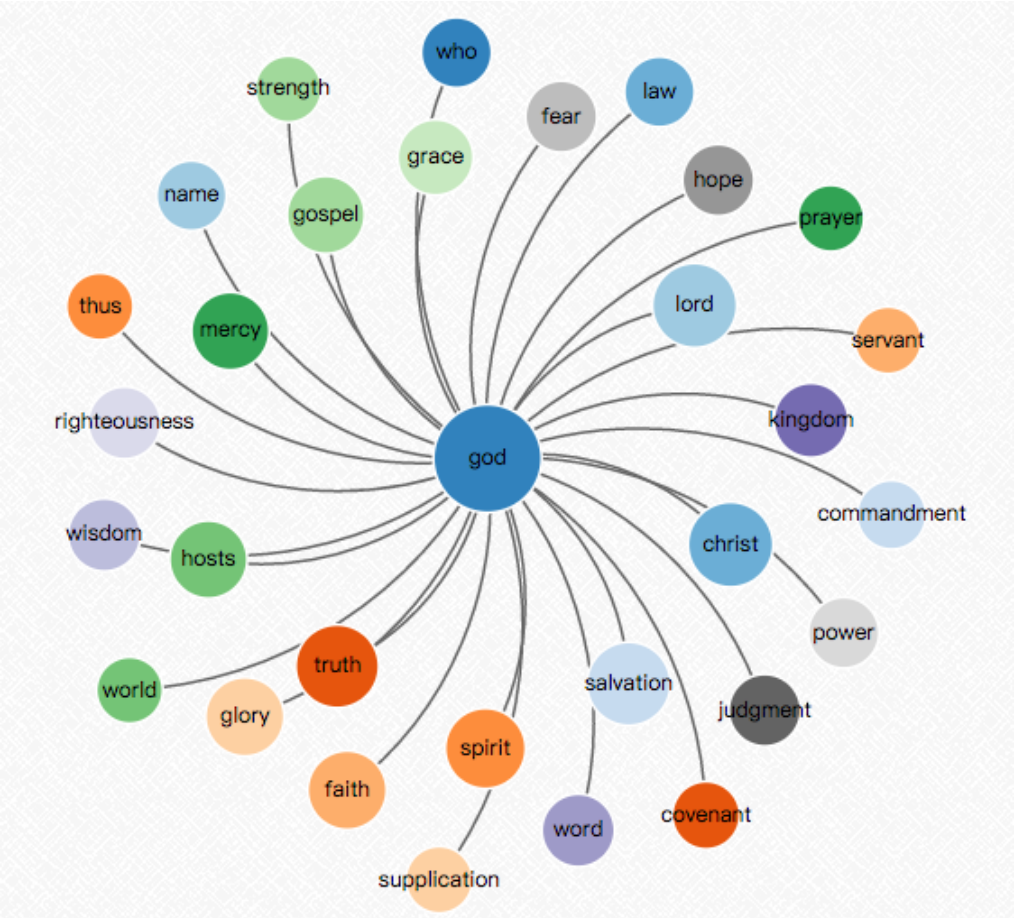
Model	Training end	Chip type	TFLOP/s (max)	Chip count	Wall clock (days)	Total time (years)	Retail (US$)	MMLU
GPT-3 175B	Apr/2020	V100	130	10,000	15 days	405y	$9M	43.9
Llama 1 65B	Jan/2023	A100	312	2,048	21 days	118y	$4M	63.4
Llama 2 70B	Jun/2023	A100	312	*2,048*	35 days	196y	$7M	68.0
Titan 200B	Apr/2023	A100	312	13,760	48 days	1,319y	$45M	70.4
GPT-4 1.7T	Aug/2022	A100	312	25,000	95 days	6,507y	$224M	86.4
Gemini	Nov/2023	TPUv4	275	57,000	100 days	15,616y	$440M	90.0
Llama 3 405B	Apr/2024	H100	989	24,576	50 days	3,366y	$125M	*85+*
GPT-5	*Apr/2024*	H100	989	*50,000*	*120 days*	*16,438y*	$612M	
Grok 2	*Jun/2024*	H100	989	20,000	*50 days*	*6,571y*	$245M	
Olympus	*Aug/2024*	H100	989					
Gemini 2	*Nov/2024*	TPUv6	*1,847*					
Grok 3	*Dec/2024*	H100	989	100,000	*50 days*	*32,855y*	$1.2B	
								Alan D. Thompson. May/2024. LifeArchitect.ai

Table. Model training compute (see working, with sources[8]).

As of May 2024; *italic* = estimate at that time

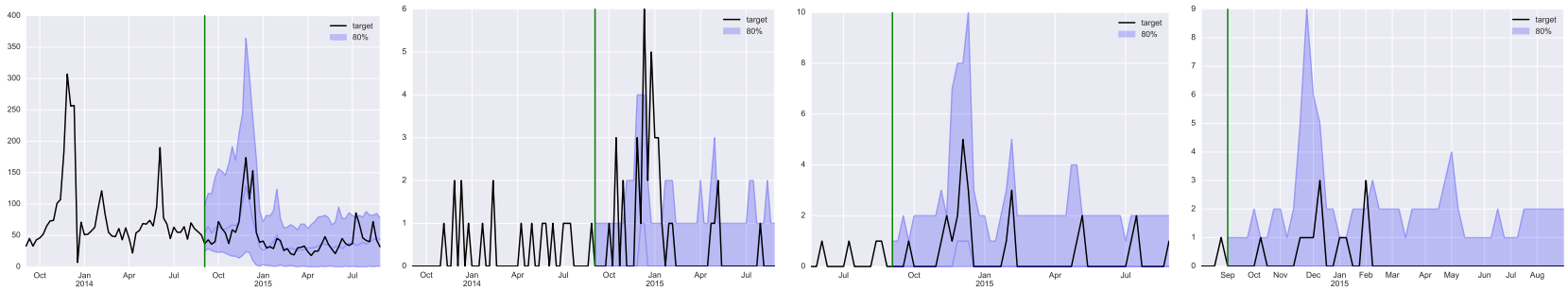
Key concept: Embeddings

- **Embeddings** are learned dense, continuous, low-dimensional representations of objects
  - Useful to represent complex objects and/or parts of objects (categorical, text, time series, audio/image/video, graph, tables, . . .)
  - Think: complex to work with objects, simple to work with embeddings
- Example: Top-30 closest word vectors to "God", trained on the Bible



Recurrent neural networks (RNN)

- **Recurrent neural networks (RNN)** are a family of neural networks for processing *sequential data*
  - Time series data (e.g., sequences of sensor readings)
  - Natural language text (e.g., sequences of characters of words)
  - Audio signals (e.g., sequences of amplitudes)
  - Images (e.g., sequences of pixels or rows)
  - Videos (e.g., sequences of frames)
  - Actions (e.g., movement of pen)
  - ...
- Example: DeepAR for probabilistic forecasting
  - Focus: scenarios with many related time series (energy consumption of individual households, demand of products)



- Related: (deep) **state space models (SSM)**

Flunkert et al., 2017

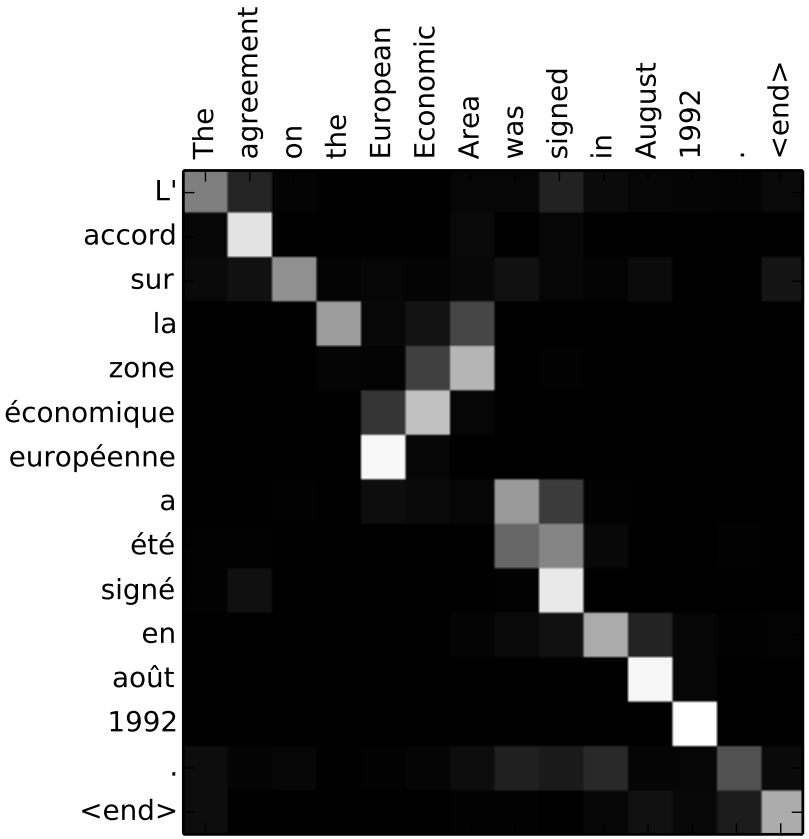
Convolutional neural networks (CNN)

- **Convolutional neural networks (CNN)** are a family of neural networks for processing *grid data*
  - Grid data means: neighboring points related
  - 1D grid $\rightarrow$ sequential data (e.g., time series, text, audio, ...)
  - 2D grids (images), 3D grids (movies, CT scans)
- Example: artistic style transfer

Quartz

Attention and Transformers

- **Attention** is a mechanism to summarize *multiple inputs*, often focusing on a small, dynamic subset of the inputs
- Can be used in conjection with other architectures (e.g., RNNs with attention) or standalone (e.g., **Transformers**)



Translation

A woman is throwing a frisbee in a park.

Image captioning

Bahdanau and Cho, 2015; Xu et al., 2016

Graph neural networks

- **Graph neural networks (GNNs)** are a family of neural networks for processing *graph data*
- Example: molecular property prediction

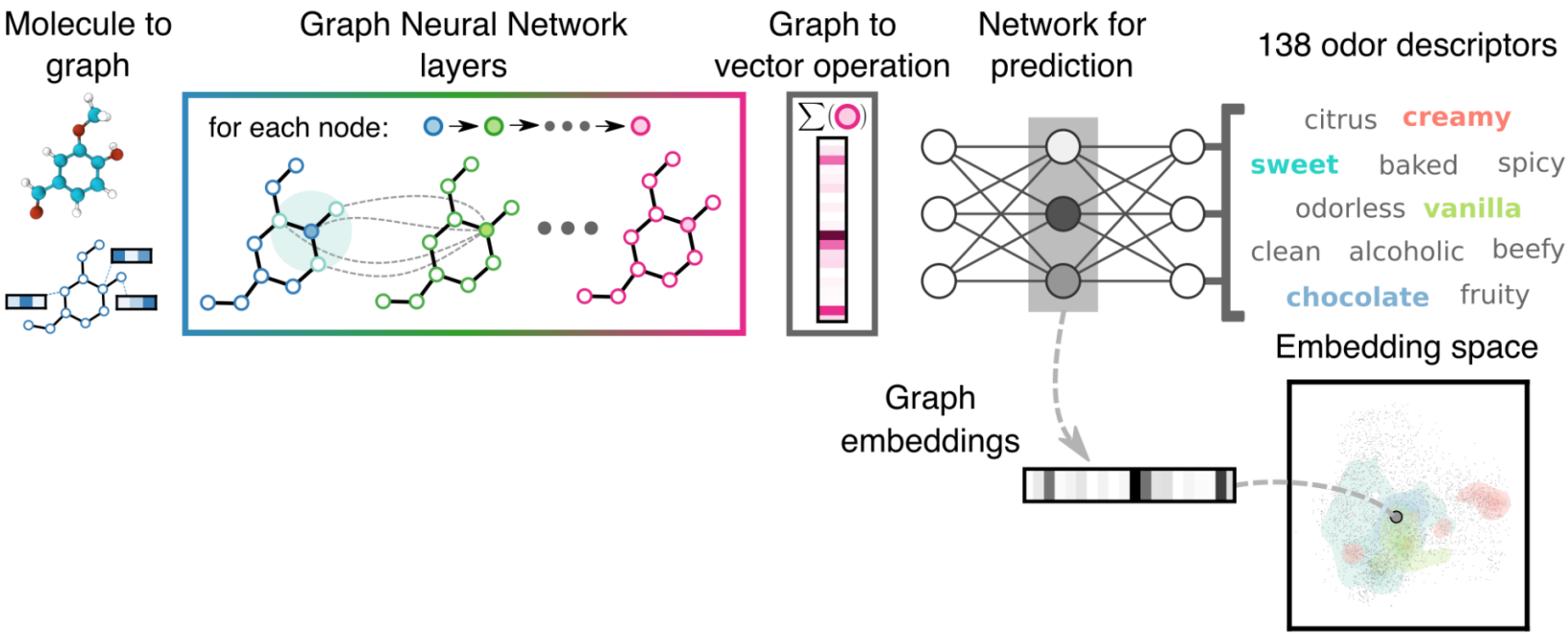
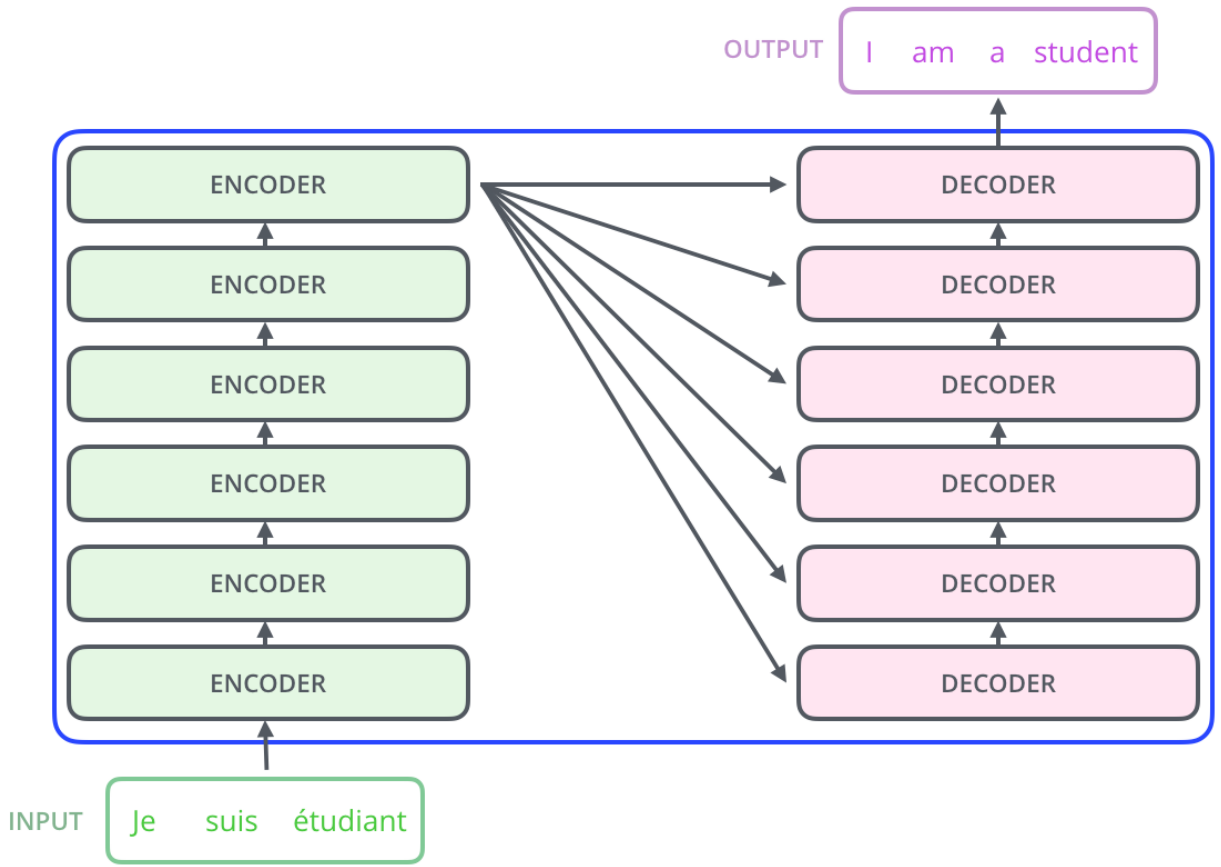


Figure 2: **Model Schematic.** Each molecule is first featurized by its constituent atoms, bonds, and connectivities. Each Graph Neural Network (GNN) layer, here represented as different colors, transforms the features from the previous layer. The outputs from the final GNN layer is reduced to a vector, which is then used for predicting odor descriptors via a fully-connected neural network. We retrieve graph embeddings from the penultimate layer of the model. An example of the embedding space representation for four odor descriptors is shown in the bottom right; the colors of the regions in this plot correspond to the colors of odor descriptors in top right.

Sanchez-Lengeling et al., 2019

Deep generative models

- **Deep generative models** use deep neural networks to define generative models for complex data distributions (e.g., text, audio, image, graphs, . . .)
  - Our focus: AR, perhaps: VAEs, GANs
- Example: T5, GPT-1/2/3/GPT-4, ChatGPT



Deep learning frameworks

- In practice, neural networks are usually trained using deep learning frameworks such as PyTorch, TensorFlow or JAX
  - Linear algebra / array processing
  - Common units, layers, models, loss functions
  - Preprocessing and data preparation methods
  - Common optimization methods
  - GPU support, parallelization
  - Support for model deployment
  - Facilities for debugging and visualization
  - ...
- Generally, to perform gradient-based parameter estimation
  - Programmers specify model (e.g., implement **forward pass**)
  - When used on training data, framework collects operations and their outputs to build **computation graph**
  - Gradient computation performed automatically from this computation graph using **backpropagation**
  - **Optimizer** uses gradient to update model
- No need to compute gradients manually, yet understanding of backpropagation and optimization methods is important

Example: PyTorch

```
# define model with one hidden layer
model = torch.nn.Sequential(
    torch.nn.Linear(dim_in, dim_hidden),
    torch.nn.ReLU(),
    torch.nn.Linear(dim_hidden, dim_out),
)

# define loss function (mean squared error)
loss_fn = torch.nn.MSELoss()

# pick optimizer (Adam)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

# run for 500 epochs
for t in range(500):
    y_pred = model(X) # forward: model output (X = examples)
    loss = loss_fn(y_pred, y)  # forward: loss (y = labels)
    model.zero_grad()          # clear old gradients
    loss.backward()            # backward: compute gradients
    optimizer.step()           # update parameters
```

Training (1)

- First, the *tool*: gradient-based methods to minimize some cost function (backpropagation, optimizers)
- Challenge 1: large, complex models
  - Fully-connected layer with n inputs and m units: $O(nm)$ parameters
  - 10 dense layers, each 200 inputs/units $\rightarrow$ 400k parameters
  - 1 dense layer, $1M$ inputs, 200 units $\rightarrow$ 200M parameters
  - E.g., T5 text-to-text transformer for NLP (small: 60M, base: 220M, large: 770M)
  - E.g., EfficientNet for CV (B0: 5.3M, B1: 7.8M, ..., B7: 66M)
- Challenge 2: limited training data
  - Large labeled datasets generally not available
  - Supervision signal alone may be insufficient to achieve reasonable performance

Training (2)

- Overfitting is a severe concern
  - Universal approximation theorem: with sufficiently many hidden neurons, FNN can perform arbitrarily well on the *training* set
  - Sufficiently large models needed for complex tasks
- Then, the *art*: selected techniques and tricks for deep learning
  - Training process and general techniques
  - Architectures
  - Both very important
- Generally, goals include
  - Improve performance of gradient-based methods
  - Reduce overfitting, improve generalizability
  - Leverage additional data
  - Reduce (task-specific) costs such as model size, computational costs, enery consumption, amount of required supervision, ...

Goals of this lecture

1. Solid understanding of key concepts
   - Architectures
   - Design patterns
   - Training methods
   - Systems
2. Ability to explore the deep learning literature on your own
   - SOTA changes at fast pace $\rightarrow$ enable you to stay up-to-date
   - Prerequisite/helpful for related lectures
3. Hands-on experience with selected frameworks and models
   - Use and adapt pre-built DL models
   - Be able to design, implement, train, and evaluate custom models / training techniques

After this course, you should be able to read, understand, apply, criticize, modify, and create DL models & techniques all by yourself.

Key applications and related lectures

- Computer vision
  - E.g., image classification, object detection, image segmentation, image generation, image restoration, ...
  - Cf. CS 646 Higher Level Computer Vision (HWS)
  - Cf. CS 668 Generative Computer Vision Models (FSS)
- Natural language processing
  - E.g., parsing, sentiment analysis, information retrieval, machine translation, chat bots, ...
  - Cf. IE 696 Advanced Methods in Text Analyticss (FSS)
  - Cf. IE 686 Large Language Models and Agents (FSS)
- Structured data; e.g., sequences/graphs/relational data
  - E.g., recommender systems, drug discovery and toxicology, CRM, bioinformatics, mobile advertising, financial fraud detection, relational learning, data integration, ...
  - Cf. IE 670 Web Data Integration (HWS)
- Generally, AI: robotics, gaming, planning, ...
  - Not in this course
  - Cf. IE 695 Reinforcement Learning (HWS)

Syllabus

Introduces basic and advanced deep learning architectures, key techniques and training methods, systems, and selected applications.

- Feedforward neural networks
- Backpropagation and parameter optimization
- Machine learning systems
- Training techniques for deep learning models
- Recurrent neural networks / state space models
- Convolutional neural networks
- Attention and Transformers
- Deep learning for graphs
- Deep generative modeling

IE 678 Deep Learning

02 – Feedforward Neural Networks

Part 0: Overview

Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-1

Outline

1. Embeddings
2. Feedforward Neural Networks
3. Linear Layers
4. Non-Linear Layers
5. Multi-Layer Perceptrons

Lessons learned

- Artificial neural networks
  - Useful for a variety of learning tasks, great results in some areas
  - Complex models, need data + compute + experience
- Feedforward neural networks
  - Discriminative models, directed flow from input to output
  - Hidden layers enable high representational capacity
  - Outputs of hidden layers can be seen as learned features (embeddings)
  - Train with backprop + tricks + tricks + tricks (see later lectures)
- Basic ML models can be represented as FNNs
  - Linear/logistic/softmax regression (no hidden layer)
  - SVD and k-Means clustering (one hidden layer)
- . . . and are a building block of more complex DL models
  - E.g., as prediction head
  - E.g., as artificial neuron

Suggested reading

- Drori, Ch. 1, 2.1–2.4
- Goodfellow et al., Ch. 6+7
- Murphy 1, Ch. 13.1+13.2

IE 678 Deep Learning

02 – Feedforward Neural Networks

Part 1: Embeddings

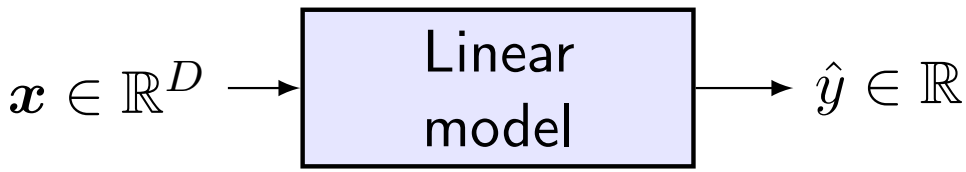
Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-1

From linear models to FNNs (1)

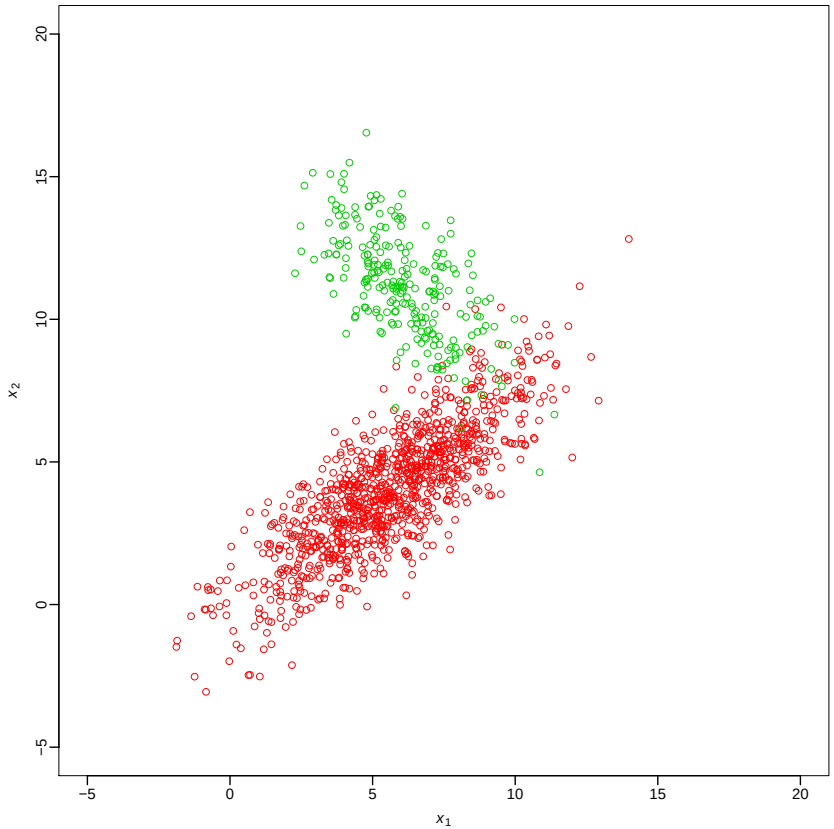
- Consider: prediction task with inputs $x \in \mathcal{X}$ and outputs $y \in \mathcal{Y}$
  - Goal: learn a function from $\mathcal{X}$ to $\mathcal{Y}$
- Simple approach: use a (generalized) **linear model**
  - Inputs must be real-valued feature vectors $\boldsymbol{x} \in \mathbb{R}^D$
  - Outputs are a real value (e.g., linear or logistic regression)
- Visually:



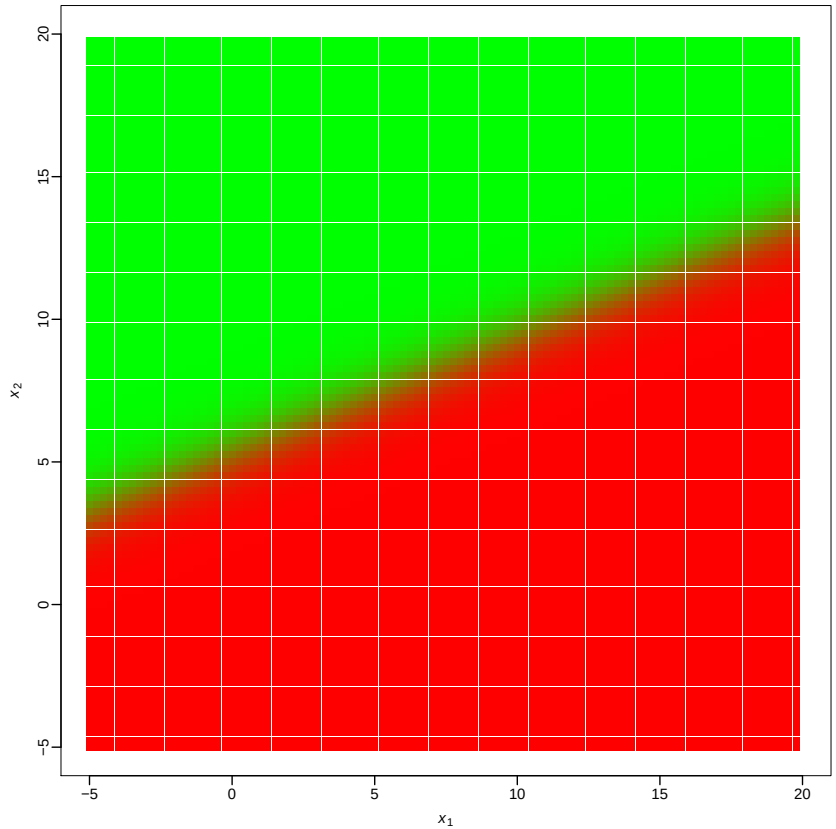
- Recall (ML, 04-1): $\hat{y} = \phi(\boldsymbol{w}^\top \boldsymbol{x} + b)$, where
  - $\boldsymbol{w} \in \mathbb{R}^D$ is a **weight vector** (one weight per feature, learned)
  - $b \in \mathbb{R}$ is a **bias term** (learned)
  - ϕ is a **mean function** (e.g., identity or logistic function)
- Problem: low representational capacity due to linearity assumption

Example: Logistic regression (from ML course)

Data ($\boldsymbol{x} \in \mathbb{R}^2$)

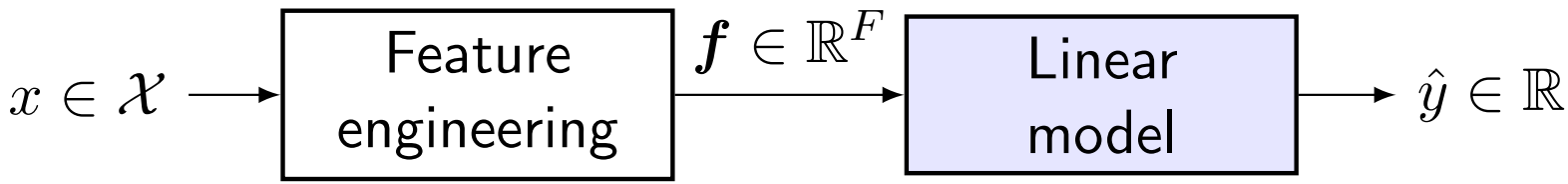


Prediction ($\hat{y} \in [0, 1]$))



From linear models to FNNs (2)

- Representational capacity can be addressed by feature engineering or using kernel methods (ML, 08)
  - Allows to use arbitrary inputs spaces $x \in \mathcal{X}$ by mapping them to real-valued vectors
  - To do so, uses *pre-specified* feature extractor $f : \mathcal{X} \to \mathbb{R}^F$
- Visually:

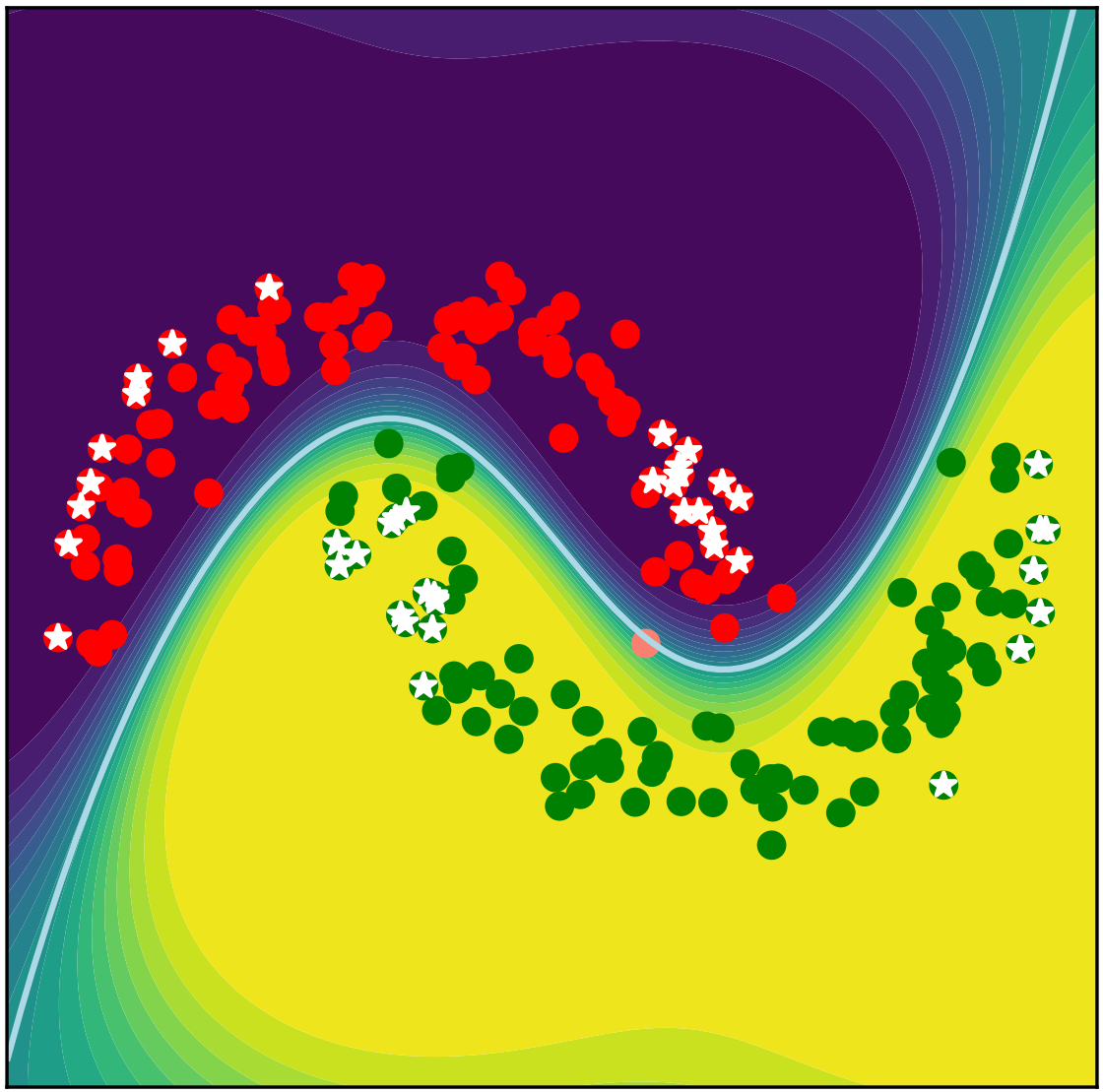


- Problem: which feature extractor?
  - Key to good performance
  - Hard to get right (domain experts, extensive experimentation, . . .)
  - To see this: can you write a suitable feature extractor for classifying images? (if not, see here)

Example: L1VM (from ML course)

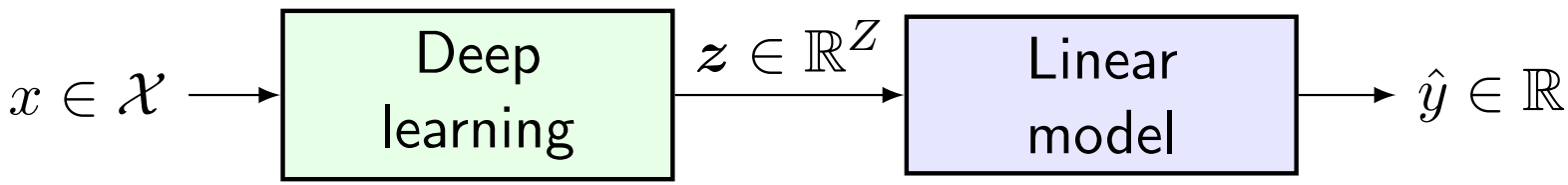
L1VM, RBF kernel, logistic regression

$\lambda = 0.1$, $\sigma^2 = 0.571$



From linear models to FNNs (3)

- DL methods can be interpreted as an approach to *learn* features
  - Input objects $x \in \mathcal{X}$ are transformed into dense, continuous, low-dimensional representations called **embeddings** $\boldsymbol{z} \in \mathbb{R}^Z$
  - $Z =$ **embedding dimensionality**
  - Useful to represent complex objects (categorical data, textual data, graph data, tabular data, images, ...)
  - Think: complex to work with objects, simple to work with embeddings
  - Useful **embedding space** = goal of **representation learning**
- Visually:



- Key point: instead of engineering features manually, embeddings are learned from data $\rightarrow$ Main topic of this course
  - Embeddings also called: **latent code**, **distributed representations**
  - Embedding space also called: **latent space**

Example: Document embeddings

804414 newswire stories, inputs = per-document rel. frequencies of 2000 most common word stems ($x \in \mathbb{R}^{2000}$), shown here is 2D embedding ($z \in \mathbb{R}^2$) of two different methods (left: linear, right: autoencoder)

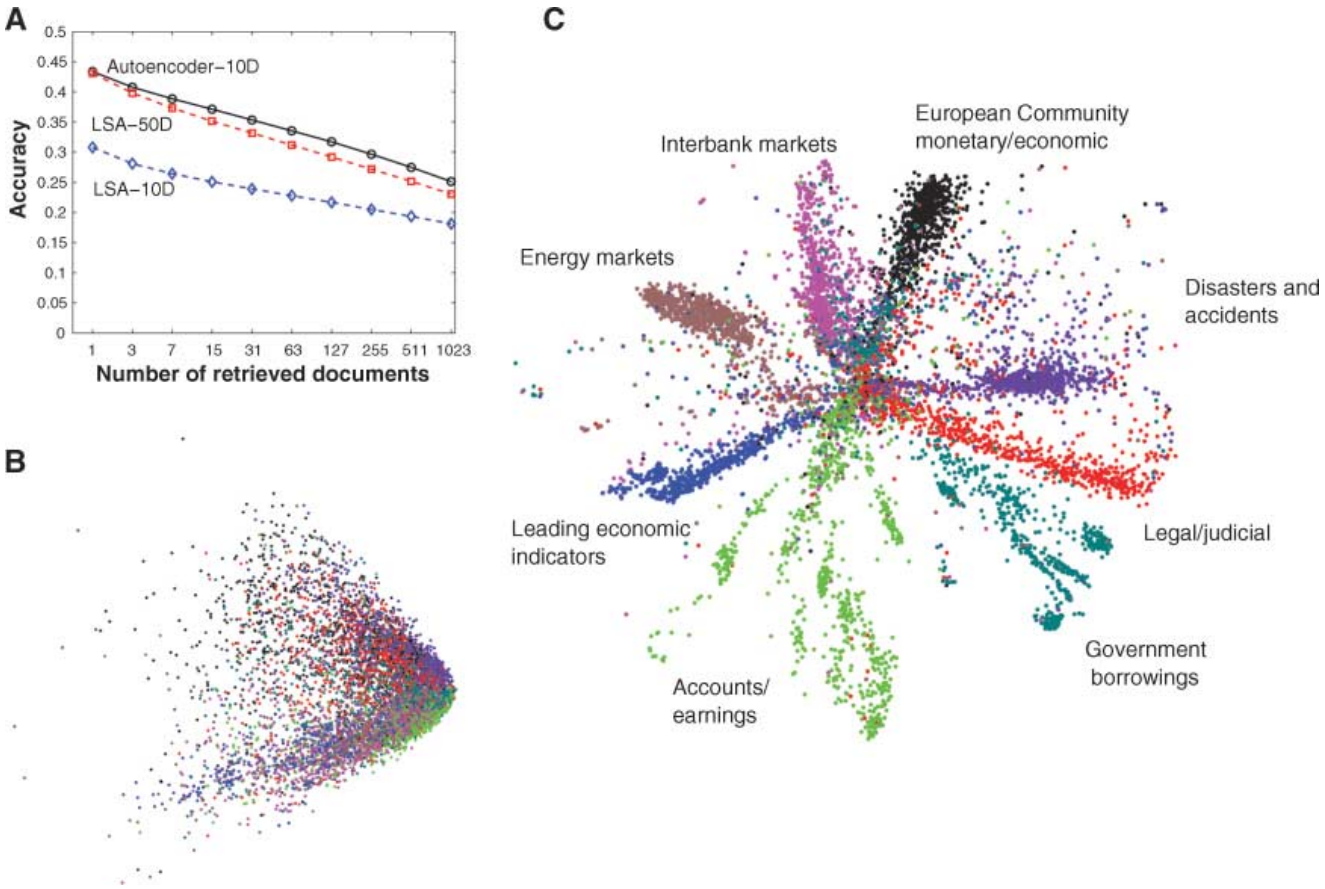
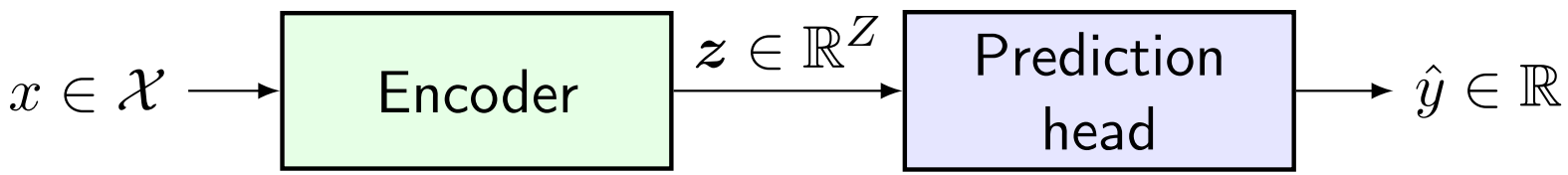


Fig. 4. (**A**) The fraction of retrieved documents in the same class as the query when a query document from the test set is used to retrieve other test set documents, averaged over all 402,207 possible queries. (**B**) The codes produced by two-dimensional LSA. (**C**) The codes produced by a 2000-500-250-125-2 autoencoder.

Hinton and Salakhutdinov, 2006

Encoders and prediction heads

- Functions that transform objects $x \in \mathcal{X}$ to embeddings $\boldsymbol{z} \in \mathbb{R}^Z$ are known as **encoders**
- Functions that transform embeddings $\boldsymbol{z} \in \mathbb{R}^Z$ to predictions $y \in \mathbb{R}$ are known as **prediction heads**
  - Can be linear or more complex
  - Typically much simpler than encoder
  - E.g., when prediction head is logistic regression, then positive and negative instances are ideally linearly separable in embedding space
  - Sometimes referred to as **unembed** operation
- Visually:



- Both encoder and prediction head are learned neural (sub)networks

Contrastive learning

- Embeddings can be used in other ways as well
- E.g., to compare objects, potentially across multiple modalities
  - Useful, for example, for zero- and few-shot prediction
  - Learned via a "contrastive learning" approach (more later)
- Example: CLIP embeddings for images and text

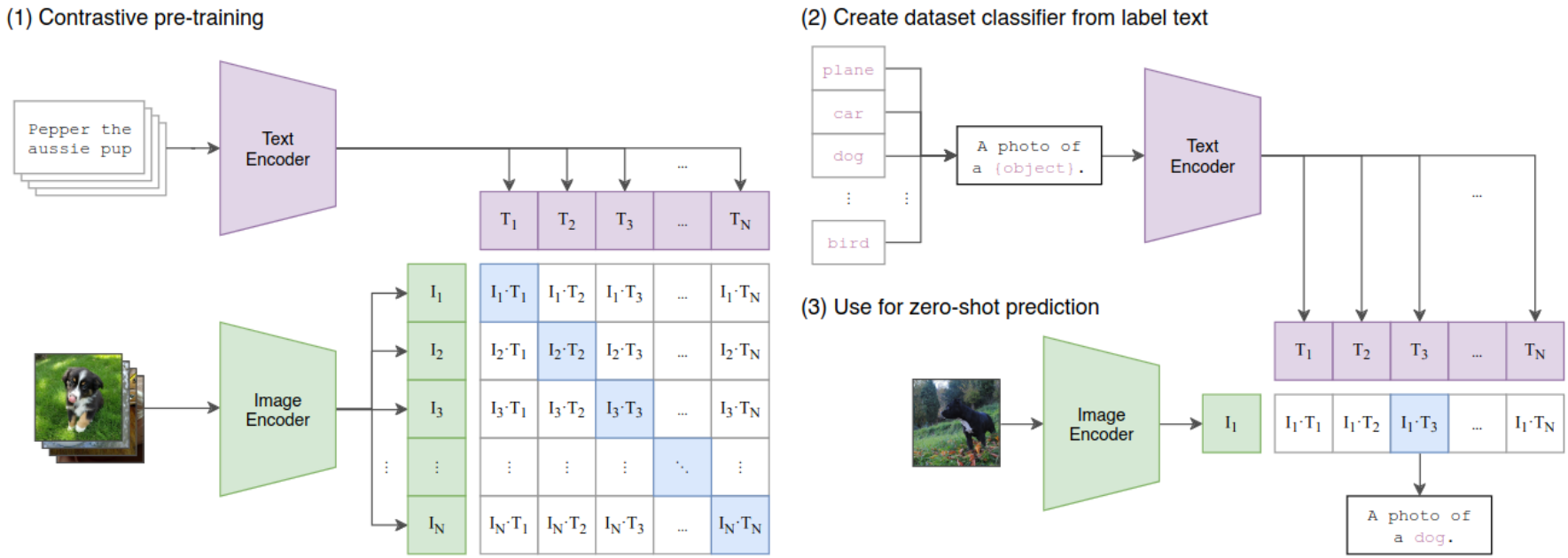
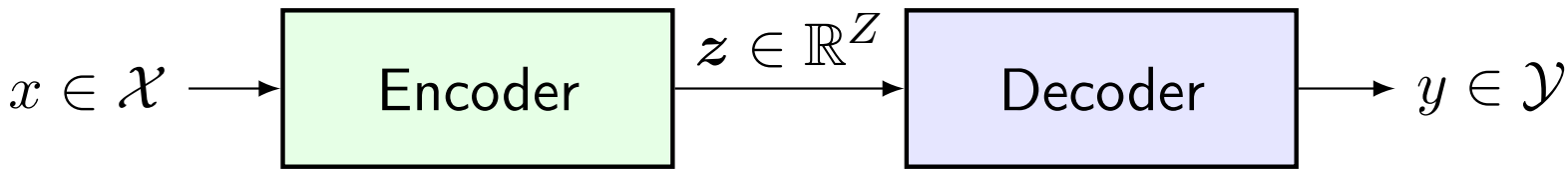


Figure 1. Summary of our approach. While standard image models jointly train an image feature extractor and a linear classifier to predict some label, CLIP jointly trains an image encoder and a text encoder to predict the correct pairings of a batch of (image, text) training examples. At test time the learned text encoder synthesizes a zero-shot linear classifier by embedding the names or descriptions of the target dataset's classes.

Structured prediction / deep generative models

- To handle more complex output spaces $\mathcal{Y}$, we may replace the prediction head by a component that "generates" output
- Functions that transform embeddings $\boldsymbol{z} \in \mathbb{R}^Z$ to (complex) outputs $y \in \mathcal{Y}$ are known as **decoders**
  - Note: in such models, embedding dimensionality Z may or may not depend on input x
- Visually:



Example: unCLIP (DALL-E 2)

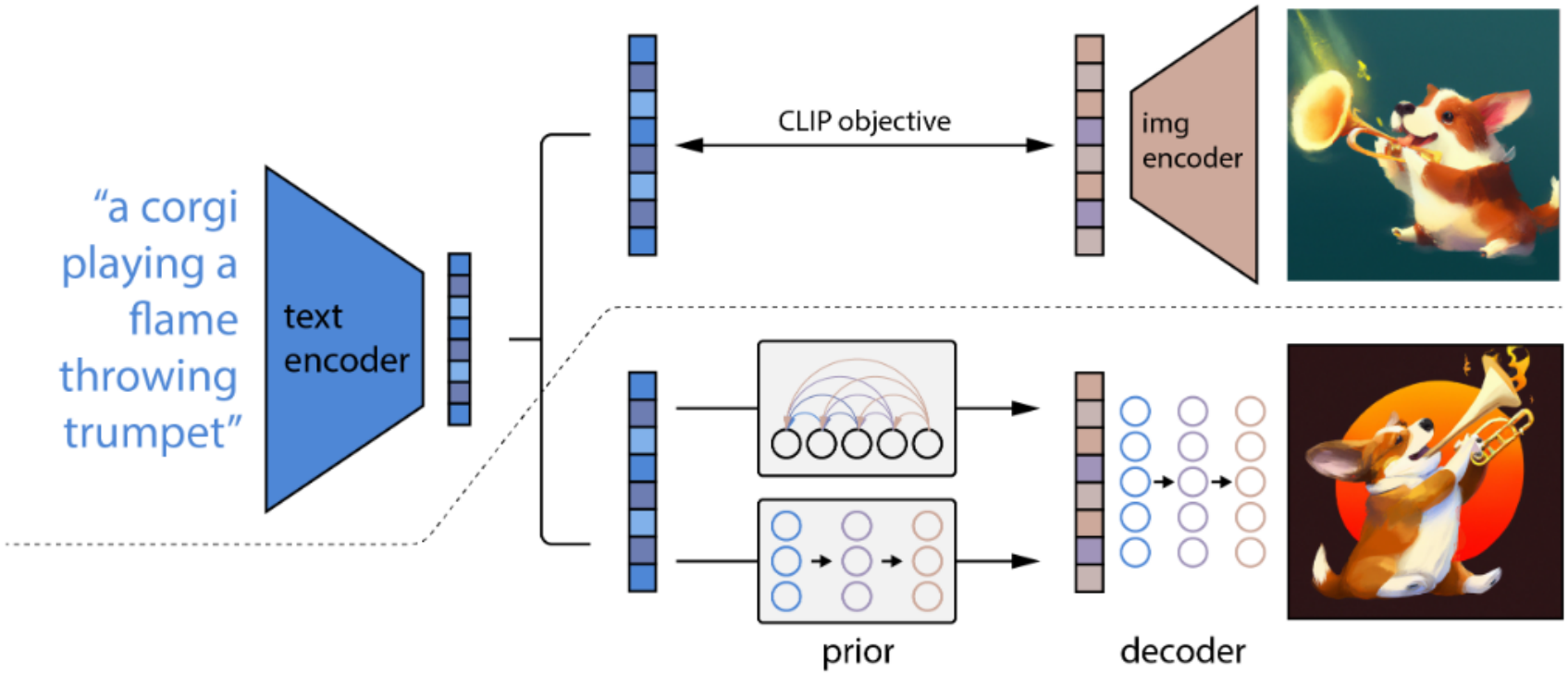


Figure 2: A high-level overview of unCLIP. Above the dotted line, we depict the CLIP training process, through which we learn a joint representation space for text and images. Below the dotted line, we depict our text-to-image generation process: a CLIP text embedding is first fed to an autoregressive or diffusion prior to produce an image embedding, and then this embedding is used to condition a diffusion decoder which produces a final image. Note that the CLIP model is frozen during training of the prior and decoder.

IE 678 Deep Learning

02 – Feedforward Neural Networks

Part 2: Feedforward Neural Networks

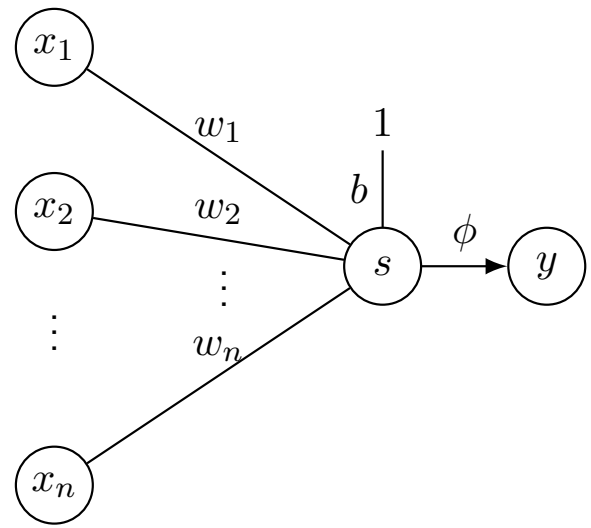
Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-2

Artificial neuron

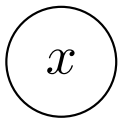
- An **artifical neuron** (AN) is a function $f : \mathbb{R}^n \to \mathbb{R}$
  - Inputs are vectors $\boldsymbol{x} \in \mathbb{R}^n$
  - Output is a value $y \in \mathbb{R}$
- f is taken from a family of functions that is parameterized by
  - A **weight vector** $\boldsymbol{w} \in \mathbb{R}^n$ (one weight per input)
  - A **bias** $b \in \mathbb{R}$
  - A **transfer function** or **activation function** $\phi : \mathbb{R} \to \mathbb{R}$
- Basic structure: $y = \phi(\boldsymbol{w}^\top \boldsymbol{x} + b)$
  - Computes the weighted sum $s = \boldsymbol{w}^\top \boldsymbol{x} + b$ of its inputs and bias (called **preactivation**)
  - Passes s through the transfer function to obtain output y (called **activation**)
  - ϕ can be deterministic or stochastic
- As before: bias can be replaced by an additional input $x_0 = 1$ and corresponding weight $w_0 = b$



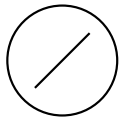
Types of artificial neurons

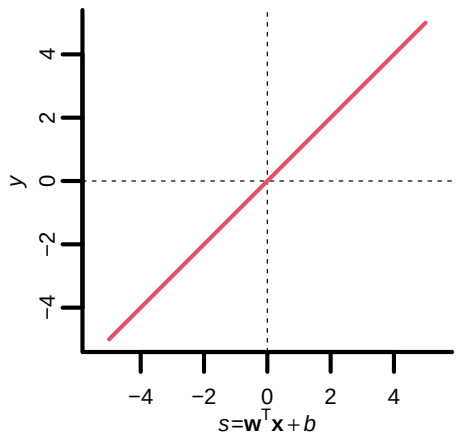
- The **type** of an AN is determined by its transfer function ϕ
- An AN of a given type can represent the family of functions

$$F_\phi = \left\{ \boldsymbol{x} \to \phi(\boldsymbol{w}^\top \boldsymbol{x} + b) \mid \boldsymbol{w} \in \mathbb{R}^n, b \in \mathbb{R} \right\}$$

- Each function in this family can be represented by its bias and weight vector
- We will see later that types are usually specified up-front, whereas weights are learned
- The simplest type of neuron is a **constant neuron**
  - No inputs; output fixed value $x \in \mathbb{R}$
  - Notation (from now on):  or simply x

Example: Linear neuron / identity

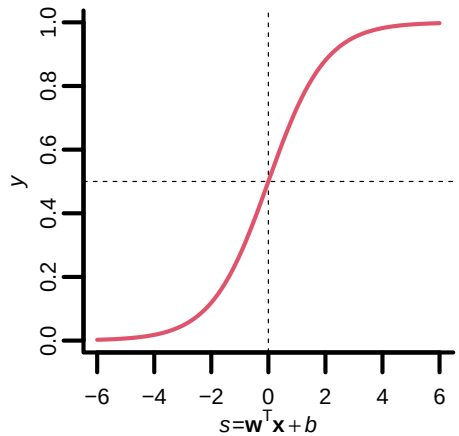
- Uses $\phi(s) = s$
- Notation: 



- Simple but computationally limited
- We often but not always want non-linear transfer functions

Example: Logistic neuron

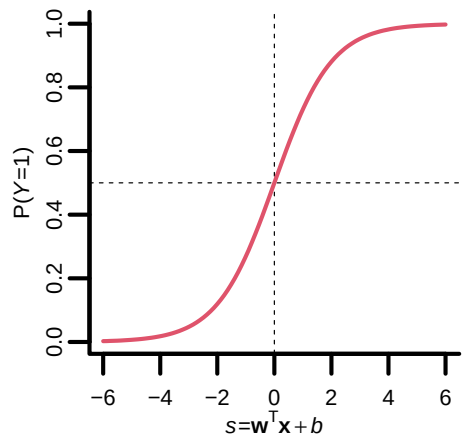
- Use logistic function $\phi(s) = \sigma(s) \stackrel{\text{def}}{=} \frac{1}{1+\exp(-s)}$
- Notation: 



- Gives a real-valued output that is smooth and bounded in $[0, 1]$
  - Negative preactivations mapped to value < 0.5
  - 0 preactivation mapped to 0.5
  - Positive preactivations mapped to value > 0.5
- Non-linear

Example: Stochastic binary neuron

- Also use logistic function
- But output of the logistic function is treated as a probability of producing a spike (1)
- I.e,. $\phi(s) = \begin{cases} 1 & \text{with probability } \sigma(s) \\ 0 & \text{otherwise} \end{cases}$



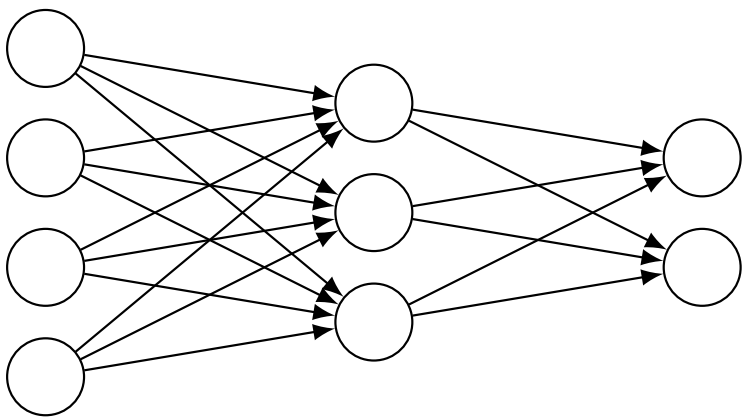
- Defines a probability distribution over outputs
- Other neurons can also be made stochastic

What is an artificial neural network?

- A network of artificial neurons
  - A set of (artificial) neurons
  - Connections between neurons (directed or undirected)
- Many different **architectures**
  - How many neurons? Of which type?
  - Are there **output neurons**?
  - Are there **hidden neurons** (neither input nor output)?
  - Which neurons are connected?
  - Are connections directed or undirected?
  - Are there cycles?
- Picking the right architecture for the problem at hand is important and requires skill/thought/compute power
  $\rightarrow$ **Architecture engineering**
- Can represent a wide range of functions (*universal approximation theorem*)

Feedforward neural networks

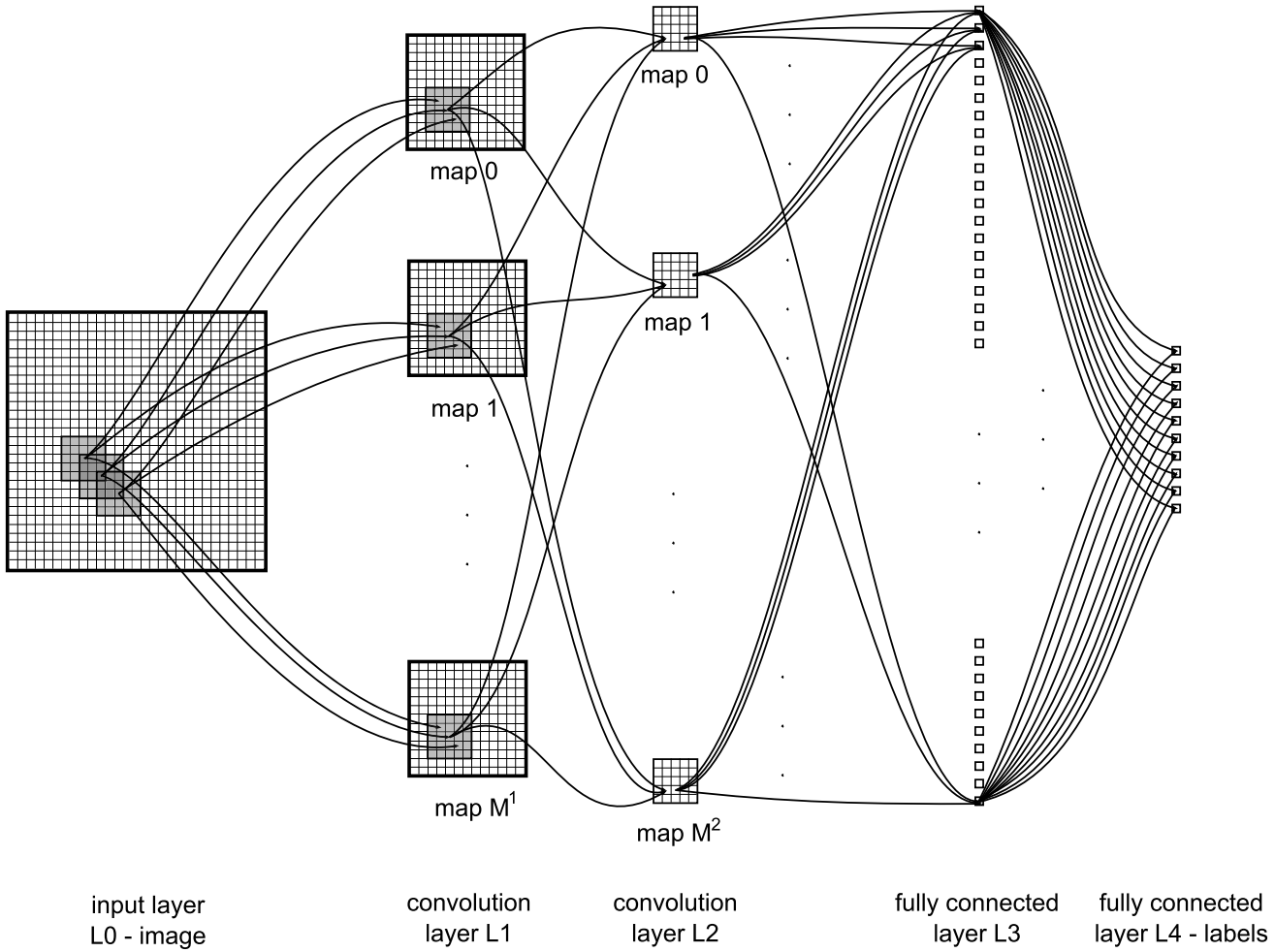
- A **feedforward neural network** (FNN) is an ANN in which
  - All connections are directed, and
  - There are no cycles (i.e., forms a DAG)
- Neurons usually grouped in **layers**
  - Input neurons: no incoming edges (first layer)
  - Output neurons: no outgoing edges (last layer)
  - Hidden neurons: all others (layer = maximum distance from input)
  - Layers do not need to be fully connected
  - Traditionally: edges only between subsequent layers
    (but: edges that skip layers are allowed, too)
- Example: an FNN with one hidden layer (omitting bias inputs)



Input layer Hidden layer Output layer

MNIST, best performer (2011), architecture

Deep convolutional neural network (no preprocessing)



Learning

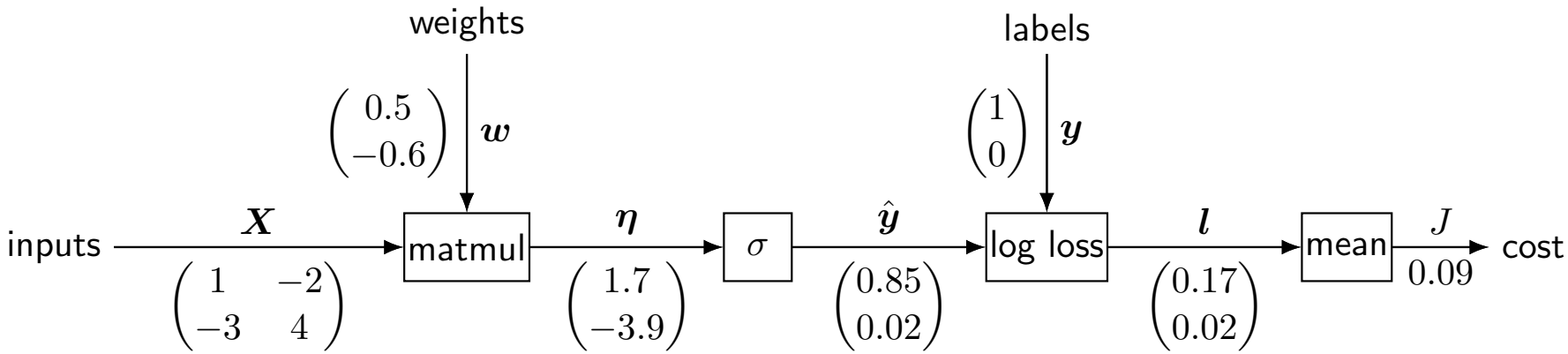
- Once we settled on an architecture, we need to learn connection strengths (weights)
- Simple approach
  - Supervised learning with labeled training examples
  - Use a suitable notion of model performance (e.g., loss function, likelihood)
  - Learn all weights jointly
    $\rightarrow$ As before: ERM/RRM, MLE/MAP, Bayesian inference (ML, 02-3)
  - Most common: empirical/regularized risk minimizaton
- We will see: simple approach is often "not enough"
  - High model complexity
  - Limited (labeled) training data
- Values of hidden units can be thought of as features, but which features are good is unknown and needs to be learned
  - This makes learning hard
  - Note: "embeddings" typically refer to the values of a "certain" hidden layer in a larger FNN (more later)

FNNs, more generally

- Choice of neuron type(s) important
  - Influences expressability
  - Influences "learnability"
  - Many more types of artificial neurons have been proposed
- **Neurons** may not follow the template discussed here
  - E.g., product neuron $y = \prod_i x_i$
  - E.g., max-pooling neuron $y = \max_i(x_i)$
  - In practice: (parameterized or fixed) **operators** instead of artificial neurons (more later)
- **Layers** may be more general
  - I.e., any parameterized function from $\mathbb{R}^n$ to $\mathbb{R}^m$
  - May compute multiple "dependent" outputs jointly (e.g., softmax layer)
  - May have additional internal structure (e.g., Transformer layers)
- Will see: generally, FNNs represented as a "compute graph"

Preview: logistic regression (2D), $N = 2$

Forward pass



IE 678 Deep Learning

02 – Feedforward Neural Networks

Part 3: Linear Layers

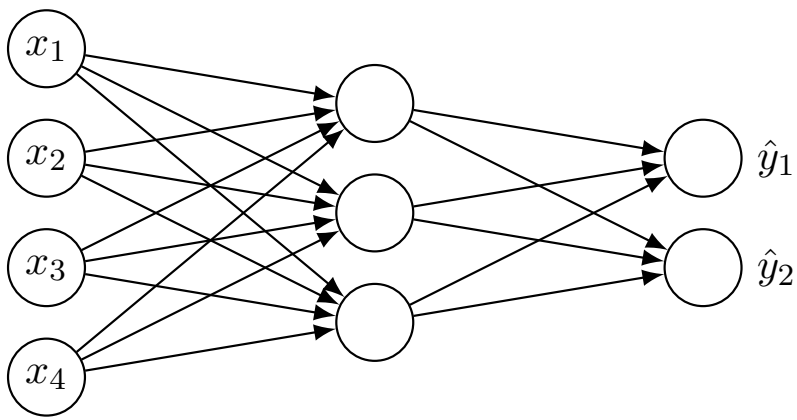
Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-2

Recall: Supervised learning with FNNs

- Supervised learning
  - Learn a mapping from inputs $\boldsymbol{x}$ to outputs $\boldsymbol{y}$
  - Training set $\mathcal{D} = \{ (\boldsymbol{x}_i, \boldsymbol{y}_i) \}_{i=1}^{N}$ of input-output pairs
  - With FNNs: for input $\boldsymbol{x}_i$, we want output $\hat{\boldsymbol{y}}_i$ "close" to $\boldsymbol{y}_i$
  - Learning means adjusting the weights such that the FNN does this
- FNNs are discriminative
  - Given an input $\boldsymbol{x}$, they compute an output $\hat{\boldsymbol{y}}$
  - But they don't allow going from outputs to inputs
- Hidden layer outputs are inputs of the next layer
  - We may also think of hidden layers as features for the next layer
  - These features are not provided upfront, but learned

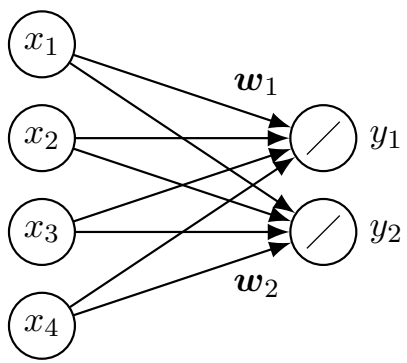


Linear layers

- Layers in which all layer inputs are connected with all layer outputs are called **dense layers** or **fully-connected layers**
- A dense **linear layer** is a layer consisting of only linear neurons
  - n layer inputs ($\boldsymbol{x} \in \mathbb{R}^n$), m layer outputs ($\boldsymbol{y} \in \mathbb{R}^m$)
  - Parameterized by weight vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m \in \mathbb{R}^n$
  - Optionally: biases $b_1, \ldots, b_m \in \mathbb{R}$
- Outputs given by

$$y_j = \sum_i [\boldsymbol{w}_j]_i x_i + b_j = \langle \boldsymbol{w}_j, \boldsymbol{x} \rangle + b_j$$

- Example: $n = 4$, $m = 2$, no bias



The action of a linear layer

- Without bias, we have: $y_j = \langle \boldsymbol{w}_j, \boldsymbol{x} \rangle$
- Let $\boldsymbol{W} \in \mathbb{R}^{n \times m}$ a **weight matrix** in which the j-th column equals the weights $\boldsymbol{w}_j$ of the j-th layer output

$$\boldsymbol{W} = \begin{pmatrix} \boldsymbol{w}_1 & \boldsymbol{w}_2 & \dots & \boldsymbol{w}_m \end{pmatrix}$$

- Then: $\boldsymbol{y} = \boldsymbol{W}^\top \boldsymbol{x}$
  - **Linear layers compute a matrix-vector product**
- For our example, $\boldsymbol{W} = \begin{pmatrix} \boldsymbol{w}_1 & \boldsymbol{w}_2 \end{pmatrix} \in \mathbb{R}^{4 \times 2}$ and

$$\boldsymbol{W}^\top \boldsymbol{x} = \begin{pmatrix} \boldsymbol{w}_1^\top \\ \boldsymbol{w}_2^\top \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} \langle \boldsymbol{w}_1, \boldsymbol{x} \rangle \\ \langle \boldsymbol{w}_2, \boldsymbol{x} \rangle \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \boldsymbol{y}$$

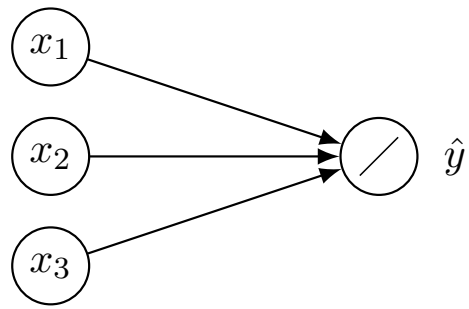
Using linear layers

- Typical uses of linear layers
  - As an output layer for regression tasks
  - As a hidden layer to perform dimensionality reduction ($m < n$) (in ML somewhat confusingly called **linear projection**)
  - Likewise, as a hidden layer to increase dimensionality ($m > n$)
- Number of parameters: nm (without bias), $nm + m$ (with bias)

n	m	# parameters	
64	64	4,096	
128	128	16,384	
256	256	65,536	
512	512	262,144	
1,024	1,024	1,048,576	
768	3,072	2,359,296	(T5-Base dense layer, dim up)
3,072	768	2,359,296	(T5-Base dense layer, dim down)

Linear regression as FNN (1)

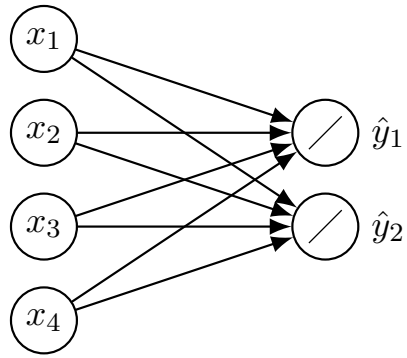
- In a **linear FNN**, all neurons/layers are linear
- Simplest linear FNN: single linear layer with one output



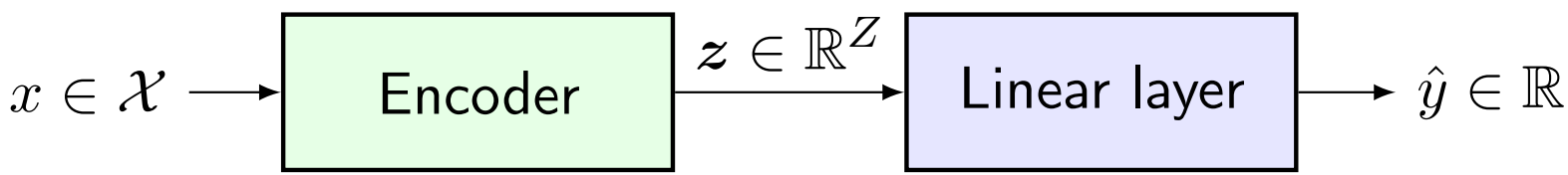
- Output $\hat{y} = \langle \boldsymbol{w}, \boldsymbol{x} \rangle + b$ is linear in input $\boldsymbol{x} \rightarrow$ linear model
- Suppose we train this network using ERM with squared loss
  - Empirical risk is $\frac{1}{N} \sum_i (y_i - \hat{y}_i)^2 \rightarrow$ minimize
  - We obtain ordinary least squares (OLS) estimate for linear regression
- Suppose we train with MLE assuming i.i.d. normal errors
  - I.e., assuming $y_i = \langle \boldsymbol{w}^*, \boldsymbol{x}_i \rangle + b^* + \epsilon_i$, where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$
  - Likelihood $\prod_i \mathcal{N}(y_i | \hat{y}_i, \sigma^2) \rightarrow$ maximize
  - Recall: solution is OLS estimate

Linear regression as FNN (2)

- With multiple outputs, we obtain multiple linear regression

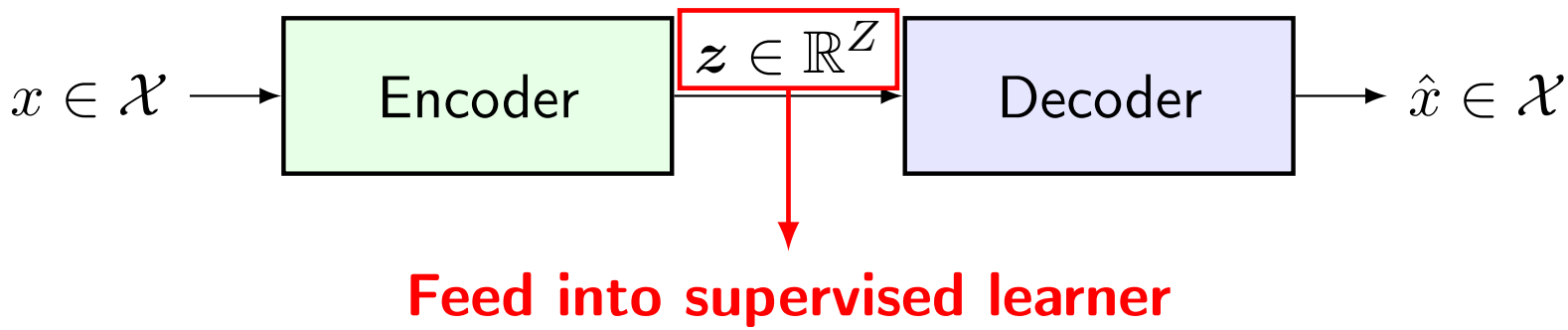


- **Linear FNN (w/o hidden layer) $\equiv$ linear regression**
  - To determine bias and weights, any suitable linear regression library can be used
- Outputs remain linear even with hidden layers ($\rightarrow$ exercise)
  $\rightarrow$ That's why we often want non-linearities
- For regression problems, linear layers often used as output layer



Autoencoders

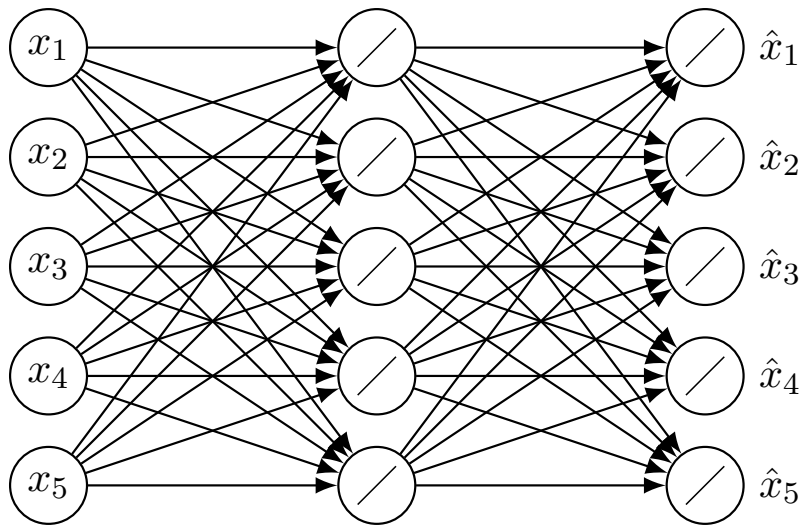
- FNNs are useful for unsupervised learning as well
  - We are given an unlabeled dataset $\mathcal{D} = \{ \boldsymbol{x}_i \}_{i=1}^N$ with $\boldsymbol{x}_i \in \mathbb{R}^D$
  - We don't have outputs
  - We want to find structure, or patterns, or reduce dimensionality
- Idea: train FNN to predict its input, i.e., set $\boldsymbol{y}_i = \boldsymbol{x}_i$
  - The resulting FNN is called an **autoencoder**



- Why autoencoders?
  - Autoencoders are a technique to **learn embeddings** ($\boldsymbol{z}$)
  - E.g., semi-supervised learning: train autoencoder on all inputs (labeled+unlabeled), use embeddings for supervised learner (labeled)
  - E.g., clustering: use embeddings as inputs to, say, K-means
  - E.g., denoising: use $\hat{x}$ instead of x
  - E.g., visualization: visualize $\boldsymbol{z}$ (e.g., using $Z = 2$)

Linear autoencoders

- Linear FNNs can do more than what may be expected at first glance
- A **linear autoencoder** uses only linear layers (in both encoder and decoder)
- A simple (but useless) linear autoencoder

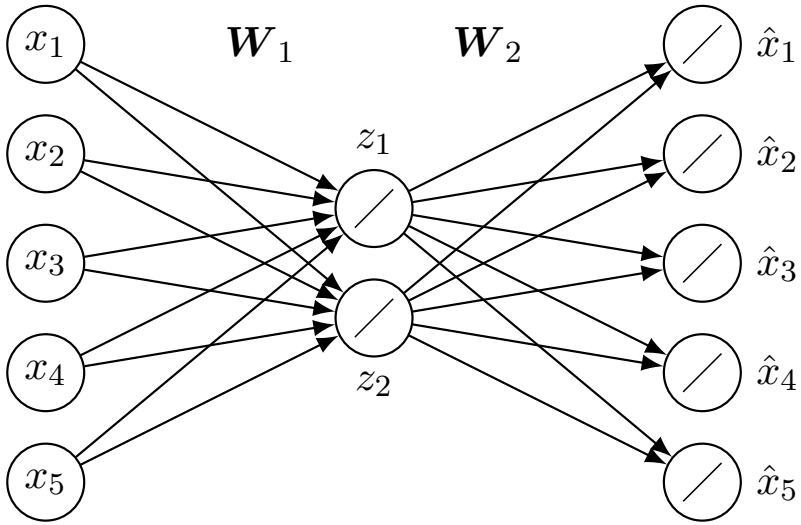


Input layer ($\boldsymbol{x}$) Hidden layer ($\boldsymbol{z}$) Output layer ($\hat{\boldsymbol{x}}$)

- Can you figure out the optimal weight matrices (such that $\hat{x}_j = x_j$)?

Bottlenecks

- Consider a linear autoencoder with $\boldsymbol{x} \in \mathbb{R}^D$ and one hidden layer with $Z < D$ hidden neurons



  - Can you still figure out the optimal weight matrices?
- A layer with few neurons is referred to as a **bottleneck**
  - I.e., fewer neurons than the surrounding layers
  - Forces FNN to "compress" information $\rightarrow$ dimensionality reduction
  - FNNs with bottlenecks *learn* how to compress
- Since autoencoder needs to **reconstruct** all inputs well, the optimal "compression" depends on all training inputs
  - E.g., above: 5D data ($\boldsymbol{x}$) compressed into a 2D representation ($\boldsymbol{z}$)

Obtaining optimal weights

- We have $\boldsymbol{z} = \boldsymbol{W}_1^\top \boldsymbol{x}$ and $\hat{\boldsymbol{x}} = \boldsymbol{W}_2^\top \boldsymbol{z} = \boldsymbol{W}_2^\top \boldsymbol{W}_1^\top \boldsymbol{x}$
- For squared error, solve $\operatorname{argmin}_{\boldsymbol{W}_1, \boldsymbol{W}_2} \left[\sum_i \sum_j (x_{ij} - \hat{x}_{ij})^2 \right]$
- The solution can be read off the **singular value decomposition** (SVD) of $\boldsymbol{X}$ (ML, 06-2)
  - Let $\boldsymbol{X}$ be the design matrix and $\boldsymbol{U}_Z \boldsymbol{\Sigma}_Z \boldsymbol{V}_Z^\top$ its size-Z truncated SVD
  - $\boldsymbol{U}_Z$ is an $N \times Z$ matrix with the first Z left-singular vectors of $\boldsymbol{X}$
  - $\boldsymbol{V}_Z$ is an $D \times Z$ matrix with the first Z right-singular vectors of $\boldsymbol{X}$
  - $\boldsymbol{\Sigma}_Z$ is an $Z \times Z$ matrix with the first Z singular values of $\boldsymbol{X}$
  - An optimal solution is $\boldsymbol{W}_1 = \boldsymbol{V}_Z$ and $\boldsymbol{W}_2 = \boldsymbol{V}_Z^\top$
  - For this solution, $\boldsymbol{z}_i^\top = \boldsymbol{x}_i^\top \boldsymbol{V}_z = [\boldsymbol{U}_Z]_{i:} \boldsymbol{\Sigma}_Z$
  - And $\hat{\boldsymbol{x}}_i^\top = \boldsymbol{z}_i^\top \boldsymbol{V}_z = [\boldsymbol{U}_Z]_{i:} \boldsymbol{\Sigma}_Z \boldsymbol{V}_Z^\top =$ the SVD reconstruction
- This is closely related to **principal component analysis** (PCA)
  - Main difference: first center the data so that each feature has mean 0
  - Then $\boldsymbol{W}_1$ contains the first Z principal components as its columns
  - And $\boldsymbol{z}_i$ contains the PCA scores for $\boldsymbol{x}_i$

Example: Weather data

$\boldsymbol{X}$	Jan	Apr	Jul	Oct	Year
Stockholm	-0.70	8.60	21.90	9.90	10.00
Minsk	-2.10	12.20	23.60	10.20	10.60
London	7.90	13.30	22.80	15.20	14.80
Budapest	1.20	16.30	26.50	16.10	15.00
Paris	6.90	14.70	24.40	15.80	15.50
Bucharests	1.50	18.00	28.80	18.00	16.50
Barcelona	12.40	17.60	27.50	21.50	20.00
Rome	11.90	17.70	30.30	21.40	20.40
Lisbon	14.80	19.80	27.90	22.50	21.50
Athens	12.90	20.30	32.60	23.10	22.30
Valencia	16.10	20.20	29.10	23.60	22.30
Malta	16.10	20.00	31.50	25.20	23.20

Example: Weights and representation

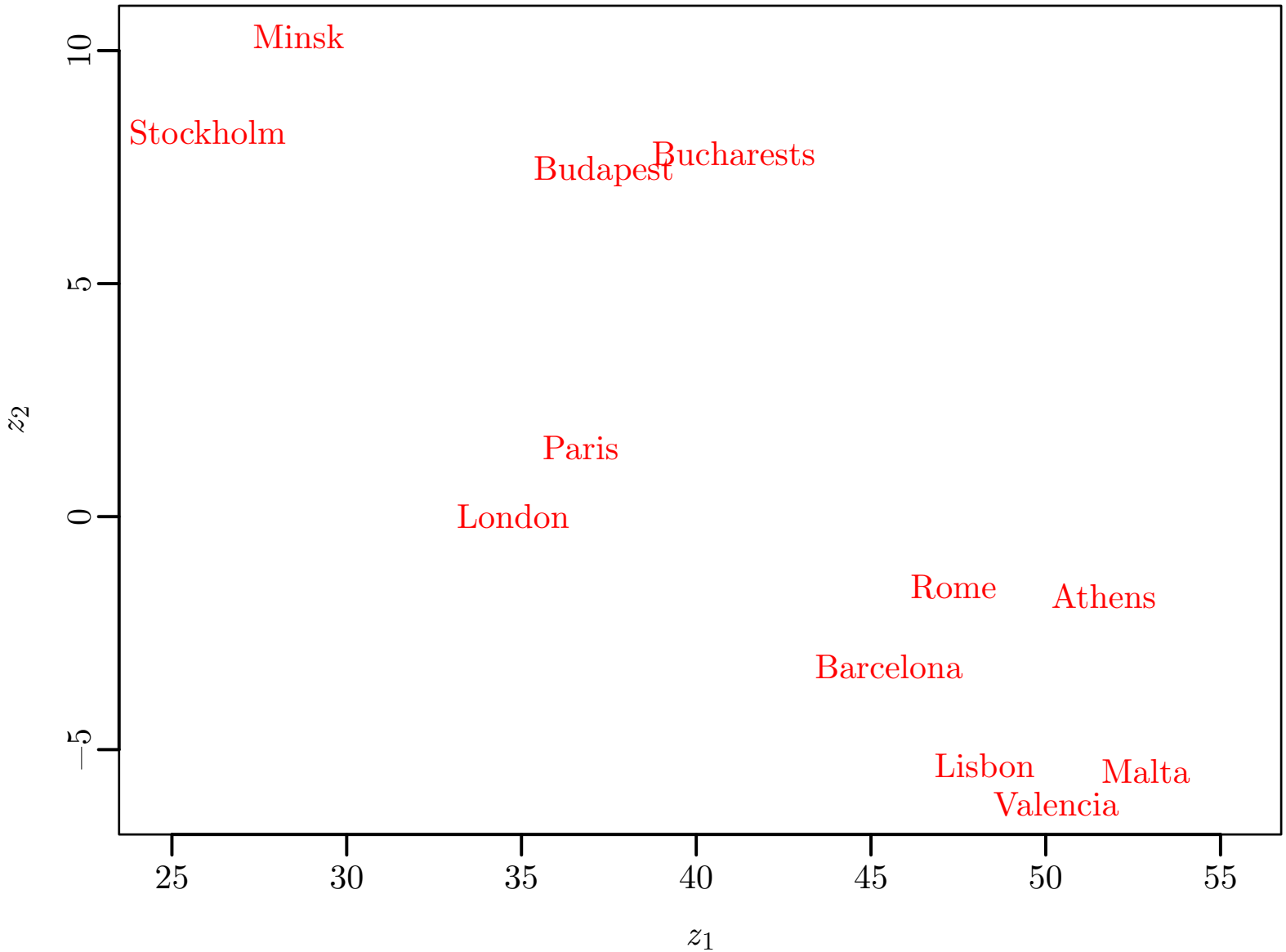
$\boldsymbol{W}_1$	1	2
Jan	0.22	-0.85
Apr	0.40	0.06
Jul	0.64	0.47
Oct	0.45	-0.18
Year	0.43	-0.14

$\boldsymbol{W}_2$	$\widehat{\text{Jan}}$	$\widehat{\text{Apr}}$	$\widehat{\text{Jul}}$	$\widehat{\text{Oct}}$	$\widehat{\text{Year}}$
1	0.22	0.40	0.64	0.45	0.43
2	-0.85	0.06	0.47	-0.18	-0.14

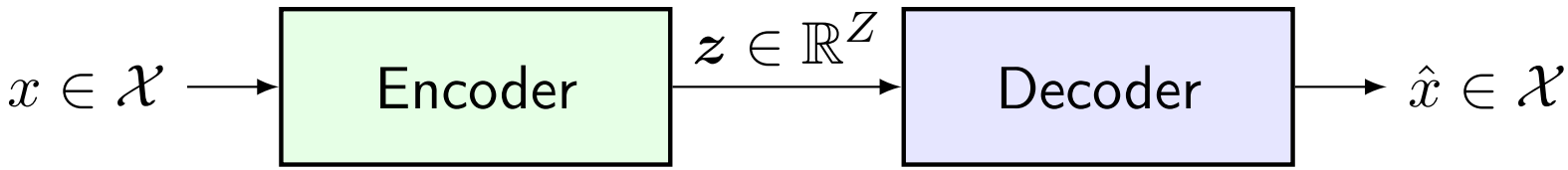
$\boldsymbol{Z}$	1	2
Stockholm	26.02	8.25
Minsk	28.63	10.30
London	34.76	0.00
Budapest	37.36	7.42
Paris	36.69	1.48
Bucharests	41.07	7.79
Barcelona	45.51	-3.22
Rome	47.36	-1.50
Lisbon	48.25	-5.34
Athens	51.66	-1.69
Valencia	50.30	-6.16
Malta	52.86	-5.45

Plot of representation

Bottlenecks of two neurons can be useful for visualization.



General autoencoders



- Encoder is a function (e.g., an FNN) that compresses input $\boldsymbol{x}$ to an embedding $\boldsymbol{z}$ (also called *code* or *distributed representation*)
- Decoder is a function (e.g., an FNN) that decompresses an embedding $\boldsymbol{z}$ to obtain reconstruction $\hat{\boldsymbol{x}}$
  - Think: approximate "inverse" of encoder
  - Decoder may be a "reversed" architecture of the encoder (e.g., layers in reverse order but with different weights)
  - Decoder may be an entirely different network
- Simplest way to train autoencoder is to use data points $\boldsymbol{x}$ as both input and reconstruction target

Example: Representing documents

804414 newswire stories, inputs = per-document rel. frequencies of 2000 most common word stems, autoencoder = logistic hidden units + linear output units

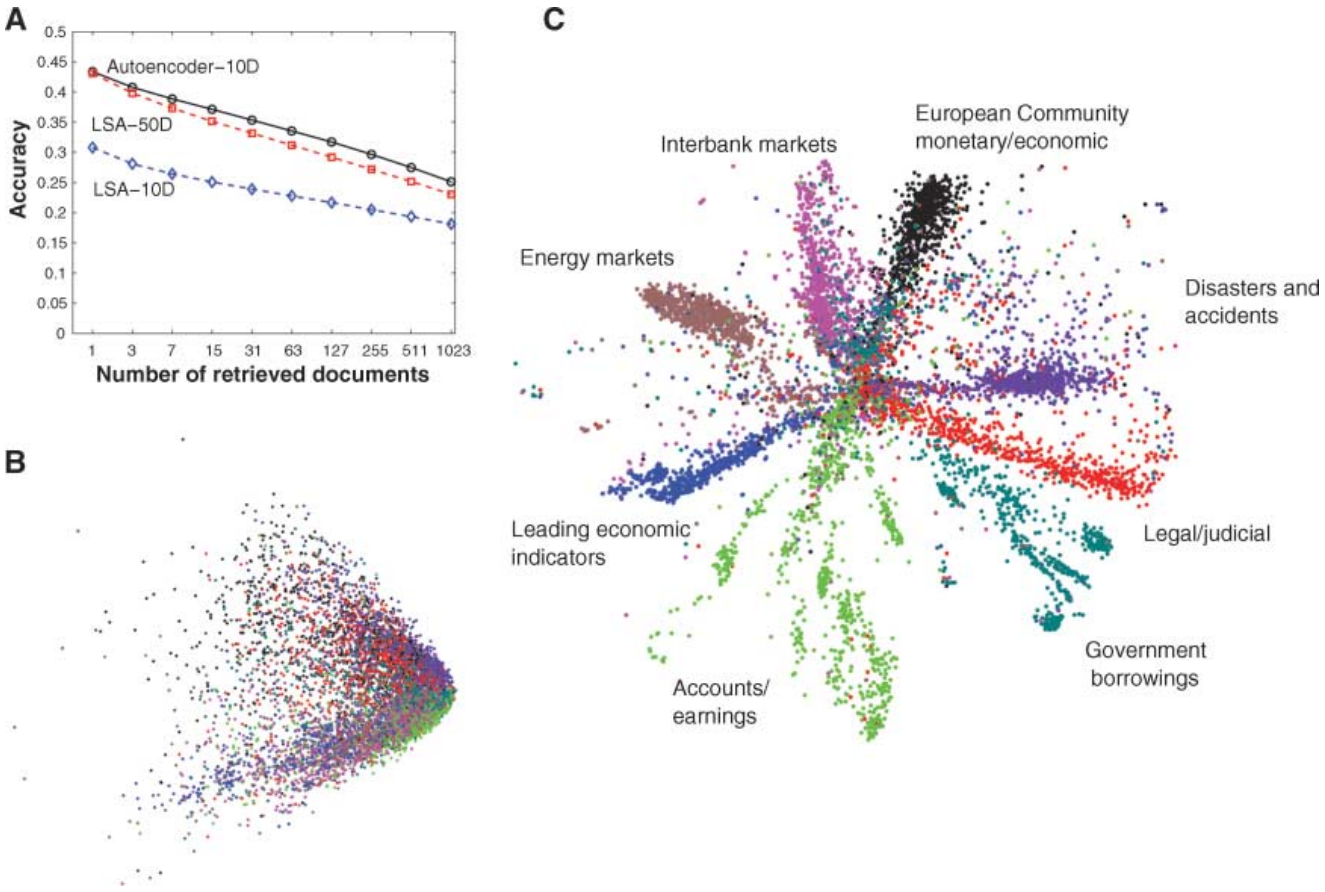


Fig. 4. (**A**) The fraction of retrieved documents in the same class as the query when a query document from the test set is used to retrieve other test set documents, averaged over all 402,207 possible queries. (**B**) The codes produced by two-dimensional LSA. (**C**) The codes produced by a 2000-500-250-125-2 autoencoder.

Hinton and Salakhutdinov, 2006

Discussion (autoencoders)

- Autoencoders are a form of **representation learning**
- Autoencoders are an example of **unsupervised pre-training**
  - I.e., learn (parts of) the weights of a network without supervision
- Many variants exists; e.g.,
  - Architecture of encoder/decoder
  - Choice of cost function
  - Construction of inputs and outputs for learning
  - Constraints on embeddings
- Examples
  - **Denoising autoencoders** perturb the input $\boldsymbol{x}$ with noise to obtain $\tilde{\boldsymbol{x}}$, and then aim to reconstruct the original input $\boldsymbol{x}$ from $\tilde{\boldsymbol{x}}$
    $\rightarrow$ noise robustness
  - **Variational autoencoders** force $\boldsymbol{z}$ to follow a specified simple distribution (e.g., diagonal Gaussian)
    $\rightarrow$ generative model
  - **Sparse autoencoders** force $\boldsymbol{z}$ to be sparse
    $\rightarrow$ sparse representations

IE 678 Deep Learning

02 – Feedforward Neural Networks

Part 4: Non-Linear Layers

Prof. Dr. Rainer Gemulla

Universität Mannheim

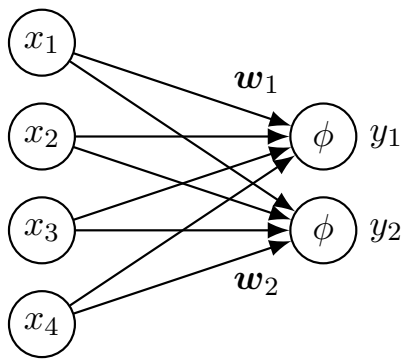
Version: 2025-1

Fully-connected layers (1)

- Recall: Layers in which all layer inputs are connected with all layer outputs are called **dense layers** or **fully-connected layers**
  - n layer inputs ($\boldsymbol{x} \in \mathbb{R}^n$), m layer outputs ($\boldsymbol{y} \in \mathbb{R}^m$)
  - Parameterized by weight vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m \in \mathbb{R}^n$
  - Optionally: biases $b_1, \ldots, b_m \in \mathbb{R}$
  - **Transfer function** $\phi : \mathbb{R} \to \mathbb{R}$
- Outputs given by

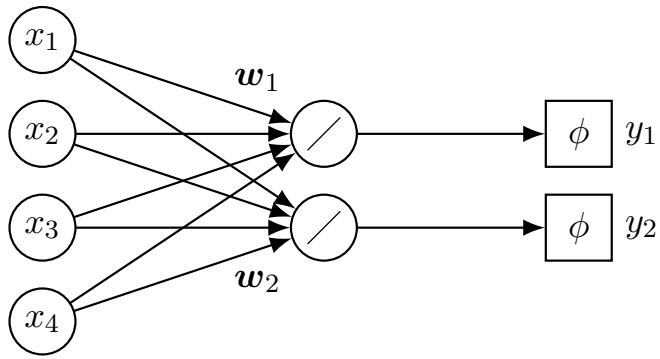
$$y_j = \phi(\langle \boldsymbol{w}_j, \boldsymbol{x} \rangle + b_j)$$

- Example: $n = 4$, $m = 2$, no bias



Fully-connected layers (2)

- We can also interpret a fully-connected layer as a (learned) linear layer followed by a (fixed) non-linearity:

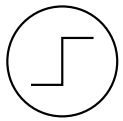
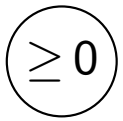
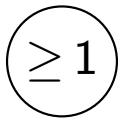


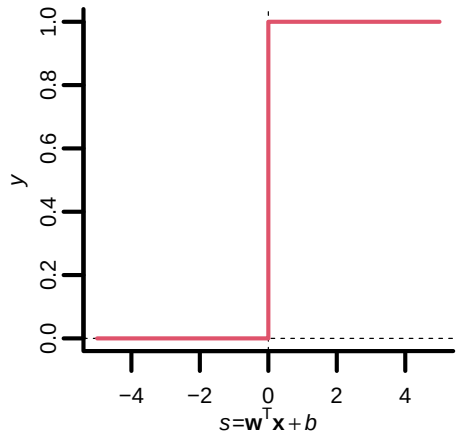
- The action of the layer (without bias) is

$$\boldsymbol{y} = \phi(\boldsymbol{W}^\top \boldsymbol{x}),$$

where we take the convention that ϕ is applied element-wise on vector inputs

Binary threshold neuron

- One of the (seemingly) simplest non-linear neurons is the **binary threshold neuron** (also called *McCulloch-Pitts neuron*)
- Uses the **binary threshold function** as transfer function: outputs fixed "spike" if input s is non-negative, else "nothing"
- I.e,. $\phi(s) = I(s \geq 0) = \begin{cases} 1 & \text{if } s \geq 0 \\ 0 & \text{otherwise} \end{cases}$
- Notation:  or with fixed bias  ,  , ...

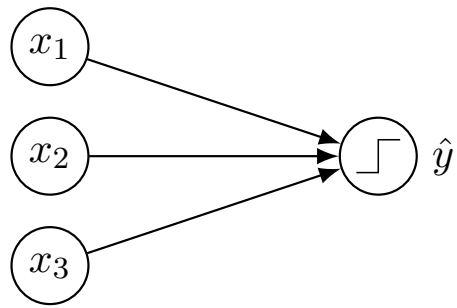


- One interpretation: each input is the truth value of some proposition, output is truth value of another proposition ($\rightarrow$ exercise)

McCulloch and Pitts, 1943

Perceptron

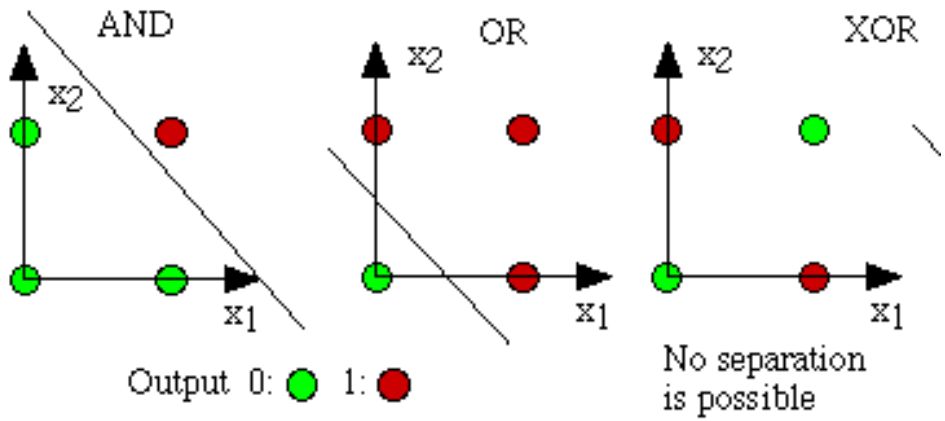
- Invented 1957 by Frank Rosenblatt at the Cornell Aeronautical Laboratory
- Corresponds to an FNN without hidden layers and binary threshold units for outputs (**single-layer perceptron**)



- Already discussed in ML course (ML 04-1, which see)
  - Linear decision boundary $= \{\, \boldsymbol{x} : \langle \boldsymbol{w}, \boldsymbol{x} \rangle + b = 0 \,\}$
- Many hopes and much controversy about what it can do at the time (see Olazaran (1996) for history)

Recap: What can perceptrons learn?

- Perceptrons can classify perfectly if there exists an affine hyperplane that separates the classes
  - I.e., when the data is **linearly separable**
- Otherwise, the perceptron must make errors on some inputs
- This is quite limited; e.g., perceptrons cannot learn the XOR function



- We will come back to this later

Image source

Complexity of perceptron learning

- Suppose we want to minimize the misclassification rate (0-1 loss)
- If the data is linearly separable $\rightarrow$ "easy"
  - In P; e.g., solve the linear program

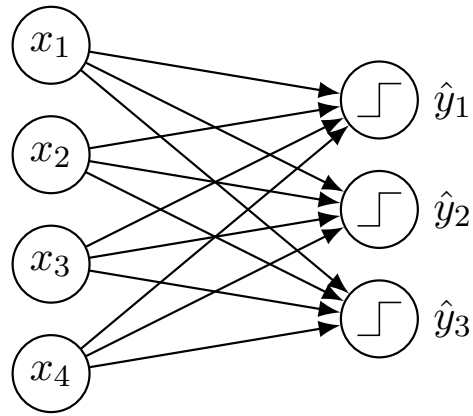
$$\begin{array}{rll} \text{minimize} & 0 & \\ \text{subject to} & \langle \boldsymbol{x}_i, \boldsymbol{w} \rangle \geq 0 & \text{for all } \boldsymbol{x}_i \text{ in pos. class } (y_i = 1) \\ & \langle \boldsymbol{x}_i, \boldsymbol{w} \rangle < 0 & \text{for all } \boldsymbol{x}_i \text{ in neg. class } (y_i = 0) \end{array}$$

- If the data is not linearly separable $\rightarrow$ "difficult"
  - Finding an optimal weight vector is NP-hard (when dimensionality n is part of the input)
  - Remains NP-hard even when weights restricted to $\{-1, 1\}$
  - NP-hard to approximate even when weights restricted to $\{-1, 1\}$
  - Fortunately, we are often able to nevertheless find sufficiently good weights in practice

Amaldi and Kann, 1995

Perceptrons with multiple output units

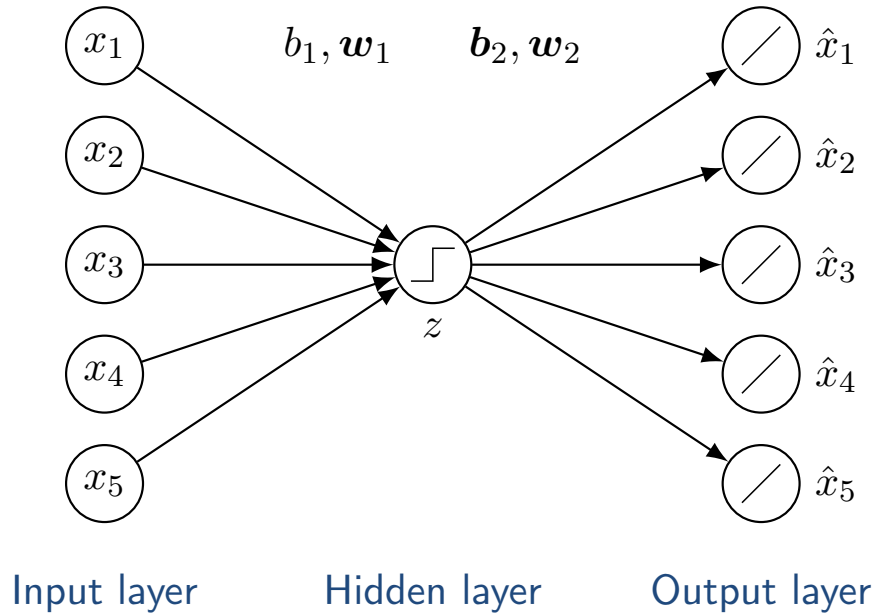
Consider a perceptron with m binary outputs for classification tasks.



1. **Multi-label classification** $\rightarrow$ works
   - Each input is associated with m binary class labels
   - Goal is to predict each of them
   - E.g.: height (small/tall), hair color (light/dark), ...
2. **Multi-class classification** (first option) $\rightarrow$ problematic
   - Each input is associated with one out of 2^m class labels
   - We associate each label with one output vector of the perceptron
   - Problem: Which label with which output vector? (choice matters)
3. **Multi-class classification** (second option) $\rightarrow$ problematic
   - Each input is associated with one out of m class labels
   - We associate each label with its indicator vector (**one-hot encoding**)
   - Problem: What if the network outputs less/more than a single 1?

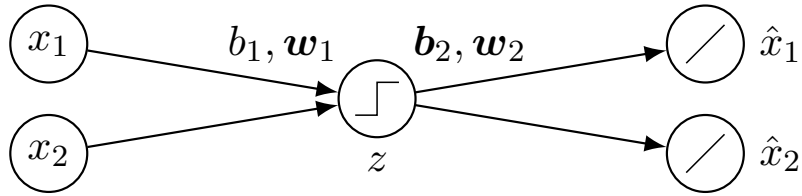
An autoencoder with a binary threshold unit

- Consider the following autoencoder (with biases)

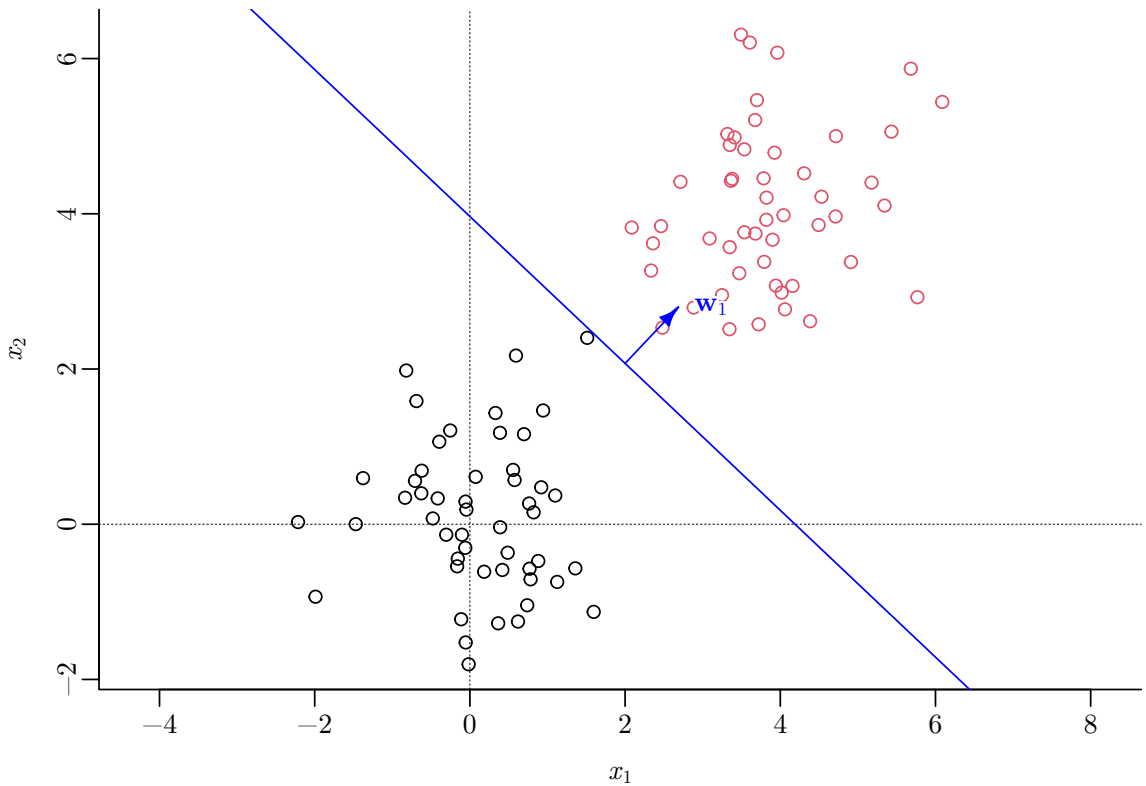


  - Observe: $z \in \{0, 1\}$ is a binary embedding (binary code)
- Assume that we want to minimize squared error over training data
  - What does this autoencoder then compute?

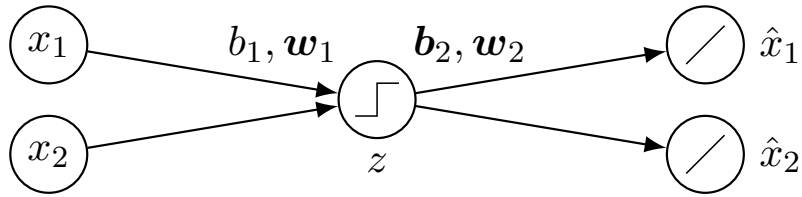
Interpreting the weights (1)



- Suppose that we are given b_1 and $\boldsymbol{w}_1$
- The binary threshold unit then acts as a linear "classifier"
  - Input $\boldsymbol{x}$ mapped to either $z = 0$ (bottom left) or $z = 1$ (top right)



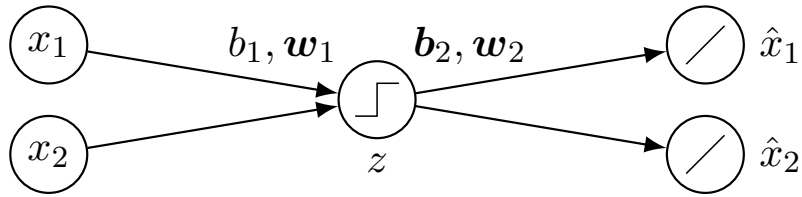
Interpreting the weights (2)



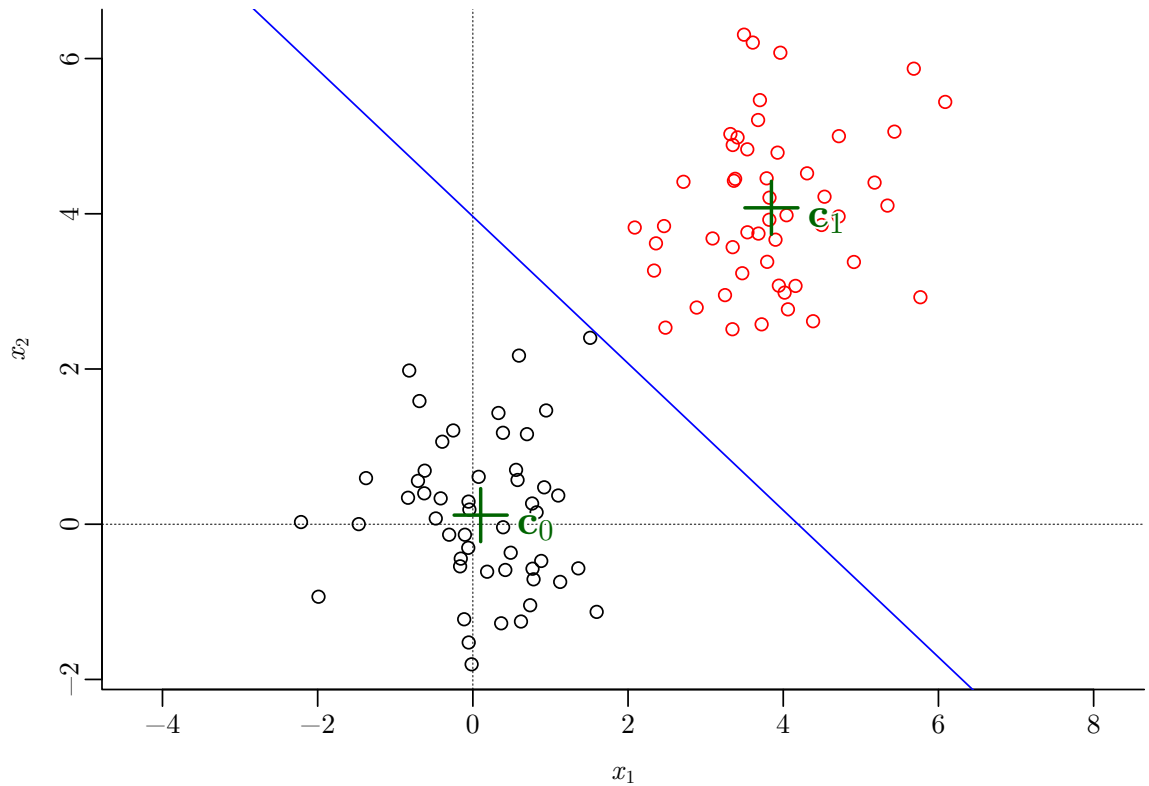
- Let's now look at $\boldsymbol{b}_2$ and $\boldsymbol{w}_2$
  - Given z, output is $\hat{\boldsymbol{x}} = \boldsymbol{w}_2 z + \boldsymbol{b}_2$
  - All points in "class" $z = 0$ are mapped to $\boldsymbol{c}_0 \stackrel{\text{def}}{=} \boldsymbol{b}_2$
  - All points in "class" $z = 1$ are mapped to $\boldsymbol{c}_1 \stackrel{\text{def}}{=} \boldsymbol{b}_2 + \boldsymbol{w}_2$
- Given b and $\boldsymbol{w}_1$, what are the optimal choices of $\boldsymbol{c}_0$ and $\boldsymbol{c}_1$?
  - Denote by z_i the class of input $\boldsymbol{x}_i$
  - Squared error is $\sum_i \sum_j (x_{ij} - \hat{x}_{ij})^2 = \sum_i \left\| \boldsymbol{x}_i - \boldsymbol{c}_{z_i} \right\|^2$
  - Alternatively: $\sum_{i:z_i=0} \left\| \boldsymbol{x}_i - \boldsymbol{c}_0 \right\|^2 + \sum_{i:z_i=1} \left\| \boldsymbol{x}_i - \boldsymbol{c}_1 \right\|^2$
  - For each class k, our goal is to minimize the squared Euclidean distance between the $\boldsymbol{x}_i$'s of the class and its representative $\boldsymbol{c}_k$
  - Optimum solution is the mean of the examples of the class

$$\boldsymbol{c}_k = \frac{1}{\sum_{i:z_i=k} 1} \sum_{i:z_i=k} \boldsymbol{x}_i$$

Interpreting the weights (3)

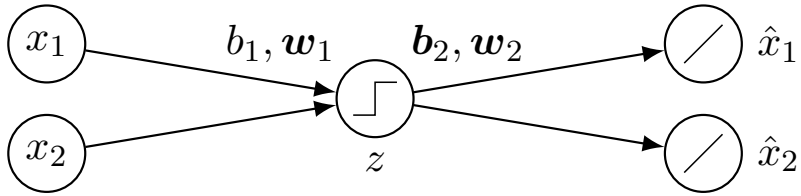


- The overall optimum solution is



- Can you see what the autoencoder does?

Interpreting the weights (4)



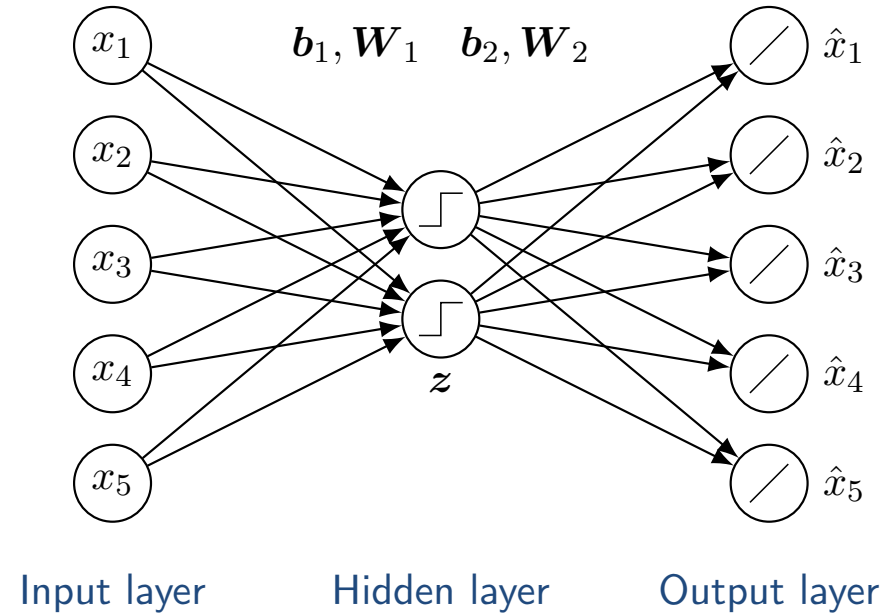
- Optimum solution agrees with K-means for $K = 2$
- K-means objective is to minimize the sum of squared distances

$$\underset{C}{\operatorname{argmin}} \sum_{k=1}^{K} \sum_{\boldsymbol{x} \in C_k} \|\boldsymbol{x} - \boldsymbol{\mu}_k\|^2 ,$$

  where $\boldsymbol{\mu}_k$ is the mean of the points in cluster C_k
- Given an optimal K-means clustering for $K = 2$
  - Each data point is associated to cluster of closest representative
  - We set $\boldsymbol{c}_k = \boldsymbol{\mu}_k$ (and thus obtain $\boldsymbol{b}_2$ and $\boldsymbol{w}_2$)
  - We set b_1 and $\boldsymbol{w}_1$ such that the decision boundary is the set of points with equal distance to $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ (see previous slide)
  - The binary threshold unit then associates each point $\boldsymbol{x}_i$ with its correct cluster z_i

An autoencoder with multiple binary threshold units

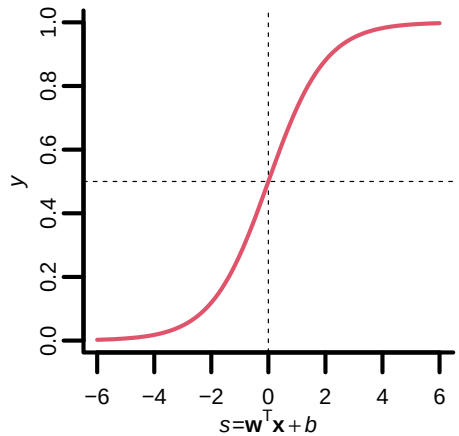
- What happens if we have multiple binary threshold units?



- This autoencoder also "clusters" the data
  - Associates each data point with a "binary code" (00, 01, 10, 11)
  - Each codeword can be seen as a cluster (2^Z in total)
- For $Z > 1$ binary threshold units, the optimum solution does does not correspond to K-means anymore (with $K = 2^Z$)
  - Why? $\rightarrow$ exercise

Recall: Logistic neuron

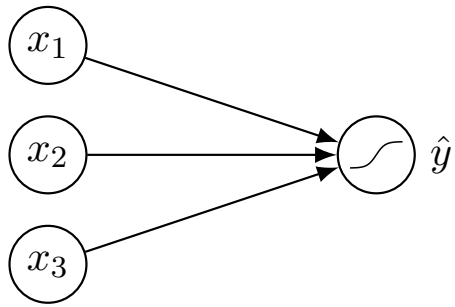
- Use logistic function $\phi(s) = \sigma(s) \stackrel{\text{def}}{=} \frac{1}{1+\exp(-s)}$
- Notation: 



- Gives a real-valued output that is smooth and bounded in $[0, 1]$
  - Negative activations mapped to value < 0.5
  - 0 activation mapped to 0.5
  - Positive activation mapped to value > 0.5
- Non-linear

An FNN with a single logistic unit

- If the binary threshold unit of a perceptron is replaced by a logistic unit, we obtain an FNN similar to a perceptron



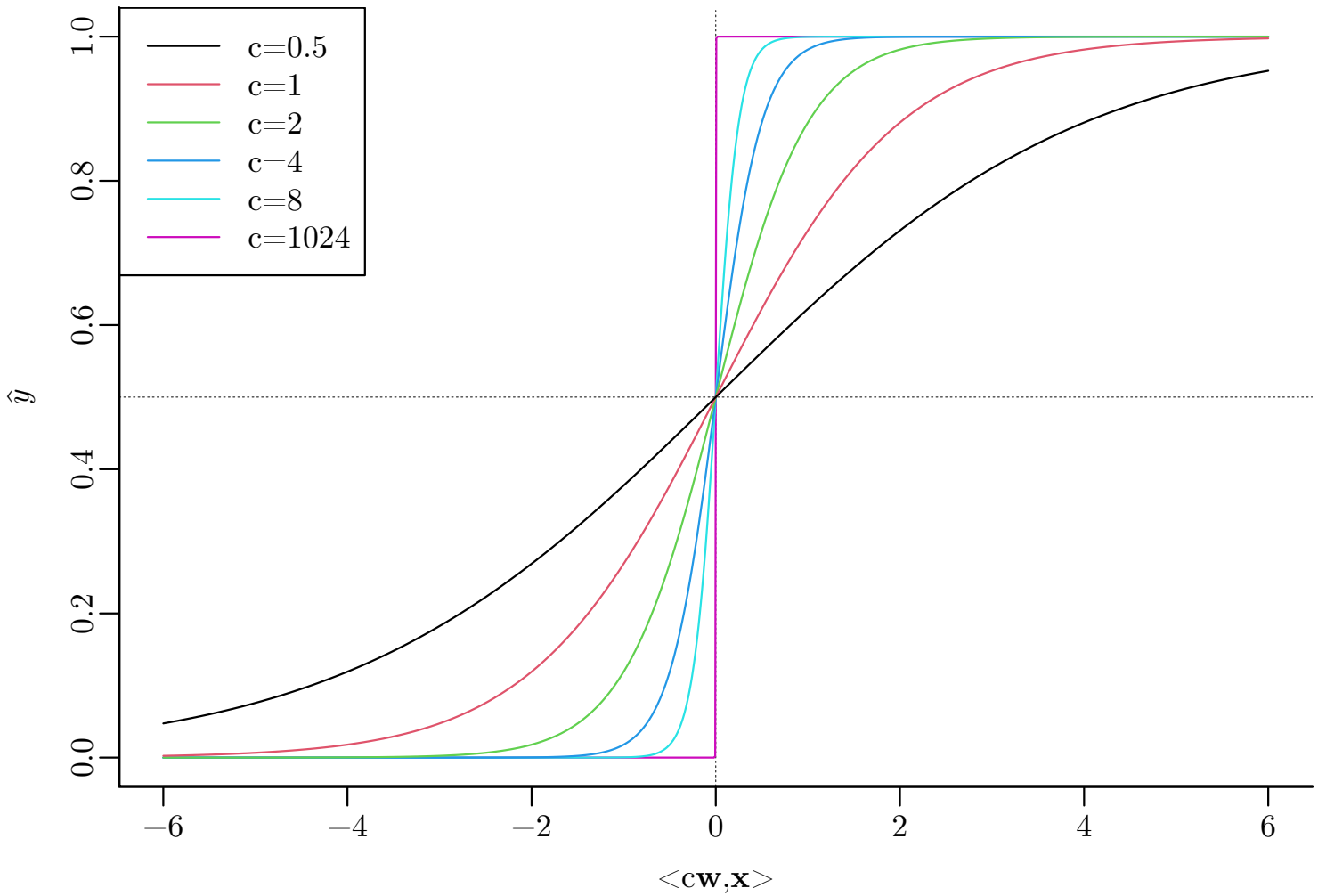
- What's the difference?
  - Fix some weight vector $\boldsymbol{w}$ (and ignore bias)
  - Above neural network outputs $\hat{y} \in [0, 1]$ with

$$\hat{y} = \sigma(\langle \boldsymbol{w}, \boldsymbol{x} \rangle) \begin{cases} < 0.5 & \langle \boldsymbol{w}, \boldsymbol{x} \rangle < 0 \\ \geq 0.5 & \langle \boldsymbol{w}, \boldsymbol{x} \rangle \geq 0 \end{cases}$$

  - If output of the logistic unit is rounded to the closest integer, one obtains output of the corresponding perceptron
  - Logistic unit can be seen as a "smooth" version of a binary threshold unit

Smoothing

If we scale the weights by some constant $c > 0$, we change the degree of smoothing.



Binary classification

- Suppose we use the network for a binary classification task
  - Given a labeled set $\mathcal{D} = \{ (\boldsymbol{x}_i, y_i) \}_{i=1}^{N}$ of input-output pairs
- We can minimize the misclassification error (**0-1 loss**)

$$\sum |y_i - \text{round}(\hat{y}_i)|$$

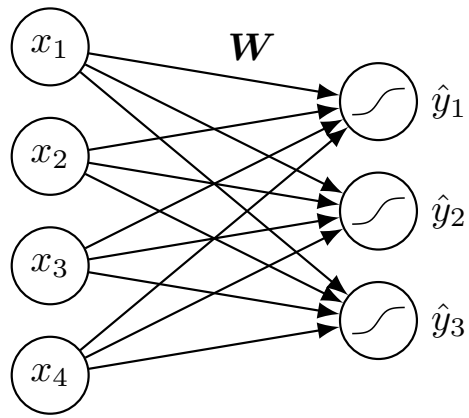
  - Equivalent to perceptron
  - Outputs related to distance from decision boundary, but no probabilistic interpretation possible
- We can maximize the log-likelihood of the provided labels

$$\ln \mathcal{L} = \sum [\, y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_i) \,]$$

  - Equivalent to logistic regression
  - Input $\langle \boldsymbol{w}, \boldsymbol{x} \rangle$ to logistic transfer function interpreted as estimate of log odds of positive class
  - Output $\hat{y}_i$ interpreted as confidence for positive class
- **Output layer of FNNs for binary classification tasks is typically a logistic neuron**

Multi-class classification (bad approach)

- Naive (bad) approach to multi-class classification
  - For C classes, use C logistic neurons
  - Associate each label with its indicator vector (one-hot encoding)
  - We may interpret output $\hat{y}_c$ as confidence in label c and predict the label with the largest confidence



- Problem: Interpretation of $\hat{y}_c$ as confidence not valid
  - Outputs $\hat{y}_c$ may not sum to one $\rightarrow$ $\hat{\boldsymbol{y}}$ is not a probability vector
- Solution: tie the output neurons appropriately
  $\rightarrow$ **softmax layer** (cf. ML 04-2)

Recap: The softmax function

- The **softmax function** $S(\boldsymbol{\eta})$
  - Takes a real vector $\boldsymbol{\eta} = (\eta_1, \ldots, \eta_C)^\top \in \mathbb{R}^C$
  - And transforms it into an C-dimensional probability vector $S(\boldsymbol{\eta})$

$$S(\boldsymbol{\eta})_c = \frac{\exp(\eta_c)}{\sum_{c'=1}^{C} \exp(\eta_{c'})}$$

  - Called this way because it exaggerates differences and acts somewhat like the max function (approximates indicator function of largest coefficient)

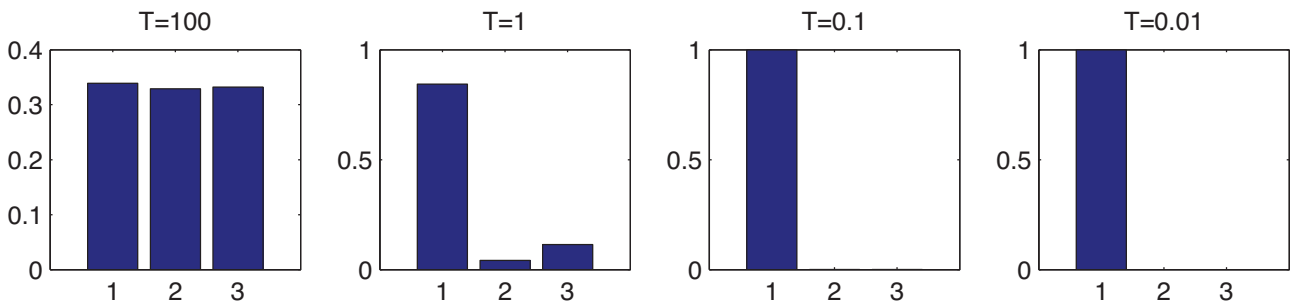
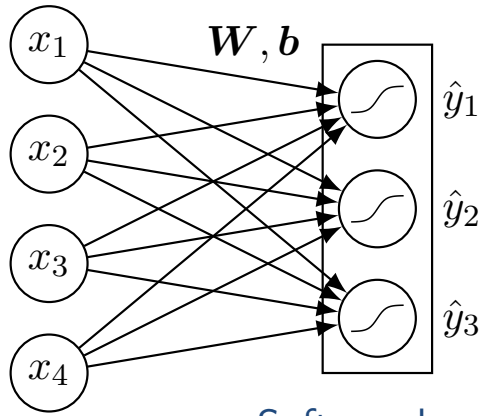


Figure 4.4 Softmax distribution $\mathcal{S}(\boldsymbol{\eta}/T)$, where $\boldsymbol{\eta} = (3, 0, 1)$, at different temperatures T. When the temperature is high (left), the distribution is uniform, whereas when the temperature is low (right), the distribution is "spiky", with all its mass on the largest element. Figure generated by `softmaxDemo2`.

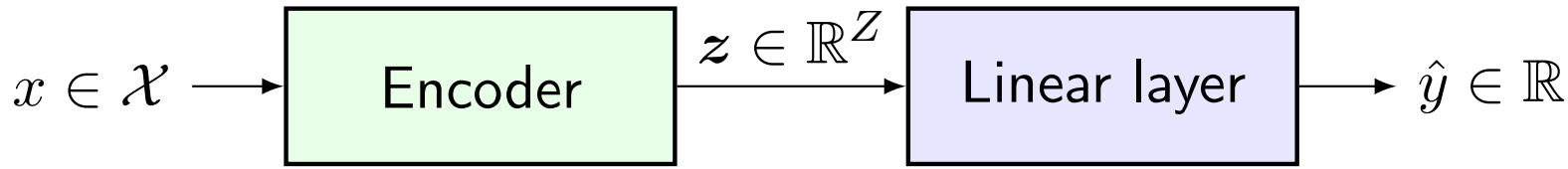
Softmax layer



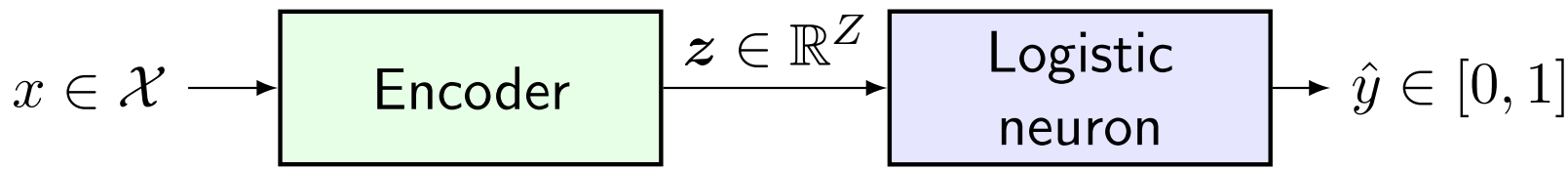
- A **softmax layer** computes $\hat{\boldsymbol{y}} = S\left(\frac{\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}}{T}\right)$
  - $\hat{\boldsymbol{y}} \in \mathcal{S}_C$ is a probability vector
  - $T \in \mathbb{R}_+$ is a hyperparameter known as the **temperature**
    $\rightarrow$ Controls smoothness of distribution (assume $T = 1$ for now)
- FNN with single softmax layer trained with MLE / ERM + log loss
  - $\hat{y}_c$ is model confidence in label c
  - Equivalent to multinomial logistic regression (softmax regression)
- **Output layer of FNNs for multi-class classification tasks is typically a softmax layer**

Summary: Typical output layers

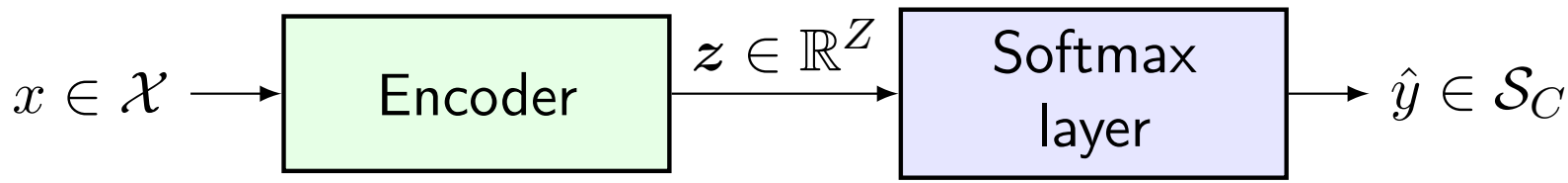
- Regression



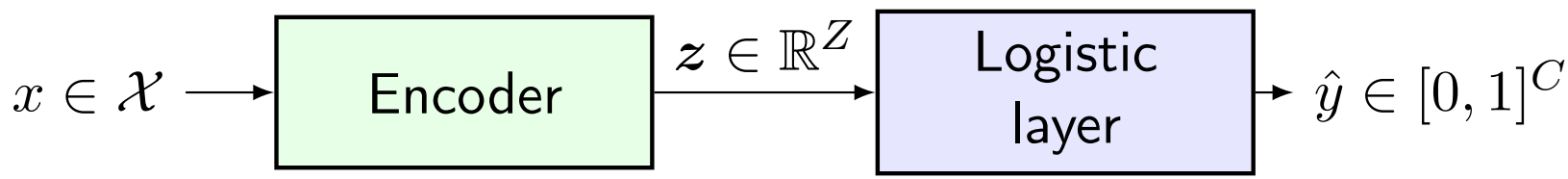
- Binary classification



- Multi-class classification (C classes)



- Multi-label classification (C labels)



IE 678 Deep Learning

02 – Feedforward Neural Networks

Part 5: Multi-Layer Perceptrons

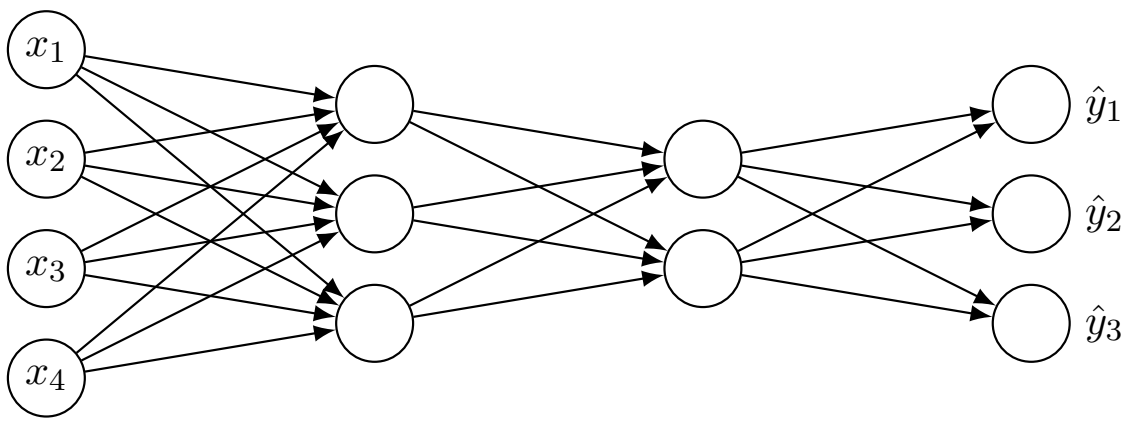
Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-1

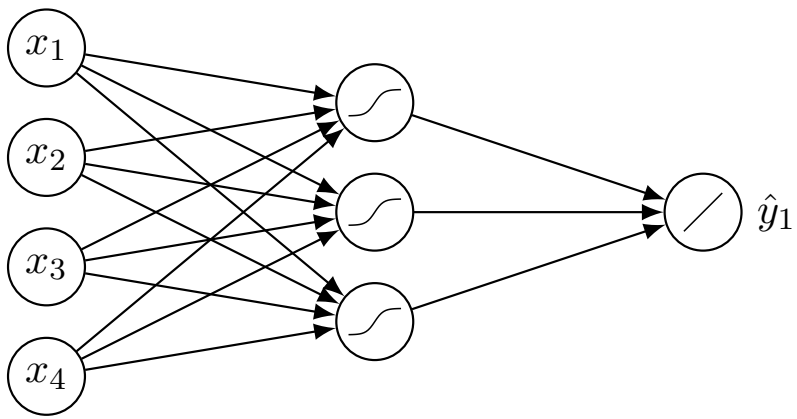
Multi-layer FNNs

- So far: mostly FNNs without hidden layers
  - These networks are limited in what they can do
  - Linear regression, logistic regression, ...
  - We can improve performance by engineering better features
- In neural networks, hidden layers are generally necessary
  - Recall: can interpret hidden layers as features for the next layer
  - By including hidden layers, we aim to let the network "do" the feature engineering
  - If there is at least one hidden layer, the network is called a **multi-layer FNN**
  - If more than one (or more than some value $L > 1$), called **deep**



How powerful are multi-layer FNNs? (1)

- Example: **multi-layer perceptron (MLP)**
  - Fully connected layers
  - Number/sizes of hidden layers are hyperparameters
  - Hidden layers all of same type and non-linear (e.g., logistic neuron)
  - Output layer is linear (regression) or logistic/softmax (classification)
- How powerful are such networks?
- Consider a basic **wide MLP** with
  - D inputs, 1 linear output
  - One fully-connected hidden layer with Z sigmoidal neurons
- **Universal approximation theorem**: This wide MLP can approximate any continuous function on $[0, 1]^D$ given sufficiently (but finitely) many hidden neurons [Cybenko, 1989]



How powerful are multi-layer FNNs? (2)

- Similar universality results exist for **deep MLPs**
  - Any continuous function on $[0, 1]^D$ can be approximated arbitrarily well given sufficiently (but finitely) many hidden layers, each with at most $D + 1$ units in each hidden layer [Hanin and Selke, 2017]
- Universal approximation means that "any" function can be represented
  - Either via sufficient width or sufficient depth
- But that doesn't mean that we can learn that function
  - Training methods may fail to find good parameterization
  - Overfitting may occur
  - Number of required units can be exponential in the input dimensionality
  - ...

Wide or deep?

- Deep models tend to show better generalization performance...
  - ...for suitable architectures ($\neq$ MLP)
  - Encode belief that function to be learned involves a composition of several simpler functions
  - Interpretation: hidden layer output = intermediate values in a multi-step computation
- But modern deep models can also be wide...
  - ...for suitable architectures ($\neq$ MLP)
  - E.g., T5-Base has maximum width of $3072 \times$ number of tokens
  - E.g., 1000 tokens $\rightarrow$ width $\approx$ 3M neurons
- And other considerations also matter; e.g., compute cost, memory, parallelizability, ...(more later)
- So, wide or deep? $\rightarrow$ Both! Depends!

Example (deep): CNNs

Task: transcribe multi-digit numbers from photographs

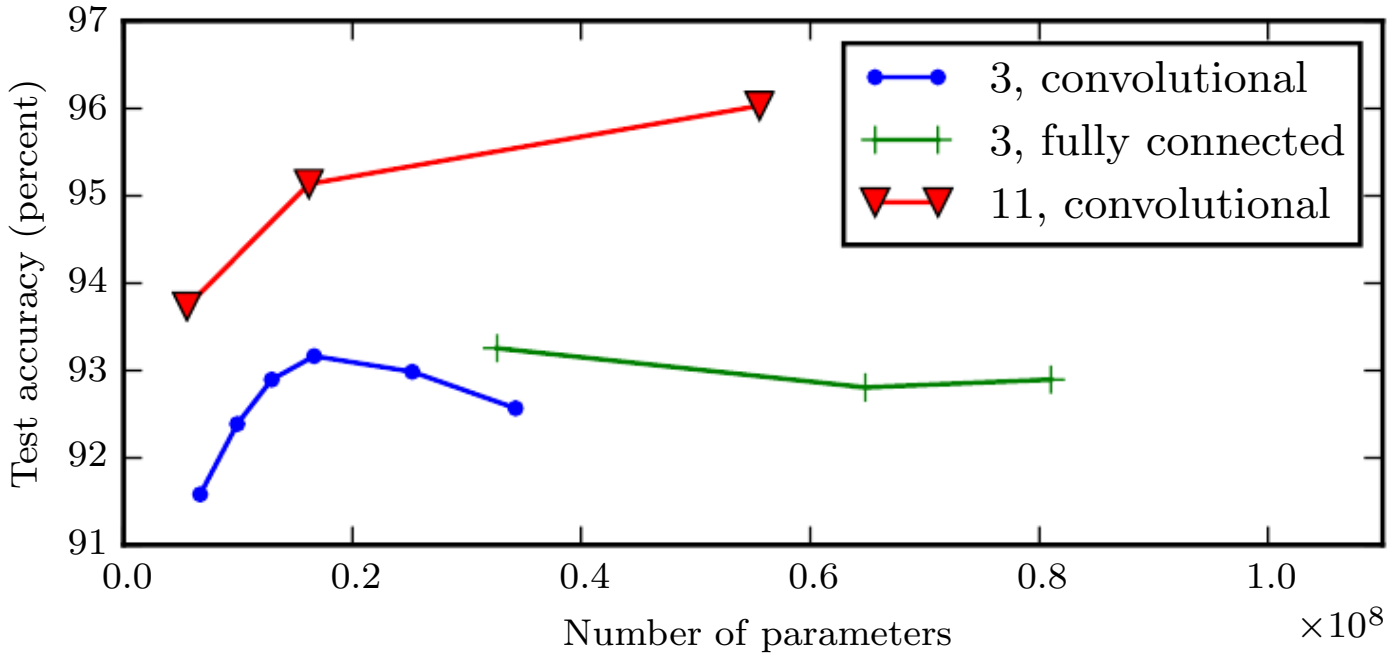
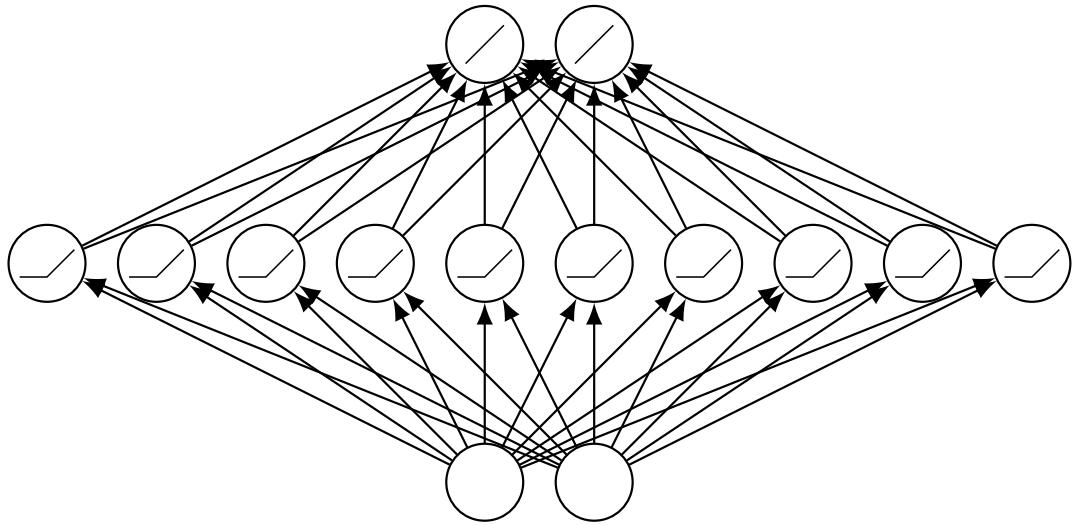


Figure 6.7: Deeper models tend to perform better. This is not merely because the model is larger. This experiment from Goodfellow *et al.* (2014d) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance. The legend indicates the depth of network used to make each curve and whether the curve represents variation in the size of the convolutional or the fully connected layers. We observe that shallow models in this context overfit at around 20 million parameters while deep ones can benefit from having over 60 million.

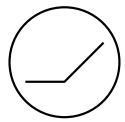
Example (wide): Inverted Bottlenecks

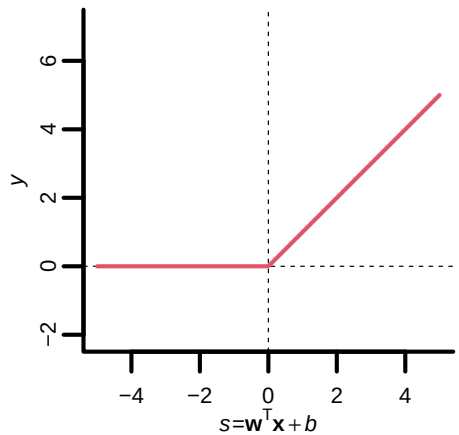
- **Inverted bottleneck** = a wider hidden layer; e.g.,



- MLP that enhances expressivity
  - Think: a more powerful compute step (in the multi-step computation interpretation)
  - Without substantially increasing depth
  - Without increasing width for subsequent parts of network
- E.g., T5-Base uses $768 \to 3072 \to 768$ "feedforward blocks"
  - Large maximum width mainly due to these inverted bottleneck blocks

Rectified linear neuron (ReLU)

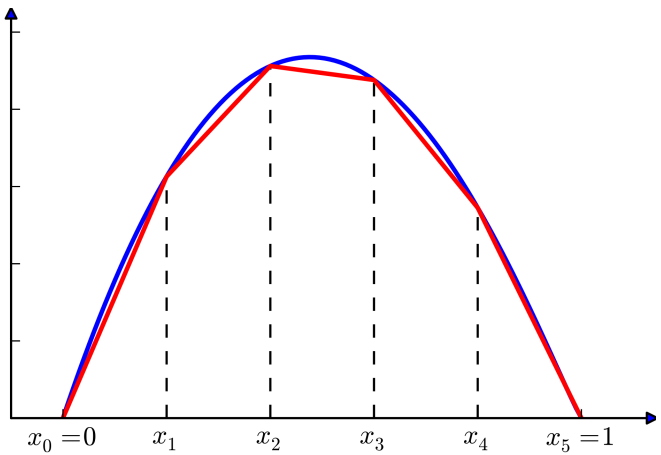
- Also called **linear threshold neuron** or **rectifier**
- $\phi(s) = \max\{0, s\} = \begin{cases} s & \text{if } s \geq 0 \\ 0 & \text{otherwise} \end{cases}$
- Notation: 



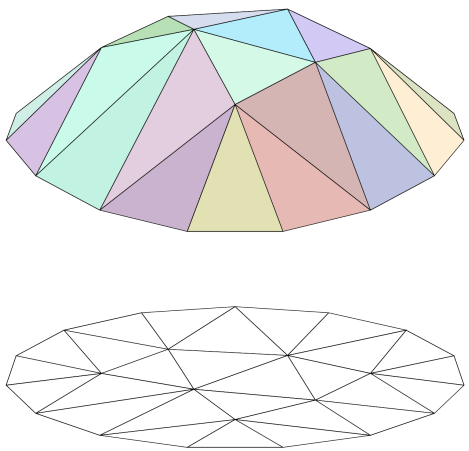
- Note: again, a non-linear transfer function
- **Common non-linearity for intermediate layers in deep NNs**

Rectifier networks

- **Rectifier network** = MLP with only rectifier units in hidden + output layers
- Function computed by rectifier network is **piecewise linear**
  - Approximates a function by decomposing it into **linear regions**
  - Roughly: the more linear regions, the more flexible / expressive



1D example



2D example

Wikipedia

Expressivity of rectifier networks

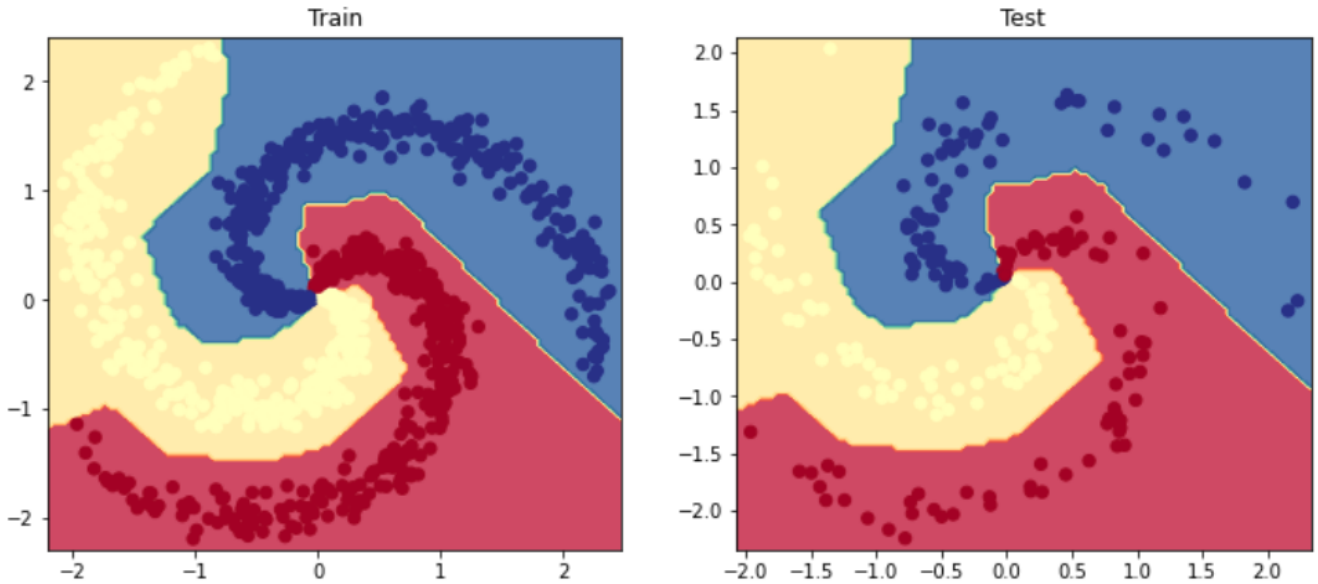
Montúfar et al. (2014) have shown:

- Consider rectifier networks of form
  - D = input dimensionality (assumed constant)
  - H = total number of hidden units
  - L = total number of hidden layers, each of width $Z \geq D$
- Number of linear regions *at most* 2^H
  - $\rightarrow$ **No more than exponentially many linear regions possible**
- Number of attainable linear regions *at least* $\Omega((Z/D)^{(L-1)D} Z^D)$
  - Attainable = maximum over all possible parameterizations
  - Polynomial in Z (width)
  - Exponential in L (depth)
  - $\rightarrow$ **Exponentially many linear regions indeed possible**

Rectifier networks and classification

Consider binary classification.

- Take rectifier network and add logistic neuron on top for output
- Each linear region then mapped to two classes separated by a hyperplane (or to one class only)
- So: part of input space for which model predicts **each class is given by (at most) as many linear regions** as rectifier network
- Similar arguments for multi-class classification



Architecture: Input (2) $\rightarrow$ ReLu (100) $\rightarrow$ Softmax (3)

Source: Notebook of Goku Mohandas

Deep Learning

03 – Gradient-Based Training

Part 0: Overview

Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-1

Supervised training of FNNs

- In principle: like any other ML model
- Often: empirical risk minimization (our focus)
  - Frequentist approach, obtains point estimate $\hat{\boldsymbol{\theta}}$ of FNN parameters $\boldsymbol{\theta}$
  - Use non-negative, real-valued **loss function** $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ between a prediction $\hat{\boldsymbol{y}}$ and a true answer $\boldsymbol{y}$
  - Minimize **empirical risk** = average loss on training data $\left\{ (\boldsymbol{x}_i, \boldsymbol{y}_i)_{i=1}^N \right\}$

$$R_{\text{emp}}(\boldsymbol{\theta}) = \frac{1}{N} \sum_i L(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i) \qquad \text{where } \hat{\boldsymbol{y}}_i = f(\boldsymbol{x}_i; \boldsymbol{\theta})$$

- Some common loss functions
  - Squared error (for regression)
  - Log loss / binary cross entropy (for binary / multi-label classification)
  - Cross entropy / KL divergence (for multi-class classification)
  - Hinge loss (for margin-based classification)
  - 0-1 loss / misclassification rate (for classification)
- Generally: use **cost function** $J(\boldsymbol{\theta})$
  - E.g., regularized risk to prevent overfitting

Gradient-based methods

- Gradient-based methods are dominant
  - Large datasets, many parameters
  - Many tricks used to make these methods work empirically
- General approach
  1. Construct a **batch** (e.g., a subset of examples)
  2. Compute gradients of cost function on batch
  3. Update parameters using an **optimizer**
  4. Repeat

Training Techniques

- Gradient-based methods are a *tool* to minimize some cost function (backpropagation, optimizers)
- Training FNNs successfully is also an *art*; generally, goals include
  - Improve performance of gradient-based methods
  - Reduce overfitting, improve generalizability
  - Leverage additional data
  - Reduce (task-specific) costs such as model size, computational costs, enery consumption, amount of required supervision, ...
- In this part of the lecture, we look at
  - Compute graphs, automatic differentiation
  - Gradient computation via **backpropagation** ("backprop"): chain rule + reuse of computations
  - Optimizers beyond plain SGD
  - Challenges in gradient-based training (vanishing/exploding gradients)
  - Mitigating architectural design patterns and their impact

Outline (Gradient-Based Training)

0. Overview
1. Backpropagation
2. Optimizers
3. Architecture design
4. Initialization

Summary

- Gradient-based methods dominant for training deep learning models
- Backpropagation
  - Technique to compute gradient of a computation w.r.t. its inputs
  - Computation modeled via a compute graph
  - Chain rule + reuse of computations
  - Forward/backward pass to compute all outputs/gradients
- Optimizers
  - Batch size and learning rate are key hyperparameters
  - Momentum and adaptive learning rates common
- Architecture design
  - Vanishing/exploding gradients can be problematic
  - Architectural choices matter (e.g., non-saturating units, residual units, skip connections)
  - Suitable initialization depends on architecture

Suggested reading

- Drori, Ch. 2, 3
- Goodfellow et al., Ch. 6, 8
- Murphy 1, Ch. 13.3, 13.4

Deep Learning

03 – Gradient-Based Training

Part 1: Backpropagation

Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-3

Backpropagation

- **Backpropagation** is an algorithm to compute gradients
  - Origins in the 60s in control theory
  - Rediscovered many times
  - Used for neural networks since the 80s
- Given a compute graph, performs
  1. Forward pass to compute (all) output(s) (**forward propagation**)
  2. Backward pass to compute (all) gradient(s) (**backward propagation**)
- For us: compute graph typically represents
  - Output $\hat{y}$ of an FNN (given $\boldsymbol{x}$, $\boldsymbol{\theta}$)
  - Loss L of an FNN (given $(\boldsymbol{x}, y)$, $\boldsymbol{\theta}$)
  - Cost function J for an FNN (given $\{ (\boldsymbol{x}_i, y_i) \}$, $\boldsymbol{\theta}$)
- And we are interested in gradients (as we will see)
  - W.r.t. weights ($\nabla_{\boldsymbol{\theta}} J$): e.g., for gradient-based training
  - W.r.t. intermediate outputs ($\nabla_{\boldsymbol{z}} L$): e.g., for model debugging
  - W.r.t. inputs ($\nabla_{\boldsymbol{x}} L$ or $\nabla_{\boldsymbol{x}} \hat{y}$): e.g., for sensitivity analysis or adversarial training

Recap: Gradient (ML, 05-2)

- For functions with multiple inputs, there are multiple **partial derivatives**; e.g.,

$$f = x_1^2 + 5x_1x_2$$
$$\frac{\partial}{\partial x_1} f = 2x_1 + 5x_2$$
$$\frac{\partial}{\partial x_2} f = 5x_1$$

- We can gather them all in a single vector, the **gradient** of f

$$\nabla_{\boldsymbol{x}^\top} f \stackrel{\text{def}}{=} \begin{pmatrix} \frac{\partial}{\partial x_1} f & \frac{\partial}{\partial x_2} f & \cdots & \frac{\partial}{\partial x_n} f \end{pmatrix}$$

- For the example above, we obtain

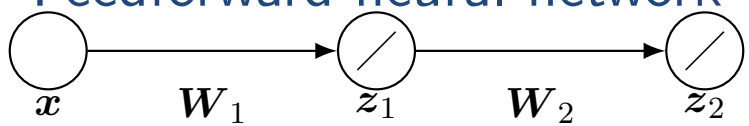
$$\nabla_{\boldsymbol{x}^\top} f = \begin{pmatrix} 2x_1 + 5x_2 & 5x_1 \end{pmatrix} \qquad \nabla_{\boldsymbol{x}} f = \begin{pmatrix} 2x_1 + 5x_2 \\ 5x_1 \end{pmatrix}$$

Numerator layout (row) Denominator layout (column)

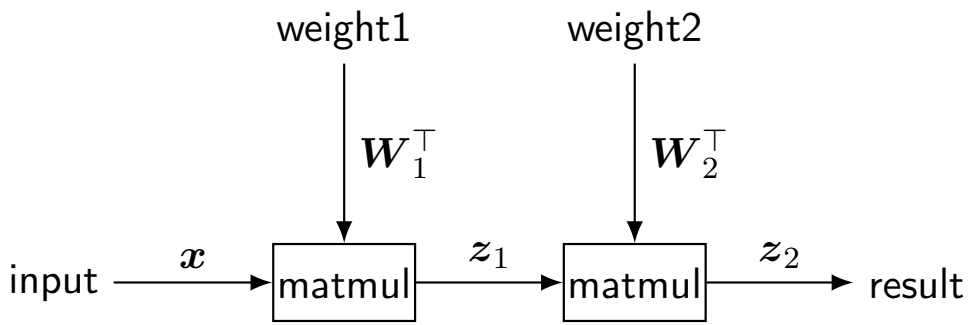
Compute graphs

- Backpropagation generally operates on a **compute graph**
- Directed, acyclic graph that models a computation (as a *data flow program*)
- Vertices correspond to operations
- Edges correspond to data passed between operations (typically tensor-valued)
- Multiple **sources** (no incoming edge): inputs, weights, ...
- One **sink** (no outgoing edge): result

Feedforward neural network

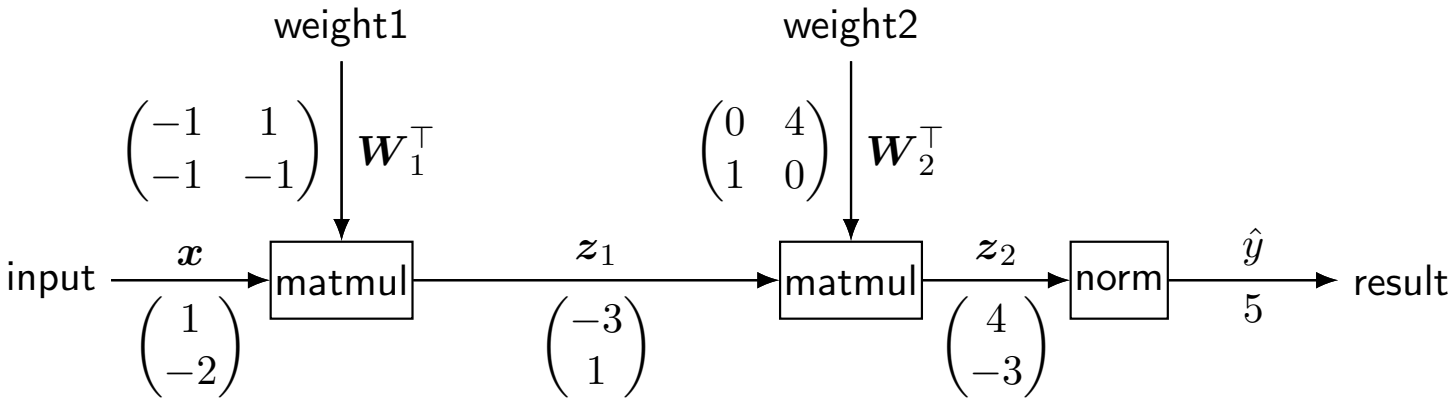


Corresponding compute graph



Forward propagation (example)

- Compute graph for example output $\hat{y}$



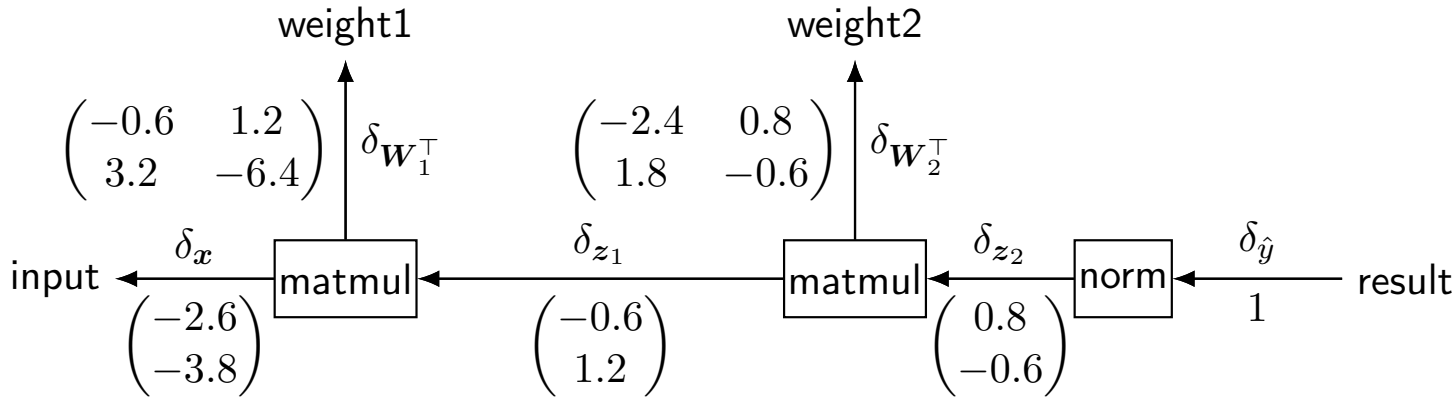
- **Forward propagation: inputs $\rightarrow$ result**
- Edges transport values
- For example:
  1. Provide inputs $\boldsymbol{x}$, $\boldsymbol{W}_1^\top$, $\boldsymbol{W}_2^\top$
  2. Evaluate first matmul: $\boldsymbol{z}_1 = \boldsymbol{W}_1^\top \boldsymbol{x}$
  3. Evaluate second matmul: $\boldsymbol{z}_2 = \boldsymbol{W}_2^\top \boldsymbol{z}_1$
  4. Evaluate norm: $\hat{y} = \|\boldsymbol{z}_2\|$

Forward propagation

- Operators are evaluated in **topological order** ("forwards")
  - Whenever an operator is evaluated, all its inputs must be available
  - Computation is **local**: only input values are required (the remainder of the compute graph does not matter)
- Inputs and/or outputs are generally tensor-valued
  - E.g., $\mathrm{matmul}(\boldsymbol{A}, \boldsymbol{B}) = \boldsymbol{A}\boldsymbol{B}$ takes two 2D tensors and produces a 2D tensor
  - Note: our visual representation of compute graph does not indicate which input is $\boldsymbol{A}$ and which is $\boldsymbol{B}$, but the actual compute graph does (and must do so)
- Intermediate results may need to be kept
  - To evaluate subsequent operators
  - To enable gradient computation with backpropagation
- Parallel processing is possible
  - Individual operators can be evaluated in parallel (e.g., matmul)
  - Different operators can be evaluated in parallel (when their respective inputs are available)
  - E.g., transformer encoders can operate on all inputs in parallel
  - E.g., RNN encoders must process inputs sequentially

Backward propagation (example)

- Backward graph for example output $\hat{y}$



- **Backward propagation: result $\rightarrow$ gradients**
- Edges transport gradients
  - Consider edge e and define

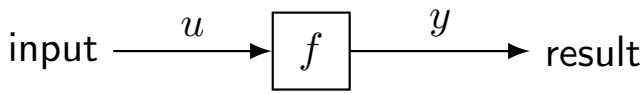
    $$\delta_e \overset{\text{def}}{=} \textbf{gradient of result w.r.t. values on edge } e$$

    evaluated at the provided inputs
- For example:
  1. Compute all values of forward pass (not shown above)
  2. $\delta_{\hat{y}} \overset{\text{def}}{=} \nabla_{\hat{y}} \text{result} = \nabla_{\hat{y}} \hat{y} = 1$
  3. $\delta_{\boldsymbol{z}_2}$ (discussed later)
  4. $\delta_{\boldsymbol{W}_2^\top}$ and $\delta_{\boldsymbol{z}_1}$ (discussed later)
  5. $\delta_{\boldsymbol{W}_1^\top}$ and $\delta_{\boldsymbol{x}}$ (discussed later)

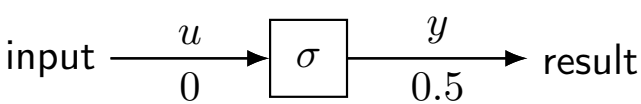
Backward propagation

- $\delta_e \stackrel{\text{def}}{=}$ **gradient of result w.r.t. values on edge** e
- Key insight of backpropagation
  - Gradients δ_e can be computed incrementally (akin to forward pass, but in reverse order)
- Operators are evaluated in **reverse topological order** ("backwards")
  - When operator evaluated, its output gradient(s) must be available
  - Computation is **local**: only input values and output gradient(s) are required (the remainder of the compute graph does not matter)
  - Recall: intermediate outputs of forward pass required
    $\rightarrow$ memory consumption (or recompute)
- Gradients are generally tensor-valued
  - Convention: δ_e **same shape as values on edge** e **in forward pass**
- Intermediate results may need to be kept
  - To evaluate gradient for prior operators
  - To debug/analyze models
- Parallel processing is possible (as before)

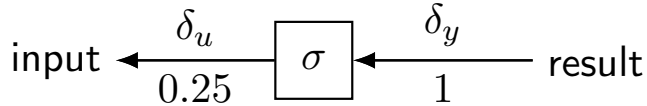
Gradient (single univariate function)



- Result: $y = f(u)$
- Gradient $\delta_y \stackrel{\text{def}}{=} \nabla_y \text{result} = \nabla_y y = 1$
- Gradient $\delta_u \stackrel{\text{def}}{=} \nabla_u \text{result} = \frac{\partial}{\partial u} f(u) = f'(u)$
- Example
  - $u = 0$, $f = \sigma$ (logistic function)
  - $y = \sigma(u) = \sigma(0)$
  - $\delta_u = \sigma'(u) = \sigma(u)(1 - \sigma(u)) = \sigma(0)(1 - \sigma(0))$



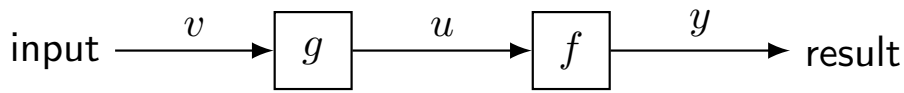
Forward pass



Backward pass

Gradient (composition of two univariate functions)

- Let's add another operator g in front



- Result: $y = f(u) = f(g(v))$
  - **Function composition**
- Gradient: $\delta_u \stackrel{\text{def}}{=} \nabla_u y = \frac{\partial}{\partial u} f(u) = f'(u) = f'(u)$
  - Same computation as before (but u now output of g)
  - Need to retain u in forward pass to compute $f'(u)$
- Gradient

$$\delta_v \stackrel{\text{def}}{=} \nabla_v y = \underbrace{\frac{\partial}{\partial v} f(g(v)) = g'(v) f'(g(v))}_{\text{chain rule}} = g'(v) f'(u)$$

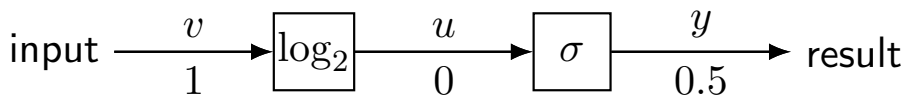
$$= g'(v)\delta_u$$

  - **Observe: that's a local computation at g**
  - Need: $\delta_u \rightarrow$ passed backwards from subsequent operators
  - Need: $v \rightarrow$ computed in forward pass
  - Need: $g' \rightarrow$ determined by g

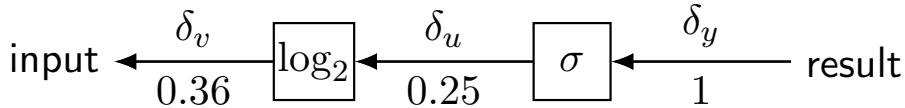
Example

- $v = 1,\ g = \log_2,\ f = \sigma$

Forward pass



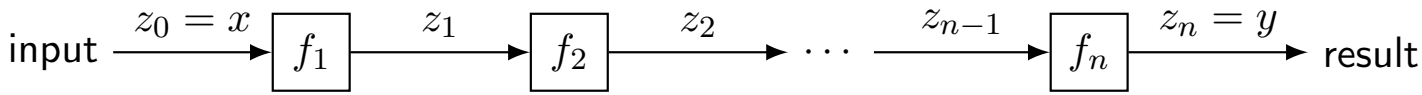
Backward pass



- $\delta_y \stackrel{\text{def}}{=} \nabla_y y = 1$
- $\delta_u \stackrel{\text{def}}{=} \nabla_u y = \sigma'(u)\delta_y = \sigma(u)(1 - \sigma(u))\delta_y = 0.25 \cdot 1 = 0.25$
- $\delta_v \stackrel{\text{def}}{=} \nabla_v y = \log_2'(v)\delta_u = \frac{1}{v \log(2)}\delta_u \approx 1.44 \cdot 0.25 = 0.36$

Gradient (composition of univariate functions)

- This generalizes; e.g., consider n operators



- We have

$$y = f_n(f_{n-1}(\cdots(f_1(x))\cdots))$$

- At each operator f_i, the required gradient can be computed as follows:

$$\delta_{z_{i-1}} \stackrel{\text{def}}{=} \nabla_{z_{i-1}} y = \underbrace{\frac{\partial y}{\partial z_{i-1}} = \overbrace{\frac{\partial y}{\partial z_i}}^{\text{output derivative}} \overbrace{\frac{\partial z_i}{\partial z_{i-1}}}^{\text{local derivative}}}_{\text{chain rule}}$$

$$= \underbrace{f_i'(z_{i-1})}_{\text{local derivative}} \cdot \underbrace{\delta_{z_i}}_{\text{output derivative}}$$

Overall gradient

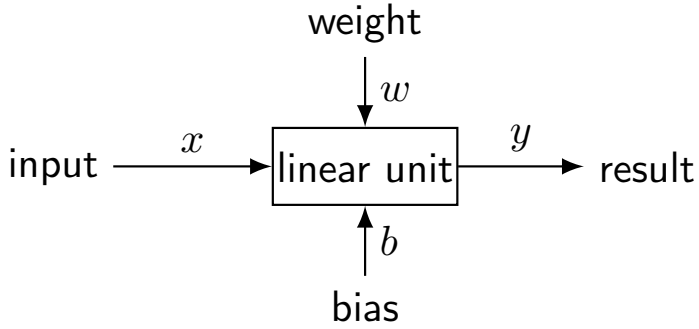
- Let's derive an expression for the gradients individually

$$
\begin{aligned}
\delta_{z_n} &= 1 \\
\delta_{z_{n-1}} &= f'_n(z_{n-1})\delta_{z_n} &&= f'_n(z_{n-1}) \\
\delta_{z_{n-2}} &= f'_{n-1}(z_{n-2})\delta_{z_{n-1}} &&= f'_{n-1}(z_{n-2})f'_n(z_{n-1}) \\
\delta_{z_{n-3}} &= f'_{n-2}(z_{n-3})\delta_{z_{n-2}} &&= f'_{n-2}(z_{n-3})f'_{n-1}(z_{n-2})f'_n(z_{n-1}) \\
&\vdots
\end{aligned}
$$

- **Gradient is product of local gradients along the path from the result to the resp. edge**
- Backpropagation avoids repeated computations by
  1. Proceeding backwards
  2. Using the chain rule to reuse intermediate results (i.e., the δ-values)

Gradient (multiple inputs)

- Operators often have multiple inputs; e.g., a simple linear unit



- In the forward pass, the operator computes

$$y = f(x, w, b) = wx + b$$

- In the backward pass, we compute gradients of result w.r.t. each edge as before (using the chain rule)

$$\delta_y = 1$$
$$\delta_x = \nabla_x y = \nabla_x f(w, x, b) \cdot \delta_y = w \cdot 1$$
$$\delta_w = \nabla_w y = \nabla_w f(w, x, b) \cdot \delta_y = x \cdot 1$$
$$\delta_b = \nabla_b y = \nabla_b f(w, x, b) \cdot \delta_y = 1 \cdot 1$$

- $\rightarrow$ **Consider each input separately**

Gradient (multiple outputs)

- Operators may have multiple outputs; e.g., consider
  - E.g., operator $f(x)$ may output n values, say, $z_1 = f_1(x)$, ..., $z_n = f_n(x)$
    - Each of these outputs contributes to result y
  - During backpropagation, we obtain $\delta_{z_1}, \ldots, \delta_{z_n}$
  - We are interested in

$$\delta_x = \nabla_x y = \underbrace{\frac{\partial y}{\partial x} = \sum_{k=1}^{n} \overbrace{\frac{\partial y}{\partial z_k}}^{\text{output gradient}} \overbrace{\frac{\partial z_k}{\partial x}}^{\text{local gradient}}}_{\text{multivariate chain rule}}$$

$$= \sum_{k=1}^{n} \overbrace{f_k'(x)}^{\text{local gradient}} \overbrace{\delta_{z_k}}^{\text{output gradient}}$$

  - $\rightarrow$ **Consider each output separately and sum up**

Gradient (multiple uses)

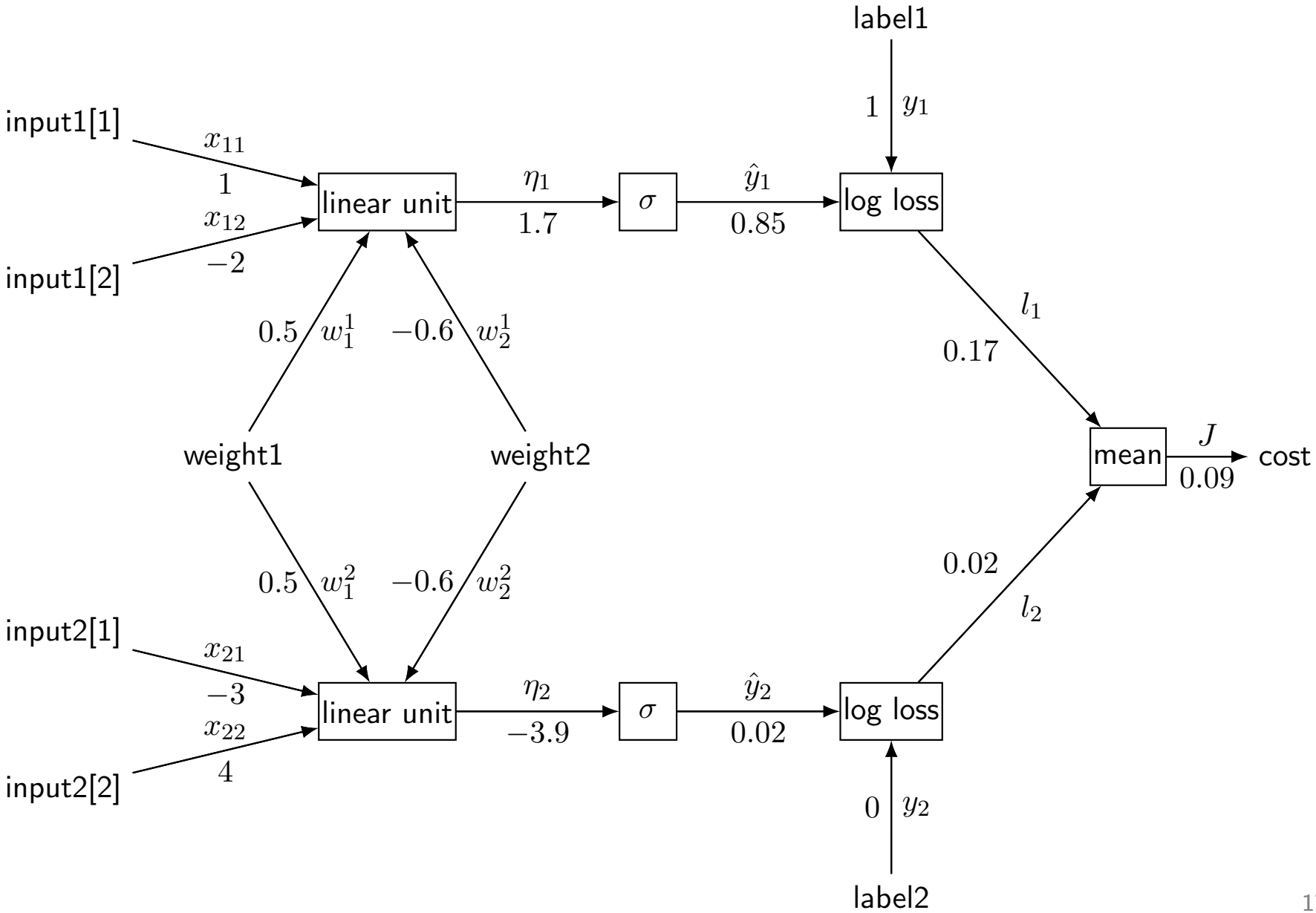
- Sometimes an operator's output is "used" multiple times
  - E.g., the (single) output of an operator $g(x)$ is used n times
  - That's equivalent to a single operator f with n identical outputs (i.e., $\boldsymbol{z} = f(x) = \mathbf{1}_n g(x)$ and thus $z_k = f_k(x) = g(x)$), each being used once
  - Using the results from the previous slide with f defined in this way:

$$\delta_x = \nabla_x y = \sum_{k=1}^{n} f_k'(x)\delta_{z_k} = \sum_{k=1}^{n} g'(x)\delta_{z_k}$$
$$= g'(x)\sum_{k=1}^{n}\delta_{z_k}$$

  - $\rightarrow$ **Sum up all incoming δ-values and proceed as before**

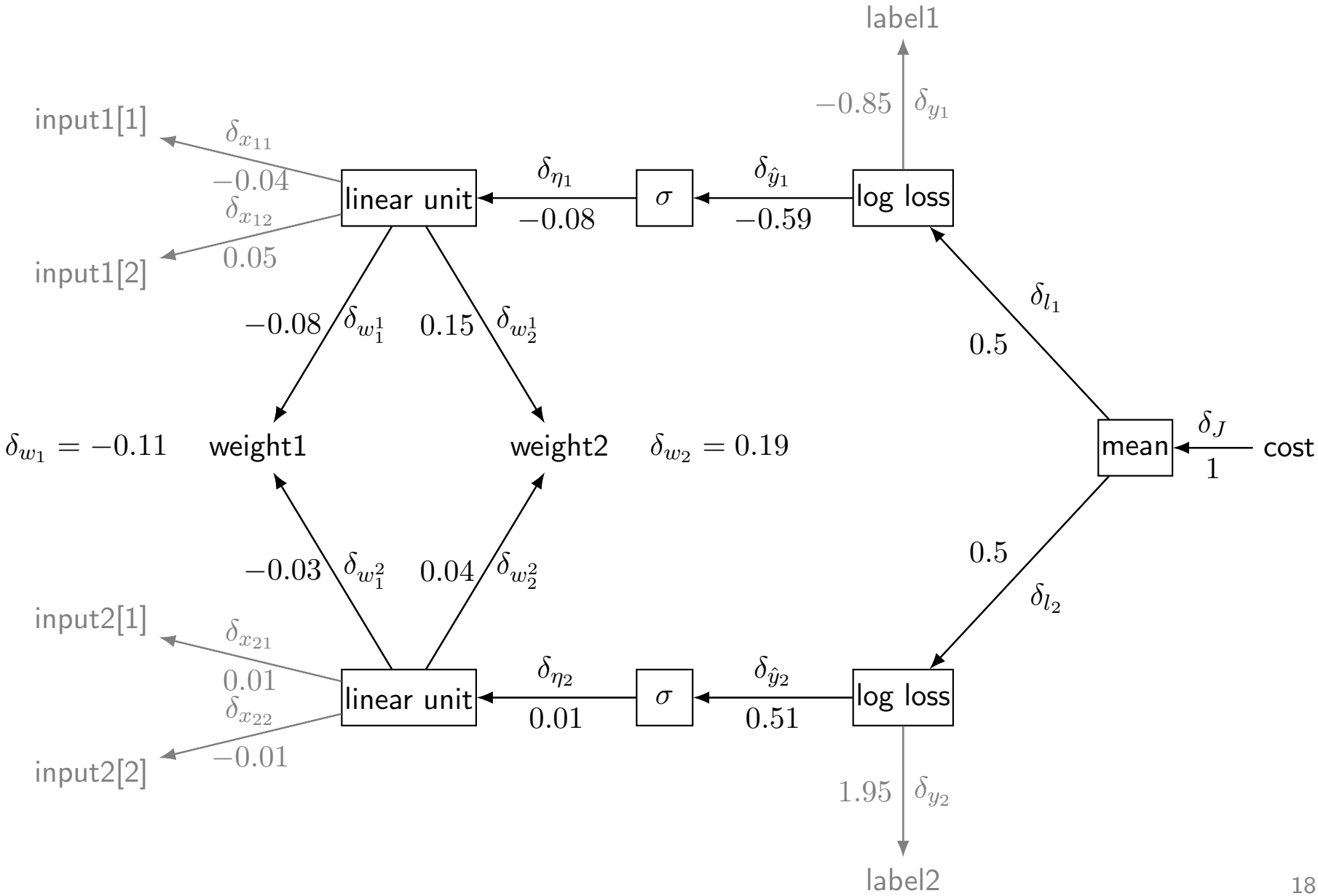
Example: logistic regression (2D), $N = 2$

Forward pass



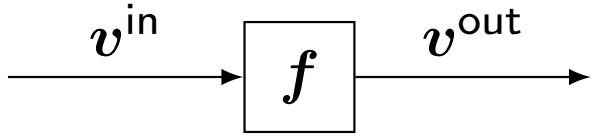
Example: logistic regression (2D), $N = 2$

Backward pass

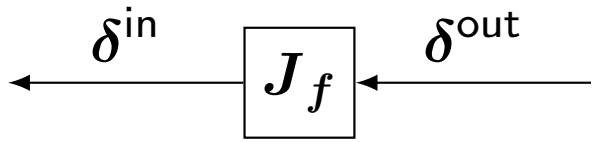


Gradient computation in general

- Consider an operator $\boldsymbol{f} : \mathbb{R}^I \to \mathbb{R}^O$
- Forward pass: $\boldsymbol{v}^{\text{out}} = \boldsymbol{f}(\boldsymbol{v}^{\text{in}})$ with $\boldsymbol{v}^{\text{in}} \in \mathbb{R}^I$ and $\boldsymbol{v}^{\text{out}} \in \mathbb{R}^O$



- Backward pass: $\boldsymbol{\delta}^{\text{in}} = \boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{v}^{\text{in}})^\top \boldsymbol{\delta}^{\text{out}}$ with $\boldsymbol{\delta}^{\text{out}} \in \mathbb{R}^O$ and $\boldsymbol{\delta}^{\text{in}} \in \mathbb{R}^I$



where we use the **Jacobian** $\boldsymbol{J}_{\boldsymbol{f}}$ of shape $O \times I$ given by

$$\boldsymbol{J}_{\boldsymbol{f}} \overset{\text{def}}{=} \nabla_{\boldsymbol{v}_{\text{in}}^\top} \boldsymbol{f} = \begin{pmatrix} \nabla_{\boldsymbol{v}_{\text{in}}^\top} v_1^{\text{out}} \\ \vdots \\ \nabla_{\boldsymbol{v}_{\text{in}}^\top} v_O^{\text{out}} \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial v_1^{\text{in}}} v_1^{\text{out}} & \cdots & \frac{\partial}{\partial v_I^{\text{in}}} v_1^{\text{out}} \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial v_1^{\text{in}}} v_O^{\text{out}} & \cdots & \frac{\partial}{\partial v_I^{\text{in}}} v_O^{\text{out}} \end{pmatrix}$$

- Intuitively, $f'(v^{\text{in}})\delta^{\text{out}}$ (for scalars) now becomes $\boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{v}^{\text{in}})^\top \boldsymbol{\delta}^{\text{out}}$
  - Can be derived by "rewriting" the discussions on multiple inputs/outputs from the previous slides into matrix form
- More in exercises and tutorials

Gradient computation in general (example)

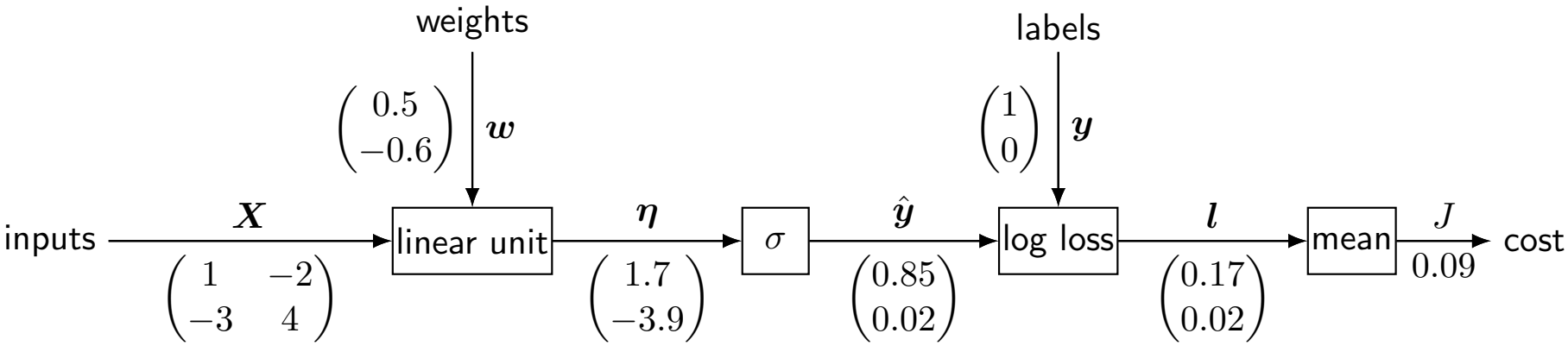
- Let $\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ and $\boldsymbol{f}(\boldsymbol{x}) = \begin{pmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{pmatrix} = \begin{pmatrix} \ln(x_1) + x_2^2 \\ x_1 x_2 \end{pmatrix}$
- Jacobian is $\boldsymbol{J}_{\boldsymbol{f}} = \begin{pmatrix} \frac{\partial}{\partial x_1} f_1 & \frac{\partial}{\partial x_2} f_1 \\ \frac{\partial}{\partial x_1} f_2 & \frac{\partial}{\partial x_2} f_2 \end{pmatrix} = \begin{pmatrix} 1/x_1 & 2x_2 \\ x_2 & x_1 \end{pmatrix}$
- Let $\boldsymbol{v}^{\text{in}} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$. Then $\boldsymbol{v}^{\text{out}} = \begin{pmatrix} 4 \\ 2 \end{pmatrix}$ and $\boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{v}^{\text{in}}) = \begin{pmatrix} 1 & 4 \\ 2 & 1 \end{pmatrix}$
- Let $\boldsymbol{\delta}^{\text{out}} = \begin{pmatrix} 10 \\ 100 \end{pmatrix}$. Then

$$\boldsymbol{\delta}^{\text{in}} = \boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{v}^{\text{in}})^\top \boldsymbol{\delta}^{\text{out}} = \begin{pmatrix} 1 & 2 \\ 4 & 1 \end{pmatrix} \begin{pmatrix} 10 \\ 100 \end{pmatrix} = \begin{pmatrix} 1 \cdot 10 + 2 \cdot 100 \\ 4 \cdot 10 + 1 \cdot 100 \end{pmatrix}$$

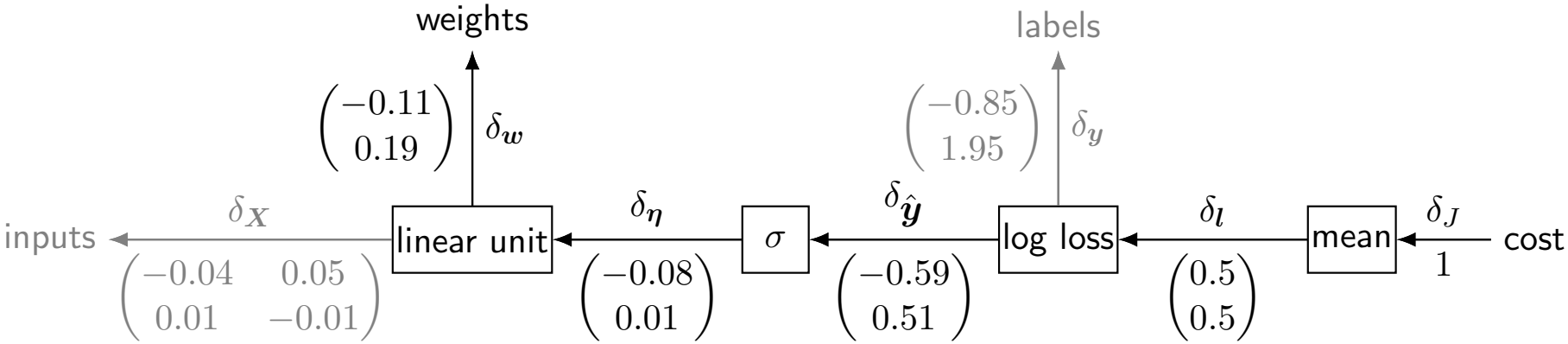
$$= \begin{pmatrix} \frac{\partial}{\partial x_1} f_1(1,2) \cdot \delta_1^{\text{out}} + \frac{\partial}{\partial x_1} f_2(1,2) \cdot \delta_2^{\text{out}} \\ \frac{\partial}{\partial x_2} f_1(1,2) \cdot \delta_1^{\text{out}} + \frac{\partial}{\partial x_2} f_2(1,2) \cdot \delta_2^{\text{out}} \end{pmatrix} = \begin{pmatrix} 210 \\ 140 \end{pmatrix}$$

Example: logistic regression (2D), $N = 2$

Forward pass

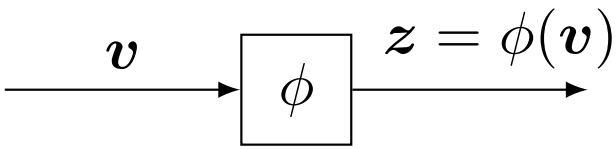


Backward pass

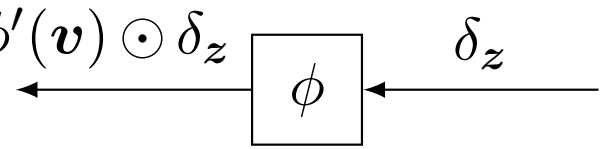


Backprop for selected operators (1)

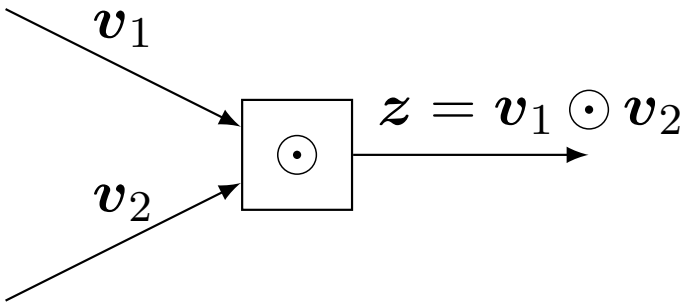
Forward: element-wise op



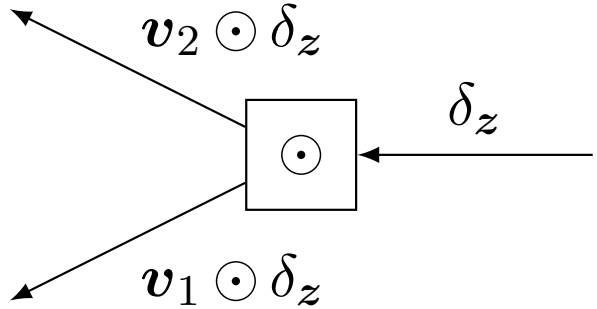
Backward: element-wise derivative



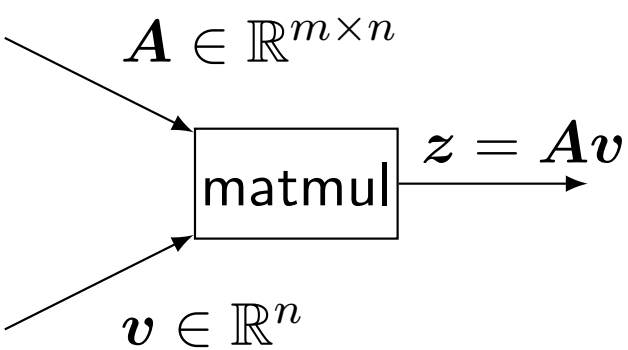
Forward: multiply (element-wise)



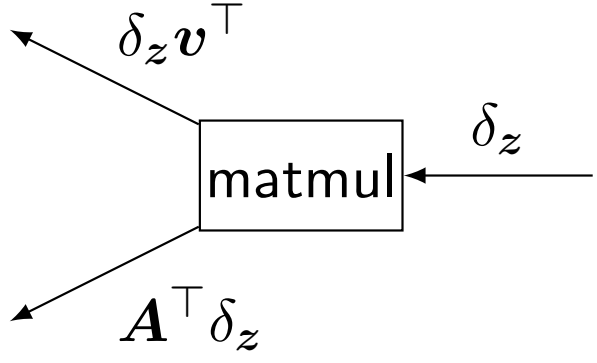
Backward: rescale w/ other



Forward: matmul

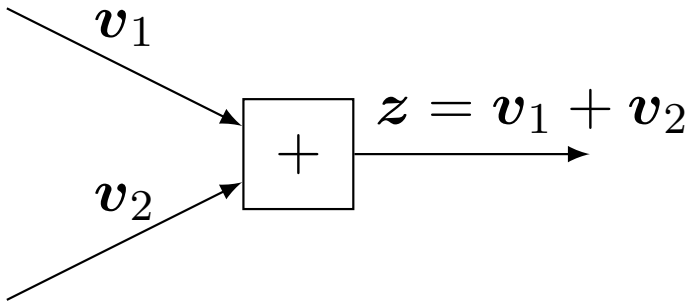


Backward: matmul w/ other

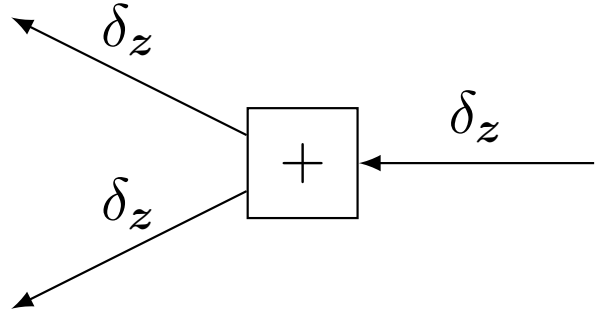


Backprop for selected operators (2)

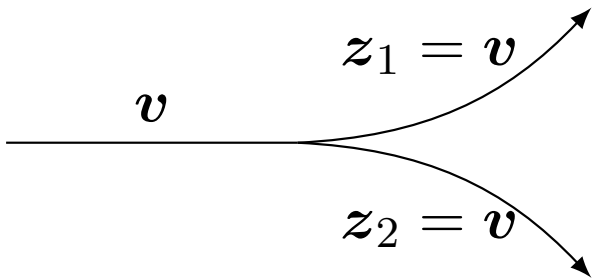
Forward: add



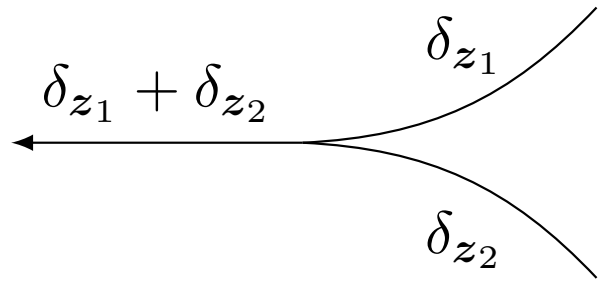
Backward: copy



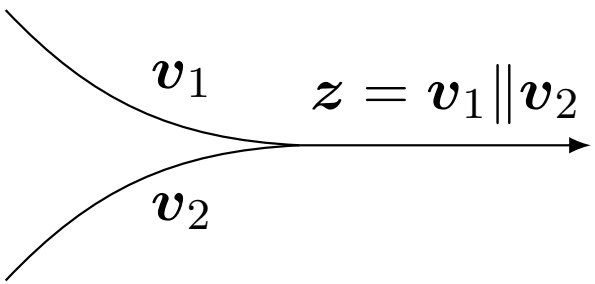
Forward: copy



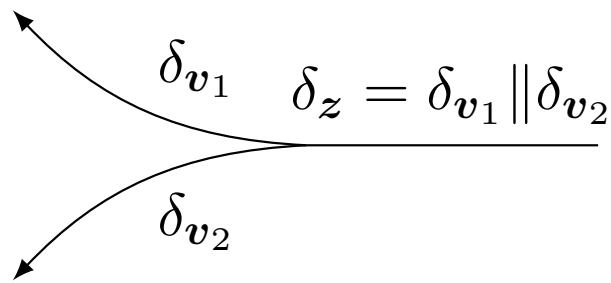
Backward: add



Forward: concatenate



Backward: split



Deep Learning

03 – Gradient-Based Training

Part 2: Optimizers

Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-1

Recap: gradient descent

- In ML lecture (05-2), we discussed vanilla gradient descent

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \epsilon_t \boldsymbol{g}_t,$$

  where $\boldsymbol{g}_t$ is the gradient (or estimate) of the objective function w.r.t. parameters $\boldsymbol{\theta}$ at time t (e.g., $\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$)

- And its variants for supervised learning
  - **Batch** gradient descent: compute exact gradient using all training examples (high cost, easy to parallizable, exact gradient)
  - **Stochastic** gradient descent: estimate gradient using *one* random training example (low cost, hard to parallelize, noisy gradient)
  - **Mini-batch** gradient descent: estimate gradient using *some* training examples (in between; no. examples called **batch size**)

- Discussion
  - Simple to implement and parallelize
  - Suitable for large datasets and models
  - Can be slow to converge (many epochs)
  - Happily gets stuck in local minima

Challenges

- During training, we aim to minimize a potentially highly non-convex cost function $J(\boldsymbol{\theta}) \rightarrow$ difficult
- To make gradient-based methods work, need
  1. Well-chosen training hyperparameters (this part)
  2. Suitable network architecture design (next parts)
- Key training hyperparameter choices include
  - Learning rate
  - Batch size
  - Optimizer

Effect of batch size

- Consider a **cost function** over training examples $\mathcal{D}$ of form

$$J(\boldsymbol{\theta}) = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} L_i(\boldsymbol{\theta}),$$

  where $L_i(\boldsymbol{\theta})$ is the loss on example i (then: J is empirical risk)

- Suppose we construct each batch $\mathcal{B}$ by sampling a fixed number of examples (uniform iid) and average losses

$$J_{\mathcal{B}}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{B}|} \sum_{z \in \mathcal{B}} L_z(\boldsymbol{\theta}) \qquad \text{(a random variable)}$$

$$E[J_{\mathcal{B}}(\boldsymbol{\theta})] = J(\boldsymbol{\theta}) \qquad \text{(since } E[L_z] = J\text{)}$$

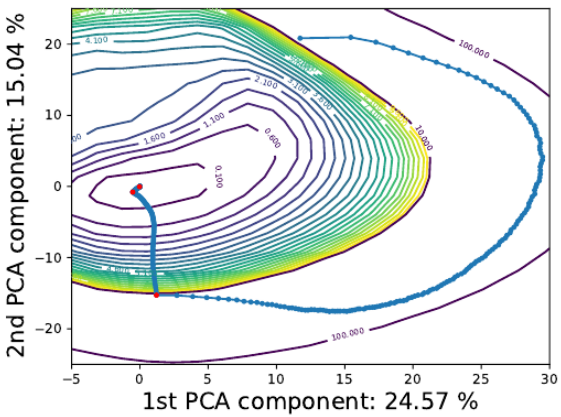
$$E[\nabla_{\boldsymbol{\theta}} J_{\mathcal{B}}(\boldsymbol{\theta})] = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \qquad \text{(since } E[\nabla_{\boldsymbol{\theta}} L_z] = \nabla_{\boldsymbol{\theta}} J\text{)}$$

$$\operatorname{var}[\nabla_{\boldsymbol{\theta}} J_{\mathcal{B}}(\boldsymbol{\theta})] = \frac{1}{|\mathcal{B}|} \operatorname{var}[\nabla_{\boldsymbol{\theta}} L_z(\boldsymbol{\theta})] \qquad \text{(covariance matrix, } \neq \mathbf{0}\text{)}$$

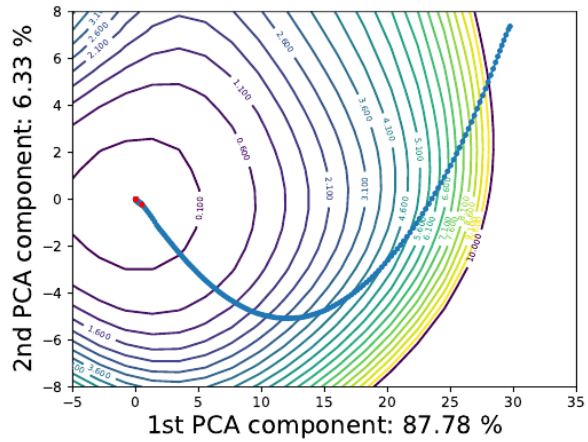
- For this cost & approach: **gradient correct in expectation, variance decreases with increasing batch size** (more in exercises)

Learning rate and batch size (1)

- Example: Projected SGD trajectories for VGG-9 (Li et al. (2018))
  - Red dot → learning rate reduced



Batch size 128



Batch size 8192

- Batch size small → high variance gradients
  - Trajectory takes "detours", regularizing effect
- Batch size large → low variance gradients
  - Trajectory attracted by "sharp" local minima, empirically often lower generalization performance (see Keskar et al. (2017))
- Learning rate low → small steps (slow trajectory)
  - High variance gradient estimates "average out"
- Learning rate high → large steps (fast trajectory)

Learning rate and batch size (2)

- At time t, we
  - Compute approximate gradient $\hat{\boldsymbol{g}}_t$ (e.g., $\hat{\boldsymbol{g}}_t = \nabla_{\boldsymbol{\theta}} J_{\mathcal{B}_t}(\boldsymbol{\theta}_t)$)
  - Perform update $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \epsilon \hat{\boldsymbol{g}}_t$
- Fix t and $\boldsymbol{\theta}_t$ (but not $\mathcal{B}_t$) and define
  - **Step length** $L = \|\epsilon E[\hat{\boldsymbol{g}}_t]\|$ (using expected gradient)
  - **Gradient variance** $V = \mathrm{var}\,[\hat{\boldsymbol{g}}_t]$
  - **Both affected by learning rate, batch size, and cost function**
- Examples (assuming uniformly iid losses as before)

		$J_{\mathcal{B}}$=**mean loss**		$J_{\mathcal{B}}$=sum loss	
	Compute	L	V	L	V
$2\times$ learning rate	stays	$2\times$	stays	$2\times$	stays
$2\times$ batch size	$2\times$	stays	$\frac{1}{2}\times$	$2\times$	$2\times$
$2\times$ l. rate, $\frac{1}{2}$ batch size	$\frac{1}{2}\times$	$2\times$	$2\times$	stays	$\frac{1}{2}\times$
$\frac{1}{2}\times$ l. rate, 2 batch size	$2\times$	$\frac{1}{2}\times$	$\frac{1}{2}\times$	stays	$2\times$

- Note: expected step length $E[\|\epsilon \hat{\boldsymbol{g}}_t\|] \geq L$ (Jensen's inquality)

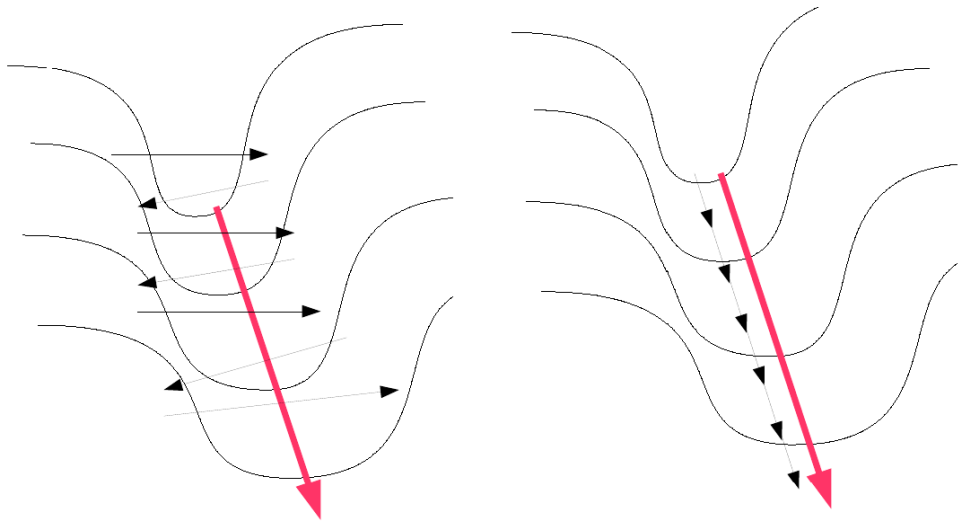
Learning rate and batch size (3)

- Which learning rate and batch size? $\rightarrow$ hyperparameter tuning
- Often a **learning rate scheduler** is used; e.g.,
  - Large learning rate initially (quickly approach vicinity of optimum)
  - Smaller learning rate later (slowly approach optimum)
- Smith et al. (2018): can also use a **batch size scheduler**; e.g.,
  - Small batch size initially (quickly approach vicinity of optimum)
  - Larger batch size later (slowly approach optimum)
  - Advantage: large batch sizes easier to parallize $\rightarrow$ faster

Figure 6: Inception-ResNet-V2 on ImageNet. Increasing the batch size during training achieves similar results to decaying the learning rate, but it reduces the number of parameter updates from just over 14000 to below 6000. We run each experiment twice to illustrate the variance.

Improving upon gradient descent

- Problem: gradient descent can get stuck in "narrow valleys" (ill-conditioned Hessian)



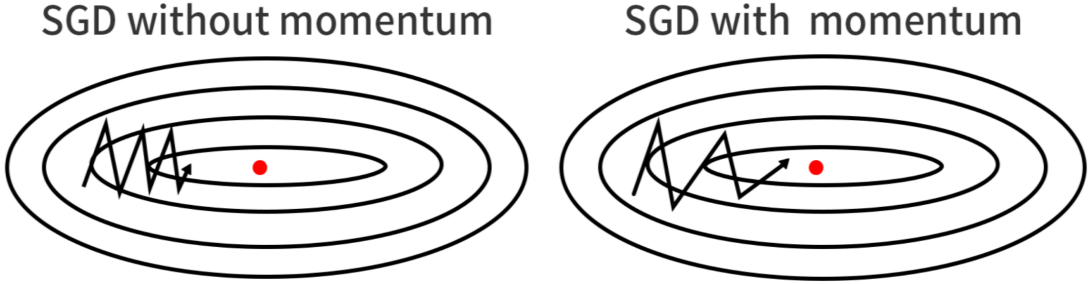
Gradient descent Ideal

- Problem: for SGD, gradient estimates $\hat{\boldsymbol{g}}_t$ may have high variance (e.g., point in "wrong" directions)
- Can we do better?
  - Gradient-based **optimizers** use $\hat{\boldsymbol{g}}_t$ to compute an update term $\boldsymbol{u}_t$ and set $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{u}_t$
  - Plain GD uses: $\boldsymbol{u}_t = -\epsilon \hat{\boldsymbol{g}}_t$
  - General goal: improve convergence properties

Martens, 2010

Momentum (1)

- Idea: build up velocity in directions that have consistent gradient



- Mitigates two problems: poor conditioning of the Hessian matrix (narrow valleys) and variance in the stochastic gradient
- Method of momentum (*heavy-ball method*; Polyak, 1964) uses exponentially-decaying moving average of negative gradient

$$\boldsymbol{v}_t \leftarrow \gamma \boldsymbol{v}_{t-1} - \epsilon \boldsymbol{g}_t$$
$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \boldsymbol{v}_t$$

  - Think of $\boldsymbol{v}$ as **velocity**
  - Hyperparameter $\gamma \in [0, 1)$ referred to as **momentum**
  - Speed (norm of update) increased up to $\frac{1}{1-\gamma}\times$ w.r.t. GD step
  - $\epsilon/(1-\gamma)$ called **effective learning rate**

Momentum (2)

- Variant: Nesterov momentum
  - Apply momentum update before computing gradient

$$\boldsymbol{v}_t \leftarrow \gamma \boldsymbol{v}_{t-1} - \epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t + \gamma \boldsymbol{v}_{t-1})$$
$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \boldsymbol{v}_t$$

- Nesterov (1983) showed: for convex functions (unique global optimum, Lipschitz) and batch gradients, convergence rate improves from $O(1/t)$ to $O(1/t^2)$
  - Does not apply to SGD, however
  - Cost functions in DL are generally not convex
  - But: bounds often loose, in practice much better performance
- Requires additional memory to store velocity $\boldsymbol{v}$
  - Velocity $\boldsymbol{v}$ is of same size as model parameters $\boldsymbol{\theta}$
  - Can be substantial for large models

Adaptive learning rates

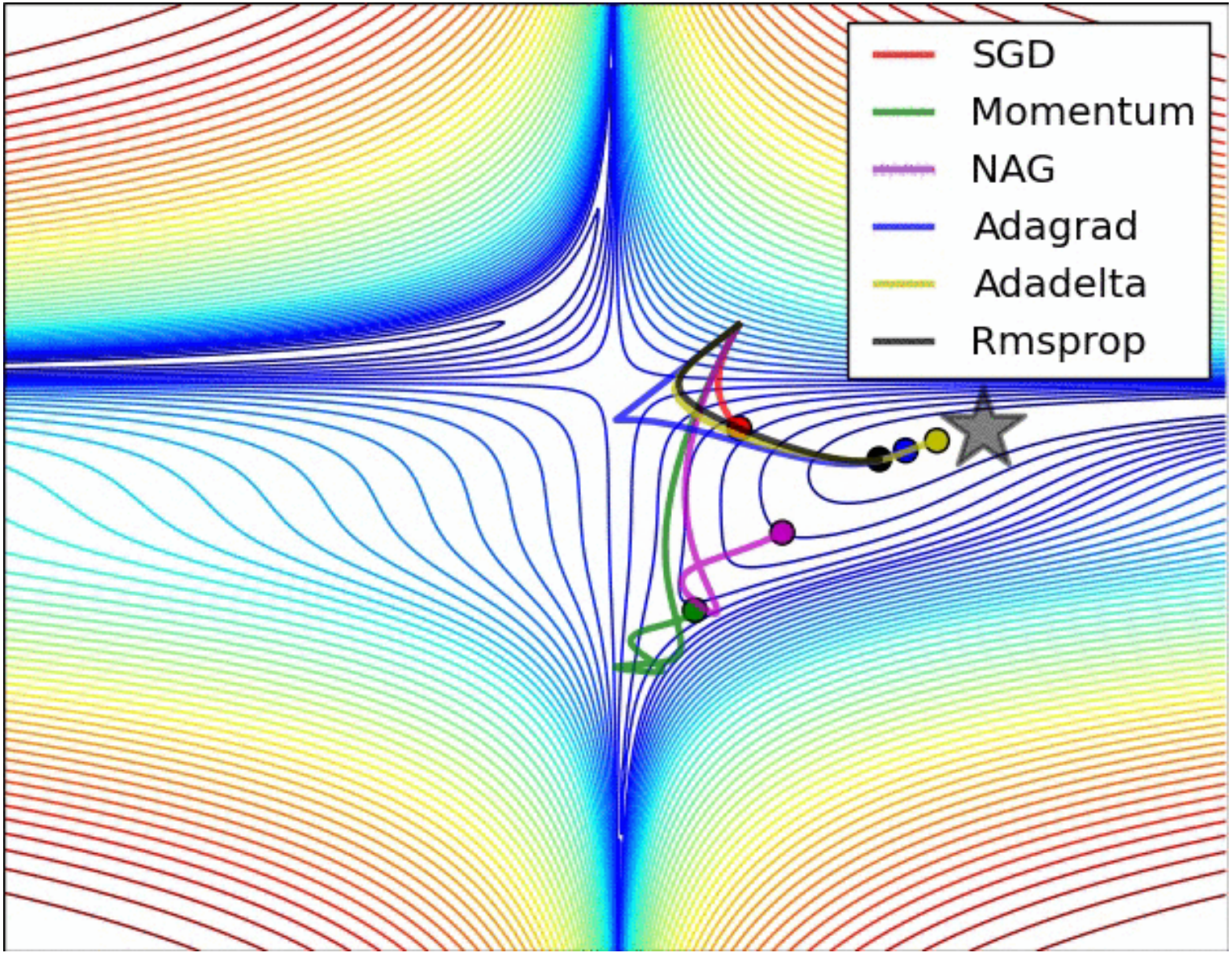
- So far: fixed learning rate for all parameters
- **Adaptive learning rate**: use per-parameter learning rate
  - Smaller learning rate for sensitive parameters (large derivative)
  - Larger learning rate for insensitive parameters (small derivative)
- Example: Adagrad (Duchi, 2011)

$$\boldsymbol{r}_t \leftarrow \boldsymbol{r}_{t-1} + \boldsymbol{g}_t \odot \boldsymbol{g}_t \qquad \text{(sum of sq. gradient)}$$

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}_t}} \odot \boldsymbol{g}_t \qquad \text{(rescale learning rate)}$$

  - Learning rate reduced over time, more so for parameters with larger derivatives
  - Good theoretical properties for convex functions
  - For deep learning, deduction can be too quick initially
- Adadelta and RMSProp (Hinton 2012) are variants that use exponentially-decaying moving average of sq. derivatives
- Requires additional memory to store $\boldsymbol{r}$ (same size as model parameters $\boldsymbol{\theta}$)

Example (click to animate)



Discussion

- Optimizers that use adaptive learning rates are popular
- Momentum and adaptive learning rates can also be combined
  - E.g., Adagrad or RMSProp with momentum, Adam, NAdam, AMSGrad, AdamX, AdamW, RAdam
  - Yet higher memory consumption (velocity and per-parameter LR)
- No best optimizer $\rightarrow$ hyperparameter
- In PyTorch, many optimizers provided
  - E.g., SGD and Adagrad (w/ and w/o momentum), Adam, ...
  - See https://pytorch.org/docs/stable/optim.html
  - Generally, called after backward pass
- More in lecture Optimization in Machine Learning by Simon Weissmann (lecture notes, slides)

Example: PyTorch

```
# define model with one hidden layer
model = torch.nn.Sequential(
    torch.nn.Linear(dim_in, dim_hidden),
    torch.nn.ReLU(),
    torch.nn.Linear(dim_hidden, dim_out),
)

# define loss function (mean squared error)
loss_fn = torch.nn.MSELoss()

# pick optimizer (Adam)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

# run for 500 epochs
for t in range(500):
    y_pred = model(X) # forward: model output (X = examples)
    loss = loss_fn(y_pred, y)  # forward: loss (y = labels)
    model.zero_grad()          # clear old gradients
    loss.backward()            # backward: compute gradients
    optimizer.step()           # update parameters
```

Deep Learning

03 – Gradient-Based Training

Part 3: Architecture Design

Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-2

Cost functions of deep models

Consider cost function $J(\boldsymbol{\theta})$ for some deep model:

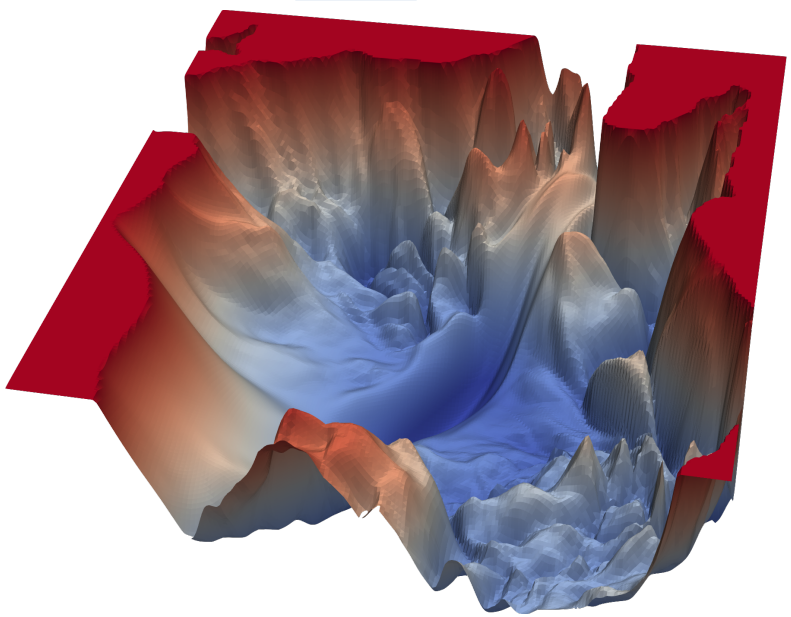
- **Non-convexity**
  - Can have a large number of local minima
  - Problematic if local minima have much higher cost than optimum
  - Can show: in many classes of random functions, this is unlikely when dimensionality is high
- **Non-identifiability**
  - Many different equivalent parameter choices (= one reason for the large number of local minima)
  - E.g., in any MLP layer, can swap two units and corresponding weights (**weight space symmetry**)
  - E.g., in ReLU unit, can scale input weights by c and "output weights" by $1/c$
- Many **saddle points**
  - Can show: in many classes of random functions, number of saddle points compared to local minima grows exponentially with dimensionality
  - May slow down learning

Come on, how bad can it be?

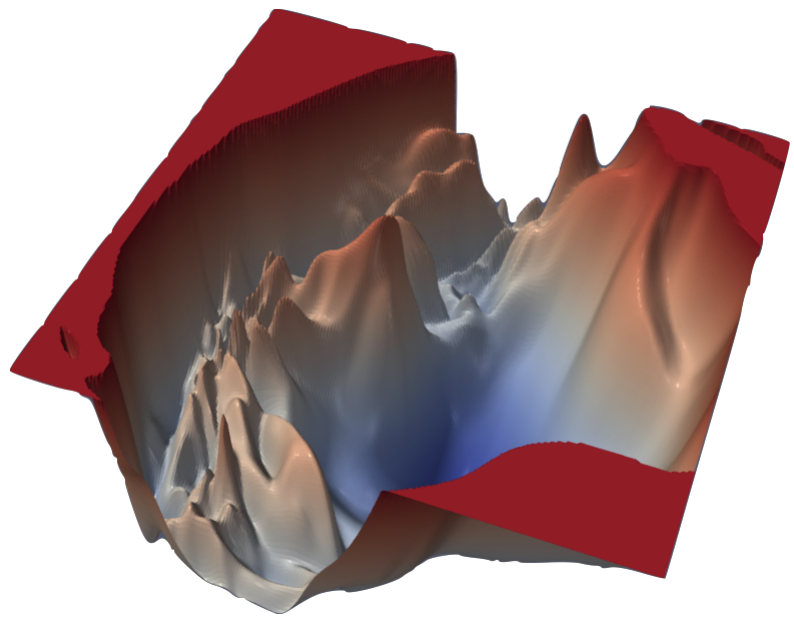
Visualization of cost landscape of by Li et al. (2018)

- Minimizer found during training in center
- Cost visualized along two random directions (in parameter space)
- More at losslandscape.com

VGG-56



VGG-110



(naming scheme: architecture – depth)

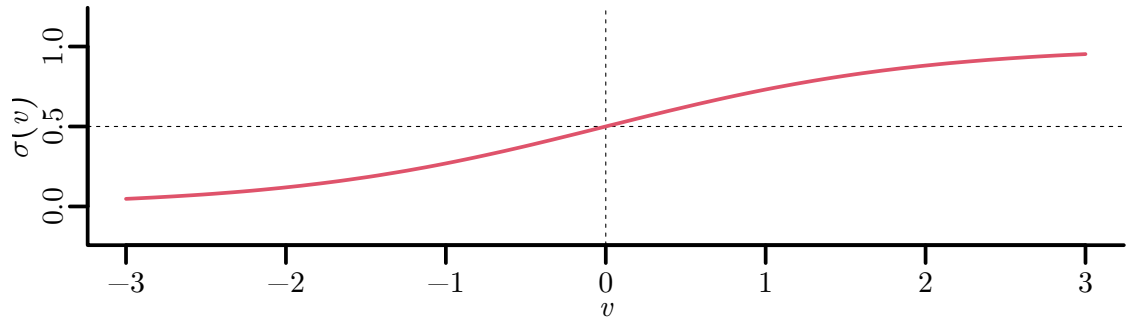
Vanishing gradient problem

- Consider an activation function ϕ and recall:

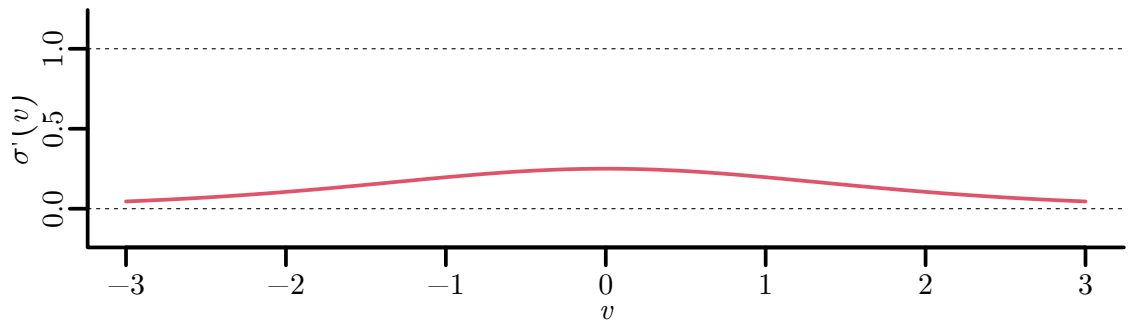
- **Vanishing gradient problem**
  - If $\phi'(v) < 1$, the gradient becomes smaller
  - If this happens in consecutive layers, **the gradient *vanishes* exponentially fast with depth** (since local gradients multiply)
  - If $\phi'(v) \approx 0$, gradient barely passes through (**saturated unit**)
  - If $\phi'(v) = 0$, gradient is gone right away (**dead unit**)
- Problematic since prior layers receive no useful gradient signal
  - E.g., consider a weight vector $\boldsymbol{w}$ used in a prior layer $\rightarrow \nabla_{\boldsymbol{w}} J \approx \mathbf{0} \rightarrow$ gradient-based learning fails (weight not updated)
  - E.g., consider input $\boldsymbol{x}$ and corresponding output $\hat{y}$ of a network $\rightarrow \nabla_{\boldsymbol{x}} \hat{y} \approx \mathbf{0} \rightarrow$ output insensitive to input changes
- Problem also arises with other layers (e.g., linear layers $\rightarrow$ exercise)

Example: Logistic activation function

- Forward: $\sigma(v) = (1 + \exp(-v))^{-1} \in [0, 1]$



- Backward: $\sigma'(v) = \sigma(v)(1 - \sigma(v)) \in [0, 0.25]$

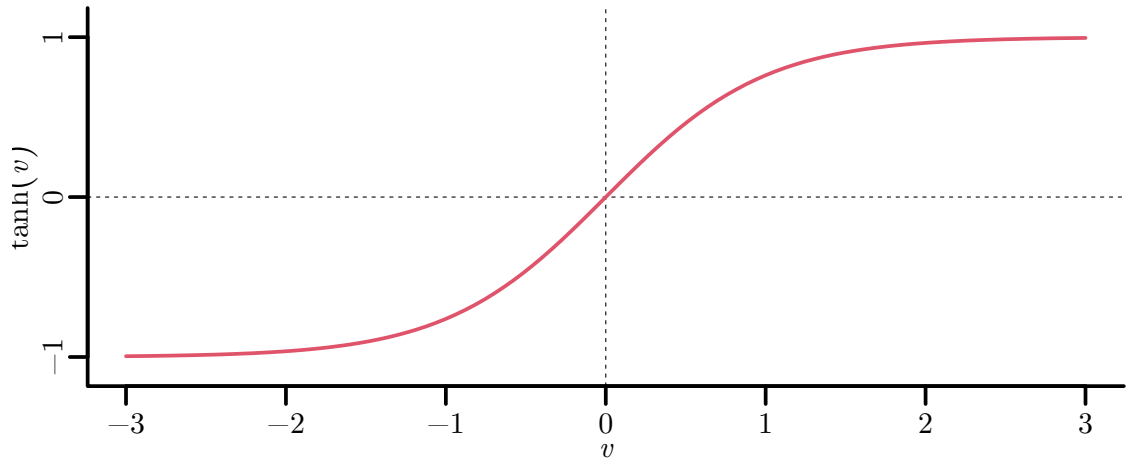


The logistic unit is not well suited as a hidden layer unit since:

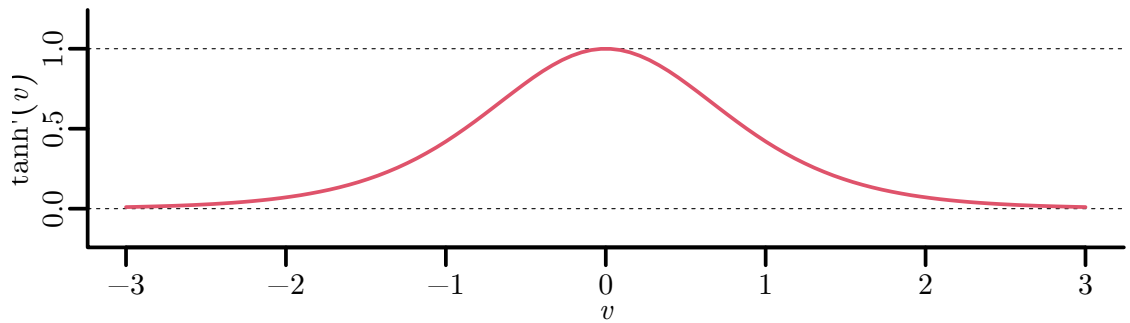
- $\sigma(0) = 0.5 \rightarrow$ zeros not "passed through"
- $\sigma'(v) \leq 0.25 \rightarrow$ gradients always reduce by at least $1/4$
- Saturated when $|v| \geq 5$

Better: tanh activation function

- Forward: $\tanh(v) = (\exp(2v) - 1)/(\exp(2v) + 1) \in [-1, 1]$



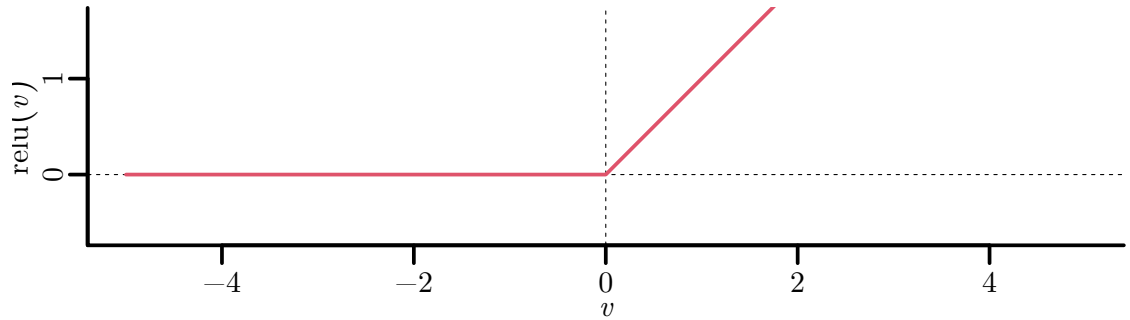
- Backward: $\tanh'(v) = 1 - \tanh(v)^2 \in [0, 1]$



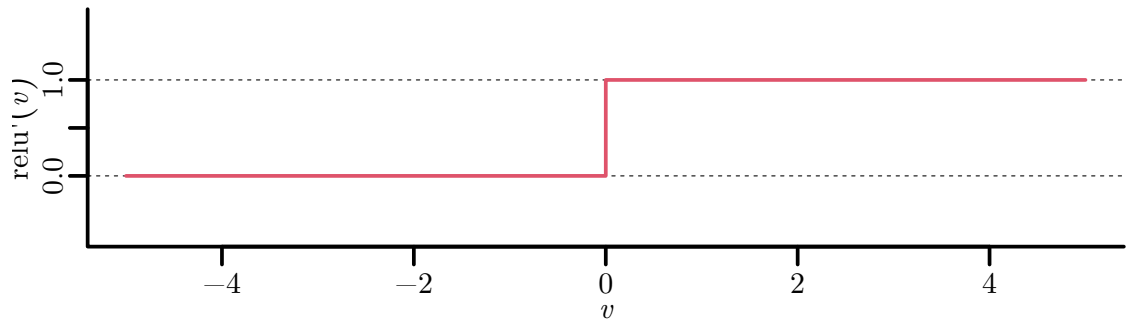
- $\tanh(0) = 0 \rightarrow$ zeros "passed through"
- $\tanh'(0) = 1 \rightarrow$ gradients around zero "passed through"
- Saturated when $|v| \geq 3$

Example: ReLU activation function

- Forward: $\mathrm{relu}(v) = \max(0, v) \in [0, \infty)$



- Backward: $\mathrm{relu}'(v) = \mathbb{I}(v > 0) \in \{0, 1\}$ for $v \neq 0$

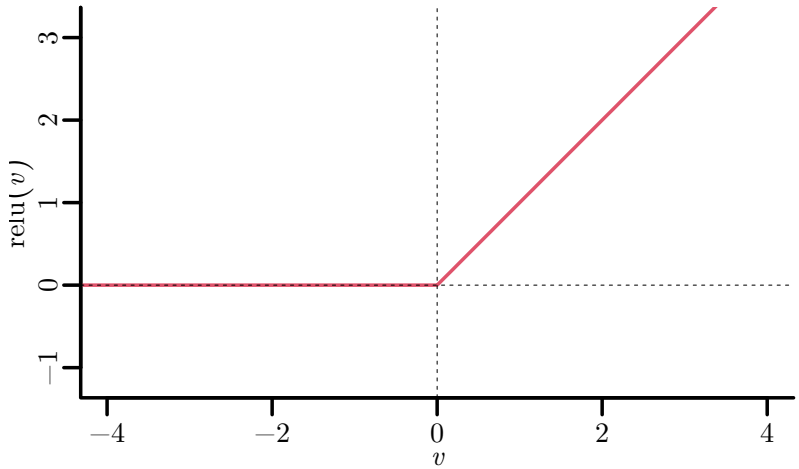


- $\mathrm{relu}(0) = 0 \rightarrow$ zeros "passed through"
- $\mathrm{relu}'(v > 0) = 1 \rightarrow$ gradients "passed through"
- $\mathrm{relu}'(v < 0) = 0 \rightarrow$ gradients gone right away (dead)

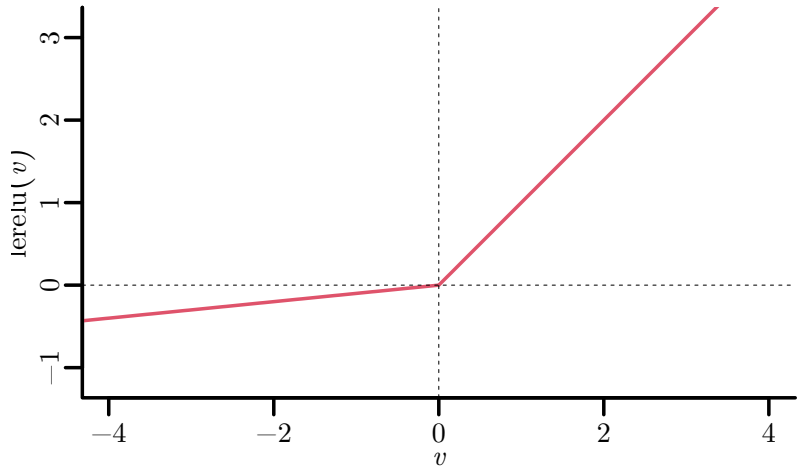
Related activation functions (forward)

The problematic behaviour of ReLU for negative values (esp. around 0) has motivated a number of related functions:

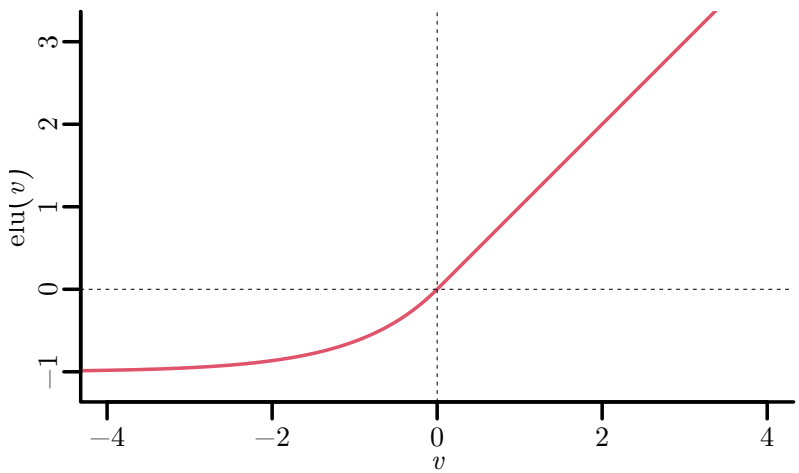
ReLU



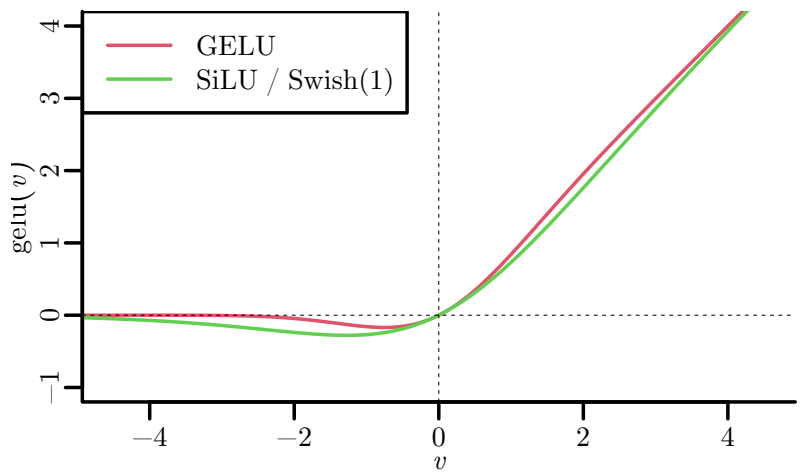
Leaky ReLU ($\alpha = 0.1$)



Exponential LU (ELU)



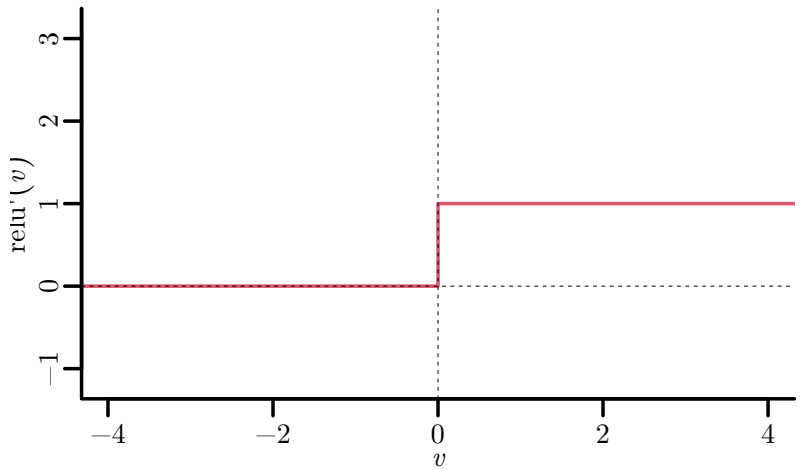
Gaussian Error LU (GELU) / Swish



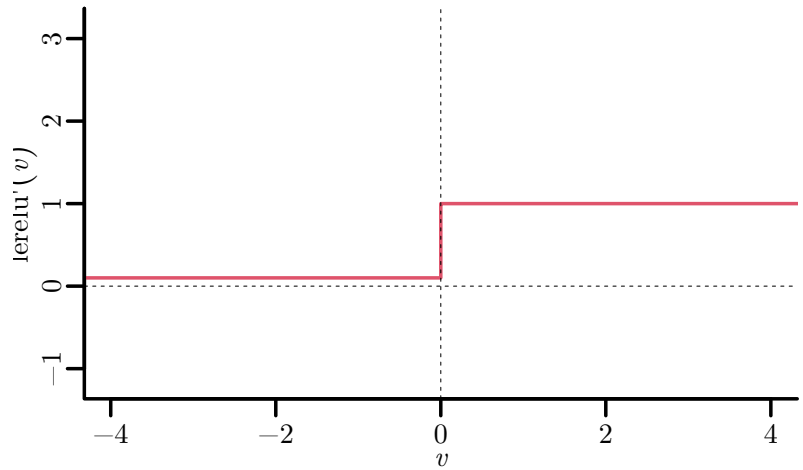
Related activation functions (backward)

The problematic behaviour of ReLU for negative values (esp. around 0) has motivated a number of related functions:

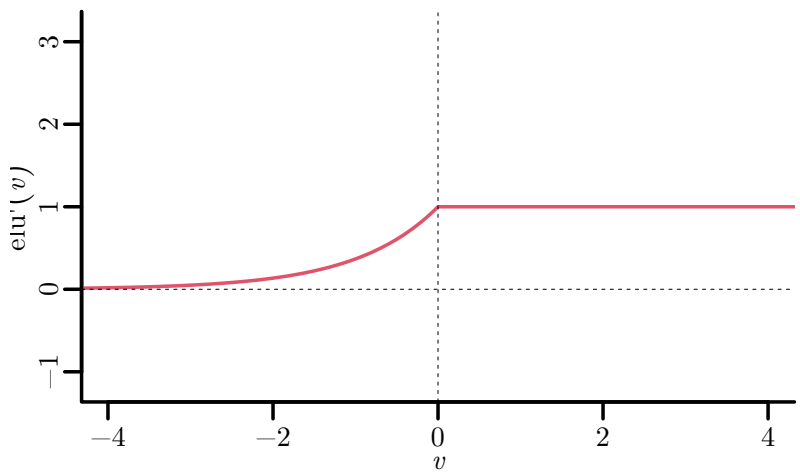
ReLU′



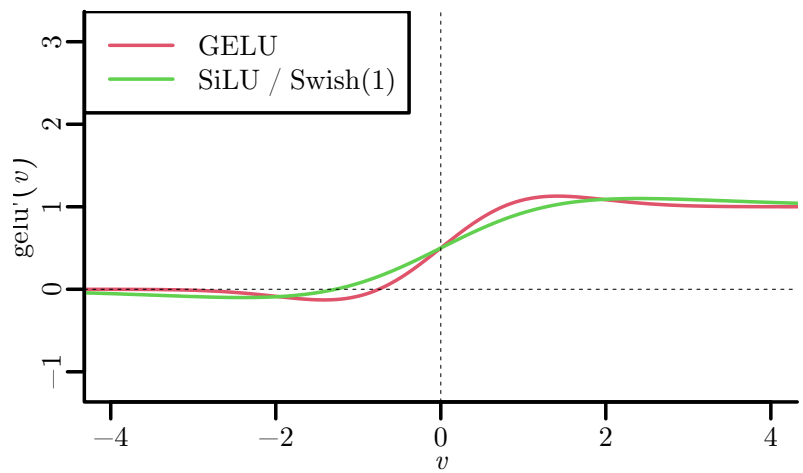
Leaky ReLU′ ($\alpha = 0.1$)



Exponential LU′ (ELU′)



Gaussian Error LU′ (GELU′) / Swish′



Discussion (activation functions)

Some desirable properties of activation functions include:

- Non-linearity $\rightarrow$ needed for expressiveness
- Differentiability $\rightarrow$ enables gradient-based learning
- Zeros pass through $\rightarrow$ avoids need to "learn" zero outputs
- Approximates linear unit around 0 $\rightarrow$ mitigates vanishing gradient
- Gradients bounded from above $\rightarrow$ stability
- Low computational cost $\rightarrow$ (Leaky) ReLU wins here
- As usual, choice depends
  - ReLU very common (e.g., MLPs, CNNs, transformers)
  - Variants of ReLU beneficial and also common
  - σ uncommon, but has special uses (e.g., as output unit, for gating)
  - tanh perhaps most traditional choice for MLPs
  - tanh also used for normalization purposes (squash a signal into $[-1, 1]$ range)

Results from Ramachandran et al. (2017)

Image classification (ImageNet 2012)

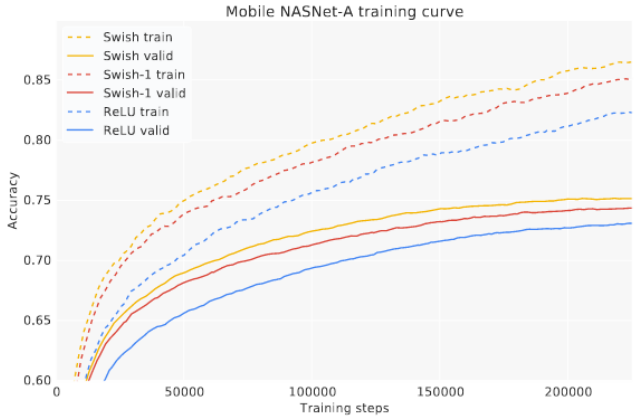


Figure 8: Training curves of Mobile NASNet-A on ImageNet. Best viewed in color

Model	Top-1 Acc. (%)			Top-5 Acc. (%)		
LReLU	73.8	73.9	74.2	91.6	91.9	91.9
PReLU	74.6	74.7	74.7	92.4	92.3	92.3
Softplus	74.0	74.2	74.2	91.6	91.8	91.9
ELU	74.1	74.2	74.2	91.8	91.8	91.8
SELU	73.6	73.7	73.7	91.6	91.7	91.7
GELU	74.6	-	-	92.0	-	-
ReLU	73.5	73.6	73.8	91.4	91.5	91.6
Swish-1	74.6	74.7	74.7	92.1	92.0	92.0
Swish	74.9	74.9	75.2	92.3	92.4	92.4

Table 6: Mobile NASNet-A on ImageNet, with 3 different runs ordered by top-1 accuracy. The additional 2 GELU experiments are still training at the time of submission.

Machine Translation (WMT 2014)

Model	newstest2013	newstest2014	newstest2015	newstest2016
LReLU	26.2	27.9	29.8	33.4
PReLU	26.3	27.7	29.7	33.1
Softplus	23.4	23.6	25.8	29.2
ELU	24.6	25.1	27.7	32.5
SELU	23.7	23.5	25.9	30.5
GELU	25.9	27.3	29.5	33.1
ReLU	26.1	27.8	29.8	33.3
Swish-1	26.2	28.0	30.1	34.0
Swish	26.5	27.6	30.0	33.1

Table 11: BLEU score of a 12 layer Transformer on WMT English→German.

Exploding gradient problem

- Consider an activation function ϕ and recall:

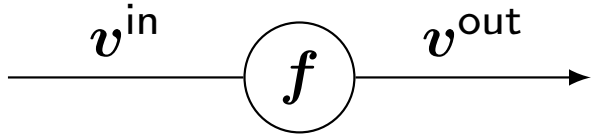
- **Exploding gradient problem**
  - If $\phi'(v) > 1$, the gradient becomes larger
  - If this happens in multiple consecutive layers, the gradient *explodes* exponentially fast (since local gradients multiply)
- Problematic since output extremely sensitive to prior layers
  - E.g., consider a weight vector $\boldsymbol{w}$ used in a prior layer $\rightarrow \nabla_{\boldsymbol{w}} J$ very large $\rightarrow$ gradient-based learning fails (weight diverges)
  - E.g., consider input $\boldsymbol{x}$ and corresponding output $\hat{y}$ of a network $\rightarrow \nabla_{\boldsymbol{x}} \hat{y}$ large, model predictions are unstable
- Usually does not occur due of the activation functions, but due to other layers (e.g., linear layers $\rightarrow$ exercise)

Architectural design patterns

- **Degradation problem**: performance of MLPs tends to deteriorate beyond a certain depth
  - Complicated optimization landscape
  - Gradients may vanish/explode exponentially fast with increasing depth in MLP
  - Can be mildened by choice of suitable activation function, but not by much
- Coming up: architectural design patterns
  - Key idea: **modify the network architecture to mitigate training challenges** $\rightarrow$ better empirical performance
- Goals include
  - Mitigate the vanishing gradient problem and, more generally, improve gradient flow through network
  - Facilitate training and effectiveness of (very) deep networks, i.e., mitigate the degradation problem
  - Simplify the optimization landscape

Representation view

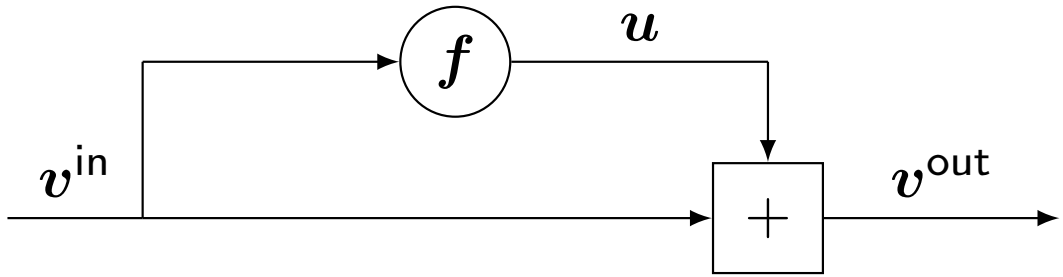
- Consider a parameterized layer $\boldsymbol{f} : \mathbb{R}^Z \to \mathbb{R}^Z$ somewhere in an FNN



  - E.g., $\boldsymbol{v}^{\text{out}} = \boldsymbol{f}(\boldsymbol{v}^{\text{in}}) = \phi(\boldsymbol{W}^\top \boldsymbol{v}^{\text{in}} + \boldsymbol{b})$
  - Note: layer parameters not explicitly shown
- **Representation view**
  - $\boldsymbol{v}^{\text{in}}$ is representation of input obtained from previous layer
  - $\boldsymbol{v}^{\text{out}}$ is updated representation of input for next layer
  - A "step" in a multi-step computation
  - How many steps? $\approx$ network depth (L)
  - How much memory? $\approx$ network width (Z)
- Observe: If $\boldsymbol{v}^{\text{in}}$ is already a good representation, model needs to learn to preserve it
  - E.g., with linear units $(\phi(x) = x)$, must learn $\boldsymbol{W} = \boldsymbol{I}$ and $\boldsymbol{b} = \boldsymbol{0}$
  - Generally, with non-linear units, more involved $\to$ exercise
- The deeper the network is, the more this matters
  $\to$ One reason for the degradation problem

Residual connections (forward)

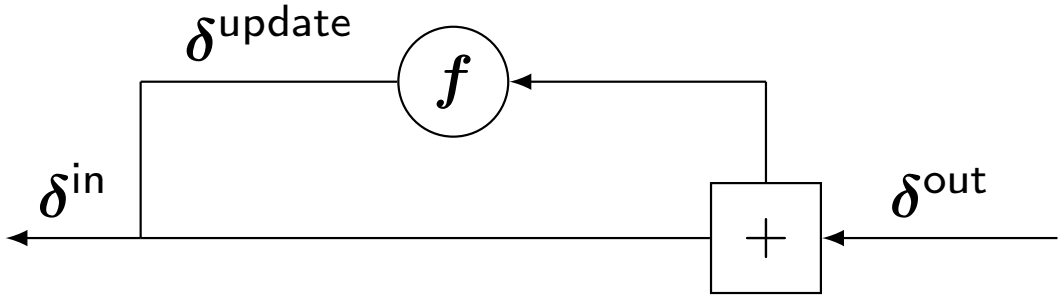
- **Residual connections** (He at al. 2015/2016) change the layer:



  - I.e., $\boldsymbol{v}^{\text{out}} = \boldsymbol{v}^{\text{in}} + \boldsymbol{f}(\boldsymbol{v}^{\text{in}})$
  - Think of $\boldsymbol{u} = \boldsymbol{f}(\boldsymbol{v}^{\text{in}})$ as an **update** or **residual**
- Recall: If $\boldsymbol{v}^{\text{in}}$ is already a good representation, model needs to learn to preserve it
  - Now "easy" to do: need to learn a zero update $(\boldsymbol{u} = \boldsymbol{0})$
  - Usually obtained by zeroing out all layer parameters
  - E.g., when $\boldsymbol{f} = \phi(\boldsymbol{W}_l^\top \boldsymbol{v}^{\text{in}} + \boldsymbol{b}_l)$ and $\phi(0) = 0$
  - Lowest regularization penalty (e.g., with ℓ_p regularization)
  - Likewise, "easy" to perform smaller updates
- Reduced degradation of performance due to "too many" layers
  - Easy for the network to learn identity mappings (all weights 0)
  - Easy for network to not use "unnecessary" layers

Residual connections (backward)

- Let's look at the backward pass



- We obtain

$$\delta^{\text{in}} = \delta^{\text{out}} + \delta^{\text{update}}$$

  - During backward pass, gradient is "updated" as well

- After L such layers, gradient is

$$\delta^{\text{in}} = \delta^{\text{out}} + \delta_L^{\text{update}} + \cdots + \delta_1^{\text{update}}$$

- Original gradient "passes through" $\rightarrow$ addresses vanishing gradient problem

Using residual connections

- Input and output dimensionality of $\boldsymbol{f}$ must match
  - But can use different dimensionality "in between"
  - Common choice: $\boldsymbol{f} = \boldsymbol{W}_2^\top \phi(\boldsymbol{W}_1^\top \boldsymbol{v}^{\text{in}})$
  - E.g., T5-Base: 768D in $\rightarrow$ project up to 3072D $\rightarrow$ activation $\rightarrow$ project back down to 768D
- We'll see residual connections through the lecture; e.g.,
  - In recurrent neural networks: e.g., LSTM or GRU units
  - In convolutional neural networks: e.g., ResNet
  - In sequence models: e.g., Transformers

Example: ResNet on CIFAR-10 (as of 2015)

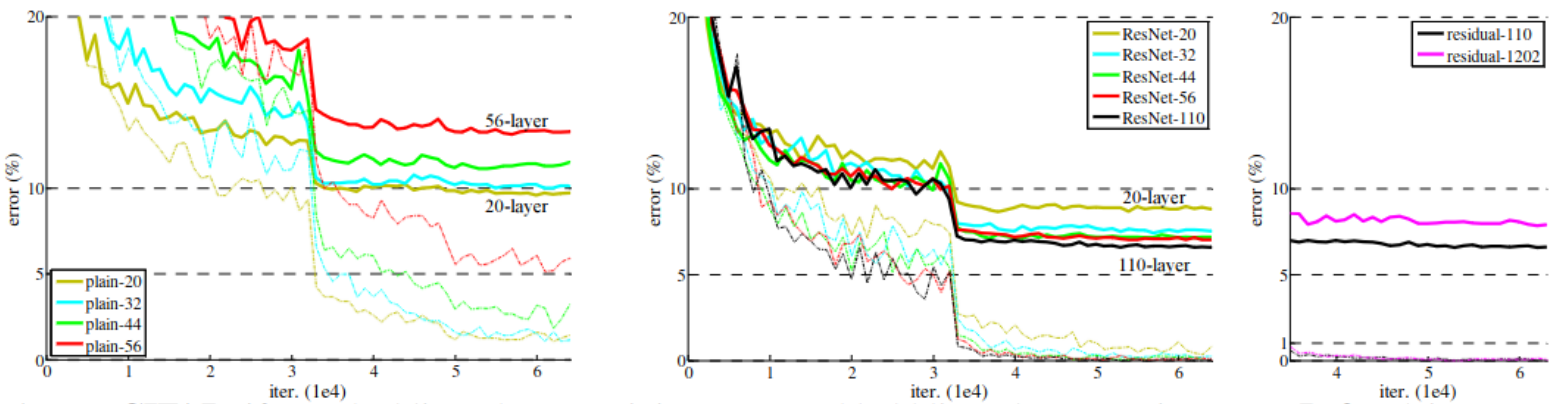
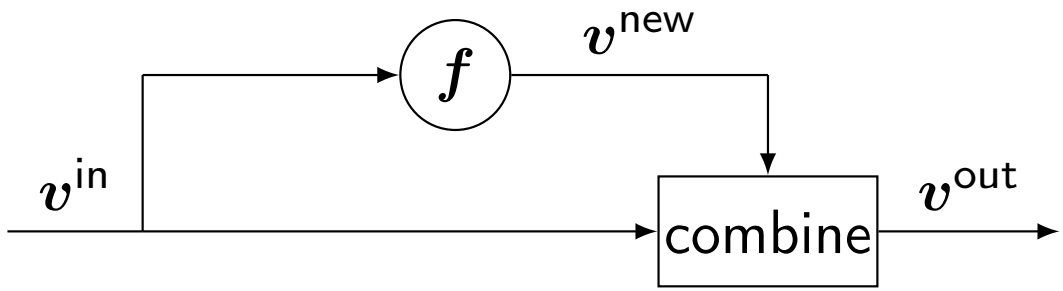


Figure 6. Training on **CIFAR-10**. Dashed lines denote training error, and bold lines denote testing error. **Left**: plain networks. The error of plain-110 is higher than 60% and not displayed. **Middle**: ResNets. **Right**: ResNets with 110 and 1202 layers.

Skip connections

- More generally, **skip connections** (also: *shortcut connections*) skip one or more layers



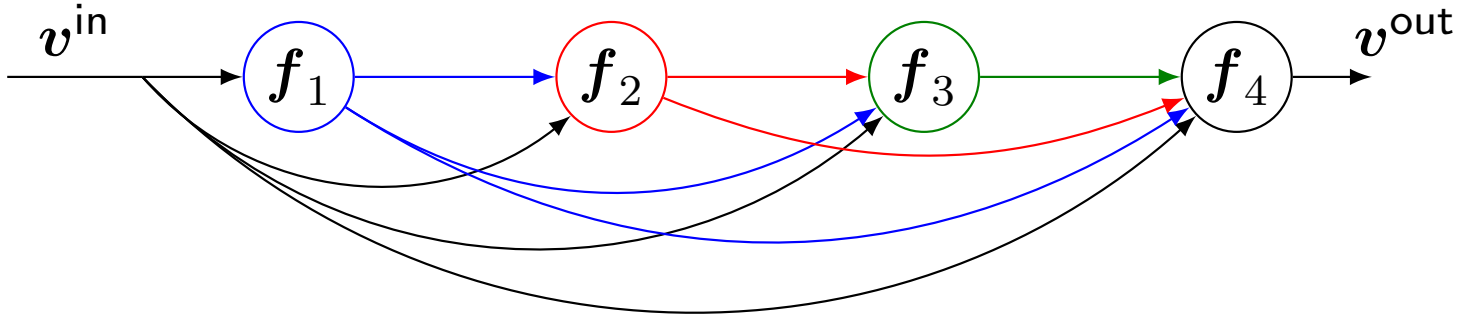
  - Here we "skip" over a single layer $\boldsymbol{f}$
  - Original representation $\boldsymbol{v}^{\text{in}}$ and the new representation $\boldsymbol{v}^{\text{new}}$ are combined

- For residual blocks, we combine by adding
  - We can then think of $\boldsymbol{v}^{\text{new}}$ as an "update"

- Other common options include: concatenate (now); average, max-pooling, attention (later)

Concatenating skip connections (1)

- One option is to concatenate representations
- Extreme case: concatenate *all* representations of previous layers



- Preservation of information across layers trivial
  - Since directly provided as additional input
  - Each layer merely "enhances" the current representation (increases effective dimensionality)
- Suppose each layer produces Z-dimensional output and is parameterized by one weight matrix
  - Layer l has lZ inputs $\rightarrow$ increases linearly with L
  - Weight matrices have form $lZ \times Z \rightarrow$ increase linearly with L
  - Total number of weights is $Z^2 \cdot L(L+1)/2$
    $\rightarrow$ increases quadratically with L

Concatenating skip connections (2)

- Examples: DenseNet (images), JK-Net (graphs)
- Interestingly, concatenating skip connections can reduce cost

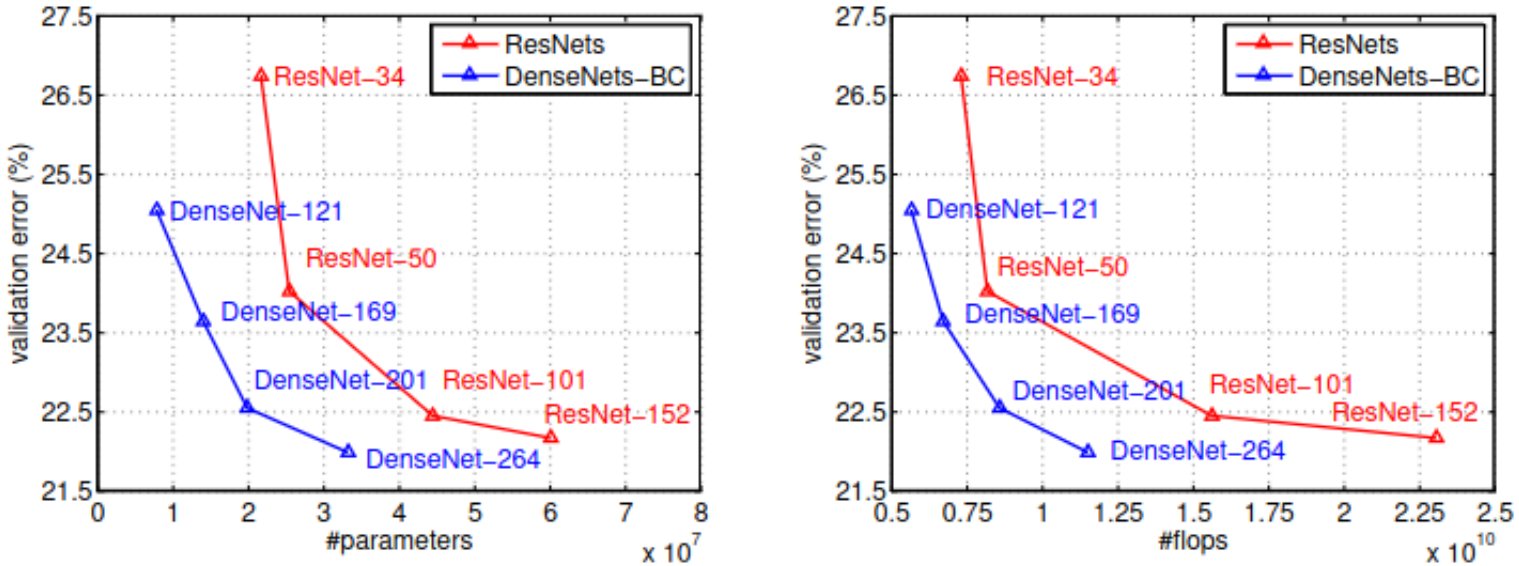
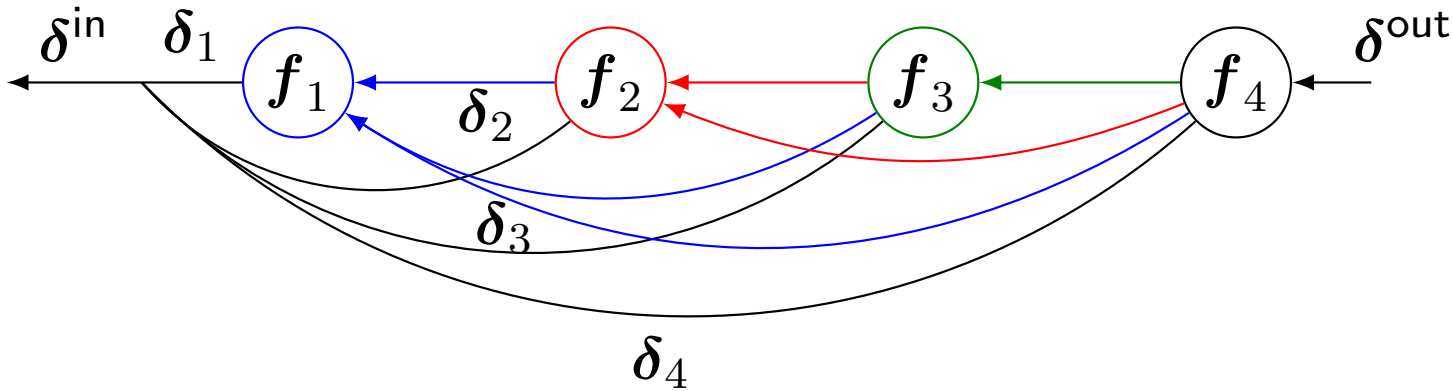


Figure 3: Comparison of the DenseNets and ResNets top-1 error rates (single-crop testing) on the ImageNet validation dataset as a function of learned parameters (*left*) and FLOPs during test-time (*right*).

- Why? Can get away with much smaller per-layer output size (Z)

Concatenating skip connections (backward)

- Let's look at the backward pass



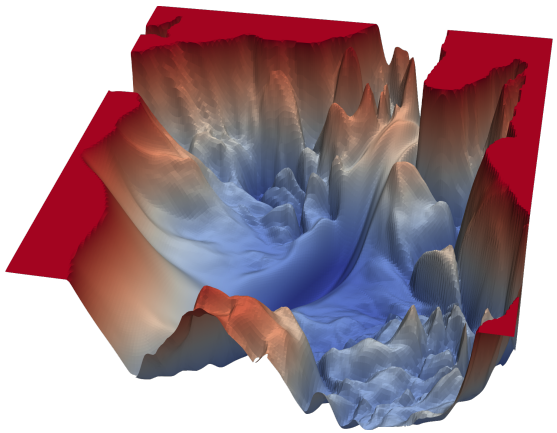
- After L such layers, gradient is

$$\boldsymbol{\delta}^{\text{in}} = \boldsymbol{\delta}_L + \cdots + \boldsymbol{\delta}_1$$

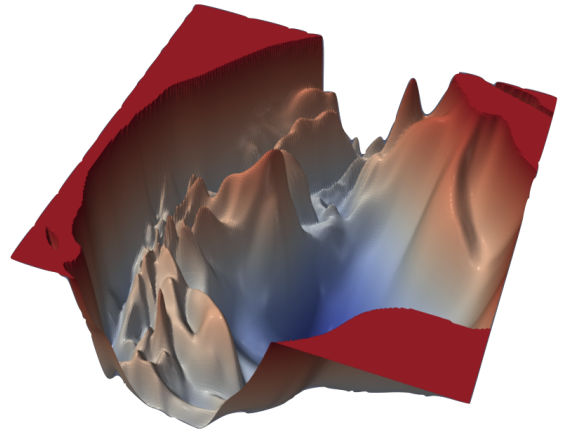
  - Again, gradient from later layers is directly passed through

Impact on optimization landscape

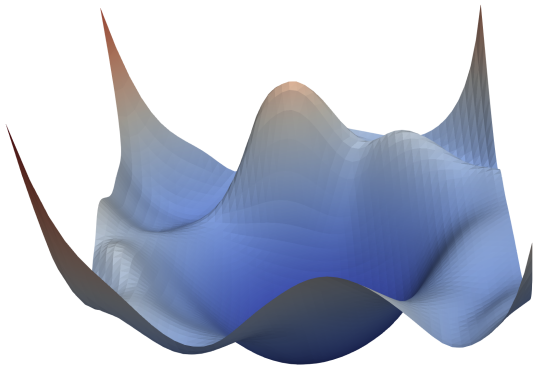
VGG-56



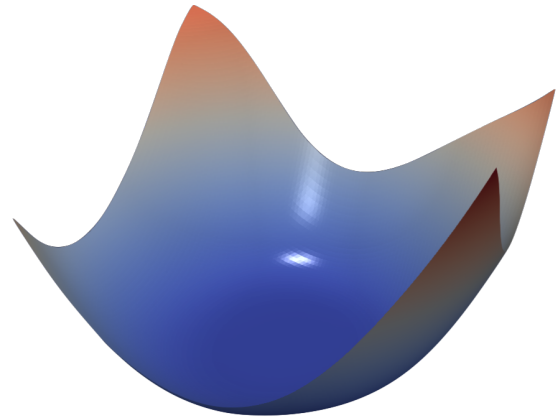
VGG-110



ResNet-56
(residual)



DenseNet-110
(skip concat)



Li et al. (2018)

Batch normalization

- Batch normalization (BN) is a layer that mitigates two problems
  1. **Covariate shift**: when parameters of one layer change, the (training distribution of) inputs to the next layer changes too
     $\rightarrow$ ignored by gradient-based methods
  2. Gradient magnitudes may vary wildly across layers
     $\rightarrow$ complicates gradient-based learning
- During training, BN normalizes each of its input features to zero mean and unit variance *within each batch*
  - I.e., input $\boldsymbol{z} \in \mathbb{R}^Z$ is normalized to output $\tilde{\boldsymbol{z}} = (\boldsymbol{z} - \boldsymbol{\mu})/\sqrt{\boldsymbol{\sigma}^2}$, where $\boldsymbol{\mu}, \boldsymbol{\sigma}^2 \in \mathbb{R}^Z$ are computed from the entire batch
  - Important: this normalization is part of the "action" of the layer and the gradient is backpropagated through these operations
  - Variant: normalize to mean $\boldsymbol{\beta}$ and variance $\boldsymbol{\gamma}$, where $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ are learned parameters
- At test time, use running average of $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ from training
- Strong empirical performance

Layer normalization

- Batch normalization sometimes problemantic
  - During training, batch size must be sufficiently large (e.g., no online learning possible)
  - When number of layers is not fixed but depends on input (e.g., RNN, Transformer), number of required mean/variance statistics varies with input lengths
- Layer normalization (LN) is a simple alternative
  - BN: normalize each input feature across the mini-batch (columns)
  - LN: normalize each input vector individually (rows)
  - For input $\boldsymbol{z}$, set

$$\mathrm{norm}(\boldsymbol{z}) = \frac{\boldsymbol{z} - \mathrm{mean}(\boldsymbol{z})}{\sqrt{\mathrm{var}(\boldsymbol{z})}},$$

  where mean/variance are computed across the elements of $\boldsymbol{z}$
- Normalization methods differ in their invariance properties

	Weight matrix re-scaling	Weight matrix re-centering	Weight vector re-scaling	Dataset re-scaling	Dataset re-centering	Single training case re-scaling
Batch norm	Invariant	No	Invariant	Invariant	Invariant	No
Weight norm	Invariant	No	Invariant	No	No	No
Layer norm	Invariant	Invariant	No	Invariant	No	Invariant

Table 1: Invariance properties under the normalization methods.

Deep Learning

03 – Gradient-Based Training

Part 4: Initialization

Prof. Dr. Rainer Gemulla

Universität Mannheim

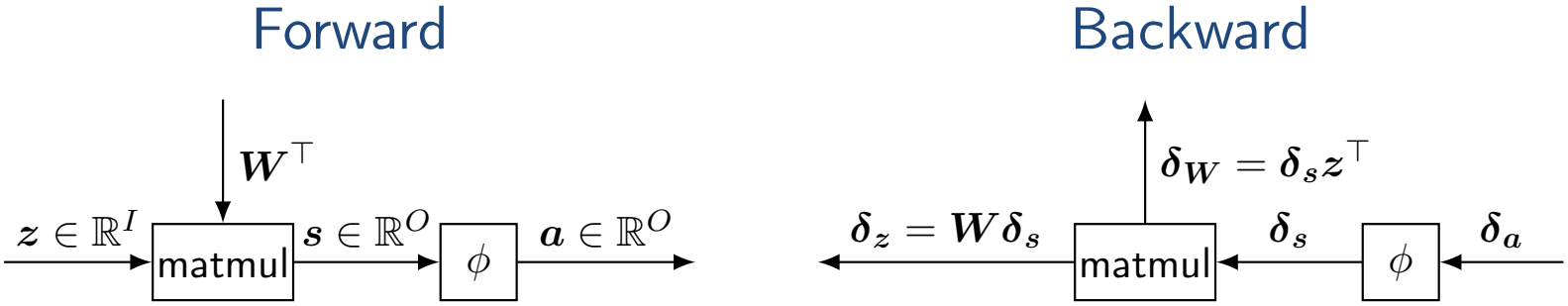
Version: 2025-1

Initialization

- **Initialization** = choose starting value for all parameters
- Important since
  - Affects solutions found by gradient-based optimizers
  - Affects performance of gradient-based optimizers
- Suitable choice generally depends on archiceture
- Generally, in standard MLP, initializing weight matrices with
  - Too small values $\rightarrow$ vanishing gradient
  - Too large values $\rightarrow$ exploding gradient
  - Constant values $\rightarrow$ bad solution

Zero initialization

- Consider a standard MLP (no skip conn.) with layers of form



  where $\boldsymbol{W} \in \mathbb{R}^{I \times O}$

- **Zero initialization** ($\boldsymbol{W} = \boldsymbol{0}$) is problematic
  - If $\boldsymbol{W} = \boldsymbol{0}$, so is $\boldsymbol{\delta}_{\boldsymbol{z}}$
  - If $\boldsymbol{\delta}_{\boldsymbol{a}} = c\boldsymbol{1}$, all weight vectors (rows of $\boldsymbol{W}$) receive same gradients
  - E.g., happens when *all* weight matrices are zero-initialized
  - Consequence: all units at each layer always have the same output (**co-adaptation**) $\rightarrow$ learning fails

- Likewise: initialization with constant problematic

Normal initialization (1)

- **Normal initialization**: $\boldsymbol{W}^{I \times O}$ initialized using iid. samples from a normal distribution $\mathcal{N}(0, \sigma^2)$
- Analysis (forward)
  - Suppose that $\boldsymbol{z} \in \mathbb{R}^I \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ has standard normal distribution; then

$$E[z_k] = 0 \qquad \text{and} \qquad \text{var}[z_k] = 1$$

  - Let $\boldsymbol{s} = \boldsymbol{W}^\top \boldsymbol{z} \in \mathbb{R}^O$; then

$$E[s_k] = 0 \qquad \text{and} \qquad \text{var}[s_k] = I\sigma^2$$

  - **Variance increases with increased input dimensionality**
  - Depending on ϕ can lead to vanishing/exploding gradients
- Solution: initialize with samples from $\mathcal{N}(0, \sigma^2/I)$
  $\rightarrow$ variance retained

Normal initialization (2)

- Analysis (backward)
  - Suppose that $\boldsymbol{\delta_s} \in \mathbb{R}^O \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ has standard normal distribution; then

$$E[\delta_{s_k}] = 0 \qquad \text{and} \qquad \text{var}[\delta_{s_k}] = 1$$

  - Let $\boldsymbol{\delta_z} = \boldsymbol{W}\boldsymbol{\delta_s} \in \mathbb{R}^I$; then

$$E[\delta_{z_k}] = 0 \qquad \text{and} \qquad \text{var}[\delta_{z_k}] = O\sigma^2$$

  - **Variance increases with increased output dimensionality**
  - Depending on ϕ can lead to exploding gradients
- Solution: initialize with samples from $\mathcal{N}(0, \sigma^2/O) \rightarrow$ variance retained

Xavier/Kaiming initialization

- Generally, for MLPs, vanishing/exploding gradients mitigated at the start when variances retained across layers
  - Forward (use σ^2/I) and backward pass (use σ^2/O) affected differently
  - Need compromise
- For $\phi = \tanh$ or linear: **Xavier initialization**
  - Also: *Glorot initialization*
  - Samples from

$$\mathcal{N}(0, 1/D) \qquad \text{or} \qquad \mathcal{U}(-\sqrt{3/D}, \sqrt{3/D}),$$

    where $D = (I + O)/2$
- For other activation functions, can multiply by **gain**
  - E.g., for relu: $\sqrt{2}$ (**Kaiming initialization**, also *He initializiation*)

Discussion

- Initialization generally important
- Suitable scale matters for standard MLPs and depends on dimensionality
- Different for other architectures; e.g.,
  - Scaling less influential when layer/batch/weight normalization is used
  - For residual layers, typically want weight matrix to be small $\rightarrow$ small update / initially close to identity
  - For ResNet (vision), may use Fixup normalization

> **Fixup initialization (or: How to train a deep residual network without normalization)**
>
> 1. Initialize the classification layer and the last layer of each residual branch to 0.
> 2. Initialize every other layer using a standard method (e.g., He et al. (2015)), and scale only the weight layers inside residual branches by $L^{-\frac{1}{2m-2}}$.
> 3. Add a scalar multiplier (initialized at 1) in every branch and a scalar bias (initialized at 0) before each convolution, linear, and element-wise activation layer.

- As usual: it depends $\rightarrow$ experience, hyperparameter search, ...

Deep Learning

04 – Layers for Categorical Data

Part 0: Overview

Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-1

One-hot encoding

- How can we handle categorical inputs and outputs in FNNs?
- Recall: **one-hot encoding** for categorical inputs
  - Encode with binary vector with one element per category
  - Element that corresponds to actual value set to 1; rest 0
  - Example: $x \in \{\,\text{red}, \text{green}, \text{blue}\,\}$
  - Then $x = \text{green}$ becomes $\boldsymbol{x} = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}^T$
- So just one-hot encode? Usually not explicitly since
  - Not efficient (increased compute cost)
  - Limited parameter sharing (decreased quality)
- How to do better? $\rightarrow$ this lecture

Outline (Layers for Categorical Data)

0. Overview
1. Embedding and softmax layers
2. Word vectors (example)
3. Softmax with many classes

Summary

- Embedding layers for categorical data
  - Standard approach to handle categorical data (e.g., PyTorch Embedding)
  - Directly store embeddings for each category
  - Low-dimensional, continuous representations
- Word vectors
  - Simple approach to obtain word representations
  - Outdated but instructive
  - CBOW: trained such that words can be reconstructed from average word vectors of neighboring words
- Softmax with many classes is expensive
  - Often prohibitively so $\rightarrow$ many approaches
  - For training: e.g., hierarchical softmax, negative sampling
  - For prediction: e.g., approximate maximum inner product search

Deep Learning

04 – Layers for Categorical Data

Part 1: Embedding and Softmax Layers

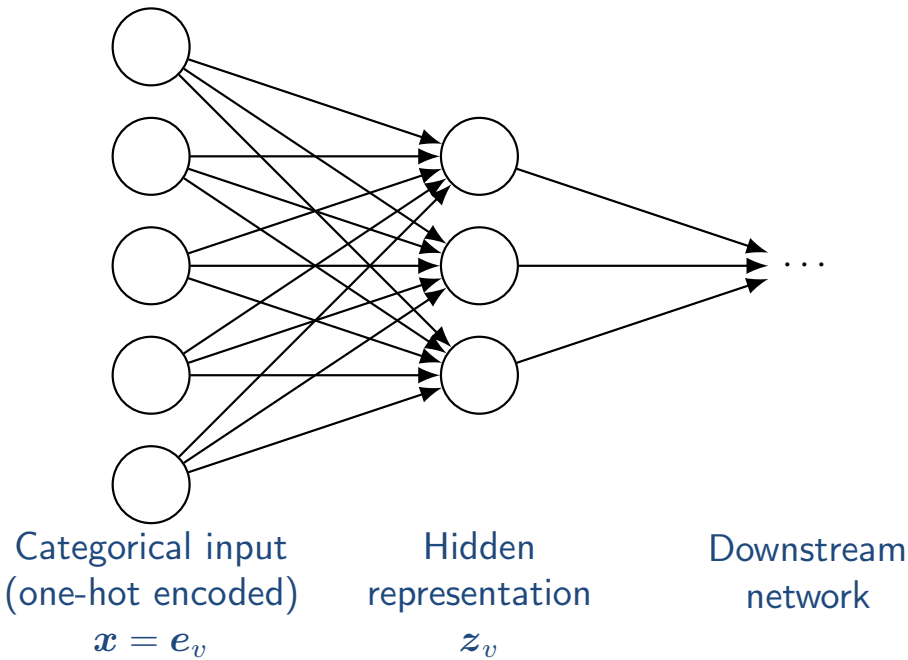
Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-4

Embedding layer (1)

- Consider an FNN with a single categorical input
  - **Vocabulary** $\mathcal{V} = \{1, \ldots, V\}$ = set of categories
  - Input $v \in \mathcal{V}$, one-hot encoded to $\boldsymbol{e}_v \in \{0, 1\}^V$ (where $\boldsymbol{e}_v$ is v-th standard basis vector)
  - Fully-connected first layer with Z hidden units



- Downstream network sees $\boldsymbol{z}_v$, but not v or $\boldsymbol{e}_v$
  - Input v represented by $\boldsymbol{z}_v$
  - What can we say about $\boldsymbol{z}_v$?

Embedding layer (2)

- Action of first layer
  - Weight matrix $\boldsymbol{W} \in \mathbb{R}^{V \times Z} = \begin{pmatrix} \boldsymbol{w}_1 & \boldsymbol{w}_2 & \cdots & \boldsymbol{w}_Z \end{pmatrix} = \begin{pmatrix} \boldsymbol{o}_1^\top \\ \boldsymbol{o}_2^\top \\ \vdots \\ \boldsymbol{o}_V^\top \end{pmatrix}$
  - Bias vector $\boldsymbol{b} \in \mathbb{R}^Z$
  - Activation function ϕ
  - Layer output

$$v \in \mathcal{V} \rightarrow \boldsymbol{z}_v \in \mathbb{R}^Z = \phi(\boldsymbol{W}^T \boldsymbol{e}_v + \boldsymbol{b}) = \phi(\boldsymbol{o}_v + \boldsymbol{b}) \stackrel{\text{def}}{=} \text{emb}(v)$$

- An **embedding layer** directly stores embeddings (no computation)
  - Maps each category $v \in \mathcal{V}$ to a vector $\text{emb}(v) \in \mathbb{R}^Z$, the **embedding** of v
  - Parameterized by an $V \times Z$ **embedding matrix** $\boldsymbol{E}$
  - Given categorical input v, outputs v-th row of $\boldsymbol{E}$: $\text{emb}(v) = \boldsymbol{E}_{v:}^\top$
  - This is more efficient than modelling via fully-connected layer
  - Variant: normalize embeddings (e.g., to unit norm)

- Examples: categories in tabular data, tokens in NLP, vertices in graphs

Discussion

- Embedding layers may use large vocabularies ($V \gg Z$)
  - Canonical example: words or tokens of text data (V is tens of thousands, Z is a few hundreds)
  - Input $\boldsymbol{e}_v$ is sparse, discrete, high-dimensional
  - Embedding $\mathrm{emb}(v)$ is dense, continuous, low-dimensional
  - Layer performs dimensionality reduction / compression
- Many parameters for large vocabularies
  - Glove word vectors: $2\text{M} \times 300 \approx 2300\,\text{MB}$
  - LLaMa-7B language model: $32\text{k} \times 4096 \approx 512\,\text{MB}$
  - ComplEx embeddings on Wikidata-5M: $5\text{M} \times 128 \approx 2400\,\text{MB}$
  - In some models, significant fraction of parameters resides in initial embedding layer
- When multiple inputs use the same categories, embedding layer is typically shared
  - E.g., all of the examples above (for words, tokens, entities, resp.)
  - Example of **parameter sharing** across layers (more later)
  - Note: that's different from one-hot encoding each input

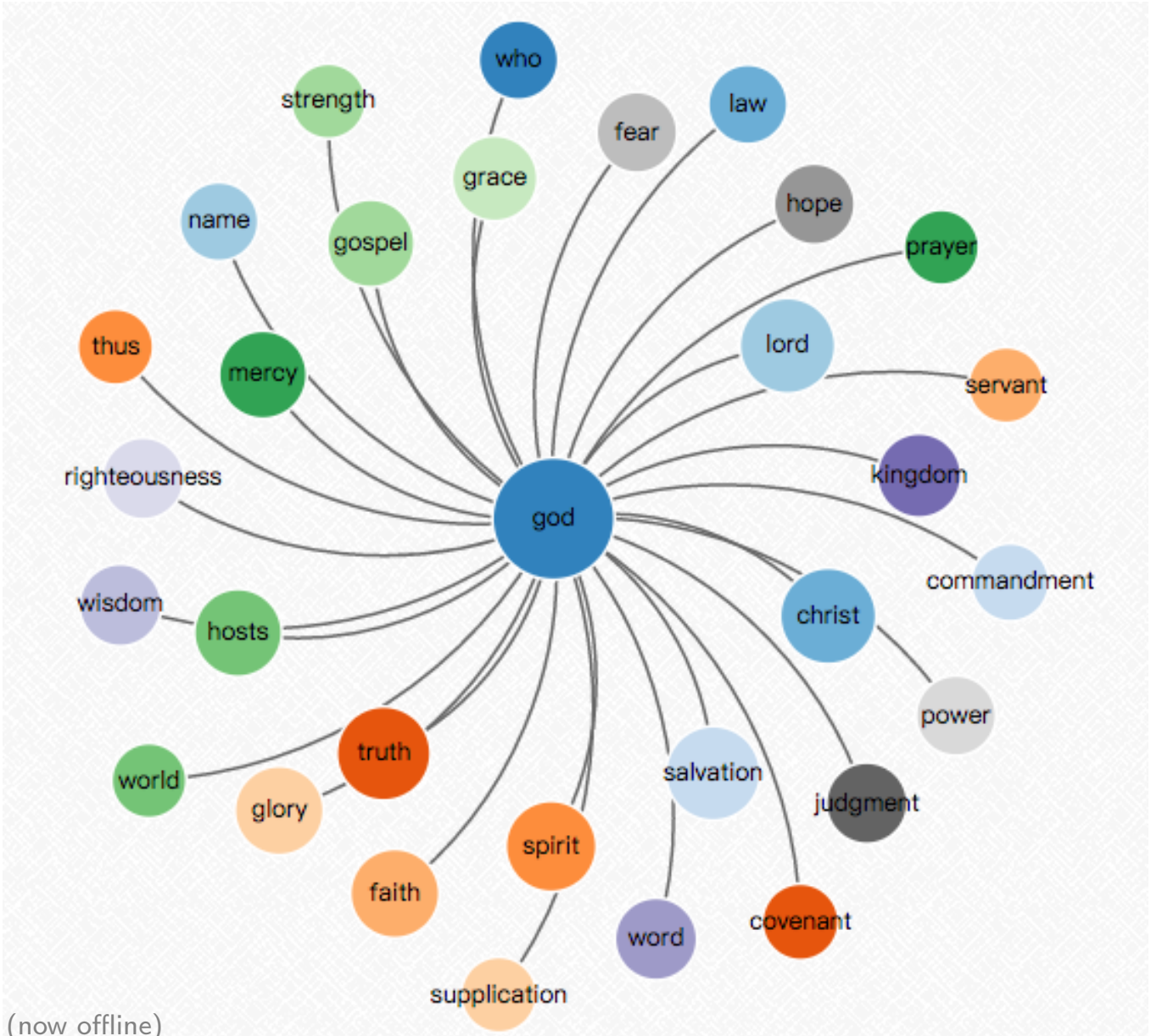
Similarity

- Consider two categories $v_1 \neq v_2$
  - Cosine similarity between $\boldsymbol{e}_{v_1}$ and $\boldsymbol{e}_{v_2}$ is 0
  - Cosine similarity between $\mathrm{emb}(v_1)$ and $\mathrm{emb}(v_2)$ generally $\neq 0$
  - When > 0: categories similar in embedding space
  - When < 0: categories dissimilar in embedding space
  - **Embeddings expose similarities**

- Intuitively, two categories should be similar in embedding space when they have similar impact on final output
  - Directly encouraged when training embedding layer as part of FNN
  - Indirectly (i.e., hopefully) when embedding layer is pretrained on other data (e.g., word vectors)

Example: Word Vectors

Top-30 closest word2vec vectors to "God", trained on the Bible



textminingonline.com (now offline)

Embeddings as representations

- Recall from 02-1: DL as an approach to *learn* features
  - Input objects $x \in \mathcal{X}$ are transformed into dense, continuous, low-dimensional representations called **embeddings** $\boldsymbol{z} \in \mathbb{R}^Z$
  - $Z =$ **embedding dimensionality**
  - Useful to represent complex objects (categorical data, textual data, graph data, tabular data, images, . . .)
  - Think: complex to work with objects, simple to work with embeddings
  - Useful **embedding space** = goal of **representation learning**

- **Embedding layers** used for categorical "non-divisible" objects
  - Directly learn embedding for each category

- For "divisible" objects, use an **encoder**
  - Embeddings are **compositional**, i.e., constructed from parts of the object, e.g., by another neural net $\rightarrow$ exploit structure of the object
  - E.g., Document embeddings, image embeddings, graph embeddings, . . . $\rightarrow$ Later lectures

t-SNE (1)

- Embedding spaces are high dimensional and difficult to visualize
- **t-SNE** (*t-distributed Stochastic Neighbor Embedding*) is popular approach
  - Maps embedding space (or any other high-dimensional input space) to 2D/3D space such that **"close neighborhoods" are approximately retained** (non-linear mapping)
  - Resulting 2D/3D representations can then be visualized
- What's close? **Similarity** of embeddings $\boldsymbol{z}_j$ to given embedding $\boldsymbol{z}_i$ measured with isotropic Gaussian kernel with bandwidth σ_i (data-point specific; cf. ML 08/1):

$$\mathrm{sim}(\boldsymbol{z}_j|\boldsymbol{z}_i) = \exp(-\|\boldsymbol{z}_j - \boldsymbol{z}_i\|^2/\sigma_i^2) \in [0,1]$$

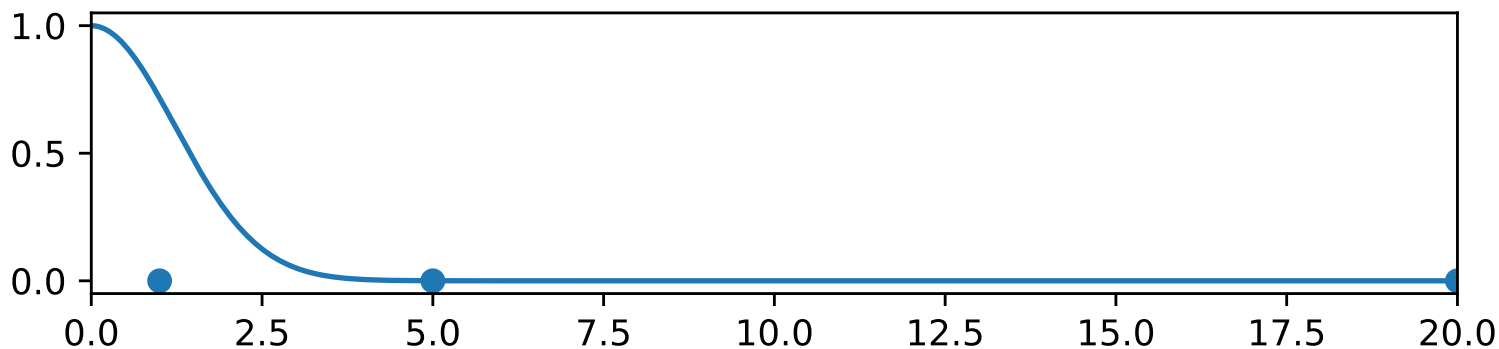
- Used to define **neighbor distribution** over all embeddings $j \neq i$

$$p(j|i) \propto \exp(-\|\boldsymbol{z}_j - \boldsymbol{z}_i\|^2/\sigma_i^2) \quad \text{for} \quad j \in \{1,\ldots,N\} \setminus \{i\}$$

  - Bandwidth σ_i controls number of effective neighbors
  - Small bandwidth $\rightarrow$ few neighbors ($p(j|i)$ concentrated)
  - Large bandwidth $\rightarrow$ many neighbors ($p(j|i)$ spread out)

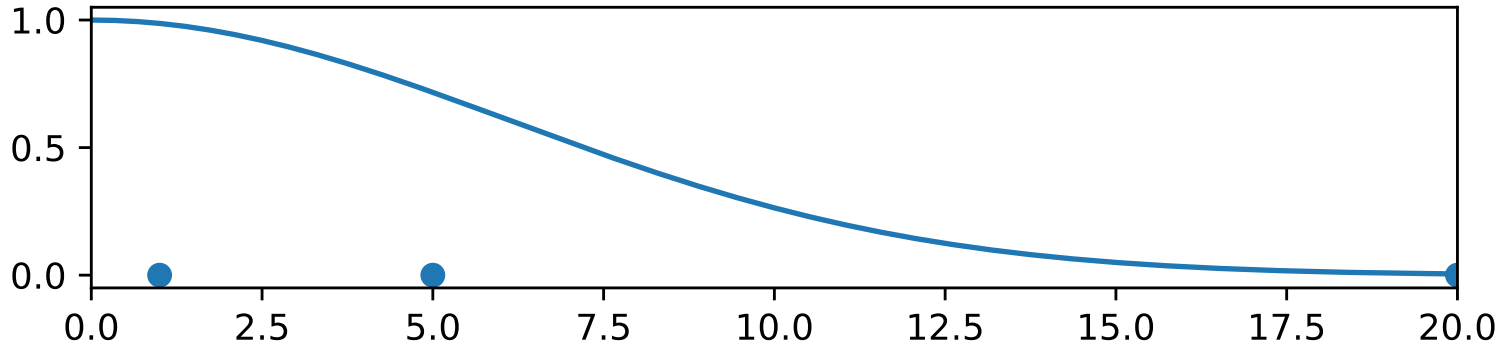
Perplexity

- Let H_i be the **Shannon entropy** of neighbor distribution $p(j|i)$
  $\rightarrow$ Intuitively: number of effective neighbors of data point i in bits (recall: depends on σ_i)
- Let **perplexity** $P_i = 2^{H_i}$
  $\rightarrow$ Intuitively: **number of effective neighbors** of data point i
- Consider a dataset with 4 examples. For example $i = 4$, let 1, 5, 20 be the Euclidean distances of examples $j = 1$, $j = 2$, and $j = 3$ to example 4, respectively.
- Small bandwidth ($\sigma_4^2 = 3$)



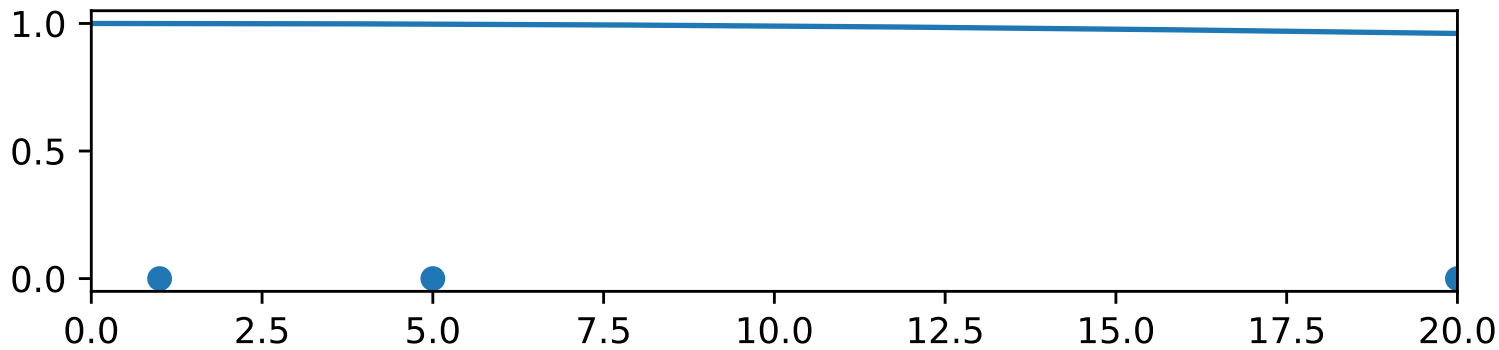
  - Similarities: $0.72, 0.00, 0.00 \rightarrow$ distribution $p(j|4)$: $1.00, 0.00, 0.00$
  - Entropy: $H_i \approx 0 \rightarrow$ perplexity: $P_i \approx 1$

- Medium bandwidth ($\sigma_4^2 = 75$)



  - Similarities: $0.99, 0.72, 0.00 \rightarrow$ distribution $p(j|4)$: $0.58, 0.42, 0.00$
  - Entropy: $H_i \approx 1 \rightarrow$ perplexity: $P_i \approx 2$

- Large bandwidth ($\sigma_4^2 = 10000$)

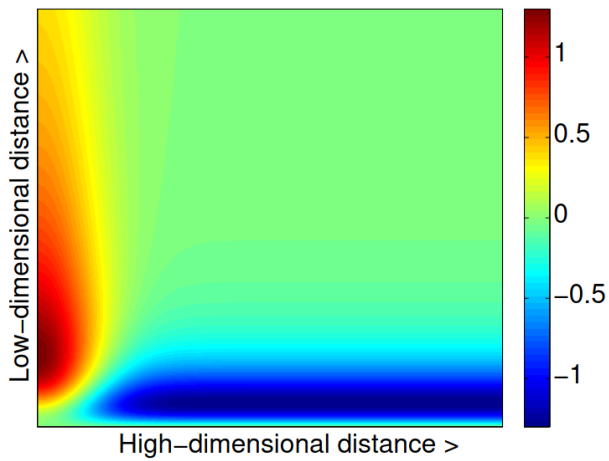


  - Similarities: $0.99, 0.99, 0.96 \rightarrow$ distribution $p(j|4)$: $0.34, 0.34, 0.32$
  - Entropy: $H_i \approx 1.58 \rightarrow$ perplexity: $P_i \approx 3$

Fixed perplexity $\rightarrow$ fixed number of effective neighbors.

t-SNE (2)

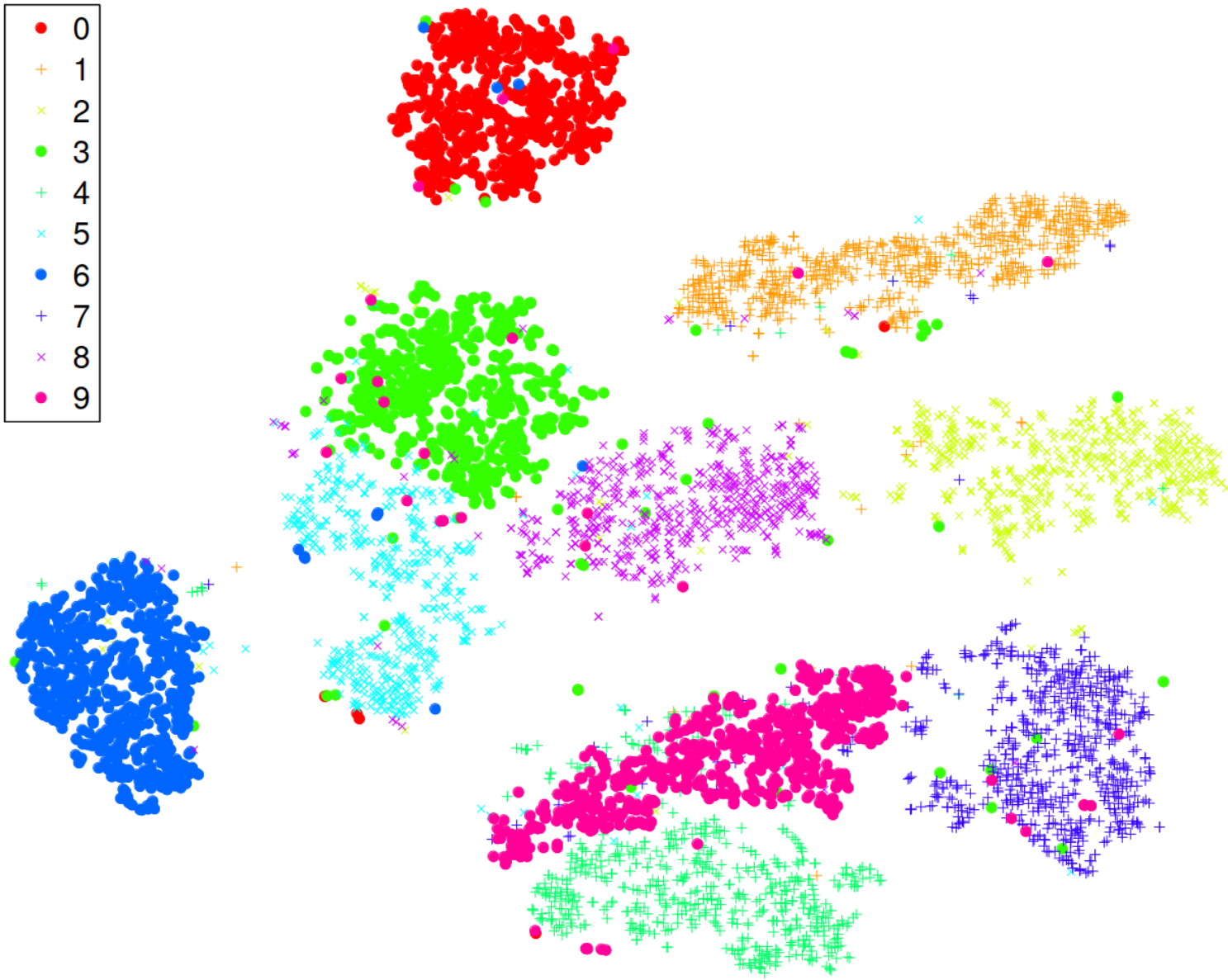
- t-SNE uses a **target perplexity** P as hyperparameter
  - Typically set to 5–50
  - σ_i^2 chosen such that $P_i = P$ (i.e., σ_i different for each i)
- In 2D/3D space, t-SNE uses the Cauchy distribution instead
  - Heavier tails than Gaussian kernel
  - Consequence: (more) ok if embeddings with "moderate" similarity are far in 2D/3D space



Gradient of t-SNE

- Fit via gradient descent (w/ momentum), using KL divergence loss between the neighborhood distributions in embeddings space and in 2D/3D space $\rightarrow$ only effective neighbors matter (!)

Example (6000 MNIST handwritten digits; $28 \times 28 = 784\text{D} \to 2\text{D}$)



t-SNE (discussion)

- Popular, useful method
  - Can visualize sets of inputs, embeddings, weight vectors, ...
  - Transductive (i.e., visualizes a fixed set)
- Hyperparameters matter
  - Perplexity can have significant impact on result
  - t-SNE is an iterative algorithm $\rightarrow$ don't stop too early
  - Learning rate has a significant impact, too
  - Repeated runs of t-SNE may produce (very) different results
- Interpretation is tricky
  - Sole goal: close data points in embedding space should stay close
  - **Distances, "cluster sizes," and "cluster locations" in t-SNE plot often meaningless**
  - Patterns visible in t-SNE plot may not be real patterns
  - Helpful: experiment with different choices of perplexity
- See Wattenberg et al. (2016) for examples and discussion

Recall: Softmax layer

- Recall: **softmax layers** for classification (categorical outputs)
  - C classes, input $\boldsymbol{z} \in \mathbb{R}^Z$ from the previous layer
  - Softmax layer computes $\hat{\boldsymbol{y}} = S\left(\frac{\boldsymbol{W}^\top \boldsymbol{z} + \boldsymbol{b}}{T}\right)$
  - $\boldsymbol{\eta} = \boldsymbol{W}^\top \boldsymbol{z} + \boldsymbol{b}$ contains softmax scores of each class
  - $\hat{\boldsymbol{y}} \in \mathcal{S}_C$ contains predicted class probabilities
  - $T \in \mathbb{R}_+$ is a hyperparameter known as the **temperature**
    $\rightarrow$ Controls smoothness of distribution
  - As $T \rightarrow 0$, the resulting distribution concentrates around the largest softmax score $\rightarrow$ most likely prediction

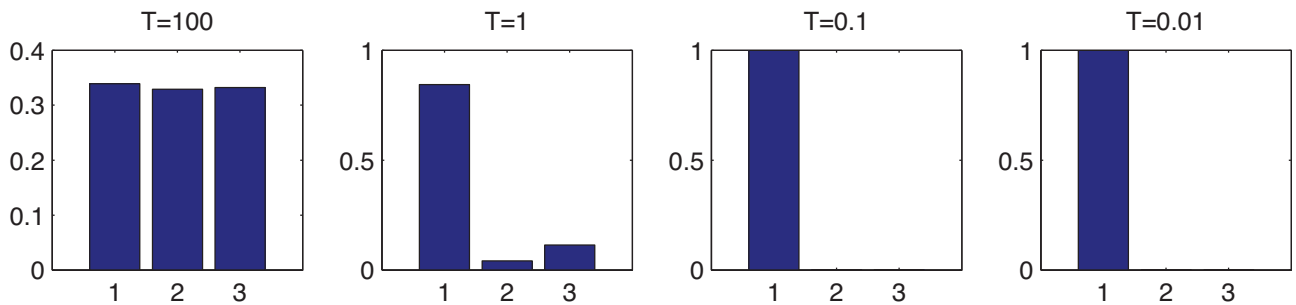


Figure 4.4 Softmax distribution $\mathcal{S}(\boldsymbol{\eta}/T)$, where $\boldsymbol{\eta} = (3, 0, 1)$, at different temperatures T. When the temperature is high (left), the distribution is uniform, whereas when the temperature is low (right), the distribution is "spiky", with all its mass on the largest element. Figure generated by `softmaxDemo2`.

Interpreting softmax layers

- Let's assume no bias and $T = 1$

$$\hat{\boldsymbol{y}} = S(\boldsymbol{W}^\top \boldsymbol{z})$$

- Class probabilities implicitly determined by **softmax scores**

$$\eta_c = \boldsymbol{w}_c^\top \boldsymbol{z} = \|\boldsymbol{z}\| \, \|\boldsymbol{w}_c\| \cos \angle(\boldsymbol{w}_c, \boldsymbol{z})$$

- Consider some input $\boldsymbol{z}$ (arbitrary, but fixed)
  - $\|\boldsymbol{z}\|$ acts as "inverse temperature"
  - Corresponding temperature is $T = 1/\|\boldsymbol{z}\|$
  - **Norm of $\boldsymbol{z}$ controls uniformity of softmax distribution**
- Suppose all weight vectors have unit norm ($\|\boldsymbol{w}_c\| = 1$)
  - **Softmax layer then measures the similarity** between input $\boldsymbol{z}$ and each class vector $\boldsymbol{w}_c$
- When weight vectors have different norms, then $\|\boldsymbol{w}_c\|$ **acts as coefficient of proportionality** (think: class weight)
- If we had a bias term, then **bias serves as a priori softmax score** (since then $\eta_c = \boldsymbol{w}_c^\top \boldsymbol{z} + b_c$; think: class prior)

Deep Learning

04 – Layers for Categorical Data

Part 2: Word Vectors (Example)

Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-1

Word vectors (1)

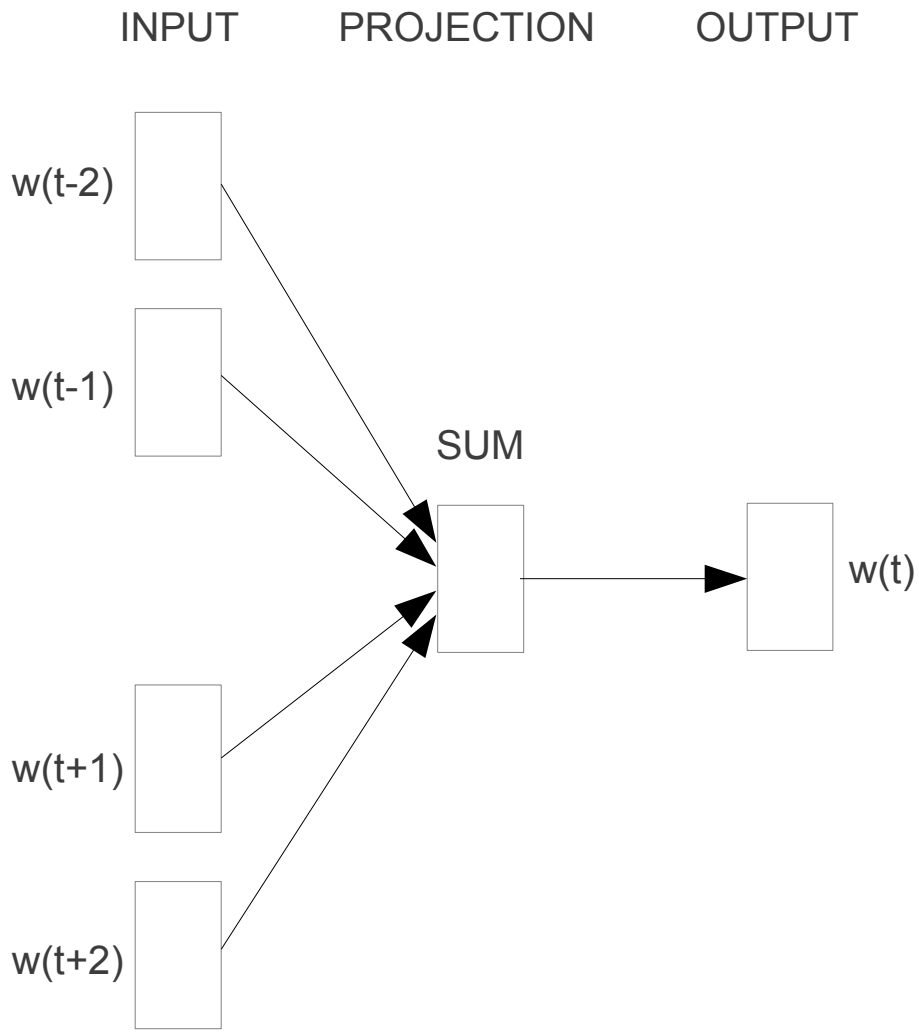
- Now: word vectors (or word embeddings)
  - An example of using embedding layers and a softmax layer
  - (Was) used to represent words in a variety of NLP tasks
  - Simple and effective
  - Not really relevant anymore, but instructive example
  - And: key ideas relevant later (starting from 05)
- **Word vectors** (*distributed word representations*) map words to continuous representations; goals:
  1. Capture *semantic similarity* between words, i.e., provide similar representations for words that have similar meanings (e.g., "God" and "Lord")
  2. Compositionality to obtain sequence representations
- Pretrained on large text corpora
- Why use word vectors? As before, downstream models then
  - Have significantly less parameters to train (recall: $Z \ll V$)
  - Uses "meaning" of words, not words themselves
  - Can handle words unseen in downstream training data

Word vectors (2)

- How to obtain word vectors?
  - **Distributional hypothesis** in linguistics states that words that occur in similar contexts tend to have similar meanings
  - Key idea: train word vectors such that words that appear in similar contexts have similar representations
  - Under this hypothesis: similar representation $\rightarrow$ similar meaning
- Example: continuous bag-of-words (CBOW) model
  - Task: predict missing word given the set of surrounding words (= **context**)
  - Hyperparameter Z: size of word vector
  - Hyperparameter W: size of left/right context

CBOW (1)

Task: predict current word w_t given its $2W$ surrounding words (= context).



CBOW (2)

- How to read this architecture?
- Input layer
  - Maps words to word vectors (each of the $2W$ words separately) via an embedding layer
  - Embedding matrix $\boldsymbol{E} \in \mathbb{R}^{V \times Z}$ *shared* across context words
    $\rightarrow$ **parameter sharing**
  - Note: embedding layer in NLP literature often implicit (i.e., not explicitly drawn, as in previous slide)
- Sum layer
  - Takes $2W$ embeddings and sums element-wise (*composition*)
    $\rightarrow$ Z-dimensional continuous representation of context
  - Note: $\sum_i \boldsymbol{e}_{w_i}$ = word counts $\rightarrow$ bag-of-words
  - Note: $\sum_i \text{emb}(w_i)$ = word vector sums $\rightarrow$ "continuous" bag-of-words
- Output layer
  - Softmax, trained to predict w_t ($C = V$ classes)
  - Parameterized by weight matrix $\boldsymbol{W} \in \mathbb{R}^{Z \times C}$
    (note: ignored for downstream models, i.e., only $\boldsymbol{E}$ used)
  - Again: softmax layer often implicit when categorical output

Discussion

- Recall key idea: train word vectors such that words that appear in similar contexts have similar representations
- Context representation $\boldsymbol{z}$ is a bottleneck
  - Z-dimensional representation of $2WV$-dimensional context
  - Forces "compression", exposes similarities
- Representation $\boldsymbol{z}$ should be "useful" to predict missing word
  - Recall: softmax score η_c of word c proportional to (cosine) similarity of context representation $\boldsymbol{z}$ and weight vector $\boldsymbol{w}_c$
  - All contexts of word c ideally represented by a vector $\boldsymbol{z}$ similar to $\boldsymbol{w}_c$
    $\rightarrow$ Exposes context similarities
  - Since $\boldsymbol{z}$ is composed of word vectors, the word vectors implictly expose word similarity
- Training
  - Slide window over text corpora, optimize loss (e.g., CE)
  - In practice, skip-gram model tends to work better (given current word, predict context words)
  - Cheap except the softmax layer (!)
    $\rightarrow$ CBOW uses "hierarchical softmax"

Deep Learning

04 – Layers for Categorical Data

Part 3: Softmax with Many Classes

Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-1

Softmax with many classes

- Consider a softmax layer with C outputs ($\boldsymbol{p}$) and Z inputs from the previous layer ($\boldsymbol{z}$). We have

$$p_c = S(\boldsymbol{W}^\top \boldsymbol{z} + \boldsymbol{b})_c = \frac{\exp(\langle \boldsymbol{w}_c, \boldsymbol{z} \rangle + b_c)}{\sum_{c'} \exp(\langle \boldsymbol{w}_{c'}, \boldsymbol{z} \rangle + b_{c'})}.$$

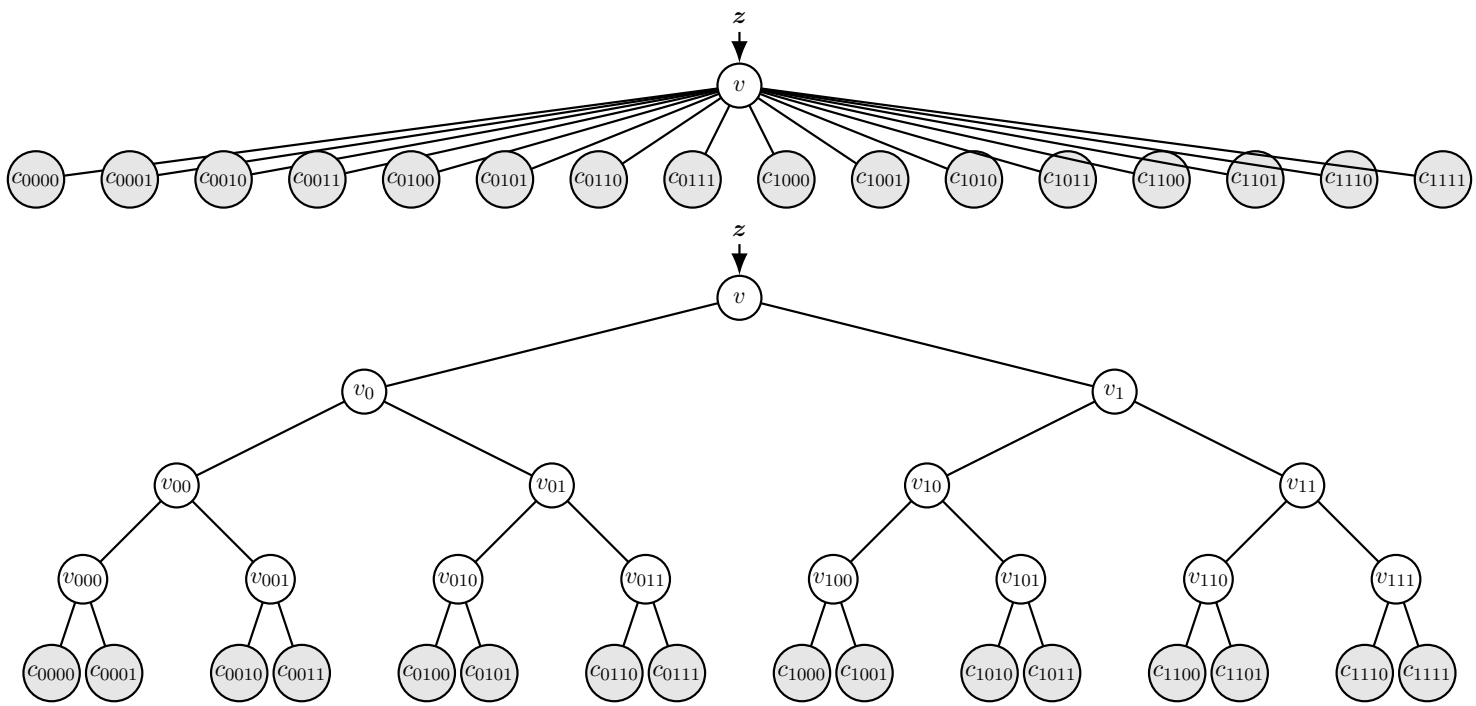
- During training, given example $(\boldsymbol{x}, y)$
  - Need to compute output during forward pass

$$p_y = p(y|\boldsymbol{x}, \boldsymbol{\theta}) = p(y|\boldsymbol{z}, \boldsymbol{W}, \boldsymbol{b}) = S(\boldsymbol{W}^\top \boldsymbol{z} + \boldsymbol{b})_y,$$

    where $\boldsymbol{z}$ depends on $\boldsymbol{x}$ and $\boldsymbol{\theta}$ (= all model parameters)
  - To do so, we need to compute *all* terms $\exp(\langle \boldsymbol{w}_{c'}, \boldsymbol{z} \rangle + b_{c'})$
  - Complexity: $O(ZC)$ *per example* $\rightarrow$ expensive when ZC large
  - Gradient in backprop generally non-zero for *all* parameters ($\boldsymbol{W}$ & $\boldsymbol{b}$)
- E.g., word vectors trained on Google News: $C = 3\text{M}$
  - CBOW with $W = 2$: Add 4 vectors to obtain hidden representation $\boldsymbol{z}$, compute 3M inner products/exponents for softmax output
  - Common approach: avoid plain softmax

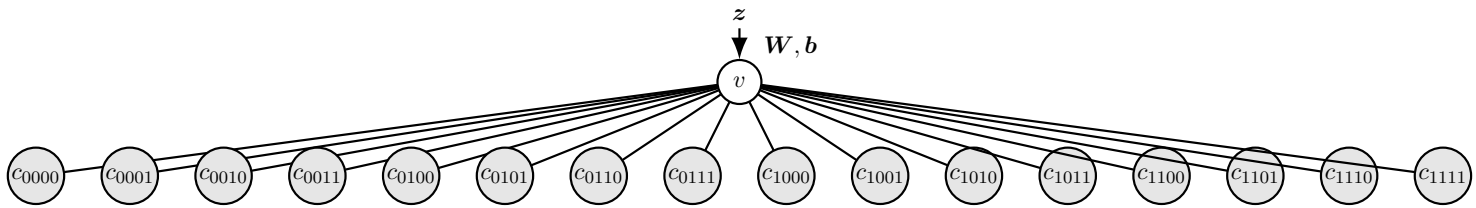
Hierarchical softmax (1)

- **Hierarchical softmax** layer: arrange classes in a "decision tree"
  - Input is $\boldsymbol{z}$ (= output of layer before softmax),
  - Output is $\boldsymbol{p}$ (= probability for each class)
  - Leaves are classes
  - Interior vertices are decision points
  - Each possible choice associated with a probability
  - Probability of class = product of probabilities of corresponding path
- Example: two alternative trees for $C = 16$



Hierarchical softmax (2)

- To model the probability distribution over the children of each interior vertex, use softmax
  - One weight matrix and bias vector per interior node
  - Probability distribution over children of v_i is $S(\boldsymbol{W}_i^\top \boldsymbol{z} + \boldsymbol{b}_i)$
  - Note: $\boldsymbol{z}$ is used at each interior node
    $\rightarrow$ Influences all decisions
- Flat tree $\rightarrow$ Output is $p_y = S(\boldsymbol{W}^\top \boldsymbol{z} + \boldsymbol{b})_y \equiv$ softmax



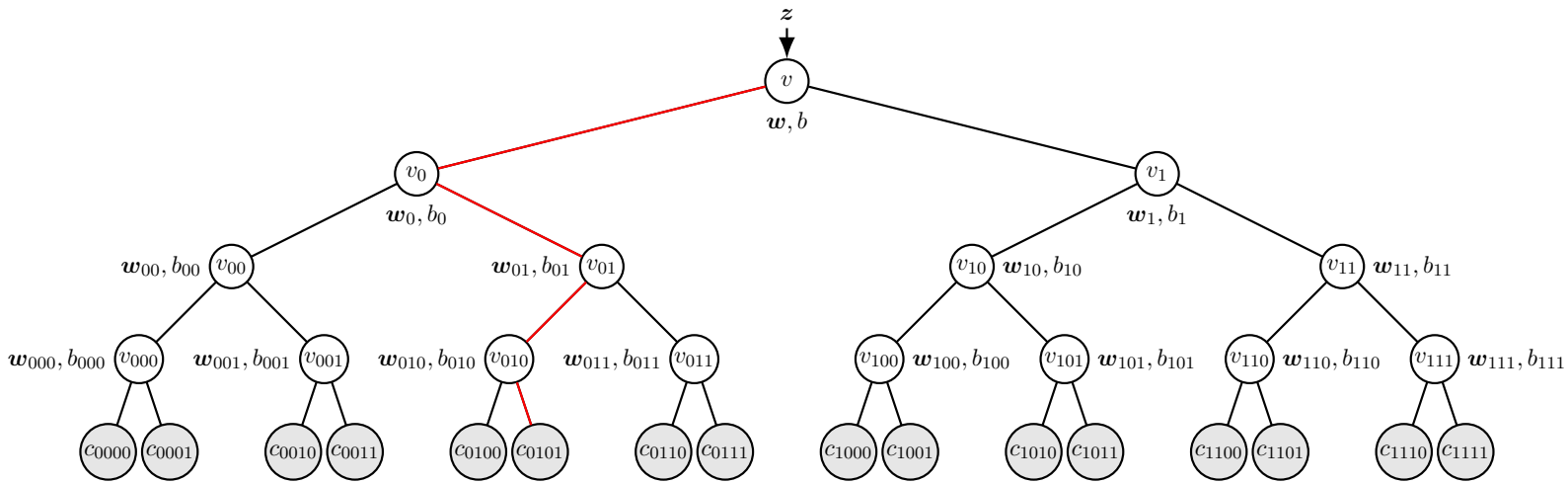
  - Output p_y depends on entire $\boldsymbol{W}$ and $\boldsymbol{b}$
    $\rightarrow$ C weight vectors and bias terms

Hierarchical softmax (3)

- Binary trees
  - Store weight vector and (scalar) bias at each interior node
  - C classes $\rightarrow$ $C-1$ interior nodes / weight vectors / bias terms
  - Probability p_y of, say, leaf $y = 0101$ is

$$S(0, \boldsymbol{w}^\top \boldsymbol{z} + b)_0 \cdot S(0, \boldsymbol{w}_0^\top \boldsymbol{z} + b_0)_1 \cdot S(0, \boldsymbol{w}_{01}^\top \boldsymbol{z} + b_{01})_0 \cdot S(0, \boldsymbol{w}_{010}^\top \boldsymbol{z} + b_{010})_1$$

    $\rightarrow$ Only depends on 4 (out of 15) weight vectors and bias terms



- Balanced binary trees
  - In general: C classes $\rightarrow$ access $\log_2 C$ weight vectors / bias terms
  - E.g., for $C = 3\text{M} \rightarrow \log_2 C \approx 22$
  - Much more efficient to compute
  - All other weight vectors / biases have zero gradient during backprop

Hierarchical softmax (4)

- Choice of tree matters for prediction performance
  - Hierarchical softmax is able to produce good predictions if the classes in the "right" subtree are easy to discriminate from the classes of the "wrong" subtrees (by softmax classifier)
  - E.g., for words: cluster words and recursively partition them into two clusters $\rightarrow$ hierarchical softmax can achieve similar prediciton performance as regular softmax
  - E.g., when classes arranged in hierarchy $\rightarrow$ use directly
- Choice of tree matters for training speed
  - Flat: as slow as softmax
  - Balanced: logarithmic cost
  - Fastest: Huffman tree based on class frequencies
    $\rightarrow$ Minimize expected path lengths (frequent classes $\rightarrow$ short path)
- No/limited runtime improvement during prediction
  - Still need to compute all probabilities to get distribution over labels
  - Largest-probability prediction can often be obtained faster (e.g., best-first search, beam search, MIPS)

Sampling-based approximate softmax (1)

- Many other approaches to approximate softmax exist, most notably, based on **sampling**
- Fix some $\boldsymbol{z}$, let $\eta_c = \langle \boldsymbol{w}_c, \boldsymbol{z} \rangle + b_c$, and $p_c = S(\boldsymbol{W}^\top \boldsymbol{z} + \boldsymbol{b})_c$
- Log-likelihood for a single example $(\boldsymbol{x}, y)$ is

$$\ell = \log p_y = \log S(\boldsymbol{W}^\top \boldsymbol{z} + \boldsymbol{b})_y = \log \frac{\exp(\eta_y)}{\sum_c \exp(\eta_c)}$$

$$= \eta_y - \log \sum_c \exp(\eta_c)$$

  where as before, $\boldsymbol{z}$ depends on $\boldsymbol{x}$ and parameters $\boldsymbol{\theta}$

- Gradient is

$$\nabla_{\boldsymbol{\theta}} \ell = \nabla_{\boldsymbol{\theta}} \eta_y - \nabla_{\boldsymbol{\theta}} \log \sum_c \exp(\eta_c) = \cdots$$

$$= \nabla_{\boldsymbol{\theta}} \eta_y - \sum_c p_c \nabla_{\boldsymbol{\theta}} \eta_c = \nabla_{\boldsymbol{\theta}} \eta_y - E_{c \sim \mathrm{Cat}(\boldsymbol{p})}[\nabla_{\boldsymbol{\theta}} \eta_c]$$

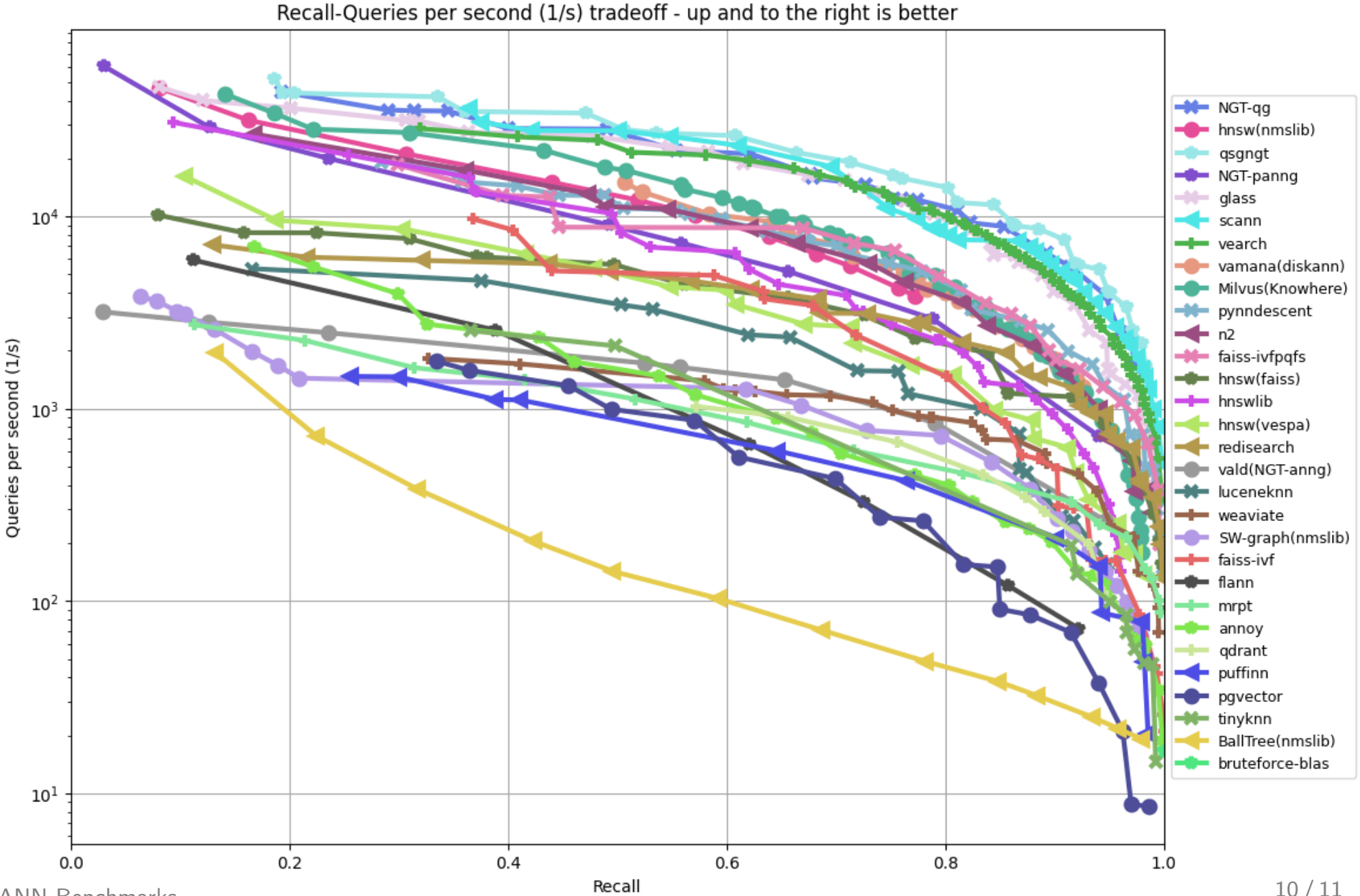
Sampling-based approximate softmax (2)

- Gradient: $\nabla_{\boldsymbol{\theta}} \ell = \nabla_{\boldsymbol{\theta}} \eta_y - E_{c \sim \mathrm{Cat}(\boldsymbol{p})}[\nabla_{\boldsymbol{\theta}} \eta_c]$
  - Part 1: $\nabla_{\boldsymbol{\theta}} \eta_y \rightarrow$ **positive reinforcement** for current class
  - Part 2: $E_{c \sim \mathrm{Cat}(\boldsymbol{p})}[\nabla_{\boldsymbol{\theta}} \eta_c] \rightarrow$ (weighted) **negative reinforcement** for other classes
- Sampling-based approaches approximate neg. reinforcement term
  - This would be easy if we could sample from distribution $\mathrm{Cat}(\boldsymbol{p})$, but computing $\boldsymbol{p}$ is what we try to avoid in the first place
  - Simple, common heuristic: **negative sampling**
    $\rightarrow$ Sample uniformly (instead of using $\boldsymbol{p}$), no guarantees, but cheap
  - Many more approaches have been proposed (e.g., adaptive importance sampling or noise contrastive estimation)
- For more on approximating softmax, see S. Ruder (2016)
- Classification with many classes sometimes called extreme classification
  - Important problem in practice
  - Active research topic

Maximum inner product search

- For prediction, may only care about most-likely prediction
  - **Most-likely prediction = largest softmax score**
  - Since: $\log p(y|\boldsymbol{x}) = \eta_y - \log \sum_c \exp(\eta_c) = \eta_y + C(\boldsymbol{x})$, where $C(\boldsymbol{x})$ depends on $\boldsymbol{x}$ only (=constant)
- Softmax scores are $\boldsymbol{W}^\top \boldsymbol{z}$ (ignoring bias)
  - So $\eta_1 = \boldsymbol{w}_1^\top \boldsymbol{z}$, $\eta_2 = \boldsymbol{w}_2^\top \boldsymbol{z}$, $\ldots$, $\eta_C = \boldsymbol{w}_C^\top \boldsymbol{z}$
  - Most-likely prediction = $\boldsymbol{w}_c$ with largest inner product with $\boldsymbol{z}$
  - Note: that's the prediction at "temperature $T = 0$" (cf. 04-2)
- **Maximum inner product search**
  - Given upfront and indexed: set of vectors $\mathcal{W} = \{ \boldsymbol{w}_1, \ldots, \boldsymbol{w}_C \}$
  - Query: another vector $\boldsymbol{z}$
  - Output: top-k inner-products (of vectors in $\mathcal{W}$ with $\boldsymbol{z}$)
  - Slow to do exactly (e.g., our work on LEMP), but many fast approximate methods exists
  - Special case of **approximate nearest neighbor search**
  - Applications include: "temperature 0" softmax prediction, search in vector stores

Example (glove-100-angular, $k = 10$)



NeurIPS'23 Competition Track: Big-ANN

Supported by Microsoft Pinecone aws zilliz

New: the latest ongoing leaderboard has been released (March 1st, 2024).

Top entries:

Filter track			OOD track			Sparse track		
Rank	**Algorithm**	**QPS@90% recall**	**Rank**	**Algorithm**	**QPS@90% recall**	**Rank**	**Algorithm**	**QPS@90% recall**
1	Pinecone-filter	85,491	1	Pinecone-ood	38,088	1	Zilliz	10,749
2	Zilliz	84,596	2	Zilliz	33,241	2	Pinecone_smips	10,440
3	ParlayANN IVF^2	37,902	3	RoarANN	22,555	3	PyANNS	8,732
4	Puck	19,193	4	PyANNS	22,296	4	shnsw	7,137
...	...	...	...	...	...	...	...	...
Baseline	FAISS	3,032	Baseline	Diskann	4,133	Baseline	Linscan	93

Note: entries by pinecone and zilliz are not open source.

Full Leaderboard, Plots, and Rules

Deep Learning

05 – Part Embeddings

Prof. Dr. Rainer Gemulla

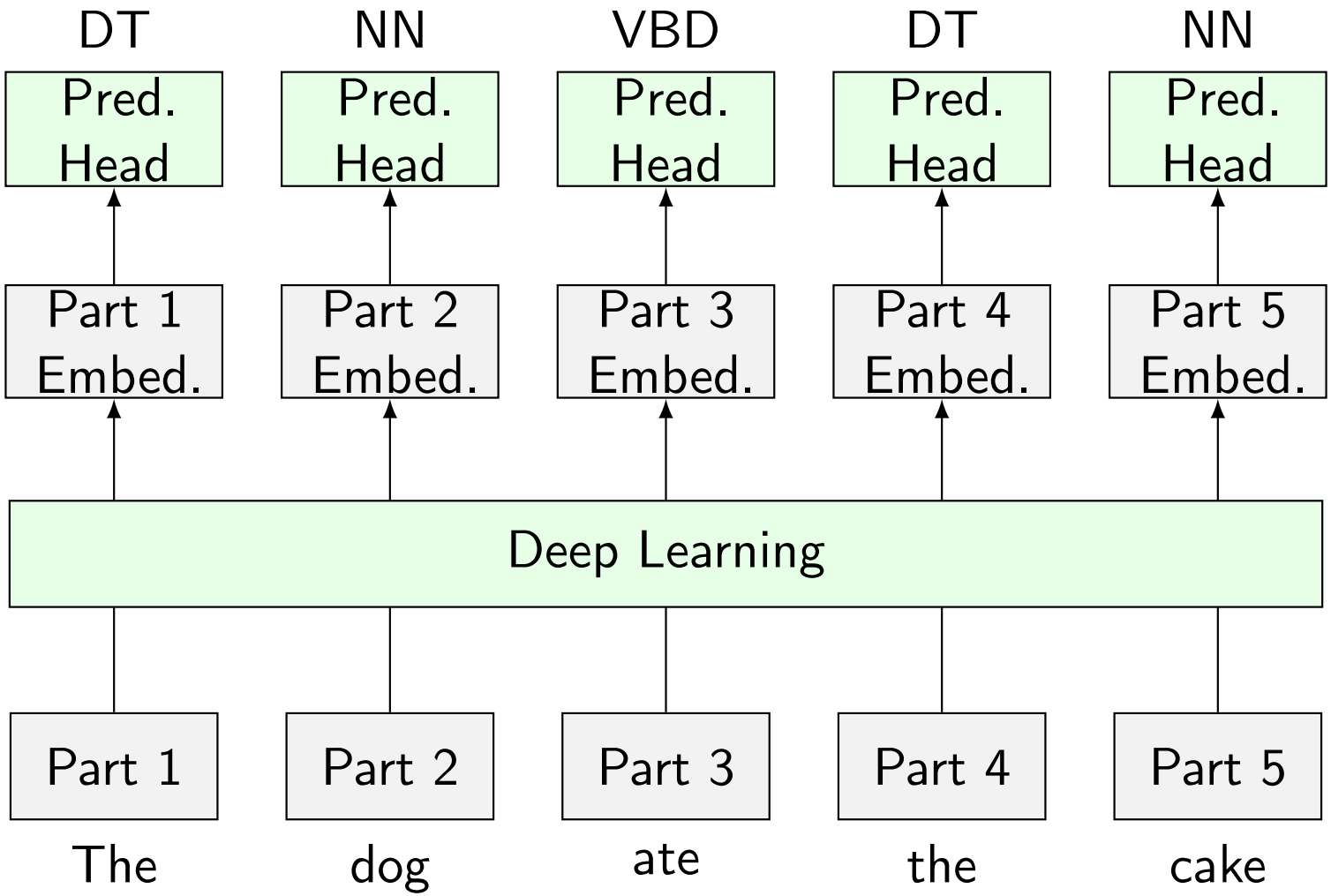
Universität Mannheim

Version: 2025-1

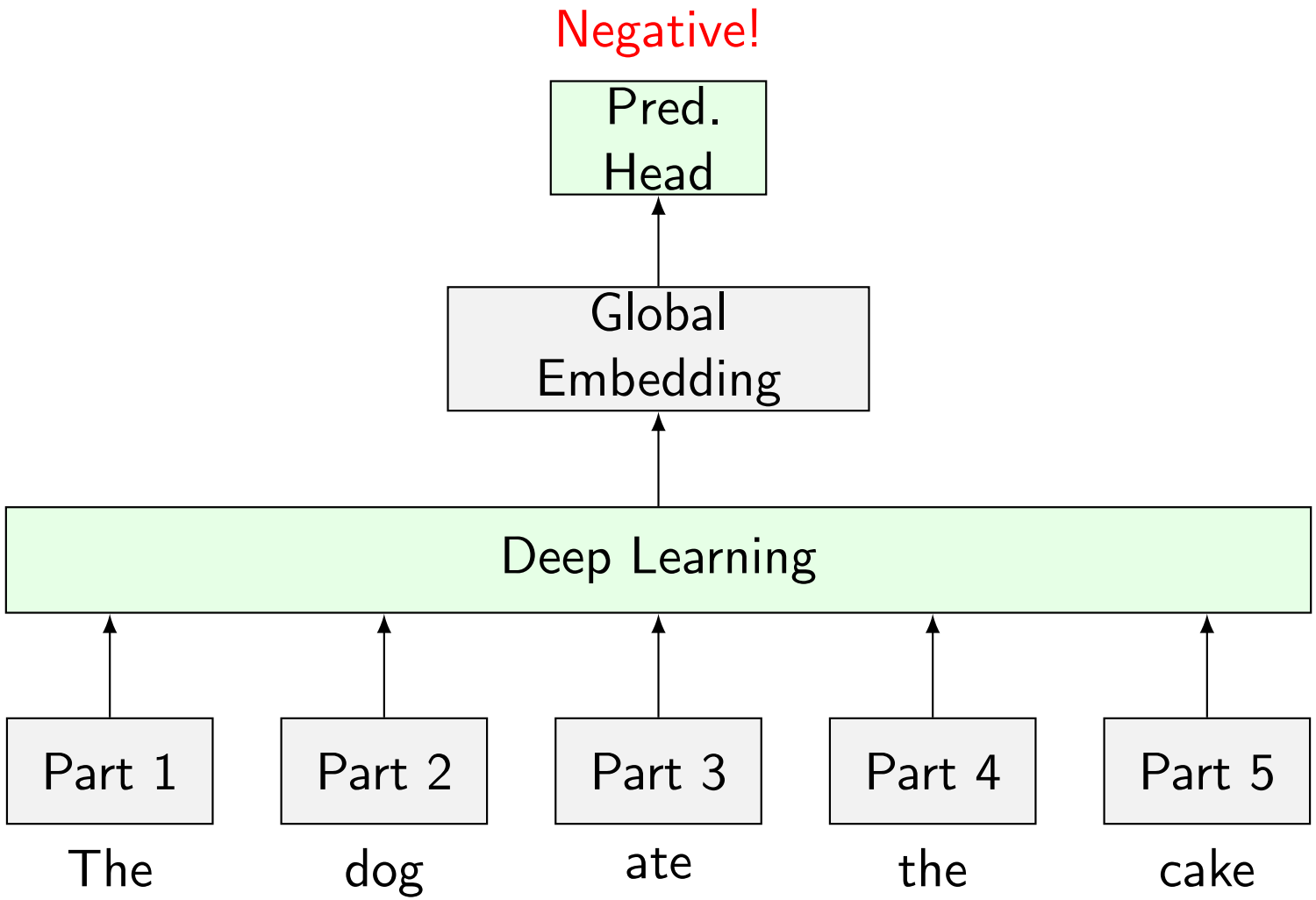
Part embeddings

- So far, we considered monolithic inputs (e.g., real-valued vectors or categorical inputs)
- From now on: (more) structured input spaces
  - I.e., inputs that consists of multiple **parts**
  - E.g., text documents $\rightarrow$ words or tokens (sequence)
  - E.g., images $\rightarrow$ pixels (grid)
  - E.g., graphs $\rightarrow$ vertices, edges (irregular)
  - E.g., shopping cart $\rightarrow$ products (set)
- **Global embeddings** represent the entire input
  - Useful for input-level tasks
- **Part embeddings** represent individual parts
  - Useful for part-level tasks
  - Also used to obtain global embeddings (recall: compositional embeddings from 04-1)
- General approach (forward pass)
  1. Obtain part embeddings and/or global embedding
  2. Add prediction head (or other downstream network) on top to obtain final output

Example: Part-of-speech tagging (part-level task)



Example: Sentiment classification (input-level task)



Example: GNNs (input-level task)

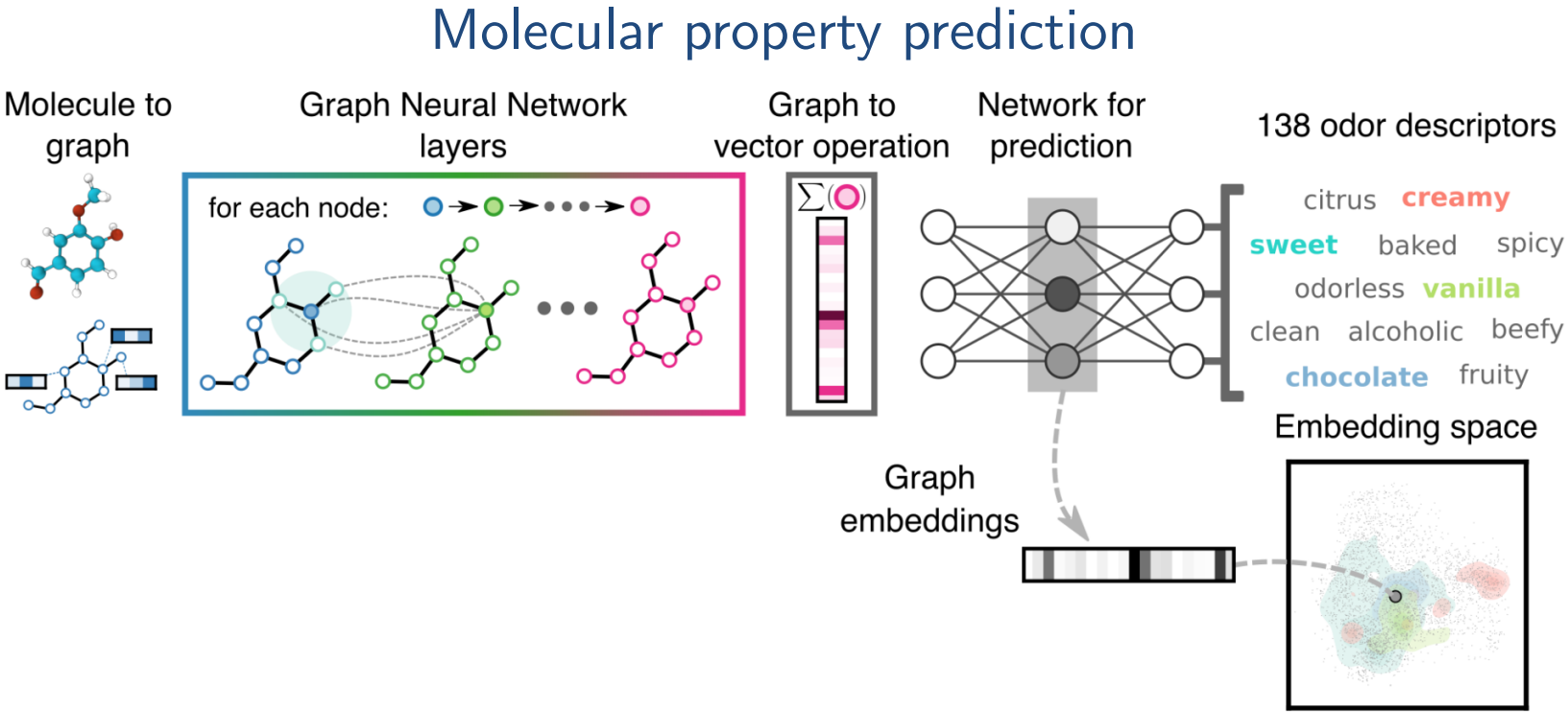


Figure 2: **Model Schematic.** Each molecule is first featurized by its constituent atoms, bonds, and connectivities. Each Graph Neural Network (GNN) layer, here represented as different colors, transforms the features from the previous layer. The outputs from the final GNN layer is reduced to a vector, which is then used for predicting odor descriptors via a fully-connected neural network. We retrieve graph embeddings from the penultimate layer of the model. An example of the embedding space representation for four odor descriptors is shown in the bottom right; the colors of the regions in this plot correspond to the colors of odor descriptors in top right.

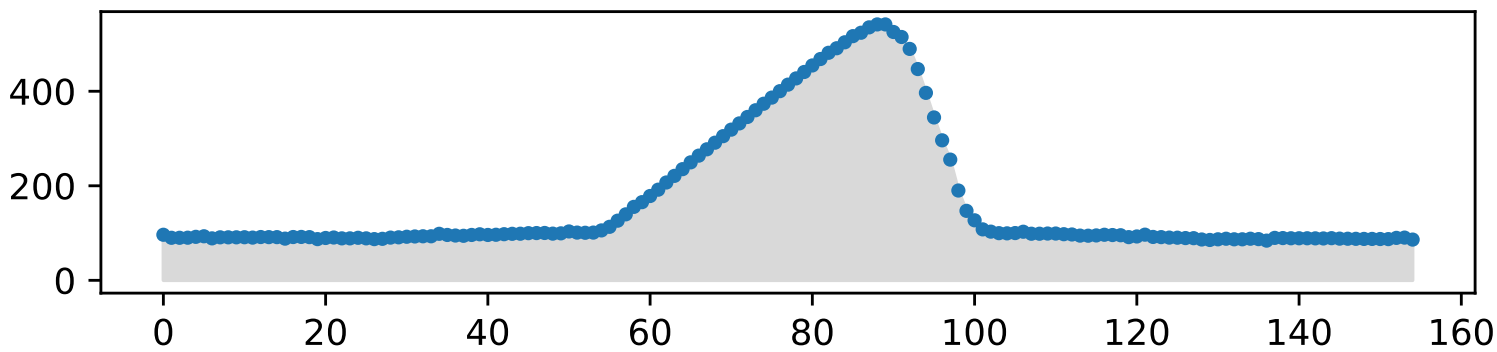
Sanchez-Lengeling et al., 2019

Key operations

- Input represented in terms of
  - Global features (if any)
  - Parts and relationship between parts (e.g., order in a sequence or edges in a graph)
  - Part features (= initial part embeddings)
- Three key operations
  1. **Contextualization**: incorporate information from other parts into each part embedding
  2. **Local compute**: update each part embedding individually
  3. **Pooling**: aggregate multiple (or all) part embeddings
- We will see: many architectures follow this high-level pattern
- Each operation can be fixed a priori or be learned from data
  - Different architectures differ in how this is done
  - Operations may occur multiple times and in different orders
  - Sometimes: multiple operations merged into one
  - Sometimes: other operations (e.g., drop parts)

0. Input representation

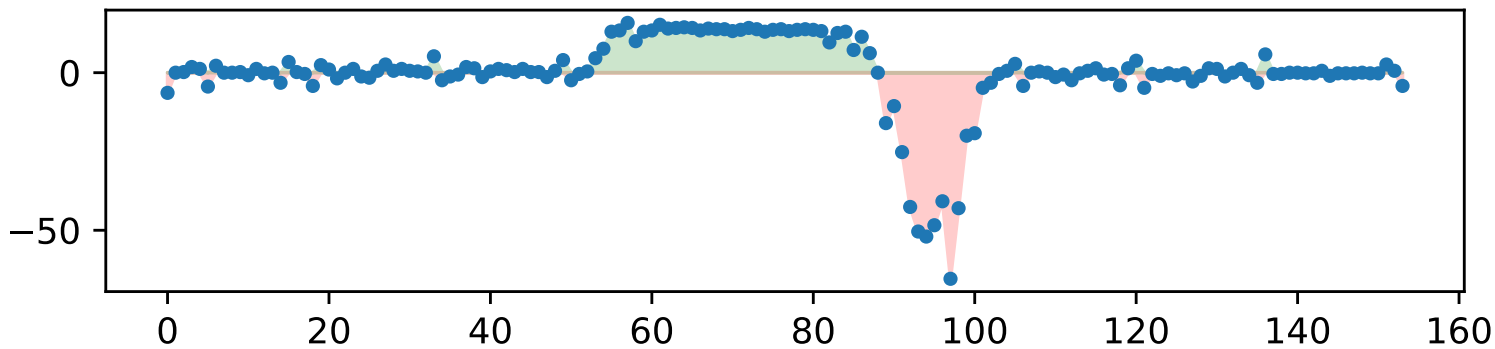
- I recently biked from Mannheim to Weißer Stein and back
  - Task: How many meters did I climb? (input-level)
  - Let's solve this task using the three key operations
- We start with in input representation
  - Here is the altitude profile



  - Parts = measurements at 1 minute intervals (ca. 150 parts)
  - Initial part representation: altitude at that time
  - Relationship: sequence (ordered)

1. Contextualization

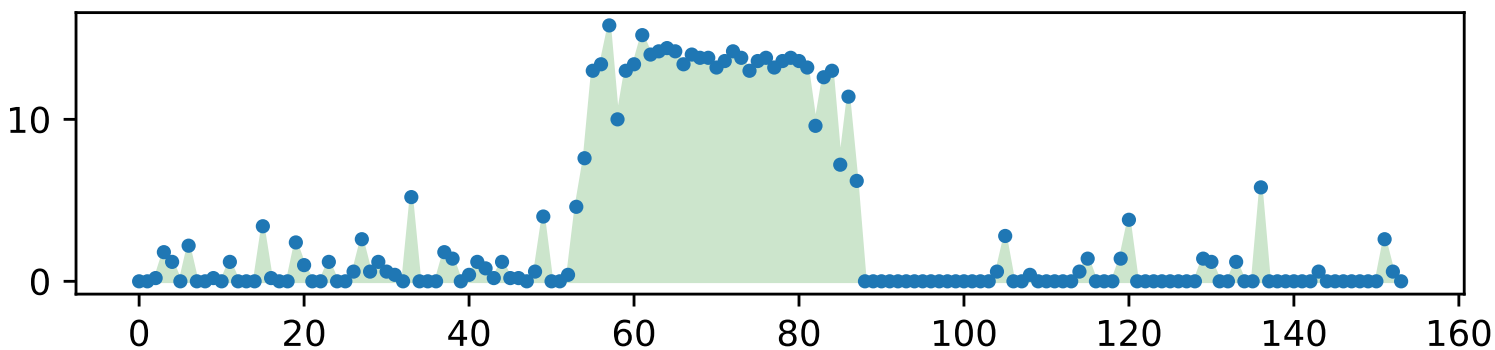
- **Contextualization**: incorporate information from other parts into each part embedding
  - Input: part embeddings
  - Output: **contextualized part embeddings**
  - Updated representation of each part depends on some/all other parts
  - Examples: convolution, recurrence, self-attention
- For our example, let's compute the increment/decrement in altitude w.r.t. the previous timestep



  - We will see: that's a "convolution" operation
  - Observe that relationship between parts (here: order) matters
  - Contextualized parts now represent altitude changes
  - Each part representation depends on other parts
    → contextualization

2. Local compute

- **Local compute**: update each part embedding individually
  1. Applied individually to each part
  2. Input: one part embedding
  3. Output: updated part embedding
  4. Updated representation of each part depends only on its own representation
  5. Example: MLP
- Let's zero out the downhill parts



  - That's a part-wise ReLU operation
  - Relationship between and values of other parts ignored
  - Updated parts now represent altitude increments

3. Pooling

- **Pooling**: aggregate multiple (or all) part embeddings
  - Input: multiple part embeddings
  - Output: aggregated embedding
  - Examples: mean pooling, sum pooling, max pooling, attention
- Special case: **readout** = pool all part embeddings to obtain global embedding
- Let's sum up the altitude increments
  - Result: $503.4\,\mathrm{m}$
  - That's a sum-pooling operation
  - It's also a readout: all "embeddings" are pooled together
  - In this case, no prediction head required

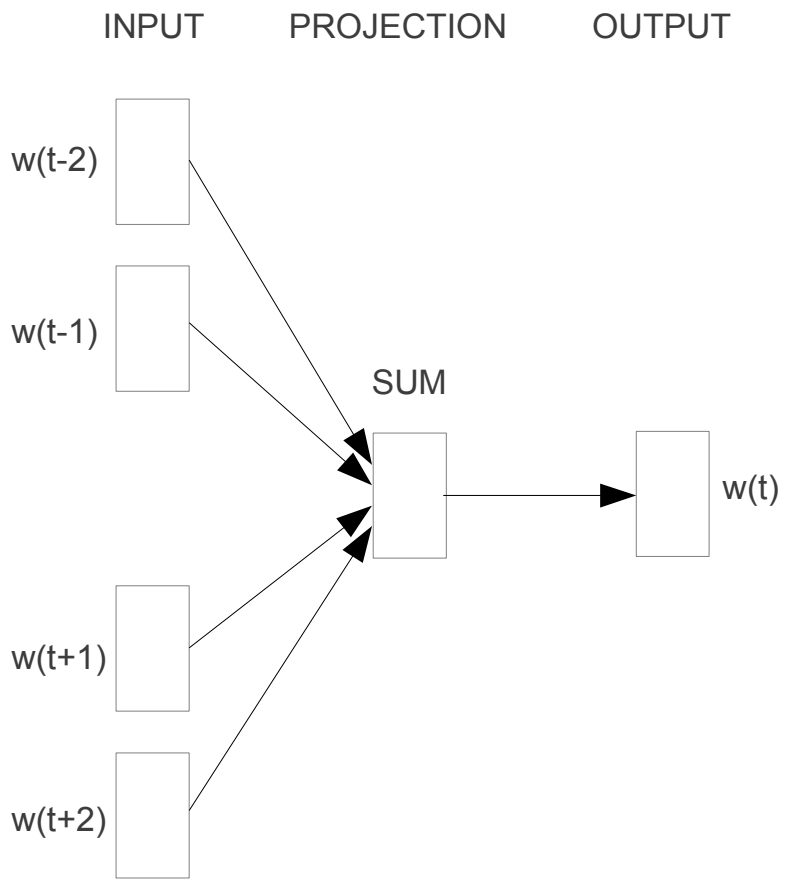
Template models, parameters sharing

- We've obtained the final result by
  - Starting with part embeddings
  - Contexualizing and updating the part embeddings
  - A final readout operation to obtain a global representation
- Instance of a **template model**
  - I.e., a set of rules used to construct a neural network
  - E.g., "compute the difference to the previous part" (contextualization)
  - E.g., "zero out negative values" (local compute)
  - The same template would also work for other inputs and with different numbers of parts (e.g., shorter or longer rides)
- We used **parameter sharing**
  - Each individual operation was the same for all parts
  - But with different inputs
  - When we use parameterized (learned) functions, we share the parameters across parts
- Both template modeling and parameter sharing are used by many DL architectures $\rightarrow$ coming up

Recall: CBOW

Task: predict current word w_t given its $2W$ surrounding words (= context).

Local compute	Pooling	Prediction head
(embedding layer)	(sum)	(softmax layer)



Mikolov et al., 2012

Preview: Parallel processing and cost

- Local compute
  - (Embarrassingly) easy to parallelize $\rightarrow$ process parts independently
- Contextualization
  - Sometimes sequential computation needed (n parts $\rightarrow$ n steps)
    $\rightarrow$ Limited parallelizability (e.g., non-linear RNNs)
  - Sometimes compute-heavy (n parts $\rightarrow$ n^2 pairwise influences)
    $\rightarrow$ Easy to parallelize, limited scalability (e.g., Transformer encoder)
  - Sometimes intermediate (each part influenced by only some parts)
    $\rightarrow$ Still parallelizable, better scalability (e.g., CNNs, GNNs)
- Pooling
  - Depends
  - Sometimes comparably cheap (e.g., readout, done once)
  - Sometimes part of contextualization (e.g., attention in Transformers, done many times)

Deep Learning

06 – Convolutional Neural Networks

Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-3

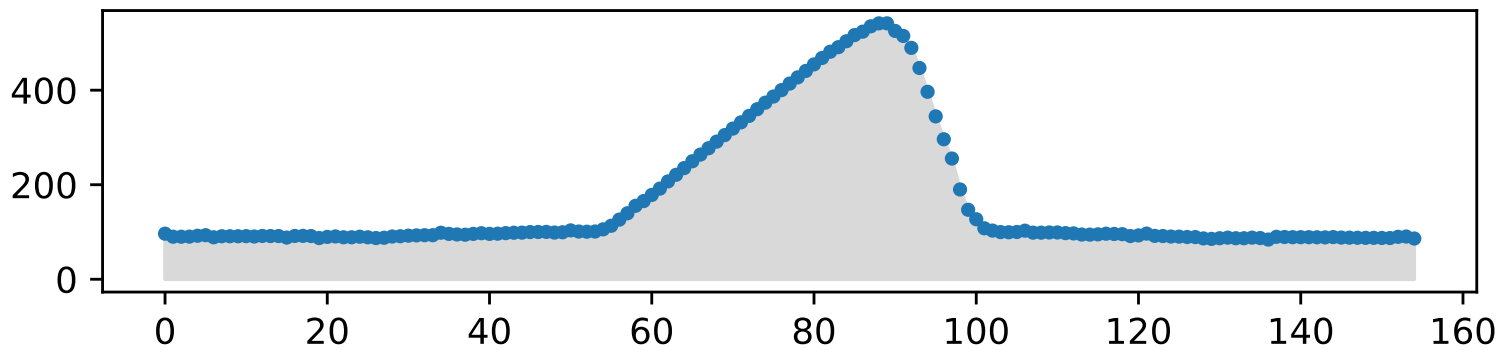
Outline

Grid data

- **Convolutional neural networks (CNN)** are a family of neural networks for processing **grid data**
  - 1D grid $\rightarrow$ sequential data (e.g., time series, text, audio, ...)
  - 2D grids (images)
  - 3D grids (movies, CT scans)
  - Data in such a grid layout referred to as **volume**
- Grid data means: neighboring points related
  - Roughly: random permutation along each grid dimension leads to loss of information
  - E.g., in time series: neighboring points are temporally close
  - E.g., in images: neighboring points are spatially close
  - E.g., in videos: neighboring points are spatially/temporally close
  - E.g., not in user-product sales matrix: row and column ordering arbitrary

Grid, volume, parts (1D)

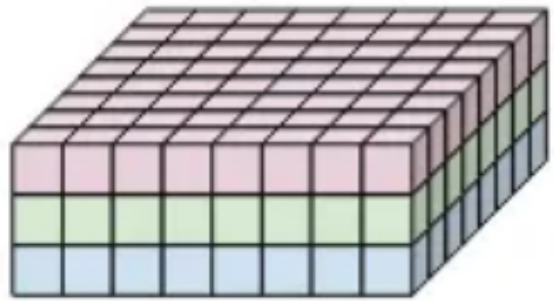
- Part-based view of grid data (cf. lecture 05)
  - Grid points correspond to parts (later also: grid regions)
  - Features correspond to part representations/embeddings
  - Parts and their representations represented in a volume
- Example: 1D grid, 1 feature (e.g, time series)



  - 1 grid dimension (time), 1 feature (altitude)
  - Grid points correspond to parts (minute intervals; ca. 150)
  - Features correspond to part representation (1 per part)
  - Both together form a 150×1 volume
- Example: 1D grid, 3 features (e.g., multivariate time series)
  - Suppose we had three features (altitude, latitude, longitude)
  - Then each part represented by 3 values $\rightarrow 150 \times 3$ volume

Grid, volume, parts (2D)

- Example: 2D grid, 3 features (e.g., images)



  - 2 grid dimensions (width, height), 3 features (red, green, blue)
  - Grid points correspond to parts (pixels; $8 \times 8 = 64$ in total)
  - Features correspond to part representation (3 per part)
  - All together form an $8 \times 8 \times 3$ volume
- Discussion
  - Neighborhood relationship on grid dimensions (parts)
  - But not on feature dimension (convention: last dimension)
  - A feature's values for all parts = **feature map** or **channel** (here: 3, each 8×8)
  - Initial part representations are input
  - (One) goal of CNNs: better (contextualized) part representations

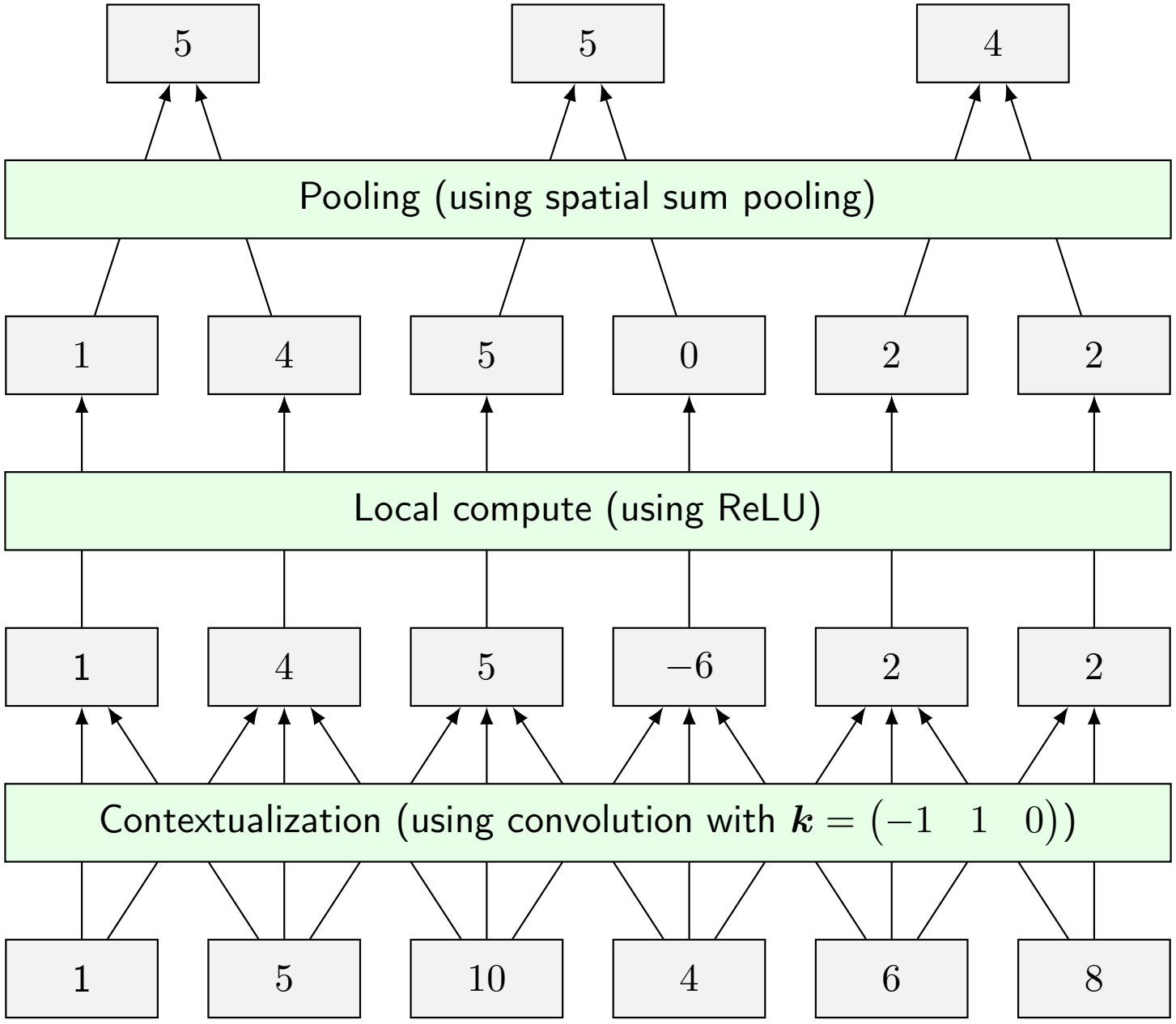
CNNs at a glance (1)

- Key ingredients
  - Part-based modeling (cf. lecture 05)
  - Template modeling
  - Parameter sharing (across parts / grid dimensions)
- Main application areas include
  - Computer vision: e.g., object recognition and classification (characters, persons, traffic signs, ...)
  - Natural language processing: e.g., character-level modeling
  - Signal processing (e.g., audio, time series, . . .)
- Example tasks
  - Part-based tasks (e.g., image segmentation, anomaly detection)
  - Global tasks (e.g., image classification)
  - Also: intermediates (e.g., object detection, forecasting)

CNNs at a glance (2)

- Recall: Three key operations of part-based models (cf. lecture 05)
  1. **Contextualization**: incorporate information from other parts into each part embedding
  2. **Local compute**: update each part embedding individually
  3. **Pooling**: aggregate multiple (or all) part embeddings
- Contextualization operation: convolution
  - Computes contextualized part embeddings
  - Technically: CNN is an FNN with at least one layer that performs a *convolution* operation (instead of matrix multiplication)
- Local compute: typically one of
  - Just an activation function (traditional CNNs)
  - Feature pooling / MLP / 1×1 convolution (modern CNNs)
- Pooling operation
  - Reduce number of parts: drop parts ("stride"), spatial pooling
  - Readout: MLP

Example CNN


5
5
4
Pooling (using spatial sum pooling)
1
4
5
0
2
2
Local compute (using ReLU)
1
4
5
−6
2
2
Contextualization (using convolution with $\boldsymbol{k} = \begin{pmatrix} -1 & 1 & 0 \end{pmatrix}$)
1
5
10
4
6
8
Part 1
Part 2
Part 3
Part 4
Part 5
Part 6

Why not use MLPs?

Drawbacks of MLPs for grid data:

1. Too many parameters
   - E.g., consider a $1000 \times 1000 \times 3$ grid
   - MLP has $3\,\mathrm{M}$ parameters per output neuron (!)
2. Locality (i.e., neighborhood relationships) not exploited
   - MLP unaware of grid structure of the data
   - In fact: training MLP on (consistently) permuted data $\rightarrow$ same result
3. No translation invariance
   - **Translation invariance**: translation of input does not affect output
   - E.g., in image classification: translated object $\rightarrow$ same class
   - MLP must individually learn to recognize a given object at different positions
4. Cannot handle inputs of varying sizes
   - CNNs (sometimes) can do this

We will see: Inductive bias of CNNs helps to mitigate these points.

Outline

Background: Convolution

- A **convolution** is defined as

$$y(t) = (x * \kappa)(t) = \int_{-\infty}^{+\infty} x(\tau)\kappa(t-\tau)\,\mathrm{d}\tau$$

- Involved functions
  - **Input signal** $x : \mathbb{R} \to \mathbb{R}$ (e.g., time $\to$ feature)
  - **Kernel** (**filter**) $\kappa : \mathbb{R} \to \mathbb{R}$ (e.g., age $\to$ weight)
  - **Output signal** y (e.g., time $\to$ contextualized feature)
  - Note: $*$ is commutative; first/second argument by convention
- Interpretation for a temporal signal (t and τ are "times")
  - t is time in output signal (e.g., 20)
  - τ is time in input signal (e.g., 15)
  - $t - \tau$ is (signed) age of input at output (e.g., 5)
  - $\kappa(t-\tau)$ is weight of time-τ input at time-t output
    $\to$ Depends only on age of input
- For each t, computes a "weighted sum" of all inputs x, where the weight of $x(\tau)$ depends its age $a = t - \tau$ via $\kappa(a)$

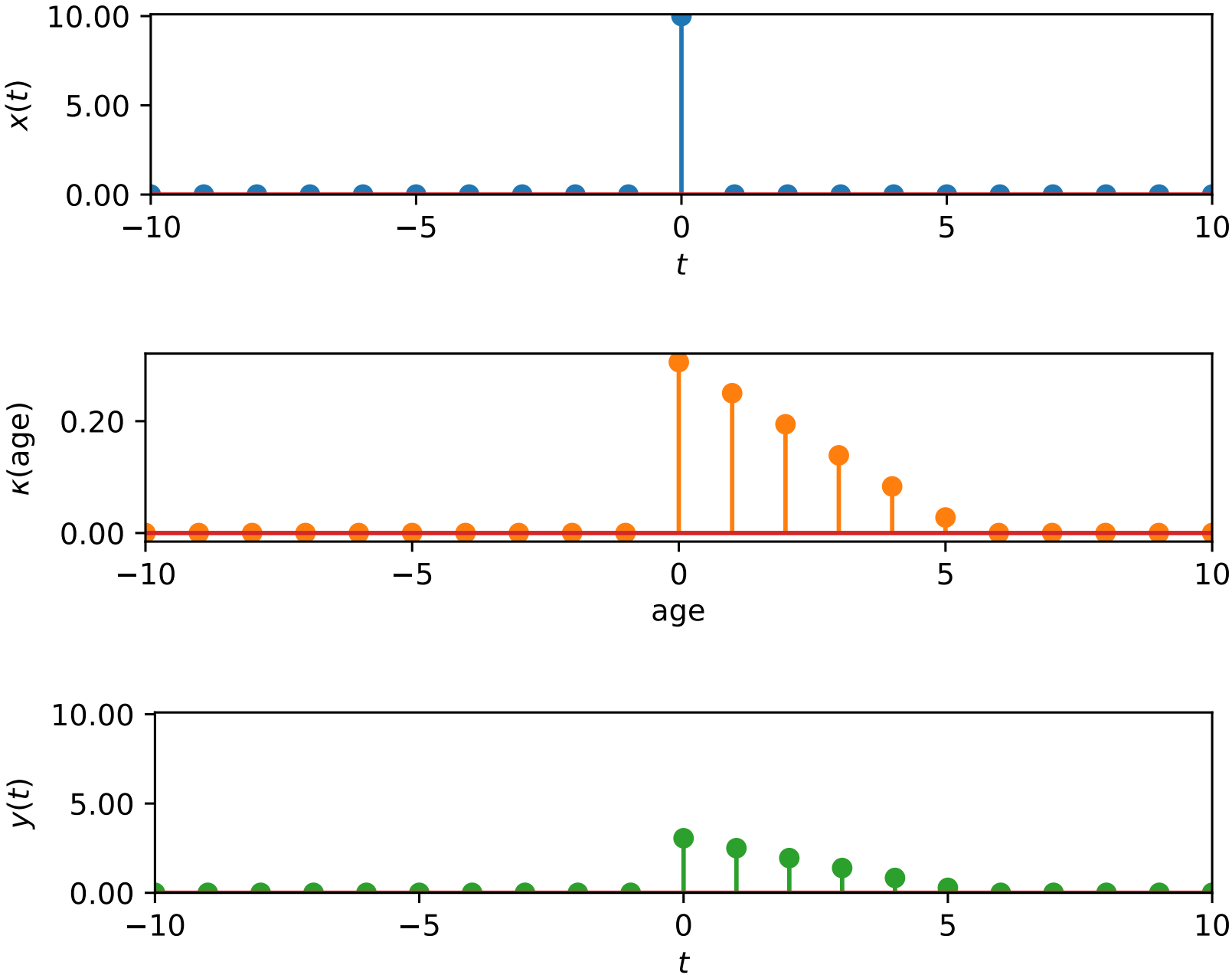
Discrete convolution

- A **discrete convolution** is defined as

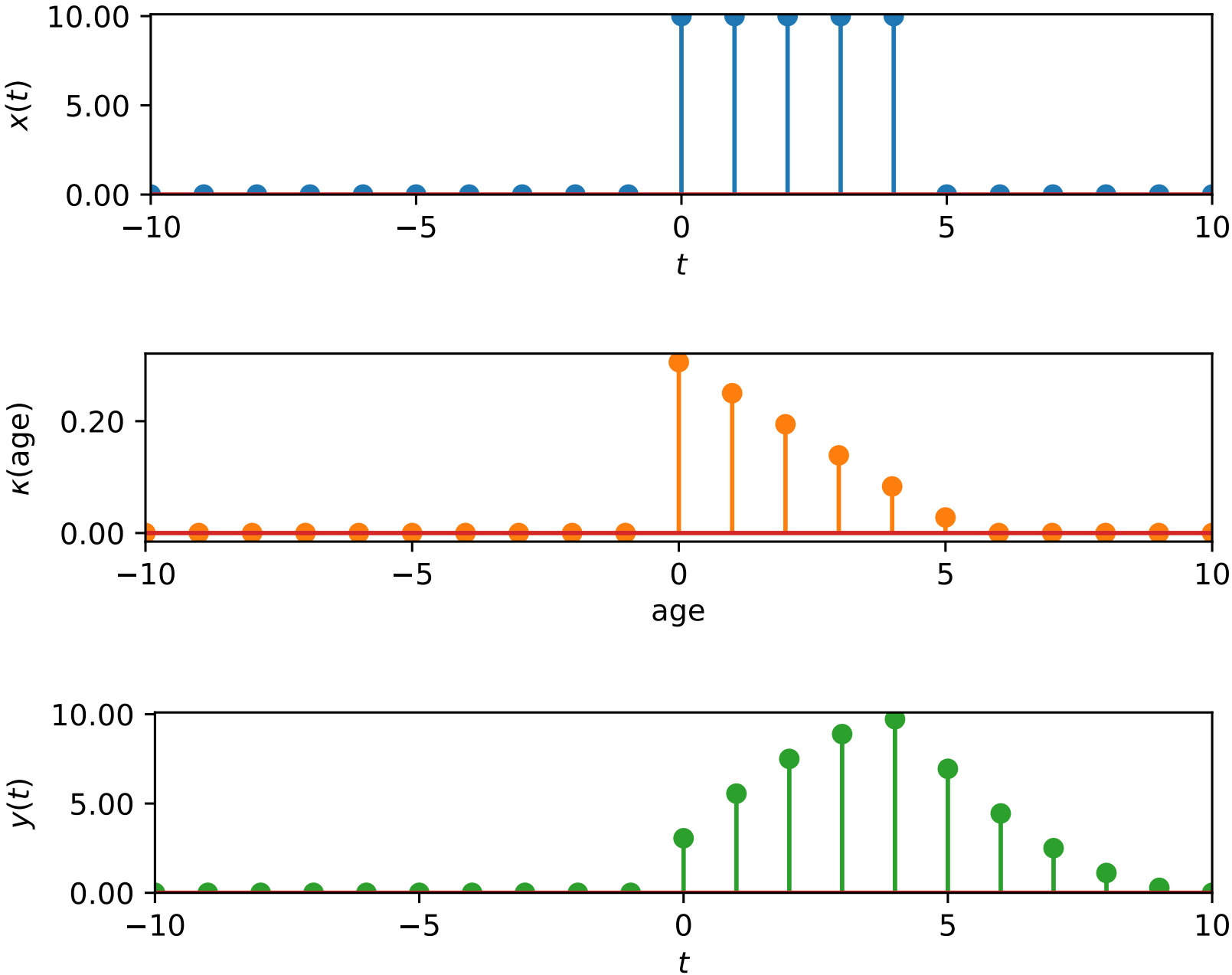
$$y(t) = (x * \kappa)(t) = \sum_{\tau=-\infty}^{+\infty} x(\tau)\kappa(t-\tau)$$

- Involved functions
  - **Input signal** $x : \mathbb{Q} \to \mathbb{R}$ (e.g., position $\to$ feature)
  - **Kernel** (**filter**) $\kappa : \mathbb{Q} \to \mathbb{R}$ (e.g., age $\to$ weight)
  - **Output signal** y (e.g., position $\to$ contextualized feature)
  - Note: $*$ is commutative; first/second argument by convention
- Interpretation for a temporal signal (t and τ are "positions")
  - t is position in output signal (e.g., 20)
  - τ is position in input signal (e.g., 15)
  - $t - \tau$ is (signed) age of input at output (e.g., 5)
  - $\kappa(t-\tau)$ is weight of position-τ input at position-t output
    $\to$ Depends only on age of input
- For each t, computes **a weighted sum** of all inputs x, where the weight of $x(\tau)$ depends its age $a = t - \tau$ via $\kappa(a)$

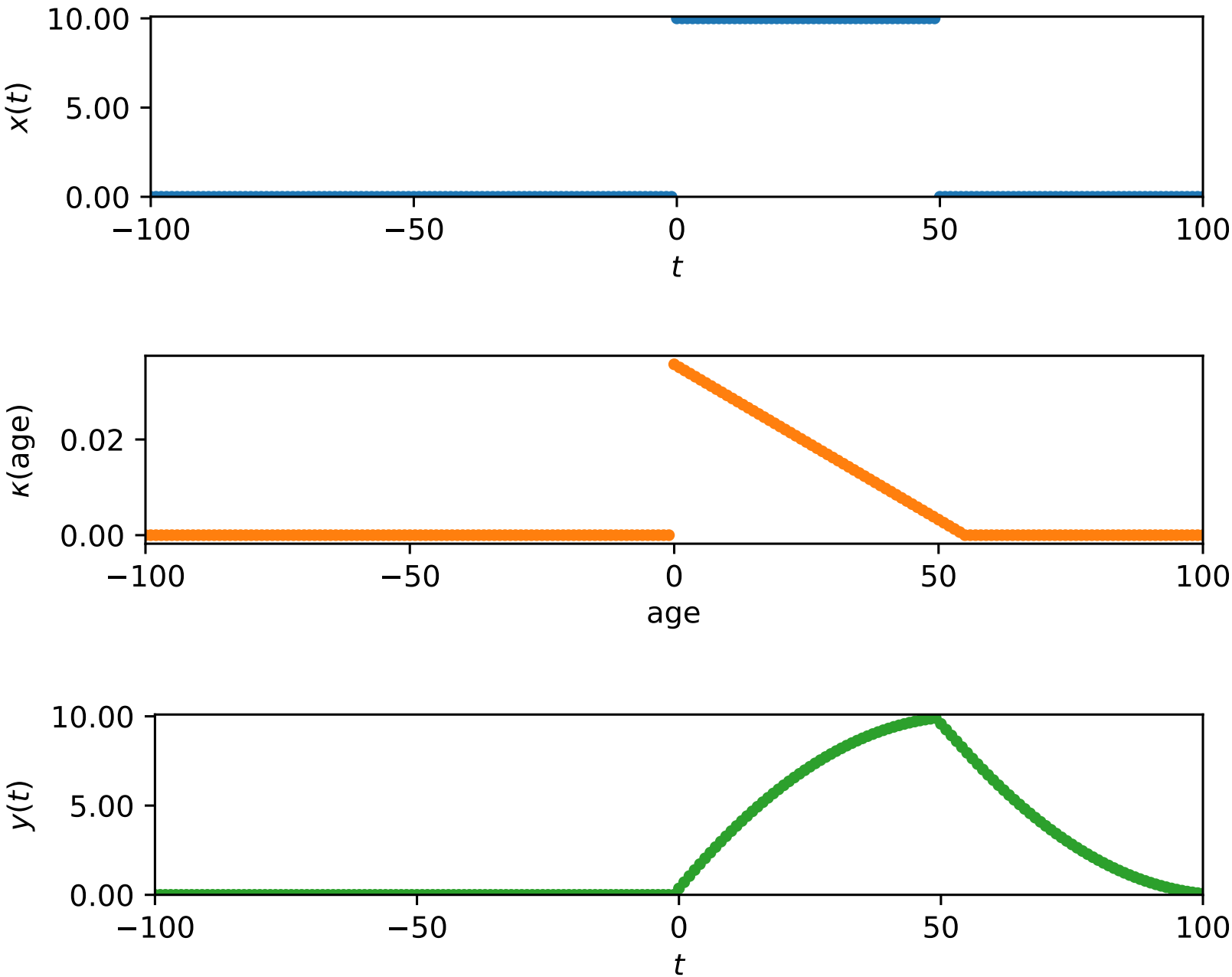
Discrete convolution example (1)



Discrete convolution example (2)



Discrete convolution example (3)

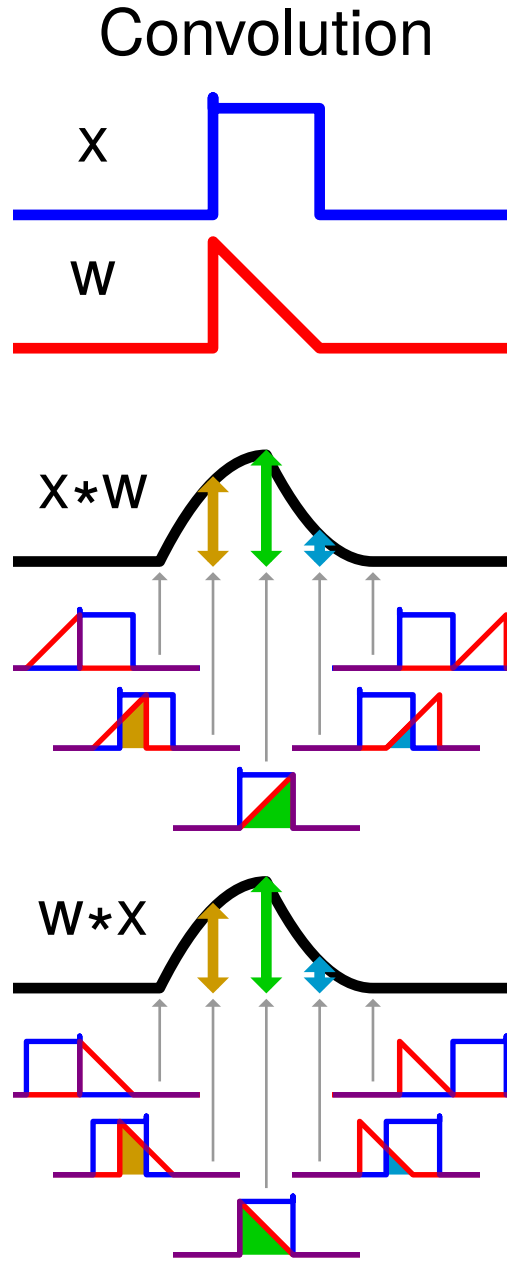


Discrete cross-correlation

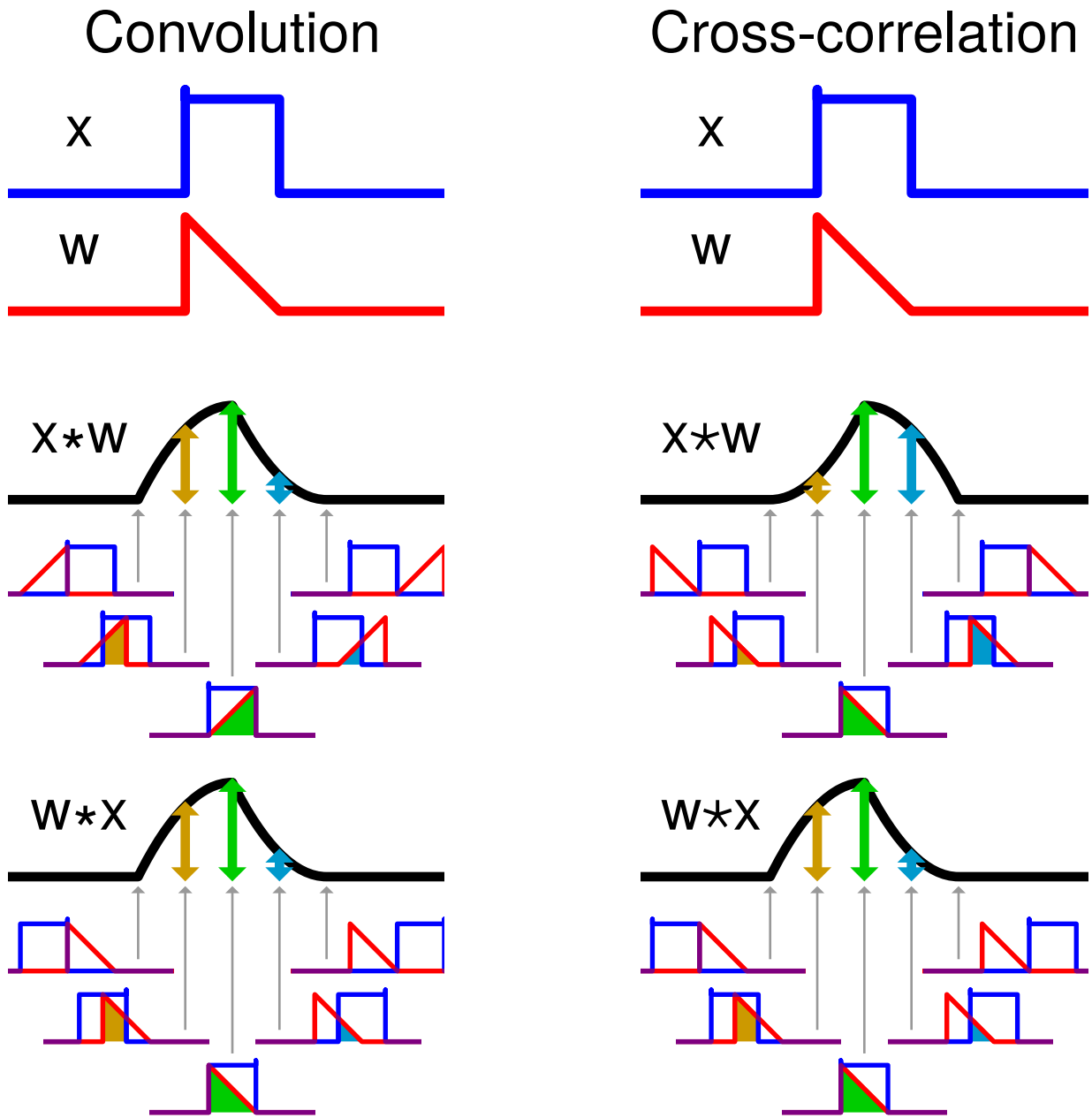
- Another interpretation
  - Flip the kernel
  - Slide the flipped kernel over the input signal
  - Take point-wise products and integrate
- Related: **discrete cross-correlation**

$$y(t) = (x \star \kappa)(t) = \sum_{\tau=-\infty}^{\infty} x(\tau)\kappa(\tau - t)$$

  - Observe: $\kappa(\tau - t) = \kappa(-(t - \tau)) = \kappa(-\text{age})$
  - Interpretation as above, but *without kernel flipping*
  - Not commutative anymore
- That's used in CNNs; called convolution *without kernel flipping* or simply convolution
- In CNNs, κ is learned $\rightarrow$ no conceptual difference

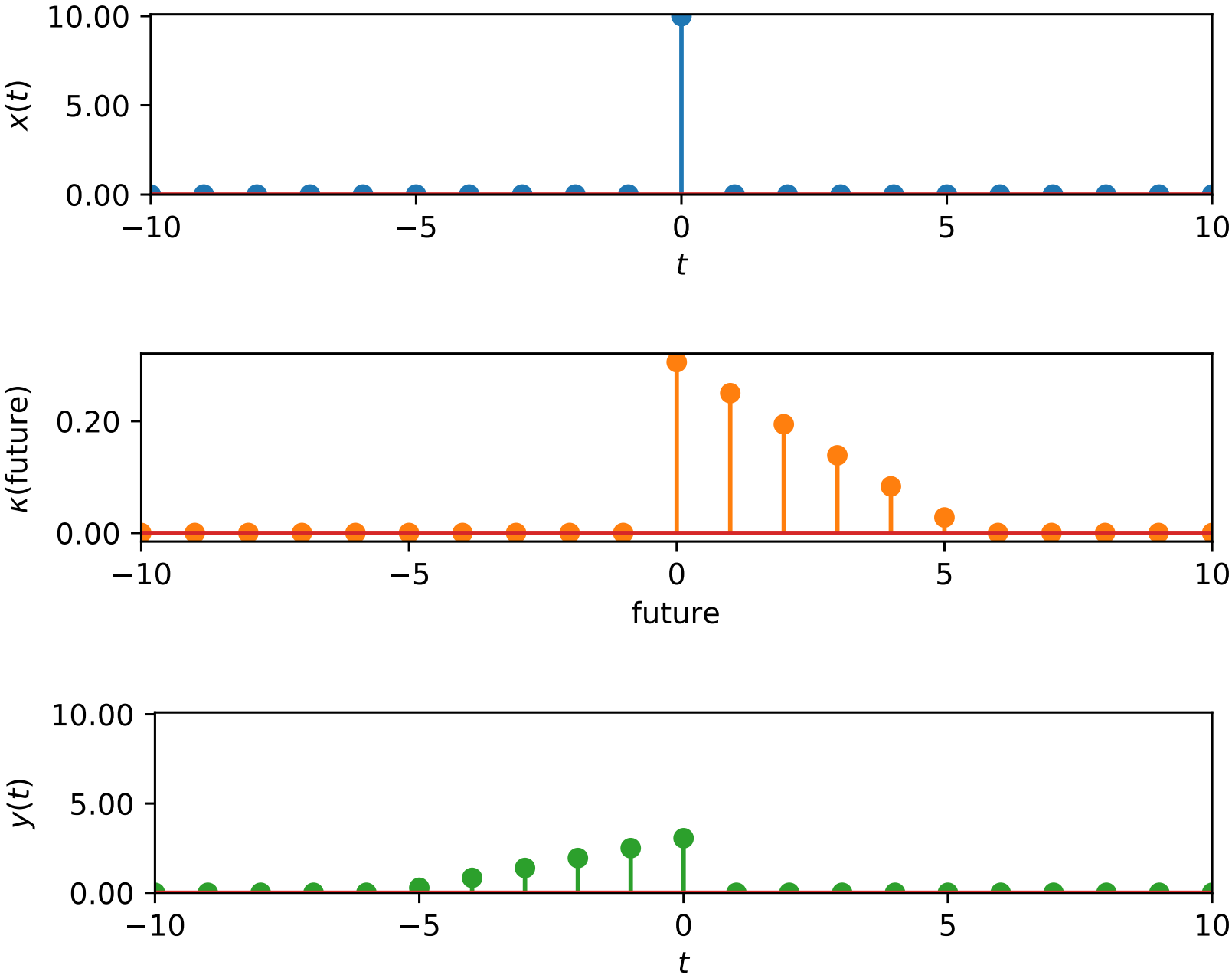


Convolution and cross-correlation

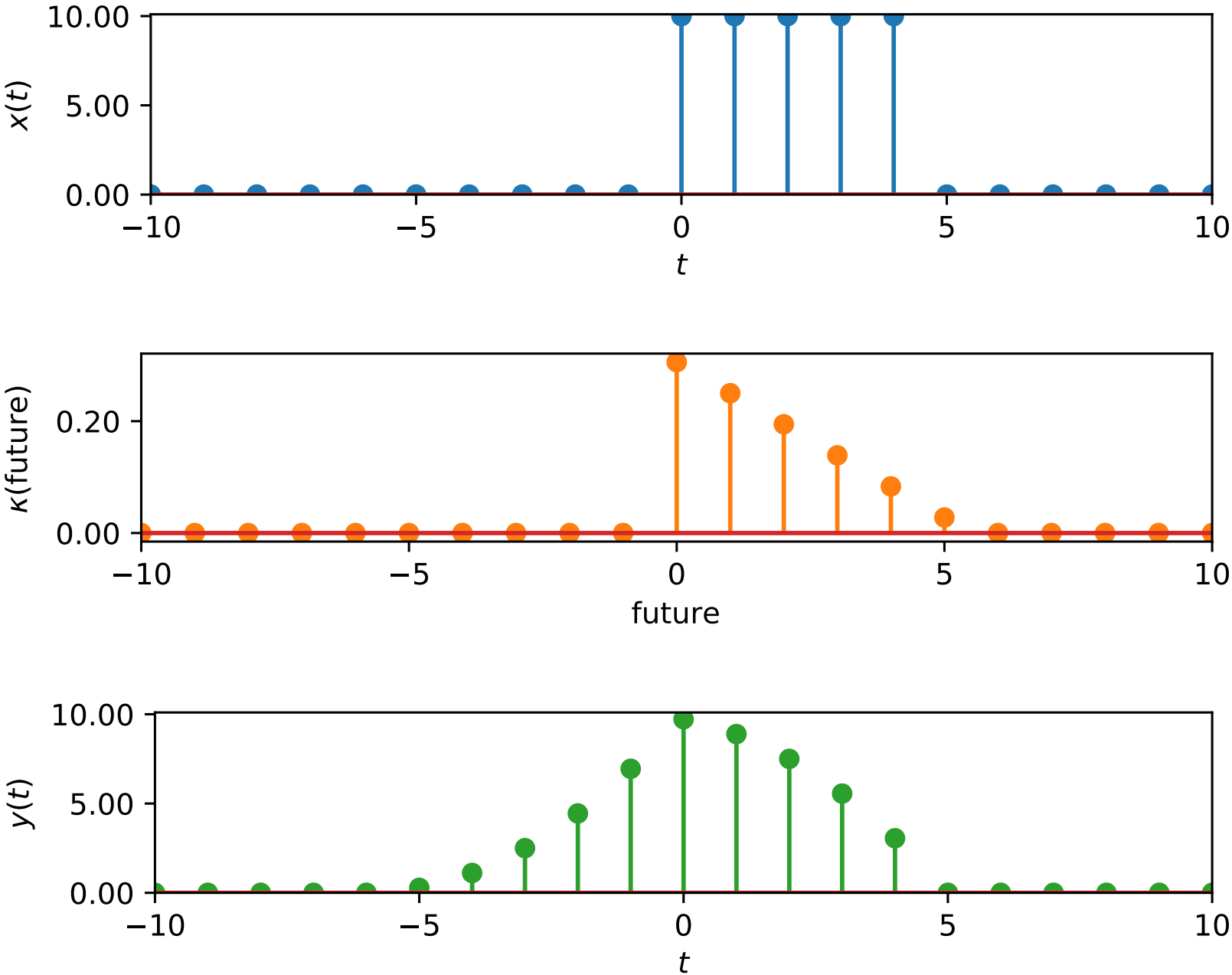


Wikipedia

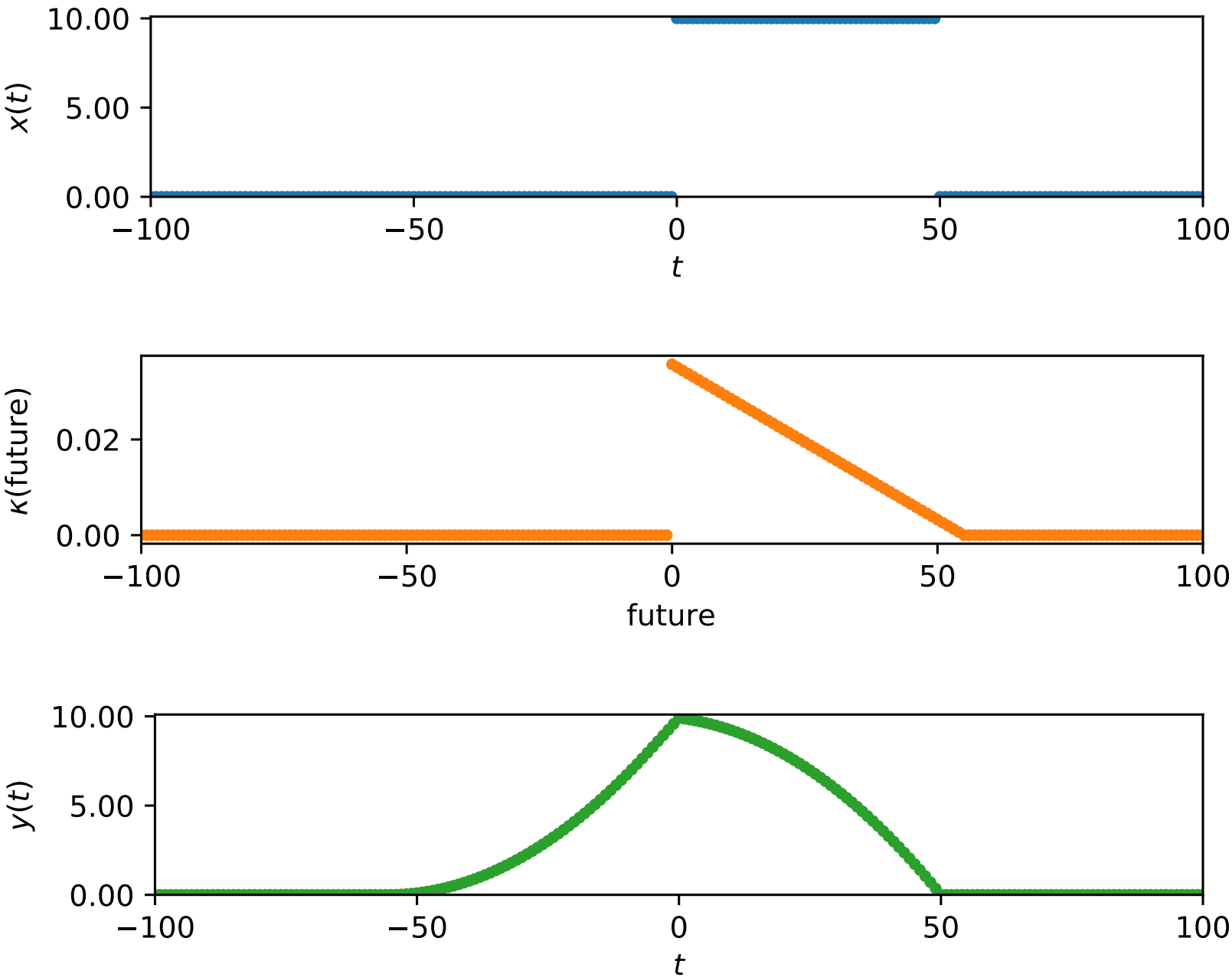
Discrete cross-correlation (1)



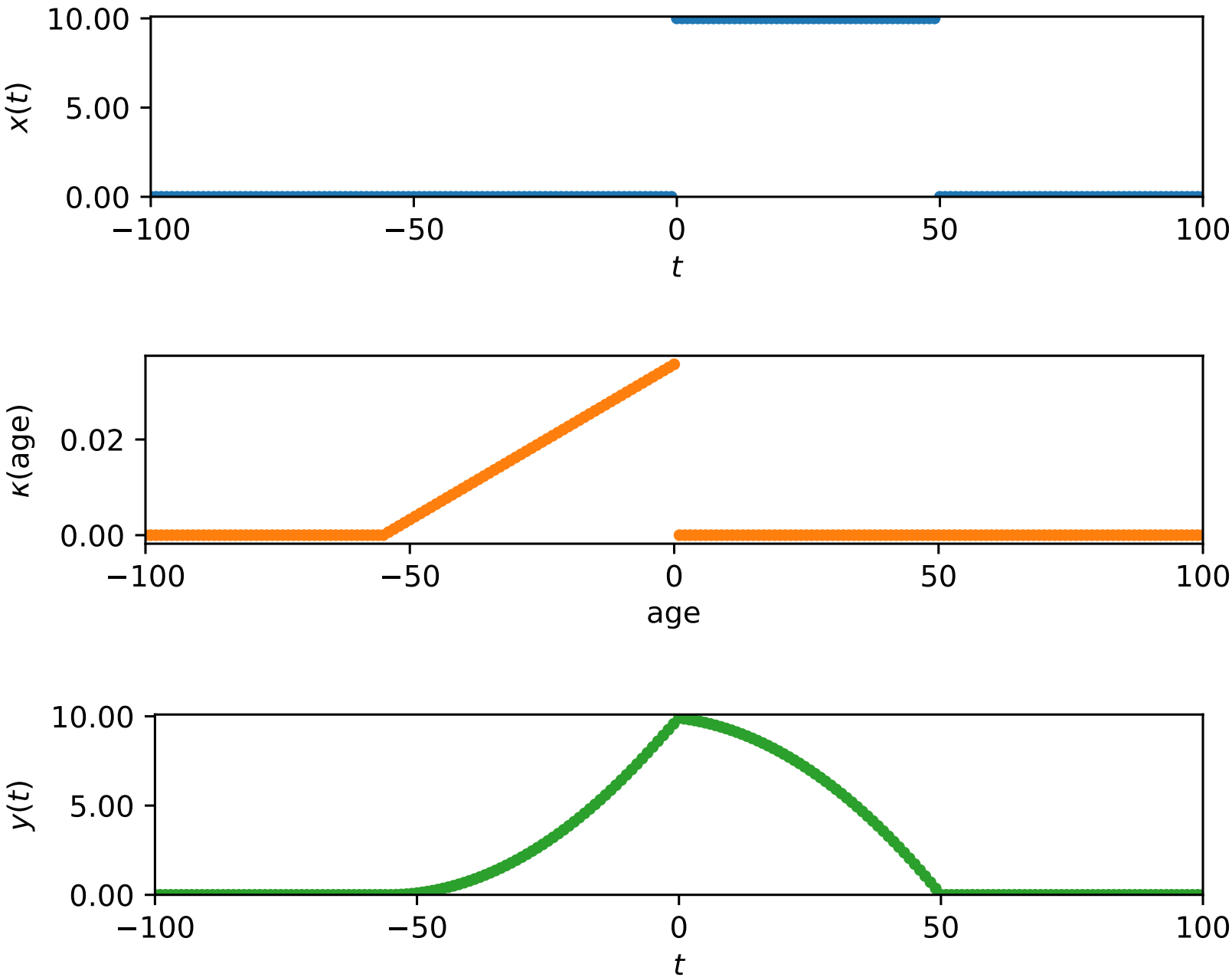
Discrete cross-correlation (2)



Discrete cross-correlation (3)



Discrete convolution after flipping kernel



Outline

Vector representation

- Let's rewrite discrete cross-correlation

$$y(t) = (x \star \kappa)(t) = \sum_{\tau=-\infty}^{\infty} x(t+\tau)\kappa(\tau)$$

- In ML, input x, kernel κ, and output y often represented by vectors $\boldsymbol{x}$, $\boldsymbol{k}$, and $\boldsymbol{y}$, resp.
  - E.g., $\boldsymbol{x} = \begin{pmatrix} 10 & 20 & 15 \end{pmatrix}^\top$
  - Corresponding function: $x(1) = 10$, $x(2) = 20$, $x(3) = 15$
- Different boundary conditions are possible $\rightarrow$ **padding**
  - E.g., what is $x(t)$ for $t \notin \{1, 2, 3\}$?
  - **Zero padding** $\rightarrow$ all 0
  - **Reflection padding**: mirror data at the boundaries (e.g., $x(4) = 15$, $x(5) = 20$, $x(6) = 10$)
- Useful: Interpret kernel as centered and zero-padded
  - E.g., $\boldsymbol{k} = \begin{pmatrix} 10 & 20 & 15 \end{pmatrix}$
  - Corresponding function: $\kappa(-1) = 10$, $\kappa(0) = 20$, $\kappa(1) = 15$

Discrete cross-correlation, vector form

- Let's suppose $\boldsymbol{k}$ has **kernel size** $K = 2W + 1$
  - $\kappa(\tau) = 0$ for $\tau < -W$ and $\tau > W$
- Discrete cross-correlation then becomes

$$y(t) = (x \star \kappa)(t) = \sum_{\tau=-W}^{W} x(t+\tau)\kappa(\tau)$$

  - Now a finite sum $\rightarrow$ easy to compute
- Example: $\boldsymbol{k} = \begin{pmatrix} 1 & 0 & -1 \end{pmatrix}$ of size $K = 3$ $(W = 1)$

Input $\boldsymbol{x}$	0	1	2	4	8	4	2	1	0
Output $\boldsymbol{y}$		-2	-3	-6	0	6	3	2	

- As discussed: Output obtained by sliding the kernel over the input, taking element-wise products, and summing up
  - E.g., for output $y_2 = -3$

Input $\boldsymbol{x}$	0	1	2	4	8	4	2	1	0
Kernel $\boldsymbol{k}$		1	0	-1					
Output $\boldsymbol{y}$		-2	-3	-6	0	6	3	2	

  - Observe: only a region of size $K = 3$ from input accessed

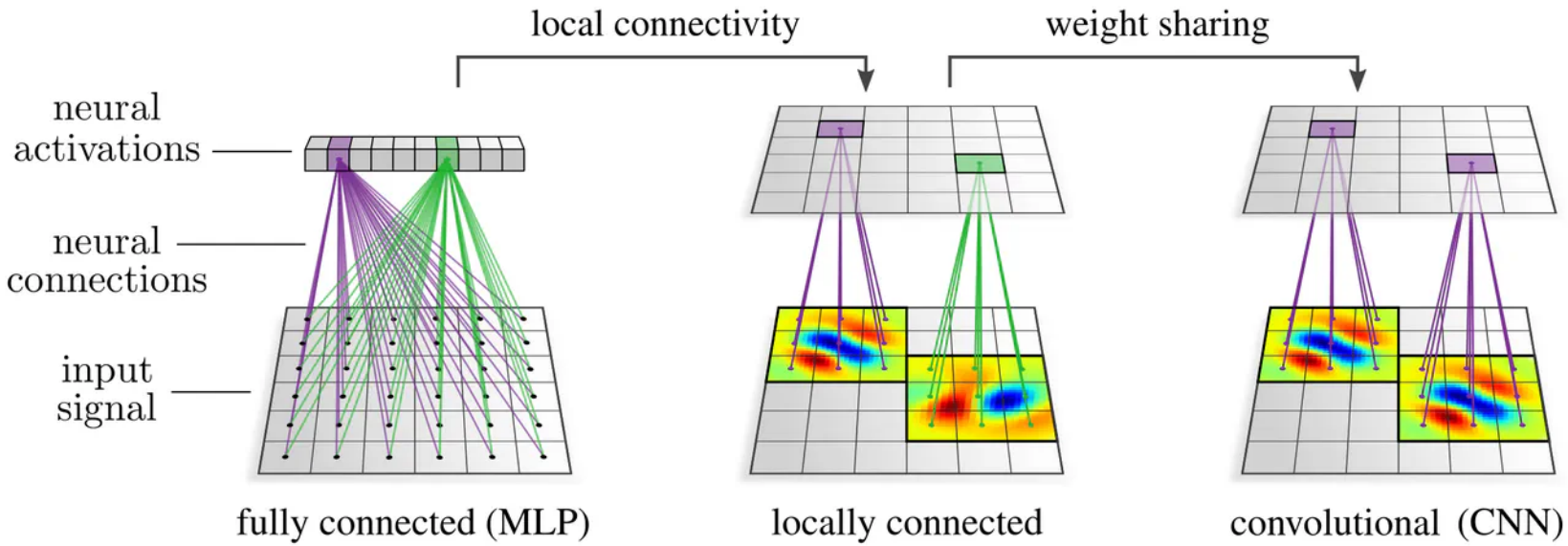
Discrete cross-correlation, tensor form

- For higher-dimensional grids, inputs, kernels, and outputs are multivariate functions
  - E.g., for a 2D grid

$$y(t_1, t_2) = \sum_{\tau_1=-W_1}^{W_1} \sum_{\tau_2=-W_2}^{W_2} x(t_1 + \tau_1, t_2 + \tau_2)\kappa(\tau_1, \tau_2)$$

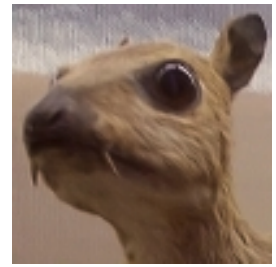
  - Kernel size is now $K_1 \times K_2 = (2W_1 + 1) \times (2W_2 + 1)$ $\rightarrow$ **kernel matrix**
  - We now slide this matrix over the input, take element-wise products, and sum up
  - The corresponding operation is called a $K_1 \times K_2$ **convolution**
- For a grid of D dimensions, **kernel tensor** also has D grid dimensions
  - Likewise, input tensor and output tensor

MLP vs. convolution layer at a glance

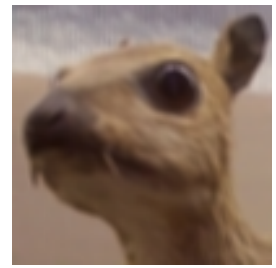


Example: 3×3 convolutions

Identity $\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$

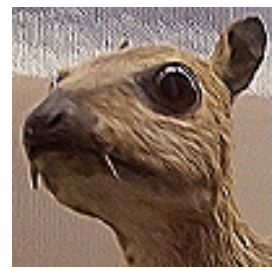


Box blur $\frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$



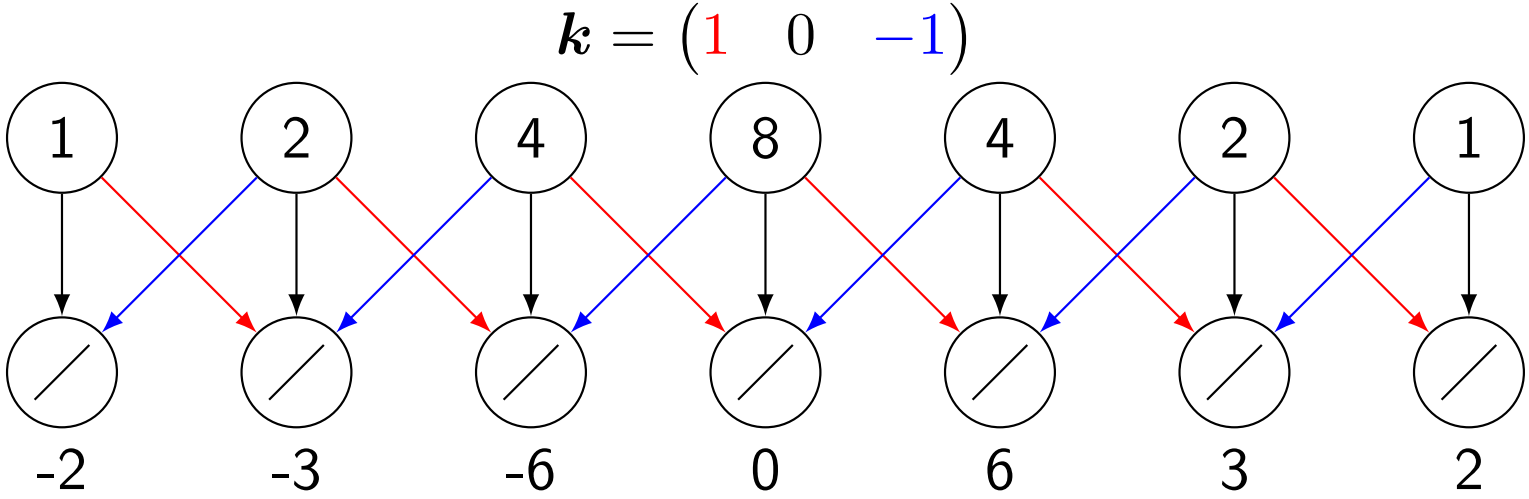
Edge detection $\begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}$

Sharpen $\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$



Convolutions as FNNs

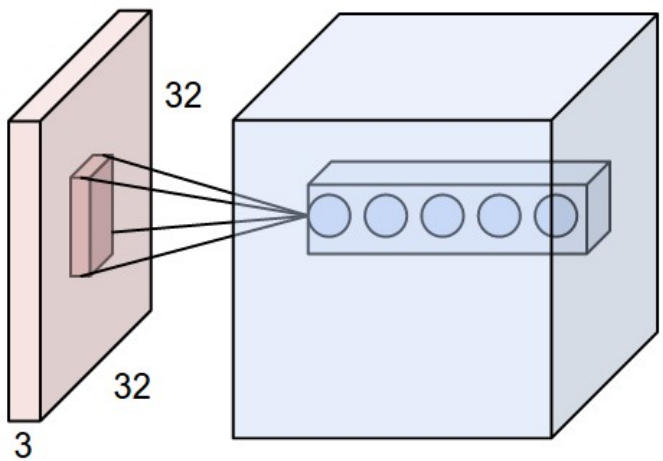
- We can represent a convolution as a linear FNN



- Weights (=kernel matrix) shared across all neurons
  $\rightarrow$ parameter sharing
- Kernel moved systematically over data
  $\rightarrow$ template model
- Kernel touches few inputs
  $\rightarrow$ sparsity
- Inputs are spatially close
  $\rightarrow$ locality

Convolutional layers

- CNNs are composed of **convolutional layers** + nonlinearity
  - Performs *multiple* convolutions, each with a different kernel
  - I.e., produces multiple feature maps (also called **channels**)
  - Generally: takes in **input volume**, produces an **output volume**
- Example: first convolutional layer for an image
  - Input: image width (32) $\times$ image height (32) $\times$ RGB colors (3)
  - Output: image width (32) $\times$ image height (32) $\times$ feature map (5)
  - Convolution performed w.r.t. first two dimensions (next slide)
  - E.g., for 7×7 convolution: layer parameterized by 5 kernel tensors $\boldsymbol{\mathcal{K}}_1, \ldots, \boldsymbol{\mathcal{K}}_5 \in \mathbb{R}^{7 \times 7 \times 3}$



Convolutional layers (details)

- In general, we have $F \geq 1$ input channels
  - Use a different kernel for each feature map and sum up (observe: no "sliding" across feature dimension)
  - E.g., for a 2D grid

$$y(t_1, t_2) = \sum_{f=1}^{F} \sum_{\tau_1=-W_1}^{W_1} \sum_{\tau_2=-W_2}^{W_2} x(t_1 + \tau_1, t_2 + \tau_2) \kappa_f(\tau_1, \tau_2)$$

- Entire operation be represented by a kernel tensor
  - E.g., 7×7 convolution with $F = 3$ features
  - 3 kernel matrices, one per feature: $\boldsymbol{K}_1, \ldots, \boldsymbol{K}_3 \in \mathbb{R}^{7 \times 7}$
  - Or 1 kernel tensor: $\boldsymbol{\mathcal{K}} \in \mathbb{R}^{7 \times 7 \times 3}$
- For $O \geq 1$ output channels, repeat O times with different convolutions
  - E.g., 5 output channels
  - Use 5 kernel tensors $\boldsymbol{\mathcal{K}}_1, \ldots, \boldsymbol{\mathcal{K}}_5 \in \mathbb{R}^{7 \times 7 \times 3}$
  - Or one of shape $\mathbb{R}^{7 \times 7 \times 3 \times 5}$
- This is the operation of **convolutional layers** in CNNs
  - Here: $7 \cdot 7 \cdot 3 \cdot 5 = 735$ parameters

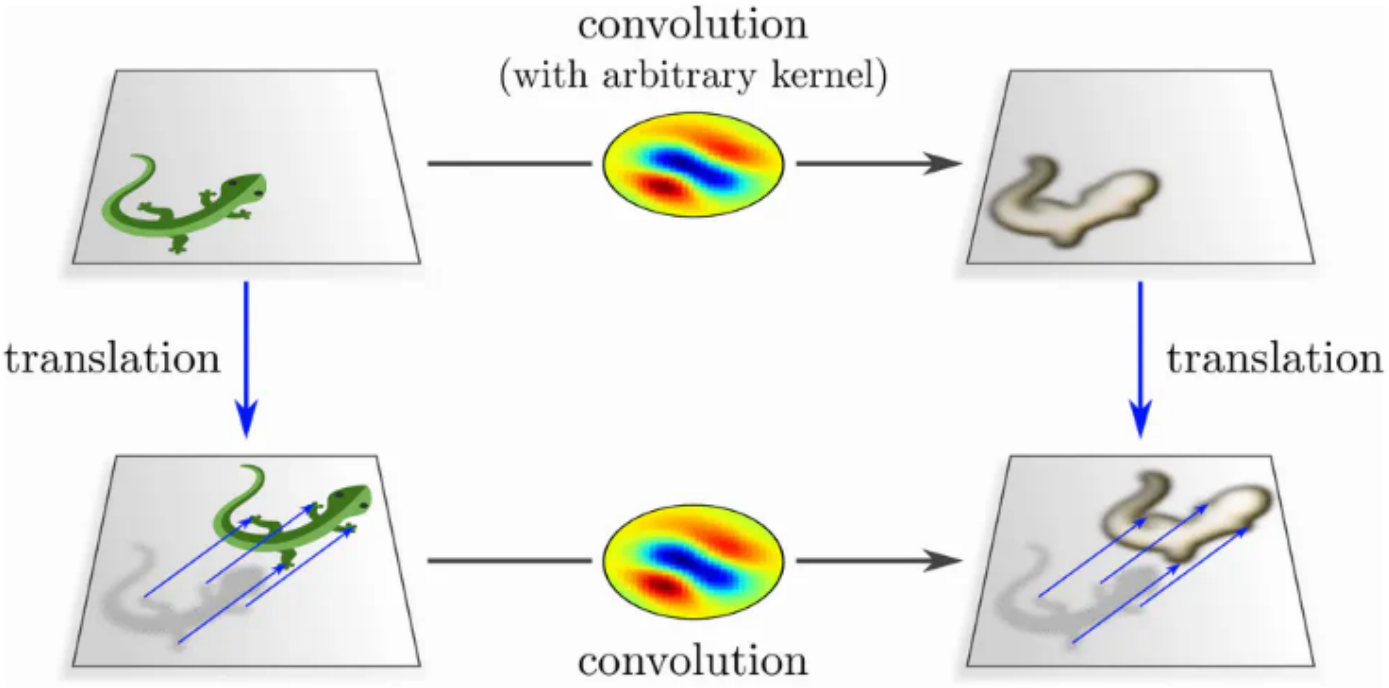
Discussion

- In convolutional layer, the kernel values are parameters
  $\rightarrow$ learned
- We can think of the kernel as **feature detector**
  - Feature = inner product of input and kernel
    (vectorized and shifted appropriately)
  - Kernel is *learned* $\rightarrow$ learned feature detector
  - May use our standard interpretations of inner product
  - E.g., interpret feature as: Is the input *similar* to the kernel?
- In part-based view, convolution corresponds to contextualization
  - Input: part embeddings
  - Output: contextualized part embeddings via convolutional layer
  - Observe: contextualized part embeddings depends on nearby part embeddings only (within kernel size)
  - Observe: contextualized part embeddings represent nearby grid points as well
- In CNNs, convolutions are followed by a nonlinearity (e.g., ReLU)
  - Often part of convolutional layers
  - In our part-based model, corresponds to local compute

Translation equivariance

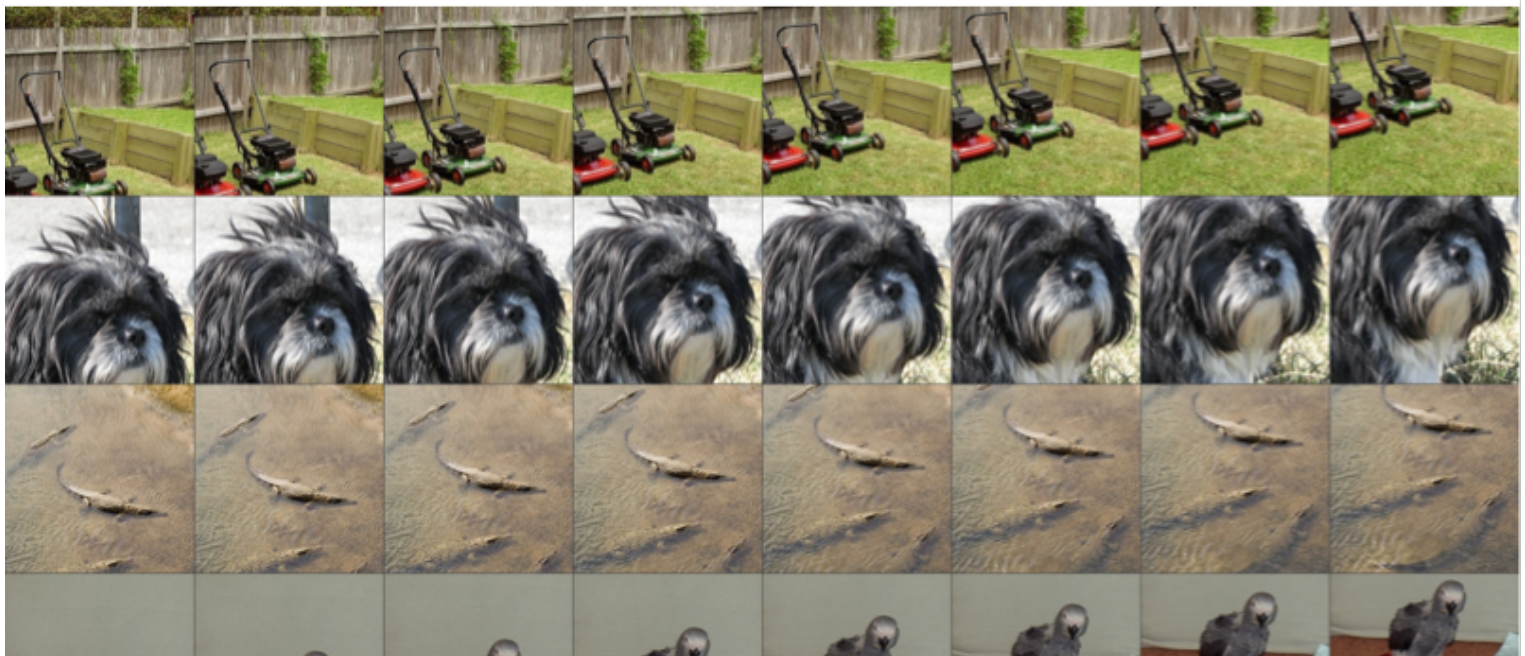
The convolution operation is **translation equivariant**.

- Means: translated input $\rightarrow$ translated output



Inductive bias: stationarity

- (One) inductive bias of convolution $\rightarrow$ **stationary data**
  - Means: distribution of values in a particular region is independent of where the region is located along the grid
- That's reflected in the feature detector of a convolutional layer
  - Due to the translation equivariance property, feature detection is also independent of location
  - I.e., we detect features independent of where they are
- E.g., in object detection, we want to detect an object irrespective of its absolute position



Outline

Local compute in CNNs

Local compute operations usually done as follows:

1. Use a non-linearity (as discussed)
2. Use a linear layer
   - Input: D-dimensional part representation $\boldsymbol{p}_{\text{in}} \in \mathbb{R}^D$
   - Output: updated C-dimensional part representation $\boldsymbol{p}_{\text{out}} \in \mathbb{R}^C$
   - Via a linear layer (applied to each grid point / part)

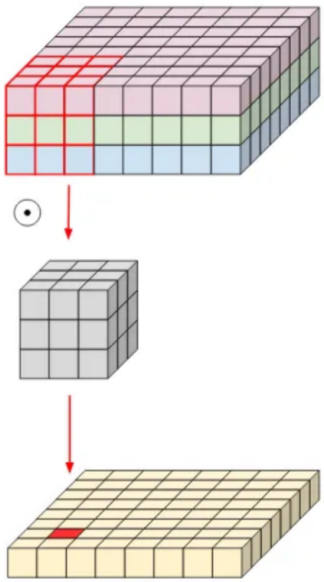
     $$\boldsymbol{p}_{\text{out}} = \boldsymbol{W}^\top \boldsymbol{p}_{\text{in}},$$

     where $\boldsymbol{W} \in \mathbb{R}^{D \times C}$ is the weight matrix
   - $C < D \rightarrow$ decreased embedding size
   - $C > D \rightarrow$ increased embedding size
   - For 2D grids, equivalent to a 1×1-**convolution** with weights $\boldsymbol{W}$ (used as kernel)
   - Often: term 1×1-convolution used for grids of other dimensionality as well (instead of, say, $1 \times 1 \times 1$ convolution for a 3D grid)
3. Perform **feature pooling** (more later)

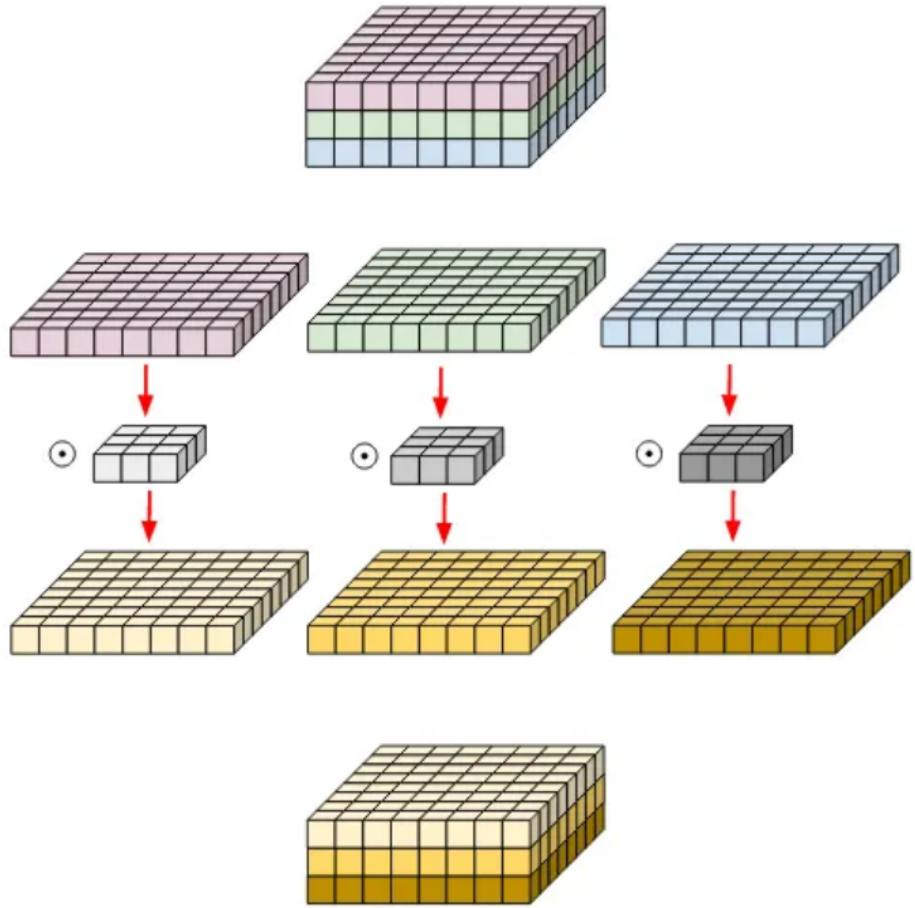
*Depth-wise separable convolution

- **Depth-wise separable convolution** layers exploit that translation equivariance is not needed across channels
  1. **Depth-wise convolution**: take in *input volume*, produce an *intermediate output volume*, where each channel of the intermediate output volume is computed from exactly one channel of the input volume (H channels in $\rightarrow$ H channels out)
  2. Apply a 1×1 **convolution**
- Example from sl. 29: first convolutional layer for an image
  - Input: image width (32) $\times$ image height (32) $\times$ RGB colors (3)
  - Output: image width (32) $\times$ image height (32) $\times$ feature map (5)
  - E.g., for d7×7 convolution: layer parameterized by 3 kernel matrices $\boldsymbol{K}_1, \ldots, \boldsymbol{K}_3 \in \mathbb{R}^{7 \times 7}$ plus "weight matrix" $\boldsymbol{W} \in \mathbb{R}^{3 \times 5}$
  - This reduces the amount of model parameters significantly
    - $7 \times 7 \rightarrow 7 \cdot 7 \cdot 3 \cdot 5 = 735$ parameters
    - d$7 \times 7 \rightarrow 7 \cdot 7 \cdot 3 + 3 \cdot 5 = 162$ parameters
- Used in many recent architectures (e.g. ConvNext; also: MobileNet, EfficientNet)

*2D convolution vs. depth-wise convolution



2D convolution



Depth-wise convolution

For depth-wise *separable* convolution, add 1×1 convolution to depth-wise convolution (not shown here).

Outline

Spatial pooling

Convolutional layers periodically followed by pooling operations.

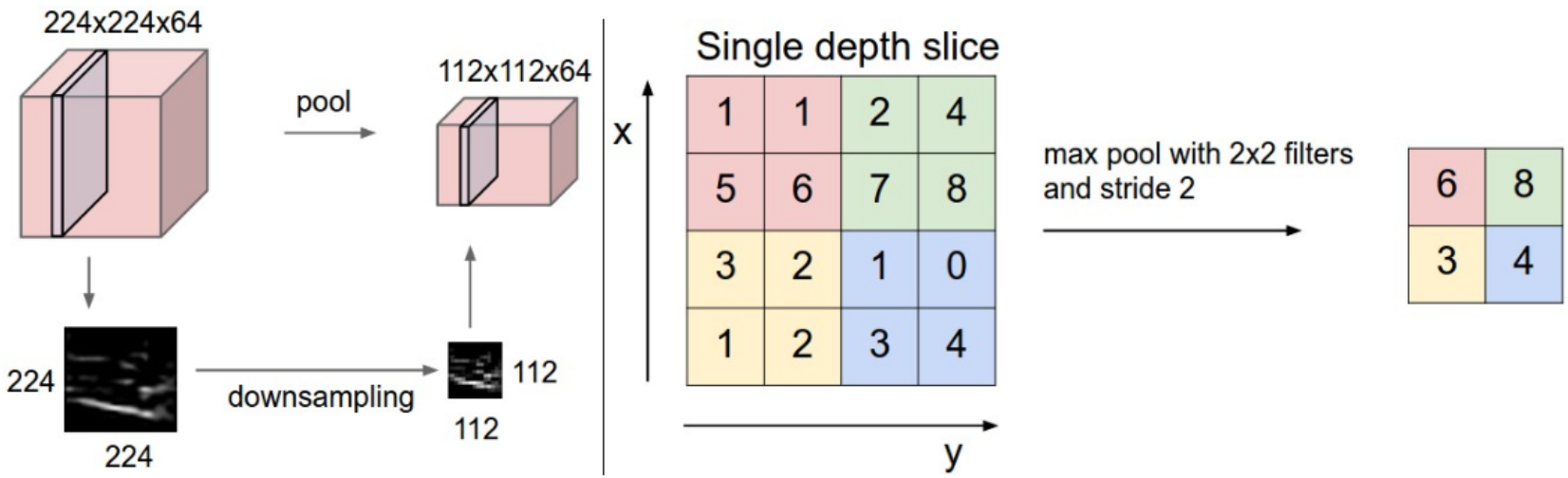
1. Drop parts
   - E.g., in 1D grid, keep only every k-th part (and drop others)
   - Corresponds to a convolution operation with **stride** k (equivalent, but more efficient)
   - Stride = steps in which to slide kernel (set separately for each grid dimension)
2. Use pooling layers (next slide)

Both operations decrease **spatial resolution**.

- I.e., grid size reduced
- E.g., 32×32, then conv. layer with stride $2 \times 2 \rightarrow 16 \times 16$
- Resulting parts then represent regions (here: 2×2 regions)

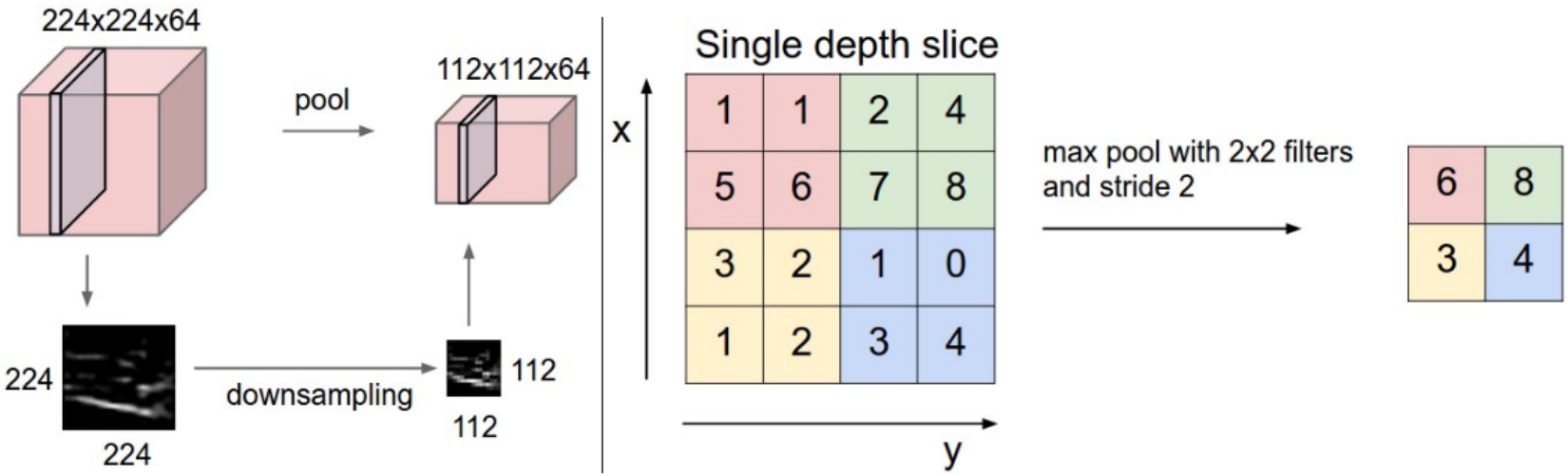
Pooling layers

- **Pooling layers**
  - Like a convolution, but hard-coded, possibly non-linear operation (e.g., **max pooling**, **avg pooling**)
  - Groups together ("pools") the values in a region
  - Reduces size: e.g., stride = no. pooled values (per dimension)
  - Sometimes referred to as **subsampling**
- Can be performed along spatial dimension and/or feature dimensions
- Example: spatial max-pooling



Invariance

- **Translation invariance**
  - Means: translated input $\rightarrow$ same output
  - Desired in tasks such as image classification
  - E.g., whether or not an image contains a flower (=class) does not depend on the location of the flower
- Spatial pooling operations increase translation invariance (slightly) $\rightarrow$ inductive bias



- Likewise, feature pooling increases **feature invariance**

Outline

Example 1: LeNet5

- LeNet5 is a well-known early architecture (1998)
- Used for digit/letter classification

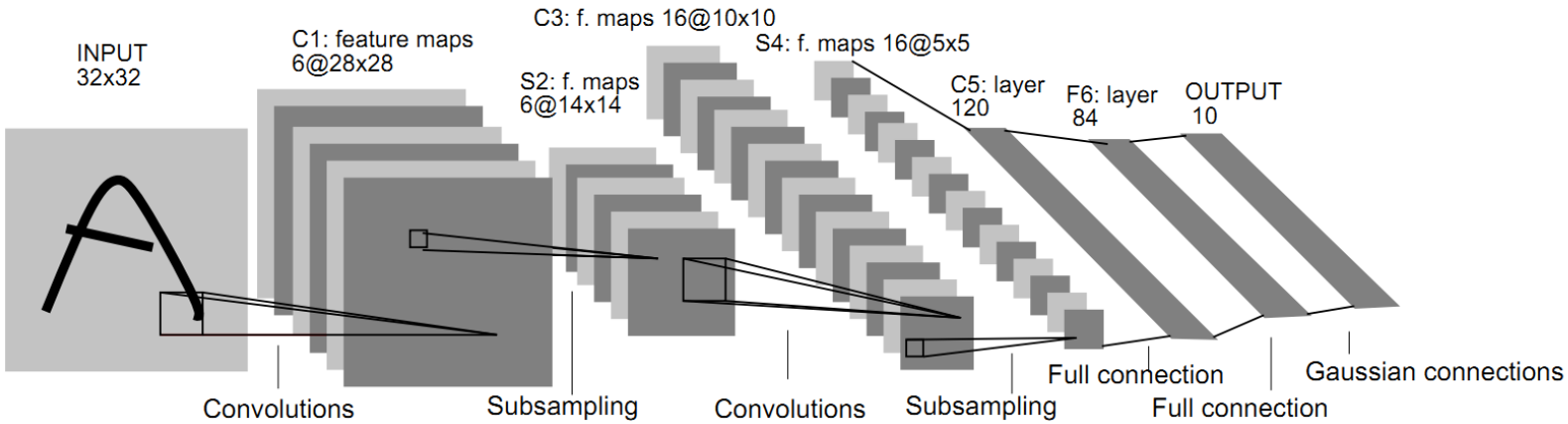
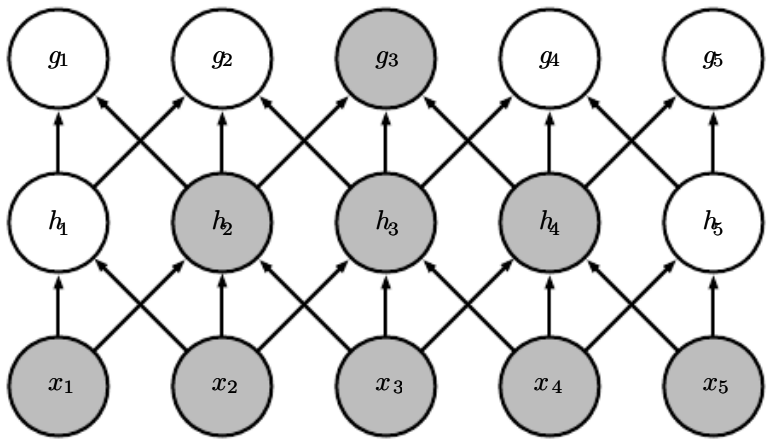


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

- (Scaled) tanh units almost everywhere
- Subsampling $= 2 \times 2$ average pooling with stride 2
- Output computes distance to manually-crafted image (one per class) $\rightarrow$ *radial basis function (RBF)*

Receptive fields

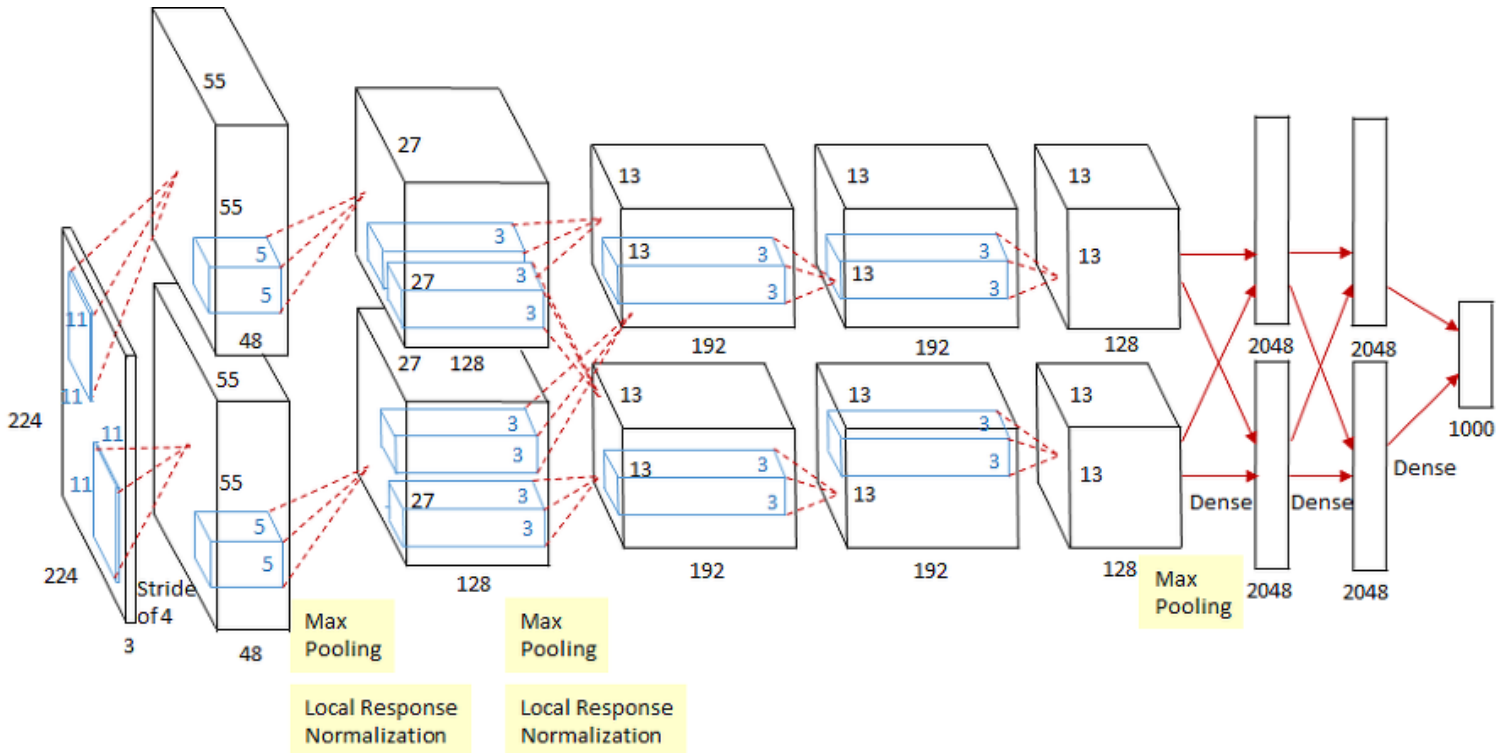
- Each feature in a CNN has a **receptive field**
  - I.e., the set of inputs that influence this feature
  - Increases via convolution or pooling operations
  - Thus: **higher-level layers have larger receptive field** and tend to provide higher-level features (e.g., edges, then parts, then objects)
  - Important design consideration: e.g., to detect an object of a certain size, want receptive field of that size
- Example: receptive field after two convolutions of width 3



- Effective receptive field often smaller and focused on center region

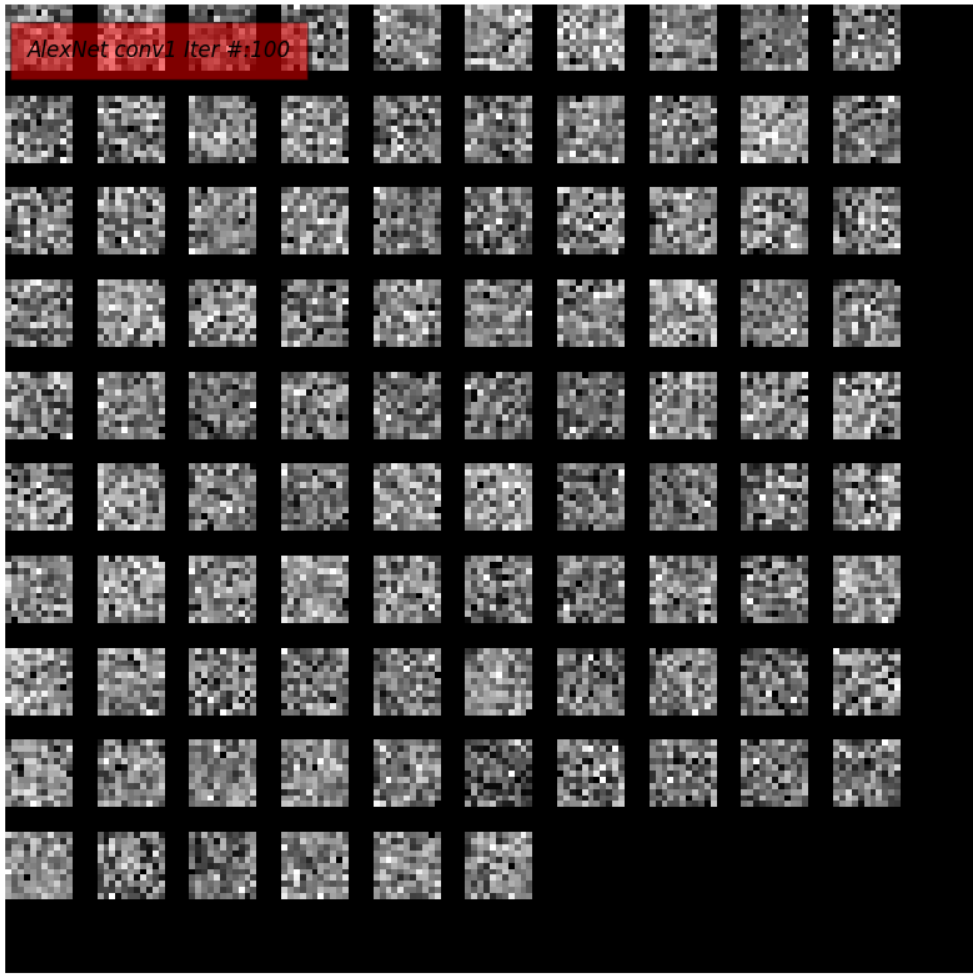
Example 2: AlexNet by Krizhevsky et al. (NeurIPS, 2012)

- AlexNet is the "first recent" architecture (2012)
  - Some history (ZDNET)
- Used for image classification, but adapted to many tasks (e.g. object detection)



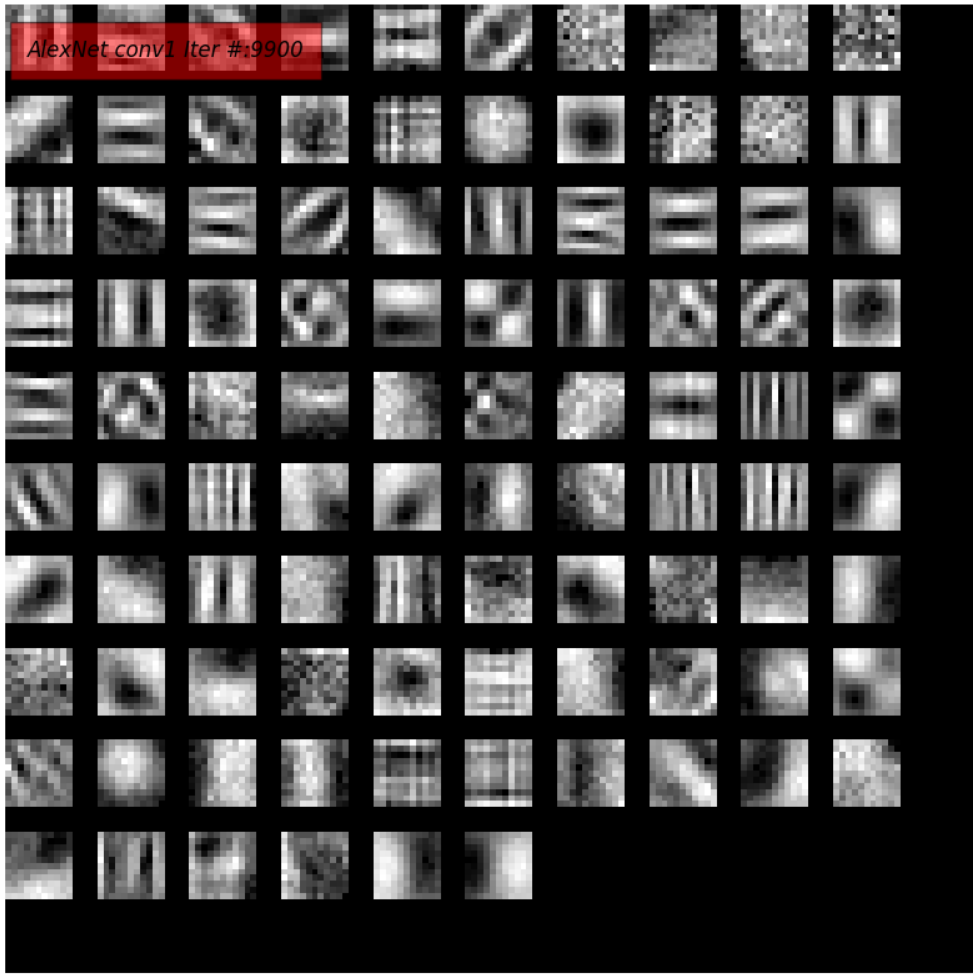
- ReLU activations
- Subsampling using max pooling
- Prediction using softmax layer, trained via cross entropy loss

What is learned in AlexNet? (1)



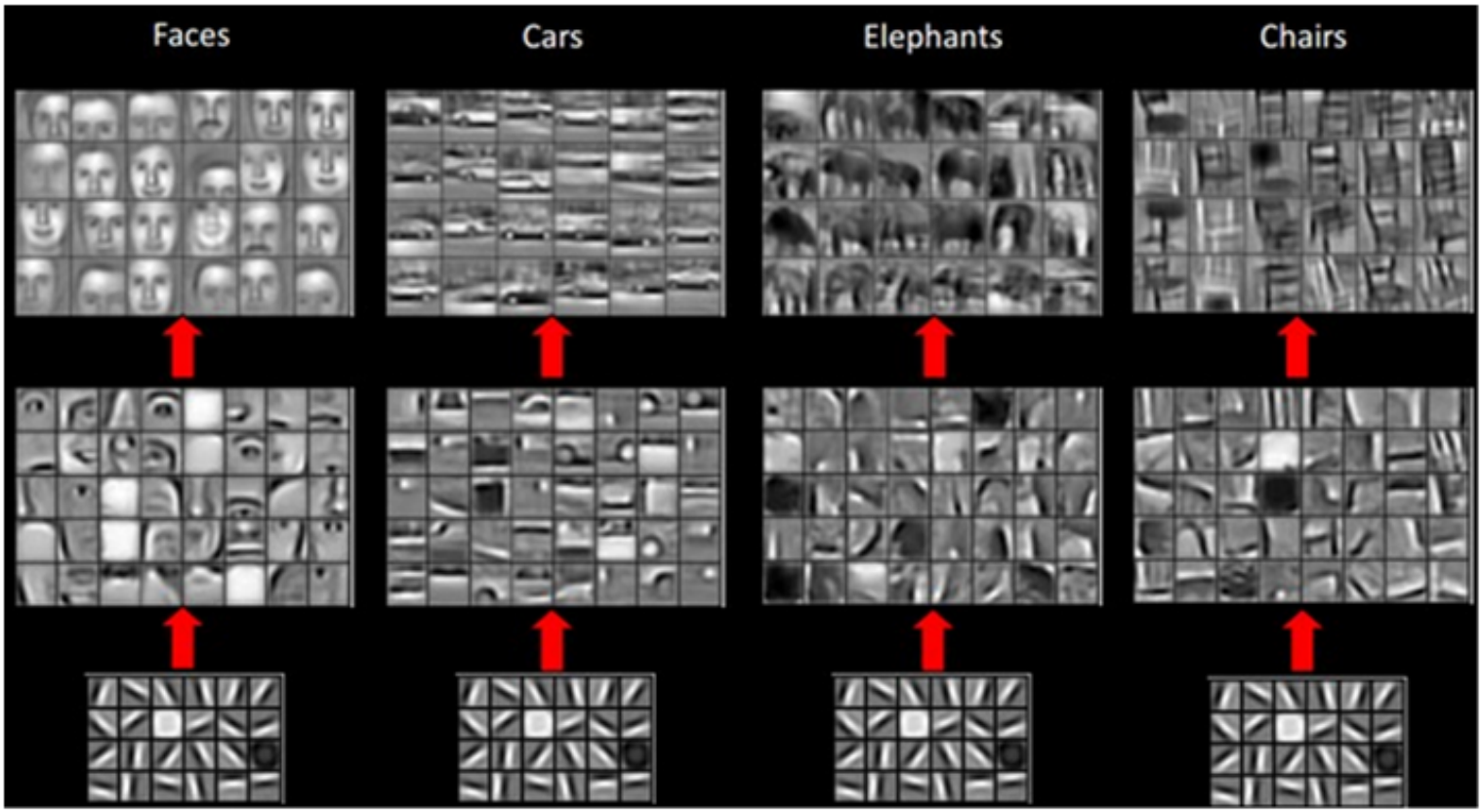
Layer 1, initialization (100 iterations)

What is learned in AlexNet? (2)



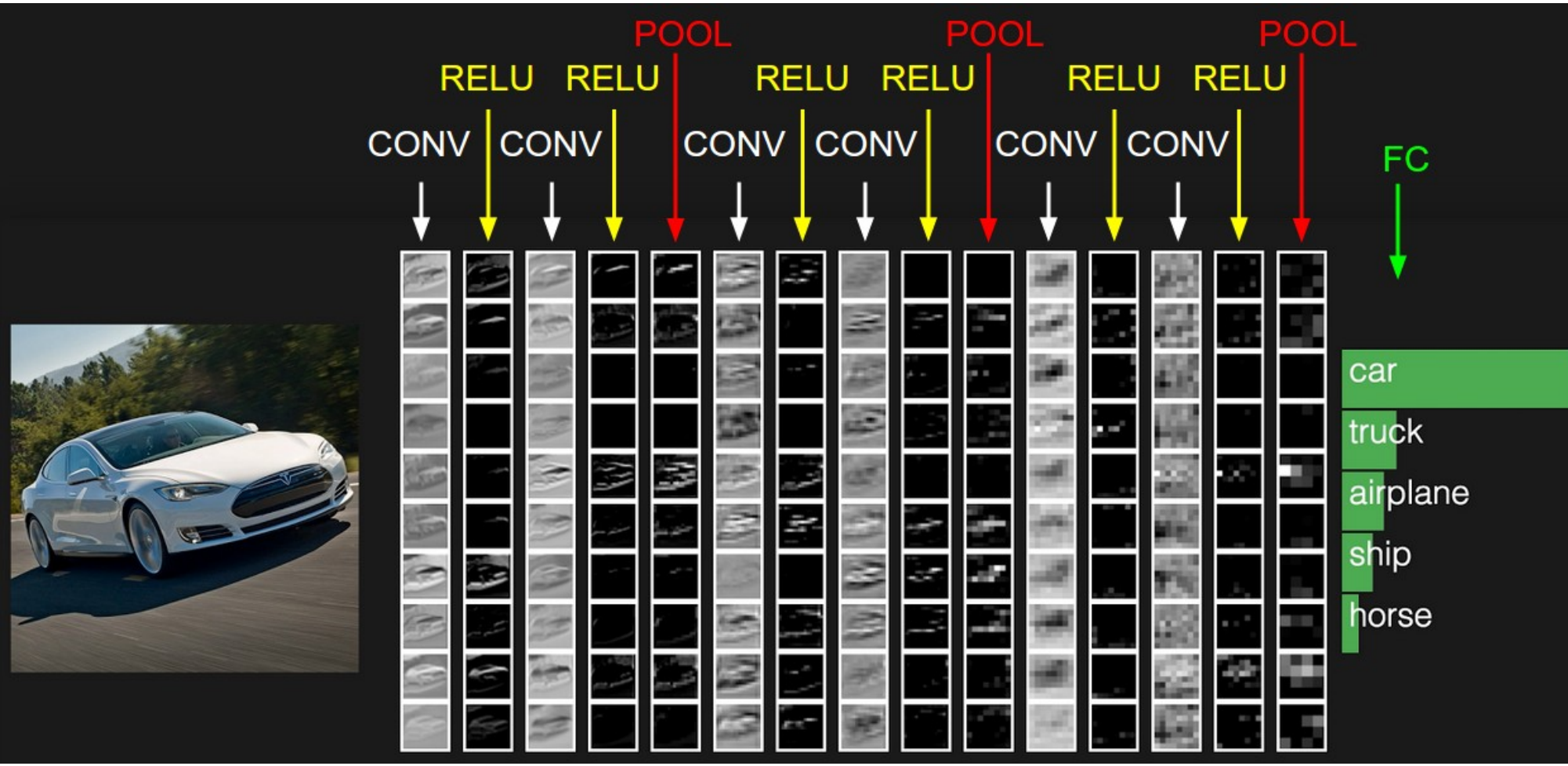
Layer 1, 1000 iterations

What is learned in AlexNet? (3)



Composition of features? Low-level features, parts, objects, ...?

Example: Outputs of a CNN



Web demo

Example features: edges, parts, objects (1)

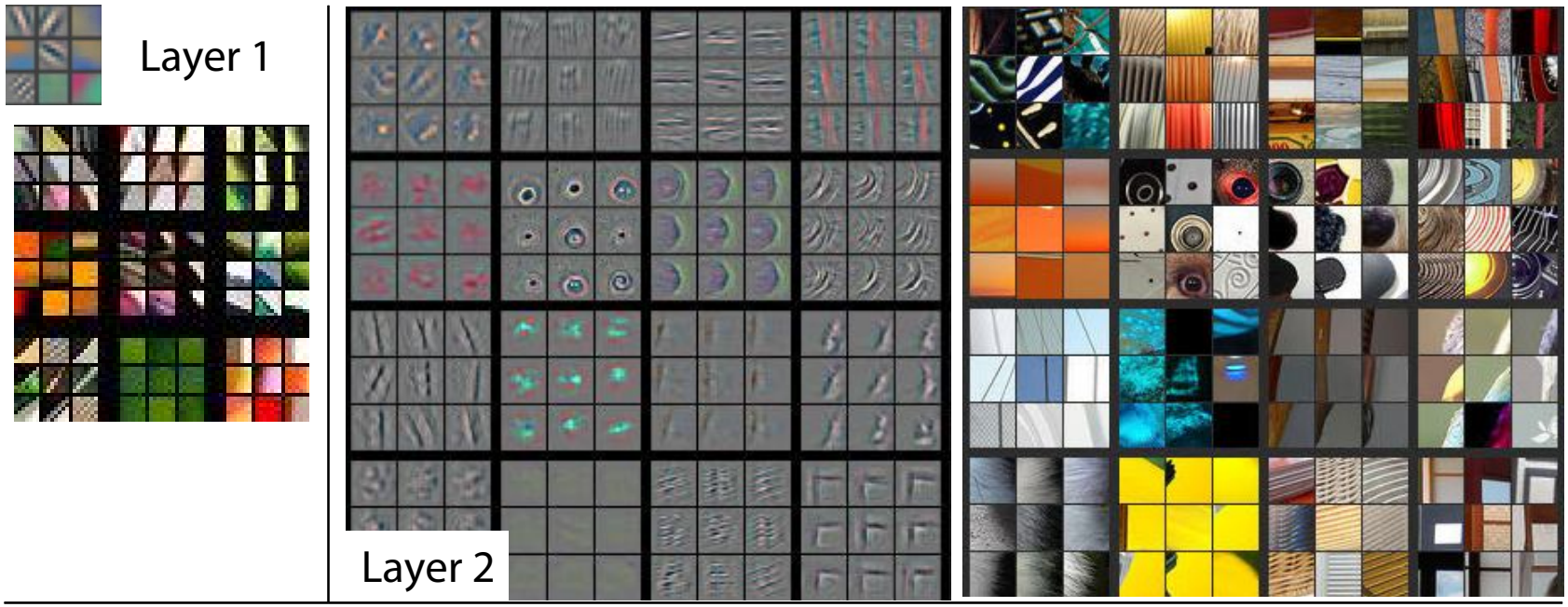


Figure 2. Visualization of features in a fully trained model. For layers 2-5 we show the top 9 activations in a random subset of feature maps across the validation data, projected down to pixel space using our deconvolutional network approach. Our reconstructions are *not* samples from the model: they are reconstructed patterns from the validation set that cause high activations in a given feature map. For each feature map we also show the corresponding image patches. Note: (i) the the strong grouping within each feature map, (ii) greater invariance at higher layers and (iii) exaggeration of discriminative parts of the image, e.g. eyes and noses of dogs (layer 4, row 1, cols 1). Best viewed in electronic form.

Zeiler and Fergus, 2013

Example features: edges, parts, objects (2)

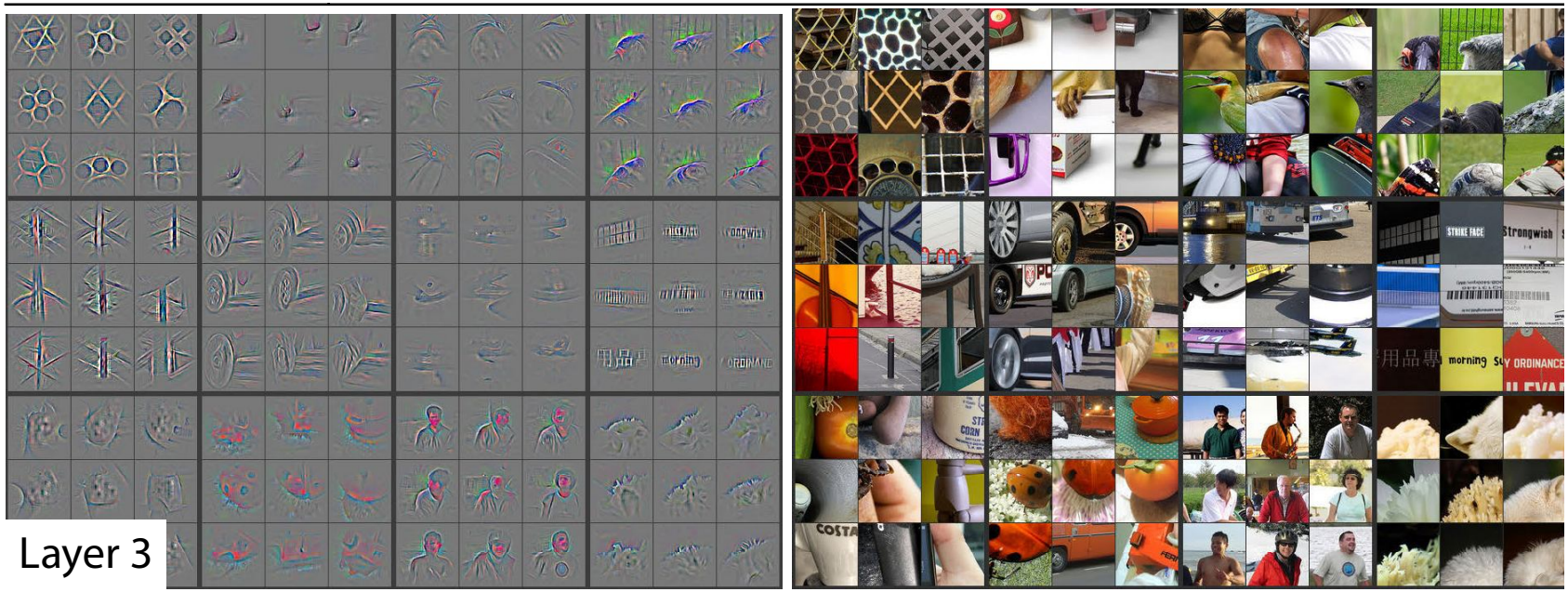
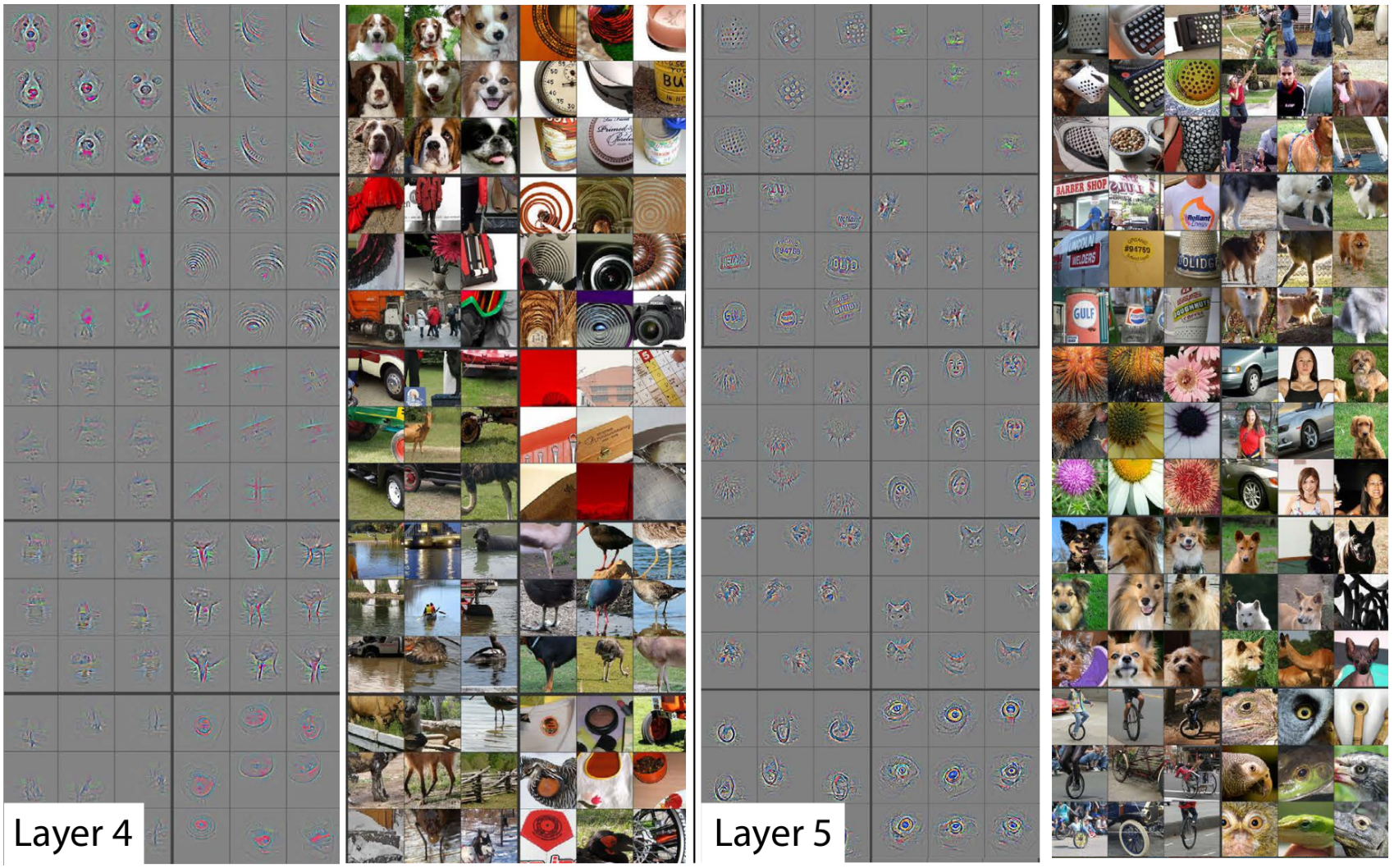


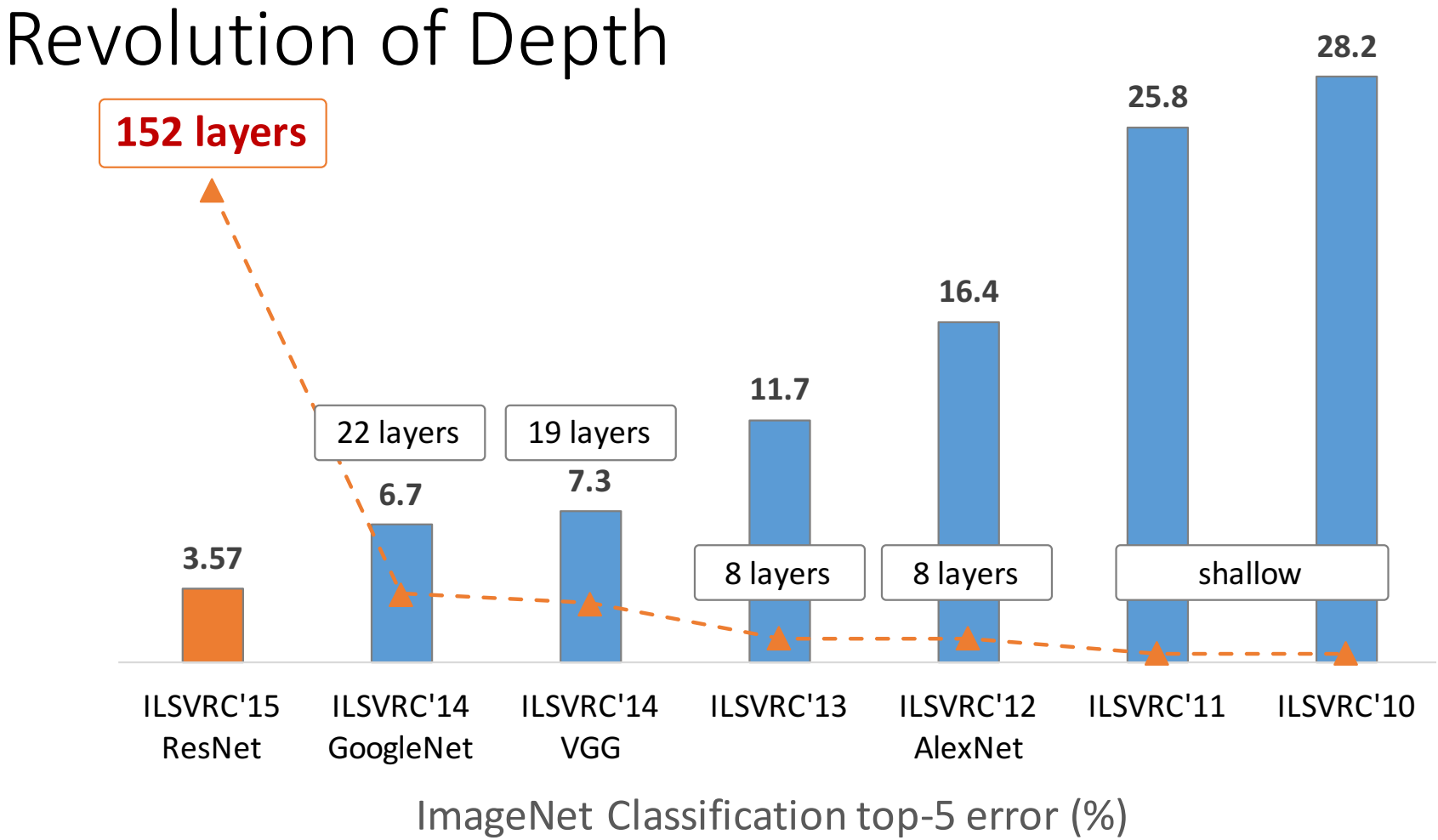
Figure 2. Visualization of features in a fully trained model. For layers 2-5 we show the top 9 activations in a random subset of feature maps across the validation data, projected down to pixel space using our deconvolutional network approach. Our reconstructions are *not* samples from the model: they are reconstructed patterns from the validation set that cause high activations in a given feature map. For each feature map we also show the corresponding image patches. Note: (i) the the strong grouping within each feature map, (ii) greater invariance at higher layers and (iii) exaggeration of discriminative parts of the image, e.g. eyes and noses of dogs (layer 4, row 1, cols 1). Best viewed in electronic form.

Example features: edges, parts, objects (3)



Zeiler and Fergus, 2013

CNNs for image classification (1)



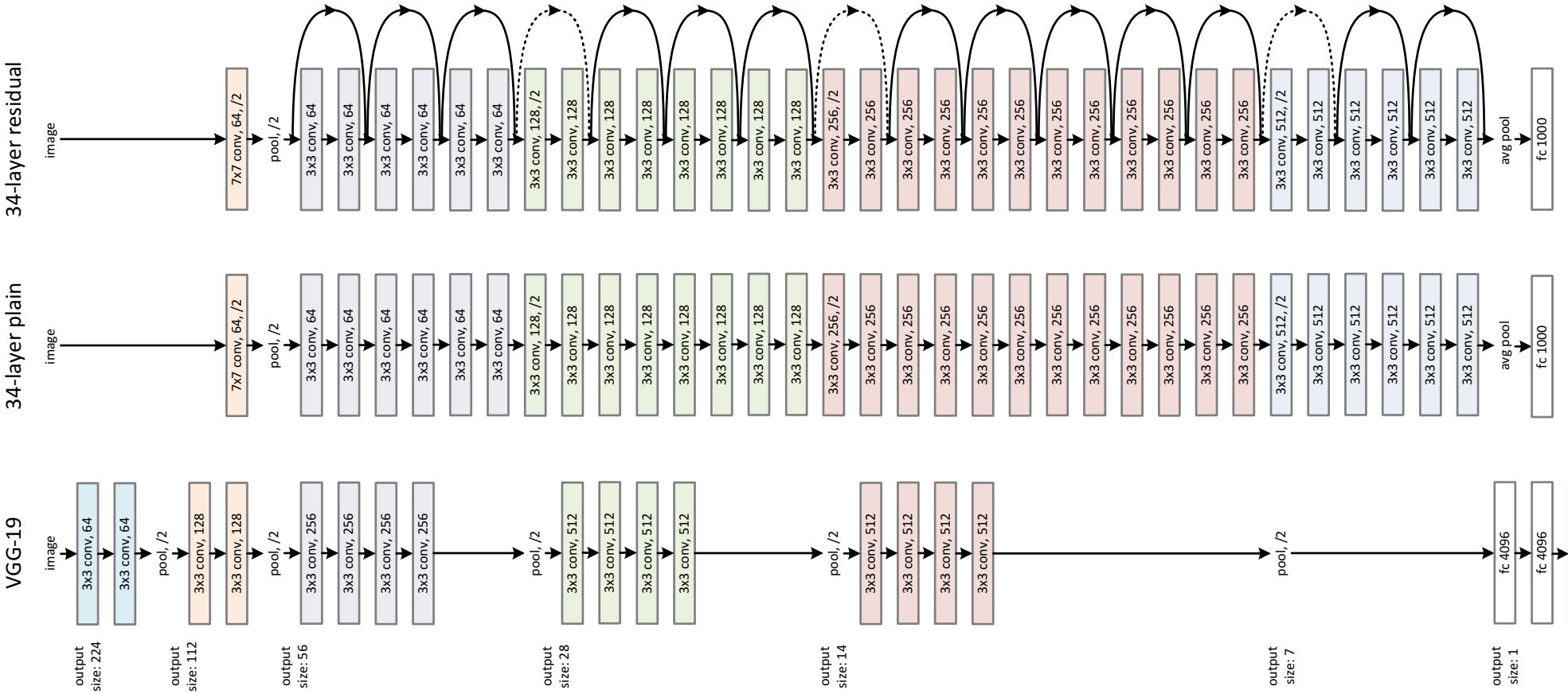
He et al., 2016

CNNs for image classification (2)

- Increase in depth possible due to suitable architectures
  - Avoid that adding depth leads to vanishing/exploding gradients or makes learning harder (e.g., residual units)
  - Avoid too-strong increase in memory consumption (e.g., separable convolutions and small filters)
- Not just depth, but also resolution and width important
- Data augmentation (e.g., crop, rotate, flip, . . .) important
- Pretraining—e.g., on ImageNet—to handle lack of training data for particular task
  - Pretrained models available; e.g., GluonCV
- Much more in CS 646 Computer Vision (HWS)
  - E.g., in-depth discussion of architectures, tasks other than image classification, and training CNNs in computer vision

Example: Microsoft's ResNet

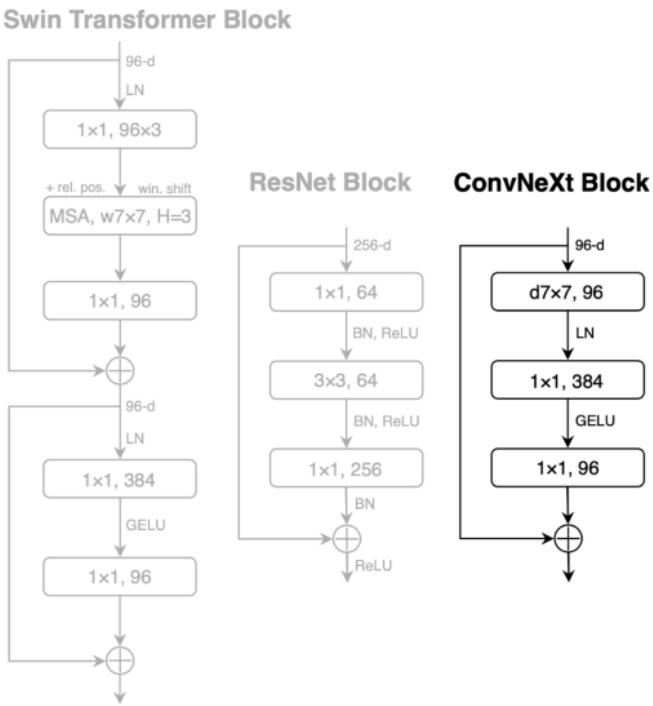
Example architectures for ImageNet's image classification task

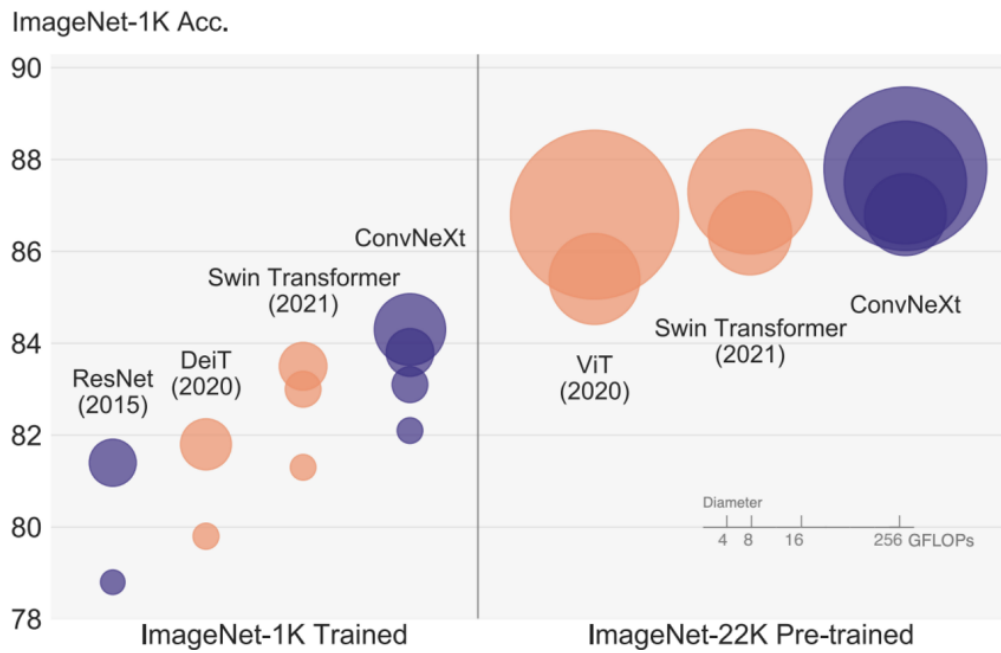


- ResNet (scaled to various depths) and some follow up models (ResNeXt, DenseNet) have defined the SotA for several years.

SotA in image classification

- Many Transformer-based methods have been proposed (more later)
- ConvNeXt and other recent CNNs keep up
  - Distant relationships modeled in deep layers
  - Translation equivariance, but larger filters (possible through depthwise separable conv.)
  - Benefits with reasonable amounts of training data
  - Borrows ideas from Transformer architecture





Deep Learning

07 – Recurrent Neural Networks and Structured State Space Models

Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-4

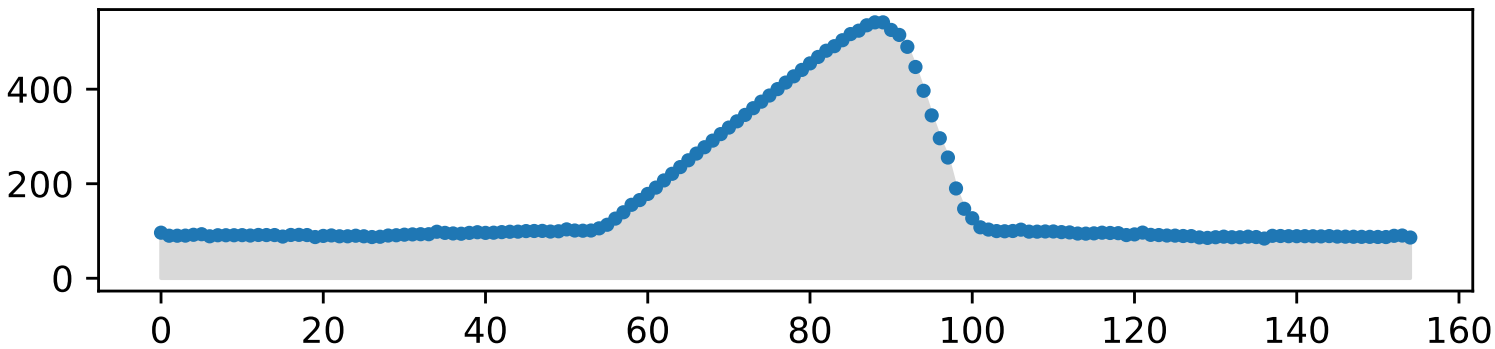
Outline

Sequential data

- **Sequence models** operate on sequential data
- **Sequential data** = data with meaningful order
  - Time series data (e.g., sequences of sensor readings)
  - Natural language text (e.g., sequences of characters / tokens)
  - Audio signals (e.g., sequences of amplitudes)
  - Images (e.g., sequences of pixels/rows/columns)
  - Videos (e.g., sequences of frames)
  - Processes (e.g., sequences of actions)
  - ...
- Data that is not naturally sequential is sometimes "sequentialized"
  - E.g., a set as a sequence of its elements
  - E.g., a graph as a sequence of its vertices and edges
  - Beware: sequence models are generally not order-invariant
    $\rightarrow$ At the very least, careful sequentialization required

Sequential data and part-based models

- Part-based view of sequential data (cf. lecture 05)
  - **time steps** ($\approx$ positions) correspond to parts
  - Features correspond to part representations/embeddings
  - Parts and their representations represented in a sequence
- Example: 1 feature (e.g., time series)



  - Time dimension, 1 feature (altitude)
  - time steps correspond to parts (minute intervals; ca. 150)
  - Features correspond to part representation (1 per part)
  - Both together form a real-number sequence of length 150
- Example: 3 features (e.g., multivariate time series)
  - Suppose we had three features (altitude, latitude, longitude)
  - Then each part represented by 3 values $\rightarrow$ seq. of 150 vectors $\in \mathbb{R}^3$

Sequential data, formally

- Inputs sequence $\boldsymbol{x} \in \mathcal{X}^*$
  - A sequence $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(\tau)}$ of length τ
  - $\boldsymbol{x}^{(t)} \in \mathcal{X}$ is input at **time step** $t \in \mathbb{N}^+$
  - Length τ may differ for different inputs
- Output sequence $\boldsymbol{y} \in \mathcal{Y}^*$
  - A sequence $\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, \ldots, \boldsymbol{y}^{(\tau_{\text{out}})}$ of elements from $\mathcal{Y}$
  - τ_{out} may or may not depend on input $\boldsymbol{x}$
  - τ_{out} may be deterministic or random (for fixed input)
- Examples from NLP domain, where $\boldsymbol{x}$ is a text document

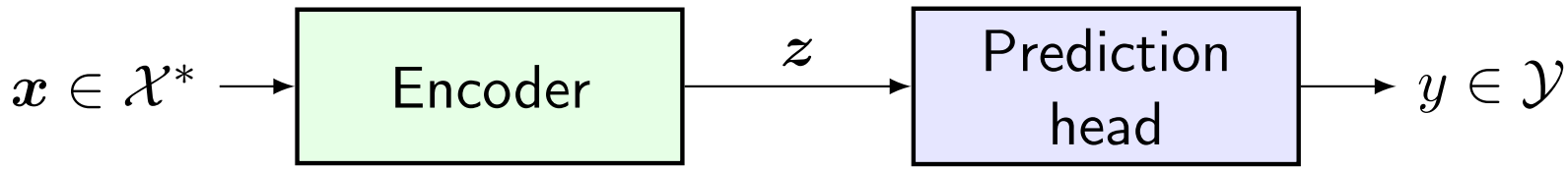
τ_{out}	**independent of $\boldsymbol{x}$**	**dependent on $\boldsymbol{x}$**
deterministic	Text classification	POS tagging
random	Language modeling	Translation

- Examples from time series domain, where $\boldsymbol{x}$ is a time series

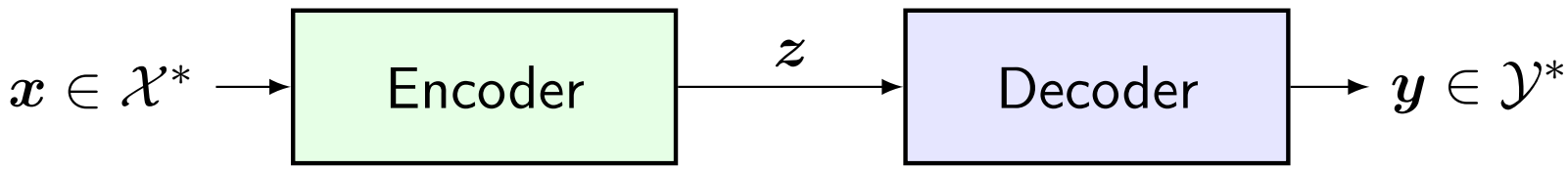
τ_{out}	**independent of $\boldsymbol{x}$**	**dependent on $\boldsymbol{x}$**
deterministic	TS classification	Anomaly detection
random	Sequence modeling	Forecasting

Types of sequence models

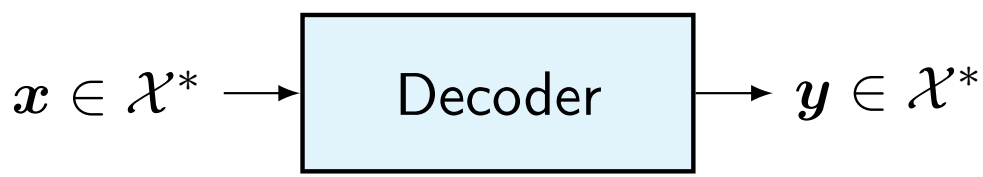
- **Encoder-only models** compute useful representations/embeddings
  - E.g., BERT for text data



- **Encoder-decoder models** add a decoder to generate sequences
  - E.g., T5 for sequence-to-sequence models



- **Decoder-only models** drop the encoder (and use decoder instead)
  - E.g., GPT-1/2/3/4 for language modelling



- Key approaches: CNNs (cf. 06), RNNs (now), Transformers (cf. 08)

Outline

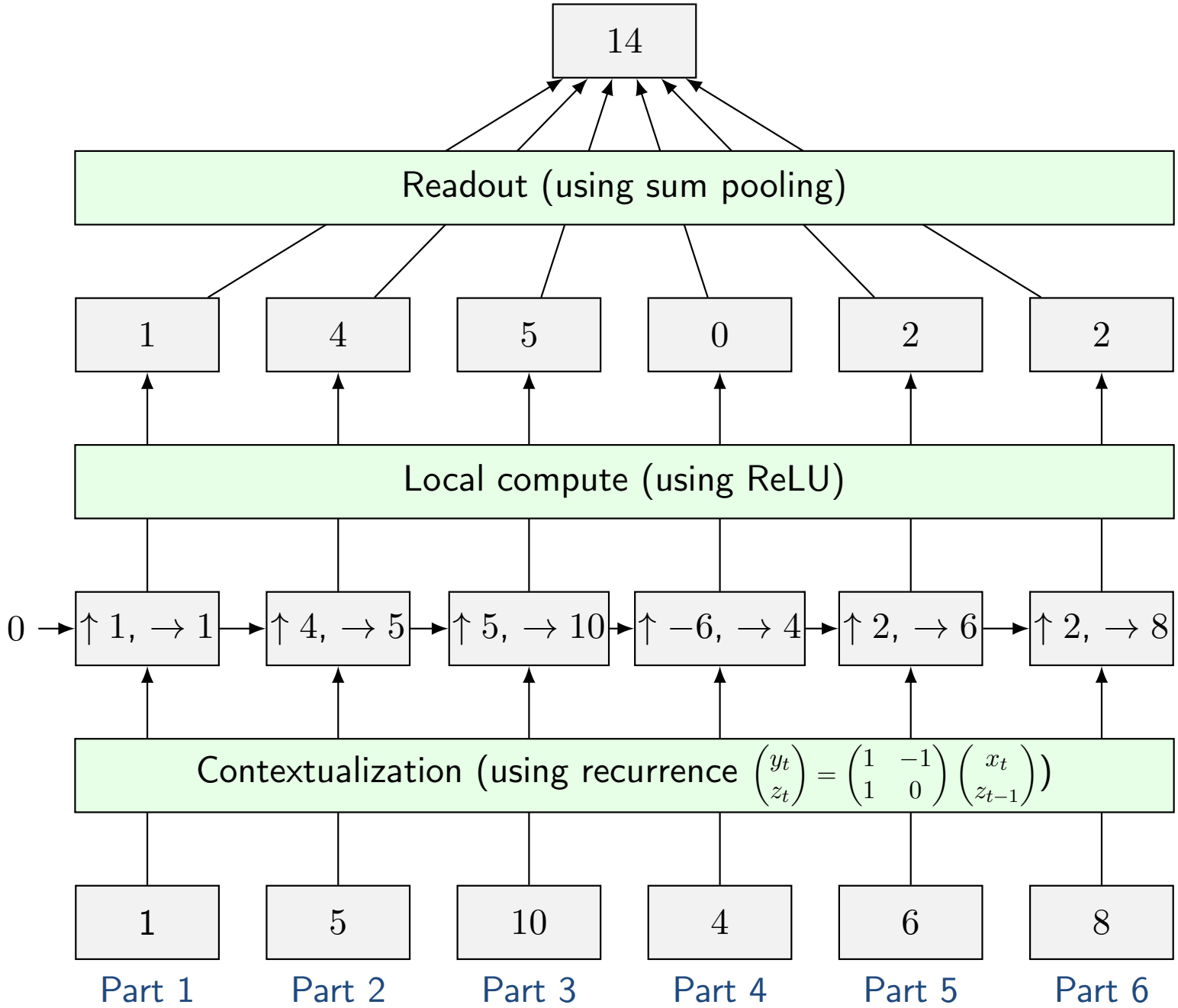
RNNs at a glance (1)

- **Recurrent neural networks (RNN)** are a family of neural networks for processing sequential data
  - Also applicable to other data modalities (e.g., grid data)
    $\rightarrow$ Not discussed here
- Key ingredients
  - Part-based modeling (cf. lecture 05)
  - Template modeling
  - Parameter sharing (across time steps)
- Example tasks
  - Part-based tasks (e.g., POS tagging, anomaly detection)
  - Global tasks (e.g., sequence classification)
  - Also: intermediates (e.g., forecasting)
- Simple & relevant (e.g., SOTA for long sequence modelling)

RNNs at a glance (2)

- Recall: Three key operations of part-based models (cf. lecture 05)
  1. **Contextualization**: incorporate information from other parts into each part embedding
  2. **Local compute**: update each part embedding individually
  3. **Pooling**: aggregate multiple (or all) part embeddings
- Contextualization operation: recurrence
  - Computes contextualized part embeddings
  - Does this by passing information "sideways" between parts via a recurrence
- Local compute: typically one of
  - Just an activation function (rare; often part of contextualization)
  - Rescaling (common; often part of contextualization)
  - MLP (modern RNNs; often separate)
- Pooling operation
  - Pooling rare, except readout
  - Readout: certain part embeddings (first/last part)
  - Readout: pooling (e.g., sum pooling)

Example RNN


14
Readout (using sum pooling)
1
4
5
0
2
2
Local compute (using ReLU)
0
↑ 1, → 1
↑ 4, → 5
↑ 5, → 10
↑ −6, → 4
↑ 2, → 6
↑ 2, → 8
Contextualization (using recurrence $\begin{pmatrix} y_t \\ z_t \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_t \\ z_{t-1} \end{pmatrix}$)
1
5
10
4
6
8
Part 1
Part 2
Part 3
Part 4
Part 5
Part 6

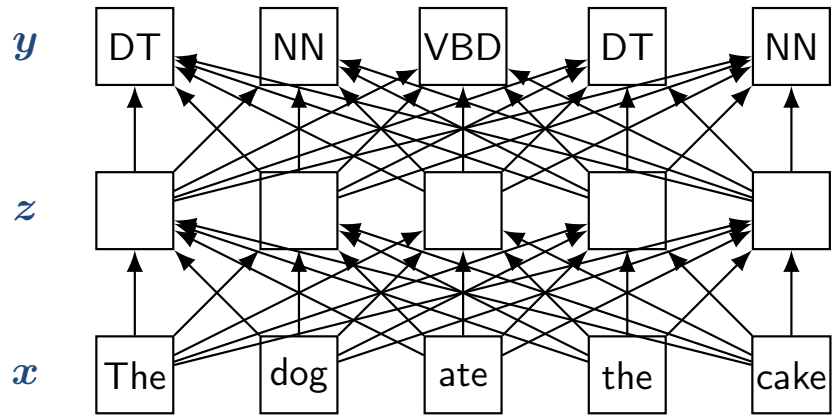
Example task: Sequence labeling

- In **sequence labeling** tasks, we assign a label to each input ($\tau_{\text{out}} = \tau$)
- Example: part-of-speech tagging

t	1	2	3	4	5
$\boldsymbol{y}$	DT	NN	VBD	DT	NN
	|	|	|	|	|
$\boldsymbol{x}$	**The**	**dog**	**ate**	**the**	**cake**

- How to solve this task with neural networks?

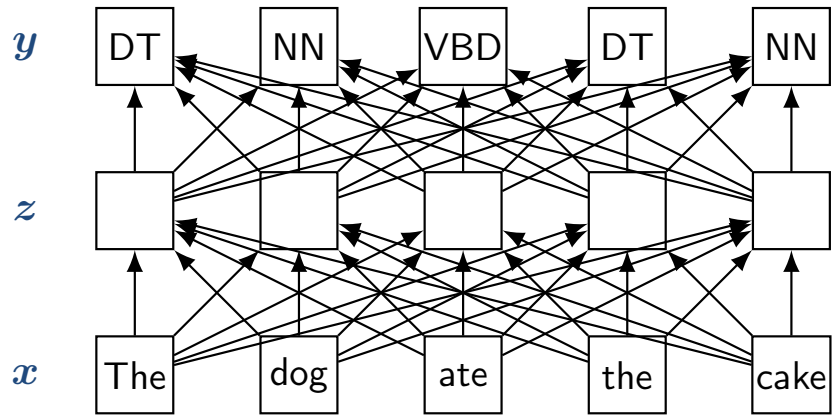
Fully-connected FNN (1)



How to read this architecture

- Each vertex corresponds to a subnetwork
- Each edge indicates directed connections between corresponding subnetworks
  - Between output neurons and input neurons
  - Not necessarily fully connected
- Other architecture details abstracted away

Fully-connected FNN (2)

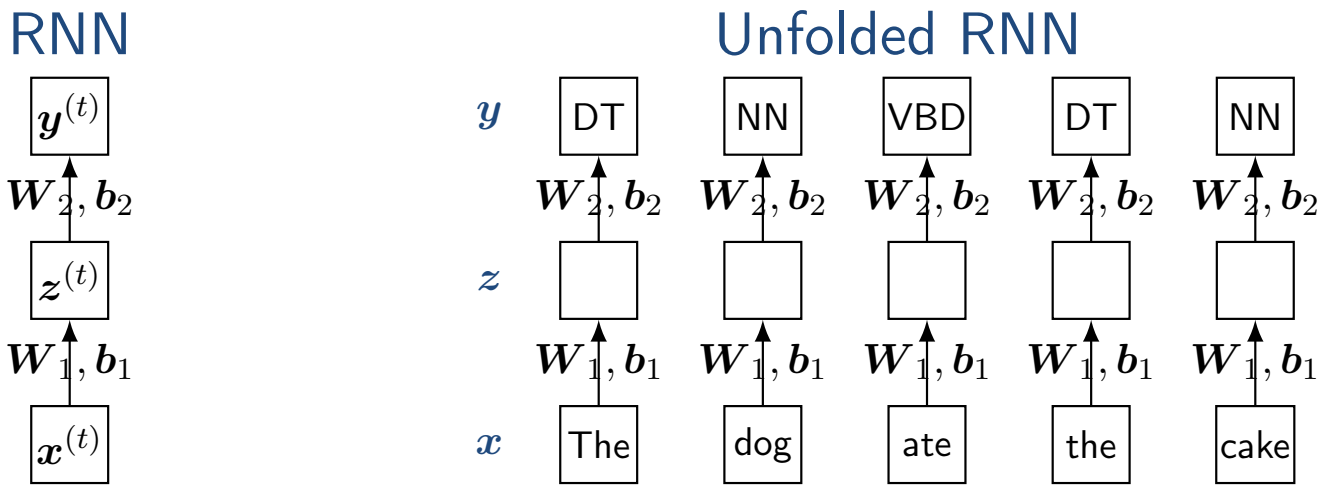


- For example,
  - Input vertex $\boldsymbol{x}^{(t)}$ corresponds to $D = 200$-dimensional embedding layer (= subnetwork)
  - Hidden vertex $\boldsymbol{z}^{(t)}$ corresponds to $Z = 50$ sigmoid units (subnetwork = 50 units, no connections in between)
  - Output vectex $\boldsymbol{y}^{(t)}$ corresponds to $C = 36$-dimensional softmax layer (= subnetwork)
  - Fully connected: when $A \to B$, then connection from each unit $a \in A$ to each unit $b \in B$

Why not use an MLP?

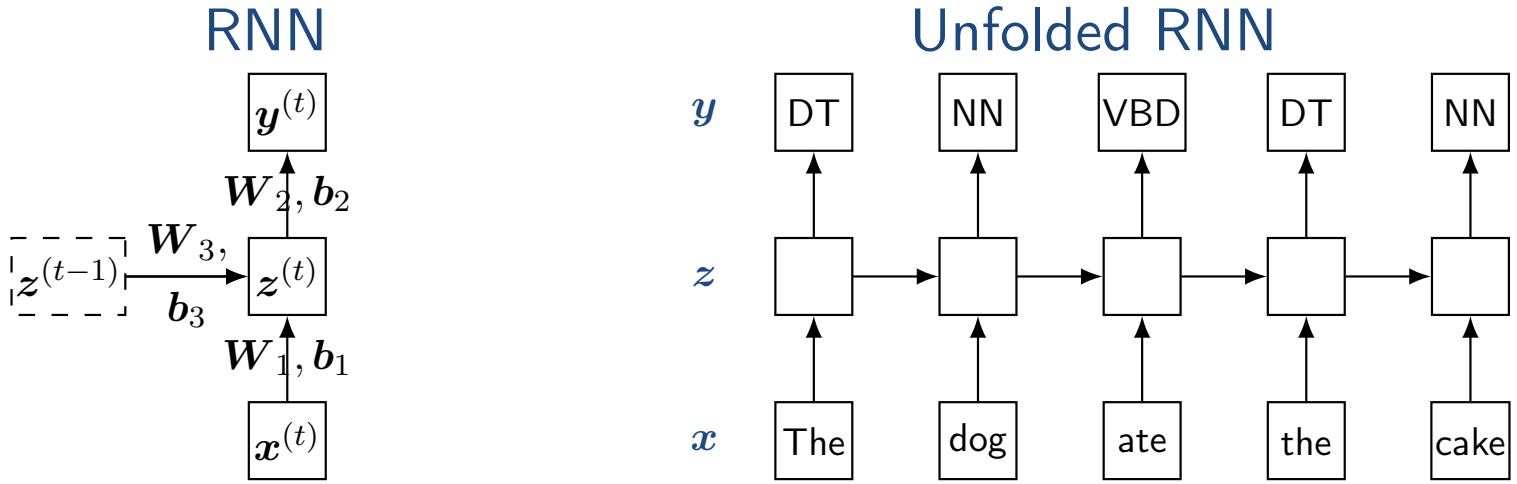
- Sequence length τ_{net} hard-coded
  - Cannot handle input sequences of length $\tau > \tau_{\text{net}}$ directly
  - Shorter sequences can be supported (at least in principle) by **padding** with a special "empty" input to length τ
- Many parameters; e.g., for our example
  - Input embedding layers: $\tau_{\text{net}} VD$ parameters (e.g., $5 \cdot 100\text{k} \cdot 200 = 100\text{M}$)
  - Hidden layers: $\tau_{\text{net}} DZ$ (e.g., $5 \cdot 200 \cdot 50 = 50\text{k}$)
  - Output layers: $\tau_{\text{net}} ZC$ (e.g., $5 \cdot 50 \cdot 36 = 9\text{k}$)
- No information sharing between time steps
  - Network needs to learn that "the" is likely to be a determiner for each time step separately
- For probabilistic models: outputs cond. independent given $\boldsymbol{z}$
  - Cannot express, for example, that two outputs must agree (e.g., 00 with 50% probability, 11 with 50% probability)
- RNNs can address all of these problems

A simple RNN



- RNNs are template models
  - Network is constructed by repeating the same **template** for every time step $\rightarrow$ called **unfolding**
  - Parameters are **shared** across time steps
- Template is important architecture decision; e.g., here:
  - Outputs only depend on their corresponding word $\rightarrow$ context ignored

Unidirectional RNN



- Templates usually include connections between time steps
- In a **unidirectional RNN**, connections between time steps go from left to right
  - Network can pass along information to subsequent time steps
  - $z^{(t)}$'s commonly referred to as **hidden states**
- For example, at $t = 2$
  - $z^{(2)}$ computed from input $x^{(2)}$ ("dog")
  - And from hidden representation $z^{(1)}$
  - Useful information to pass along via $z^{(1)}$, for example: "word at time step 1 was likely to be a determiner"

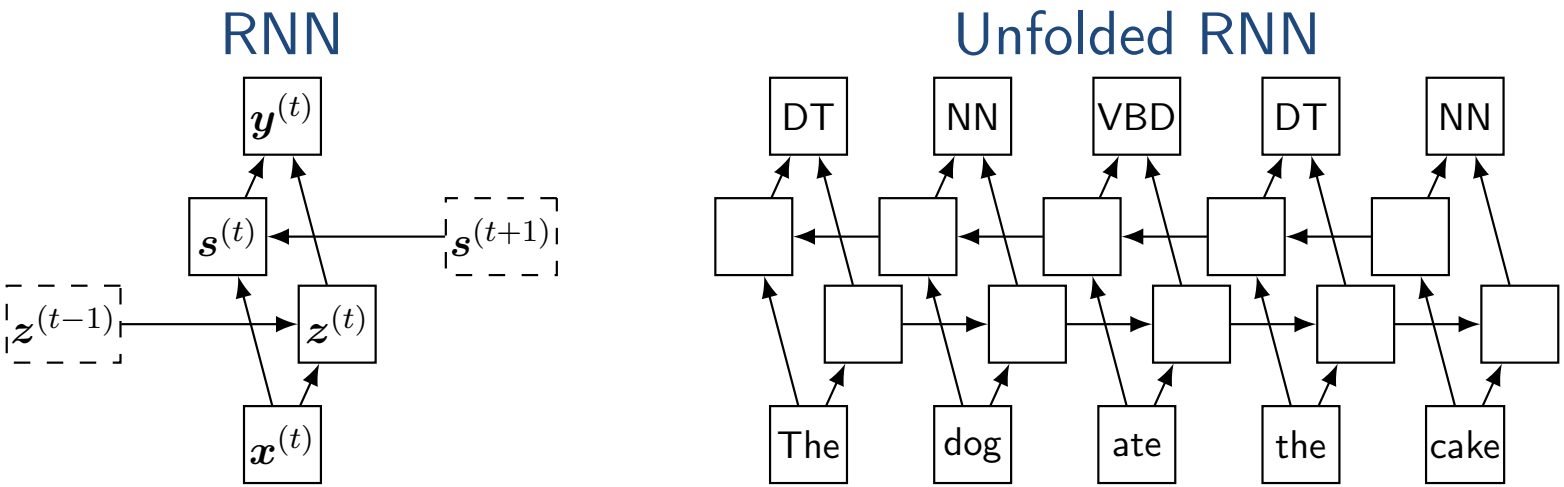
Discussion (1)

- Network is recurrent
  - E.g., a recurrence of form $\boldsymbol{z}^{(t)} \leftarrow \boldsymbol{f}(\boldsymbol{x}_t, \boldsymbol{z}_{t-1}, \boldsymbol{y}_{t-1})$
    - $\boldsymbol{x}_t$ is **input** connection
    - $\boldsymbol{z}_{t-1}$ is **hidden-to-hidden** connection
    - $\boldsymbol{y}_{t-1}$ is **output-to-hidden** connection
  - Can be done "infinitely" often
  - May also access $\boldsymbol{x}_{t-k}$, $\boldsymbol{z}_{t-k}$ and/or $\boldsymbol{y}_{t-k}$ for $k > 0$
- Unfolded network is **deep in time**
  - Even when template is shallow
- Hidden state representation serves two purposes
  1. Prediction: provide good features to output layer (at current time step)
  2. Memory: provide useful information to subsequent time step(s) (e.g., previous word was likely a determiner)
- Suitable representations not prespecified, but learned
- "Multi-purpose" representations are common
  - E.g., $\boldsymbol{z}$ in hierarchical softmax served multiple purposes: determines probability distribution at *each* vertex in decision tree

Discussion (2)

- When only hidden-to-hidden connections used: outputs cond. independent given hidden layer
  - Fixed by adding output-to-hidden/output connections (more later)
- Parameter sharing $\rightarrow$ inductive bias: **position independence**
  - Same operation but different "data" at each position
  - Selection of useful input features does not depend on position
  - Selection of useful information to pass along does not depend on pos.
- RNNs are universal (can simulate any Turing machine)
- Can be slow as computation cannot be parallelized over time
  - Since operation at time step t needs input from time step $t-1$
- Advantage: for inference, unrolling not necessary
  - Process time steps incrementally
  - Only keep information from previous time step
- Advantage: real-time prediction possible (e.g., for data streams)
  - Can output $\boldsymbol{y}^{(t)}$ as soon as we see $\boldsymbol{x}^{(t)}$
- Disadvantage: data in subsequent time steps ignored (e.g., for NER: "Green Mile is a movie." vs. "Green is a color.")

Bidirectional RNN



- **Bidirectional RNNs** also have backwards connections
  - Information about "past" and "future" captured in hidden units
  - Every output depends on every input $\rightarrow$ better predictions
- Main drawbacks
  - No real-time predictions
  - More resource-intensive during prediction

Outline

RNNs and vanishing gradients (1)

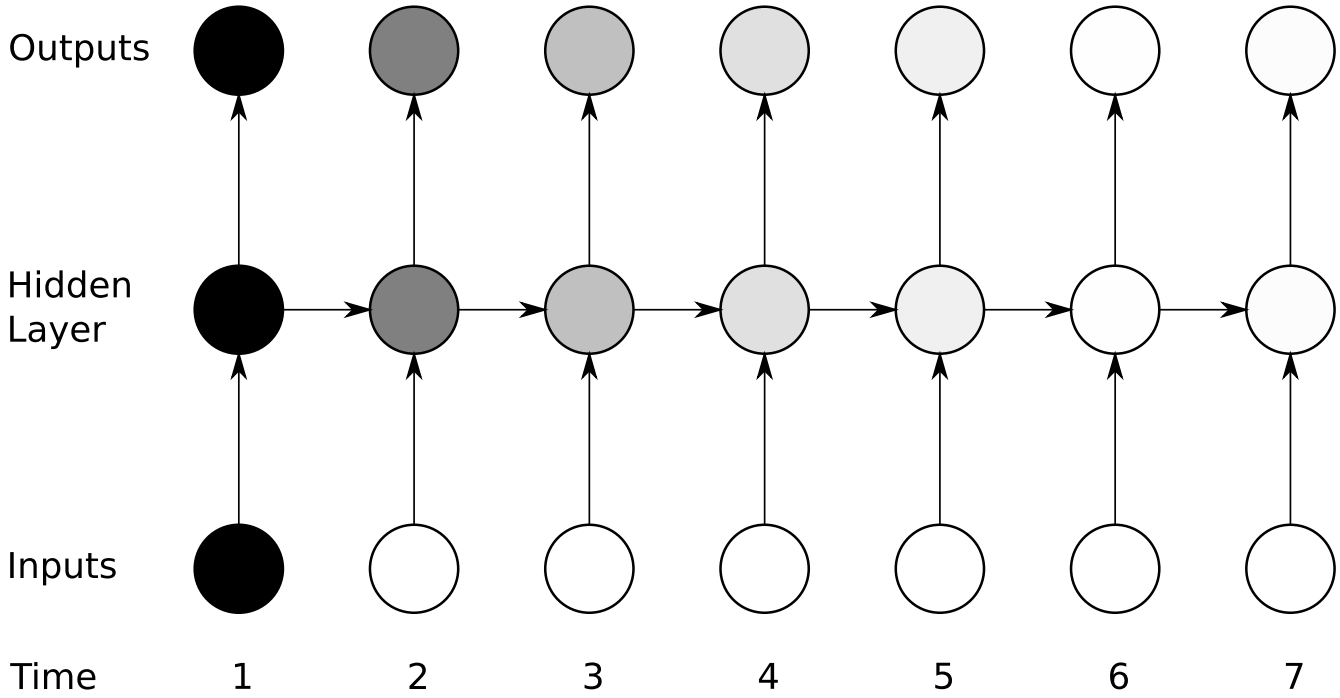


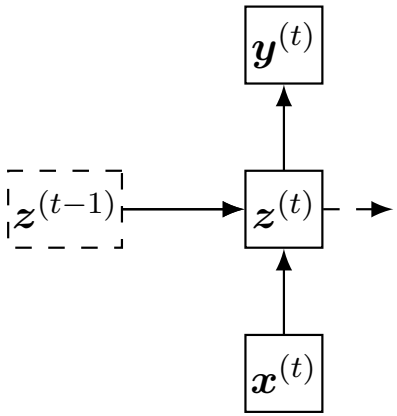
Figure 4.1: **The vanishing gradient problem for RNNs.** The shading of the nodes in the unfolded network indicates their sensitivity to the inputs at time one (the darker the shade, the greater the sensitivity). The sensitivity decays over time as new inputs overwrite the activations of the hidden layer, and the network 'forgets' the first inputs.

RNNs and vanishing gradients (2)

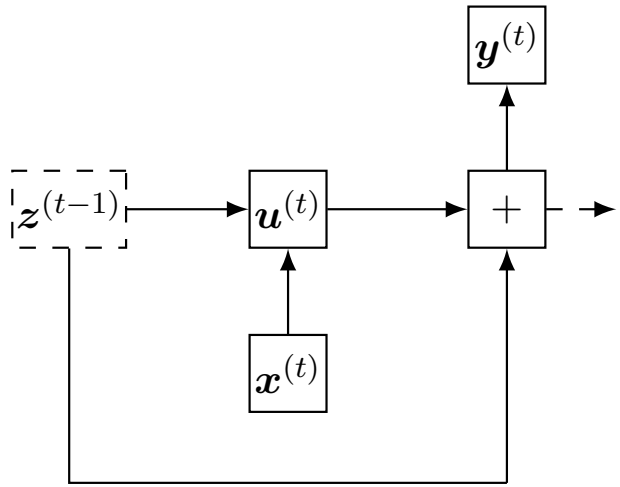
- Recall vanishing gradient problem
  - Gradient decreases quickly with distance from output
- RNNs are deep in time, thus translates to:
  - Gradient of past input $\boldsymbol{x}^{(t_{\text{past}})}$ w.r.t. to current output $\boldsymbol{y}^{(t)}$ quickly decreases over time $(t - t_{\text{past}})$
  - I.e., network becomes insensitive to changes in past input
  - I.e., it "forgets" past input = **catastrophic forgetting**
- Why is this a problem?
  - Intuitively: network has difficulties when predictions depend on inputs seen in the far past
- Two common approaches
  - Use **gating mechanisms** in hidden layers to mitigate this effect (e.g., LSTM, GRU)
  - Let the network directly access past information via **attention** (cf. lecture 08)
  - Also: do both

Why not use residual connections?

Plain RNN



RNN w/ residual connection



- Simple "solution": add a residual connection (e.g., as above)
- Not suitable for RNNs
  - $\boldsymbol{z}^{(t)} = \boldsymbol{z}^{(t-1)} + \boldsymbol{u}^{(t)} = \boldsymbol{z}^{(t-2)} + \boldsymbol{u}^{(t-1)} + \boldsymbol{u}^{(t)} = \cdots = \boldsymbol{z}^{(0)} + \sum_{t'=1}^{t} \boldsymbol{u}^{(t')}$
  - At time t, $\boldsymbol{u}^{(t)}$ has same "contribution" as $\boldsymbol{u}^{(t-1)}$, $\boldsymbol{u}^{(t-2)}$, ...
    $\rightarrow$ Not time-dependent
- Instead, we'd like to
  - Capture **short-range dependencies** (more recent $\rightarrow$ more contribution)
  - Capture **long-range dependencies** (far away $\rightarrow$ large contribution)
  - Need trade-off $\rightarrow$ gating mechanisms
  - How to trade-off? $\rightarrow$ learned

Gating mechanisms

Key idea: use "gates" to control whether or not new information is allowed to override hidden state

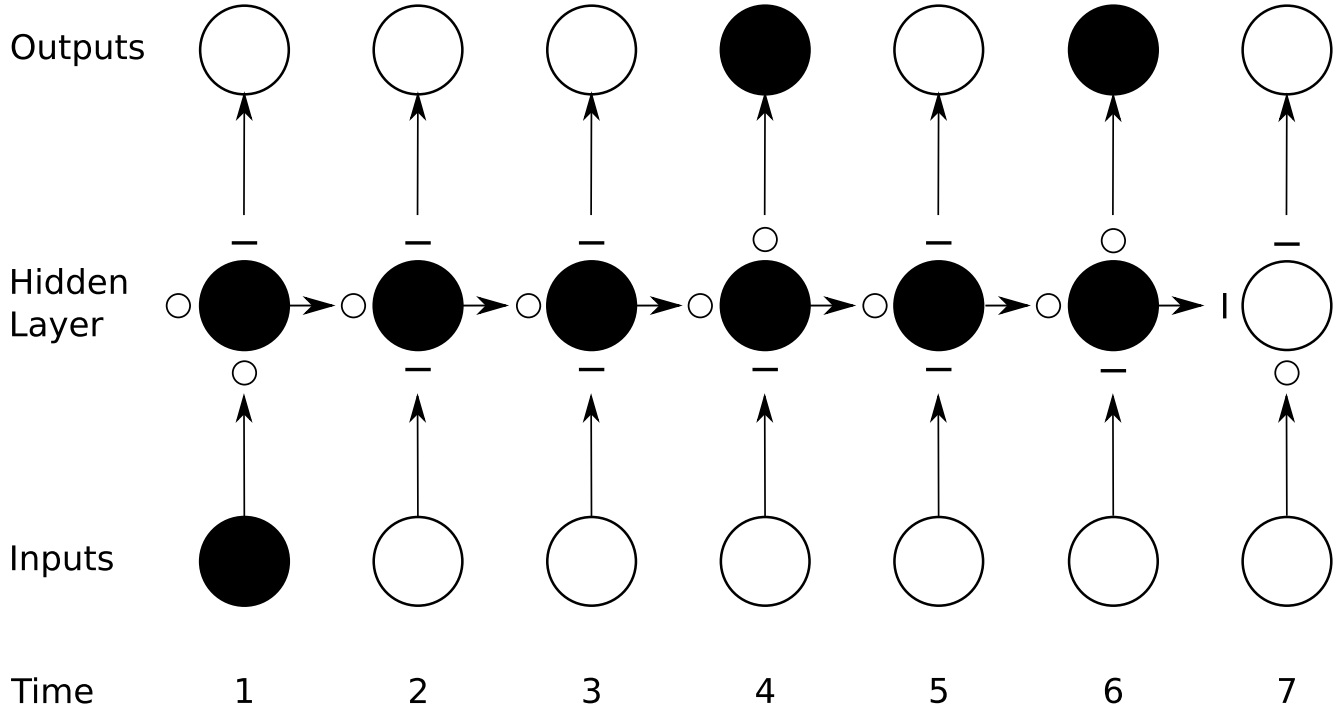
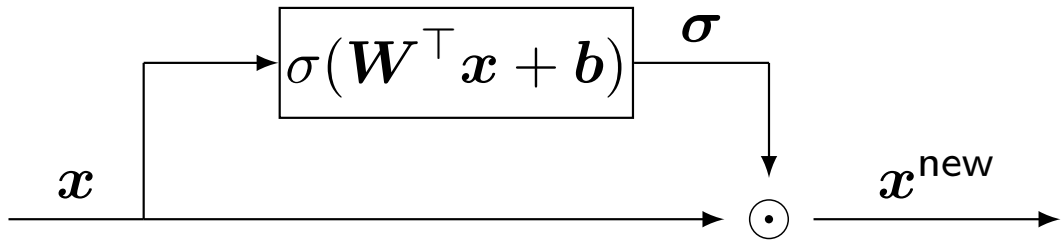


Figure 4.4: **Preservation of gradient information by LSTM.** As in Figure 4.1 the shading of the nodes indicates their sensitivity to the inputs at time one; in this case the black nodes are maximally sensitive and the white nodes are entirely insensitive. The state of the input, forget, and output gates are displayed below, to the left and above the hidden layer respectively. For simplicity, all gates are either entirely open ('O') or closed ('—'). The memory cell

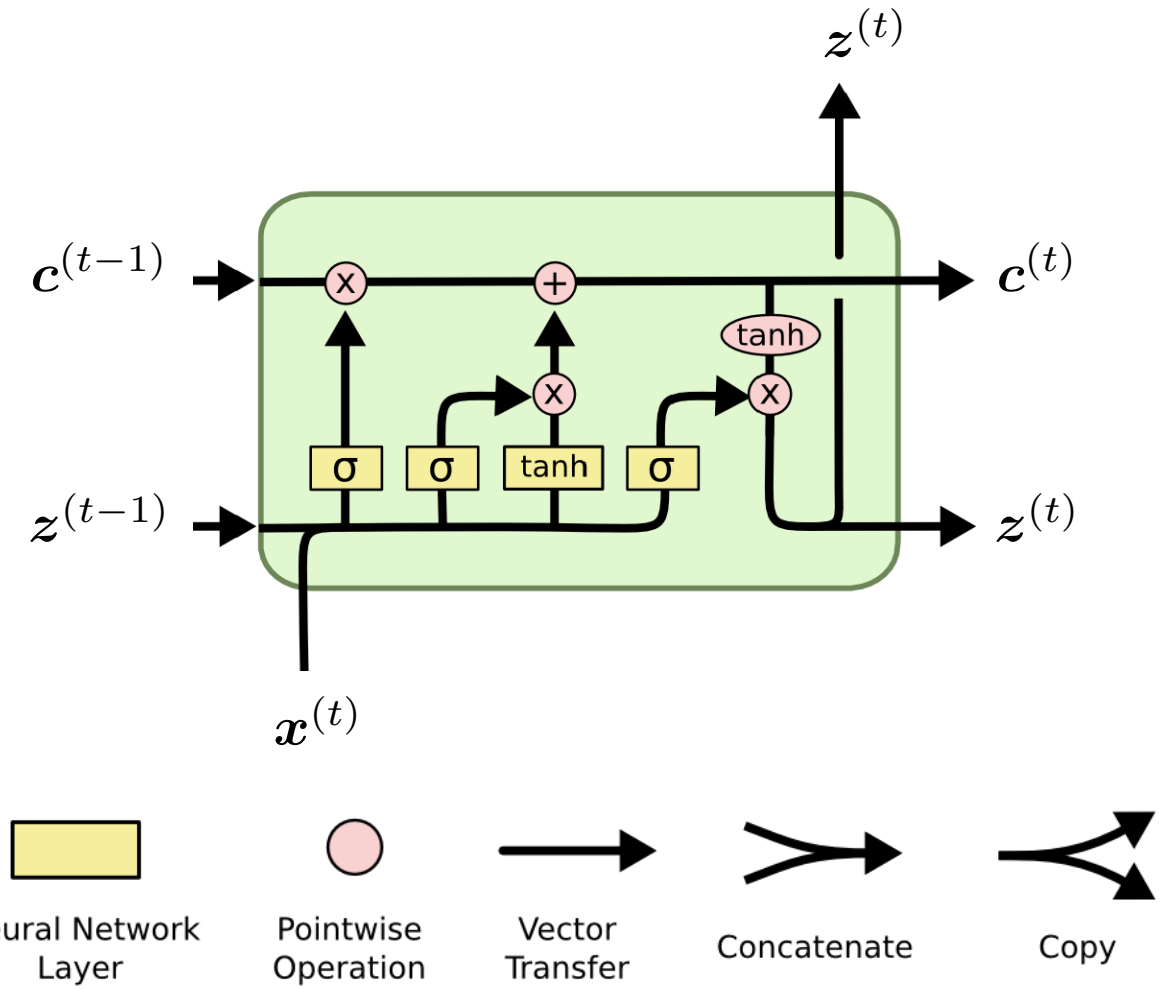
A simple gate



- Input: $\boldsymbol{x} \in \mathbb{R}^D$
- Filter $\boldsymbol{\sigma} \in [0,1]^D$
  - Computed from $\boldsymbol{x}$ (here: via logistic units)
  - Think: σ_k = Percentage of x_k to retain
  - Learned (here: via $\boldsymbol{W}, \boldsymbol{b}$)
- Output: $\boldsymbol{x}^{\text{new}} \in \mathbb{R}^D$ is "gated" input
  - Since $x_k^{\text{new}} = x_k \cdot \sigma_k$
  - Gate is "open" when $\sigma_k = 1 \rightarrow x_k^{\text{new}} = x_k$
  - Gate is "closed" when $\sigma_k = 0 \rightarrow x_k^{\text{new}} = 0$
- Used in RNNs, but also other architectures

Long short term memory (LSTM)

LSTMs are a (sub)architecture for the hidden layer of an RNN.



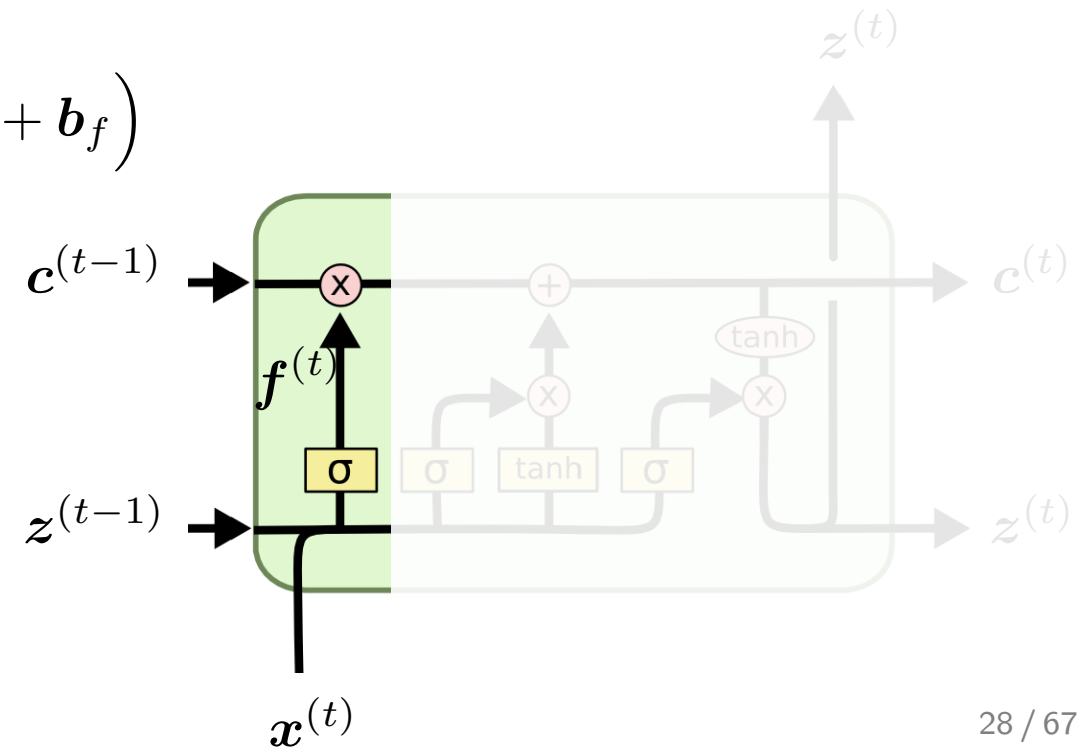
Olah, 2015

Key ideas

- A D-dimensional **LSTM unit** (or *LSTM cell*) has
  - A **cell state** $\boldsymbol{c} \in \mathbb{R}^D$ = "memory"
  - A **hidden state** $\boldsymbol{z} \in [-1, 1]^D$ = "output"
- Three gates control how states are updated at each time step
  - Forget gate controls to what extent cell state is retained
  - Input gate controls to what extent cell state is updated
  - Output gate controls outputs
- All gates are controlled by the unit's states
  - Again, hidden representations serve multiple purposes
- Carefully designed such that gradient information can flow backwards
  - Cell state updated, but not replaced/recomputed
  - Effective weight of input $\boldsymbol{x}^{(t_{\text{past}})}$ = product of forget gate activations $\boldsymbol{f}^{(t_{\text{past}+1})} \odot \cdots \odot \boldsymbol{f}^{(t)}$ to output $\boldsymbol{y}^{(t)}$
  - Gradient cannot explode, but can be kept high (when $\boldsymbol{f}^{(t)} \approx \mathbf{1}$)
  - More in exercise

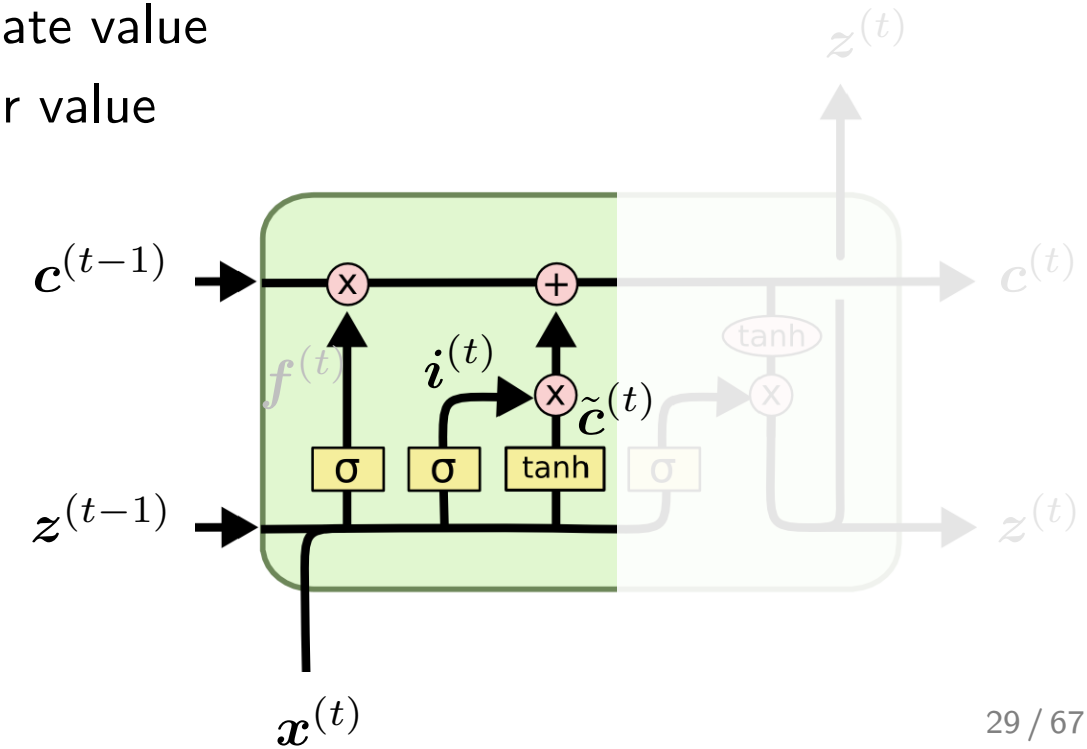
Forget gate

- **Forget gate** controls to what extent state is kept = $\boldsymbol{f}^{(t)} \in [0,1]^D$
  - When $f_k^{(t)} = 1$, gate is open
    $\rightarrow$ element c_k not modified (history kept)
  - When $f_k^{(t)} = 0$, gate is closed
    $\rightarrow$ element c_k zeroed out (history forgotten)
- Decision made based on input $\boldsymbol{x}^{(t)}$ and hidden state $\boldsymbol{z}^{(t-1)}$
  - $\boldsymbol{f}^{(t)} = \sigma\left(\boldsymbol{W}_f^T \begin{pmatrix} \boldsymbol{z}^{(t-1)} \\ \boldsymbol{x}^{(t)} \end{pmatrix} + \boldsymbol{b}_f\right)$
  - Weights learned during training



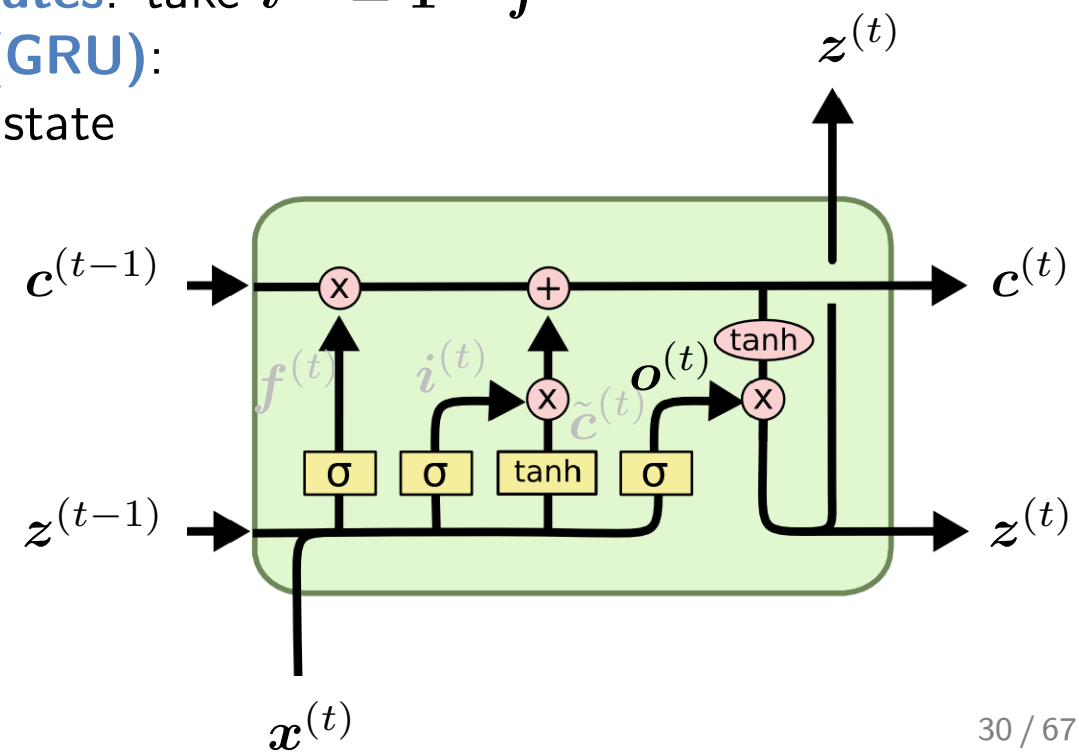
Input gate

- **Input gate** decides to what extent state is updated
  - Proposed update: $\tilde{\boldsymbol{c}}^{(t)} \in [-1, 1]^D$
  - Extent by which to update: $\boldsymbol{i}^{(t)} \in [0, 1]^D$
  - For both: weights learned during training
- Example updates
  - $f_k^{(t)} = 1,\ i_k^{(t)} = 0 \rightarrow$ old value
  - $f_k^{(t)} = 0,\ i_k^{(t)} = 1 \rightarrow$ new value
  - $f_k^{(t)} = 1,\ i_k^{(t)} = 1 \rightarrow$ update value
  - $f_k^{(t)} = 0,\ i_k^{(t)} = 0 \rightarrow$ clear value



Output gate

- **Output gate** decides to what to output
  - Output = filtered version of new cell state = new hidden state
  - Cell state pushed into $[-1, 1]$ via $\tanh$ function
  - Extent by which to filter: $\boldsymbol{o}^{(t)} \in [0, 1]^D$
- Variants ($\rightarrow$ exercise)
  - **Peephole connections**: gate layers (sigmoids) take cell state as additional input
  - **Coupled input/forget gates**: take $\boldsymbol{i}^{(t)} = \mathbf{1} - \boldsymbol{f}^{(t)}$
  - **Gated recurrent units (GRU)**: additionally combine cell state and hidden state (pushes filtering to obtain hidden state to next time step)



Using RNN encoders

- So far, we have focused on RNN encoders
- Provide **contextualized representations** $\boldsymbol{z}^{(t)}$ for each input $\boldsymbol{x}^{(t)}$
  - Contextualized since $\boldsymbol{z}^{(t)}$ depends on surrounding inputs
  - As discussed: Useful for element-level tasks (e.g., sequence labeling)
- Provide fixed-dimensional sequence representation known as **thought vector**
  - Obtained via a form of readout
  - For uni-directional RNNs: last state $\boldsymbol{z}^{(\tau)}$
  - For bi-directional RNNs: additionally first state $\boldsymbol{s}^{(1)}$ in backward direction
  - As discussed: useful for sequence-level tasks (e.g., sequence classification)
- Coming up: RNN decoders
  - Example of a deep autoregressive model

Outline

Excursion: Deep generative models

- **Generative model** = distribution $p(\boldsymbol{y})$ for $\boldsymbol{y} \in \mathcal{Y}$
  - Examples from ML course: beta-binomial model, Naive Bayes, factor analysis & probabilistic PCA, Gaussian mixture model
- **Conditional generative model** = conditional distribution $p(\boldsymbol{y}|\boldsymbol{c})$
  - May think of $\boldsymbol{c} \in \mathcal{C}$ as an input / condition / prompt / context
  - And of $p(\boldsymbol{y}|\boldsymbol{c})$ as the corresponding output
  - In contrast to discriminative models, usually multiple "correct" (and often structured) outputs
- **Deep generative models** use deep neural networks to define a generative model for complex data distributions (e.g., text, audio, image, graph, . . .)
- Tasks of interest include
  1. Train a generative model $p(\boldsymbol{y})$ from samples of data distribution $p_D(\boldsymbol{y})$
  2. Sample from $p(\boldsymbol{y})$
  3. Determine the top-k highest-probability outputs
  4. Given $\boldsymbol{y}$, compute density $p(\boldsymbol{y})$
  5. Sometimes: obtain **latent codes** $\boldsymbol{z}$ for a given data point $\boldsymbol{y}$

  (each optionally conditioned on $\boldsymbol{c}$)

Sampling from DGMs (1)

By sampling, we create "new" data. Goals are:

- High *quality*: samples are part of data distribution
- High *diversity*: all modes of data distribution captured
- *Generalization*: samples generalize beyond training data

Left: sample, right: 1st/2nd/3rd/4th nearest neighbor in training data

Sampling from DGMs (2)

Image generation (GAN, DCGAN, CoupledGAN, ProgressiveGAN, StyleGAN)

Text generation (GPT-1/2/3/4)

PROMPT:

```
In a shocking finding, scientist discovered a herd of unicorns living in a remote,
previously unexplored valley, in the Andes Mountains. Even more surprising to the
researchers was the fact that the unicorns spoke perfect English.
```

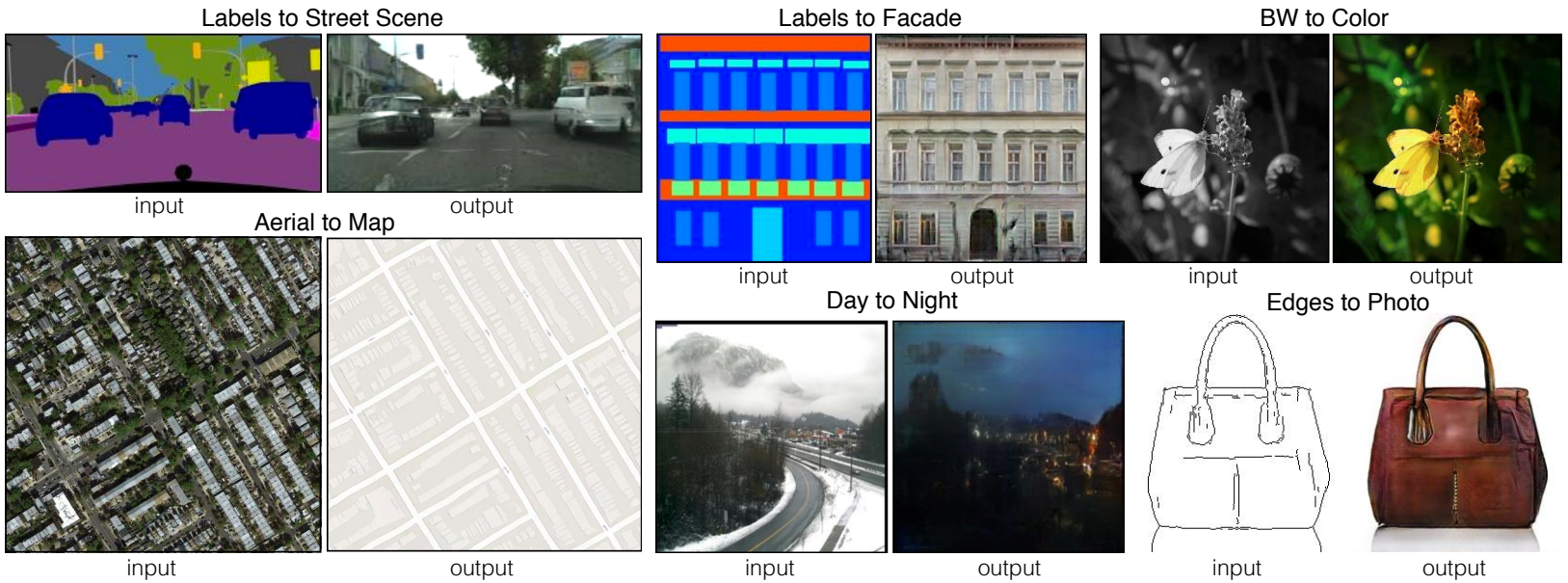
RESPONSE:

```
The scientist named the population, after their distinctive horn, Ovid's Unicorn.
These four-horned, silver-white unicorns were previously unknown to science.
Now, after almost two centuries, the mystery of what sparked this odd phenomenon
is finally solved. Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz,
and several companions, were exploring the Andes Mountains when they found a small valley,
with no other animals or humans....
```

Sampling from DGMs (3)

Conditional generative models can be used for A-to-B problems, where $A, B \in \{\text{text}, \text{speech}, \text{image}, \text{graph}, \ldots\}$.

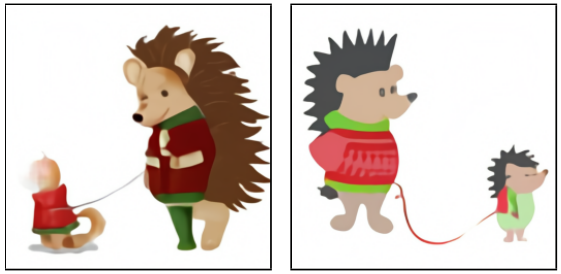
image-to-image (cGAN)



Sampling from DGMs (4)

text-to-image (DALL-E 1/2)

(a) an armchair in the shape of an avocado.



(b) an illustration of a baby hedgehog in a christmas sweater walking a dog

text-to-text, few shot learning (GPT-3)

A "whatpu" is a small, furry animal native to Tanzania. An example of a sentence that uses the word whatpu is:
We were traveling in Africa and we saw these very cute whatpus.

To do a "farduddle" means to jump up and down really fast. An example of a sentence that uses the word farduddle is:
One day when I was playing tag with my little sister, she got really excited and she started doing these crazy farduddles.
A "yalubalu" is a type of vegetable that looks like a big pumpkin. An example of a sentence that uses the word yalubalu is:
I was on a trip to Africa and I tried this yalubalu vegetable that was grown in a garden there. It was delicious.

DGMs can also be used for...

- Density estimation
  - Outlier detection
  - Data compression
  - Generative classifiers
  - Model comparison
- Imputation to fill in missing values
- Structure discovery (via latent variable z)
- Representation learning (via latent variable z)
- Interpolation between data points
- Training data generation
- Few-shot learners
- ...
- For CV, see related course: CS 668 Generative Computer Vision Models (FSS)

Generating sequences

- So far, we focused on RNNs encoders
  - E.g., to obtain fixed-dimensional element representation (**contextualized representation**)
  - E.g., to obtain fixed-dimensional sequence representation (**thought vector**)
- We now look at deep generative models for sequence data...
  - Distribution $p(\boldsymbol{y}) \in \mathcal{Y}^*$ over sequences of elements from $\mathcal{Y}$
  - Example: **Language model**, where $\mathcal{Y}$ is a discrete set of characters / tokens / words
  - Example: Time series model, where $\mathcal{Y} = \mathbb{R}$ (univariate) or $\mathcal{Y} = \mathbb{R}^D$ (multivariate)
  - Special case: (probabilistic) RNN decoder
- ...and then at conditional generative models
  - Special cases: encoder-decoder RNN, decoder-only RNN
- We focus on plain RNNs throughout
  - But discussion also applies to other sequence models such as RNNs with attention or Transformer models (cf. lecture 08)

Autoregressive generative models

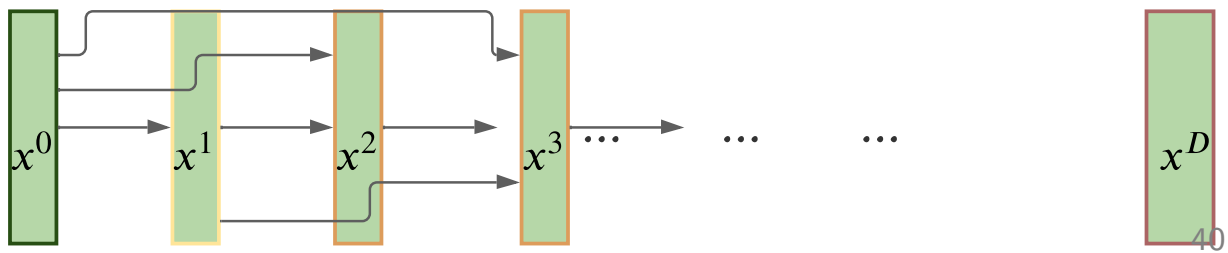
- To keep notation uncluttered, we write y_t for $y^{(t)}$ etc.
- **Autoregressive generative models** decompose the joint distribution into **next-element distributions** using the product rule

$$p(y_{1:\tau}) = p(y_1)p(y_2|y_1)p(y_3|y_1,y_2)\cdots$$

$$= \prod_{t=1}^{\tau} \underbrace{p(\overbrace{y_t}^{\text{next element}} \mid \overbrace{y_{1:t-1}}^{\text{past elements}})}_{\text{next-element distribution } p_t(y_t)}$$

  - Can be done for *any* distribution
  - E.g., for categorical outputs, $p_t(y)$ = categorical distribution
  - E.g., for real-valued outputs, $p_t(y)$ = continuous distribution (e.g., normal distribution)

Autoregressive model: Learn conditional of each variable given past



Next-element distributions

- Recall: $p_t(y) \stackrel{\text{def}}{=} p(y|y_{1:t-1})$
  - Convention: y for random variable, y_t for concrete sample
- Example: categorical data
  - C categories $\rightarrow$ can represent $p_t(y)$ with probability vector $\boldsymbol{p}_t \in \mathcal{S}^C$
  - Deep autoregressive models compute $\boldsymbol{p}_t$ from past outputs $y_{1:t-1}$ (e.g., via a deep network + softmax layer)
  - Example: Categories $\{$ red, green, blue $\}$ $(C = 3)$
  - $\boldsymbol{p}_t = \begin{pmatrix} 0.8 & 0.05 & 0.15 \end{pmatrix}$
  - Samples: red (with prob. 80%), green (5%), or blue (15%)
- Example: real-valued data, normal distribution
  - $p_t(y)$ has form $\mathcal{N}(y; \mu_t, \sigma_t^2)$
  - Deep autoregressive models compute μ_t and σ_t^2 from past outputs (e.g., via a deep network + linear layer)
  - Example: $\mu_t = 5$, $\sigma_t^2 = 2$
  - Samples: 5.337371, 3.874985, 6.089283, $\ldots$
- Generally, at each time step
  1. Determine (parameters of) p_t via deep model
  2. Sample from p_t

Sampling from autoregressive generative models

- To generate, **forward sample** iteratively from $p_t(y) = p(y|y_{1:t-1})$
  1. Generate y_1 by sampling from $p_1(y) = p(y)$ *The*
  2. Generate y_2 by sampling from $p_2(y) = p(y|y_1)$ *dog*
  3. Generate y_3 by sampling from $p_3(y) = p(y|y_{1:2})$ *ate*
  4. Generate y_4 by sampling from $p_4(y) = p(y|y_{1:3})$ *the*
  5. Generate y_5 by sampling from $p_5(y) = p(y|y_{1:4})$ *cake*
  6. ...
- Sampling process can continue endlessly
  - I.e., infinitely long sequences can be generated
  - To handle finite sequences of different lengths, a special **end-of-sequence (EOS)** marker can be used to stop generation
  - E.g., *end-of-sequence token* used in language models: stop generation as soon as $y_t = \text{EOS\_TOKEN}$
- Finding top-k most likely outputs more involved
  - E.g., most probable sequence may start with a low-probability element $\rightarrow$ greedy methods don't work
  - Common heuristic: beam search

Assumptions needed

- In principle, any distribution $p(\boldsymbol{y})$ can be modeled
  - But: next-element distributions become more and more complex
- Example: binary data, next-element distributions specified using conditional probability tables
  - $p_1(y) \rightarrow$ 2 entries ($p(y = 0)$ and $p(y = 1)$)
  - $p_2(y) \rightarrow$ 4 entries ($p(y = 0|y_1 = 0)$, ...)
  - ...
  - $p(y|y_{1:t-1}) \rightarrow 2^t$ entries $\rightarrow$ exponential increase in parameters
- Need to make further assumptions to reduce complexity
- Example: **Markov chain** of **order** k (*window size*, *context size*)
  - Makes **Markov assumption**

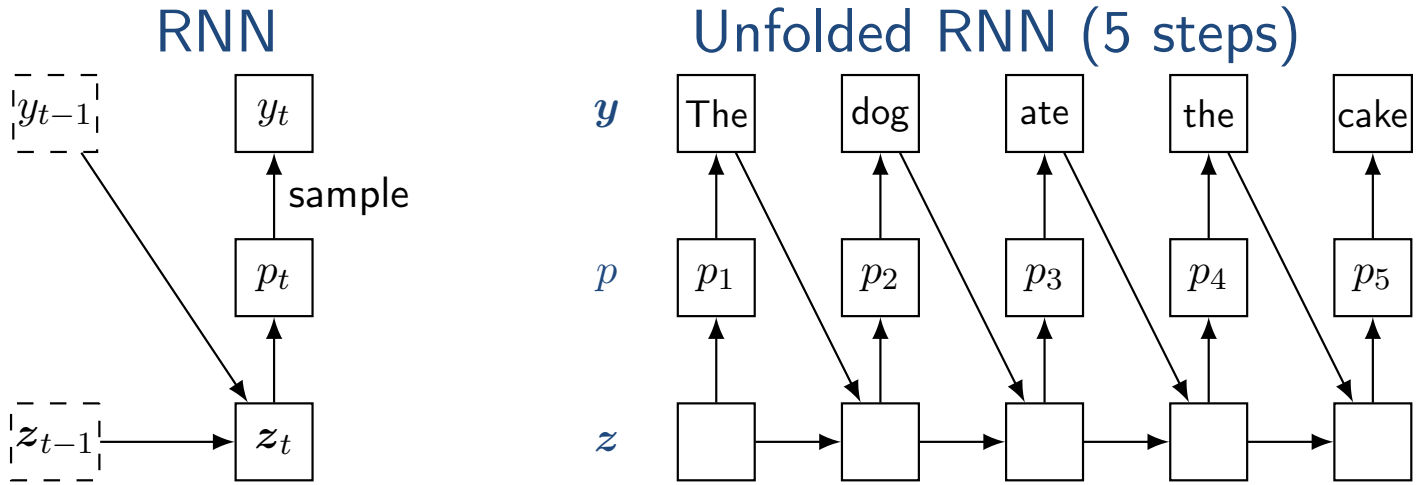
$$p(y|y_{1:t-1}) = p(y|y_{t-k:t-1}) = p(y_{k+1} = y|y_{1:k} = y_{t-k:t-1})$$

  - I.e., only look at k most recent outputs & use same distribution p across time steps
  - For example above, 2^k parameters in total

Outline

Autoregressive modelling using an RNN (1)

- Can use an RNN to specify the next-element distribution $p_t(y)$
  - Must be unidirectional
    $\rightarrow$ Autoregressive (cannot access future outputs)
  - Outputs y_t must be stochastic
    $\rightarrow$ Sampled from probability distribution $p_t(y)$
  - RNN must have output-to-hidden connections
    $\rightarrow$ Else output y_t cond. independent from $y_{1:t-1}$ given $\boldsymbol{z}_{t-1}$
- Example architecture of such an **RNN decoder**



$$p_t(y) = p(y|y_{1:t-1}) = p(y|y_{t-1}, \boldsymbol{z}_{t-1})$$

Autoregressive modelling using an RNN (2)

- Generally: unidirectional, stochastic outputs, output-to-hidden
  - E.g., $\boldsymbol{z}_t = f_{\boldsymbol{\theta}}(\boldsymbol{z}_{t-1}, y_{t-1})$ and y_t sampled from $p_t(y) \stackrel{\text{def}}{=} p_{\boldsymbol{\theta}}(y|\boldsymbol{z}_t)$
  - Example (RNN): LSTM, then $\boldsymbol{z}_t = (\boldsymbol{c}_t, \boldsymbol{h}_t)$
  - Example (discrete output): $p_{\boldsymbol{\theta}}(y|\boldsymbol{z}_t)$ obtained by sampling from categorical distribution (with parameters $\boldsymbol{p}_t = g_{\boldsymbol{\theta}}(\boldsymbol{z}_t)$)
  - Example (continuous output): $p_{\boldsymbol{\theta}}(y|\boldsymbol{z}_t)$ obtained by sampling from normal distribution (with parameters $(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t) = g_{\boldsymbol{\theta}}(\boldsymbol{z}_t)$)
- Computational cost at each time step constant
  - In particular, does not increase over time
- Additional information can be provided and used
  - E.g., for time series, may provide known "inputs" x_t such as date, weekday, time, ... at each time step (cf. slide 50)
  - E.g., condition $\boldsymbol{c}$ in conditional generative models
  - E.g., attention to prior outputs (more later)

Conditional deep autoregressive models

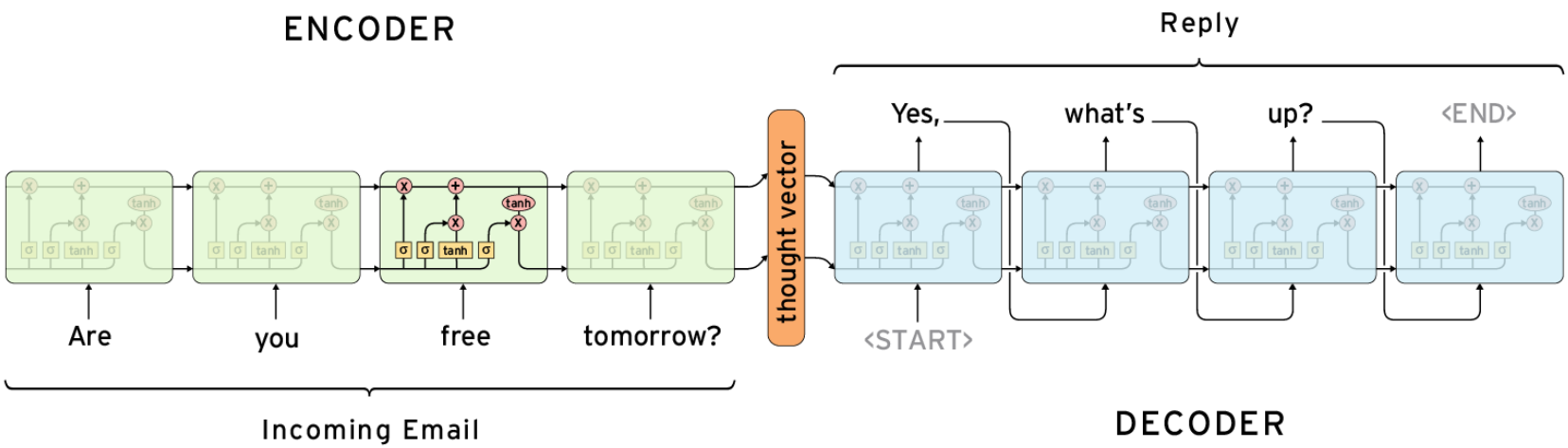
- Deep autoregressive models can also be used as conditional generative models
  - Recall: we model $p(\boldsymbol{y}|\boldsymbol{c})$, where $\boldsymbol{c}$ can be seen as an "input"
- General approach: use input-dependent next-element distributions of form

$$p_t(y) = p(y|y_{1:t-1}, \boldsymbol{c})$$

- Key approaches
  - Encoder-decoder models
  - Decoder-only models

Encoder-decoder models

- **Encoder-decoder models** generally
  - Use an encoder to compute representation $\boldsymbol{z}$ of input $\boldsymbol{c}$
  - Use $\boldsymbol{z}$ to condition the decoder, i.e., $p_t(y) = p(y_t|y_{1:t-1}, \boldsymbol{z})$
  - Generally, input and output may be of different types (A-to-B models)
- Plain **encoder-decoder RNNs** do this by feeding thought vector of encoder as the initial hidden state into the decoder
  - Note: Encoder can be bi-directional
  - Example: Google smart reply (from 2015)



Google Research Blog, 2015

Decoder-only models

- **Decoder-only models** condition on $\boldsymbol{c}$ as a "prior output"

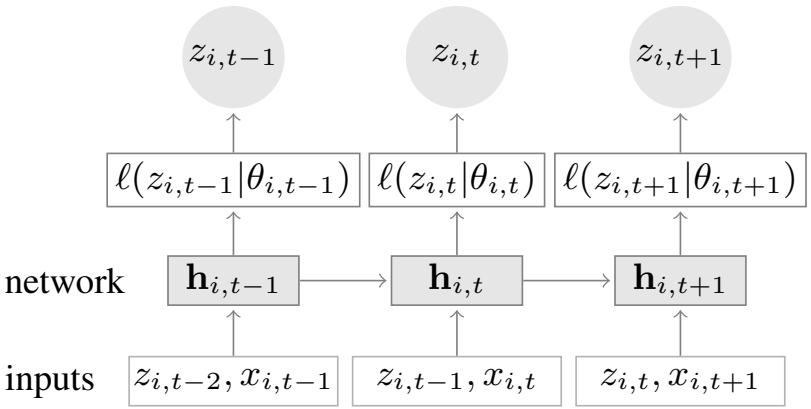
$$p(\boldsymbol{y}|\boldsymbol{c}) = \frac{p(\boldsymbol{c}\|\boldsymbol{y})}{p(\boldsymbol{c})}$$

  - Here $\|$ means concatenation
  - Think: Probability of outputting $\boldsymbol{y}$ after $\boldsymbol{c}$ has already been generated
  - Input and output must be of same type
- Example: forecasting for a time series
  - $\boldsymbol{c}$ = observed past values
  - $\boldsymbol{y}$ = future values
- Example: prompts in (large) language models
  - $\boldsymbol{c}$ = (Are, you, free, tomorrow?, END_OF_INPUT)
  - $\boldsymbol{y}$ = (Yes, what's, up?)
  - $\boldsymbol{c}\|\boldsymbol{y}$ = (Are, you, free, tomorrow?, END_OF_INPUT, Yes, what's, up?)

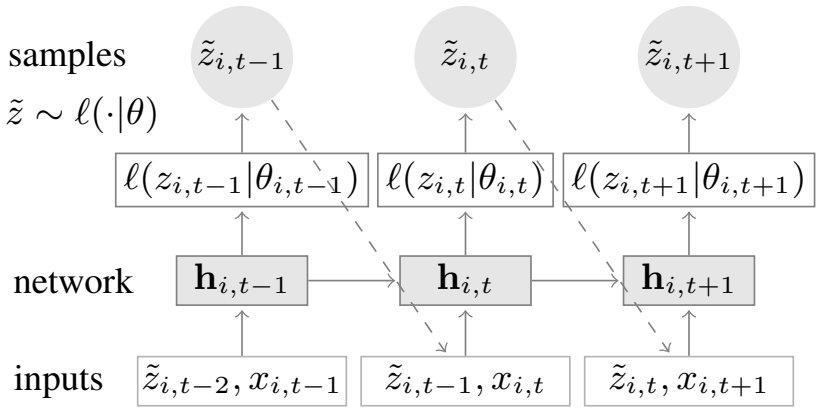
Example: DeepAR (Amazon)

- Probabilistic forecasting based on RNNs with stochastic units
- Focus: scenarios with many related time series
  (energy consumption of individual households, demand of products)
- Allows to fit more complex models, little feature engineering

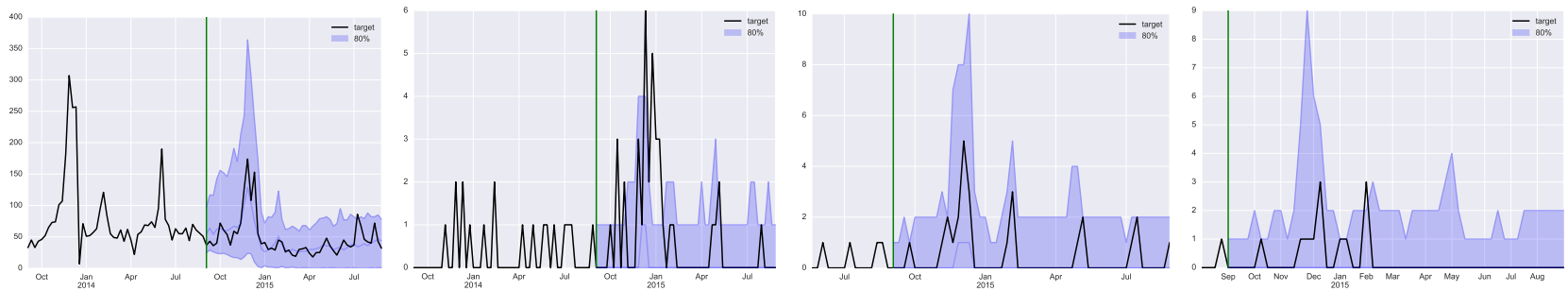
Training



Prediction



Example



Flunkert et al., 2017

Example: RNNs for language modeling (1)

- Language model = distribution over sequences of characters/tokens/words
  - Can use RNN trained to predict next character/token/word
- Fun results with RNNs; e.g., write like Shakespeare:

  Second Senator:
  They are away this miseries, produced upon my soul,
  Breaking and strongly should be buried, when I perish
  The earth and thoughts of many states.

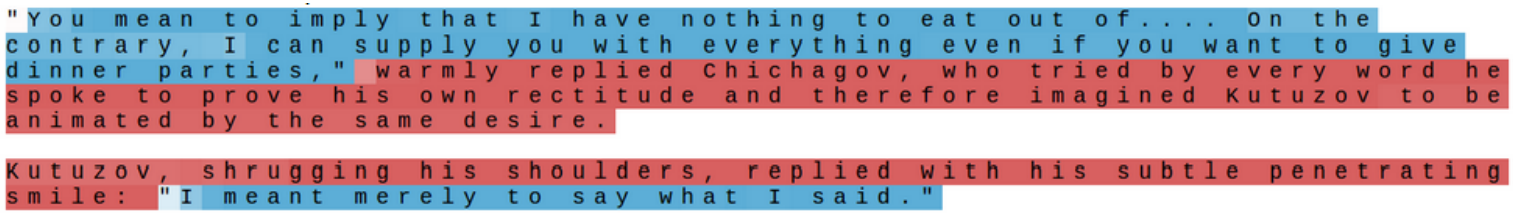
  DUKE VINCENTIO:
  Well, your wit is in the care of side and that.

  Second Lord:
  They would be ruled after this chamber, and
  my fair nues begun out of the fact, to be conveyed,
  Whose noble souls I'll have the heart of the wars.

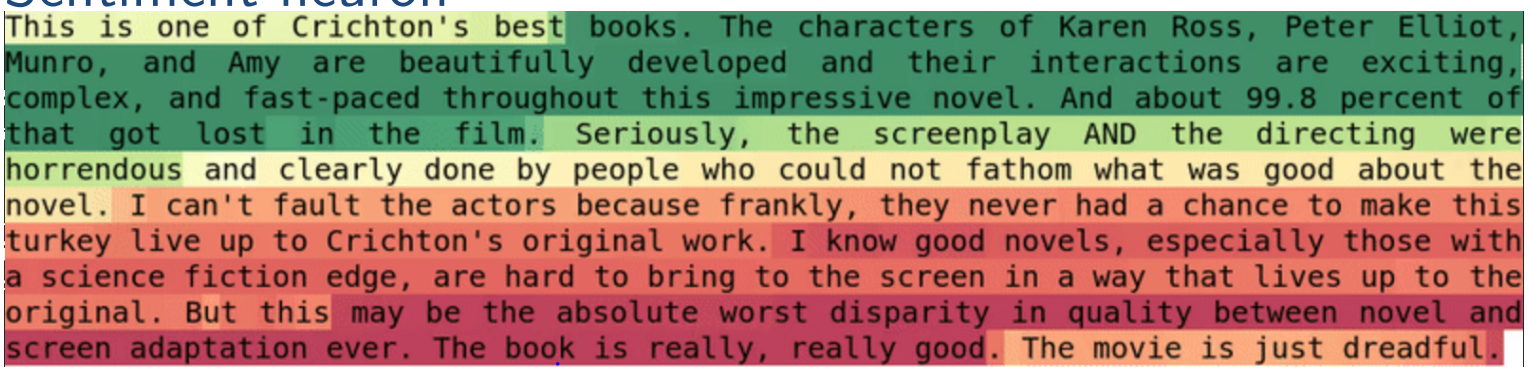
Example: RNNs for language modeling (2)

Visualization of the activation of some particular hidden neuron in each time step of a character-level language model.

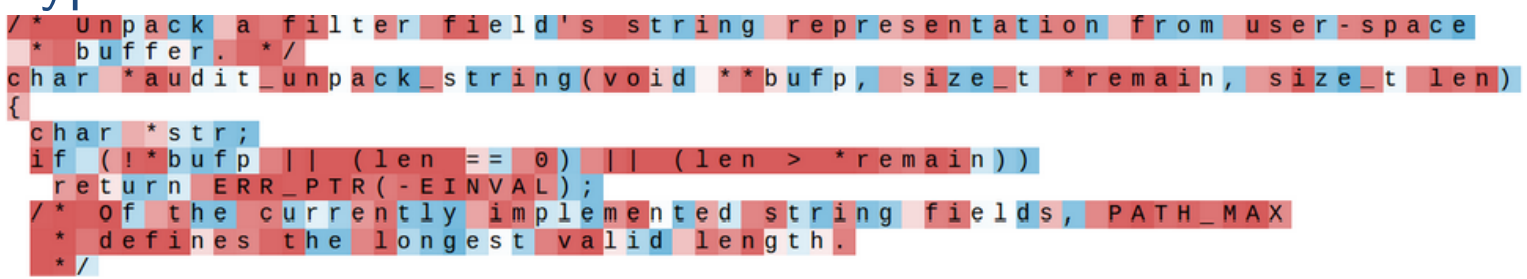
Quotes neuron



Sentiment neuron



Typical neuron



Radford et al., 2017; Karpathy, 2015

Discussion

- Encoder-decoder models
  - Input and output may be of different types
  - Separate models for input and output
  - Trained with suitable input/output examples
- Decoder-only models (also called: **causal decoder-only**)
  - Input and output of same type
  - Same model for input and output (both uni-directional)
  - No need to decide on what is input and output during training
- Intermediate: **non-causal decoder-only models**
  - Input and output of same type
  - Bi-directional on input, uni-directional on output
  - Same model for input/output, but "future" connections on output "cut"
  - Trained with suitable input/output examples
- When to use which? $\rightarrow$ depends...
  - Encoder-decoder models when input/output types/dist. differ
  - Decoder-only models natural for forecasting applications
  - E.g., for NLP tasks, Wang et al. (2022): decoder-only had strong zero-shot performance, but non-causal decoder-only models superior with multi-task training data

Training deep autoregressive models

- Parameters $\boldsymbol{\theta}$ of deep autoregressive models can be learned from training data
- For training example $\boldsymbol{y}^*$, we have

$$p_{\boldsymbol{\theta}}(\boldsymbol{y}^*) = \prod_t p_{\boldsymbol{\theta}}(y_t^* | y_{1:t-1}^*)$$

- Quantities $p_{\boldsymbol{\theta}}(y_t^* | y_{1:t-1}^*)$ correspond to
  - Probability that model generates correct element y_t^* at position t
  - Given that it generated all previous elements correctly
- E.g., for categorical data, may use ERM with log loss

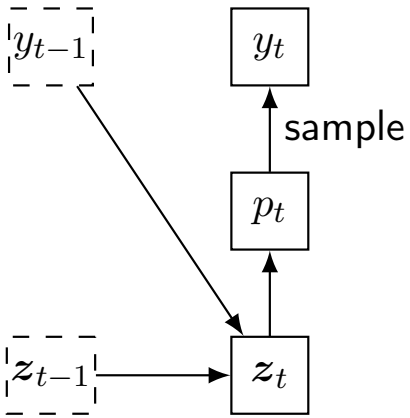
$$-\log p_{\boldsymbol{\theta}}(\boldsymbol{y}^*) = -\sum_t \log p_{\boldsymbol{\theta}}(y_t^* | y_{1:t-1}^*)$$

  - Observe: Model trained for next-element prediction
  - Sometimes referred to as (full) **language modelling objective**
- This approach is called **teacher forcing**
  - Errors in earlier time steps do not propagate (during training)
  - Alternative: use $\prod_t p_{\boldsymbol{\theta}}(y_t^* | y_{1:t-1})$, where $y_{1:t-1}$ are sampled model outputs

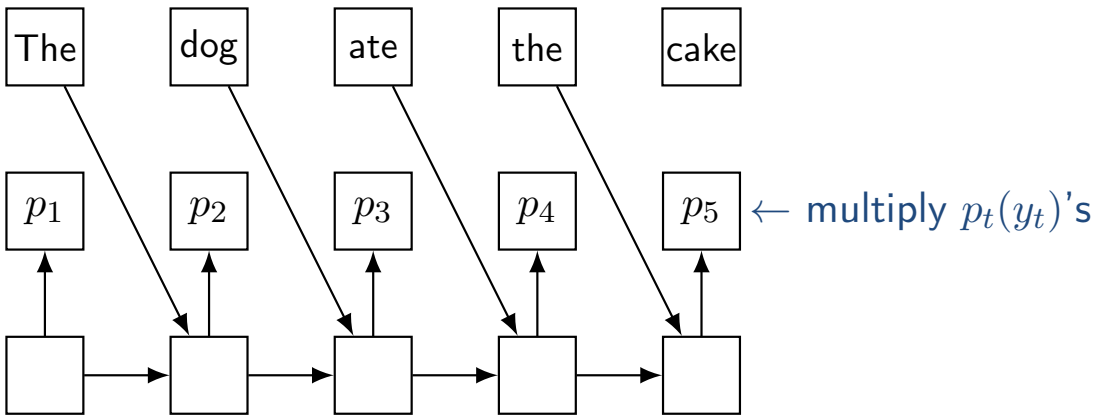
Training RNN decoders

For RNNs, we can compute $p_{\boldsymbol{\theta}}(\boldsymbol{y}^*)$ for a given $\boldsymbol{y}^*$ by omitting the sampling step as follows:

RNN



Unfolded RNN to compute $p(\boldsymbol{y}^*)$



- Note: RNN decoders can also be non-probabilistic
  - Direct prediction, no sampling step
  - Then not a generative model, but produces a fixed sequence for fixed inputs
  - Generally only "useful" for conditional models and/or models with inputs at each time step

Outline

Parallel processing

- GPUs and TPUs are often used for training/inference
  - Have many "processors" that work (somewhat) independently (10,000s per unit)
  - Goal: use all these processors $\rightarrow$ high utilization $\rightarrow$ high efficiency
- Local compute (e.g., MLP) in part-based models
  - Computations for each part are independent
  - Can be processed in parallel (e.g., one processor per part)
- What about contextualization?
  - In CNNs: easy to parallelize, as computations at each "kernel position" are independent (e.g., one processor per position)
  - In RNNs: hard to parallelize, as operation at time step t needs input from time step $t-1$ (e.g., $\boldsymbol{z}_{t-1}$) $\rightarrow$ slow
- Traditionally, one reason to avoid RNNs
  - E.g., in favor of CNNs or Transformers
- Can we do better?
  - Yes, for encoders and when recurrences are linear
  - Decoders: limited as output-to-hidden connections impose sequential processing

Linear recurrences (univariate case)

- Consider a **linear recurrence** of form

$$z_t = az_{t-1} + bx_t \qquad y_t = cz_{t-1} + dx_t$$

  - $a, b, c, d \in \mathbb{R}$ are parameters
  - $x_t \in \mathbb{R}$ is part representation (original input/output of previous layer)
  - $z_t \in \mathbb{R}$ is hidden state $\rightarrow$ linear in z_{t-1} and x_t
  - $y_t \in \mathbb{R}$ is contextualized part representation (output)
    $\rightarrow$ linear in z_{t-1} and x_t (equivalently: z_t and x_t)
  - Each computation takes time $O(1)$

- Suppose initial state $z_0 = 0$; we then have

$$\begin{aligned} z_1 &= bx_1 & y_1 &= dx_1 \\ z_2 &= abx_1 + bx_2 & y_2 &= cdx_1 + dx_2 \\ z_3 &= a^2bx_1 + abx_2 + bx_3 & y_3 &= c^2dx_1 + cdx_2 + dx_3 \\ z_t &= \textstyle\sum_{k=1}^{t} a^{t-k}bx_k & y_t &= \textstyle\sum_{k=1}^{t} c^{t-k}dx_k \end{aligned}$$

- We can parallelize this computation via a parallel scan
  $\rightarrow$ very general method; in this lecture: variant for our setting

Parallel scan (univariate case, algorithm)

Suppose we have P processors. Divide the sequence of length τ into P parts of length $L = \tau/P$.

1. Run the original RNN on each part $1 \le p \le P$ to compute the final hidden states z_L^p, i.e.:

$$z_L^p = \sum_{k=1}^{L} a^{L-k} b x_k^p \qquad \text{where } x_k^p \text{ is } k\text{-th element of } p\text{-th part}$$

2. Run the following linear RNN on these P final states to compute *starting states* s_p for each part $1 \le p \le P$

$$s_1 = 0 \qquad s_{p+1} = a^L s_p + z_L^p$$

3. Run the original RNN again on each part, but this time starting with initial state s_p on each part p
   - Part 1: $s_1 = 0 \to$ state z_0 of original RNN
   - Part 2: $s_2 = z_L^1 = \sum_{k=1}^{L} a^{L-k} b x_k \to$ state z_L of original RNN
   - Part 3: $s_3 = a^L z_L^1 + z_L^2 = \sum_{k=1}^{2L} a^{2L-k} b x_k \to$ state z_{2L} of original RNN
   - ...

We thus obtain the outputs of the original RNN.

Parallel scan (univariate case, discussion)

What did we gain?

1. Run the original RNN on each part
   - Can be run in parallel for each part $\rightarrow$ time $O(\tau/P)$
2. Run a linear RNN to obtain starting states
   - Need $a^L \rightarrow$ time $O(1)$
   - Naively: sequential in time $O(P)$
   - Better (for large P): use another parallel scan to do this
3. Run the original RNN again on each part
   - Can be run in parallel for each part $\rightarrow$ time $O(\tau/P)$

Discussion

- Can use different parameter $a_t \in \mathbb{R}$ instead of a at time step t $\rightarrow$ (more) general parallel scan (used in practice in some models)
- Can show: overall $O(\tau/P + \log P) \rightarrow$ essentially parallel
- Same approach to compute y_t's & for backprop

Linear recurrences (multivariate case)

- Consider a **linear recurrence** of form

$$\boldsymbol{z}_t = \boldsymbol{A}\boldsymbol{z}_{t-1} + \boldsymbol{B}\boldsymbol{x}_t \qquad \boldsymbol{y}_t = \boldsymbol{C}\boldsymbol{z}_{t-1} + \boldsymbol{D}\boldsymbol{x}_t$$

  - $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D} \in \mathbb{R}^{D \times D}$ are parameters
  - $\boldsymbol{x}_t \in \mathbb{R}^D$ is part representation (original input/output of previous layer)
  - $\boldsymbol{z}_t \in \mathbb{R}^D$ is hidden state $\rightarrow$ linear in $\boldsymbol{z}_{t-1}$ and $\boldsymbol{x}_t$
  - $\boldsymbol{y}_t \in \mathbb{R}^D$ is contextualized part representation (output)
    $\rightarrow$ linear in $\boldsymbol{z}_{t-1}$ and $\boldsymbol{x}_t$ (equivalently: $\boldsymbol{z}_t$ and $\boldsymbol{x}_t$)
  - Each computation takes time $O(D^2)$

- Suppose initial state $\boldsymbol{z}_0 = \boldsymbol{0}$; we then have

$$\begin{aligned}
\boldsymbol{z}_1 &= \boldsymbol{B}\boldsymbol{x}_1 & \boldsymbol{y}_1 &= \boldsymbol{D}\boldsymbol{x}_1 \\
\boldsymbol{z}_2 &= \boldsymbol{A}\boldsymbol{B}\boldsymbol{x}_1 + \boldsymbol{B}\boldsymbol{x}_2 & \boldsymbol{y}_2 &= \boldsymbol{C}\boldsymbol{D}\boldsymbol{x}_1 + \boldsymbol{D}\boldsymbol{x}_2 \\
\boldsymbol{z}_3 &= \boldsymbol{A}^2\boldsymbol{B}\boldsymbol{x}_1 + \boldsymbol{A}\boldsymbol{B}\boldsymbol{x}_2 + \boldsymbol{B}\boldsymbol{x}_3 & \boldsymbol{y}_3 &= \boldsymbol{C}^2\boldsymbol{D}\boldsymbol{x}_1 + \boldsymbol{C}\boldsymbol{D}\boldsymbol{x}_2 + \boldsymbol{D}\boldsymbol{x}_3 \\
\boldsymbol{z}_t &= \textstyle\sum_{k=1}^{t} \boldsymbol{A}^{t-k}\boldsymbol{B}\boldsymbol{x}_k & \boldsymbol{y}_t &= \textstyle\sum_{k=1}^{t} \boldsymbol{C}^{t-k}\boldsymbol{D}\boldsymbol{x}_k
\end{aligned}$$

- We can parallelize this computation via a parallel scan
  $\rightarrow$ very general method; in this lecture: variant for our setting

Parallel scan (multivariate case, algorithm)

Suppose we have P processors. Divide the sequence of length τ into P parts of length $L = \tau/P$.

1. Run the original RNN on each part $1 \leq p \leq P$ to compute the final hidden states $\boldsymbol{z}_L^p$, i.e.:

$$\boldsymbol{z}_L^p = \sum_{k=1}^{L} \boldsymbol{A}^{L-k} \boldsymbol{B} \boldsymbol{x}_k^p \qquad \text{where } \boldsymbol{x}_k^p \text{ is } k\text{-th element of } p\text{-th part}$$

2. Run the following linear RNN on these P final states to compute *starting states* $\boldsymbol{s}_p$ for each part $1 \leq p \leq P$

$$\boldsymbol{s}_1 = \boldsymbol{0} \qquad \boldsymbol{s}_{p+1} = \boldsymbol{A}^L \boldsymbol{s}_p + \boldsymbol{z}_L^p$$

3. Run the original RNN again on each part, but this time starting with initial state $\boldsymbol{s}_p$ on each part p
   - Part 1: $\boldsymbol{s}_1 = \boldsymbol{0} \to$ state $\boldsymbol{z}_0$ of original RNN
   - Part 2: $\boldsymbol{s}_2 = z_L^1 = \sum_{k=1}^{L} \boldsymbol{A}^{L-k} \boldsymbol{B} \boldsymbol{x}_k \to$ state $\boldsymbol{z}_L$ of original RNN
   - Part 3: $\boldsymbol{s}_3 = \boldsymbol{A}^L \boldsymbol{z}_L^1 + \boldsymbol{z}_L^2 = \sum_{k=1}^{2L} \boldsymbol{A}^{2L-k} \boldsymbol{B} \boldsymbol{x}_k \to$ state $\boldsymbol{z}_{2L}$ of orig. RNN
   - ...

We thus obtain the outputs of the original RNN.

Parallel scan (multivariate case, discussion)

What did we gain?

1. Run the original RNN on each part
   - Can be run in parallel for each part $\rightarrow$ time $O(D^2\tau/P)$
2. Run a linear RNN to obtain starting states
   - Need $\boldsymbol{A}^L \rightarrow$ time $O(D^3)$
   - Naively: sequential in time $O(D^3 + D^2 P)$
   - Better (for large P): use another parallel scan to do this
3. Run the original RNN again on each part
   - Can be run in parallel for each part $\rightarrow$ time $O(D^2\tau/P)$

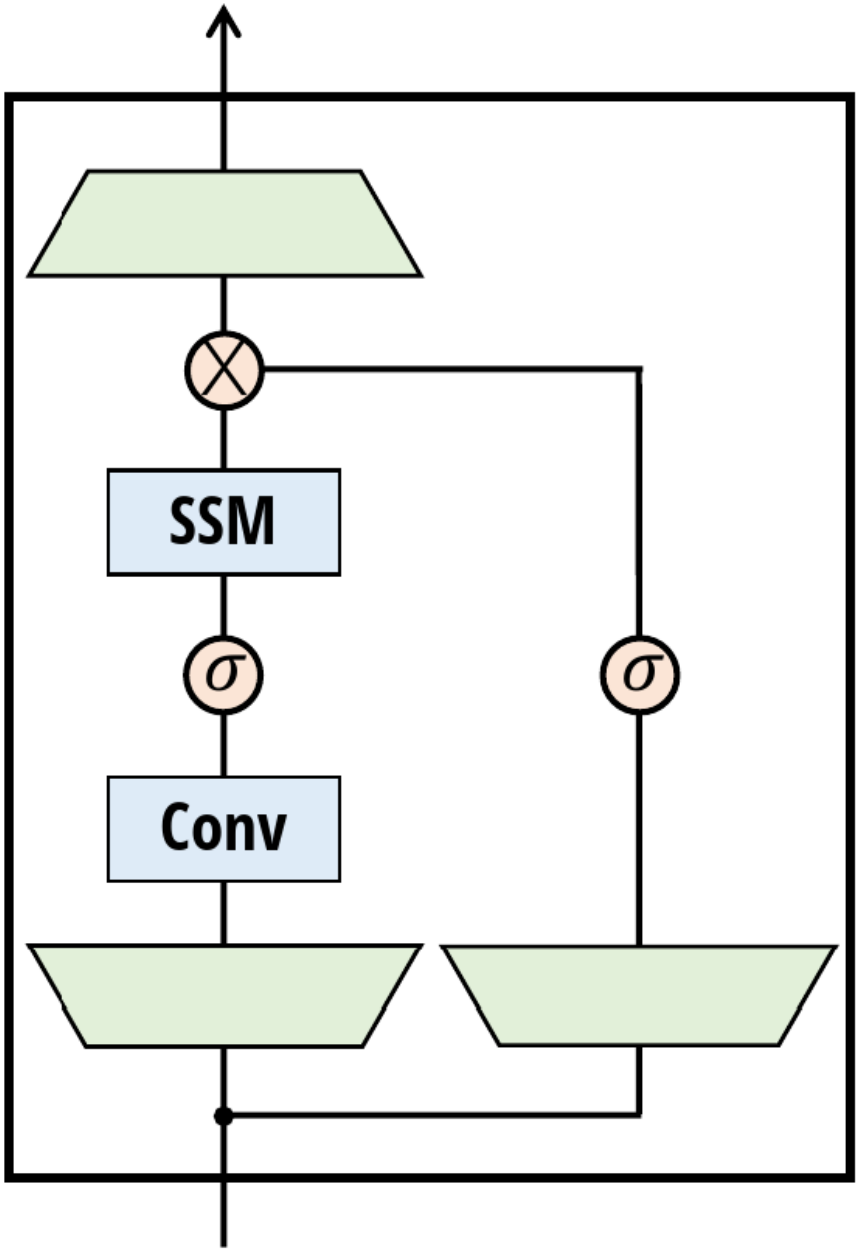
Discussion

- Can use different parameter $\boldsymbol{A}_t \in \mathbb{R}^{D\times D}$ instead of $\boldsymbol{A}$ at time step t
  $\rightarrow$ (more) general parallel scan (used in practice in some models)
- Can show: overall $O((D^3 + D^2)(\tau/P + \log P))$
  $\rightarrow$ For $D \geq P$, **slower than sequential RNN** (!)
- Same approach to compute $\boldsymbol{y}_t$'s & for backprop

Structured state space models

- Problem is essentially the $O(D^3)$ cost of matrix multiplication
- Can we improve? Yes, by imposing *structure* on $\boldsymbol{A}$ (and $\boldsymbol{C}$)
  - E.g.: $\boldsymbol{A}$ (and $\boldsymbol{C}$) diagonal
  - Matrix powers then have cost $O(D) \rightarrow$ problem solved
  - But: less flexible in choice of $\boldsymbol{A}$ (and $\boldsymbol{C}$) $\rightarrow$ tradeoff
- Idea is at the heart of (deep) **structured state space** models (**SSM**)
  - (Discretized) state space model $\approx$ hidden states + linear recurrences
  - Structured $\approx$ restrictions on $\boldsymbol{A}$ (and $\boldsymbol{C}$)
  - Network can be entirely linear...
  - ...or introduce non-linearities in local compute parts (only)
  - May use $\boldsymbol{A}_t$, $\boldsymbol{B}_t$, $\boldsymbol{C}_t$, and $\boldsymbol{D}_t$ "computed" from $\boldsymbol{x}_t$ (e.g., Mamba)
- Many deep SSM architectures exist
  - **Think: modern RNNs**
  - Total cost linear in sequence length & highly parallelizable
  - Empirically: very good performance in sequence modelling tasks, esp. for very long sequences (up to millions of elements)
  - Empirically: parameter efficient
  - Many additional "tricks" (e.g., Centaurus, 2025)

Example: Mamba, 2023 (building block)



Linear projection

Sequence transformation

Nonlinearity (activation or multiplication)

Example: Mamba, 2023 (speech generation)

Table 4: (**SC09**) Automated metrics for unconditional generation on a challenging dataset of fixed-length speech clips. (*Top to Bottom*) Autoregressive baselines, non-autoregressive baselines, Mamba, and dataset metrics.

Model	Params	NLL ↓	FID ↓	IS ↑	mIS ↑	AM ↓
SampleRNN	35.0M	2.042	8.96	1.71	3.02	1.76
WaveNet	4.2M	1.925	5.08	2.27	5.80	1.47
SaShiMi	5.8M	1.873	1.99	5.13	42.57	0.74
WaveGAN	19.1M	-	2.03	4.90	36.10	0.80
DiffWave	24.1M	-	1.92	5.26	51.21	0.68
+ SaShiMi	23.0M	-	1.42	5.94	69.17	0.59
Mamba	6.1M	**1.852**	0.94	6.26	88.54	0.52
Mamba	24.3M	1.860	**0.67**	**7.33**	**144.9**	**0.36**
Train	-	-	0.00	8.56	292.5	0.16
Test	-	-	0.02	8.33	257.6	0.19

Example: Mamba, 2023 (language modelling)

Table 3: (**Zero-shot Evaluations**.) Best results for each size in bold. We compare against open source LMs with various tokenizers trained for up to 300B tokens. Pile refers to the validation split, comparing only against models trained on the same dataset and tokenizer (GPT-NeoX-20B). For each model size, Mamba is best-in-class on every single evaluation result, and generally matches baselines at twice the model size.

Model	Token.	Pile ppl ↓	LAMBADA ppl ↓	LAMBADA acc ↑	HellaSwag acc ↑	PIQA acc ↑	Arc-E acc ↑	Arc-C acc ↑	WinoGrande acc ↑	Average acc ↑
Hybrid H3-130M	GPT2	—	89.48	25.77	31.7	64.2	44.4	24.2	50.6	40.1
Pythia-160M	NeoX	29.64	38.10	33.0	30.2	61.4	43.2	24.1	**51.9**	40.6
Mamba-130M	NeoX	**10.56**	**16.07**	**44.3**	**35.3**	**64.5**	**48.0**	**24.3**	**51.9**	**44.7**
Hybrid H3-360M	GPT2	—	12.58	48.0	41.5	68.1	51.4	24.7	54.1	48.0
Pythia-410M	NeoX	9.95	10.84	51.4	40.6	66.9	52.1	24.6	53.8	48.2
Mamba-370M	NeoX	**8.28**	**8.14**	**55.6**	**46.5**	**69.5**	**55.1**	**28.0**	**55.3**	**50.0**
Pythia-1B	NeoX	7.82	7.92	56.1	47.2	70.7	57.0	27.1	53.5	51.9
Mamba-790M	NeoX	**7.33**	**6.02**	**62.7**	**55.1**	**72.1**	**61.2**	**29.5**	**56.1**	**57.1**
GPT-Neo 1.3B	GPT2	—	7.50	57.2	48.9	71.1	56.2	25.9	54.9	52.4
Hybrid H3-1.3B	GPT2	—	11.25	49.6	52.6	71.3	59.2	28.1	56.9	53.0
OPT-1.3B	OPT	—	6.64	58.0	53.7	72.4	56.7	29.6	59.5	55.0
Pythia-1.4B	NeoX	7.51	6.08	61.7	52.1	71.0	60.5	28.5	57.2	55.2
RWKV-1.5B	NeoX	7.70	7.04	56.4	52.5	72.4	60.5	29.4	54.6	54.3
Mamba-1.4B	NeoX	**6.80**	**5.04**	**64.9**	**59.1**	**74.2**	**65.5**	**32.8**	**61.5**	**59.7**
GPT-Neo 2.7B	GPT2	—	5.63	62.2	55.8	72.1	61.1	30.2	57.6	56.5
Hybrid H3-2.7B	GPT2	—	7.92	55.7	59.7	73.3	65.6	32.3	61.4	58.0
OPT-2.7B	OPT	—	5.12	63.6	60.6	74.8	60.8	31.3	61.0	58.7
Pythia-2.8B	NeoX	6.73	5.04	64.7	59.3	74.0	64.1	32.9	59.7	59.1
RWKV-3B	NeoX	7.00	5.24	63.9	59.6	73.7	67.8	33.1	59.6	59.6
Mamba-2.8B	NeoX	**6.22**	**4.23**	**69.2**	**66.1**	**75.2**	**69.7**	**36.3**	**63.5**	**63.3**
GPT-J-6B	GPT2	–	4.10	68.3	66.3	75.4	67.0	36.6	64.1	63.0
OPT-6.7B	OPT	–	4.25	67.7	67.2	76.3	65.6	34.9	65.5	62.9
Pythia-6.9B	NeoX	6.51	4.45	67.1	64.0	75.2	67.3	35.5	61.3	61.7
RWKV-7.4B	NeoX	6.31	4.38	67.2	65.5	76.1	67.8	37.5	61.0	62.5

Deep Learning

08 – Attention and Transformers

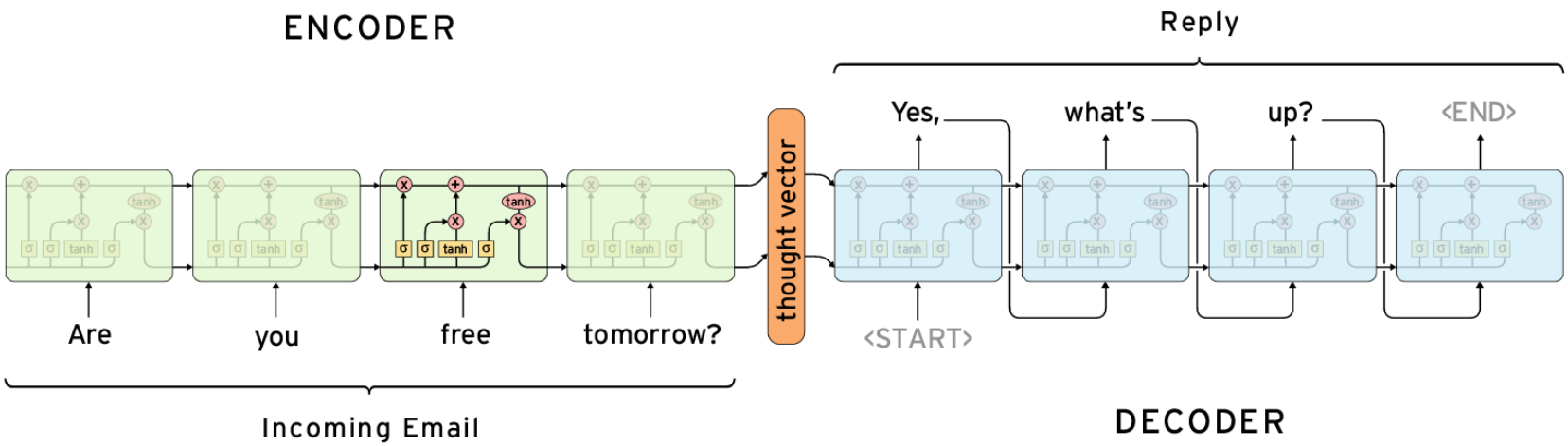
Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-3

Outline

Recap: RNN encoder-decoder architecture



- Network first "reads" the input $\boldsymbol{x}$ = **encoding**
  - Encodes relevant information into a global representation, say, the **thought vector** $\boldsymbol{z}$
- Network then "generates" the output $\boldsymbol{y}$ = **decoding**
  - Output (distribution of) $\boldsymbol{y}$ solely based on thought vector $\boldsymbol{z}$
  - Inputs $\boldsymbol{x}$ of encoder not accessed: output $\boldsymbol{y}$ cond. independent of input $\boldsymbol{x}$ given thought vector $\boldsymbol{z}$, i.e.,

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = p(\boldsymbol{y} \mid \boldsymbol{x}, \boldsymbol{z}) = p(\boldsymbol{y} \mid \boldsymbol{z})$$

- This is often too limiting!

Google Research Blog, 2015

Motivating example: The center-number task

- Consider the following *center-number task*
  - Input: a sequence $x_1, \ldots, x_\tau$ of numbers, where τ varies
  - Desired output: $x_{\tau/2}$, i.e., the number in the center position
  - E.g., $\begin{pmatrix} 4 & 2 & 3 & 5 & 1 \end{pmatrix} \rightarrow 3$
  - E.g., $\begin{pmatrix} 4 & 2 & 3 & 5 & 1 & 8 & 9 & 6 & 7 \end{pmatrix} \rightarrow 1$
- Cannot be solved with a (unidirectional) encoder-decoder RNN
  - To enable perfect prediction, after reading t numbers, thought vector needs to encode the last $t/2$ numbers $x_{t/2}, \ldots, x_t$
  - Why? Each of these integers may be a potential output (e.g., $x_{t/2}$ for $\tau = t$, x_t for $\tau = 2t$)
  - Generally impossible since **thought vector has fixed dimensionality**
- Task trivial if decoder could access the input again
  - Thought vector then only needs to encode length τ of input
  - Decoder "computes" $\tau/2$, then accesses and outputs input $x_{\tau/2}$
  - That's the key idea of RNNs with attention
  - We say: the decoder **attends** to input $x_{\tau/2}$

Attention

- **Attention** groups multiple inputs into a *fixed-length* representation
  - Inputs are *unordered* (i.e., form a multiset)
  - *Number* of inputs *not fixed*, but may vary
  - E.g., τ elements in $\mathbb{R}^D$ to one value in $\mathbb{R}^D$
- Grouping is
  - **Simple**, e.g., a weighted average
  - Often **focused** on a small subset of the inputs
  - **Dynamic** in that it depends on network state and values of inputs
  - E.g., in the decoder of an RNN, relevant inputs (high attention weights) for each output element may change over time (cf. sl. 6)
- Generally, a mechanism for network design; many variants
  - Can be used with RNNs, CNNs, standalone, ...
  - Often, in a part-based model, the inputs to attention correspond to the *parts*
- Our focus: attention for encoder-decoder RNNs (first), self-attention/Transformers (afterwards)

Examples

Translation

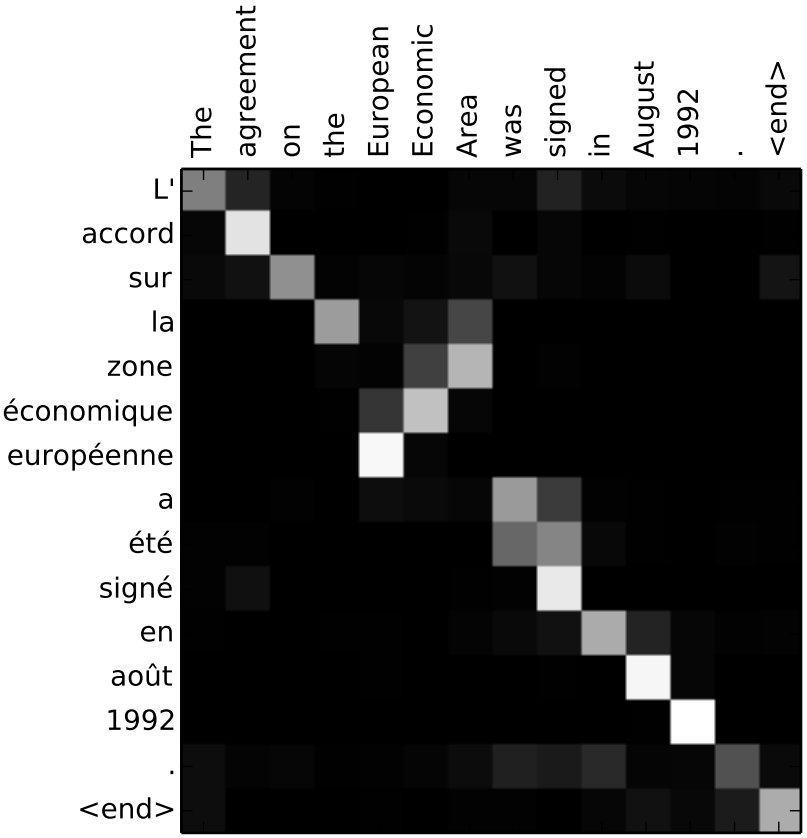


Image captioning

A woman is throwing a frisbee in a park.

Bahdanau and Cho, 2015; Xu et al., 2016

Why attention?

- Attention provides "access" to all inputs (soft memory)
  - No fixed-length representations as in a pure encoder-decoder RNN
  - I.e., longer inputs $\rightarrow$ larger representation
- Attention is dynamic and learned
  - Different parts of input may be relevant for different output elements $\rightarrow$ dynamic grouping
  - Often hard to determine apriori which inputs are relevant $\rightarrow$ learned grouping
- Attention is general
  - Arbitrary multiset inputs (not necessarily sequence or grid data)
  - Positional information can be (and often needs to be) added: { (1, Positional), (2, information), (3, can), (4, be), (5, added) }
- Attention is easy to parallelize
  - A flat operation: network depth independent of input length
  - Unlike RNNs (recall: in general, no parallelization over time possible)
- Component of state-of-the-art models across many domains

Associative memory

- Suppose we store the input set in associative memory
  - Recall: associative memory stores **keys** and corresponding **values**
  - E.g., processor cache: key = memory address, value = content of memory cell
  - E.g., for an RNN, key = time step, value = part

Key k	Value v
1	The
2	dog
3	ate
4	the
5	cake

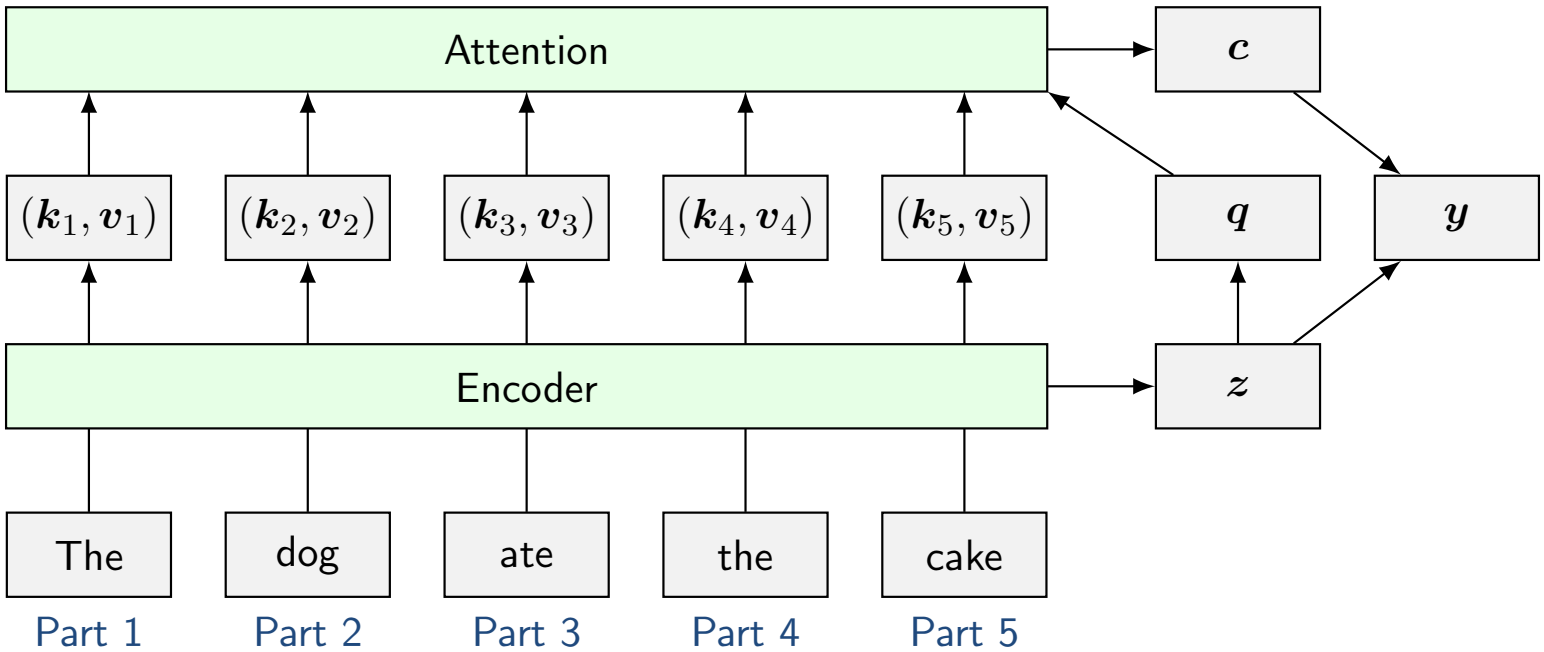
- We can then retrieve a value by for a given **query**
  - E.g., processor cache: query = memory address
  - E.g., query = 3 $\rightarrow$ output = ate
- Here we assigned to each input x_i ...
  - A key k_i that **describes** input i
  - A value v_i that **represents** input i
- ...and then used
  - A query q_i to express what is considered **relevant**

Attention as soft memory

- Attention can be viewed as a soft form of associative memory
  - We use neural representations to represent keys, values, and query
  - All these representations are learned
  - **Keys** are learned descriptions of the input (instead of addresses)
  - **Values** learned representations of the input elements (instead of the elements themselves)
  - A **query** is a learned description of the information need (instead of an "address" or "time step")
- Let's represent input $x_1, \ldots, x_\tau$ and query as follows:
  - Keys $\boldsymbol{k}_1, \ldots, \boldsymbol{k}_\tau \in \mathbb{R}^{d_K}$
  - Values $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_\tau \in \mathbb{R}^{d_V}$
  - Query $\boldsymbol{q} \in \mathbb{R}^{d_K}$
  - Note: keys and values will be "computed" by some neural network
  - Note: Dimensionalities of query/keys (d_K) and values (d_V) may differ
- How do we "answer" a query?
  - Answer is a "grouping" of values $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_\tau \in \mathbb{R}^{d_V}$
  - Typically: Answer $\boldsymbol{c} \in \mathbb{R}^{d_V}$ has same shape as a value

Attention over inputs (encoder-only model)

The general approach for attention in a part-based model is as follows:



1. Encoder computes keys, values, and global representation
2. Prediction head computes query from global representation
3. Attention is used to obtain an answer, called **context vector**
4. Final prediction obtained from global representation *and context vector*

Attention

Let's fill in the details. To answer a query, we

1. Use an **attention model** $a(\boldsymbol{q}, \boldsymbol{k})$
   - Measures the "compatibility" of each input key $\boldsymbol{k}_i$ to the query $\boldsymbol{q}$ via **attention score**

     $$e_i = a(\boldsymbol{q}, \boldsymbol{k}_i)$$

     to form an attention score vector $\boldsymbol{e} \in \mathbb{R}^\tau$
   - Higher/lower score $\rightarrow$ more/less relevant
   - E.g., dot product: $a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{k}^\top \boldsymbol{q}$
   - E.g., small MLP: $a(\boldsymbol{q}, \boldsymbol{k}) = f_{\boldsymbol{\theta}}(\boldsymbol{q}, \boldsymbol{k})$
2. Compute query-dependent **attention weights** $\boldsymbol{\alpha}$ (of size τ)
   - E.g., **soft attention**: $\boldsymbol{\alpha} = S(\boldsymbol{e}) \in \mathcal{S}_\tau \rightarrow$ weighted average
   - E.g., **hard attention**: $\alpha_i = 1$ for largest score e_i, else 0
   - E.g., **linear attention**: $\boldsymbol{\alpha} = \boldsymbol{e} \in \mathbb{R}^\tau \rightarrow$ weighted sum
3. Output a **context vector** $\boldsymbol{c}$, obtained by weighting each value by its attention weight

   $$\boldsymbol{c} = \sum_i \alpha_i \boldsymbol{v}_i,$$

Example (dot-product attention)

- E.g., key elements: noun or verb? / subject or object? / random

Input x_i	Key $\boldsymbol{k}_i^\top$	Value $\boldsymbol{v}_i^\top$
The	$(0.1 \quad 0 \quad 2)$	$\cdots$
dog	$(5 \quad 4 \quad 3)$	$\cdots$
ate	$(-4 \quad 0 \quad -3)$	$\cdots$
the	$(-0.2 \quad 0 \quad 1)$	$\cdots$
cake	$(5 \quad -3 \quad -4)$	$\cdots$

- Queries for (i) nouns, (ii) verbs, (iii) subjects, (iv) bag of words

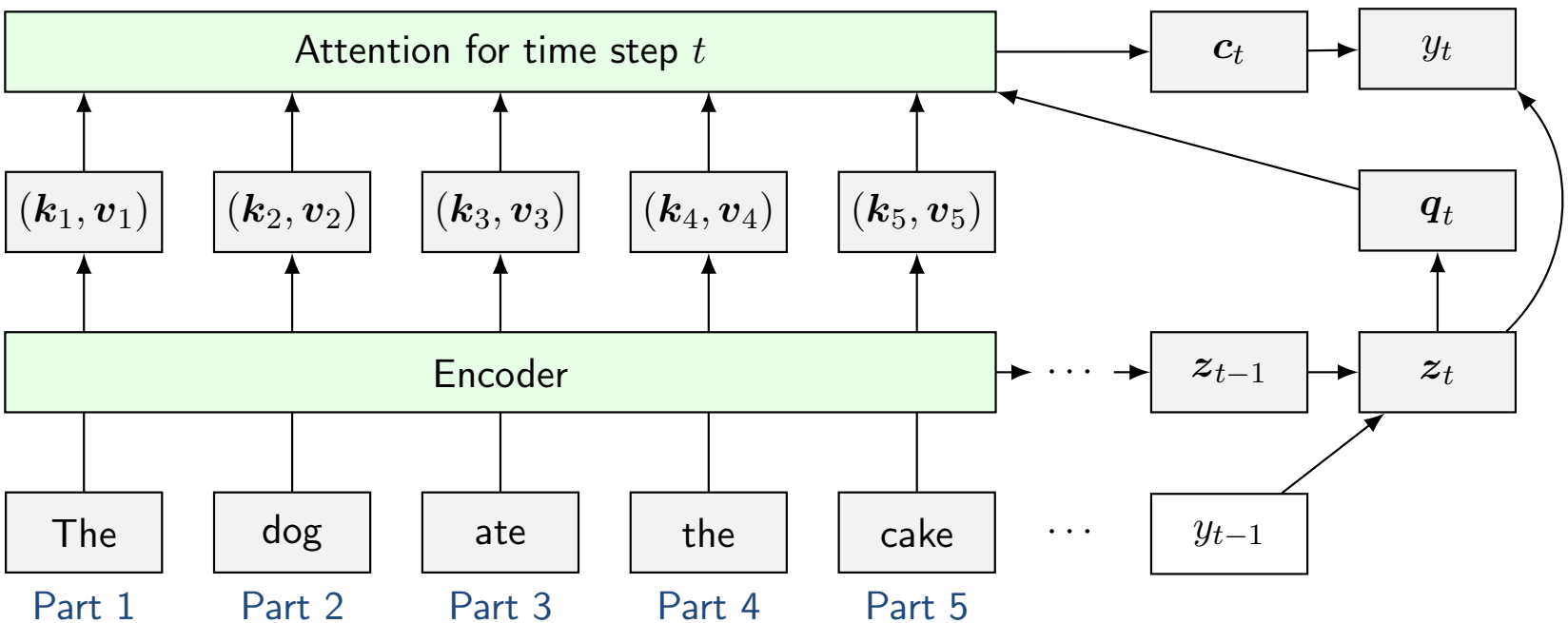
	Query $\boldsymbol{q}^T$	Scores $\boldsymbol{e}^\top$	Weights $\boldsymbol{\alpha}^\top$
(i)	$(5 \quad 0 \quad 0)$	$(0.5 \quad 25 \quad -20 \quad -1 \quad 25)$	$(0 \quad 0.5 \quad 0 \quad 0 \quad 0.5)$
(ii)	$(-5 \quad 0 \quad 0)$	$(-0.5 \quad -25 \quad 20 \quad 1 \quad -25)$	$(0 \quad 0 \quad 1 \quad 0 \quad 0)$
(iii)	$(5 \quad 4 \quad 0)$	$(0.5 \quad 41 \quad -20 \quad -1 \quad 13)$	$(0 \quad 1 \quad 0 \quad 0 \quad 0)$
(iv)	$(0 \quad 0 \quad 0)$	$(0 \quad 0 \quad 0 \quad 0 \quad 0)$	$(0.2 \quad 0.2 \quad 0.2 \quad 0.2 \quad 0.2)$

Attention over inputs (encoder-decoder model)

We can use attention over inputs also in a decoder.

- Now: Attention run once *per output element*
- At time step t, use query $\boldsymbol{q}_t \rightarrow$ different context vector $\boldsymbol{c}_t$ at each time step $\rightarrow$ dynamic
- Known as **cross attention**: one model (here: the decoder) attends over the outputs of another model (here: the encoder)

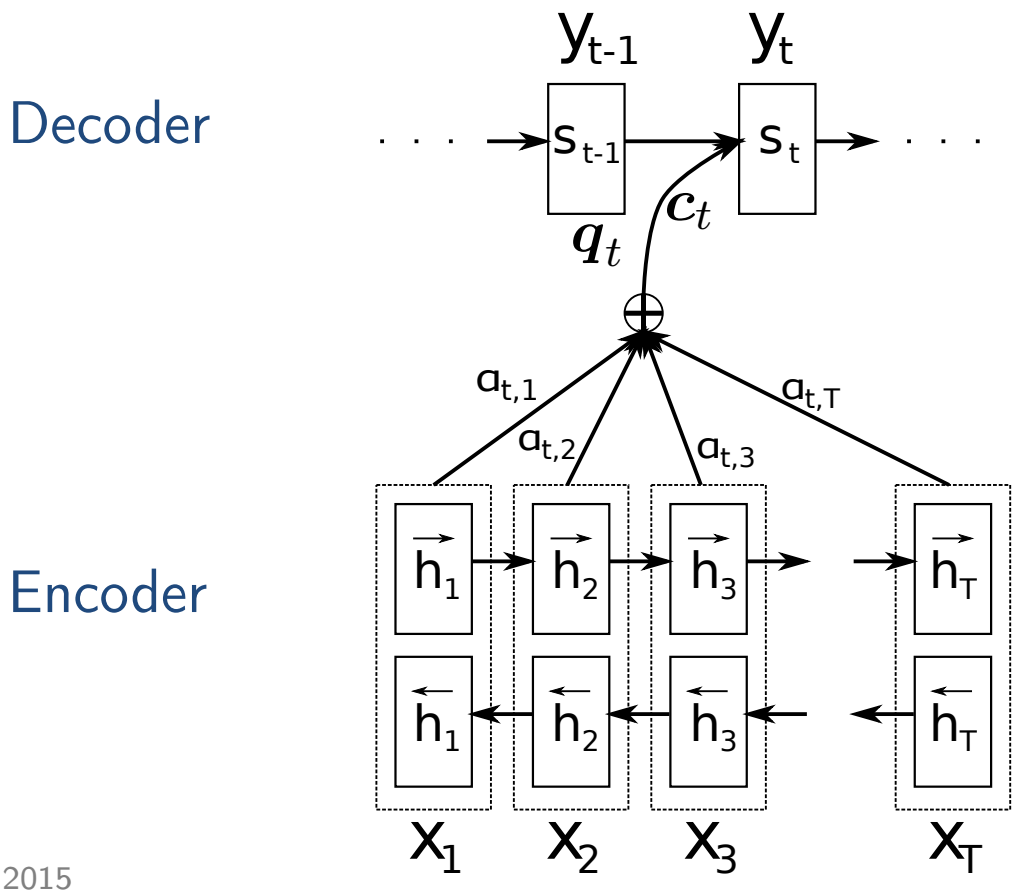
Example: compute $\boldsymbol{q}_t$ from hidden state $\boldsymbol{z}_t$



Example (early architecture)

Architecture of Bahdanau and Cho (2015) for machine translation

- Uses $\boldsymbol{k}_i = \boldsymbol{v}_i = (\overrightarrow{\boldsymbol{h}_i}, \overleftarrow{\boldsymbol{h}_i}) \rightarrow$ hidden states of bidirect. RNN encoder
- Uses $\boldsymbol{q}_t = \boldsymbol{z}_{t-1} \rightarrow$ hidden state of RNN decoder
- MLP as attention model
- Example weights shown on slide 6



Discussion

- Attention allows RNN to focus on important parts of input
  - More powerful in that available information is not limited by the size of the hidden representation
  - Access to input simplifies architecture and learning
  - Exercise: write an RNN with attention for the central-number task
- In general, can attend to
  - The prior encoder outputs in an RNN encoder
  - The encoder outputs in an RNN decoder (cross-attention)
  - The prior decoder outputs in an RNN decoder
  - Or any combination of the above (individually or jointly)
- Many successful architectures and applications
  - Part of many SOTA architectures for NLP, signal processing, vision, neural programming
  - E.g., content-based addressing in Neural Turing Machines
  - E.g., self-attention instead of RNNs (coming up next)

Outline

Attention is all you need

- So far, we discussed attention in the context of an RNN
  - But is an RNN actually needed?
- Vaswani et al., NIPS 2017: Attention Is All You Need
  - Introduced self-attention, multi-head attention, Transformers
  - Attention only: no recurrence, no convolution
  - Achieved SOTA results (at the time) on machine translation tasks
  - Achieved lower training costs than competitive prior models
  - Architecture facilitates parallel processing
- Highly influential (>170k cites) & important part of toolbox today
  - E.g., BERT for text representation
  - E.g. GPT-1/2/3/4 for language modelling
  - E.g., Vision Transformer for computer vision tasks
  - E.g., wav2vec 2.0 for speech processing
  - E.g., GROVER for molecular data (graphs)
  - E.g., PatchTST for time series forecasting
- Coming up: vanilla architecture and key concepts
- Later lectures: applications, concrete architectures, (pre-)training, ...

Dot-product attention as a layer

- Let's stack the keys and values into matrices
  - Given: $\boldsymbol{q} \in \mathbb{R}^{d_K}$, $\boldsymbol{K} \in \mathbb{R}^{\tau \times d_K}$, $\boldsymbol{V} \in \mathbb{R}^{\tau \times d_V}$
- Dot-product attention scores: $\boldsymbol{e} = \boldsymbol{K}\boldsymbol{q} \in \mathbb{R}^{\tau}$
- Soft attention: $\boldsymbol{\alpha} = S(\boldsymbol{K}\boldsymbol{q})$
- **Dot-product attention**

$$\mathrm{DPA}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = S(\boldsymbol{K}\boldsymbol{q})^\top \boldsymbol{V} \in \mathbb{R}^{d_V}$$

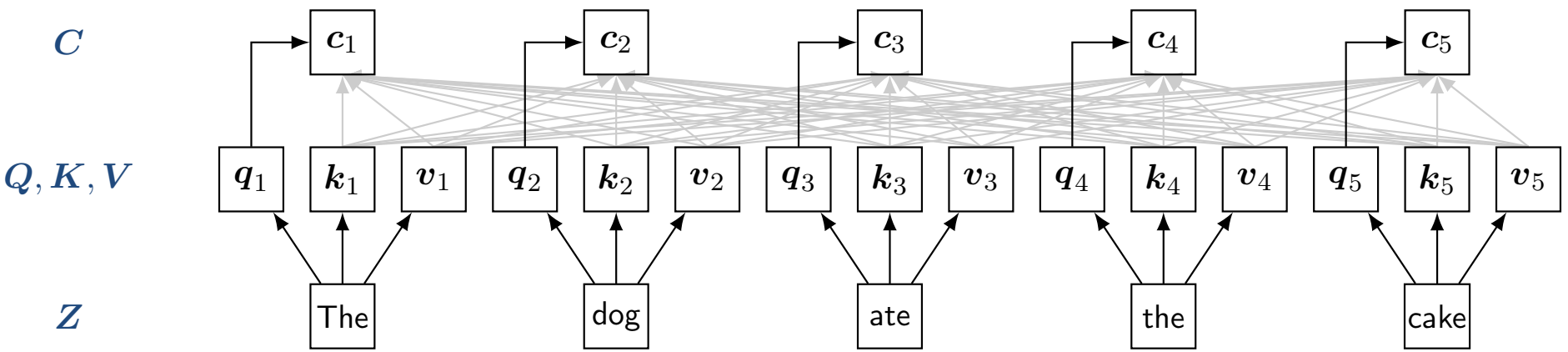
- **Scaled dot-product attention** (used in original transformer)

$$\mathrm{ScaledDPA}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = S(\boldsymbol{K}\boldsymbol{q}/\sqrt{d_K})^\top \boldsymbol{V}$$

  - Vasami et al: *"We suspect that for large values d_K, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counteract this effect, we scale the dot products by $\sqrt{d_k}$."*
  - More in exercise

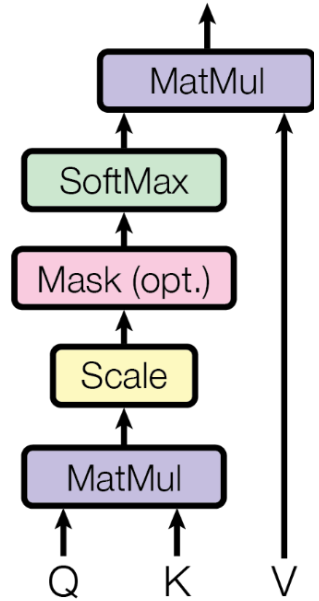
Self-attention

- **Self-attention** refers to a type of *layer* for contextualization
  - Inputs are element representations $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_\tau \in \mathbb{R}^d$ (i.e , "parts")
  - Outputs are contextualized element representation $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_\tau \in \mathbb{R}^{d_V}$
  - τ may vary between inputs
- Outputs are computed from inputs using attention
  - One query $\boldsymbol{q}_i$, key $\boldsymbol{k}_i$, value $\boldsymbol{v}_i$ per input i, each computed from $\boldsymbol{z}_i$
    $\rightarrow$ This motivates the name *self*-attention
  - Query $\boldsymbol{q}_i$ determines what is relevant for the i-th output
    $\rightarrow$ Output $\boldsymbol{c}_i$ "corresponds" to input $\boldsymbol{z}_i$
  - Key $\boldsymbol{k}_i$ serve as description of the i-th input
  - Value $\boldsymbol{v}_i$ serve as representation of the i-th input
- We have $\boldsymbol{c}_i = \mathrm{Attention}(\boldsymbol{q}_\mathrm{i}, \boldsymbol{K}, \boldsymbol{V})$



Single-head attention

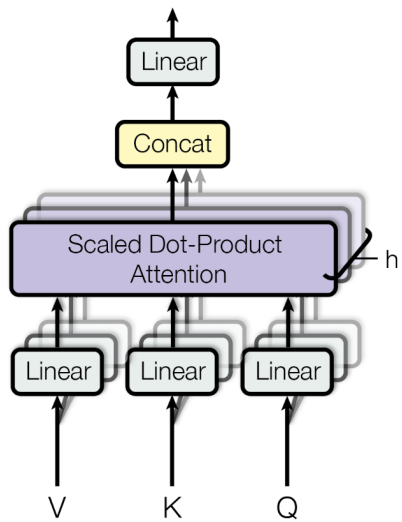
Scaled DPA



- Compact notation: $\boldsymbol{C} = \mathrm{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V})$
  - $\boldsymbol{Q} \in \mathbb{R}^{\tau \times d_K}$ and $\boldsymbol{C} \in \mathbb{R}^{\tau \times d_V}$ contain queries and outputs as rows
  - E.g., $\mathrm{DPA}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = S_{\mathsf{row}}(\boldsymbol{Q}\boldsymbol{K}^\top)\boldsymbol{V}$
  - E.g., $\mathrm{ScaledDPA}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = S(\boldsymbol{Q}\boldsymbol{K}^\top/\sqrt{d_K})\boldsymbol{V}$
- Let $\boldsymbol{Z}^{\tau \times d}$ be the input representations
- Example: plain self-attention (not used this way)
  - $\boldsymbol{Q} = \boldsymbol{Z}$; $\boldsymbol{K} = \boldsymbol{Z}$, $\boldsymbol{V} = \boldsymbol{Z}$
  - Generally not a good idea (exercise)
- Example: **single-head attention**
  - Use linear projections (optionally also biases)
  - $\boldsymbol{Q} = \boldsymbol{Z}\boldsymbol{W}_Q$ where $\boldsymbol{W}_Q \in \mathbb{R}^{d \times d_K}$
  - $\boldsymbol{K} = \boldsymbol{Z}\boldsymbol{W}_K$ where $\boldsymbol{W}_K \in \mathbb{R}^{d \times d_K}$
  - $\boldsymbol{V} = \boldsymbol{Z}\boldsymbol{W}_V$ where $\boldsymbol{W}_V \in \mathbb{R}^{d \times d_V}$
  - $\boldsymbol{W}_Q$, $\boldsymbol{W}_K$, and $\boldsymbol{W}_V$ are parameters
  - d_K and d_V are hyperparameters (typically $d_K = d_V = d$)

Multi-head attention

- **Multi-head attention**: attend multiple times on same inputs
  - Terminology: **head** = one single-head attention layer
  - Each head $h \in \{1, \ldots, H\}$ uses its own parameters $\boldsymbol{W}_Q^{(h)}, \boldsymbol{W}_K^{(h)} \in \mathbb{R}^{d \times d_K}$ and $\boldsymbol{W}_V^{(h)} \in \mathbb{R}^{d \times d_V}$
  - Outputs $\boldsymbol{C}^{(h)}$ of each head are concatenated
  - And then projected to input space using parameter $\boldsymbol{W}_O \in \mathbb{R}^{Hd_v \times d}$
  - Original Transformer: $d_K = d_V = d/H$ (with $d = 512$ and $H = 8$)
  - Then: number of parameters/cost similar to single-head attention
- Beneficial because model can choose which representation subspaces to attend to in a position-dependent way



Computation and path lengths

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

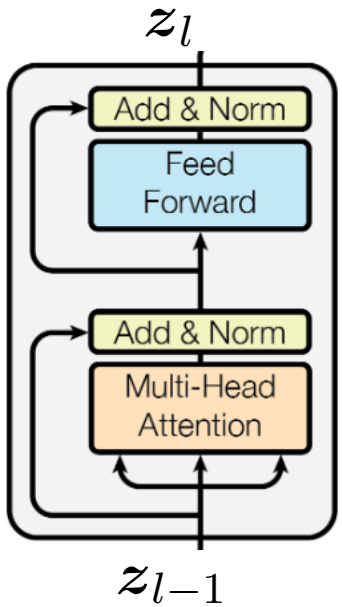
(lower is better)

- Computational complexity
  - Self-attention scales well with dimensionality but not sequence length
  - Recurrent/convolutional: the other way around
  - **But note:** table misleading, self-attention is really $O(n^2 d + nd^2)$ (due to computation of queries/keys/values, e.g., when $d_K = d/H$)
- Sequential operations limit parallelizability $\rightarrow$ drawback of RNNs
- Larger path length between inputs and outputs makes it harder to exploit long-range dependencies $\rightarrow$ advantage of self-attention
- As discussed, mitigated by linear RNNs / SSMs / RNN with attention

Vaswani et al., 2017

Transformer encoder layer

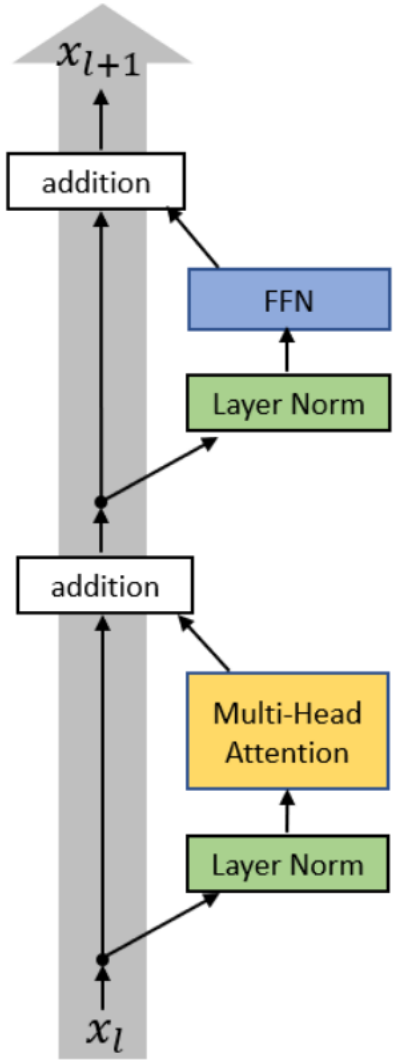
- **Transformer encoder layer** (*encoder block*)
  - Input: lower-level element representations $\boldsymbol{z}^{l-1} = \left\{ \boldsymbol{z}_1^{l-1}, \ldots, \boldsymbol{z}_\tau^{l-1} \right\}$
  - Output: higher-level element representations $\boldsymbol{z}^{l} = \left\{ \boldsymbol{z}_1^{l}, \ldots, \boldsymbol{z}_\tau^{l} \right\}$
- Two steps: (1) contextualization, (2) local compute
  1. Multi-head attention across elements
     $\rightarrow$ incorporate information from other elements
  2. Local MLP for each element
     $\rightarrow$ update each element individually



- Residual connections / layer normalization in between (cf. 03-3)
  - Both attention and MLP updates do not overwrite but "modify" the element representation (additively)
- Every element representation depends on *all other* elements (cf. bidirectional RNN)
- Template model, parameter sharing, position-independence
  - Same projections for queries/keys/values across input elements
  - Same parameters of MLP for each input element

Example: T5 encoder

- Uses **pre-layer normalization** (Xiong et al., 2020)
  - Layer norm right before each subcomponent
  - Stabilizes training
- Main path also known as **residual stream**
  - Updated additively by each subcomponent
- Concrete components for T5-Base
  - $d = 768$-dimensional element representations
  - $H = 12$ heads, each with key/value dimensionality for each head is $d_K = d_V = 64$ (=768 in total)
  - MLP: dense linear layer (to 3072D) $\rightarrow$ ReLU $\rightarrow$ dense linear layer (to 786D)
  - $L = 12$ such T5 encoder layers **stacked on top of each other**



Examples on how use to a Transformer encoder

- Transformer encoders produce contextualized representations $\boldsymbol{z}_i^L$ of input elements $\boldsymbol{x}_i$
  - Used as input representation for subsequent modules
  - Many different ways to do this
- As in part-based models for **element-level tasks**, may
  - Add element-wise prediction head h on top of each $\boldsymbol{z}_i^L$
  - Train supervised with ERM using element-level loss $L(h(\boldsymbol{z}_i^L), y_i^*)$
- As in part-based models for **sequence-level tasks**, may
  - Pool element-level representations (e.g., sum, mean, attention) to obtain sequence-level representation $\boldsymbol{z}$
  - Add prediction head h, train supervised with ERM using loss $L(h(\boldsymbol{z}), y^*)$
- Alternative: use special **classification token CLS**
  - Input data: $x_1, \ldots, x_\tau$
  - Input to encoder: $\text{CLS}, x_1, \ldots, x_\tau$
  - Add prediction head on only $\boldsymbol{z}_{\text{CLS}}^L$, which serves as a sequence-level representation
  - Train supervised using $L(h(\boldsymbol{z}_{\text{CLS}}^L), y^*)$

Outline

From sets to sequences

- Transformer encoders are **permutation-equivariant**
  - Means: permutation of input $\rightarrow$ output permuted correspondingly
  - That's good for sets, but not for sequential data
- Approach 1: modify input
  - I.e., feed $f(\boldsymbol{x}_i, i)$ instead of $\boldsymbol{x}_i$ into the Transformer
  - Common approach: **position embeddings**

    $$f(\boldsymbol{x}_i, i) = \boldsymbol{x}_i + \mathrm{posemb}(i),$$

    i.e, position-dependent embedding "applied" to input (here: added)
  - Fixed (e.g., original Transformer, RoPE) or learned (e.g., BERT, GPT)
  - Note: when padding inputs to common length, pad on the *right* (!)
- Approach 2: modify attention mechanism
  - Based on relative distance $i_{\text{query}} - i_{\text{key}}$ between input elements $\rightarrow$ **relative positions**
  - E.g., add distance-dependent bias to attention scores (e.g., T5)
  - E.g., add distance-dependent embedding to keys/values
  - Again, fixed or learned
- Note: similar ideas used for images, graphs, tables, ...

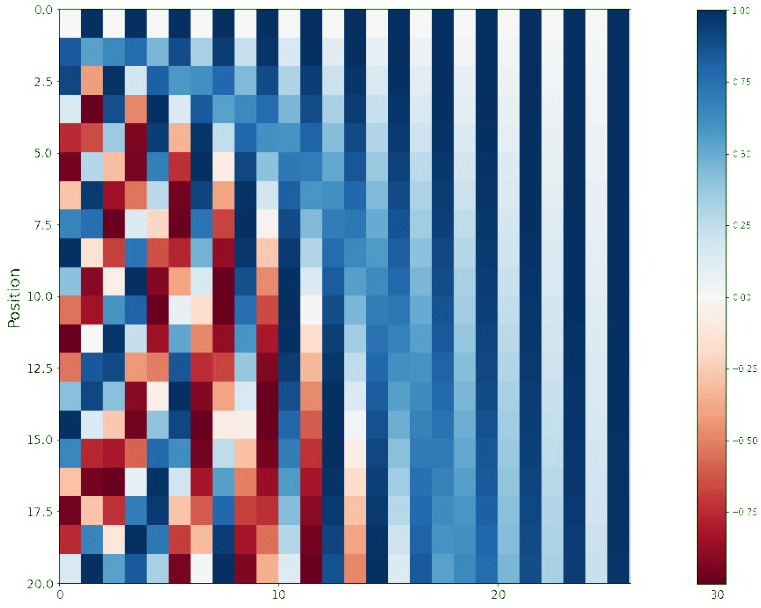
Position encodings of original Transformer

For d-dimensional embeddings, the original Transformer used

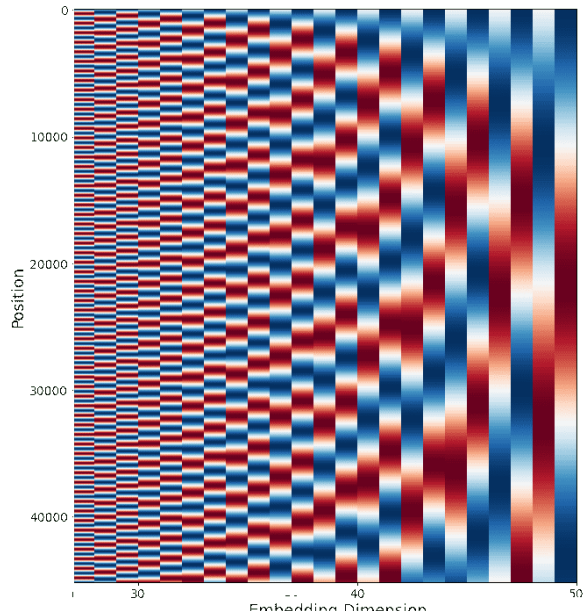
$$\text{posemb}(i)[2k] = \sin(i/10000^{2k/d})$$
$$\text{posemb}(i)[2k+1] = \cos(i/10000^{2k/d})$$

- i = input position; k = index in position embedding
- Sinusoidal in i, frequency decreases with k
- Transformer encoder learns how to exploit this (when trained with these embeddings applied)

First positions/elements



Later positions/elements



Learned position encodings

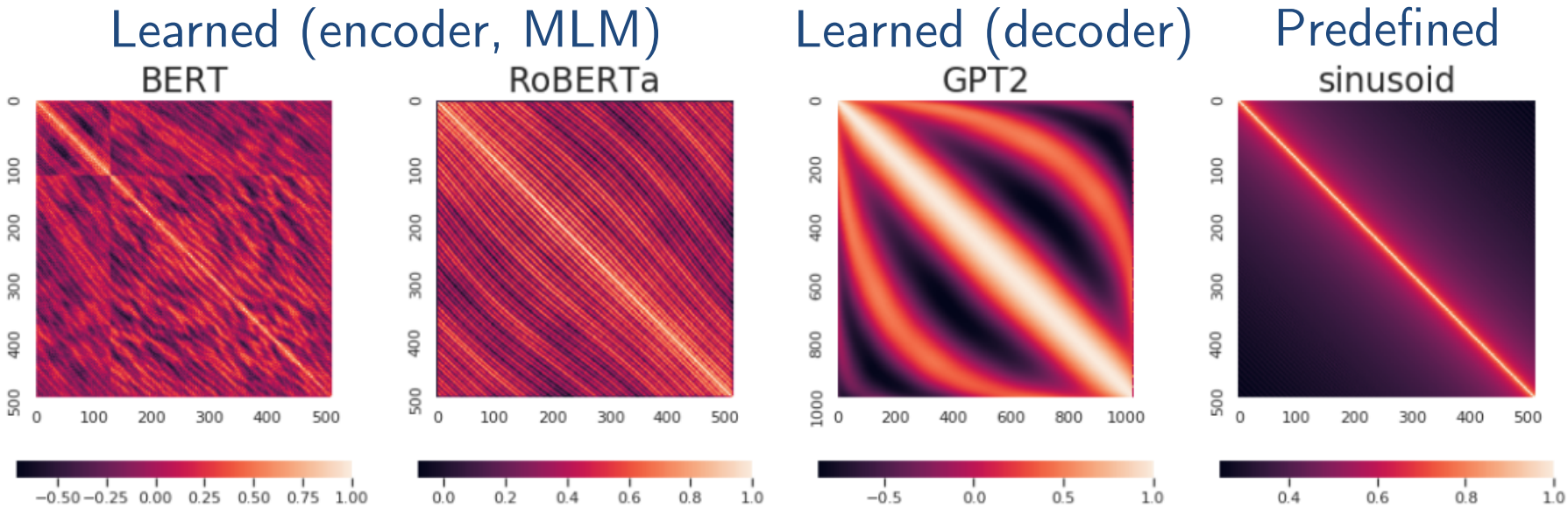


Figure 1: Visualization of position-wise cosine similarity of different position embeddings. Lighter in the figures denotes the higher similarity.

Type	PE	MAE
Learned	BERT	34.14
	RoBERTa	6.06
	GPT-2	1.03
Pre-Defined	sinusoid	0.0

Table 1: Mean absolute error of the reversed mapping function learned by linear regression.

Type	PE	Error Rate
Learned	BERT	19.72%
	RoBERTa	7.23%
	GPT-2	1.56%
Pre-Defined	sinusoid	5.08%

Table 2: Error rate of the relative position regression.

Wang and Chen, 2020

Scalar relative position encodings

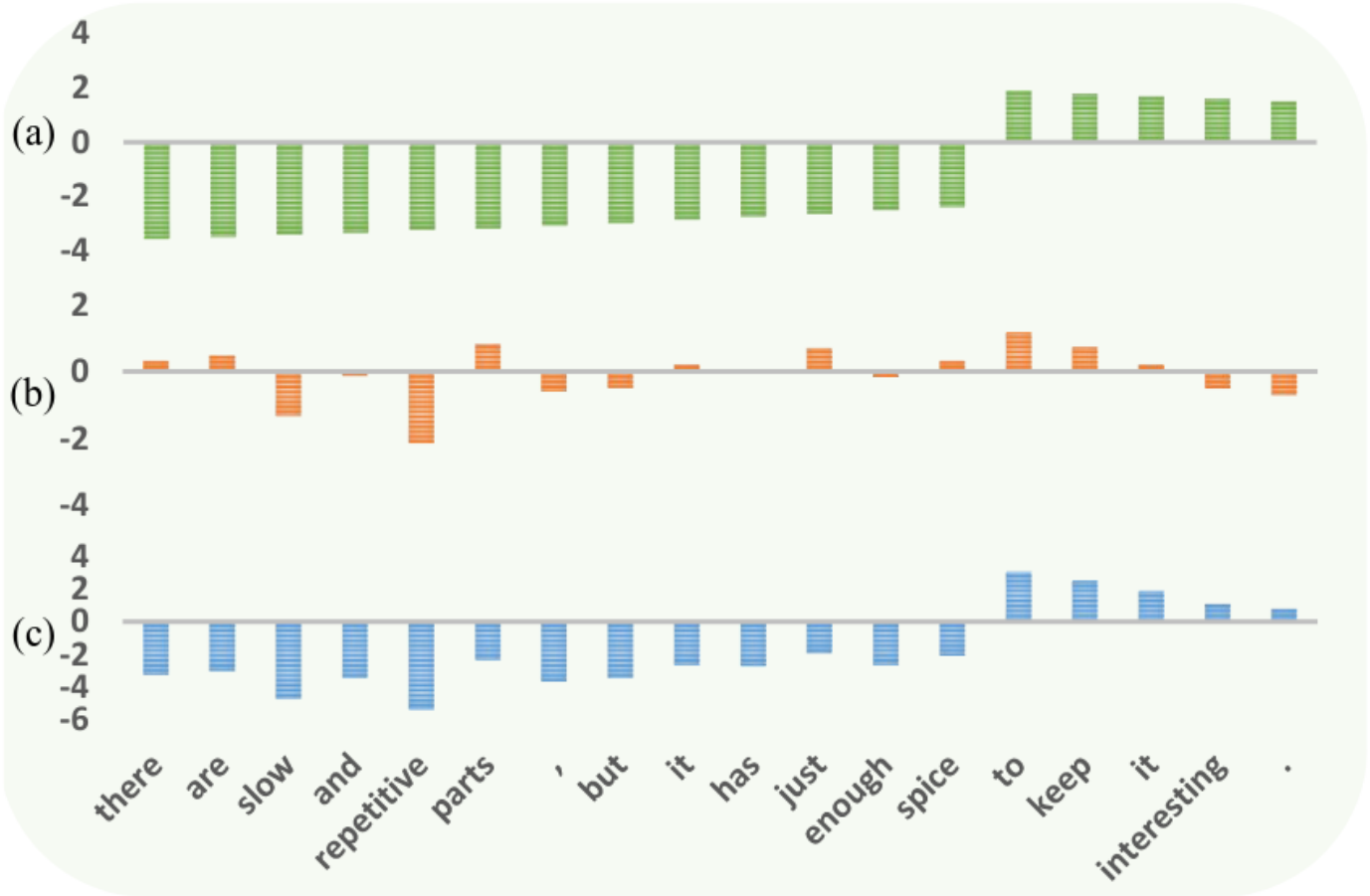


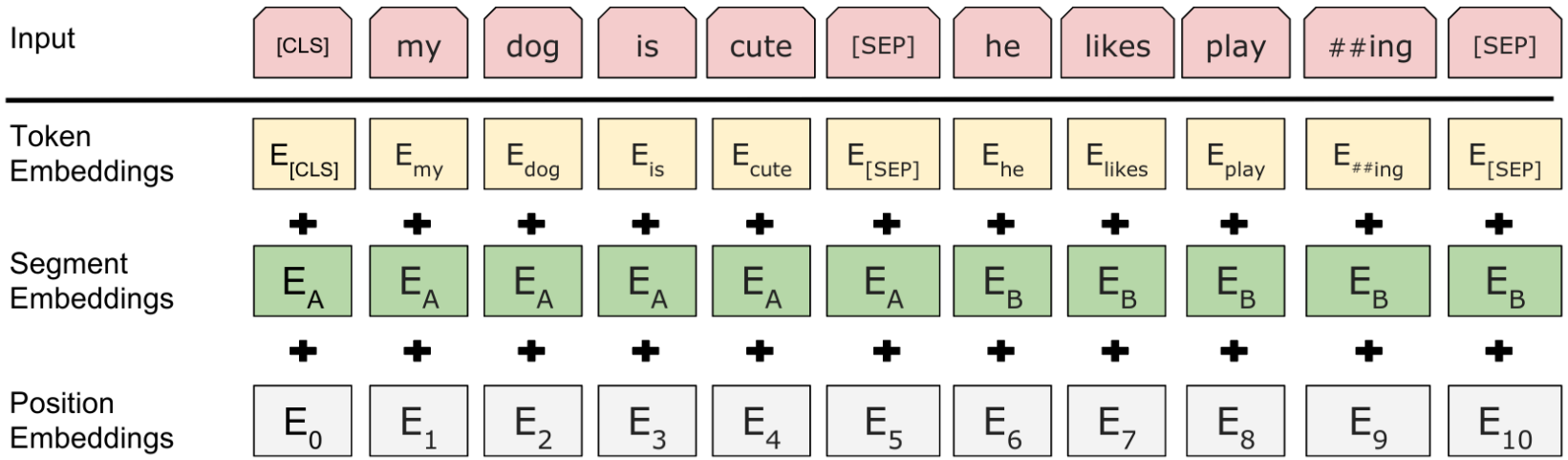
Figure 2: Scores of each token attending to the current central word *spice* of one attention head in SRPE (Scalar Relative Positional Encoding). (a): Relative position scores (b): Context scores (c): Final attentive scores.

Wu et al., 2021

Example: BERT (encoder-only model, 2018)

- BERT (Bidirectional Encoder Representations from Transformers) is a sequence representation model
  - Many variants; e.g., Sentence-BERT obtain sentence embeddings
- Given an input sequence, BERT produces representations of each token using the Transformer encoder (no decoder)
  - As discussed, tokens correspond to words or parts of words (e.g., byte-pair encoding or WordPiece)
  - Special tokens are introduced to enhance the input or output representation
- Key innovations of BERT (more later)
  - A simple bidirectional encoder-only model that achieved SOTA results across a number of NLP tasks at the time
  - Pre-trained on large textual corpora using a suitable (i) input encodings and (ii) self-supervised pretraining tasks
  - Fine-tuned in supervised fashion for a particular task at hand
  - Open-sourced (code and model)

BERT (input representations)

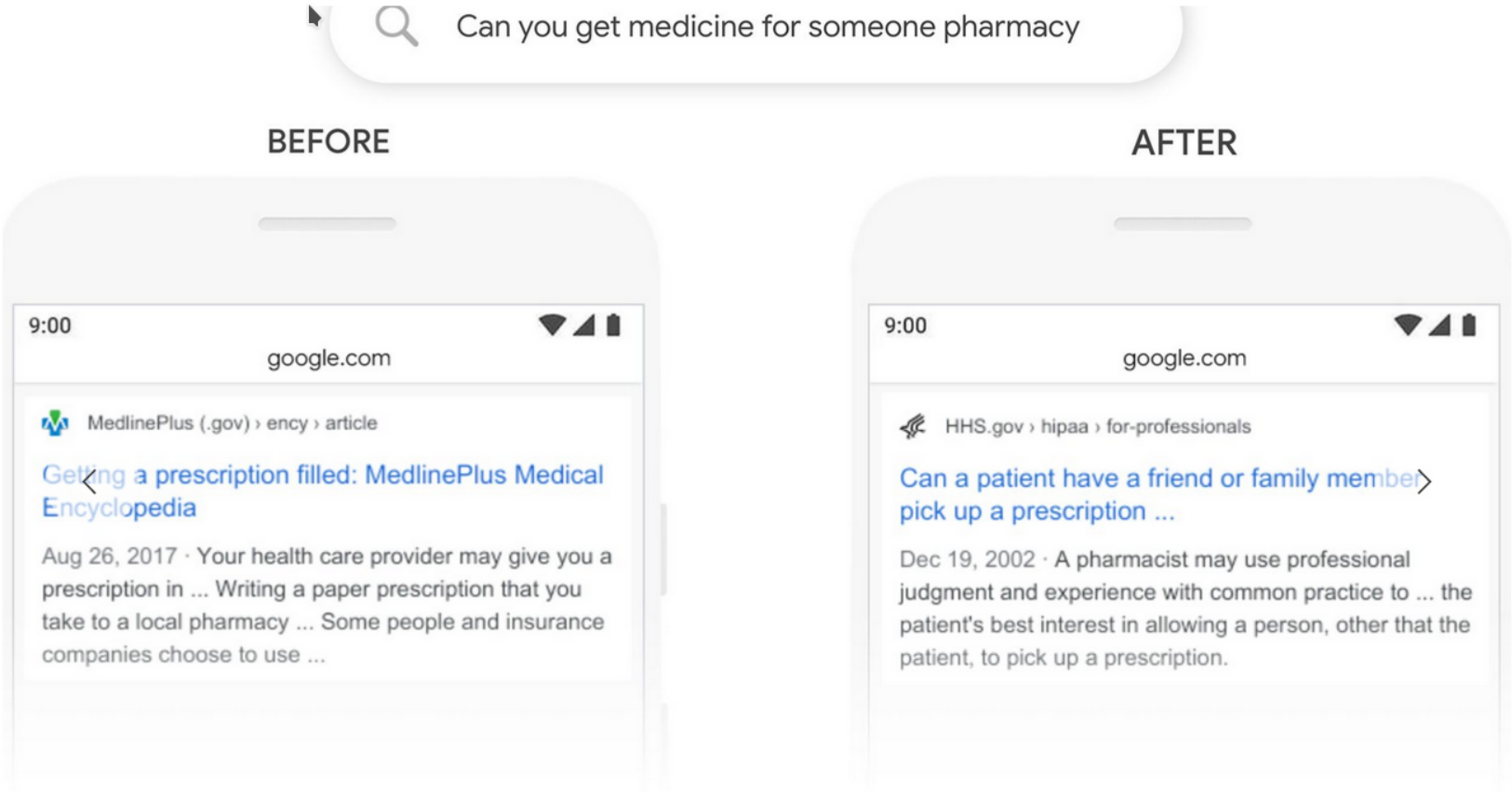


- Token embeddings = embedding layer
- Segment embeddings provide additional information about input
  - E.g., "first sentence" and "second sentence"
  - E.g., "question" and "answer"
  - Other side information can be included in this way as well
- Position embeddings to encode ordering
- Special tokens
  - [CLS] token serves as representation of entire input
  - [SEP] token used to separate sentences

Devlin et al., 2018

Example: BERT for search

As of 2019 (and apparently still in 2023), BERT models are used to serve Google search queries.



With the BERT model, we can better understand that "for someone" is an important part of this query, whereas previously we missed the meaning, with general results about filling prescriptions.

Google Blog, 2019

Example: Vision Transformer (encoder)

- Same idea applicable to grid data (below), graph data (later), ...
- Example: Vision Transformer (ViT, 2020)
  - Input element = patch embedding + position (learned embedding)
  - And a special [class] token (as in BERT)
  - No spatial inductive bias → distant relationships can be modeled

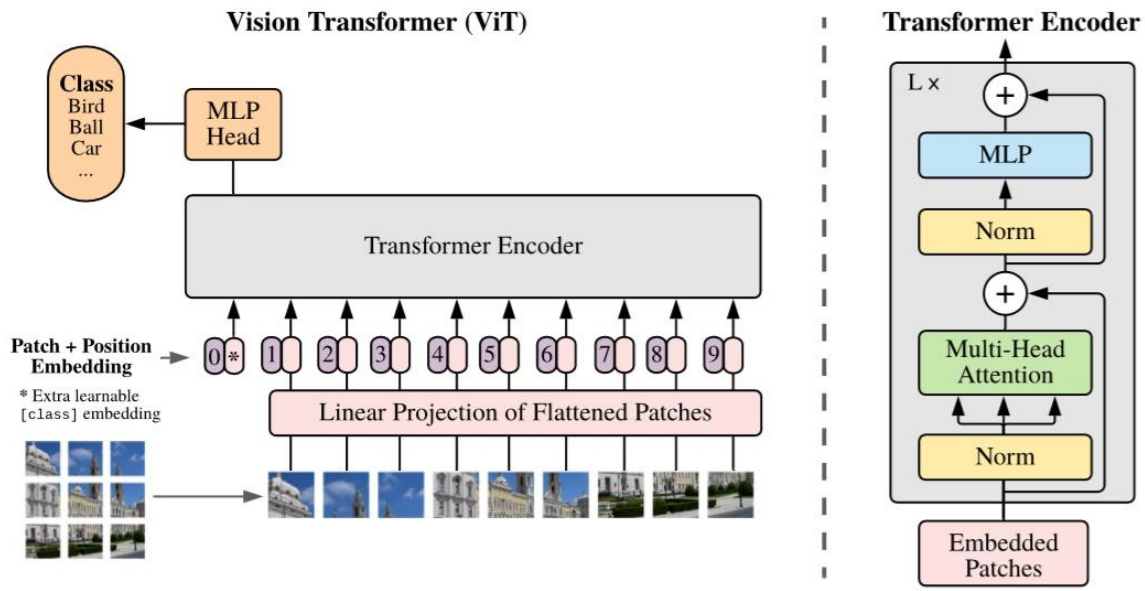
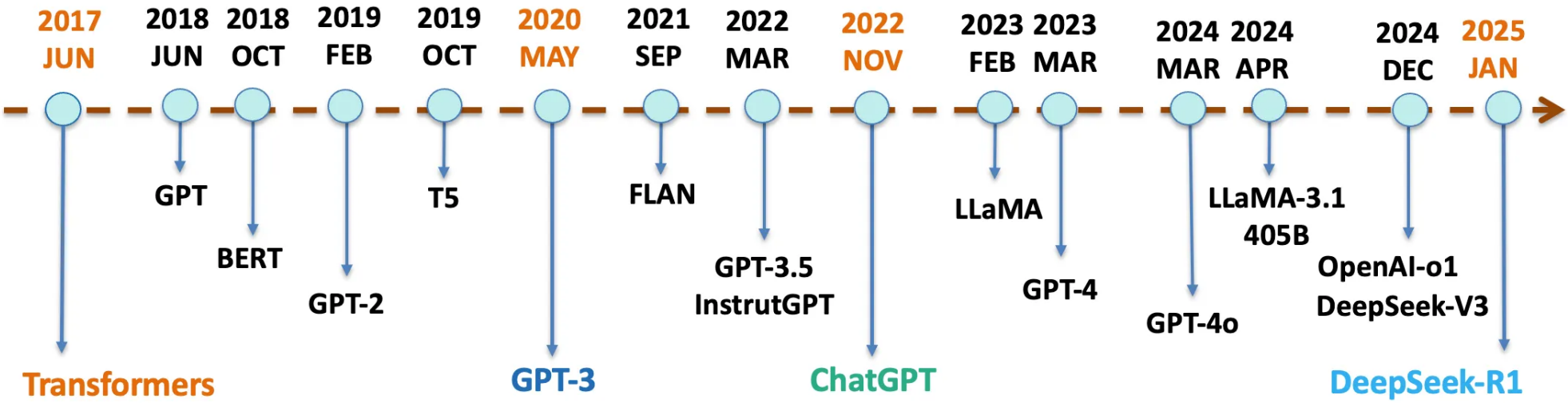


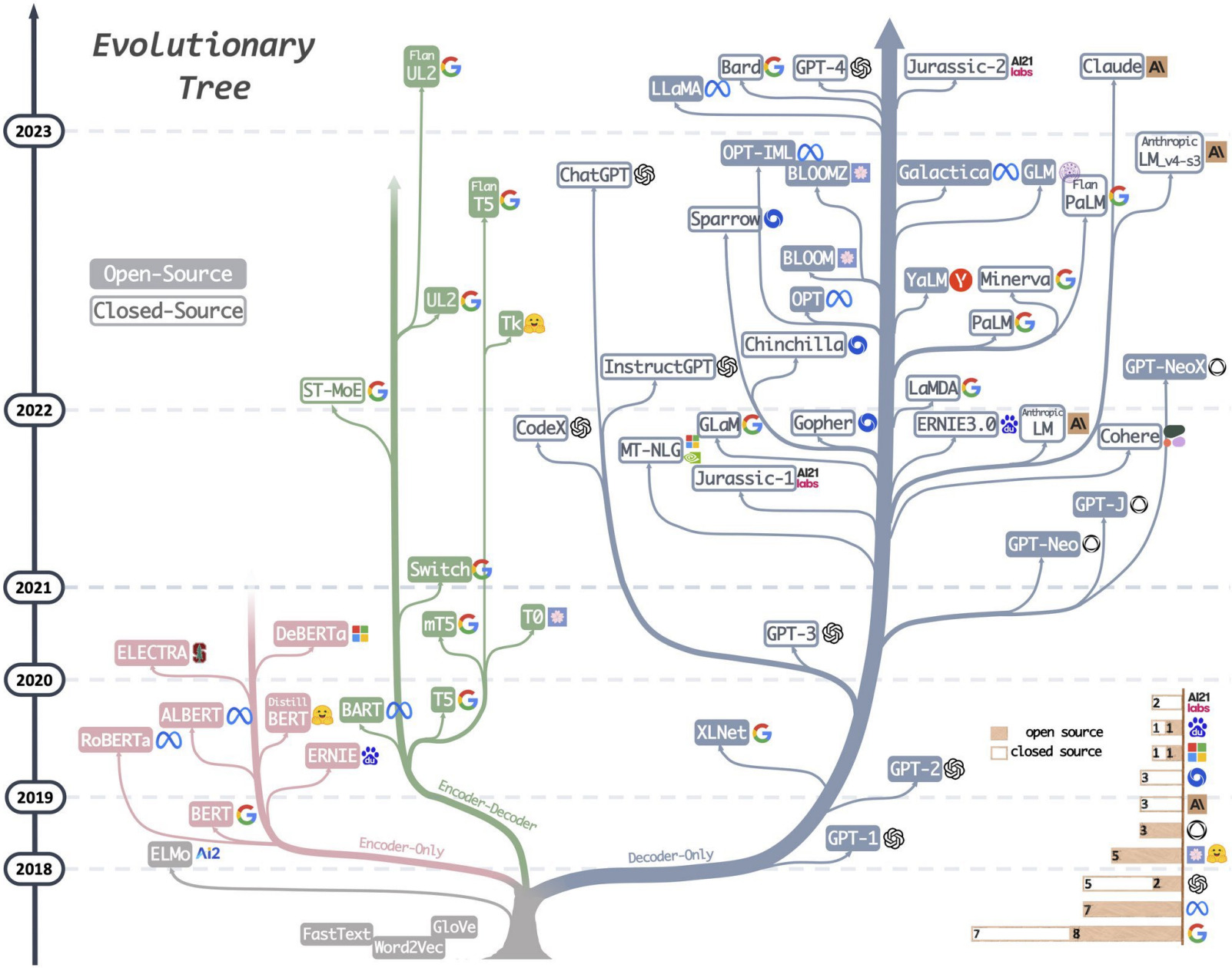
Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

Outline

Some Transformer models for language (1)



Some Transformer models for language (2)



Recap: Autoregressive Generative Models (from 07)

- Generative model for sequence data $y_1, y_2, \ldots, y_\tau \in \mathcal{Y}$
- Output distribution decomposed into next-element distributions

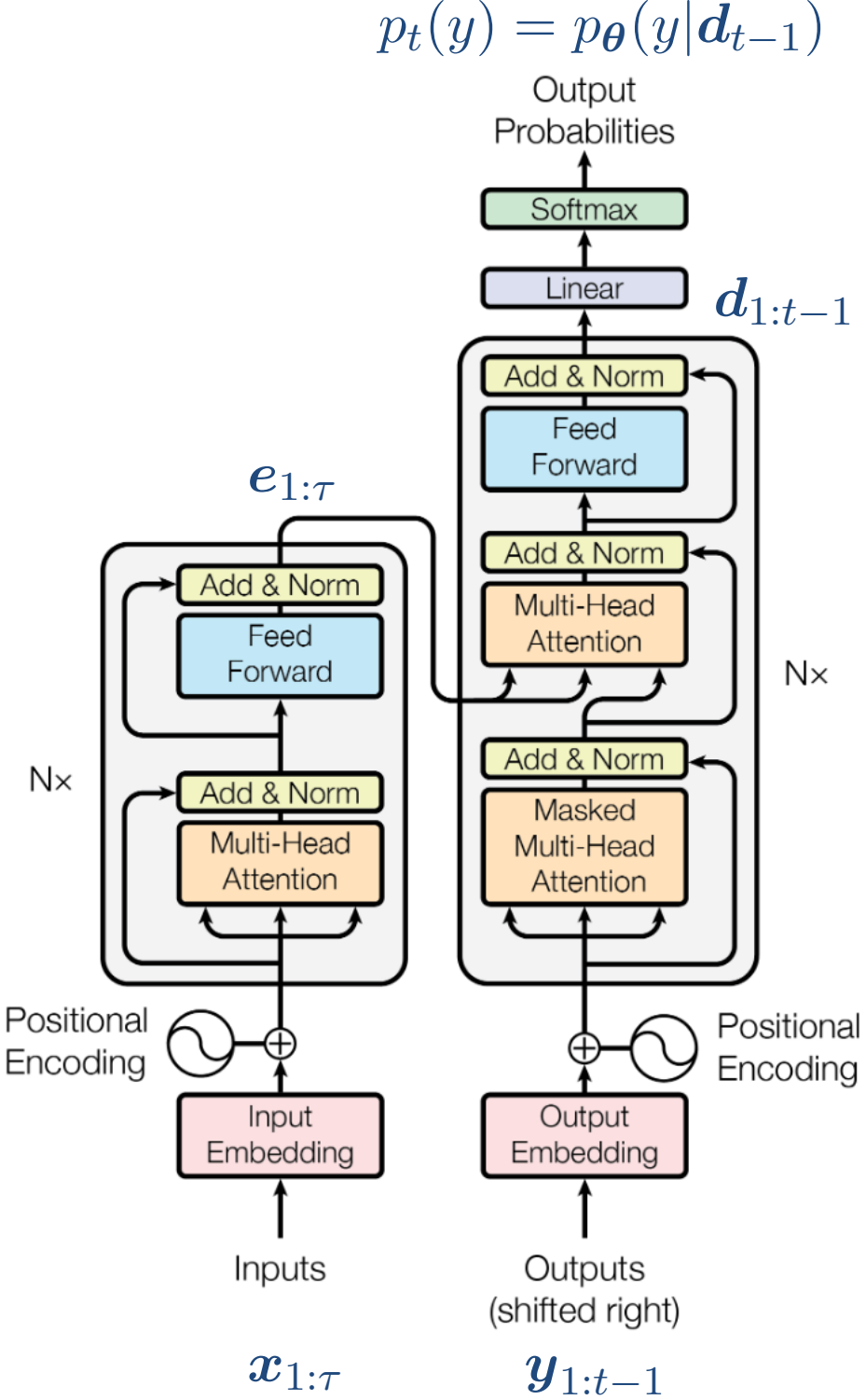
$$p(y_{1:\tau}) = \prod_{t=1}^{\tau} \underbrace{p(\overbrace{y_t}^{\text{next element}} \mid \overbrace{y_{1:t-1}}^{\text{past elements}})}_{\text{next-element distribution } p_t(y_t)}$$

- To generate, forward sample from $p_1(y)$, $p_2(y)$, ...
- We discussed: RNN decoder with hidden state $\boldsymbol{z}_t$
  - Use $p_t(y) \stackrel{\text{def}}{=} p_{\boldsymbol{\theta}}(y|\boldsymbol{z}_t)$
  - Example (discrete output): $p_{\boldsymbol{\theta}}(y|\boldsymbol{z}_t)$ is categorical distribution (with parameters $\boldsymbol{p}_t = g_{\boldsymbol{\theta}}(\boldsymbol{z}_t)$)
  - Example (continuous output): $p_{\boldsymbol{\theta}}(y|\boldsymbol{z}_t)$ is normal distribution (with parameters $(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t) = g_{\boldsymbol{\theta}}(\boldsymbol{z}_t)$)
- **Transformer decoders work similarly**; in a nutshell
  - p_t from: hidden state $\boldsymbol{z}_t \rightarrow$ last-output representation $\boldsymbol{d}_{t-1}$
  - Prior outputs: Recurrence/attention $\rightarrow$ masked self-attention
  - Inputs: initial state/cross-attention $\rightarrow$ cross-attention

Encoder-decoder Transformer (1)

Encoder-decoder Transformer for seq2seq; figure summarizes generation of the t-th output element y_t (for discrete elements).

- Left: encoder
- Right: decoder



Encoder-decoder Transformer (2)

- Transformer encoders map inputs $x_1, \ldots, x_\tau$ to representations $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_\tau$ (last layer's output)
- **Transformer decoders** define next-element distr. $p(y_t | \boldsymbol{x}, \boldsymbol{y}_{1:t-1})$
  - To do so, decoder maps outputs $y_1, \ldots, y_{t-1}$ produced so far to representations $\boldsymbol{d}_1, \ldots, \boldsymbol{d}_{t-1}$ (last layer's output)
  - $p_t(y)$ computed from (only) $\boldsymbol{d}_{t-1}$ (e.g., via softmax layer)
- Uses **Transformer decoder layers**: similar to encoder layers but perform attention twice:
  1. **Masked self-attention** to attend over all *prior* outputs
     $\rightarrow$ Replaces hidden-to-hidden and output-to-hidden conn. of RNNs
  2. **Cross attention** to attend over inputs (i.e., encoder outputs)
     $\rightarrow$ Replaces attention over input in RNN decoders with attention
- Decoding y_t involves
  1. Compute $\boldsymbol{d}_{t-1}$ via transformer decoder layers
  2. Determine $p_t(y) = p_{\boldsymbol{\theta}}(y | \boldsymbol{d}_{t-1})$
  3. Sample $y_t \sim p_t$

Example: T5 (2020) / T5X (2022)

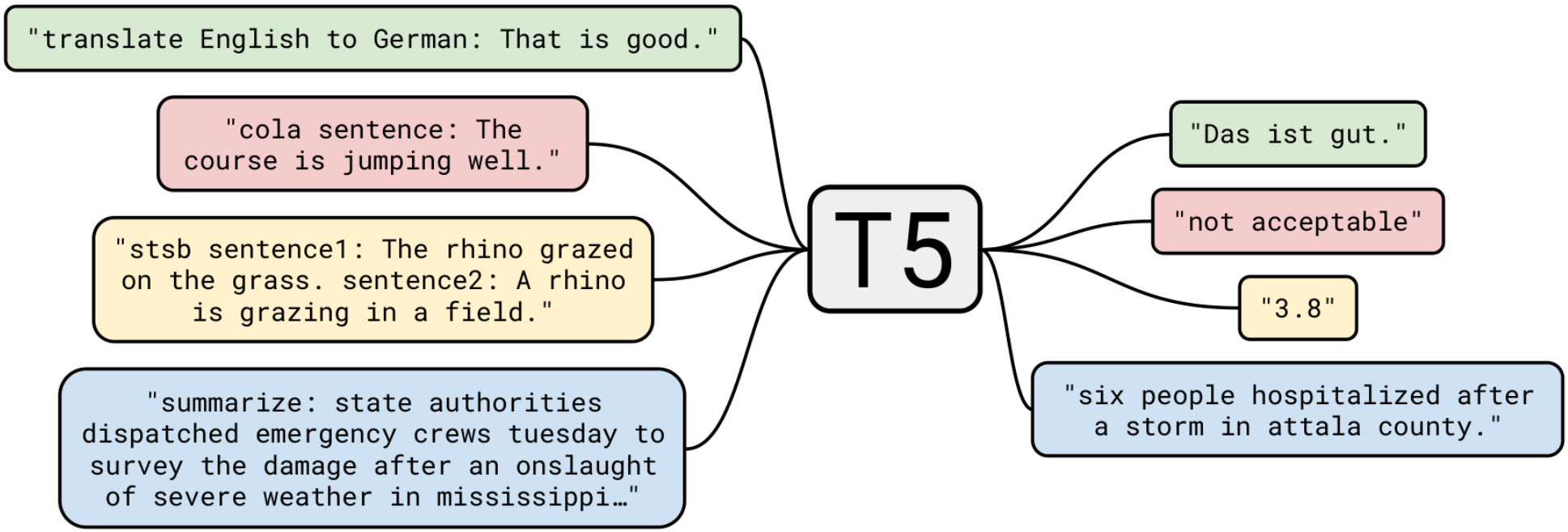


Figure 1: A diagram of our text-to-text framework. Every task we consider—including translation, question answering, and classification—is cast as feeding our model text as input and training it to generate some target text. This allows us to use the same model, loss function, hyperparameters, etc. across our diverse set of tasks. It also provides a standard testbed for the methods included in our empirical survey. "T5" refers to our model, which we dub the "**T**ext-**t**o-**T**ext **T**ransfer **T**ransformer".

Masked self-attention

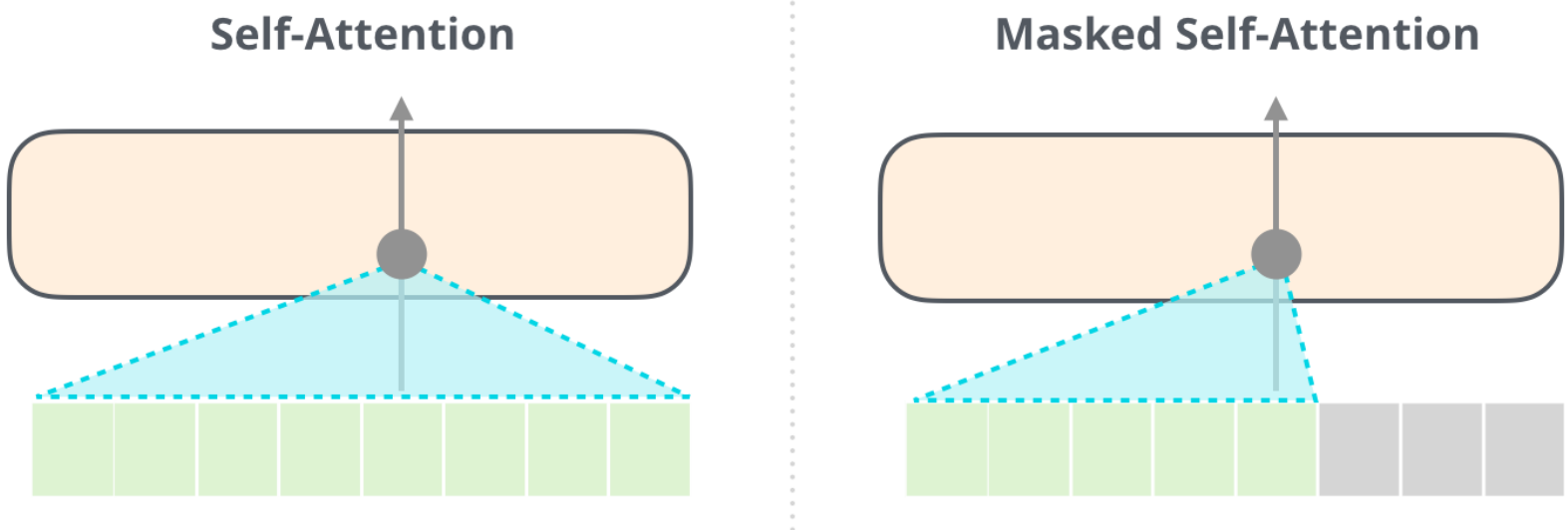
- **Masked self-attention** = attend over only a subset of inputs
  - **Mask** describes set of elements to attend over
  - Can be (and typically is) different for each position
    $\rightarrow$ controls information flow across elements
  - Specified apriori
- Example: for position 3 out of 5, attend only over inputs 1, 2, 3

$\downarrow$ Element 3's / $\rightarrow$ for input	1	2	3	4	5
Attention scores	3	5	-1	4	3
Mask	1	1	1	0	0
Effective attention scores	3	5	-1	$-\infty$	$-\infty$
Attention weights	0.119	0.879	0.002	0	0

- Observe: updated representation of element 3 depends
  - On query of element 3
  - On keys/values of elements 1–3
  - But not on query/keys/values of elements 4–5
    $\rightarrow$ **Excluded elements do not affect output**

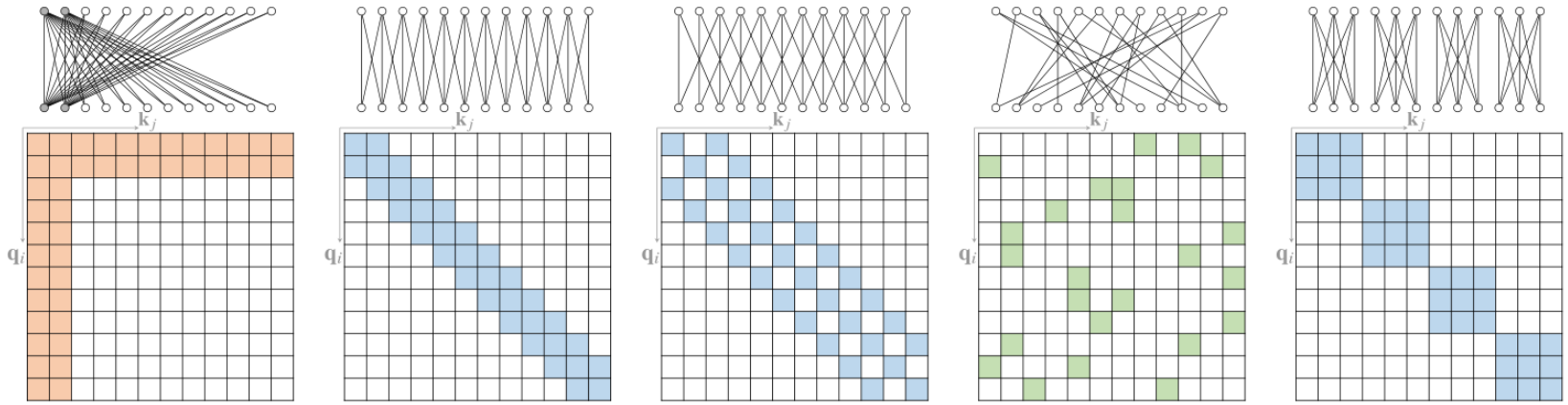
Masked self-attention in Transformer decoders

- Masked attention in Transformer decoders
  - At each output position k, only attend over positions 1–k
  - Ensures **uni-directional information flow**: element representations $\boldsymbol{d}_k^l$ do not depend on $y_{k+1}, y_{k+2}, \ldots$
  - As a consequence, element representations do not change when new elements are appended by the decoder $\rightarrow$ no need to recompute
  - E.g., to produce $p_t(y)$, "only" need to compute $\boldsymbol{d}_{t-1}^1, \ldots, \boldsymbol{d}_{t-1}^L$



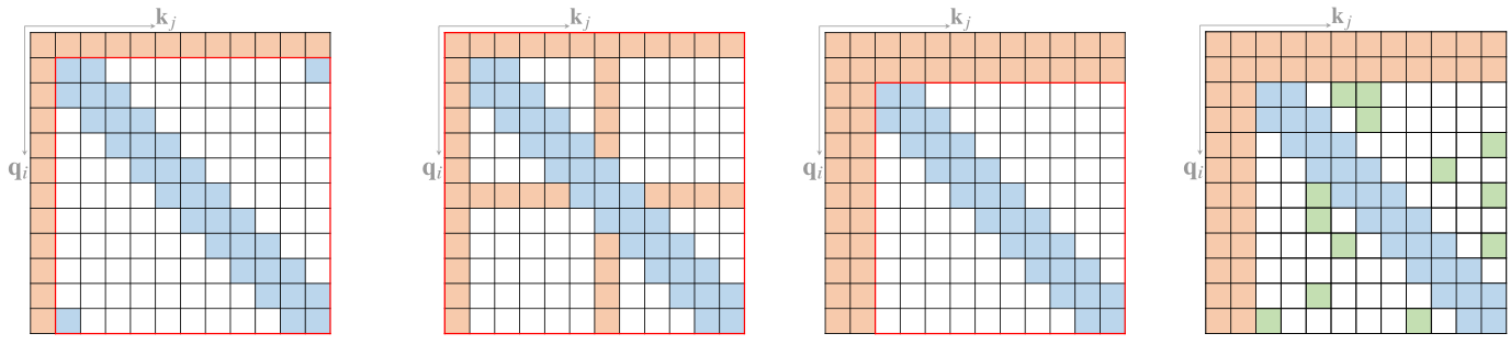
Masked self-attention in Transformer encoders

Masked attention also useful to reduce $O(\tau^2)$ cost of self-attention.
Trade-off: representational power vs. computational cost.



(a) global (b) band (c) dilated (d) random (e) block local

Fig. 4. Some representative atomic sparse attention patterns. The colored squares means corresponding attention scores are calculated and a blank square means the attention score is discarded.



(a) Star-Transformer (b) Longformer (c) ETC (d) BigBird

Fig. 5. Some representative compound sparse attention patterns. The red boxes indicate sequence boundaries.

Lin et al. (2022)

Cross attention

- Recall: **Cross attention** generally means to attend over elements from a different model or embedding space
- In Transformer decoders: **encoder-decoder cross-attention**
  - Keys and values come from input elements x_k (i.e., computed from *final* encoder representations $\boldsymbol{e}_k$)
  - Query comes from decoder (from current representation of resp. output element)
- Again, very similar to decoder RNN of sl. 14, which
  - Used RNN encoder states directly instead of computing keys/values
  - Used RNN decoder state as query instead of element representation
- To obtain $\boldsymbol{d}_{t-1}$, need keys/values for inputs and prior outputs
  - Can be recomputed every time, but that's expensive
  - As these keys/values don't change, typically managed in a so-called **key-value cache** (= the "dots" on next slide)
  - When processing an element in the decoder, its keys/values are added to the cache

All together now (start of decoding)

2 decoder layers shown

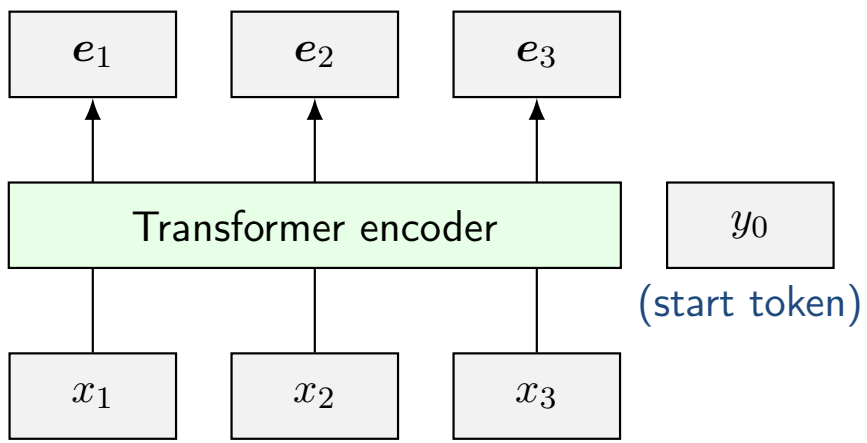
MSA = masked self-attention block
xSA = cross attention block
MLP = MLP block

Each block is residual and
includes normalization
(both not shown)

Red arrow = query
Blue dot = k/v pair
Blue arrow = use of k/v pair

Each query is computed
using $\boldsymbol{W}_Q$ of its
consuming layer

Each k/v pair is computed
using $\boldsymbol{W}_K$ and $\boldsymbol{W}_V$
of its consuming layer



All together now (generating y_1)

2 decoder layers shown

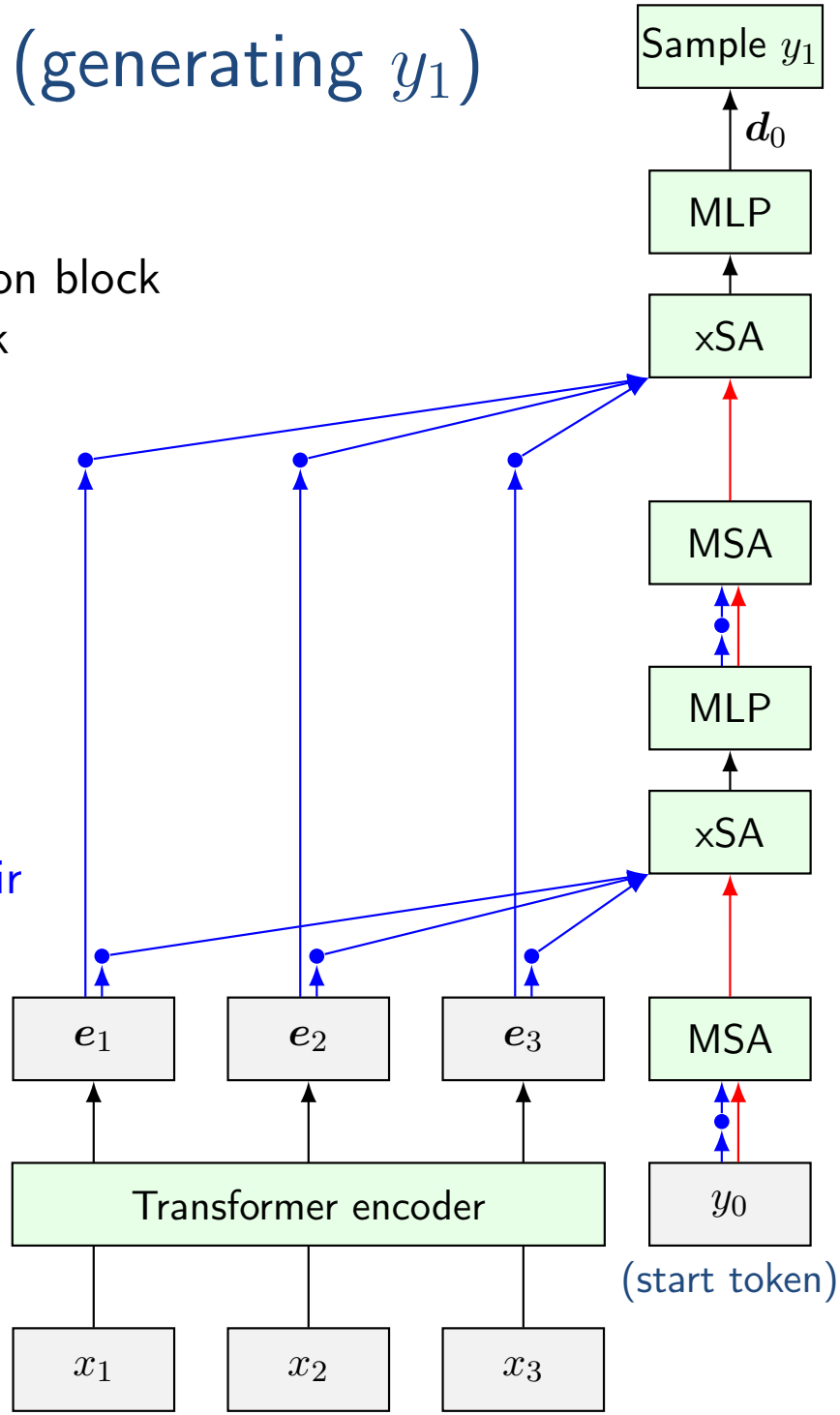
MSA = masked self-attention block
xSA = cross attention block
MLP = MLP block

Each block is residual and includes normalization (both not shown)

Red arrow = query
Blue dot = k/v pair
Blue arrow = use of k/v pair

Each query is computed using $\boldsymbol{W}_Q$ of its consuming layer

Each k/v pair is computed using $\boldsymbol{W}_K$ and $\boldsymbol{W}_V$ of its consuming layer



All together now (generating y_2)

2 decoder layers shown

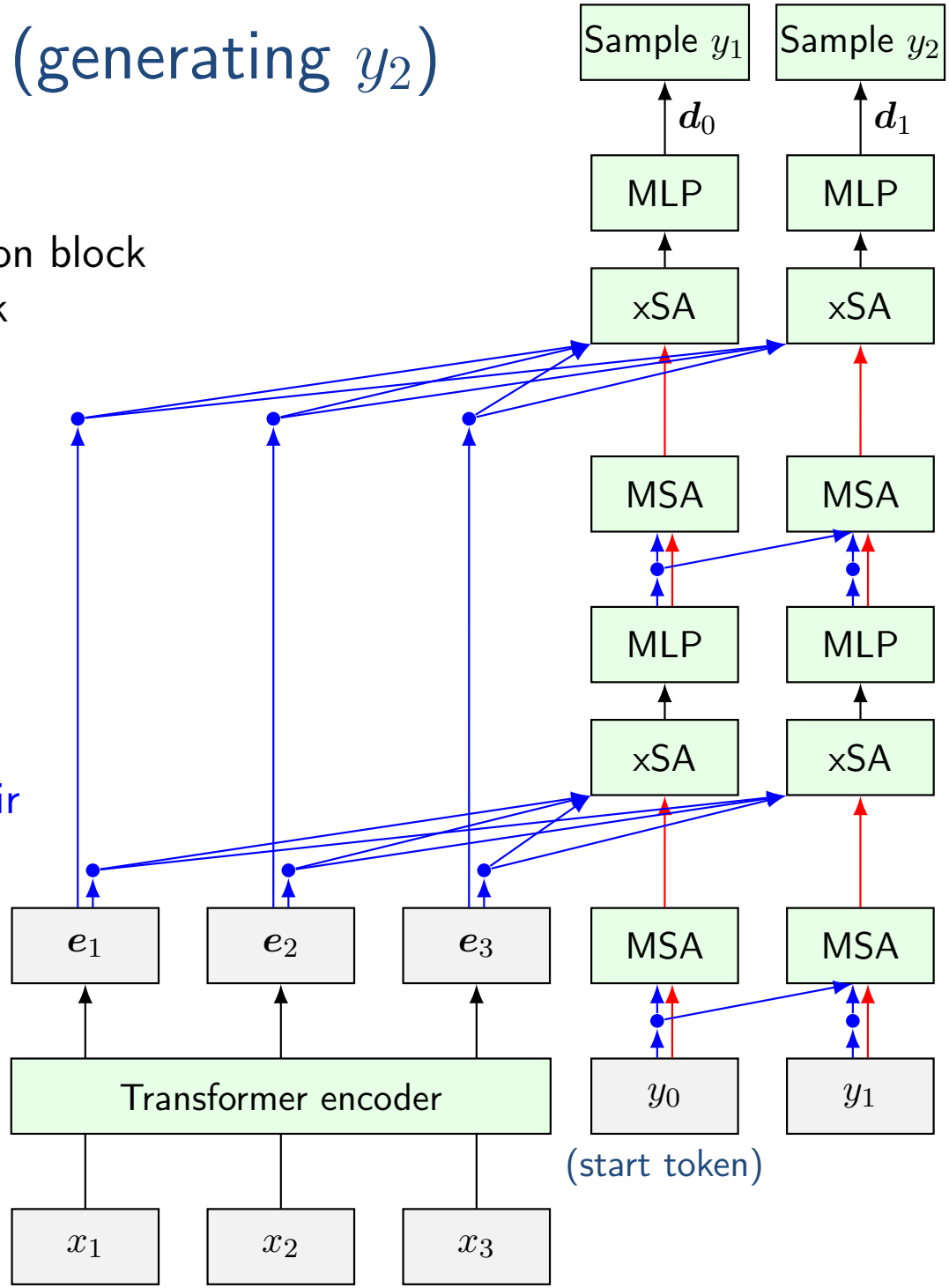
MSA = masked self-attention block
xSA = cross attention block
MLP = MLP block

Each block is residual and includes normalization (both not shown)

Red arrow = query
Blue dot = k/v pair
Blue arrow = use of k/v pair

Each query is computed using $\boldsymbol{W}_Q$ of its consuming layer

Each k/v pair is computed using $\boldsymbol{W}_K$ and $\boldsymbol{W}_V$ of its consuming layer



All together now (generating y_3)

2 decoder layers shown

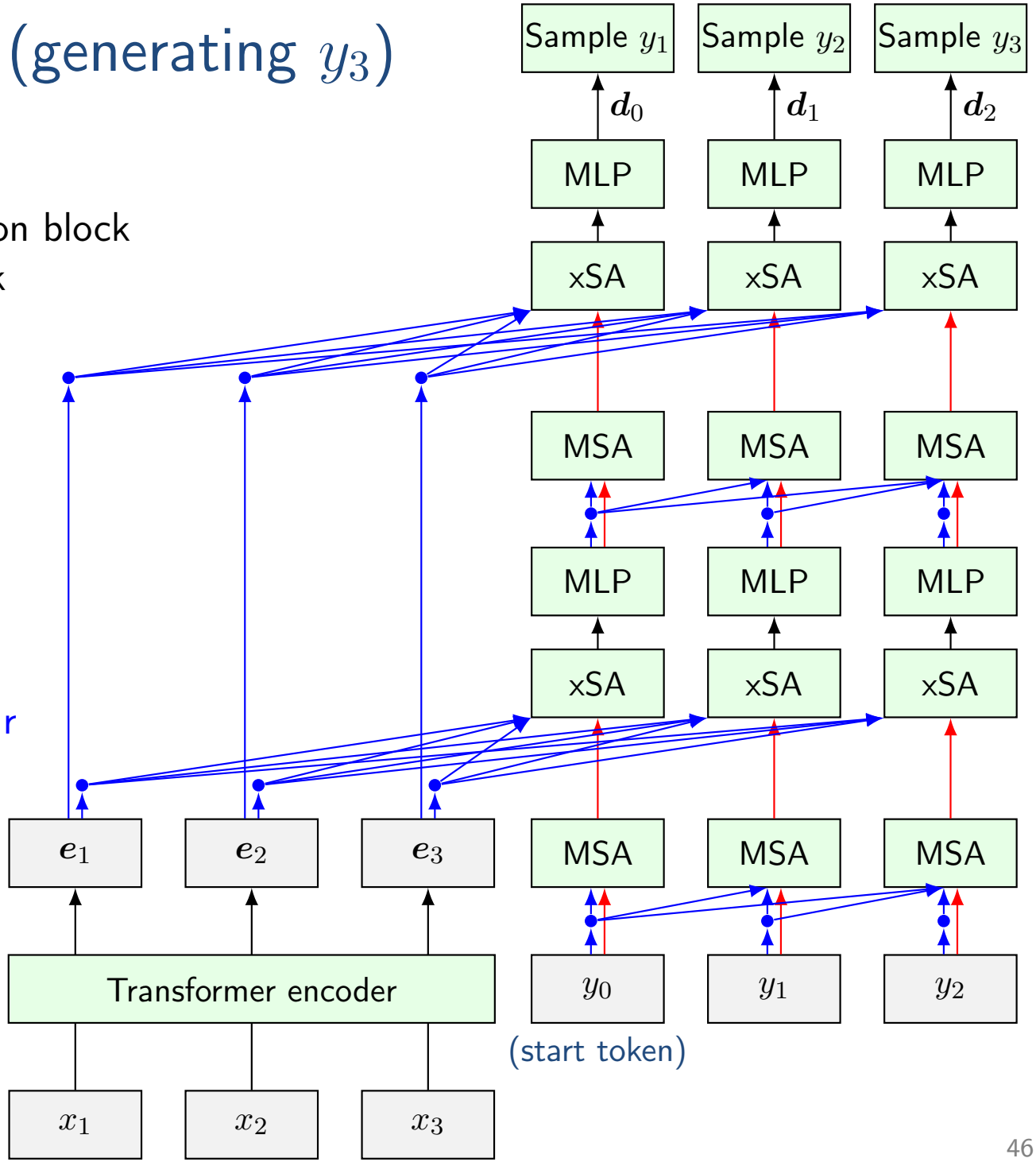
MSA = masked self-attention block
xSA = cross attention block
MLP = MLP block

Each block is residual and includes normalization (both not shown)

Red arrow = query
Blue dot = k/v pair
Blue arrow = use of k/v pair

Each query is computed using $\boldsymbol{W}_Q$ of its consuming layer

Each k/v pair is computed using $\boldsymbol{W}_K$ and $\boldsymbol{W}_V$ of its consuming layer



Decoder-only Transfomers

- **Decoder-only Transformers** do not use an encoder, but solely decoder-only blocks trained autoregressively (no cross attention)
  - As before (cf. 07), can be causal (MSA only) or non-causal (full self-attention for input parts, MSA for output parts)
- Example: GPT-1/2/3/4, ChatGPT (all causal)
- Used as a (conditional) deep generative model

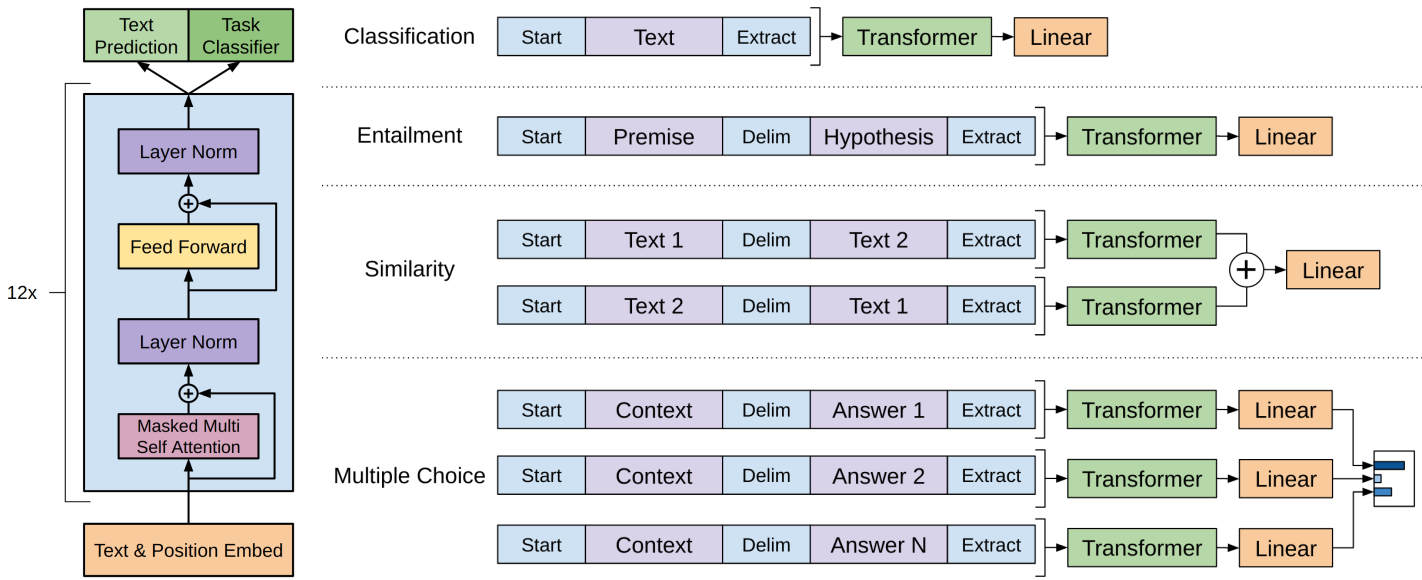


Figure 1: **(left)** Transformer architecture and training objectives used in this work. **(right)** Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

Discussion

- Key ideas of Transformers are very general
  - Self-attention over parts (e.g., a token, an image patch, a feature vector, a graph vertex, ...)
  - Position information added to parts, instead of being "hard-coded" (may use different encodings for sets, sequences, images, graphs)
  - Many variants (some discussed later); see survey by Lin et al. (2022)
- Implementation and pre-trained models readily available
  - E.g., using the Transformers library

```
from transformers import pipeline
classifier = pipeline("sentiment-analysis")
classifier('The dog ate the cake.')
[{'label': 'NEGATIVE', 'score': 0.7528}]
```

- More on training and using powerful models such as Transformers in lecture 10

Deep Learning

09 – Graph Learning

Part 0: Overview

Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-1

Graph learning

- Graphs everywhere, e.g.
  - World Wide Web, social networks, citation graphs, protein-protein interactions, purchasing networks, similarity graphs, road networks, bitcoin transactions, knowledge graphs, ...
- Graph learning = machine learning with graphs
- Examples
  - Vertex or edge classification (e.g., topic of website, friend or foe)
  - Graph classification (e.g., properties of a molecule)
  - Link prediction (e.g., facts in knowledge graph) or regression (preference in recommender systems)
  - Interesting vertices (e.g., influential bloggers, important web pages)
  - Vertex clustering (e.g., similar products)
  - Interesting subgraphs (e.g., communities, frequent subgraphs)
  - Generating graphs (e.g., neural architecture search, scene graphs in computer vision)
  - Exploitation of background or domain knowledge represented as (knowledge) graphs

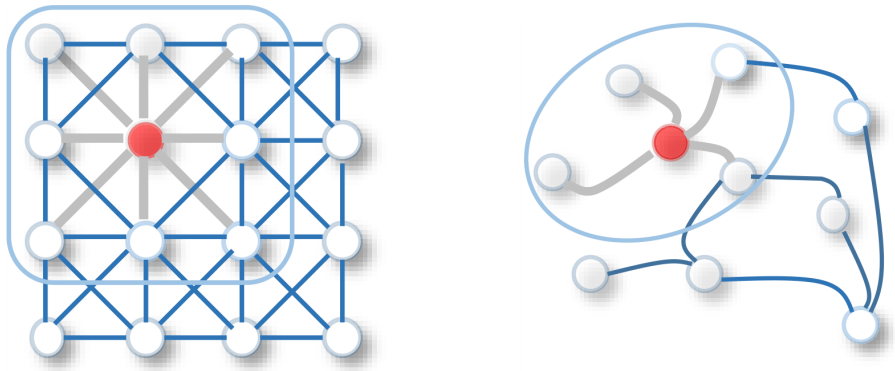
Our focus: Spectral/neural methods for graphs.

Some challenges in graph learning

- Irregular structures
  - We looked at sequences and grids so far
  - Relationships between graph vertices much more irregular
  - How to apply operations such as recurrence, convolutions, self-attention or pooling on such data?
- Heterogeneity and diversity
  - Within a graph: e.g., vertex degrees, types, features, modalities
  - Across different graphs: e.g., weighted/unweighted, directed/undirected, signed/unsigned, labeled/unlabeled, ...
  - Widely varying tasks and domains
- No independent examples
  - Each vertex (example) related to other vertices via links
- Scalability
  - Graphs may be very large
  - Learning methods may be complex/expensive
- Dynamic graphs
  - Vertices, edges, features may change over time

Deep learning for graphs

- Deep learning methods can be applied to graph learning
- Generally, use graph structure to facilitate reasoning
  - Cf. CNNs, where we used a fixed grid structure
  - Now: graph structure expresses relationships



- Part-based view also applicable: input represented in terms of
  - Global features (if any)
  - Parts (vertices) and relationship between parts (edges)
    $\rightarrow$ Relationship between parts now input-dependent
  - Part features (= initial part embeddings): vertex and/or edge features

Key operations

We generally use the same type of operations as before:

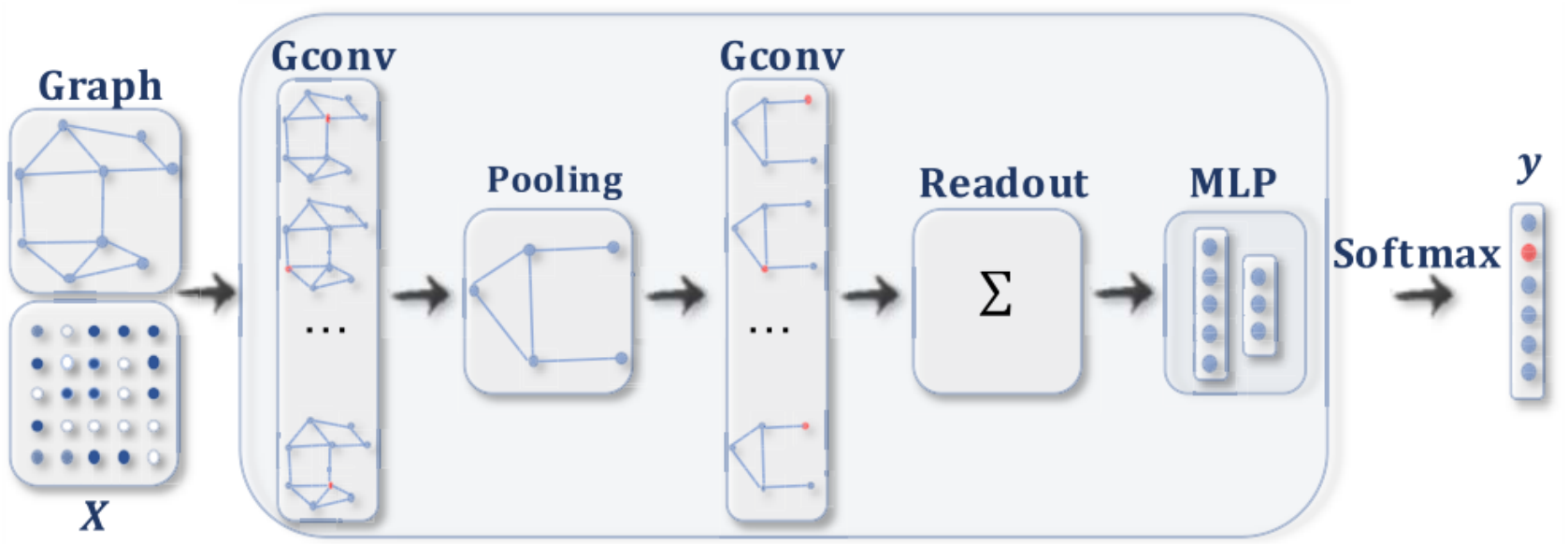
1. **Contextualization**: incorporate information from other parts into each part embedding; e.g.,
   - Spectral embeddings
   - Graph convolutions
   - Message passing and graph recurrences
   - Graph transformers

   $\rightarrow$ Increase "receptive field" of each part's representation
2. **Local compute**: update each part embedding individually
   - As before; e.g., an MLP
3. **Pooling**: aggregate multiple (or all) part embeddings
   - Readout as before (e.g., pool vertex embeddings)
   - Pooling to change resolution (e.g., coarsen a graph) more involved

   $\rightarrow$ Increase "spatial invariance"

These operations are used to obtain higher-level representations (embeddings) of each part and/or of the entire graph.

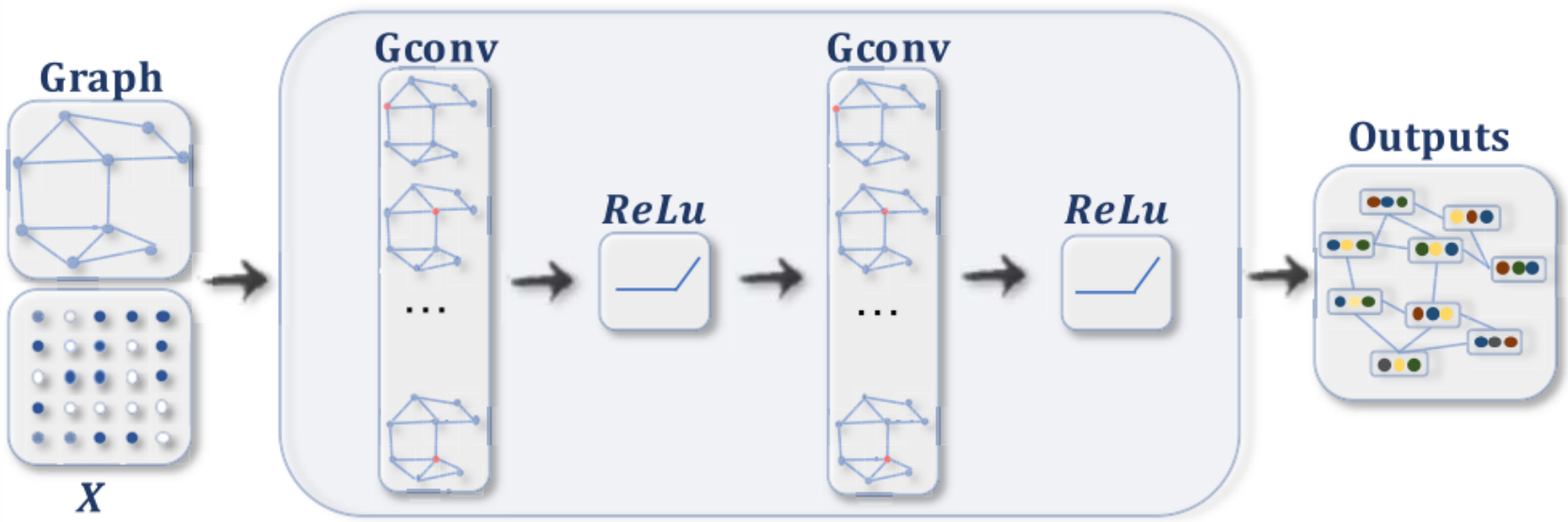
Example: Graph classification (supervised)



- Many graphs, often smaller (e.g., molecules, ego-networks, ASTs)
- Given a set of labeled graphs, learn to classify new graphs
- Supervised learning
  - Learn graph embeddings to use as input features for a (learned) prediction head (e.g., softmax regression)
  - Use input features *and connectivity* structure

Wu et al., 2021

Example: Vertex classification (transductive)



- Single graph, often large (e.g., networks, knowledge graphs)
- Some vertices labeled $\rightarrow$ Goal: label the unlabeled vertices
- Semi-supervised learning (*transductive inference*)
  - Learn vertex embeddings to use as input features for a (learned) prediction head (e.g., softmax regression)
  - Use input features *and connectivity* structure
- For example, using holdout validation
  - Train using training labels and full graph
  - Validate using holdout labels and full graph
  - Predict labels of unlabeled vertices of full graph

Outline (Graph Learning)

Lessons learned

- Connectivity structure of graphs can be represented by matrices (and vice versa)
  - Adjacency matrix
  - Degree matrix
  - Graph Laplacian
- Spectral properties of graph Laplacian relates to structural properties of the graph
- Deep learning for graphs
  - Computation layers to obtain higher-level vertex representations
    $\rightarrow$ Based on recurrence, spectral/spatial convolution, transformers
  - Pooling layers for coarsen the graph
  - Readout operations to obtain graph representation
- Many variants and design choices
  - Hyperparameter choice/optimization important

Literature

- Ulrike von Luxburg
  A Tutorial on Spectral Clustering
  Statistics and Computing, 17(4), 2007

- Wu et al.
  A Comprehensive Survey on Graph Neural Networks
  IEEE Transactions on Neural Networks and Learning Systems, 2021

- Zhang et al.
  Deep Learning on Graphs: A Survey
  IEEE Transactions on Knowledge and Data Engineering, 2020

- Xia et al.
  Graph Learning: A Survey
  IEEE Transactions on Artificial Intelligence, 2021

Deep Learning

09 – Graph Learning

Part 1: Spectral Embeddings

Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-2

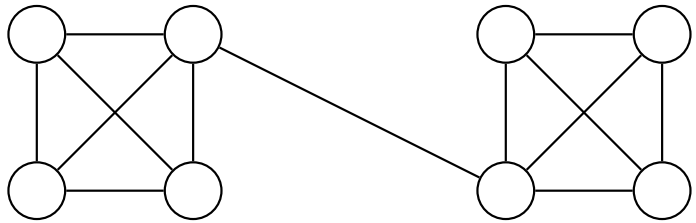
This lecture

- Preliminaries for later lectures
- Introduction to graph signals & graph Laplacian
- Spectral embeddings
  - Low-dimensional, continuous representation of each vertex in a graph (i.e., a form of vertex embedding)
  - Computed solely from and captures the graph structure
  - Represents "position" of each vertex in the graph
- Useful to
  - Perform spectral clustering (= clustering of the graph's vertices)
  - Obtain features for graph learning
  - Perform graph convolutions
  - Obtain position embeddings for graph transformers
  - ...

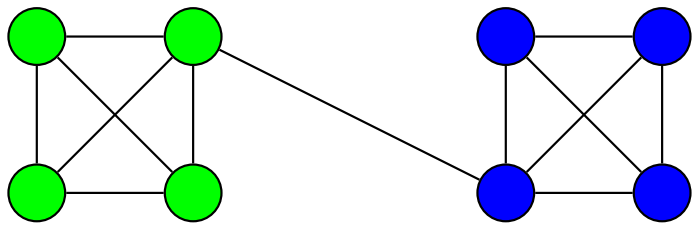
Outline

Spectral clustering

- Consider the following graph



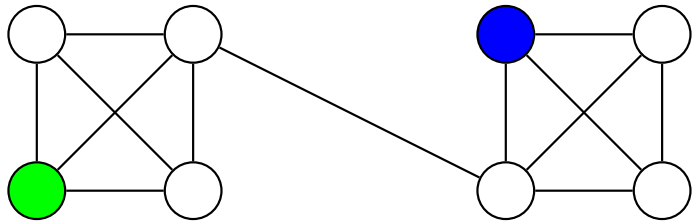
- Can we cluster the vertices into two clusters (or "communities") such that
  1. Neighboring vertices tend to be in same cluster
  2. Cluster sizes do not not vary "too much"
- Sure!



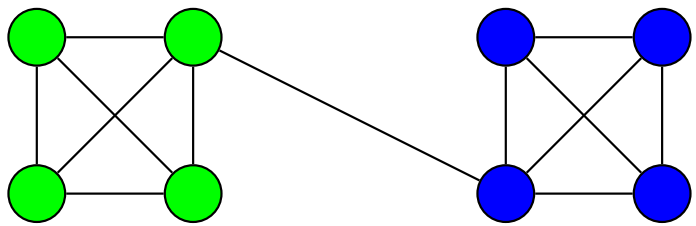
- Spectral clustering is one approach to do this
  1. Compute spectral embeddings of each vertex in the graph
  2. Cluster spectral embeddings (e.g., using k-Means)

Graph-based semi-supervised learning (GSSL)

- Consider the following partially-labeled graph



- Can we label the remaining the vertices such that neighboring vertices tend to have the same label?
- Sure!



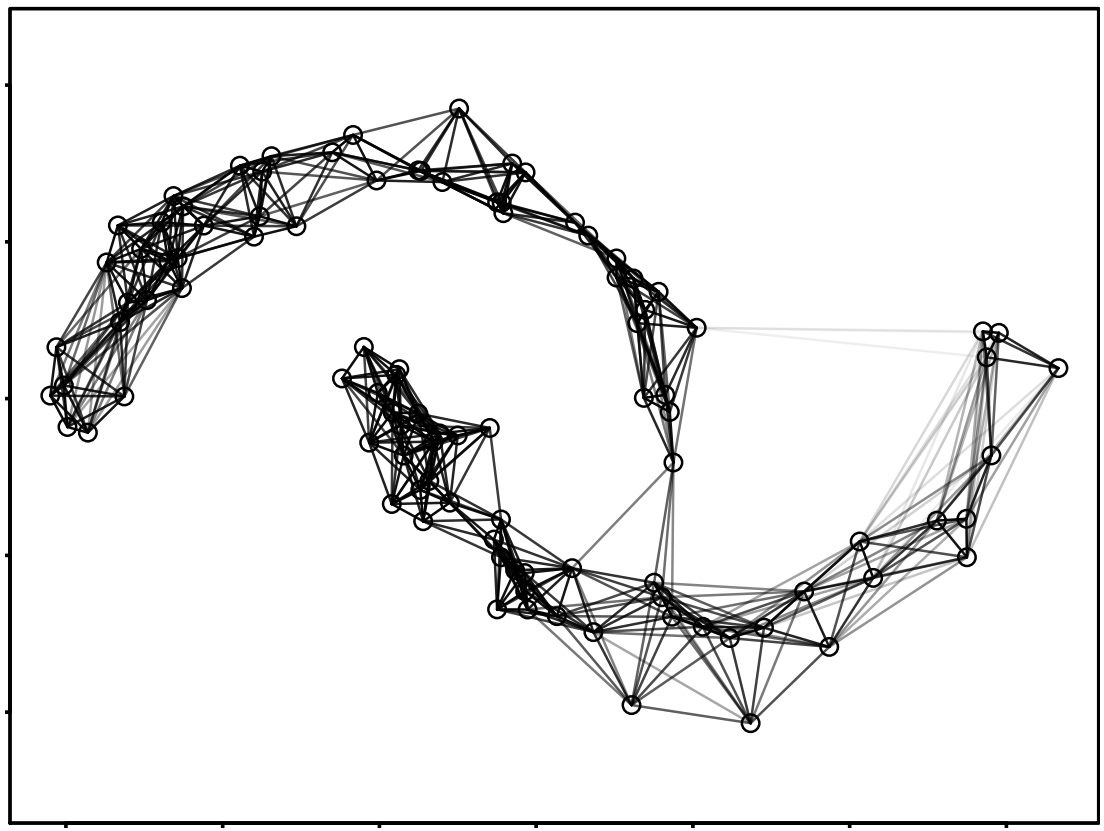
- Graph-based semi-supervised learning is one approach to do this
  - Learn a vertex classifier (e.g., based on vertex features)
  - During learning, add a penalty term to the cost function that penalizes assigning different labels to neighboring vertices
  - One approach: penalize using graph Laplacian

Similarity graphs

- When we interpret each edge as expressing similarity, then
  - Spectral clustering $\rightarrow$ cluster similar vertices
  - GSSL $\rightarrow$ label similar vertices similarly (exploiting homophily)
- Given any dataset $\mathscr{D} = \{ \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \}$ and a similarity function $s(\boldsymbol{x}, \boldsymbol{x}') \rightarrow \mathbb{R}^+$ can construct a **similarity graph**
  - Vertices correspond to data points
  - Edges connect similar data points
  - Optionally: edges weighted by similarity
- Many approaches to construct similarity graph; common:
  - **k-Nearest neighbor graph**: connect each vertex to its k nearest neighbors
  - **ϵ-neighborhood graph**: connect vertices $\boldsymbol{x}$ and $\boldsymbol{x}'$ when $s(\boldsymbol{x}, \boldsymbol{x}') \geq \epsilon$
- Why? Can then use graph-based learning on $\mathscr{D}$ via resulting graph

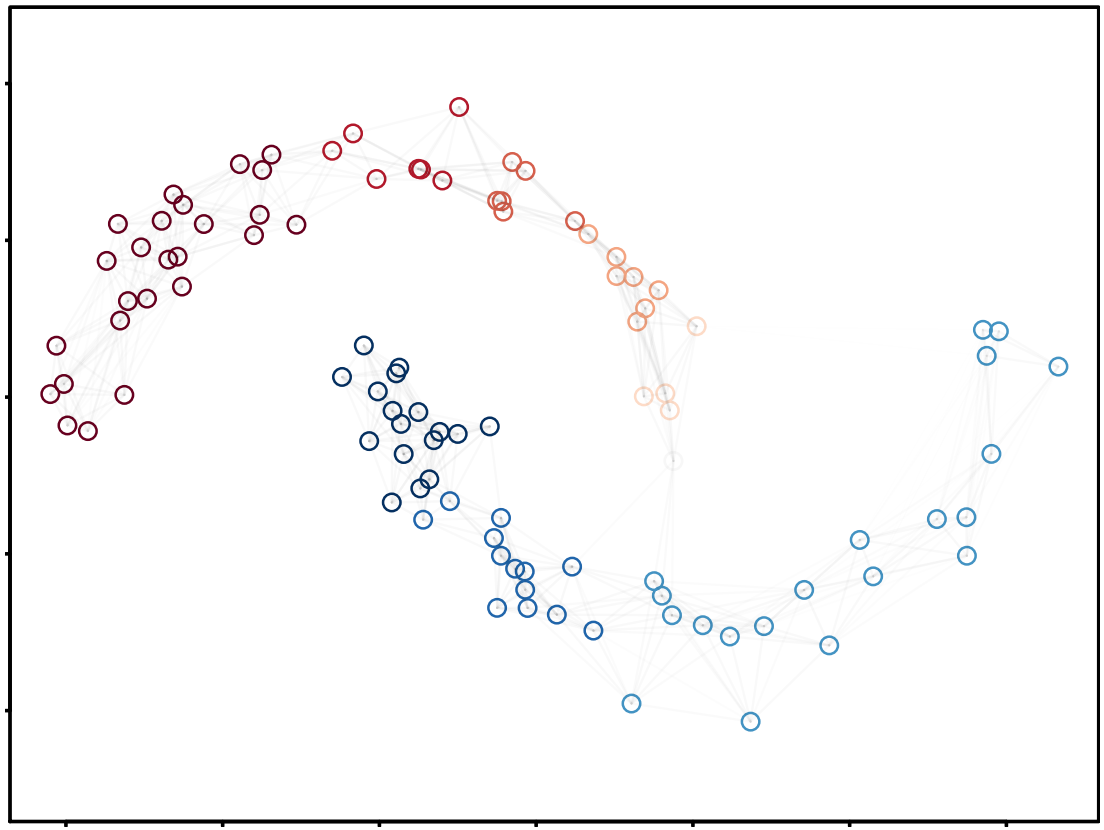
Example: Similarity graph for half-moon dataset

- 10NN similarity graph of 2D Euclidean data
- Gaussian kernel used as similarity function
- Note: similarity graphs model *local* similarities (but no dissimilarities)



Example: Spectral embeddings

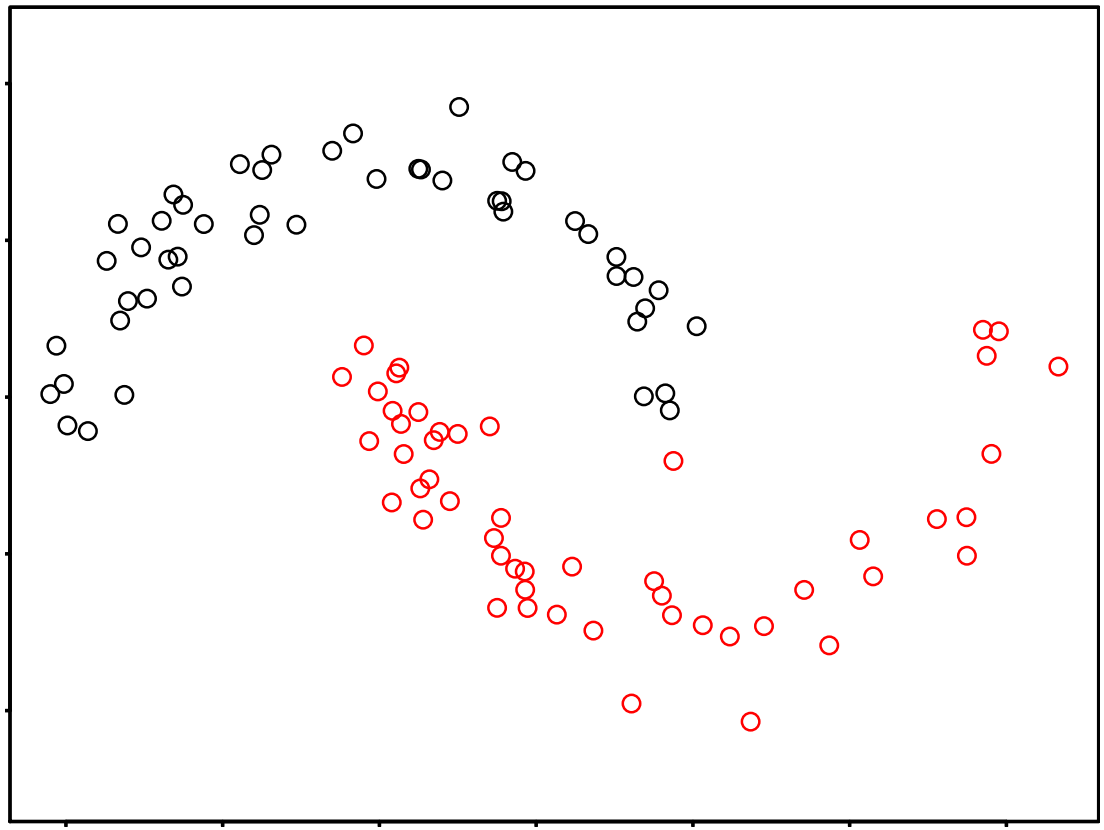
The spectral embeddings for this graph are:



- Shown here is the second component of the spectral embedding (a single real value per vertex)
- Red: negative
- Blue: positive

Example: Spectral clustering

Spectral clustering of this graph into two clusters gives:

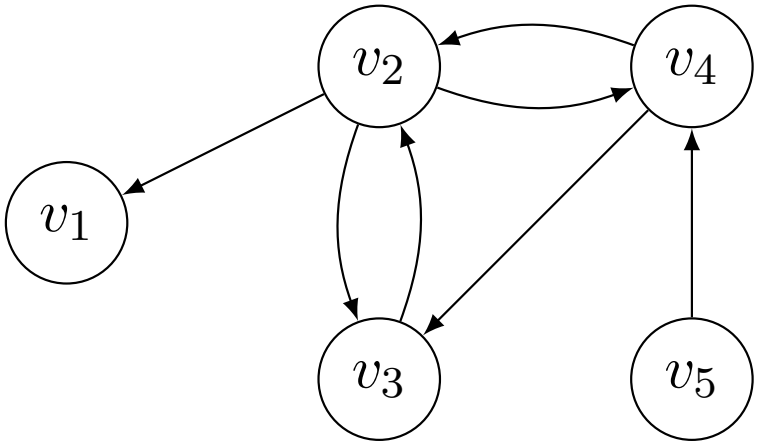


- Observe: points that are far away may end up in same cluster
  $\rightarrow$ Clustering only considers local similarities
- Clustering methods such as k-Means used on original data are global and give very different results

Outline

A graph is a matrix is a graph

- Let $G = (V, E)$ be a (weighted) graph
- Vertices $V = \{ v_1, \ldots, v_n \}$
- Edge $(i, j) \in E$ has positive weight a_{ij} (1 if unweighted)
- Convention: absent edges $(i, j) \notin E$ have weight $a_{ij} = 0$
- **Adjacency matrix** $\boldsymbol{A}$ is $n \times n$ matrix with entries a_{ij}
- Undirected graph $\implies$ $\boldsymbol{A}$ symmetric ($\boldsymbol{A} = \boldsymbol{A}^\top$)
- (Out-)degree of vertex i given by $d_i = \sum_j a_{ij} = \boldsymbol{a}_i^\top \mathbf{1}$
- **Degree matrix** $\boldsymbol{D}$ is $n \times n$ diagonal matrix with $d_{ii} = d_i$



$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Graph features

- Graph vertices and edges may be associated with features
  - Vertex features: $\boldsymbol{X} \in \mathbb{R}^{n \times D}$
  - Edge features: $\boldsymbol{X}^e \in \mathbb{R}^{|E| \times C}$
  - More generally, **property graph model**: arbitrary key-value pairs

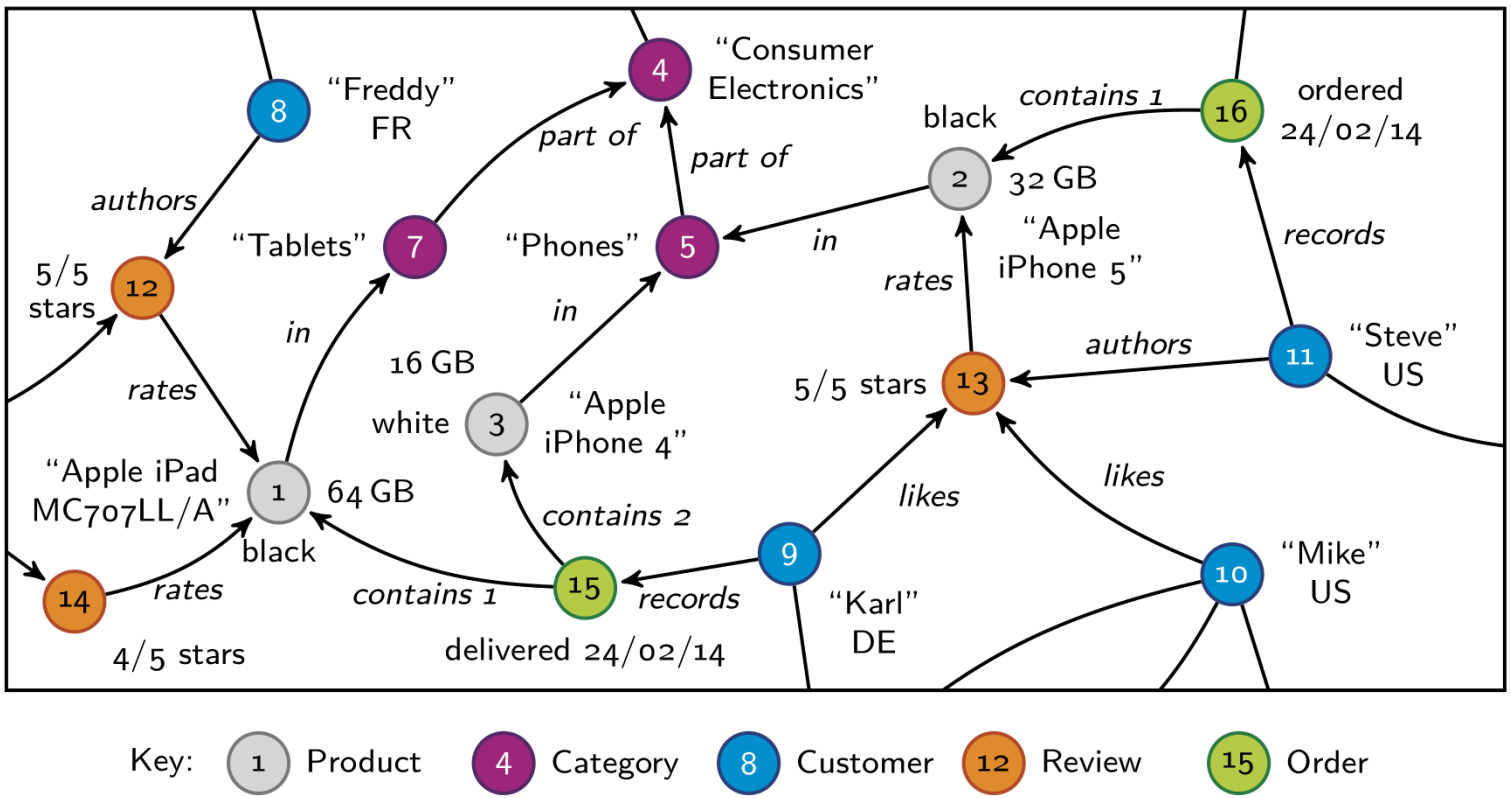
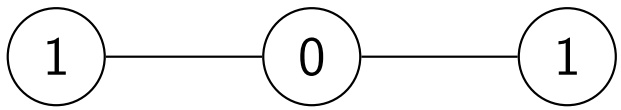


Fig. 1. Property graph illustrating a typical business scenario.

Graph signals

- A **graph signal** is a function $f : V \to \mathbb{R}$
  - Maps every vertex v to a real number f_v
    $\to$ Can be represented as a vector $\boldsymbol{f} \in \mathbb{R}^n$
  - Example: real-valued vertex features ($\boldsymbol{x}_{:f}$) and (as we will see) vertex embeddings or hidden vertex representations



$$f(v) = \begin{cases} 1 & v = v_1 \\ 0 & v = v_2 \\ 1 & v = v_3 \end{cases} \qquad \boldsymbol{f} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

- A key tool to **graph signal processing** is a matrix known as the graph Laplacian
  - More specifically, the eigendecomposition of the graph Laplacian
  - Eigenvalues and eigenvectors **expose structural graph properties** (e.g., connected components, spectral clustering)
  - Eigenvalues and eigenvectors allow to define operations such as **filters and convolutions** on graph signals

Graph Laplacian

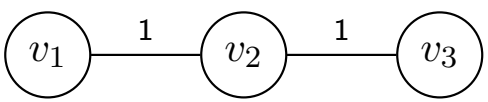
Definition

Let G be an undirected graph with positive edge weights. Denote by $\boldsymbol{A}$ the (weighted) adjacency matrix of G, and by $\boldsymbol{D}$ the degree matrix of G. Then

$$\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$$

is called the **(unnormalized) graph Laplacian** of G.

Note that self edges ($a_{ii} > 0$) do not affect the graph Laplacian.



G

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$\boldsymbol{D}$

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

$\boldsymbol{A}$

$$\begin{pmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix}$$

$\boldsymbol{L}$

Normalized graph Laplacians

Definition

There are two common normalizations of the graph Laplacian:

$$\boldsymbol{L}_{\text{sym}} = \boldsymbol{D}^{-1/2}\boldsymbol{L}\boldsymbol{D}^{-1/2} = \boldsymbol{I} - \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$$
$$\boldsymbol{L}_{\text{rw}} = \boldsymbol{D}^{-1}\boldsymbol{L} = \boldsymbol{I} - \boldsymbol{D}^{-1}\boldsymbol{A}$$

- Normalization is performed w.r.t. degree
- $\boldsymbol{L}_{\text{sym}}$ is symmetric, $\boldsymbol{L}_{\text{rw}}$ is not

$$\begin{pmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & -1/\sqrt{2} & 0 \\ -1/\sqrt{2} & 1 & -1/\sqrt{2} \\ 0 & -1/\sqrt{2} & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & -1 & 0 \\ -0.5 & 1 & -0.5 \\ 0 & -1 & 1 \end{pmatrix}$$
$$\boldsymbol{L} \qquad \boldsymbol{L}_{\text{sym}} \qquad \boldsymbol{L}_{\text{rw}}$$

Properties of the graph Laplacian (1)

Theorem

For every graph signal $\boldsymbol{f} \in \mathbb{R}^n$, $\boldsymbol{f}^\top \boldsymbol{L}\boldsymbol{f} = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}(f_i - f_j)^2$.

- $\boldsymbol{f}^\top \boldsymbol{L}\boldsymbol{f}$ is a **quadratic form** and small when neighboring vertices (connected with high-weight edges) take similar values according to $\boldsymbol{f}$ $\rightarrow$ I.e., small when values change "slowly"

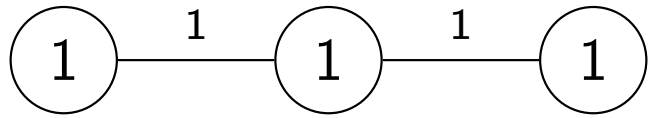
Proof.

$$\begin{aligned}
\boldsymbol{f}^\top \boldsymbol{L}\boldsymbol{f} &= \boldsymbol{f}^\top \boldsymbol{D}\boldsymbol{f} - \boldsymbol{f}^\top \boldsymbol{A}\boldsymbol{f} = \sum_{i=1}^{n} d_i f_i^2 - \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} f_i f_j \\
&= \frac{1}{2}\left(\sum_{i=1}^{n} d_i f_i^2 - 2\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} f_i f_j + \sum_{j=1}^{n} d_j f_j^2\right) \\
&= \frac{1}{2}\left(\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}(f_i^2 - 2 f_i f_j + f_j^2)\right) = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}(f_i - f_j)^2
\end{aligned}$$

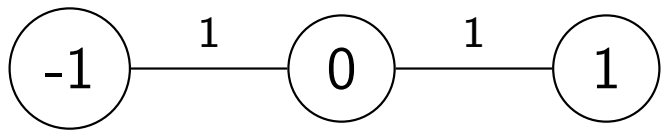
□

Properties of the graph Laplacian (2)

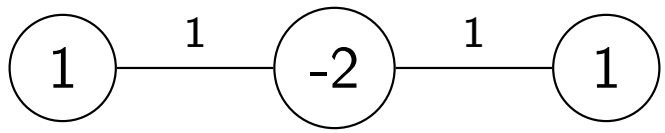
$$\boldsymbol{f}^\top \boldsymbol{L}\boldsymbol{f} = \frac{1}{2}\sum_{i,j} a_{ij}(f_i - f_j)^2$$



$$\boldsymbol{f}^\top \boldsymbol{L}\boldsymbol{f} = 0$$



$$\boldsymbol{f}^\top \boldsymbol{L}\boldsymbol{f} = 2$$



$$\boldsymbol{f}^\top \boldsymbol{L}\boldsymbol{f} = 18$$

Properties of the graph Laplacian (3)

Theorem

$\boldsymbol{L}$ is symmetric and positive semi-definite.

- Recall: A matrix $\boldsymbol{A}^{n\times n}$ is called **positive semi-definite** if $\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} \geq 0$ for any $\boldsymbol{x} \in \mathbb{R}^n$.
- Implies that $\boldsymbol{x}^\top \boldsymbol{L}\boldsymbol{x}$ is a convex function (in $\boldsymbol{x}$)
- Implies that $\boldsymbol{L} = \boldsymbol{P}\boldsymbol{P}^\top$ for some $\boldsymbol{P}$ (oriented incidence matrix)

Proof. Since $\boldsymbol{D}$ and $\boldsymbol{A}$ are symmetric, so is $\boldsymbol{L}$. Since $\boldsymbol{x}^\top \boldsymbol{L}\boldsymbol{x} \geq 0$ (see slide 16) for all $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{L}$ is positive semi-definite.

Outline

Background: Eigenvectors and eigenvalues

- A non-zero vector $\boldsymbol{v} \in \mathbb{R}^n$ is an **eigenvector** of $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ if

$$\boldsymbol{A}\boldsymbol{v} = \lambda \boldsymbol{v}$$

  - If $\boldsymbol{v}$ in an eigenvector of $\boldsymbol{A}$ so is $\boldsymbol{v}' = c\boldsymbol{v}$ for $0 \neq c \in \mathbb{R}$ as $\boldsymbol{A}\boldsymbol{v}' = c\boldsymbol{A}\boldsymbol{v} = c\lambda\boldsymbol{v} = \lambda\boldsymbol{v}'$
- λ is the corresponding **eigenvalue**
- Collection of eigenvalues is called **spectrum** of $\boldsymbol{A}$
- The eigenvalues are the roots of the **characteristic polynomial**

$$p_{\boldsymbol{A}}(\lambda) = \det(\boldsymbol{A} - \lambda \boldsymbol{I})$$

- We can factor $p_{\boldsymbol{A}}(\lambda)$ as

$$p_{\boldsymbol{A}}(\lambda) = (\lambda - \lambda_1)^{n_1} \cdots (\lambda - \lambda_N)^{n_N},$$

  where $1 \leq n_i \in \mathbb{N}$ and $\sum_i n_i = n$
  - n_i is called the **algebraic multiplicity** of λ_i
  - There are $1 \leq m_i \leq n_i$ linearly independent eigenvectors associated with eigenvalue λ_i
  - m_i is called the **geometric multiplicity** of λ_i
  - Note: Some eigenvectors can be complex

Background: Eigendecomposition

- The **eigendecomposition** of $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is given by

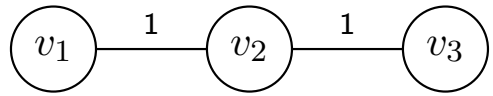
$$\boldsymbol{A} = \boldsymbol{Q} \boldsymbol{\Lambda} \boldsymbol{Q}^{-1},$$

  where
  - $\boldsymbol{Q}$ is square and has eigenvectors as its columns
  - $\boldsymbol{\Lambda}$ is diagonal and has eigenvalues on its diagonal

- Does not always exist; if it does, $\boldsymbol{A}$ is called **diagonalizable**
- Some properties
  1. When $\boldsymbol{A}$ is symmetric, $\boldsymbol{\Lambda}$ and $\boldsymbol{Q}$ are real-valued and $\boldsymbol{Q}$ can be chosen to be orthogonal
     - For example, when $\boldsymbol{L} = \boldsymbol{P}\boldsymbol{P}^\top$, consider SVD of $\boldsymbol{P} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$
     - $\boldsymbol{L} = \boldsymbol{P}\boldsymbol{P}^\top = (\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top)(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top)^\top = \boldsymbol{U}\boldsymbol{\Sigma}^2\boldsymbol{U}^\top = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$
  2. When $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}$, then $\operatorname{tr}(\boldsymbol{A}) \stackrel{\text{def}}{=} \sum_i a_{ii} = \sum_i \lambda_i = \operatorname{tr}(\boldsymbol{\Lambda})$
- Note:
  1. $\boldsymbol{L}$ (and $\boldsymbol{L}_{\text{sym}}$) are symmetric
  2. $\operatorname{tr}(\boldsymbol{L})$ = sum of degrees = $2\times$ sum of edge weights

Eigendecomposition of graph Laplacian

- Eigendecomposition $\boldsymbol{L} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$ always exists, since $\boldsymbol{L}$ symmetric
  - $\boldsymbol{Q}$ orthogonal
  - Convention in this lecture: order eigenvalues in *ascending* order
- Example



$$\boldsymbol{L} = \begin{pmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix}$$

$$\boldsymbol{Q} = \begin{pmatrix} 1/\sqrt{3} & -1/\sqrt{2} & 1/\sqrt{6} \\ 1/\sqrt{3} & 0 & -2/\sqrt{6} \\ 1/\sqrt{3} & 1/\sqrt{2} & 1/\sqrt{6} \end{pmatrix} \qquad \boldsymbol{\Lambda} = \begin{pmatrix} 0 & & \\ & 1 & \\ & & 3 \end{pmatrix}$$

- $\boldsymbol{q}_2$ known as **Fiedler vector** (cf. slide 8)
  - I.e., eigenvector corresponding to second-smallest eigenvalue
  - Here $\boldsymbol{q}_2 = \begin{pmatrix} -1/\sqrt{2} & 0 & 1/\sqrt{2} \end{pmatrix}^\top$

Spectrum of the graph Laplacian

Theorem

All eigenvalues are non-negative and real-valued and the smallest eigenvalue is $\lambda_1 = 0$*, i.e.,*

$$0 = \lambda_1 \leq \ldots \leq \lambda_{n-1} \leq \lambda_n.$$

An eigenvector corresponding to $\lambda_1 = 0$ *is the all-ones vector* $\mathbf{1}$*.*

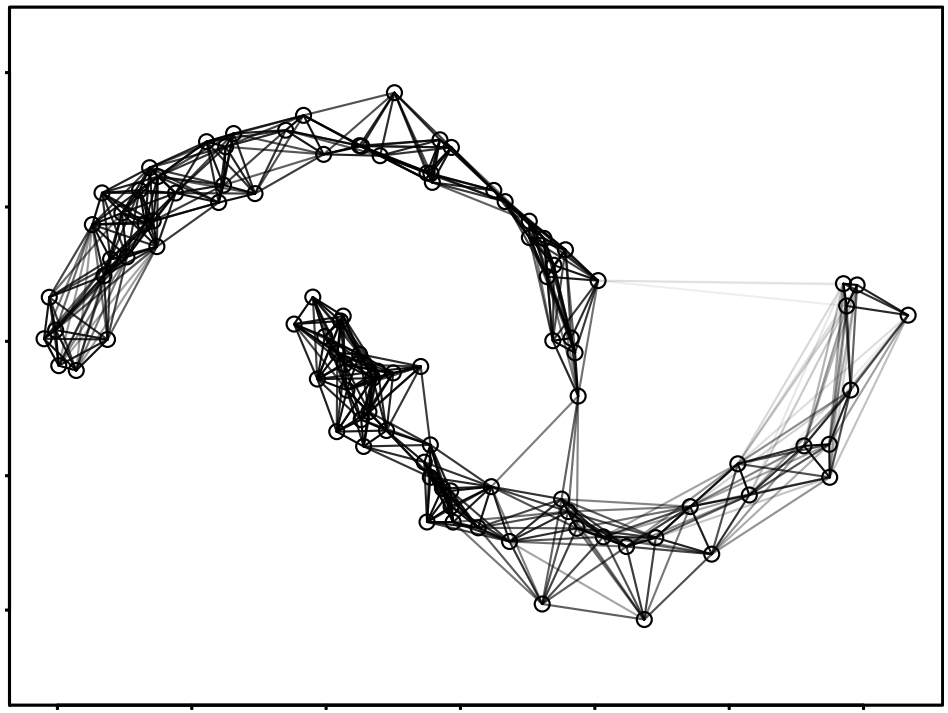
- I.e., $\lambda_1 = 0$, $\boldsymbol{q}_1 = \mathbf{1}/\sqrt{n}$

Proof. All eigenvalues of a symmetric matrix are real. To see that they are also non-negative, recall from slide 16 that

$$\boldsymbol{v}^\top \boldsymbol{L}\boldsymbol{v} = \frac{1}{2}\sum_{ij} a_{ij}(v_i - v_j)^2 \geq 0$$

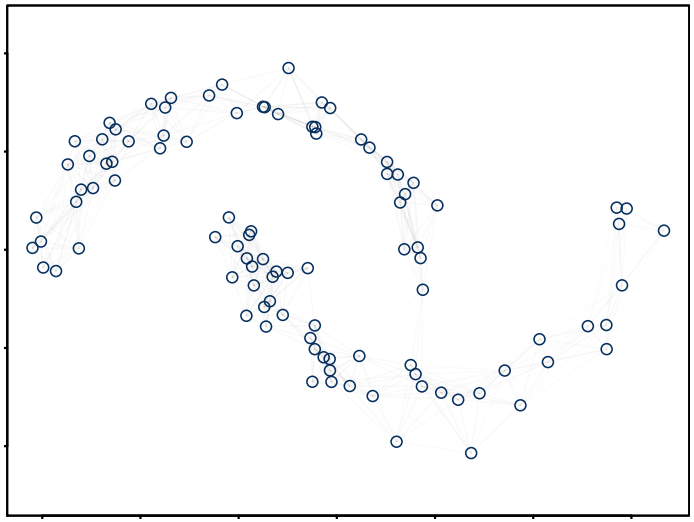
since all $a_{ij} \geq 0$. If $\boldsymbol{L}\boldsymbol{v} = \lambda\boldsymbol{v}$, then $0 \leq \boldsymbol{v}^\top \boldsymbol{L}\boldsymbol{v} = \lambda \left\|\boldsymbol{v}\right\|^2$ and thus $\lambda \geq 0$. Finally, $\mathbf{1}$ is an eigenvector with eigenvalue 0 as the row sums of $\boldsymbol{L}$ are all 0 by construction; hence $\boldsymbol{L}\mathbf{1} = \mathbf{0} = 0\mathbf{1}$.

Example: Eigenvalues/-vectors of graph Laplacian (1)

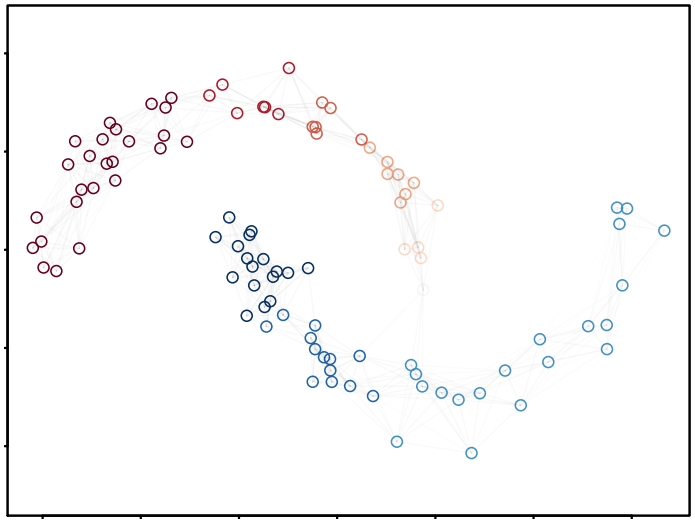


- We can view each eigenvector $\boldsymbol{q}_i \in \mathbb{R}^n$ of the graph Laplacian as a graph signal (with $f(v) = [\boldsymbol{q}_i]_v$)
- It holds: $\boldsymbol{q}_i^\top \boldsymbol{L} \boldsymbol{q}_i = \lambda_i$ (as we chose $\boldsymbol{Q}$ orthogonal so that $\|\boldsymbol{q}_i\| = 1$)
- As eigenvalues are sorted in ascending order, the first eigenvectors are more "consistent" with the graph (in that $\boldsymbol{q}_i^\top \boldsymbol{L} \boldsymbol{q}_i$ smaller / neighboring values more similar) than the later eigenvectors

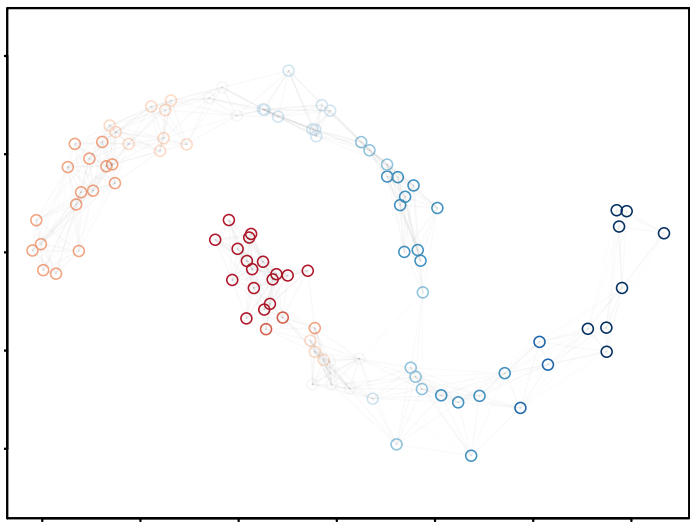
Example: Eigenvalues/-vectors of graph Laplacian (2)



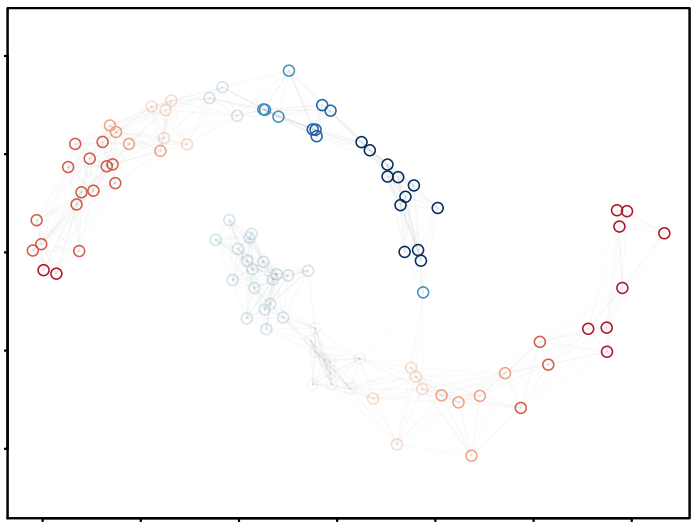
$\lambda_1 = 0$



$\lambda_2 = 0.04$ (**Fiedler vector**)



$\lambda_3 = 0.22$

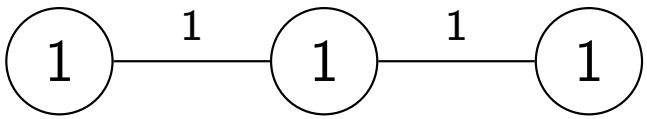


$\lambda_4 = 0.29$

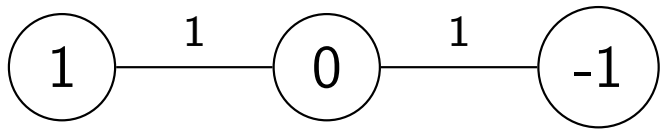
Connected graphs

Theorem

If G is connected, then eigenvalue 0 has multiplicity 1, i.e., $\lambda_2 > 0$.



$\lambda_1 = 0$



$\lambda_2 = 1 > 0$

Proof. Recall that $\mathbf{1}$ is an eigenvector of $\boldsymbol{L}$ with eigenvalue 0. Suppose that $\mathbf{0} \neq \boldsymbol{v} \neq c\mathbf{1}$ is an eigenvector of $\boldsymbol{L}$ with eigenvalue λ. Since G is connected, this implies that there are two neighboring vertices i' and j' such that $v_{i'} \neq v_{j'}$. Now

$$\lambda \|\boldsymbol{v}\|^2 = \boldsymbol{v}^\top \boldsymbol{L}\boldsymbol{v} = \frac{1}{2}\sum_{i,j} a_{ij}(v_i - v_j)^2 \geq a_{i'j'}(v_{i'} - v_{j'})^2 > 0$$

so that $\lambda > 0$.

Connected components

Theorem

The multiplicity k of eigenvalue 0 is equal to the number of connected components $G_1, \ldots, G_k$ of G. The corresponding eigenspace is spanned by the indicator vectors $\mathbf{1}_{G_i}$ (value 1 for vertices in G_i, value 0 otherwise).

Proof. Let $\boldsymbol{L}_1, \ldots, \boldsymbol{L}_k$ be the graph Laplacians of the connected components. We have $\lambda_1(\boldsymbol{L}_i) = 0$, $\lambda_2(\boldsymbol{L}_i) > 0$, and $\boldsymbol{v}_1(\boldsymbol{L}_i) = \mathbf{1}$. Order w.l.o.g. the vertices by their component so that

$$\boldsymbol{L} = \begin{pmatrix} \boldsymbol{L}_1 & & & \\ & \boldsymbol{L}_2 & & \\ & & \ddots & \\ & & & \boldsymbol{L}_k \end{pmatrix}.$$

Since $\boldsymbol{L}$ is block-diagonal, the spectrum of $\boldsymbol{L}$ is given by the union of the spectra of the $\boldsymbol{L}_i$. The corresponding eigenvectors are the eigenvectors of $\boldsymbol{L}_i$, filled with 0 at positions of other blocks.

Connected components (example)

$$\boldsymbol{L} = \begin{pmatrix} 1 & -1 & 0 & & & \\ -1 & 2 & -1 & & & \\ 0 & -1 & 1 & & & \\ & & & 1 & -1 & 0 \\ & & & -1 & 2 & -1 \\ & & & 0 & -1 & 1 \end{pmatrix}$$

$\lambda_1 = 0$

$\lambda_2 = 0$

$\lambda_3 = 1$

Rayleigh-Ritz theorem (1)

- Recall that we can interpret $\boldsymbol{f}^\top \boldsymbol{L}\boldsymbol{f}$ as a way to measure whether $\boldsymbol{f}$ assigns similar values to neighboring vertices
- **How can we pick $\boldsymbol{f}$ optimally?**
- Trivial solution: constant values
  - $f(v) = 1$ for all $v \in V$ implies $\boldsymbol{f}^\top \boldsymbol{L}\boldsymbol{f} = 0$
  - Not very helpful
  - Note: $\boldsymbol{f} \propto \boldsymbol{q}_1$ and $\boldsymbol{f}^\top \boldsymbol{L}\boldsymbol{f} = 0 = \lambda_1$
    $\rightarrow$ First eigenvector of $\boldsymbol{L}$ is trivial solution
- Non-trivial solution easy if G is not connected
  - Let G_i be i-th connected component
  - Keep signal constant within component (see previous slide)
  - $f(v) = \mathbb{I}[v \in G_i]$ implies $\boldsymbol{f}^\top \boldsymbol{L}\boldsymbol{f} = 0$
  - More helpful, exposes connected component
  - Note: $\boldsymbol{f} \propto \boldsymbol{q}_i$ and $\boldsymbol{f}^\top \boldsymbol{L}\boldsymbol{f} = 0 = \lambda_i$
    $\rightarrow$ i-th eigenvector (appropriately ordered) of $\boldsymbol{L}$ is solution
- What if G is not connected?

Rayleigh-Ritz theorem (2)

Theorem

The solution to the optimization problem

$$\begin{aligned} \textit{minimize} \quad & \boldsymbol{f}^\top \boldsymbol{L} \boldsymbol{f} \\ \textit{subject to} \quad & \boldsymbol{f} \perp \mathbf{1} \\ & \|\boldsymbol{f}\| = 1 \end{aligned}$$

is given by $\boldsymbol{q}_2$, i.e., the Fiedler vector.

- This is a consequence of the *Rayleigh-Ritz theorem*
  - States that when $\boldsymbol{L}$ is symmetric, eigenvalues/eigenvectors are the critical points of the **Rayleigh quotient** $\boldsymbol{f}^\top \boldsymbol{L} \boldsymbol{f} / \boldsymbol{f}^\top \boldsymbol{f}$ with $\boldsymbol{f} \neq 0$
- Consequence: $\boldsymbol{q}_2$ is best non-trivial solution
- Likewise: k-th best (i.e., orthogonal) solution given by $\boldsymbol{q}_k$
  - I.e., orthogonal to "prior" solutions $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_{k-1}$

Rayleigh-Ritz theorem (3)

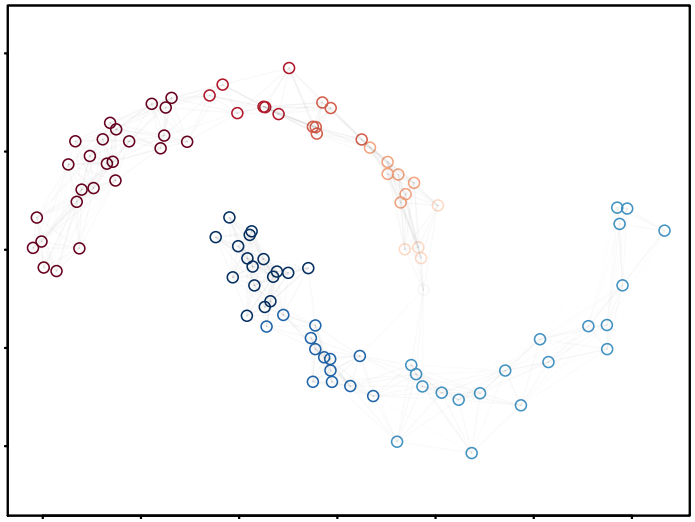
- Problem: $\min \boldsymbol{f}^\top \boldsymbol{L}\boldsymbol{f}$ s.t. $\boldsymbol{f} \perp \mathbf{1}$ and $\|\boldsymbol{f}\| = 1$
- Can show: problem is relaxation of "minimum ratio cut" problem
  - Roughly: partition a graph into two partitions such that partition sizes are balanced and few connections between partitions
  - Cost of minimum ratio cut $\geq \lambda_2$
- λ_2 tells us **how well** we can partition a graph
  - The smaller, the better
- Corresponding eigenvector $\boldsymbol{q}_2$ tells us **how to partition**
  - Simple heuristic: use sign of $\boldsymbol{q}_2$'s entries
  - Or run 2-means on entries of $\boldsymbol{q}_2$
  - Insight used in spectral clustering graph partitioning method

Outline

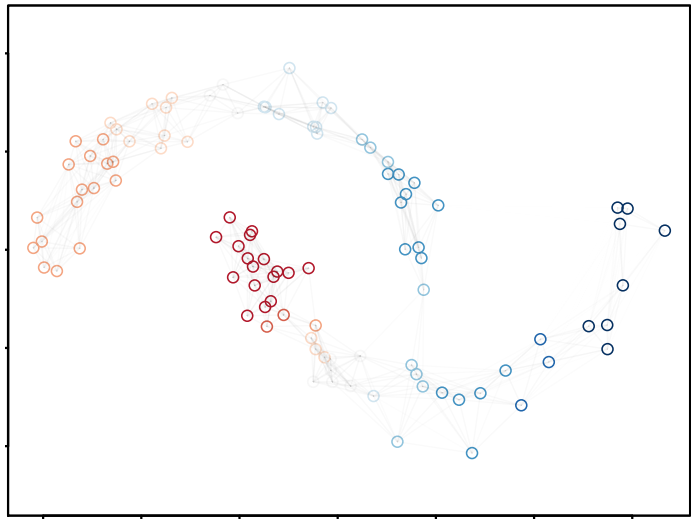
Spectral embeddings

- We can associate a **vertex embedding** $\boldsymbol{z}_v \in \mathbb{R}^D$ with each vertex $v \in V$
  - D is dimensionality of embedding
  - $\boldsymbol{z}_v$ is a low-dimensional representation of this vertex
- Vertices represented in D-dimensional coordinate system
  - k's entry of $\boldsymbol{z}_v$ = value of k-th coordinate for vertex v
- **Spectral embeddings**
  - Vertex embeddings obtained from eigenvectors $\boldsymbol{Q}$ of graph Laplacian
  - Let $\boldsymbol{Q}_D$ be $n \times D$ matrix of the first D columns of $\boldsymbol{Q}$ (i.e., eigenvectors corresponding to D smallest eigenvalues)
  - Spectral embedding of v = v'th row of $\boldsymbol{Q}_D$ (i.e., corresponding values of v in the D eigenvectors)
- Spectral clustering into K partitions $\approx$ run K-means on $\boldsymbol{Q}_K$
  $\rightarrow$ Spectral embeddings enhances clustering property of the data

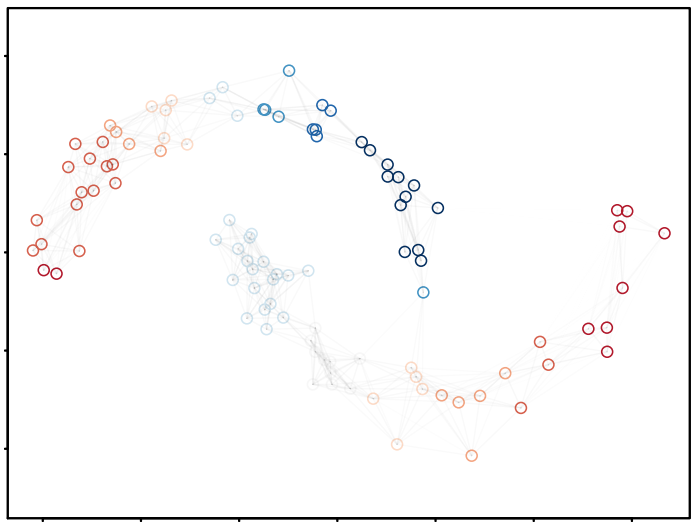
Example: Spectral embeddings



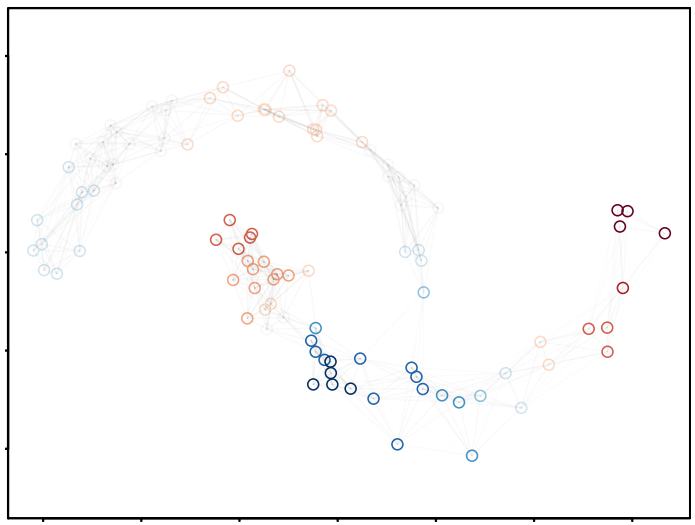
$\lambda_2 = 0.04$ (**Fiedler vector**)



$\lambda_3 = 0.22$

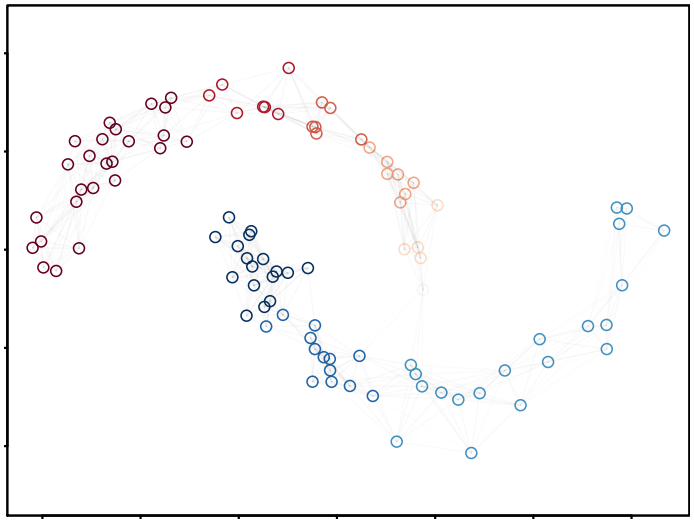


$\lambda_4 = 0.29$

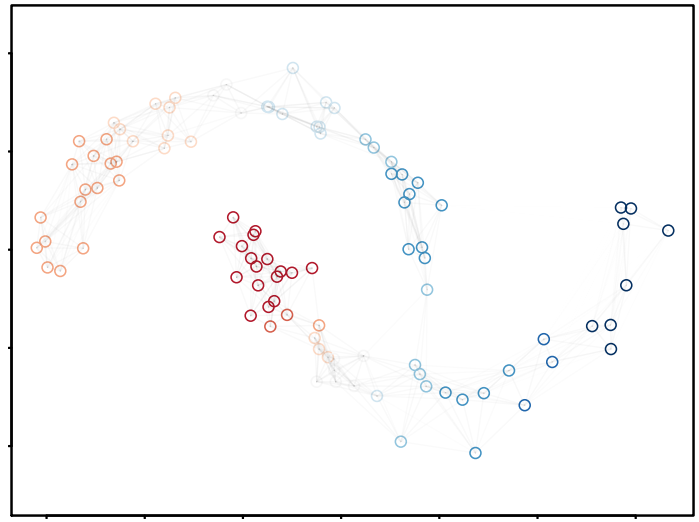


$\lambda_5 = 0.83$

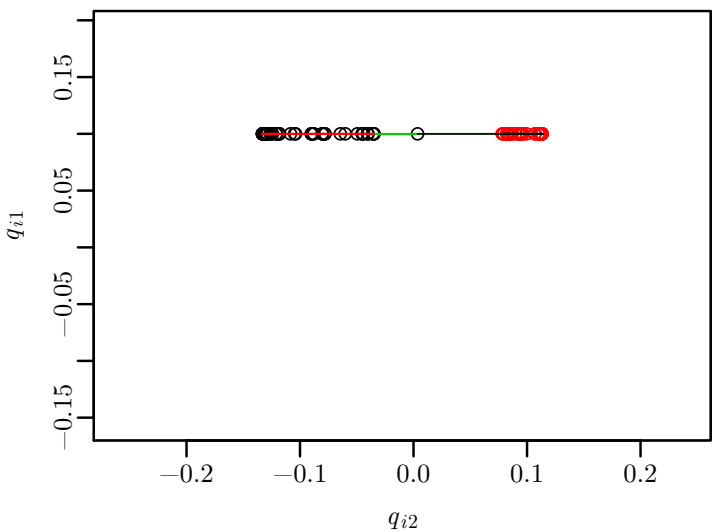
Example: Spectral clustering (1)



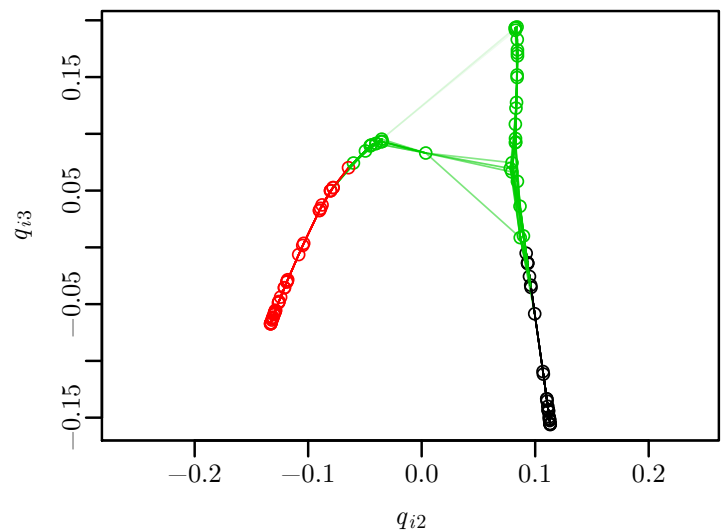
$\lambda_2 = 0.04$ (**Fiedler vector**)



$\lambda_3 = 0.22$

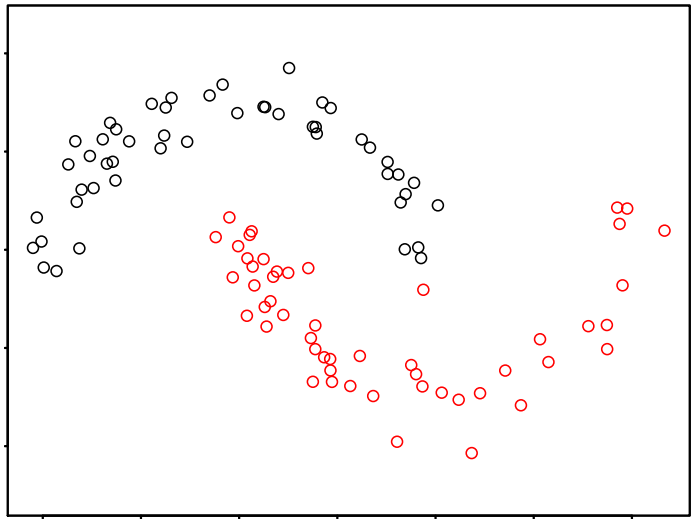


$\boldsymbol{Q}_2$

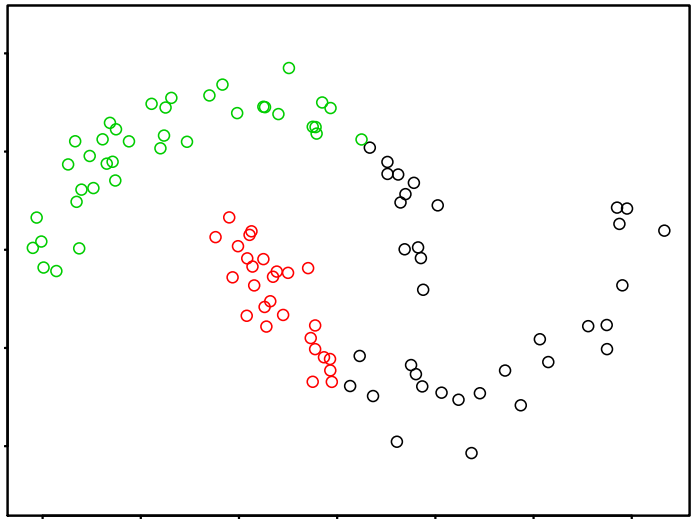


$\boldsymbol{Q}_3$ (w/o 1st column)

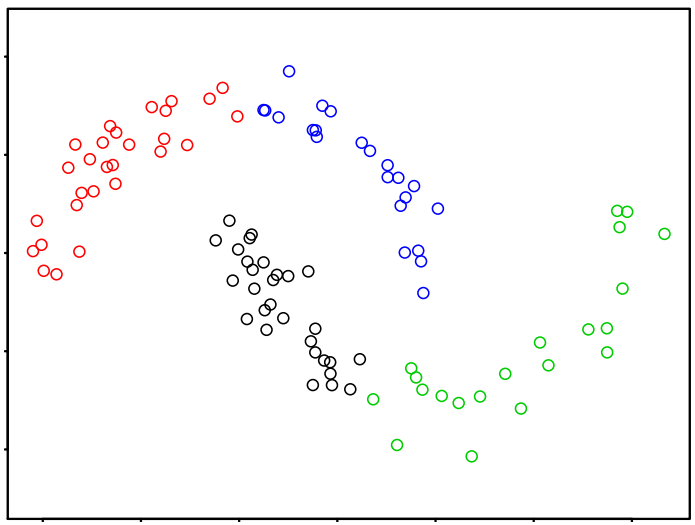
Example: Spectral clustering (2)



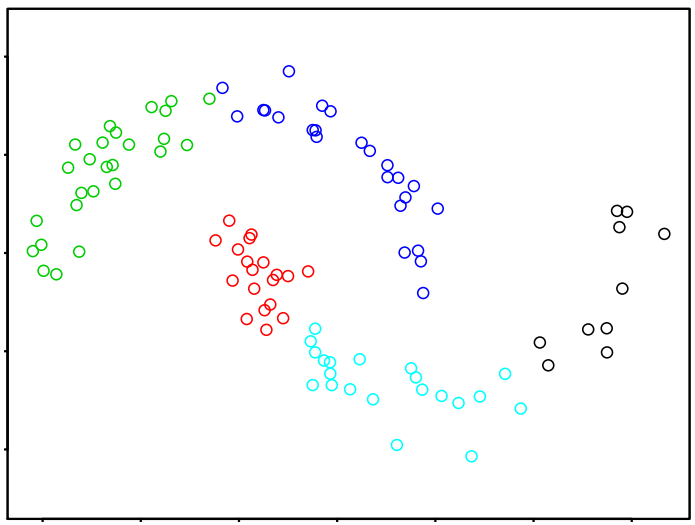
Spectral clustering ($k = 2$)



Spectral clustering ($k = 3$)



Spectral clustering ($k = 4$)



Spectral clustering ($k = 5$)

Deep Learning

09 – Graph Learning

Part 2: Deep Learning for Graphs

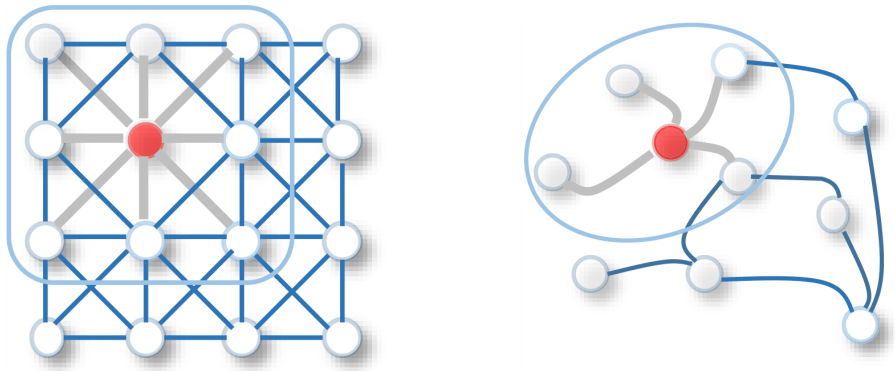
Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-1

Recap: Deep learning for graphs

- Deep learning methods can be applied to graph learning
- Generally, use graph structure to facilitate reasoning
  - Cf. CNNs, where we used a fixed grid structure
  - Now: graph structure expresses relationships



- Part-based view also applicable: input represented in terms of
  - Global features (if any)
  - Parts (vertices) and relationship between parts (edges)
    $\rightarrow$ Relationship between parts now input-dependent
  - Part features (= initial part embeddings): vertex and/or edge features

Wu et al., 2021

Recap: Key operations

We generally use the same type of operations as before:

1. **Contextualization**: incorporate information from other parts into each part embedding; e.g.,
   - Spectral embeddings
   - Graph convolutions
   - Message passing and graph recurrences
   - Graph transformers

   $\rightarrow$ Increase "receptive field" of each part's representation
2. **Local compute**: update each part embedding individually
   - As before; e.g., an MLP
3. **Pooling**: aggregate multiple (or all) part embeddings
   - Readout as before (e.g., pool vertex embeddings)
   - Pooling to change resoluti on (e.g., coarsen a graph) more involved

   $\rightarrow$ Increase "spatial invariance"

These operations are used to obtain higher-level representations (embeddings) of each part and/or of the entire graph.

Learned contextualization

- Last lecture: Spectral embeddings
  - Obtained solely from graph structure
  - Input: an undirected, unsigned graph
  - Output: vertex embeddings that encode "position"
  - Task-dependent information (such as labels) or additional information (e.g., graph/vertex/edge features) not taken into account
  - In this sense: Not learned
- This lecture: learned embeddings
  - Learned for the task at hand
  - Incorporate task-dependent and/or additional information
  - Typically: more general types of graphs (e.g., directed graphs)
- Key questions
  - How to perform contextualization and pooling for graph data?
  - How to design DL architectures for graph data?
  - How do different approaches compare to each other?

General approach

- Main focus: undirected graph $G = (V, E)$ with vertex features $\boldsymbol{x}_v$ for $v \in V$
  - Edge directions, edge features, and graph-level features can be handled by most of the approaches as well
  - We'll discuss this examplarily as we go
- Notation
  - $n = |V|$ for number of vertices
  - Z for embeddings dimensionality
  - Initial embeddings $\boldsymbol{z}_v^{(0)} \in \mathbb{R}^Z$ given by vertex features (e.g., $\boldsymbol{z}_v^{(0)} = \boldsymbol{x}_v$) and/or learned (e.g., transductive settings)
  - Operations compute higher-level features $\boldsymbol{z}_v^{(l)} \in \mathbb{R}^Z$
  - Represented as a matrix $\boldsymbol{Z}^{(l)} \in \mathbb{R}^{n \times Z}$
  - Readout and prediction head applied to final features

Outline

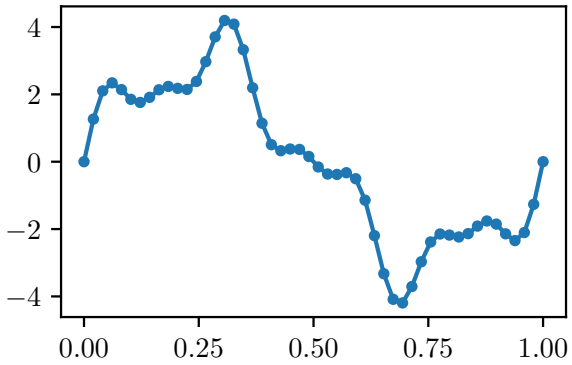
*Background: Discrete Fourier transform and convolutions

- **Discrete Fourier transform (DFT)**
  - Signal $\boldsymbol{x} \in \mathbb{C}^n$ in time domain
  - Transformed signal $\hat{\boldsymbol{x}} \in \mathbb{C}^n$ in frequency domain
  - Can be expressed via an $n \times n$ complex **DFT matrix** $\boldsymbol{F} \in \mathbb{C}^{n \times n}$ such that $\hat{\boldsymbol{x}} = \boldsymbol{F}\boldsymbol{x}$ (DFT) and $\boldsymbol{x} = \boldsymbol{F}^{-1}\hat{\boldsymbol{x}}$ (inverse DFT)
  - $\boldsymbol{F}/\sqrt{n}$ is unitary (complex analogue of orthogonal) $\rightarrow \boldsymbol{F}^{-1} = \frac{1}{n}\boldsymbol{F}^*$
- Elements $\hat{x}_k = \boldsymbol{f}_k^\top \boldsymbol{x}$ of $\hat{\boldsymbol{x}}$ called **Fourier coefficients**
- Original signal $\boldsymbol{x} = \boldsymbol{F}^{-1}\hat{\boldsymbol{x}} = \frac{1}{n}\sum_k \hat{x}_k \boldsymbol{f}_k^*$
  - Columns $\boldsymbol{f}_k^*$ of $\boldsymbol{F}^*$ are (scaled) samples from a complex sinusoid with **frequency** $(k-1)/N$ (i.e., $0, \frac{1}{N}, \ldots, \frac{N-1}{N}$)
    $\rightarrow$ Original signal = linear comb. of signals of multiple frequencies
  - Elements of $\boldsymbol{f}_k^*$ are slowly changing over time for small k (low frequency); in particular, $\boldsymbol{f}_1^* \propto \mathbf{1}$
  - And quickly changing for larger k (high frequency)
- $\hat{x}_k$ encodes "**strength**" (amplitude $|\hat{x}_k|$) of frequency $(k-1)/N$
- **Property:** (discrete circular) convolution in time domain corresponds to element-wise multiplication in frequency domain

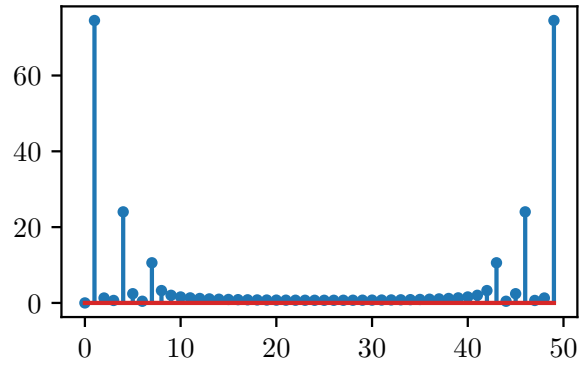
$$\widehat{\boldsymbol{x} \circledast \boldsymbol{w}} = \hat{\boldsymbol{x}} \odot \hat{\boldsymbol{w}}$$

Signal: $x(t) = 3\sin(2\pi t) + \sin(2\pi 4t) + 0.5\sin(2\pi 7t)$, 50 samples in $[0, 1]$

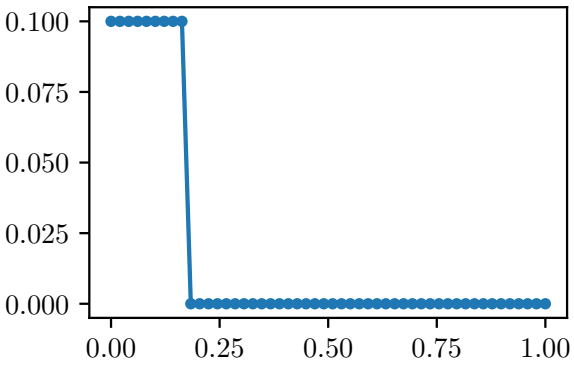
x_t



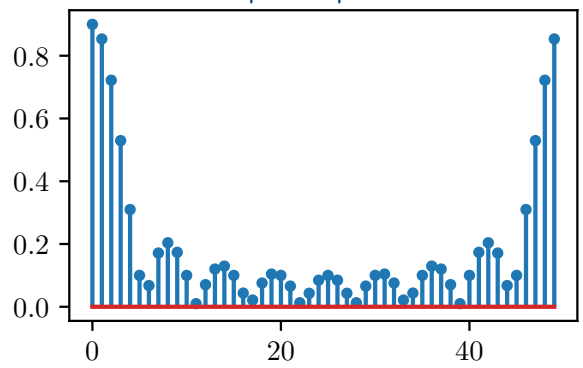
$|\hat{x}_k|$



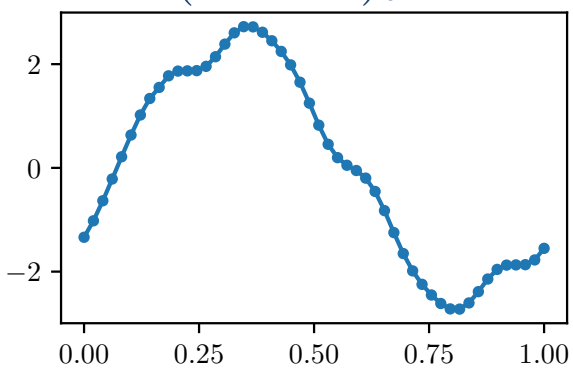
w_t



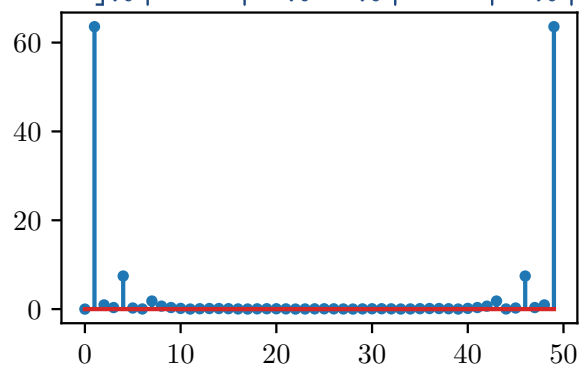
$|\hat{w}_k|$



$(\boldsymbol{x} \circledast \boldsymbol{w})_t$



$\widehat{|[\boldsymbol{x} \circledast \boldsymbol{w}]_k|} = |\hat{x}_k \hat{w}_k| = |\hat{x}_k|\,|\hat{w}_k|$



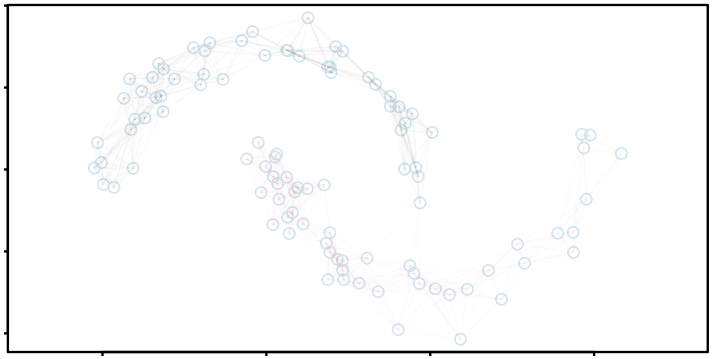
Graph Fourier transform

- **Graph Fourier transform**
  - Graph signal $\boldsymbol{x} \in \mathbb{R}^n$ in spatial (vertex) domain
  - Transformed signal $\hat{\boldsymbol{x}} \in \mathbb{R}^n$ in spectral domain
  - Fourier matrix $\boldsymbol{Q}^\top$ taken from eigendecomposition $\boldsymbol{L} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$ of graph Laplacian; we have $\hat{\boldsymbol{x}} = \boldsymbol{Q}^\top \boldsymbol{x}$ and $\boldsymbol{x} = \boldsymbol{Q}\hat{\boldsymbol{x}}$
  - $\boldsymbol{Q}$ is orthogonal $\rightarrow \boldsymbol{Q} = (\boldsymbol{Q}^\top)^{-1}$
- Elements $\hat{x}_k = \boldsymbol{q}_k^\top \boldsymbol{x}$ are **Fourier coefficients**
- Original signal $\boldsymbol{x} = \boldsymbol{Q}\hat{\boldsymbol{x}} = \sum_k \hat{x}_k \boldsymbol{q}_k$
  - Columns $\boldsymbol{q}_k$ of $\boldsymbol{Q}$ are eigenvectors and correspond to "**frequency**" λ_k (recall $0 = \lambda_1 \leq \cdots \leq \lambda_N$)
    $\rightarrow$ Original signal = linear comb. of signals of multiple frequencies
  - Elements of $\boldsymbol{q}_k$ are slowly changing over graph for small k (since $\boldsymbol{q}_k^\top \boldsymbol{L} \boldsymbol{q}_k = \lambda_k$); in particular, $\boldsymbol{q}_1 \propto \mathbf{1}$
  - And quickly changing for larger k (high frequency)
- $\hat{x}_k$ encodes "**strength**" of frequency λ_k
- **Define:** **graph convolution** in spatial domain as element-wise multiplication in frequency domain

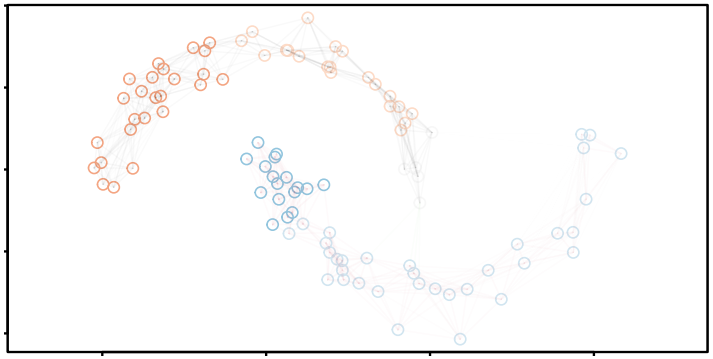
$$\widehat{\boldsymbol{x} *_G \boldsymbol{w}} \triangleq \hat{\boldsymbol{x}} \odot \hat{\boldsymbol{w}}$$

Eigenvectors/-values correspond to signals/frequencies

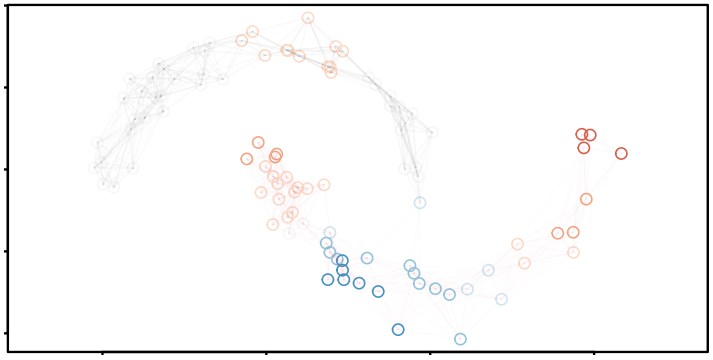
$\boldsymbol{q}_1,\ \lambda_1 = 0$



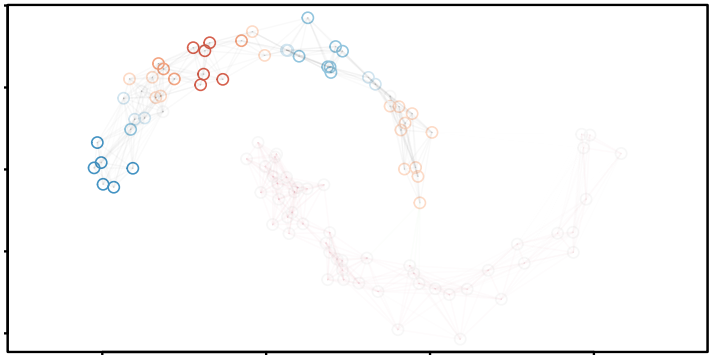
$\boldsymbol{q}_2,\ \lambda_2 = 0.04$



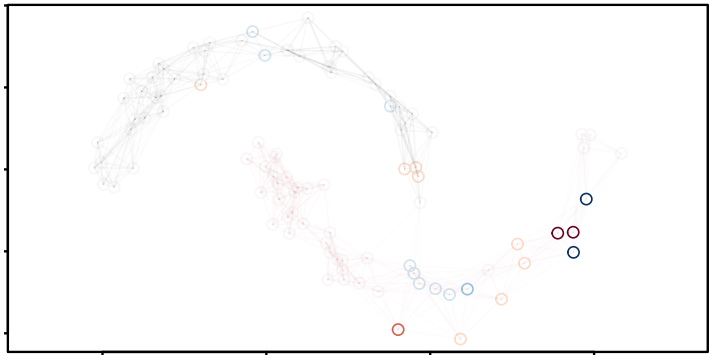
$\boldsymbol{q}_5,\ \lambda_5 = 0.83$



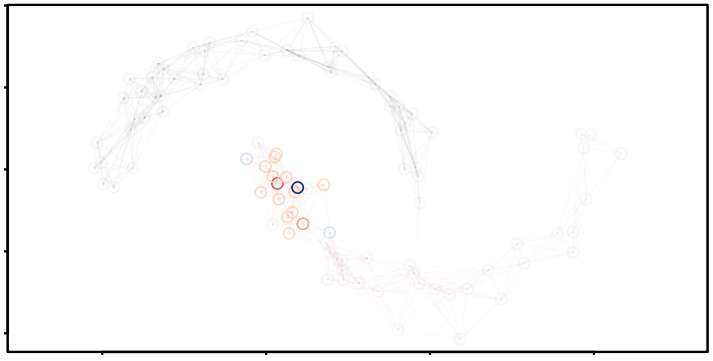
$\boldsymbol{q}_8,\ \lambda_8 = 2.03$



$\boldsymbol{q}_{20},\ \lambda_{20} = 7.09$

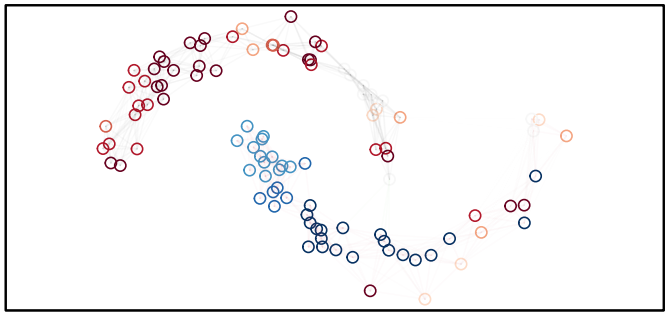


$\boldsymbol{q}_{100},\ \lambda_{100} = 16.00$

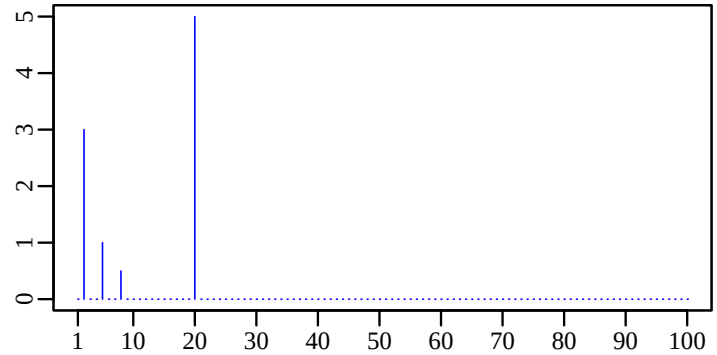


Signal: $\boldsymbol{x} = 3\boldsymbol{q}_2 + \boldsymbol{q}_5 + 0.5\boldsymbol{q}_8 + 5\boldsymbol{q}_{20}$, $n = 100$

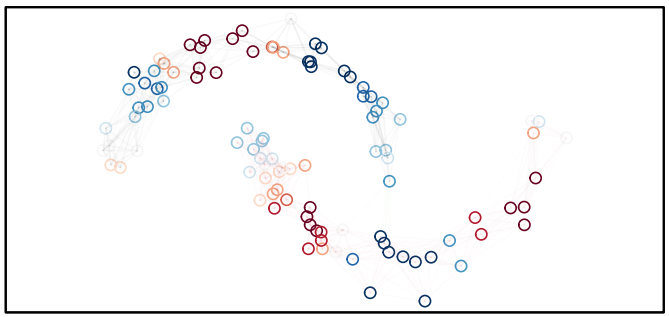
$\boldsymbol{x}$



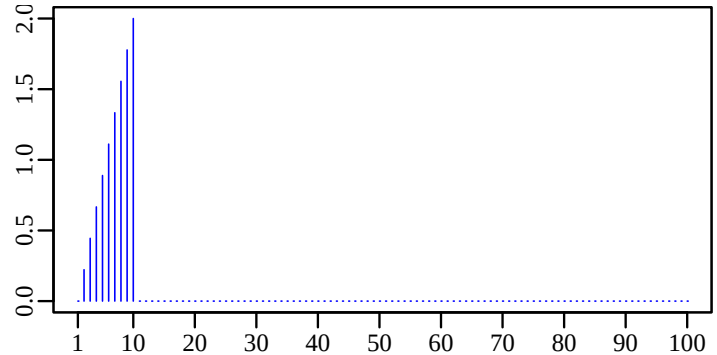
$\hat{\boldsymbol{x}}$



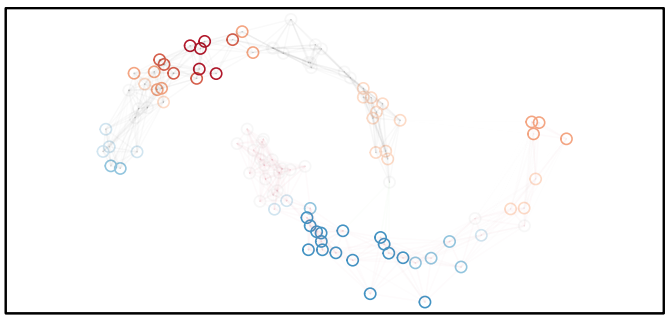
$\boldsymbol{w}$



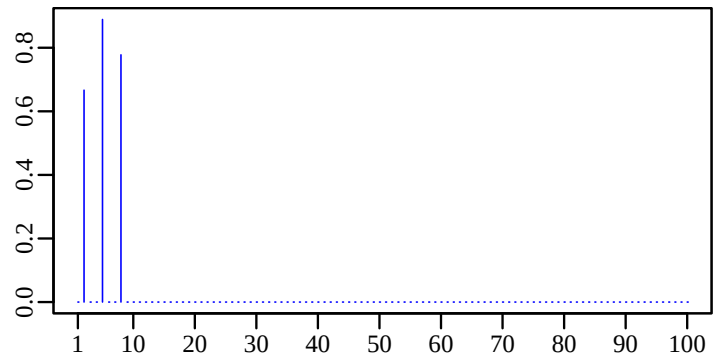
$\hat{\boldsymbol{w}}$ (= filter $\boldsymbol{\theta}$)



$\boldsymbol{x} *_G \boldsymbol{w}$



$\widehat{\boldsymbol{x} *_G \boldsymbol{w}} = \hat{\boldsymbol{x}} \odot \hat{\boldsymbol{w}}$



Spectral convolution and learnable filters

- Define: $\widehat{\boldsymbol{x} *_G \boldsymbol{w}} \triangleq \hat{\boldsymbol{f}} \odot \hat{\boldsymbol{w}}$
- Equivalently,
$$\boldsymbol{w} *_G \boldsymbol{x} = \boldsymbol{Q}\big(\underbrace{(\boldsymbol{Q}^\top \boldsymbol{w})}_{\hat{\boldsymbol{w}}} \odot \underbrace{(\boldsymbol{Q}^\top \boldsymbol{x})}_{\hat{\boldsymbol{f}}} \big)$$
- $\boldsymbol{\theta} \overset{\text{def}}{=} \hat{\boldsymbol{w}} = \boldsymbol{Q}^\top \boldsymbol{w}$ is called a **filter**
- **Spectral convolution with learnable filter**
$$\text{conv}_G(\boldsymbol{x}|\boldsymbol{\theta}) \overset{\text{def}}{=} \boldsymbol{Q}(\boldsymbol{\theta} \odot (\boldsymbol{Q}^\top \boldsymbol{x})) = \boldsymbol{Q}\boldsymbol{\Theta}\boldsymbol{Q}^\top \boldsymbol{x}$$
where $\boldsymbol{\Theta} = \text{diag}\,(\boldsymbol{\theta})$
  - We learn filter $\boldsymbol{\theta}$ directly
  - More efficient than learning $\boldsymbol{w}$ and transforming it
  - Easier to interpret: elements of $\boldsymbol{\theta}$ refer to spectral domain (frequencies), not graph domain (vertices)
  - I.e., entry $\boldsymbol{\theta}_k$ describes whether to reduce (< 1) / retain ($= 1$) / boost (> 1) corresponding frequency of graph signal $\boldsymbol{x}$

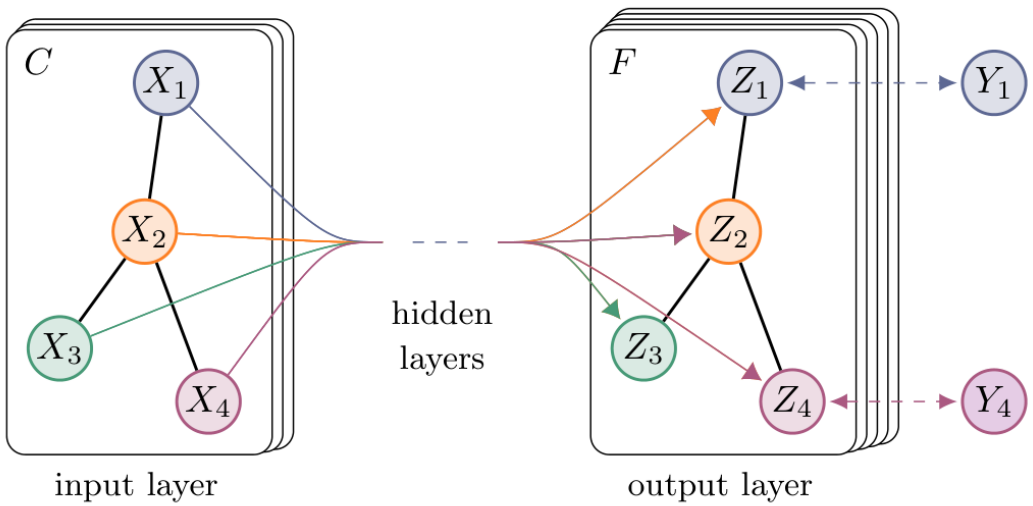
Spectral convolutional layers

- **Spectral convolutional layers** operate on multiple graph signals (**channels**) simultaneously
  - Z input channels, represented by an $n \times Z$ matrix $\boldsymbol{Z}^{(l-1)}$
  - Z' output channels, represented by an $n \times Z'$ matrix $\boldsymbol{Z}^{(l)}$
- Output channels are computed as follows

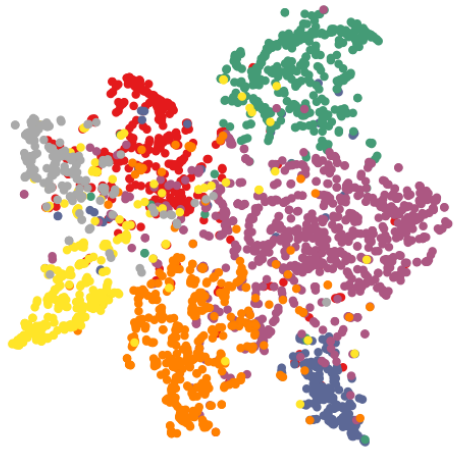
$$\boldsymbol{z}_{j'}^{(l)} = \sum_{j} \mathrm{conv}_G(\boldsymbol{z}_j^{(l-1)} | \boldsymbol{\theta}^{(j,j')})$$

  - I.e., j'-th output channel is sum of filtered input channels
  - One learnable filter for each input-output channel combination $\rightarrow Z \cdot Z'$ filters in total
  - Entire layer has $Z \cdot Z' \cdot n$ parameters
- Typically followed by a non-linearity

Example: vertex classification (transductive)



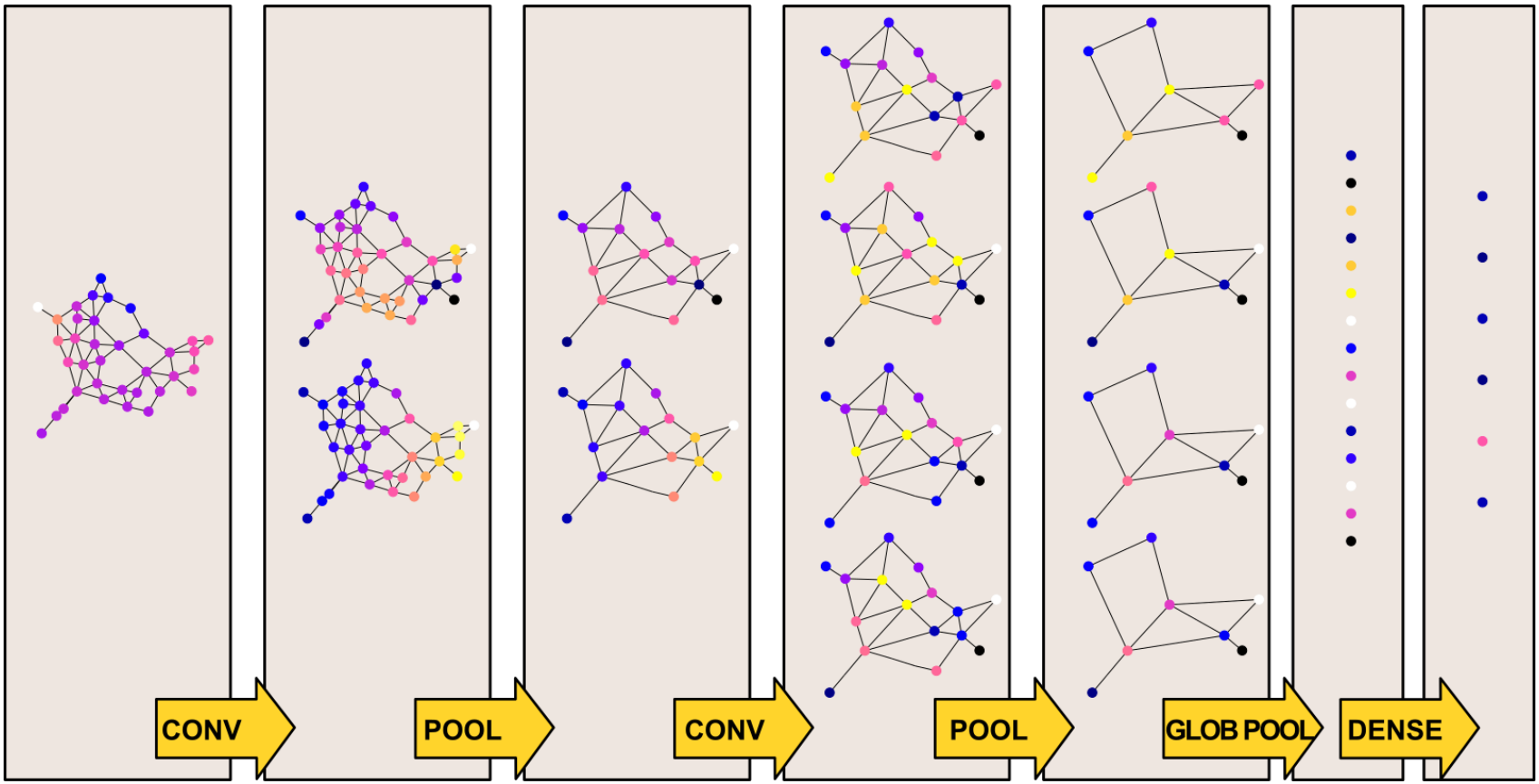
(a) Graph Convolutional Network



(b) Hidden layer activations

Figure 1: *Left*: Schematic depiction of multi-layer Graph Convolutional Network (GCN) for semi-supervised learning with C input channels and F feature maps in the output layer. The graph structure (edges shown as black lines) is shared over layers, labels are denoted by Y_i. *Right*: t-SNE (Maaten & Hinton, 2008) visualization of hidden layer activations of a two-layer GCN trained on the Cora dataset (Sen et al., 2008) using 5% of labels. Colors denote document class.

Example: graph classification



Supported graphs

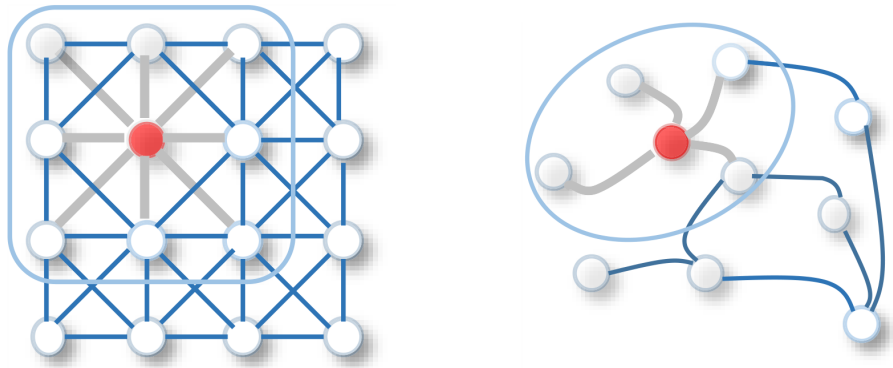
- Spectral convolution directly supports graphs that are undirected and optionally weighted (with non-negative weights)
  - Required for Laplacian / spectral analysis
  - Example: similarity graphs (GSSL is a key application, cf. sl. 14)
- But: can be more generally applied
- Directed graphs supported by
  1. Dropping edge directions (looses information)
  2. Applying convolution for each direction separately (and optionally pool)
- Multi-relational graphs (i.e., different edge categories) supported by
  - Dropping edge categories (looses information)
  - Applying convolution for each edge category separately (and optionally pool)
- More general vertex features (e.g., categories or textual data) supported by first embedding them into $\mathbb{R}^D$ using a subnetwork
- Similar ideas also applicable to other types of graph neural networks

Discussion

- Plain spectral convolution layers often infeasible in practice
  - Many variants exists to make them (more) feasible
- Large number of parameters; possible solutions:
  1. Use only first eigenvectors
  2. Interpolate $\boldsymbol{\theta}$ using a cubic spline
  3. Use a localized filter of form $\mathrm{diag}(\boldsymbol{\theta}) = \sum_{k=0}^{K} \alpha_k \boldsymbol{\Lambda}^k$ with parameters α_k
- Parameters cannot easily be shared across multiple graphs
  - Depend on graph Laplacian and, in particular, its eigenbasis $\boldsymbol{Q}$
- Not scalable to large graphs
  - Filters not localized in spatial domain
    $\rightarrow$ Each node's representation may be affected by all other nodes
  - Expensive; at least $O(n^2)$ for forward and backward pass
- Scalabilty problem addressed by localized filters
  - Localized filters are **K-localized**
    $\rightarrow$ Each node's representation only affected by its K-hop neighborhood
  - Can be computed in time complexity of only $O(K|E|)$
  - Especially effective/efficient when stacking multiple $K = 1$ layers
  - Reveals connection between spectral and spatial convolution (later)

Outline

Spatial convolutions and MPNNs



- Recall the convolution operations of CNNs: for each grid point,
  1. Consider the corresponding local region
  2. Compute a value for each grid point (using a filter)
  3. Aggregate the so-obtained values (sum up)
- Can we apply this idea to graphs?
- Yes! For each vertex:
  - Consider the 1-hop neighborhood of the vertex
  - Compute a value for each vertex (and the corresponding edge) in the 1-hop neighborhood
  - Aggregate the so-obtained values (e.g., sum up)
- The resulting operation is called **spatial convolution**
  - Graph neural networks (GNNs) using this operation are called **message-passing graph neural networks** (MPNNs)

Vertex-centric programming

We can express message passing operations using a framework known as **vertex-centric programming**.

1. Define four functions
   - $\textsc{Init}(v)$: initial vertex values
   - $\textsc{Message}(u, v)$: value passed along edge $u \to v$
   - $\textsc{Aggregate}(v, M)$: aggregate a set of messages
   - $\textsc{Update}(v, a)$: update vertex value given an aggregate
2. Assign an initial value $\boldsymbol{z}_v^{(0)} = \textsc{Init}(v)$ to each vertex $v \in V$
3. Perform **message passing** to compute $\boldsymbol{Z}^{(l)}$ from $\boldsymbol{Z}^{(l-1)}$

   for each $v \in V$
   $M_{\to v}^{(l)} \leftarrow \{ \boldsymbol{m}_{u \to v}^{(l)} \stackrel{\text{def}}{=} \textsc{Message}^{(l)}(u, v) : (u, v) \in E \}$
   $\boldsymbol{a}_v^{(l)} \leftarrow \textsc{Aggregate}^{(l)}(v, M_{\to v}^{(l)})$
   $\boldsymbol{z}_v^{(l)} \leftarrow \textsc{Update}^{(l)}(v, \boldsymbol{a}_v^{(l)})$
4. Optionally: repeat message passing multiple times (potentially using different $\textsc{Message}$, $\textsc{Aggregate}$, $\textsc{Update}$ functions)

Message-passing neural networks (MPNNs)

- Many graph algorithms can be expressed using message passing; e.g., shortest paths, number of connected components, PageRank
- An **MPNN** consists of multiple consecutive **MPNN layers**
  - Input: vertex embeddings $\boldsymbol{Z}^{(l-1)} \in \mathbb{R}^{n \times Z}$
  - Output: contextualized vertex embeddings $\boldsymbol{Z}^{(l)} \in \mathbb{R}^{n \times Z}$
  - Common (and perhaps confusing): terms GNN and GCN also used to refer to MPNNs in literature
- Each MPNN layer performs message passing using learned functions
  - Typically: learned $\textsc{Message}$
  - Sometimes: learned $\textsc{Init}$, $\textsc{Aggregate}$ and/or $\textsc{Update}$
  - Usually small subnetworks with a prefined architecture (e.g., GraphSAGE, MPNN, GAT, and GIN)
  - Separate set of parameters for each layer
- Given an MPNN architecture, learning/prediction as discussed

Example: GCN of Kipf and Welling (2017)

- Kipf and Welling (2017) motivated spatial convolution as a first-order approximation to spectral convolution
- Their **spatial GCN** uses: $\boldsymbol{Z}^{(l)} = \phi(\hat{\boldsymbol{A}}\boldsymbol{Z}^{(l-1)}\boldsymbol{W}^{(l)})$
  - $\boldsymbol{Z}^{(l)} \in \boldsymbol{R}^{n \times Z}$ are vertex representations at layer l
  - $\boldsymbol{W}^{(l)} \in \mathbb{R}^{Z \times Z}$ represents a learnable linear transformation
  - ϕ is a non-linearity (e.g., ReLU)
  - $\hat{\boldsymbol{A}}$ is degree-normalized and symmetric adjacency matrix
    - $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$ is adjacency matrix with self-edges
    - $\hat{\boldsymbol{A}} = \tilde{\boldsymbol{D}}^{-\frac{1}{2}}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-\frac{1}{2}}$
- We obtain (exercise):

$$\boldsymbol{m}_{u \to v}^{(l)} = \boldsymbol{m}_u^{(l)} \stackrel{\text{def}}{=} (\boldsymbol{W}^{(l)})^\top \boldsymbol{z}_u^{(l-1)} \qquad \text{(linear projection)}$$

$$\boldsymbol{a}_v^{(l)} = \sum_{\boldsymbol{m}_u^{(l)} \in M_{\to v}^{(l)}} \frac{1}{\sqrt{\tilde{d}_u}} \frac{1}{\sqrt{\tilde{d}_v}} \boldsymbol{m}_u^{(l)} \qquad \text{(normalized sum)}$$

$$\boldsymbol{z}_v^{(l)} = \phi(\boldsymbol{a}_v^{(l)}) \qquad \text{(non-linearity)}$$

Example: Graph Attention Networks (GAT)

- GATs use attention to aggregate over neighbors
  $\rightarrow$ learned aggregation
- With single-head attention, the model is

$$\boldsymbol{m}_{u\to v}^{(l)} = \boldsymbol{m}_u^{(l)} \stackrel{\text{def}}{=} (\boldsymbol{W}^{(l)})^\top \boldsymbol{z}_u^{(l-1)} \qquad \text{(linear projection)}$$

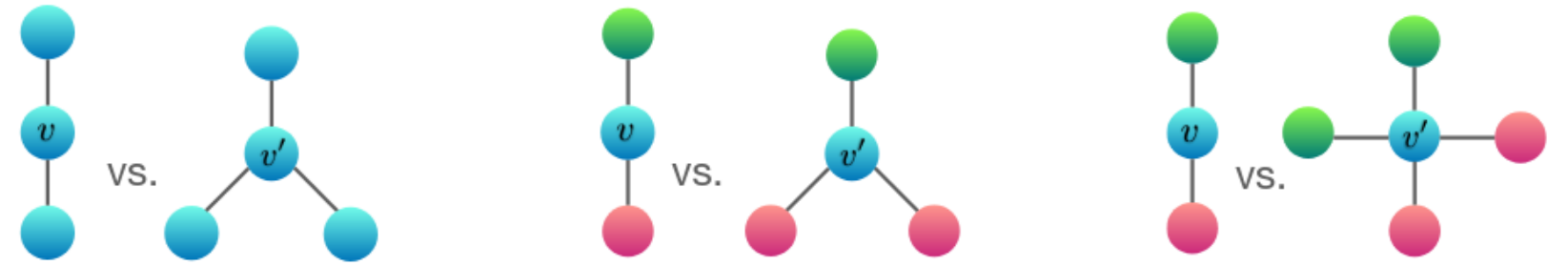
$$\boldsymbol{a}_v^{(l)} = \sum_{\boldsymbol{m}_u^{(l)} \in M_{\to v}^{(l)}} \alpha_{u,v}^{(l)} \boldsymbol{m}_u^{(l)} \qquad \text{(weighted average)}$$

$$\boldsymbol{z}_v^{(l)} = \phi(\boldsymbol{a}_v^{(l)}) \qquad \text{(non-linearity)}$$

- $\alpha_{u,v}^{(l)}$ is attention weight for message $\boldsymbol{m}_u^{(l)}$ from $u \to v$
  - Attention is performed over all messages in $M_{\to v}^{(l)}$
  - Key $\boldsymbol{k}_{u,v}^{(l)}$ is given by $\boldsymbol{m}_u^{(l)} \| \boldsymbol{m}_v^{(l)}$, where $\|$ denotes concatenation
  - Query $\boldsymbol{q}^{(l)}$ is learned
  - Attention scores obtained via $\mathrm{LeakyReLU}((\boldsymbol{q}^{(l)})^\top \boldsymbol{k}_{u,v}^{(l)})$
  - Attention weights via soft attention (softmax)

Aggregation and expressive power

- Expressive power of MPNNs analyzed by Xu et al. (2019)
- Aggregation function plays crucial role



(a) Mean and Max both fail (b) Max fails (c) Mean and Max both fail

Figure 3: **Examples of graph structures that mean and max aggregators fail to distinguish.**

- Figure shows neighborhood aggregation with mean and max; colors indicate messages (same color = same message sent)
- If v (left) and v' (right) have same aggregate ($\boldsymbol{a}_v = \boldsymbol{a}_{v'}$), MPNN fails to distinguish v from v', even though their neighborhood differs
- **Mean-aggregation** ignores size of neighborhood
- **Max-aggregation** additionally ignores repeated messages
- **Sum-aggregation** works in all the above cases (multiset)
- **Principled neighorhood aggregation (PRA)**: all of those and more, empirically strong

Graph isomorphism networks (GIN)

- Xu et al. (2019) showed that GINs given by

$$\boldsymbol{m}_{u\to v}^{(l)} = \boldsymbol{z}_u^{(l-1)} \qquad \text{(pass lower-level representation)}$$

$$\boldsymbol{a}_v^{(l)} = (1+\epsilon^{(l)})\boldsymbol{z}_v^{(l-1)} + \sum_{\boldsymbol{z}_u^{(l-1)} \in M_{\to v}^{(l)}} \boldsymbol{z}_u^{(l-1)} \qquad \text{(weighted sum)}$$

$$\boldsymbol{z}_v^{(l)} = f(\boldsymbol{a}_v^{(l)} | \boldsymbol{\theta}^{(l)}) \qquad \text{(local computation, parameterized)}$$

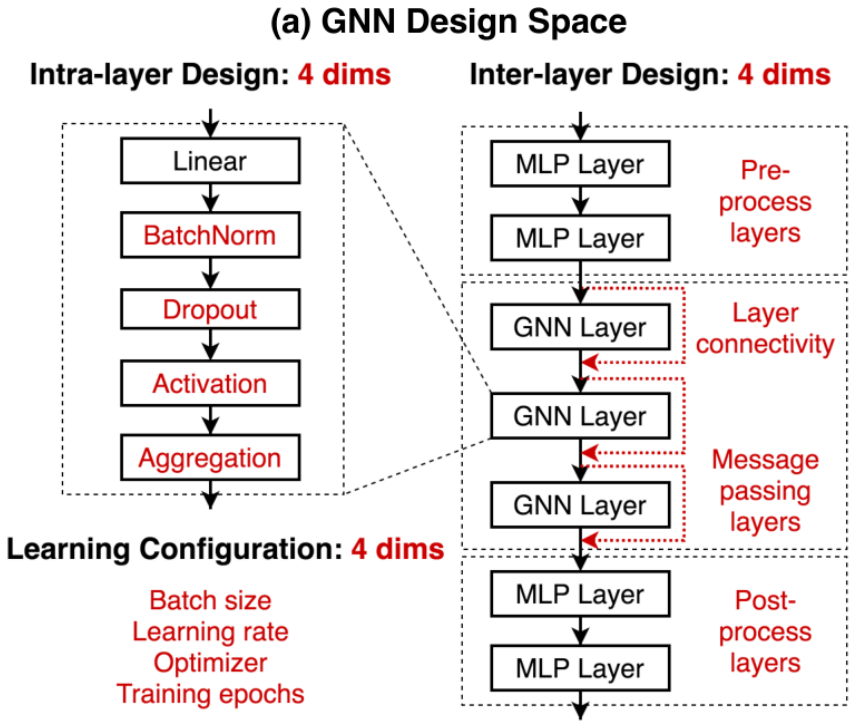
  are **maximally powerful** among all MPNNs
  - Requires that $f(\cdot|\boldsymbol{\theta}^{(l)})$ and final readout are injective functions
  - Roughly: GINs provably can produce different representations of "many" non-isomorphic graphs
  - "Many" means: non-isomorphism between each pair of graphs can be determined by Weisfeiler-Lehman test of isomorphism (a heuristic test)
  - No MPNN can distinguish other non-isomorphic graphs
    $\to$ MPNNs obtain same graph embeddings $\to$ same predictions (!)

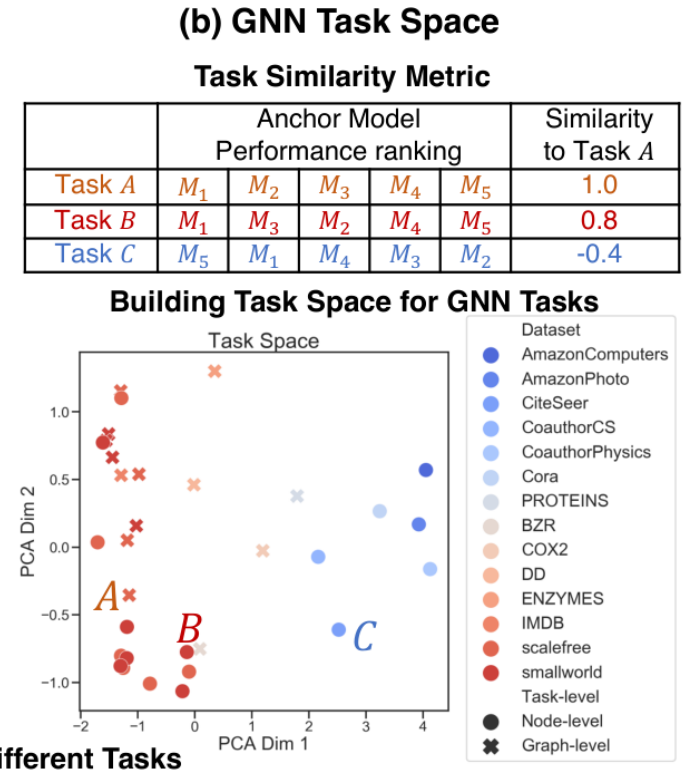
- Note: In practice, MPNNs can do more (e.g., distinguish two isomorphic graphs if they differ in their input vertex features)

Discussion

- Spatial convolution is common approach in practice
  - Relatively simple
  - Can be extended to support vertex/edge/graph attributes
  - Can be extended to support different types of edges (e.g., R-GCN)
  - Some libraries: DGL, Jraph, PyG
- Can be readily applied to many-graphs settings / new graphs
  - Natural due to use of aggregation function (vs. graph Laplacian)
  - Choice matters, esp. when graph properties differ significantly
  - E.g., different aggregation functions for different degrees
  - E.g., use first K neighbors based on suitable vertex ordering
- Efficiency can be a concern for large graphs
  - For high-degree vertices, aggregation can be expensive
  - "Neighborhood sampling" methods often used

Example: Results of You et al. (2020) (1)





(c) Best GNN Designs Found in Different Tasks

	Pre-process layers	Message passing layers	Post-process layers	Layer connectivity	Aggregation
Task A: graph-IMDB	2	8	2	skip-sum	sum
Task B: node-smallworld	1	8	2	skip-sum	sum
Task C: node-CiteSeer	2	6	2	skip-cat	mean

Figure 1: **Overview of the proposed GNN design and task space**. **(a)** A GNN design space consists of 12 design dimensions for intra-layer design, inter-layer design and learning configuration. **(b)** We apply a fixed set of "anchor models" to different tasks/datastes, then use the Kendall rank correlation of their performance to quantify the similarity between different tasks. This way, we build the GNN task space with a proper similarity metric. **(c)** The best GNN designs for tasks A, B, C. Notice that tasks with higher similarity share similar designs, indicating the efficacy of our GNN task space.

Example: Results of You et al. (2020) (2)

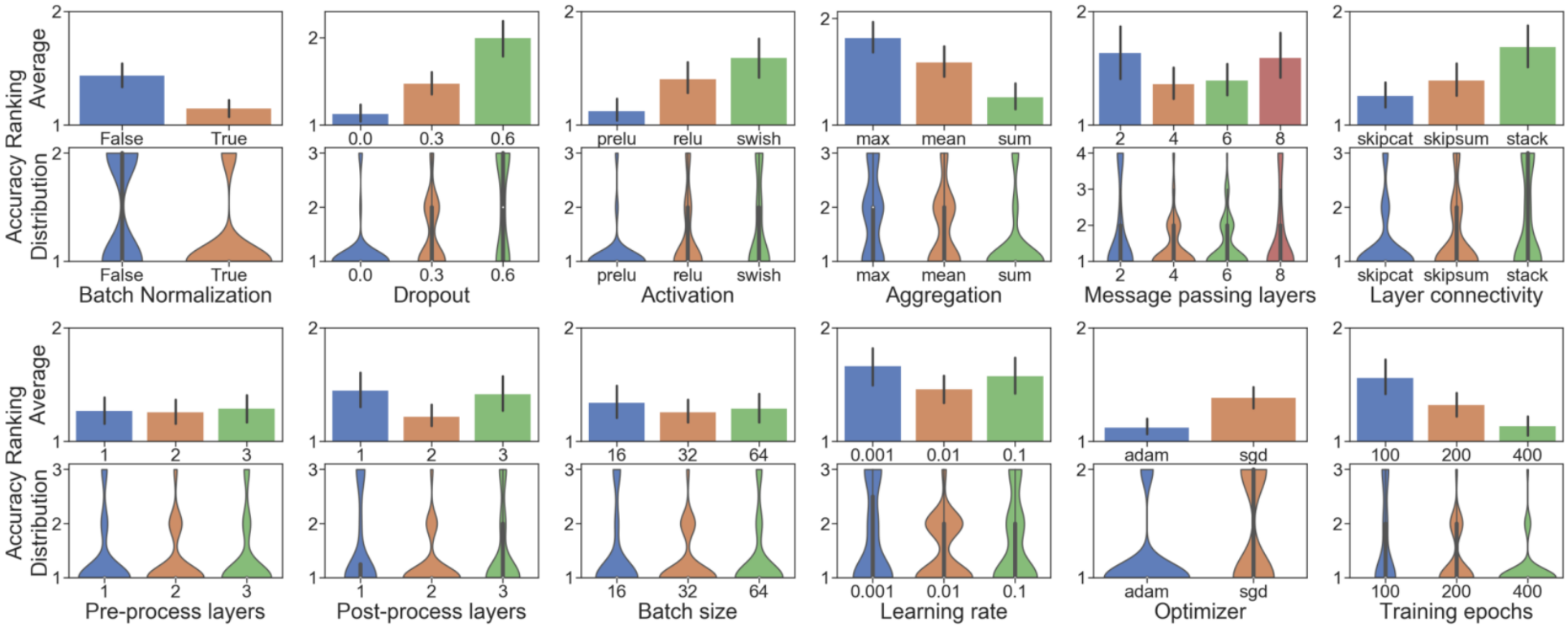


Figure 3: **Ranking analysis for GNN design choices in all 12 design dimensions**. Lower is better. A tie is reached if designs have accuracy / ROC AUC differences within $\epsilon = 0.02$.

Example: NBFNet for link prediction (1)

Neural Bellman Ford networks (NBFNet) are a method for link prediction.

- In a single large graph with different edge types (such as knowledge graphs or heterogeneous information networks)
- Use the **labeling trick**
  - Label the "source vertex" for which to predict an outgoing edge with a (learned) set of features, all other vertices with zero features
  - Perform message passing with a GNN (accounting for edge types)
  - Compute the dot product of the source vertex embedding to all other vertex embeddings $\rightarrow$ the higher, the more likely a link
- Method trained to predict edges present in a given graph
- Resulting model applied to predict missing edges

Example: NBFNet for link prediction (2)

Table 3: Knowledge graph completion results. Results of NeuraLP and DRUM are taken from [46]. Results of RotatE, HAKE and LowFER are taken from their original papers [52, 76, 1]. Results of the other embedding methods are taken from [52]. Since GraIL has scalability issues in this setting, we evaluate it with 50 and 100 negative triplets for FB15k-237 and WN18RR respectively and report MR based on an unbiased estimation.

Class	**Method**	**FB15k-237**					**WN18RR**				
		MR	**MRR**	**H@1**	**H@3**	**H@10**	**MR**	**MRR**	**H@1**	**H@3**	**H@10**
Path-based	Path Ranking [35]	3521	0.174	0.119	0.186	0.285	22438	0.324	0.276	0.360	0.406
	NeuralLP [69]	-	0.240	-	-	0.362	-	0.435	0.371	0.434	0.566
	DRUM [46]	-	0.343	0.255	0.378	0.516	-	0.486	0.425	0.513	0.586
Embeddings	TransE [6]	357	0.294	-	-	0.465	3384	0.226	-	-	0.501
	DistMult [68]	254	0.241	0.155	0.263	0.419	5110	0.43	0.39	0.44	0.49
	ComplEx [58]	339	0.247	0.158	0.275	0.428	5261	0.44	0.41	0.46	0.51
	RotatE [52]	177	0.338	0.241	0.375	0.553	3340	0.476	0.428	0.492	0.571
	HAKE [76]	-	0.346	0.250	0.381	0.542	-	0.497	0.452	0.516	0.582
	LowFER [1]	-	0.359	0.266	0.396	0.544	-	0.465	0.434	0.479	0.526
GNNs	RGCN [48]	221	0.273	0.182	0.303	0.456	2719	0.402	0.345	0.437	0.494
	GraIL [55]	2053	-	-	-	-	2539	-	-	-	-
	NBFNet	**114**	**0.415**	**0.321**	**0.454**	**0.599**	**636**	**0.551**	**0.497**	**0.573**	**0.666**

Excursion: GRNNs

- **Receptive field of MPNN with K layers = K-hop neighborhood**
  - May not be sufficent, esp. since K often small (say, 2–8)
  - MPNNs cannot solve tasks such as shortest paths or determining the number of connected components (exercise)
- Can be mitigated; e.g., by readout/pooling or by adding transformer layers (both coming up)
- Alternative: **graph recurrent neural networks**
  - Lift the notion of recurrence from RNNs to graphs
  - Hidden state = vertex features
  - Recurrence = an MPNN layer
  - Approach: apply MPNN layer repeatedly until a fix point is reached (nothing changes anymore)
  - Number of layers (and hence receptive field) thus not hard-coded
    $\rightarrow$ Above problems can be solved (exercise)
  - Used, for example, to model graph processes (fixed-structure graphs, time-dependent features; e.g., weather station networks)

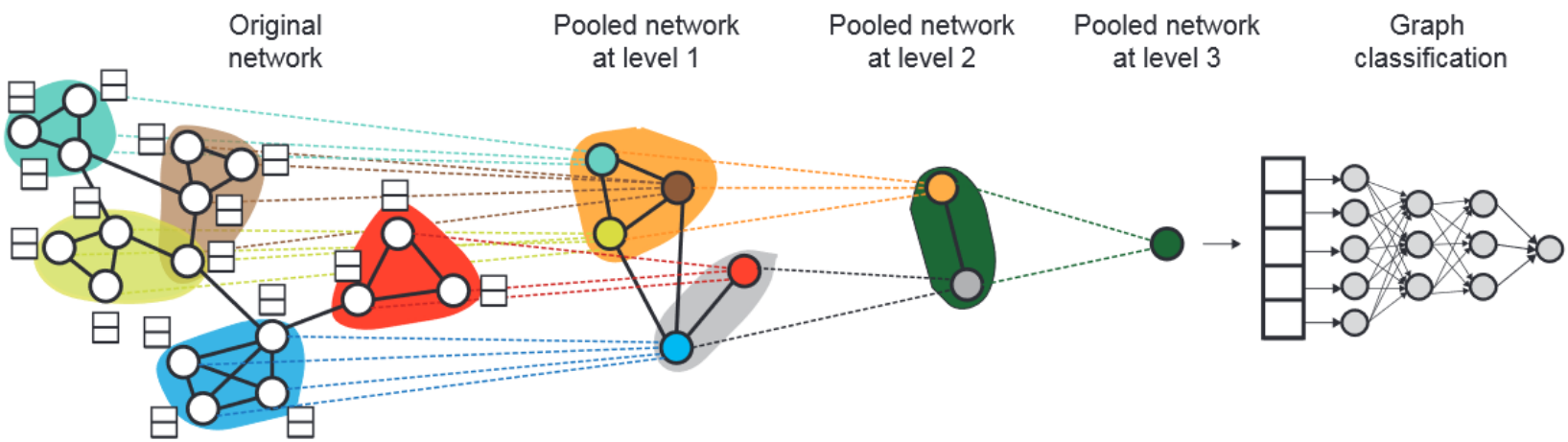
Outline

Readout

- Recall: **readout** operation obtains a graph-level representation
  - Generally want **order invariance**
  - Means: vertex or edge order does not affect outcome
  - Relationship to graph isomorphism problem implies that structurally different graphs may end up with same representation
- Basic approaches
  - Statistics such as element average/sum/maximum of vertex embeddings
    $\rightarrow$ May not be representative to distinguish different graphs
  - Fully connected layers
    $\rightarrow$ Order invariance not guaranteed, small graphs only
  - Use a special "global" vertex that is connected to all vertices
- Effectiveness increased via pooling

Pooling

- Cf. CNNs: gradually decrease resolution via pooling (or strided convolutions), then fully connected layers
  - How can we perform such pooling on a graph?
- General idea: use a (hierarchical) graph clustering algorithm
  - Cluster the vertices (e.g., using spectral clustering)
  - Pool the embeddings of vertices of each cluster

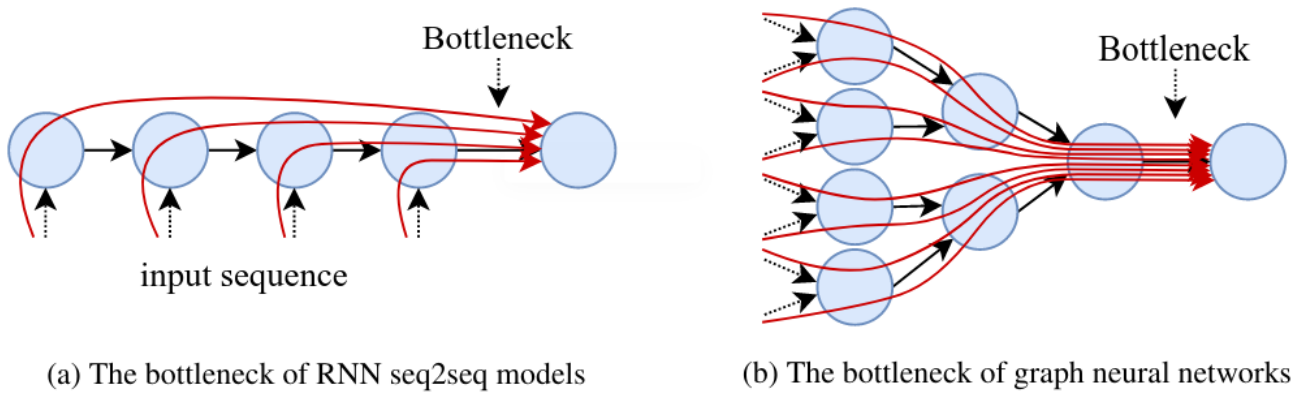


Ying et al., 2018

Outline

Problems with message-passing GNNs

- **Oversmoothing**: (certain) deeper GNNs can perform worse than shallower ones, since repeated neighborhood information can "wash out" structure information (Xu et al., 2018)
- **Oversquashing**: as GNN depth increases, information from a (potentially) exponential number of paths squashed into fixed-size representations (Alon and Urav, 2021)



(a) The bottleneck of RNN seq2seq models

(b) The bottleneck of graph neural networks

Figure 1: The bottleneck that existed in RNN seq2seq models (before attention) is strictly more harmful in GNNs: information from a node's exponentially-growing receptive field is compressed into a fixed-size vector. Black arrows are graph edges; red curved arrows illustrate information flow.

- **Hard inductive bias**: graph structure limits model's computation graph
- Limited expressive power (see slides 24 and 25)

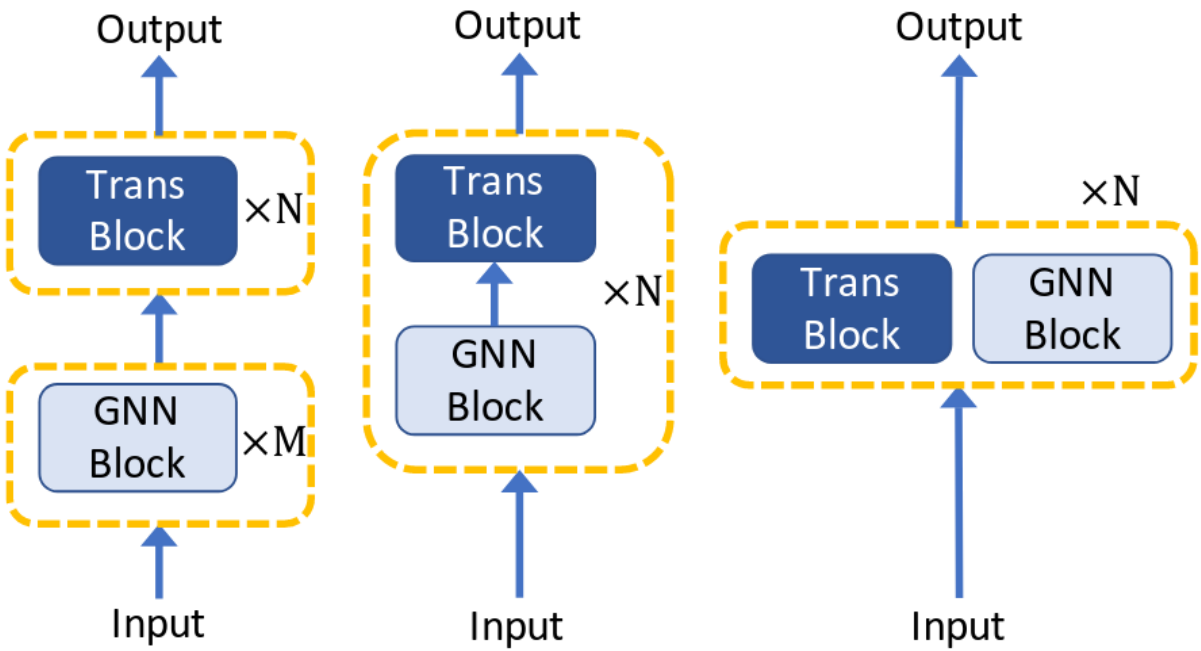
Transformers to the rescue

- Idea: Add Transformer layers to compute graph
  - Self-attention $\approx$ message passing between all nodes
  - All vertex embeddings updated based on self-attention
  - Every updated vertex embedding (in principle) depends on all of the vertices (i.e., update is **global**)
  - Avoids oversmoothing, since no (simple) neighborhood aggregation
  - Mildens oversquashing, since information is "distributed" across all vertices
  - Avoids hard inductive bias, since computation graph not determined by (but instead informed by) graph structure

- How to best use Transformers for graphs?
  - A direct application ($\boldsymbol{Z}^{(l-1)}$ in, $\boldsymbol{Z}^{(l)}$ out) of a Transformer layer ignores graph structure
  - Key question: **How to incorporate graph-structure information?** $\rightarrow$ active research area
  - Our focus: overview of general approaches

Approach 1: Mix-in GNN layers

- Combine Transformer layers with GNNs
  - GNNs: local, structure-aware
  - Transformers: global, not structure-aware when applied directly



(a) Before Trans (b) Alternatively (c) Parallel

Approach 2: Structure-aware positional embeddings

- Recall that Transformers for NLP use positional embeddings (PE) to encode ordering
  - E.g., sine functions of varying frequency; Vaswani (2017) used

$$\text{PE}(pos)_{2i} = \sin(pos/10000^{2i/d_{\text{model}}})$$
$$\text{PE}(pos)_{2i+1} = \cos(pos/10000^{2i/d_{\text{model}}})$$

  - Low frequency (i large): change slowly in neighborhood
  - High frequency (i small): change quickly in neighborhood
- Natural analogue for graphs are spectral embeddings (cf. 09-1 and sl. 9ff)
  - Each eigenvector corresponds to a "frequency"
  - Again, we use the eigenvectors associated with the smallest eigenvalues (lower frequencies)
  - E.g., directly used by Dwivedi and Bresson (2020)
  - E.g., **spectral attention networks** by Beaini et al. (2021) (next slide)

Example: Spectral Attention Networks (SAN, 2021)

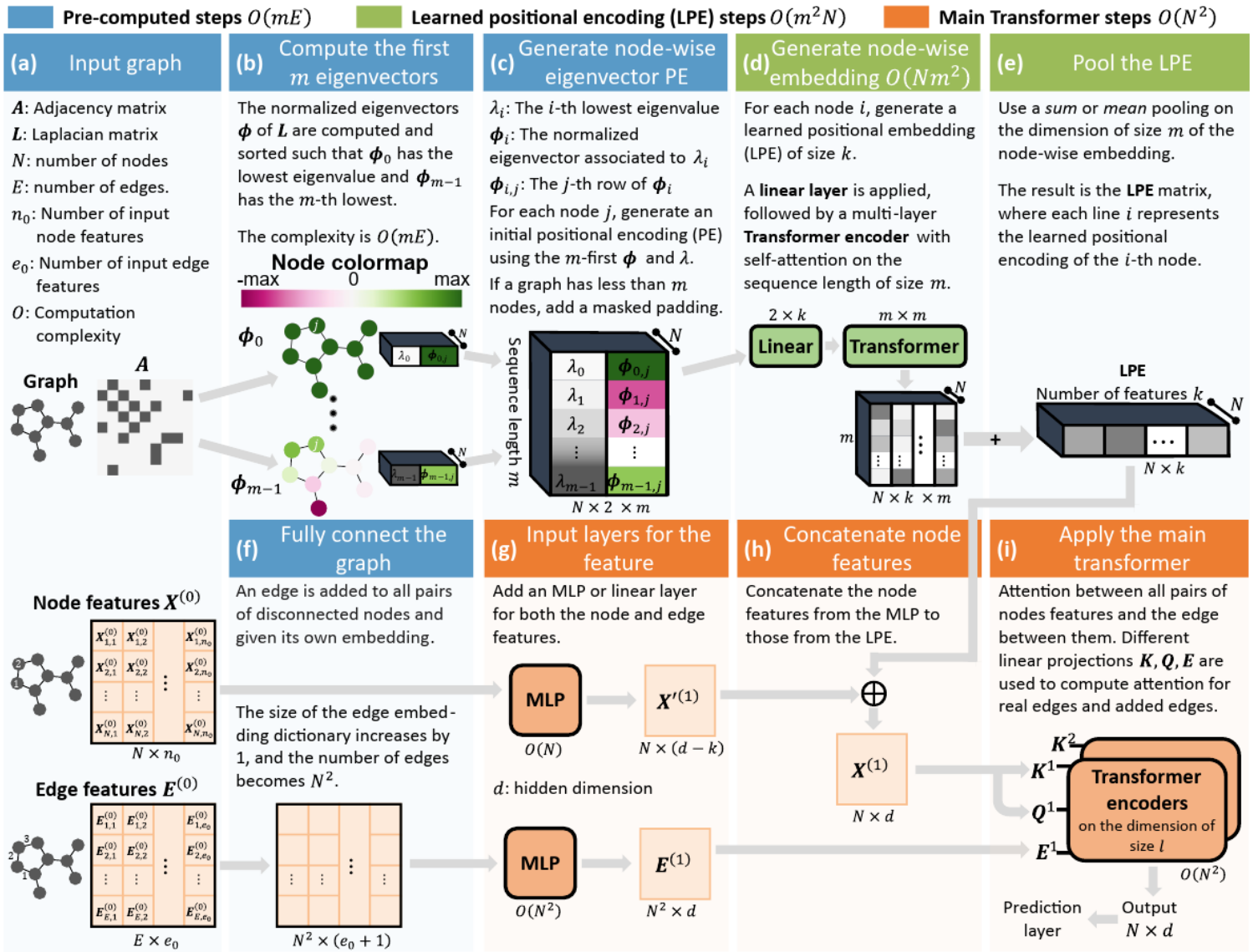


Figure 1: The proposed SAN model with the node LPE, a generalization of Transformers to graphs.

Example: SAN's experimental results

	ZINC	PATTERN	CLUSTER	MOLHIV	MOLPCBA
Model	MAE	% Acc	% Acc	% ROC-AUC	% AP
GCN	0.367 ± 0.011	71.892 ± 0.334	68.498 ± 0.976	76.06 ± 0.97	20.20 ± 0.24
GraphSage	0.398 ± 0.002	50.492 ± 0.001	63.844 ± 0.110	-	-
GatedGCN	0.282 ± 0.015	85.568 ± 0.088	73.840 ± 0.326	-	-
GatedGCN-PE	0.214 ± 0.013	86.508 ± 0.085	76.082 ± 0.196		
GIN	0.526 ± 0.013	85.387 ± 0.136	64.716 ± 1.553	75.58 ± 1.40	22.66 ± 0.28
PNA	0.142 ± 0.010	-	-	79.05 ± 1.32	28.38 ± 0.35
DGN	-	-	-	**79.70 ± 0.97**	**28.85 ± 0.30**
Attention-based					
GAT	0.384 ± 0.007	78.271 ± 0.186	70.587 ± 0.447	-	-
GT (sparse)	0.226 ± 0.014	84.808 ± 0.068	73.169 ± 0.662	-	-
GT (full)	0.598 ± 0.049	56.482 ± 3.549	27.121 ± 8.471	-	-
SAN	**0.139 ± 0.006**	**86.581 ± 0.037**	**76.691 ± 0.65**	77.85 ± 0.247	27.65 ± 0.42

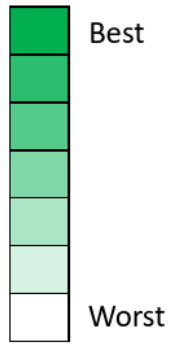


Figure 7: Comparing our tuned model on datasets from [15, 21], against GCN [25], GraphSage [18], GIN [39], GAT [37], GatedGCN [5], PNA [11], and DGN [4]. Means and uncertainties are derived from four runs with different seeds, except MolHIV which uses 10 runs with identical seed. The number of parameters is fixed to $\sim 500k$ for ZINC, PATTERN and CLUSTER.

Note: Not SOTA anymore.

Approach 3: Modify attention mechanism

- Another approach is to restrict/modify to which other nodes each node can attend to
  - E.g., only direct neighbors (Dwivedi and Bresson, 2020)
  - E.g., **Graphormer** uses shortest-path distances (among others) to modify attention weights (Ying et al., 2021)

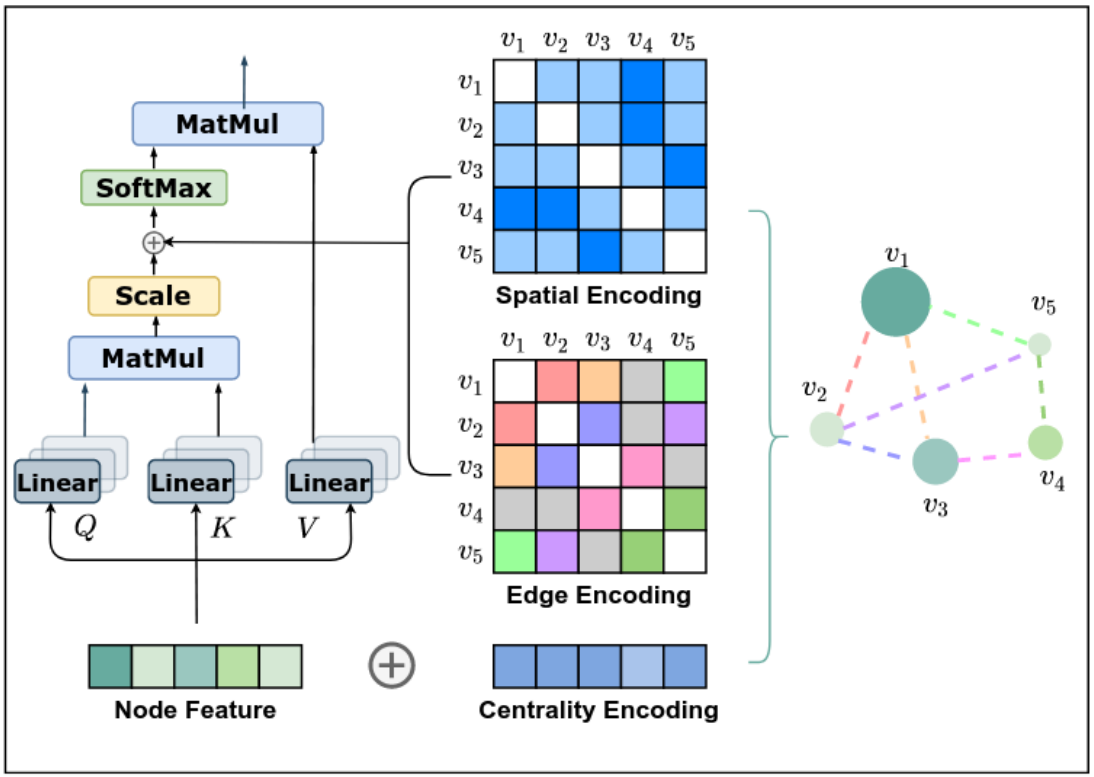


Figure 1: An illustration of our proposed centrality encoding, spatial encoding, and edge encoding in Graphormer.

Discussion

- Difficult to choose suitable approach
  - E.g., some evidence for each approach in study of Min et al., 2022
  - E.g., some evidence for approach 2 in study of Beaini et al. (2021)
- Scalability often a problem
  - Generally ok for many small graph scenarios
  - Very problematic for large graphs
- Many (!) graph learning methods have been and are still being proposed; e.g., survey of Ju et al. (2024)
  - Transformer-based, GNN-based, ...
  - No golden bullet
  - Extensive experimentation often required
  - More comparative studies / benchmarks needed
- Stay tuned!

Deep Learning

10 – Training Techniques

Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2025-1

From the tools and the models...

- The *tools*: backpropagation, optimizers, hyperparameter tuning
- The *models*: large, complex neural networks
  - Fully-connected layer with n inputs and m units: $O(nm)$ parameters
    - 10 dense layers, each 200 inputs/units $\rightarrow$ 400k parameters
    - 1 dense layer, $1M$ inputs, 200 units $\rightarrow$ 200M parameters
  - E.g., T5 text-to-text transformer
    (small: 60M, base: 220M, large: 770M, 3B, 11B)
  - E.g., ConvNext for CV
    (T: 60M, S: 82M, B: 121, L: 235M, XL: 391M)
  - E.g., DimeNet++-XL, a GNN for modelling atomic systems
    (Base: 1.8M, Large: 10.8M, XL: 240M)
- And now?
  - Which training data, which training objectives, which model?
    $\rightarrow$ this and, more comprehensively, related lectures (cf. slide 01/19)
  - How to train at scale?
    $\rightarrow$ not in this course

. . . to the art

The *art*: Which training data, which training objectives, which model?

- Sufficiently expressive (i.e., large) models needed for complex tasks
- Overfitting is a severe concern
  - Universal approximation theorem: with sufficiently many hidden neurons, FNN can perform arbitrarily well on the *training* set
- Limited labeled training data
  - Large labeled datasets for the task at hand generally not available
  - Supervision signal alone may be insufficient to achieve reasonable performance
- Design space is large
  - Experimentation is costly
  - Experience and domain knowledge is key
- Generally, **goals** include
  - Reduce overfitting, improve generalizability, reduce biases
  - Leverage additional data
  - Reduce (task-specific) costs such as model size, computational costs, amount of required supervision, . . .

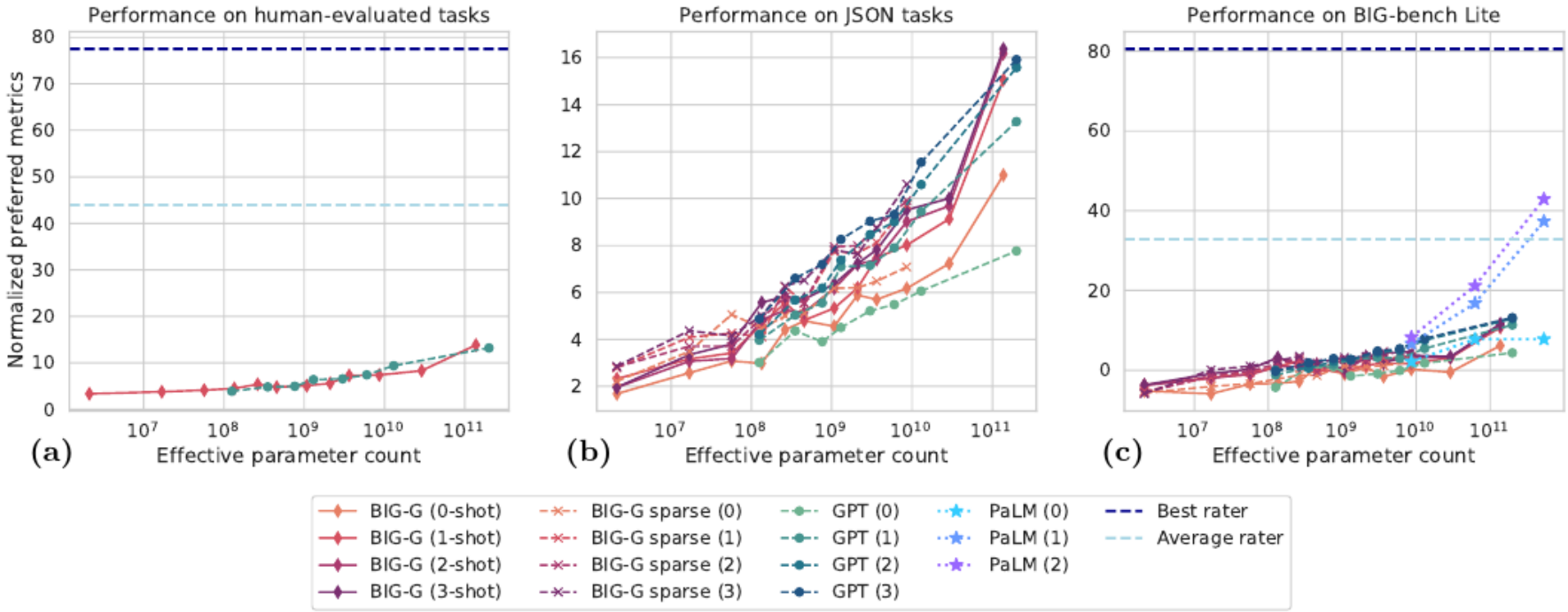


Figure 1: **Aggregate performance on BIG-bench improves with increasing model size and increasing shot count, but all models perform poorly in an absolute sense.** For sparse models, the x axis indicates the number of non-embedding parameters active during inference. In the legend, all parenthetical numbers indicate shot count. Each task has a unique *preferred metric* (Section 2.1). The aggregate performance is the average of each task's preferred metric normalized such that 0 represents poor performance and 100 very good performance (see Section 3.1 for a more detailed discussion). **(a)** Aggregate performance on programmatic and JSON tasks (JSON tasks are evaluated with one-shot prompting. See Section 2.1 for details on our task types), compared with human rater performance (see Section 2.3.2). BIG-G (see Section 2.3.1 for details on models) was evaluated on all tasks for which human performance is available (171 tasks), and GPT was evaluated on 146 tasks. Model performance improves with scale, but remains well below human rater baseline performance. **(b)** Aggregate performance on all JSON tasks with different shot values. BIG-G and BIG-G sparse were evaluated on all 161 JSON tasks; GPT was evaluated on 156 JSON tasks. Model performance is remarkably similar across model classes for a fixed number of parameters. **(c)** Aggregate performance over BIG-bench Lite, a curated subset of 24 tasks intended for lightweight evaluation. Results include the Pathways language models (PaLM) (Chowdhery et al., 2022).

Larger models may exhibit more bias

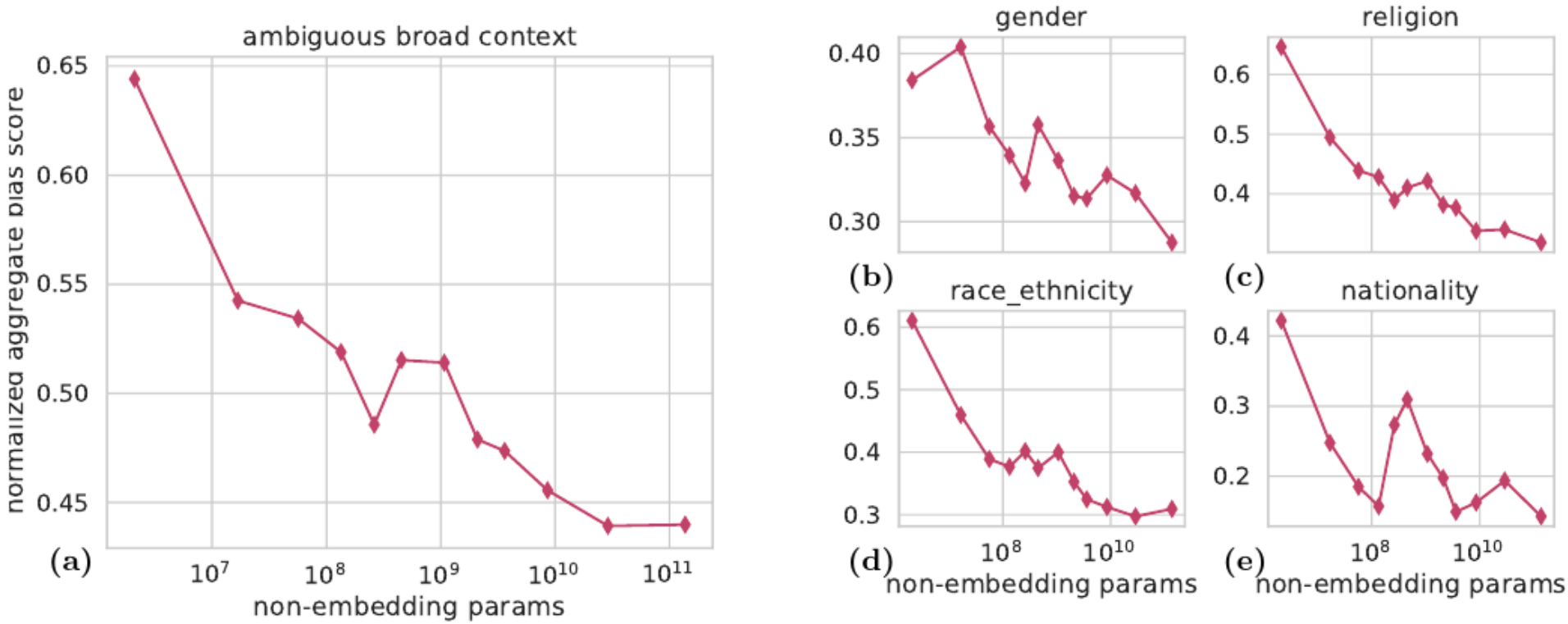


Figure 12: **Bias increases with scale for BIG-bench tasks with broad or ambiguous contexts.** Note that a higher score indicates better (less biased) performance on these tasks. **(a)** Aggregate performance across `bbq_lite`, `bias_from_probabilities`, `diverse_social_bias`, `gender_sensitivity_english`, `muslim_violence_bias`, and `unqover` decreases with model scale, indicating that larger models are more biased in the setting of broad or ambiguous contexts. **(b–e)** This trend appears robust across tasks probing bias with respect to gender, religion, race/ethnicity, and nationality (these constitute all the bias categories with more than one relevant BIG-bench task). Un-normalized individual task scores are shown in Supplementary Figure App.3.

Larger models require more data

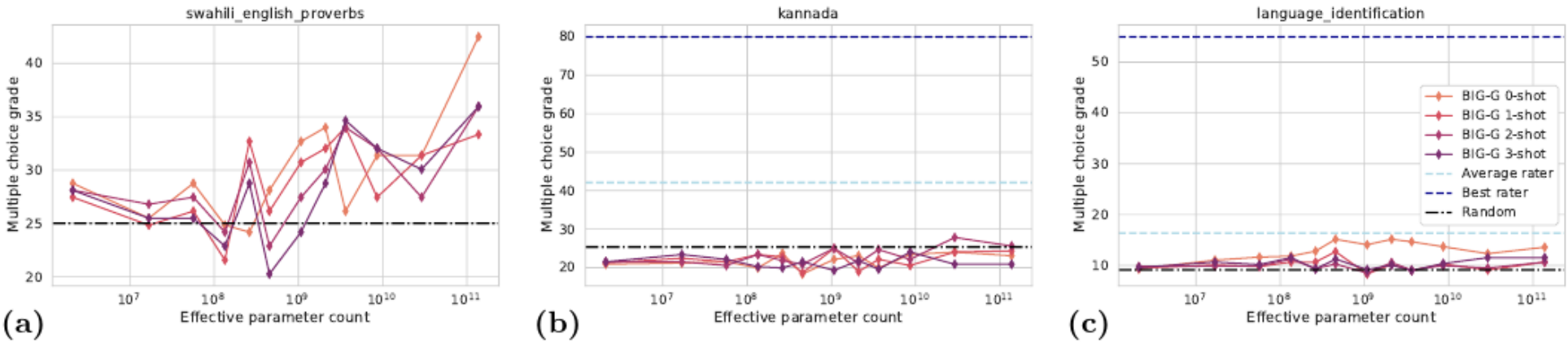


Figure 15: **Performance on low-resource language tasks is generally low.** **(a)** The `swahili_english_proverbs` task sees performance improvements with model scale, whereas performance on **(b)** `kannada` and **(c)** `language_identification` remains near chance.

Overview

Many strategies to avoid overfitting and/or improve the training process (for a particular task) exist.

- Standard strategies, e.g.,
  - Use parameter norm penalties or max-norm constraints
  - Use **early stopping** during training (i.e., don't train until convergence)
  - To some extent: use simpler models
  - Not discussed here (see ML course)
- Deep learning-specific strategies; e.g.,
  - Careful **architecture engineering** (as discussed)
    (e.g., sparse connections, residual connections, parameter tying, deep over wide, normalization, ...)
  - **Data augmentation**: create additional training data
  - **Pretraining and fine-tuning**: start from "suitable" model & adapt it for task at hand
  - **Prompting**: use one model for many tasks

Outline

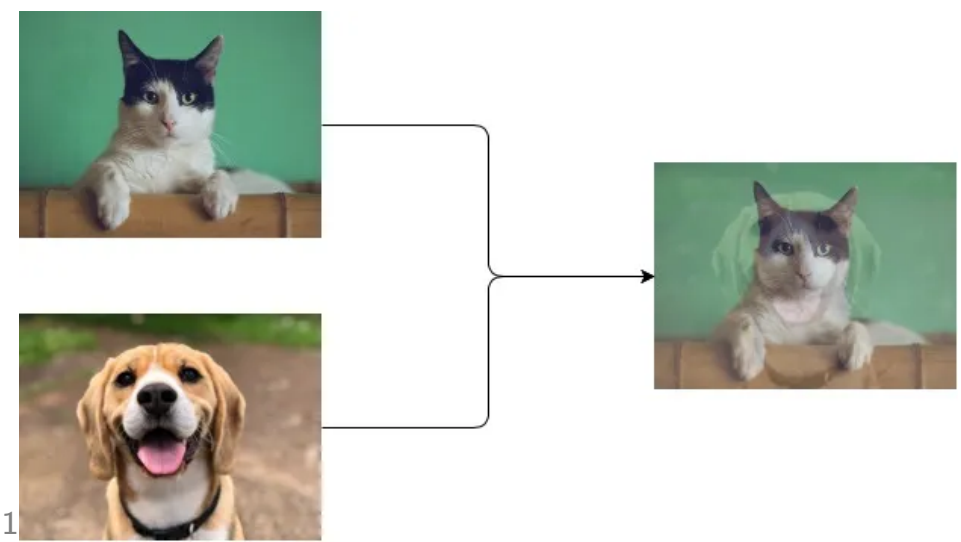
Class-preserving transformations

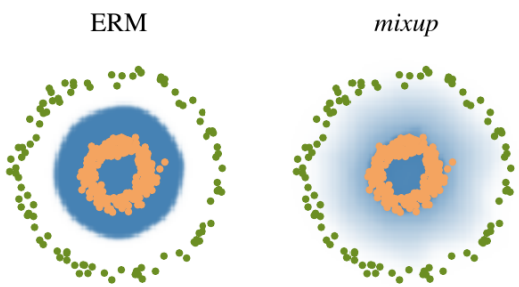
- **Data augmentation**: add generated data to training data
  - Aims to combat data scarcity
  - Often based on available training data
  - Fully automatic
  - When done well, can dramatically reduce the generalization error
- Example: **Class-preserving transformations**
  - Given labeled example $(\boldsymbol{x}, y)$, create augmented example $(\tilde{\boldsymbol{x}}, y)$ such that $\tilde{\boldsymbol{x}}$ and $\boldsymbol{x}$ (likely) belong to the same class
  - Increases invariance of model to such transformations
  - E.g., in CV: crop/rotate/shift/scale, color space, filters, synthesis, ...
  - E.g., in NLP: insertion/deletion/swaps, synonyms/paraphrasing, back translation, ...

Mixup augmentation

- Given two labeled examples $(\boldsymbol{x}_1, y_1)$ and $(\boldsymbol{x}_2, y_2)$, create **mixup example** $(\tilde{\boldsymbol{x}}, \tilde{y})$ such that
  - $\tilde{\boldsymbol{x}}$ "lies between" $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$
  - $\tilde{y}$ "lies between" y_1 and y_2
- Helps to "smoothen" decision boundaries
- Example: **linear mixup**
  - Inputs (e.g., grid data) and labels (e.g., one-hot encoded class) are real-valued vectors
  - Given **mixup ratio** $\lambda \in [0, 1]$, interpolate linearly:

$$\tilde{\boldsymbol{x}} = \lambda \boldsymbol{x}_1 + (1 - \lambda)\boldsymbol{x}_2$$
$$\tilde{\boldsymbol{y}} = \lambda \boldsymbol{y}_1 + (1 - \lambda)\boldsymbol{y}_2$$





(b) Effect of *mixup* ($\alpha = 1$) on a toy problem. Green: Class 0. Orange: Class 1. Blue shading indicates $p(y = 1|x)$.

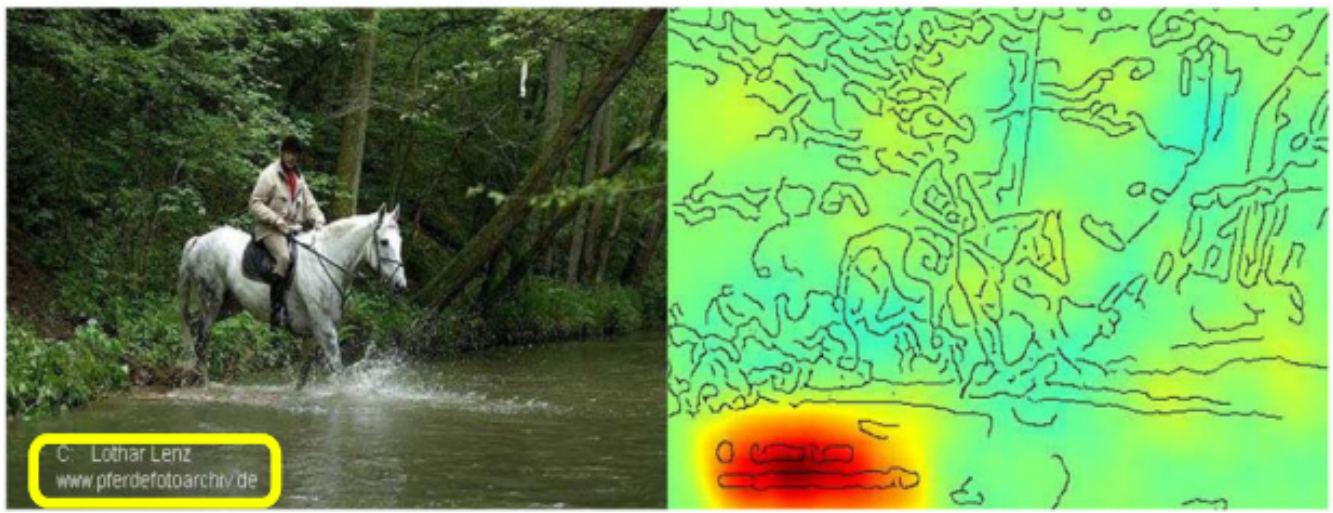
Masking

- **Masking** means to "hide" information during training
  1. E.g., parts of input data
  2. E.g., some (parts of) higher-level part representations
  3. E.g., some (parts of) model weights
- Model then less reliant on specific information; e.g.,
  1. Encourages model to not focus on a small part of the input & model learns to handle missing data & impact of spurious correlations may be reduced
  2. Encourages model to not rely on a small subset of features
  3. Encourages model to not rely on a small set of weights
- Often performed stochastically
- How to hide a part? E.g.,
  - Zero out part
  - Replace part by its mean
  - When part is categorical (e.g., a token in LLMs), set it to a special MASK category (e.g., a MASK token)
- When done well, can provably and empirically reduce generalization error

Spurious correlations

Spurious correlations means that model exploits features associated with but not causally related to output.

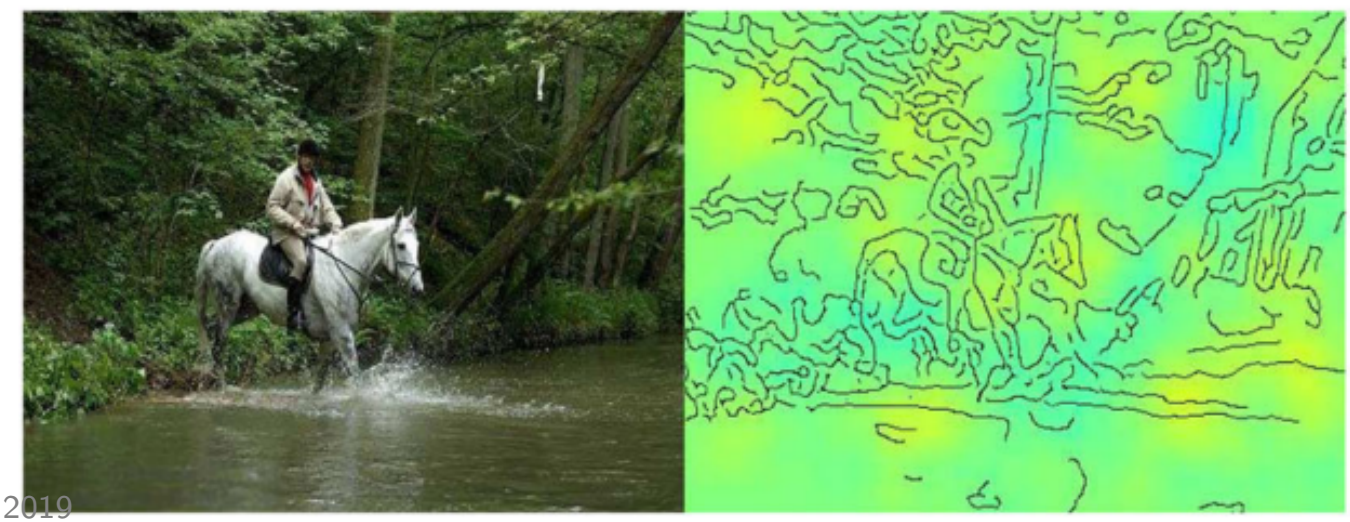
Horse-picture from Pascal VOC data set



Source tag present

↓

Classified as horse



No source tag present

↓

Not classified as horse

Dropout

- Dropout randomly zeros-out activations of inputs/hidden layers during training (e.g., 80% of input, 20% of hidden layer)
- Forces network to be accurate even in absence of some information
- Do the same during inference (multiple times) or use **weight scaling** heuristic (by inverse dropout probability)
- Can be seen as approximation of bagging (with parameter sharing)

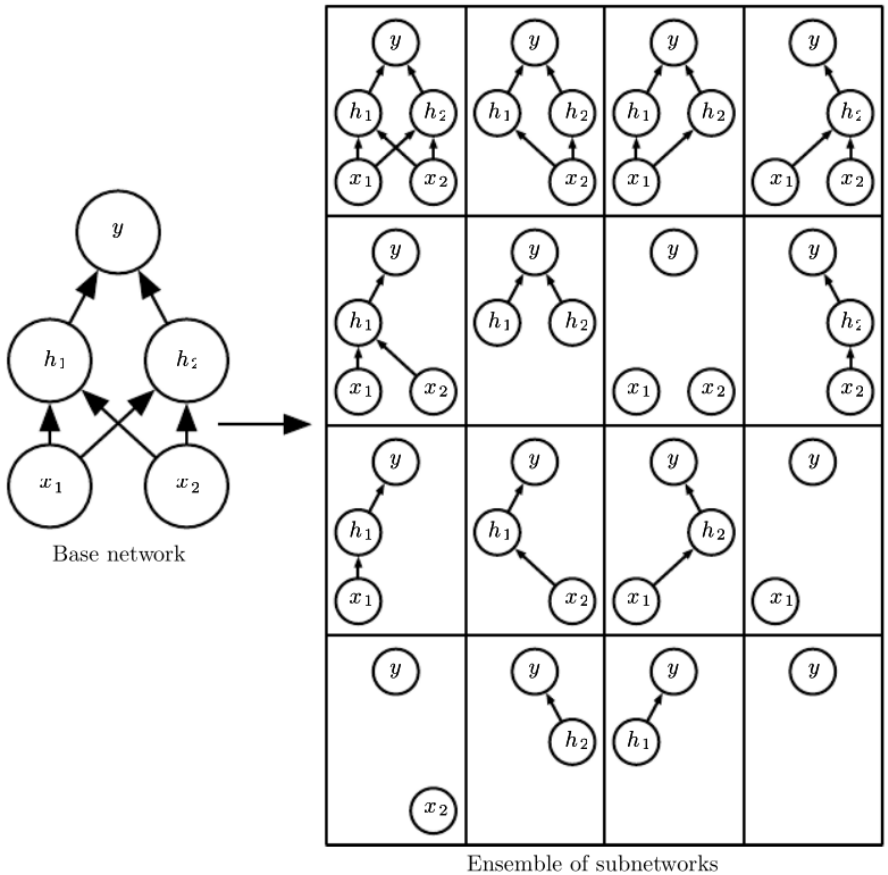


Figure 7.6: Dropout trains an ensemble consisting of all sub-networks that can be constructed by removing non-output units from an underlying base network. Here, we begin with a base network with two visible units and two hidden units. There are sixteen possible subsets of these four units. We show all sixteen subnetworks that may be formed by dropping out different subsets of units from the original network. In this small example, a large proportion of the resulting networks have no input units or no path connecting the input to the output. This problem becomes insignificant for networks with wider layers, where the probability of dropping all possible paths from inputs to outputs becomes smaller.

Noise injection

- **Noise injection** perturbs inputs/activations/weights/targets
  - To make network more robust to noise
  - To alleviate overfitting to noise in training data
- Example: **dropout** (as discussed)
- Example: **label smoothing**
  - Replace classification targets (e.g., $\{0, 1\}$) by smoothed versions (e.g., $\{\epsilon, 1 - \epsilon\}$)
  - Prevents extreme predictions
  - Alleviates impact of incorrect labels (**label noise**)

Adversarial training

- **Adversarial example**

Figure 7.8: A demonstration of adversarial example generation applied to GoogLeNet (Szegedy *et al.*, 2014a) on ImageNet. By adding an imperceptibly small vector whose elements are equal to the sign of the elements of the gradient of the cost function with respect to the input, we can change GoogLeNet's classification of the image. Reproduced with permission from Goodfellow *et al.* (2014b).

- **Adversarial training**: augment training data with adversarial examples
  - Increase model robustness
  - Defend against adversarial attacks
  - May be expensive

Outline

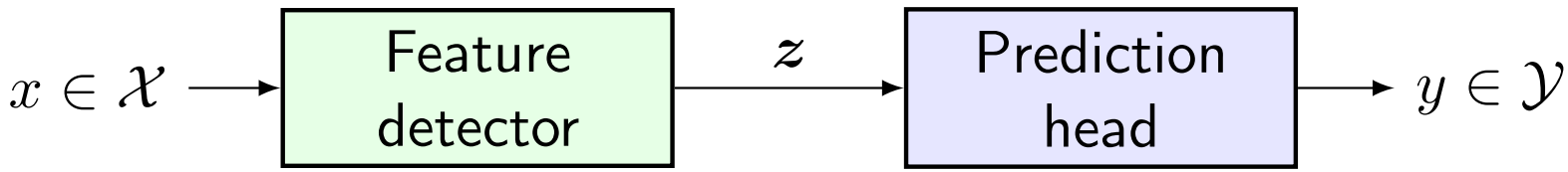
Pretrained models

Success of deep learning to a large extent based on collecting and leveraging large amounts of data (and compute) to obtain **pretrained models**.

- General approach
  - Train powerful model on *"available" data* and *carefully-chosen tasks* instead of actual the target task
  - Customize model for task at hand (e.g., fine-tuning, prompting)
  - **Often reduces task-specific cost and improves model performance**
- Many pretrained models available; e.g., for language (T5, RoBERTa, Mistral 7B) or vision (EfficientNet, Stable Diffusion) or tabular data (TabPFN) or time series (Chronos)
- Many private; trained models are a business asset
  - Provide API services + usage-based pricing model
  - GPT-4, Gemini, DALL·E 3, ...

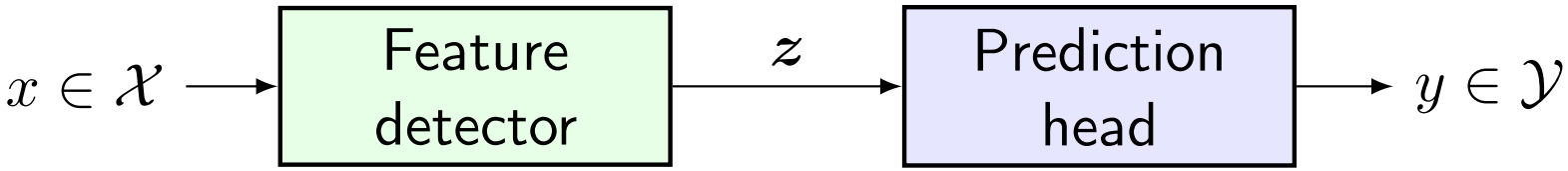
Feature detectors and prediction heads

- Consider a supervised learning task
  - Inputs $\mathcal{X}$, outputs $\mathcal{Y}$
  - "Small" training set $\mathcal{D} \subseteq \mathcal{X} \times \mathcal{Y}$
- Given a model architecture, let's divide it into a part that extracts features and part that predicts based on these features



- Division such that
  1. **Feature detector** (*backbone*, *base model*) performs most of the "heavy lifting" and provides suitable features for prediction
  2. **Prediction head** is simple (e.g., softmax layer or a small MLP)
- Cf. lecture 02-1 on embeddings

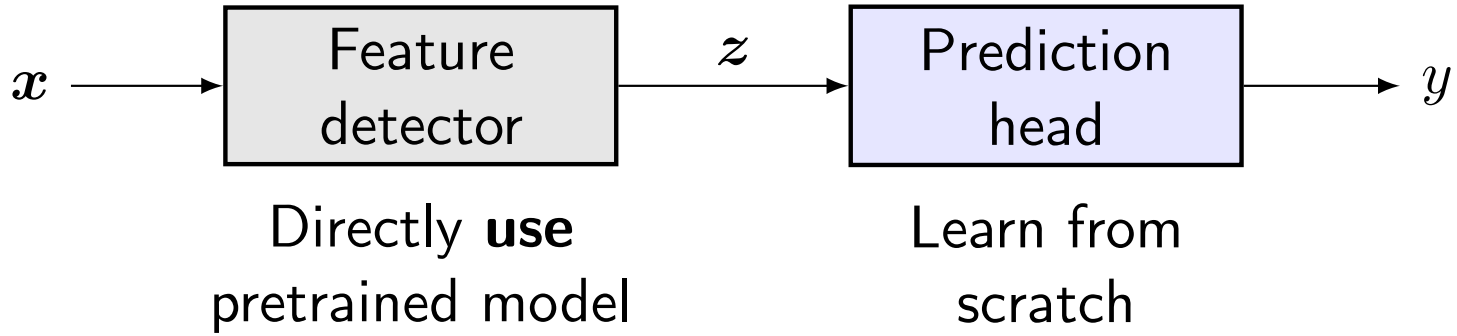
Key idea of pretraining



- Key idea of pretraining
  1. Learn feature detector using auxiliary data and/or tasks
     $\rightarrow$ **Pretrained model**
  2. Learn prediction head using task-specific data $\mathcal{D}$
     $\rightarrow$ Model for **downstream task**
- Why pretrain?
  1. Make use of auxiliary data
  2. Less labeled data and computational cost to train prediction head
  3. Can train different prediction heads for different (but related) tasks
  4. Powerful base models available for certain tasks; e.g.,
     - https://paperswithcode.com/
     - https://modelzoo.co/
     - https://www.tensorflow.org/hub
     - https://pytorch.org/hub/
     - https://huggingface.co/models
     - ...

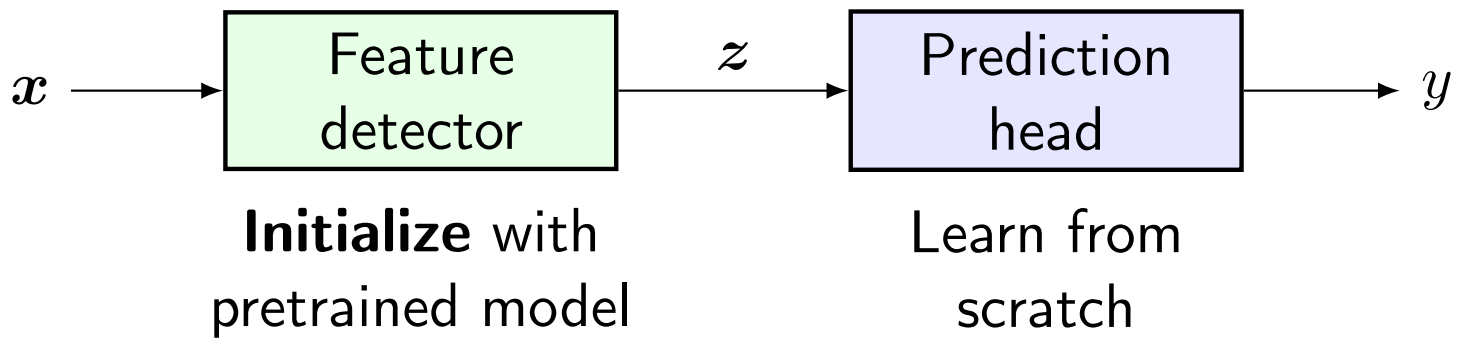
Fine-tuning

- **Without fine-tuning**: train only head in supervised fashion



  - Need labeled data
  - But less challenging when feature detector suitable: can get away with relatively simple predictor and less training data

- **With fine-tuning**: also retrain (parts of) feature detector



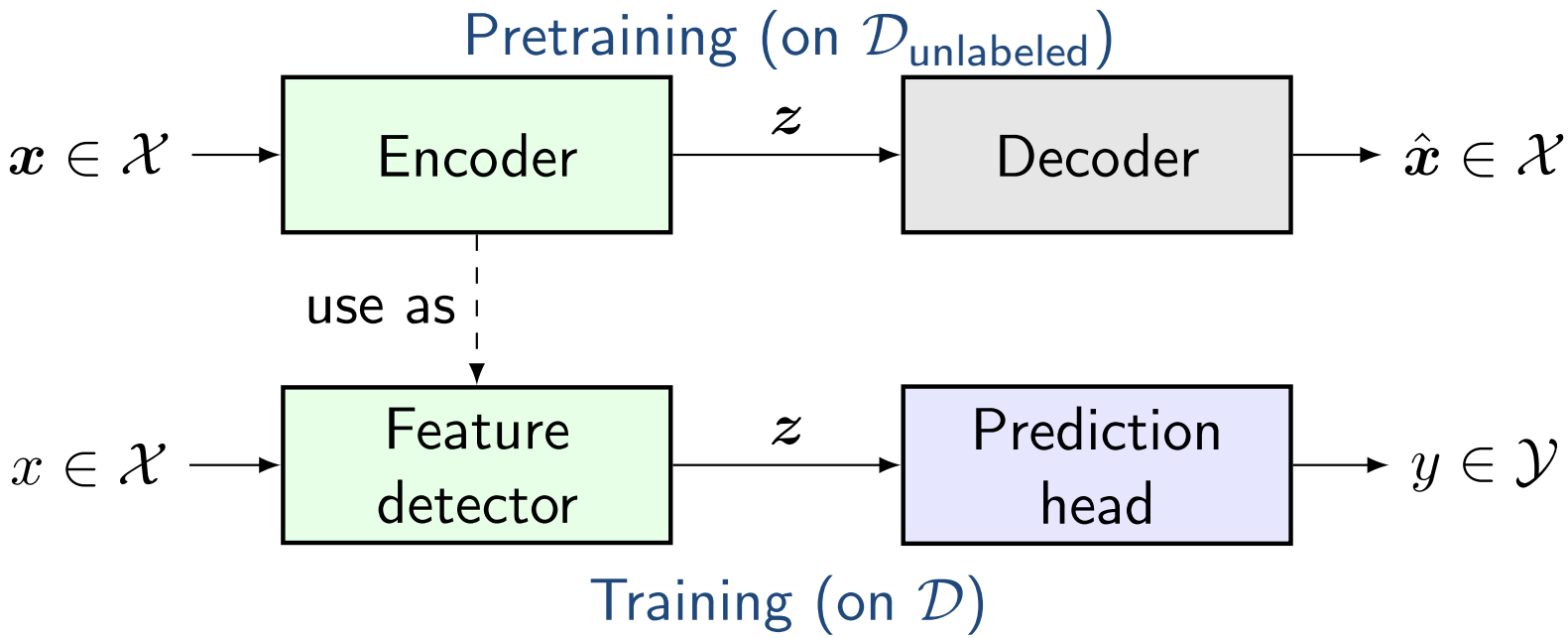
  - More expensive, but can improve feature detection for actual task
  - Higher risk of overfitting

Pretraining data

- **Labeled data**; e.g.,
  - Labeled images (ImageNet, COCO, Visual Genome)
  - Task collections (BIG-bench)
- **Paired data**
  - Images and text (PMD, LAION-5B)
  - Parallel multilingual text corpora (OPUS)
- **Unlabeled data**
  - Textual data such as web data (CommonCrawl, WebText) or books (BookCorpus, Wikipedia)
  - Social media data (Twitter, Reddit, Stack Overflow)
  - Source code (GitHub)
  - Multimodal data such as text interleaved with images
  - Knowledge graphs (Wikidata, DBpedia)
- **Synthetic data**
  - Generated to model reasonable data distributions (e.g., TabPFN, Rendered.ai, self play, digital twins)
- ...

Unsupervised pretraining

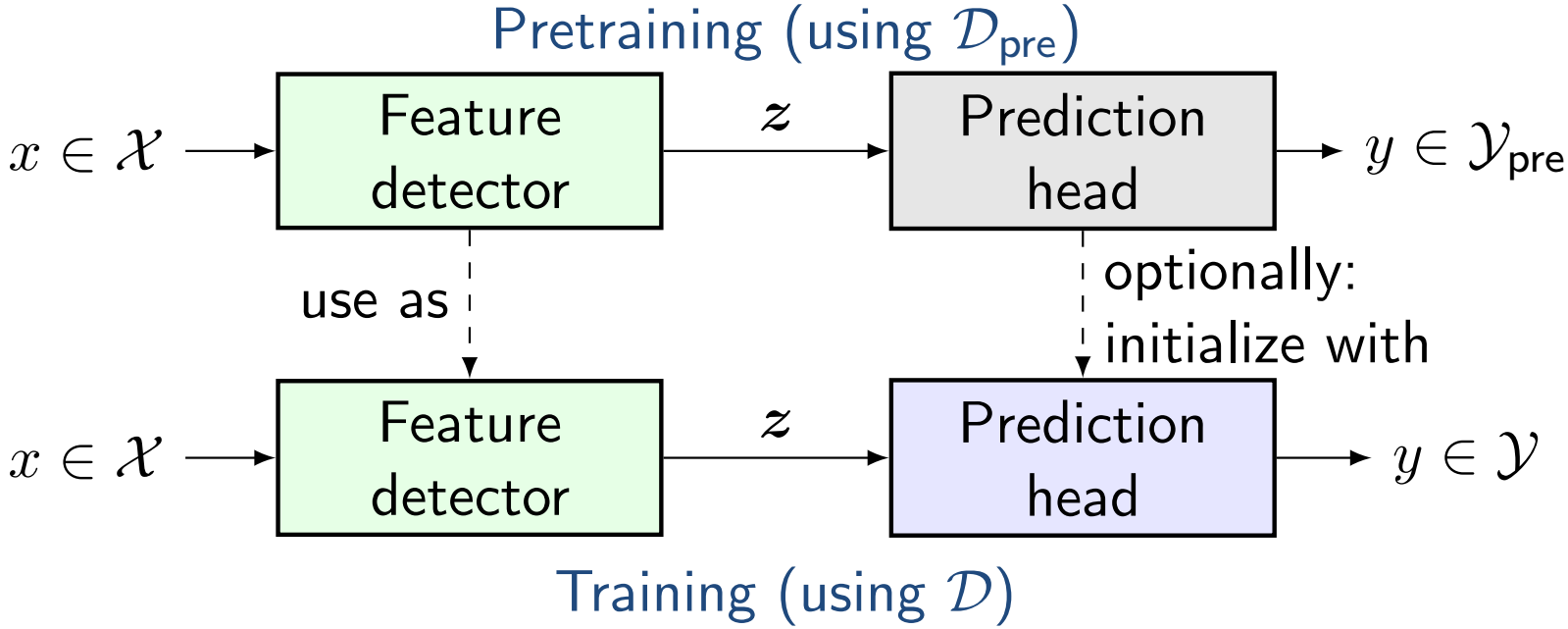
- **Unsupervised pretraining**
  - Based on (additional) unlabeled data $\mathcal{D}_{\text{unlabeled}} \subseteq \mathcal{X}$
  - Suitable when (useful) unlabeled data are readily available
- Example: unsupervised pretraining using an autoencoder
  - Encoder serves as pretrained feature detector
  - Decoder thrown away afterwards



- Hope that features are more useful for downstream tasks (e.g, feature detector may perform dimensionality reduction)

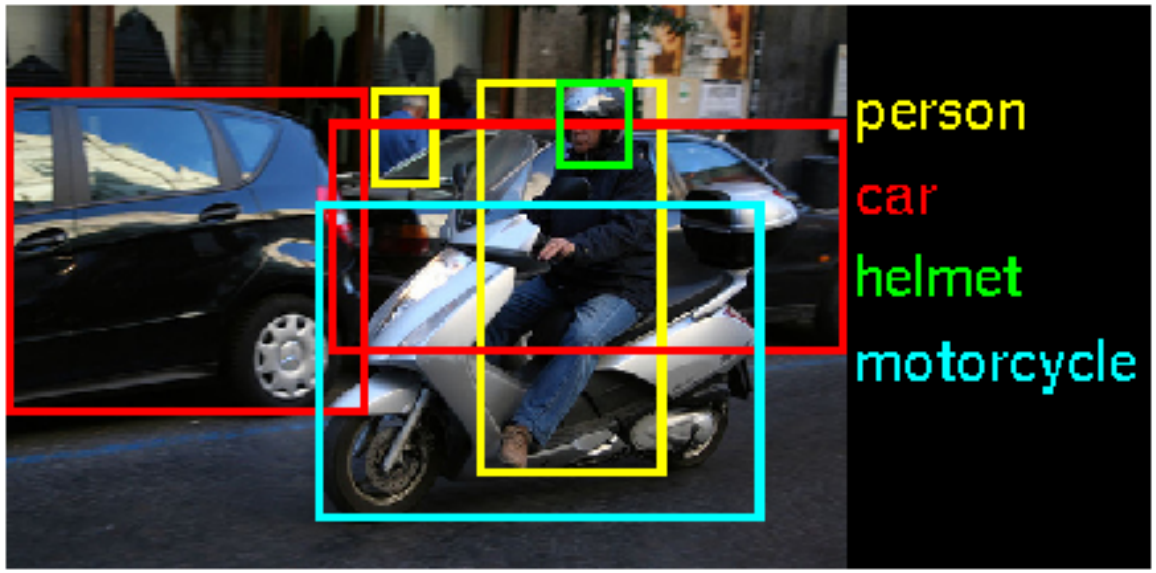
Supervised pretraining

- **Supervised pretraining**
  - Based on additional labeled data $\mathcal{D}_{\text{pre}} \subseteq \mathcal{X} \times \mathcal{Y}_{pre}$
  - Suitable when (useful) data for a different but related task is available



Example: CV models pretrained on ImageNet data

https://www.image-net.org/

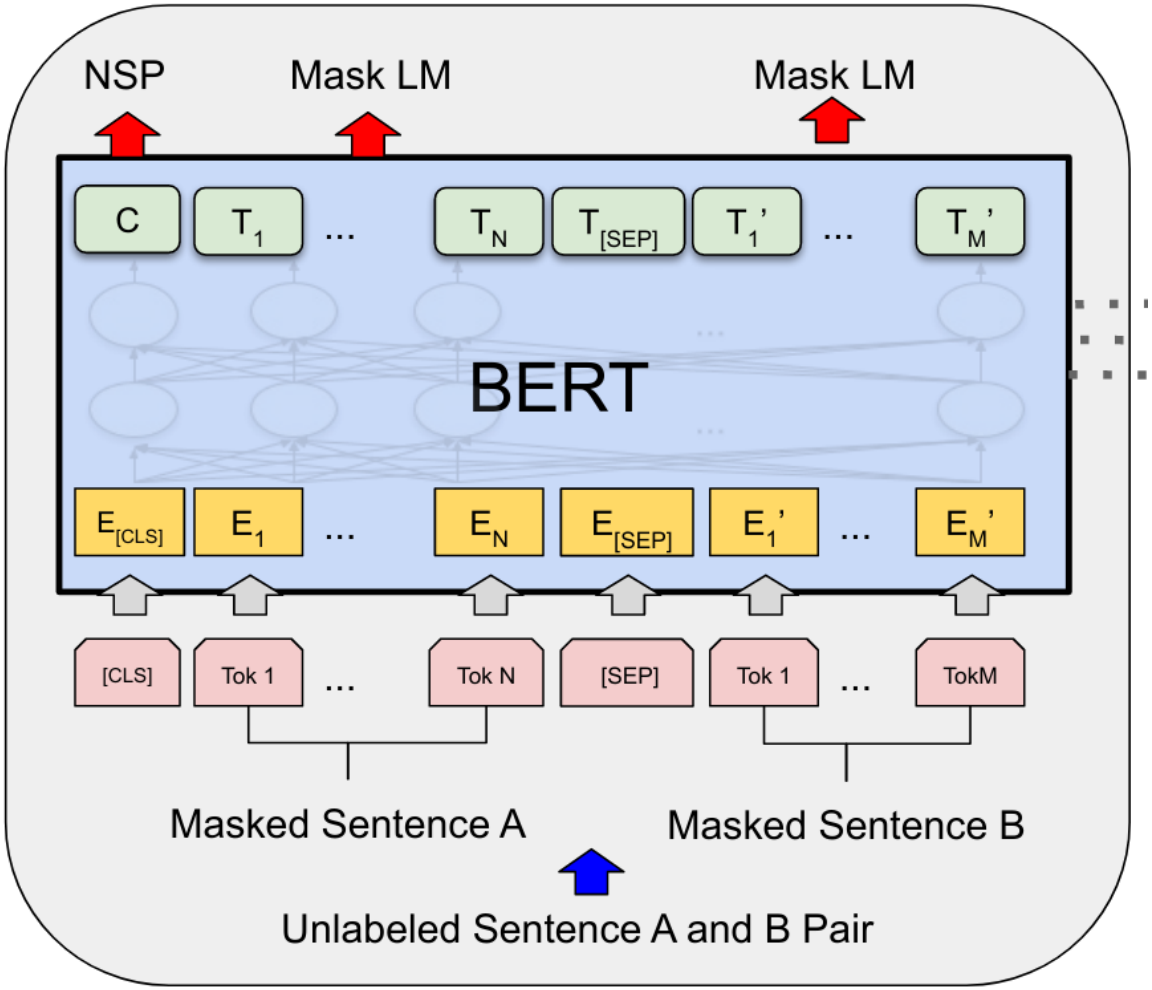


- 10,000,000 labeled images depicting 10,000+ object categories
- Pretrain using popular ILSVRC 2012-2017 image classification and localization task

Self-supervised pretraining

- **Self-supervised** methods use unlabeled data $\mathcal{D}_{\text{unlabeled}}$ to automatically build supervised prediction tasks
  - Cf. autoencoder
  - But: well-designed, domain-specific self-supervised tasks often more powerful $\rightarrow$ **key ingredient of state-of-the-art models**
- Generally, **mask-and-reconstruct** tasks very powerful
  - **Mask**: as discussed, hide part of the input (tokens in text, parts of images, cells in tables, vertices/edges in graphs)
  - **Reconstruct**: train model to model predict the masked part
- E.g., for NLP: mask-and-reconstruct tokens in input text
  - **Language model** (LM): provide prefix, predict next token(s)
  - **Masked language model** (MLM): provide input with some intermediate tokens masked out, predict the masked tokens
  - Models that perform well on these tasks are very powerful
    - Albert Einstein was born in ? (*factual knowledge*)
    - I like to eat ? (*common sense knowledge*)
    - All humans are mortal. Socrates is human. Therefore Socrates is ? (*reasoning*)

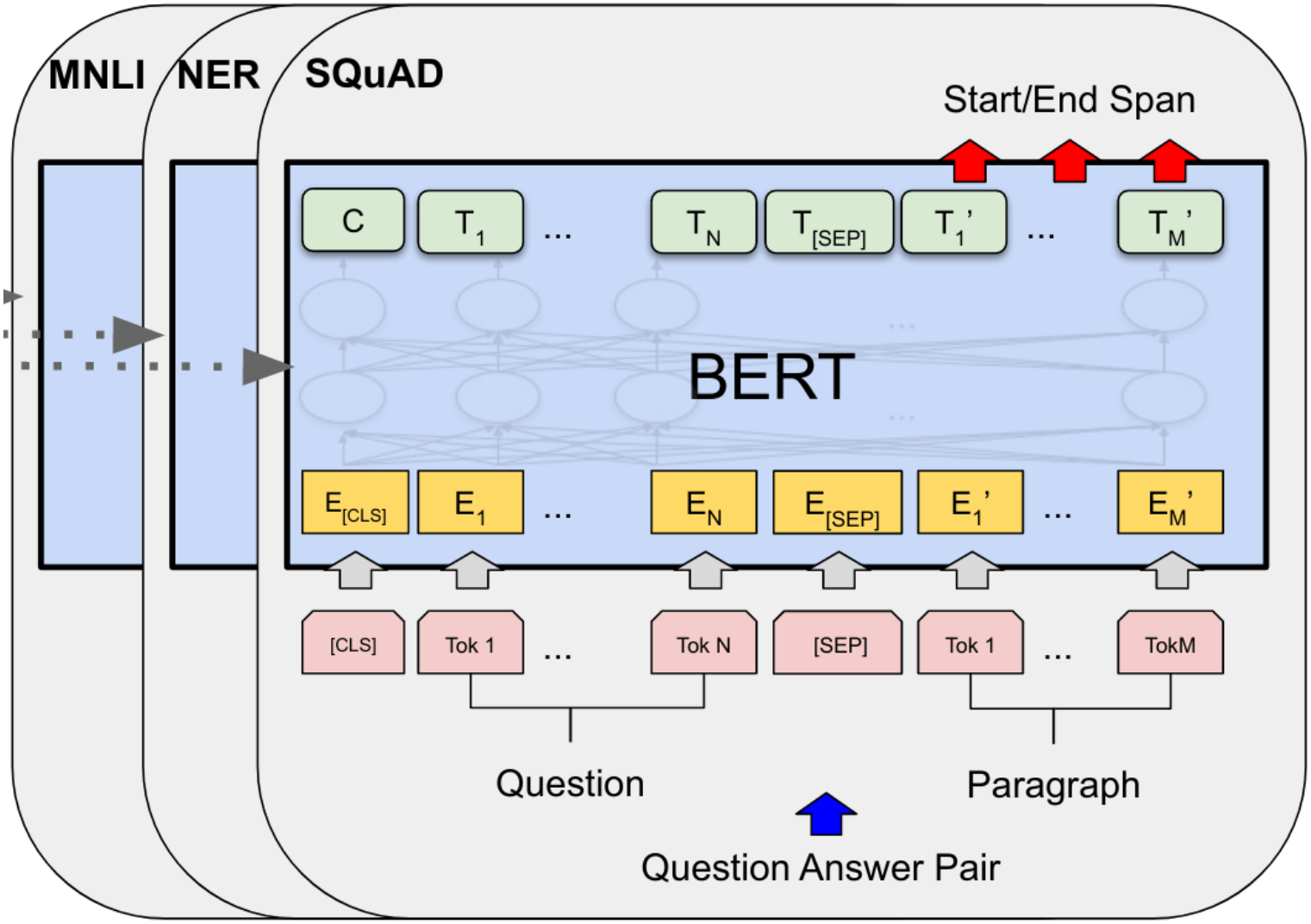
Example: BERT (pretraining)



(NSP = Next Sentence Prediction, another self-supervised task)

Devlin et al., 2018

Example: BERT (fine-tuning)



(SQuAD is reading comprehension benchmark)

Devlin et al., 2018

Example: GPT

- GPT models are pretrained using language modelling (left)
- Observe: downstream tasks expressed in "textual" form as well (right, cf. slide 38)

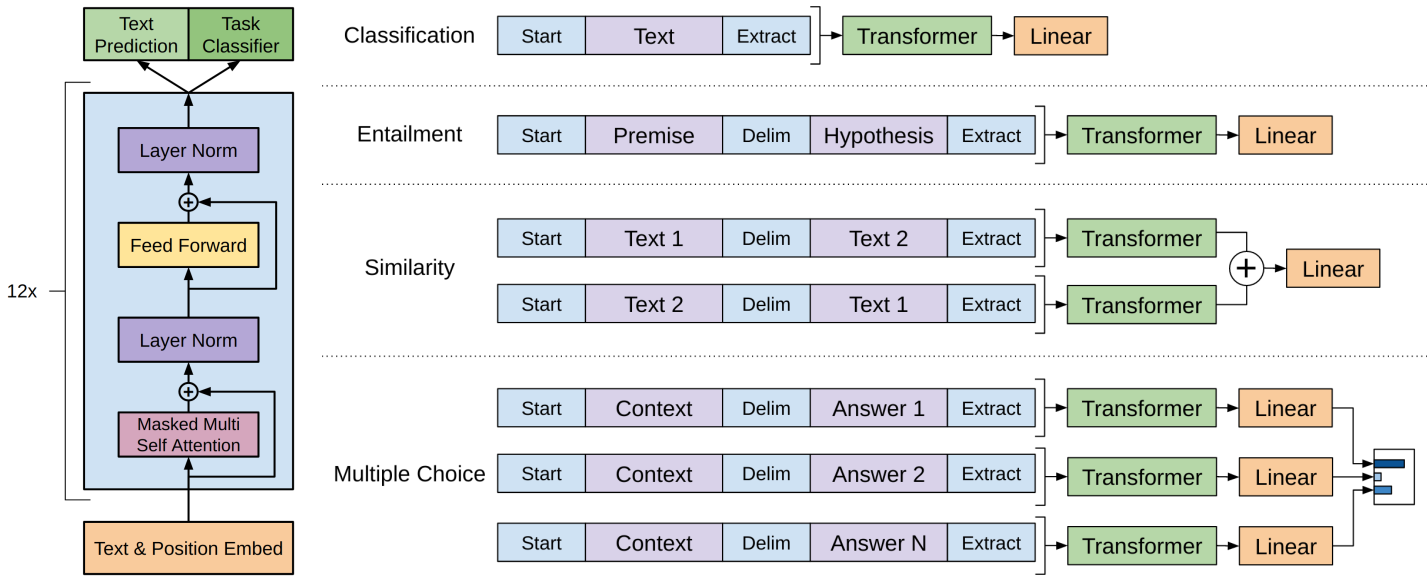
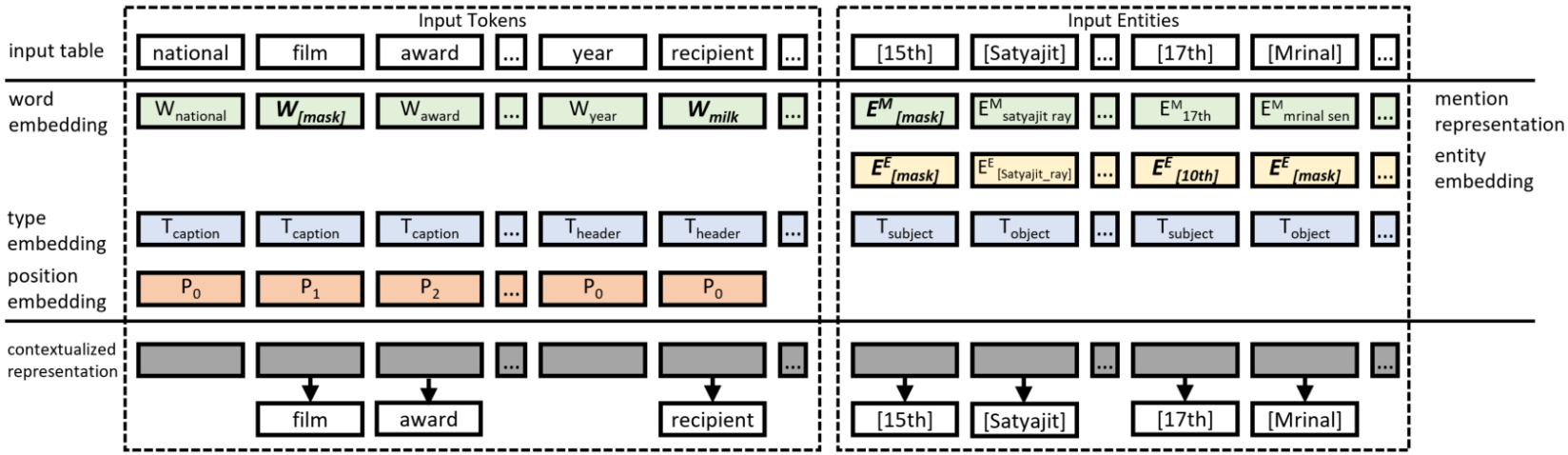


Figure 1: **(left)** Transformer architecture and training objectives used in this work. **(right)** Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

Example: TURL, Grover

TURL for tabular data



GROVER for molecular graphs

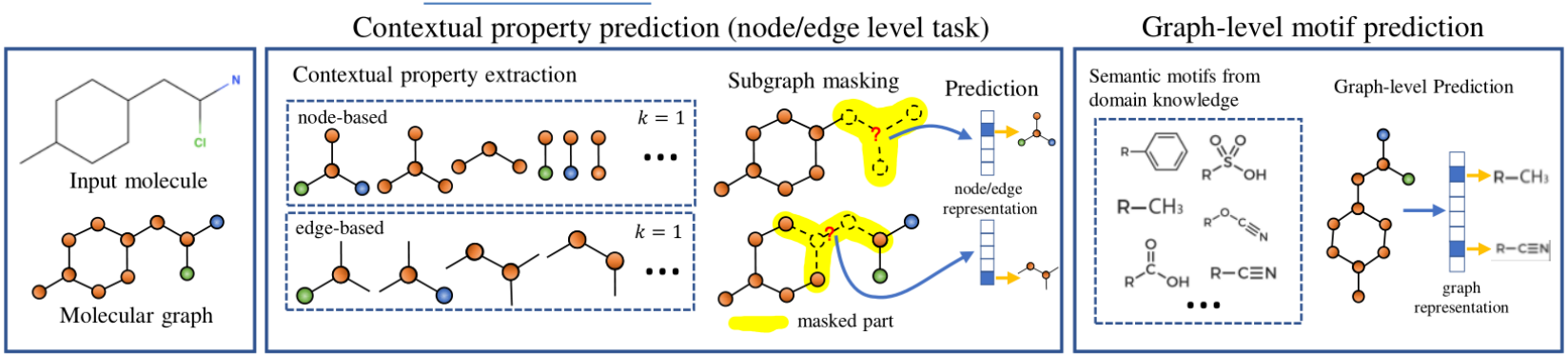


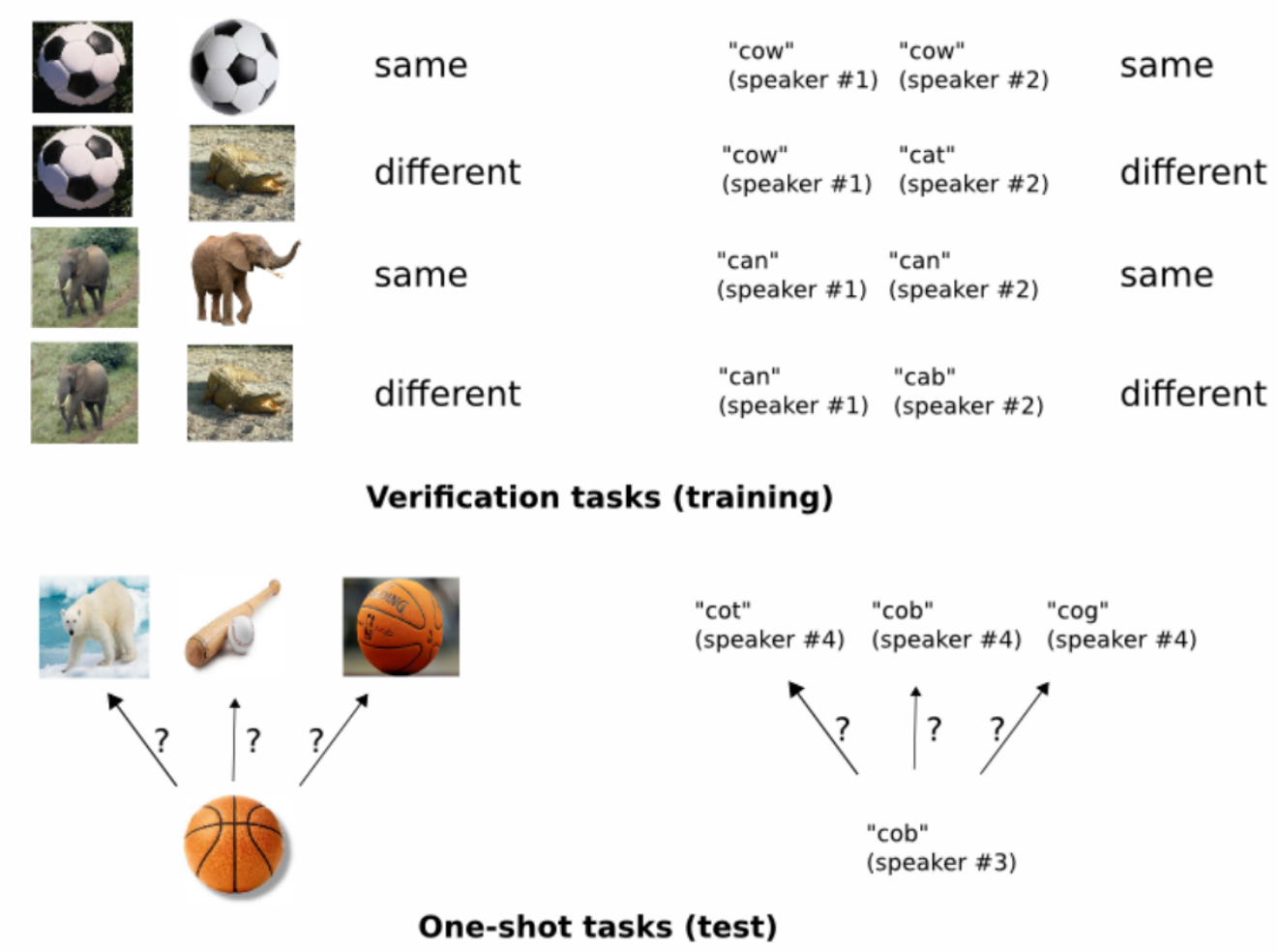
Figure 2: Overview of the designed self-supervised tasks of GROVER.

Deng et al., 2020; Rong et al., 2020

Contrastive Learning

- **Contrastive learning** is a form of self-supervised learning
  - Roughly: learn to *compare* two inputs
  - Trained to discriminate *similar* inputs (positives) and *dissimilar* inputs (negatives)
  - Choice of training examples very important
- Example: **Siamese neural networks** (also called: **bi-encoder**)
  - Use the *same* neural network on each input to obtain features
  - Followed by a simple comparison (e.g., cosine similarity)
    $\rightarrow$ Good models assign similar features to similar inputs
- Use contrastive model as feature detector (as before)
  - Drop the "comparison" part, keep the rest
  - Fine-tune afterwards
- Also: use directly (**metric learning**)
  - E.g., use for classification (as in a nearest-neighbor classifier)
  - E.g., use for information retrieval (as in nearest neighbor search)
  - E.g., use for **one-shot learning** (only one example for class) or **few-shot learning**

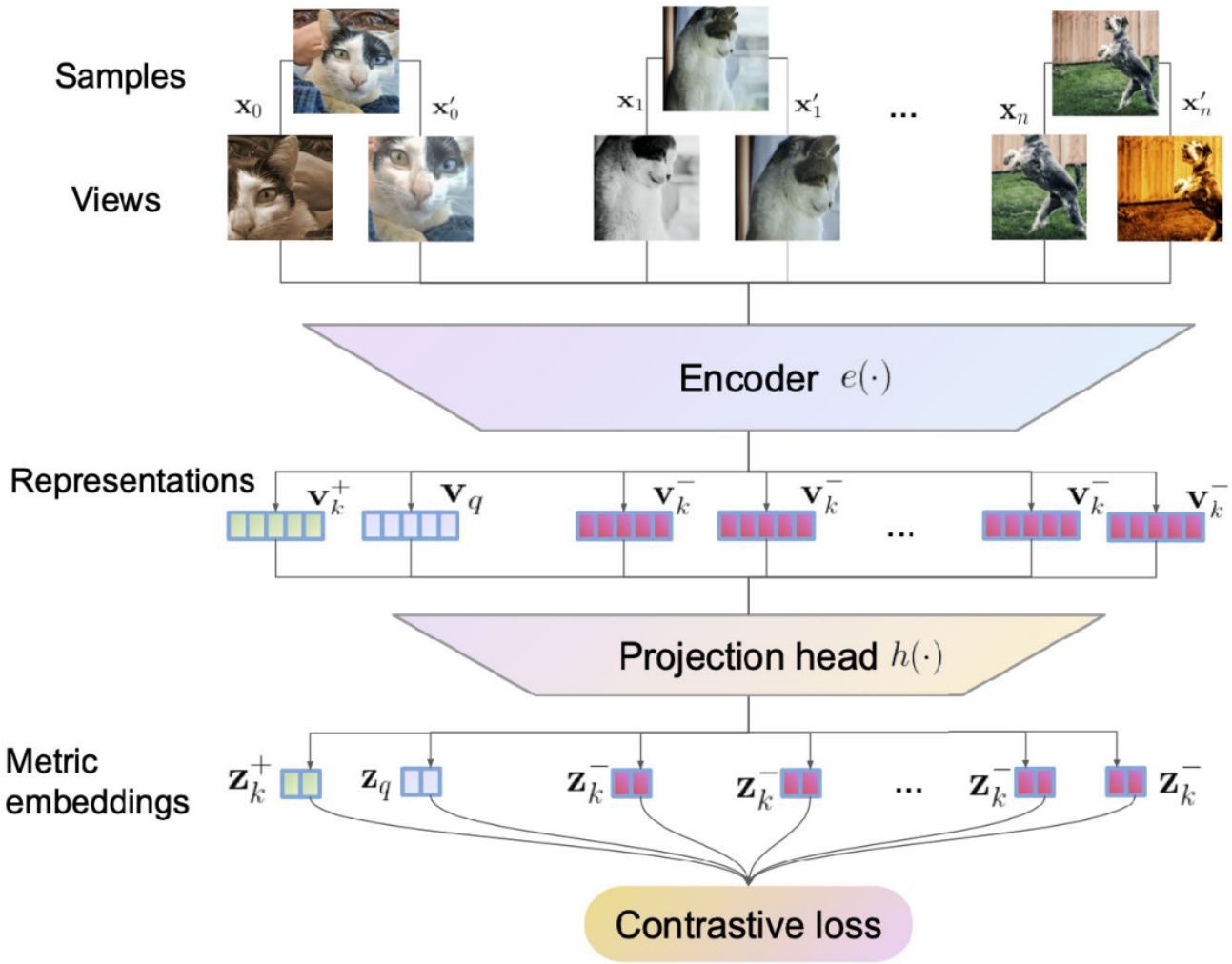
Example: One-shot learning with Siamese networks



Koch et al., 2015

Example: SimCLR

- Train without labels: each batch consists of two augmentations of each of N images
- Model trained to determine corresponding augmentations



Le-Khac et al., 2020

Fine-tuning and overparameterization

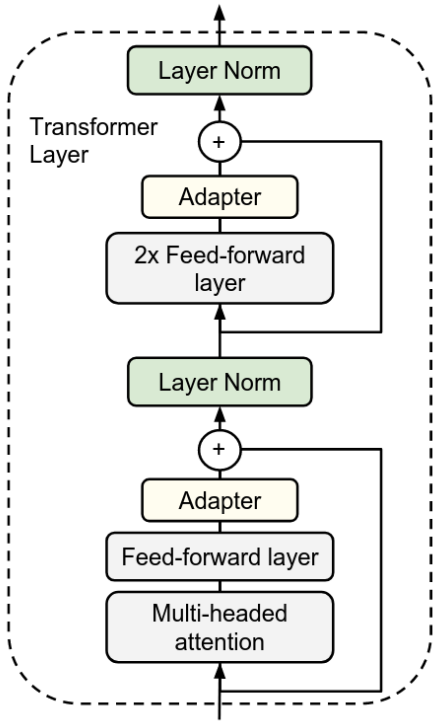
- Fine-tuning large models is costly
  - A separate model (updated base model + prediction head) is obtained for each task
  - Base models are large
- Can we reduce this cost? Goals:
  - Retain good performance (quality, fine-tuning/inference cost)
  - Allow task-specific fine-tuning (i.e., individually, not multi-task)
  - Small number of parameters per task
- Key insight: base models are often overparametrized
  - Lottery ticket hypothesis: "dense, randomly-initialized, feed-forward networks contain [small] subnetworks (*winning tickets*) that—when trained in isolation—reach test accuracy comparable to the original network in a similar number of iterations"
  - Consequently, fine-tuned models are also overparametrized
  - Can be exploited indirectly to **prune** trained networks
  - Can be exploited directly to reduce cost of fine-tuning

Low-cost fine-tuning

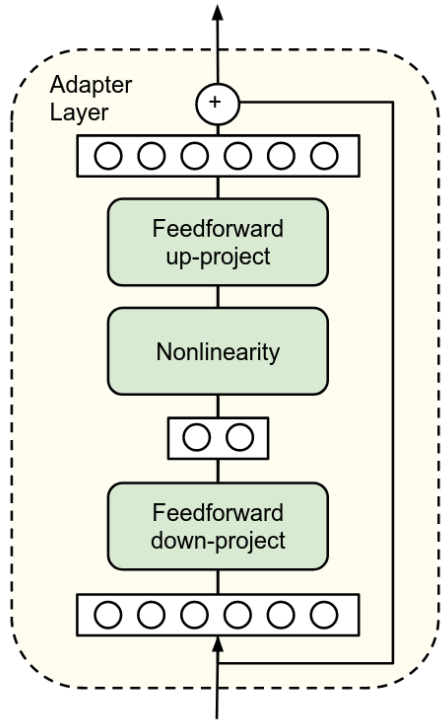
- **Adapter layers**
  - Before fine-tuning, inject (residual) adapter layers at certain places
  - Typically a bottleneck $\rightarrow$ few parameters
  - Initialized randomly, then fine-tuned (original weights remain frozen)
- **Low-rank adaptation (LoRA)**
  - Only learn *updates* to the model weights (plus all biases)
  - Update is low-rank matrix = down-, then up-project $\rightarrow$ bottleneck
  - Similar to adapters, but linear and at different location (e.g., on weights in multi-head attention layer)
  - Leads to a residual update of outputs for linear layers $\rightarrow$ can be computed in parallel (i.e., compute update of output while computing the base models' output)
- Simple baseline: **BitFit**
  - Only retrain bias weights

Adapter layers

Injected adapter layers



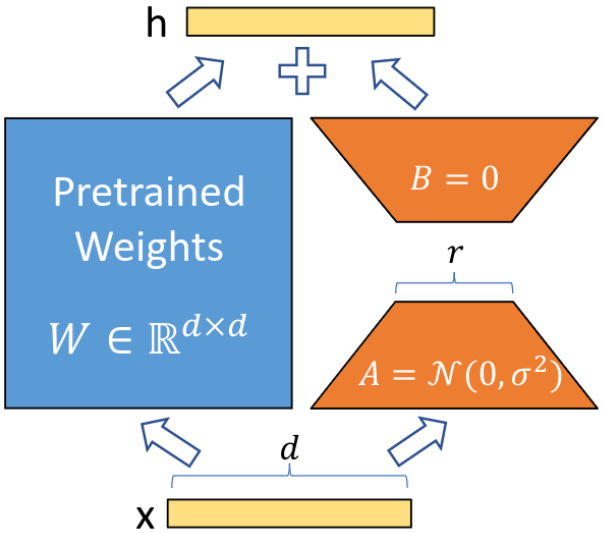
Adapter layer



	Total num params	Trained params / task	CoLA	SST	MRPC	STS-B	QQP	MNLI_m	MNLI_mm	QNLI	RTE	Total
BERT_LARGE	9.0×	100%	60.5	94.9	89.3	87.6	72.1	86.7	85.9	91.1	70.1	80.4
Adapters (8-256)	1.3×	3.6%	59.5	94.0	89.5	86.9	71.8	84.9	85.1	90.7	71.5	80.0
Adapters (64)	1.2×	2.1%	56.9	94.2	89.6	87.3	71.8	85.3	84.6	91.4	68.8	79.6

Table 1. Results on GLUE test sets scored using the GLUE evaluation server. MRPC and QQP are evaluated using F1 score. STS-B is evaluated using Spearman's correlation coefficient. CoLA is evaluated using Matthew's Correlation. The other tasks are evaluated using accuracy. Adapter tuning achieves comparable overall score (80.0) to full fine-tuning (80.4) using 1.3× parameters in total, compared to 9×. Fixing the adapter size to 64 leads to a slightly decreased overall score of 79.6 and slightly smaller model.

LoRA



$\boldsymbol{A}$ and $\boldsymbol{B}$ are learned during fine-tuning

Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
RoB_{base} (FT)*	125.0M	**87.6**	94.8	90.2	**63.6**	92.8	**91.9**	78.7	91.2	86.4
RoB_{base} (BitFit)*	0.1M	84.7	93.7	**92.7**	62.0	91.8	84.0	81.5	90.8	85.2
RoB_{base} (Adpt^{D})*	0.3M	$87.1_{\pm.0}$	$94.2_{\pm.1}$	$88.5_{\pm1.1}$	$60.8_{\pm.4}$	$93.1_{\pm.1}$	$90.2_{\pm.0}$	$71.5_{\pm2.7}$	$89.7_{\pm.3}$	84.4
RoB_{base} (Adpt^{D})*	0.9M	$87.3_{\pm.1}$	$94.7_{\pm.3}$	$88.4_{\pm.1}$	$62.6_{\pm.9}$	$93.0_{\pm.2}$	$90.6_{\pm.0}$	$75.9_{\pm2.2}$	$90.3_{\pm.1}$	85.4
RoB_{base} (LoRA)	0.3M	$87.5_{\pm.3}$	$\mathbf{95.1}_{\pm.2}$	$89.7_{\pm.7}$	$63.4_{\pm1.2}$	$\mathbf{93.3}_{\pm.3}$	$90.8_{\pm.1}$	$\mathbf{86.6}_{\pm.7}$	$\mathbf{91.5}_{\pm.2}$	**87.2**
$\text{RoB}_{\text{large}}$ (FT)*	355.0M	90.2	**96.4**	**90.9**	68.0	94.7	**92.2**	86.6	92.4	88.9
$\text{RoB}_{\text{large}}$ (LoRA)	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.9}_{\pm1.2}$	$\mathbf{68.2}_{\pm1.9}$	$\mathbf{94.9}_{\pm.3}$	$91.6_{\pm.1}$	$\mathbf{87.4}_{\pm2.5}$	$\mathbf{92.6}_{\pm.2}$	**89.0**
$\text{RoB}_{\text{large}}$ (Adpt^{P})†	3.0M	$90.2_{\pm.3}$	$96.1_{\pm.3}$	$90.2_{\pm.7}$	$\mathbf{68.3}_{\pm1.0}$	$\mathbf{94.8}_{\pm.2}$	$\mathbf{91.9}_{\pm.1}$	$83.8_{\pm2.9}$	$92.1_{\pm.7}$	88.4
$\text{RoB}_{\text{large}}$ (Adpt^{P})†	0.8M	$\mathbf{90.5}_{\pm.3}$	$\mathbf{96.6}_{\pm.2}$	$89.7_{\pm1.2}$	$67.8_{\pm2.5}$	$\mathbf{94.8}_{\pm.3}$	$91.7_{\pm.2}$	$80.1_{\pm2.9}$	$91.9_{\pm.4}$	87.9
$\text{RoB}_{\text{large}}$ (Adpt^{H})†	6.0M	$89.9_{\pm.5}$	$96.2_{\pm.3}$	$88.7_{\pm2.9}$	$66.5_{\pm4.4}$	$94.7_{\pm.2}$	$92.1_{\pm.1}$	$83.4_{\pm1.1}$	$91.0_{\pm1.7}$	87.8
$\text{RoB}_{\text{large}}$ (Adpt^{H})†	0.8M	$90.3_{\pm.3}$	$96.3_{\pm.5}$	$87.7_{\pm1.7}$	$66.3_{\pm2.0}$	$94.7_{\pm.2}$	$91.5_{\pm.1}$	$72.9_{\pm2.9}$	$91.5_{\pm.5}$	86.4
$\text{RoB}_{\text{large}}$ (LoRA)†	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.2}_{\pm1.0}$	$68.2_{\pm1.9}$	$\mathbf{94.8}_{\pm.3}$	$91.6_{\pm.2}$	$\mathbf{85.2}_{\pm1.1}$	$\mathbf{92.3}_{\pm.5}$	**88.6**
DeB_{XXL} (FT)*	1500.0M	91.8	**97.2**	92.0	72.0	**96.0**	92.7	93.9	92.9	91.1
DeB_{XXL} (LoRA)	4.7M	$\mathbf{91.9}_{\pm.2}$	$96.9_{\pm.2}$	$\mathbf{92.6}_{\pm.6}$	$\mathbf{72.4}_{\pm1.1}$	$\mathbf{96.0}_{\pm.1}$	$\mathbf{92.9}_{\pm.1}$	$\mathbf{94.9}_{\pm.4}$	$\mathbf{93.0}_{\pm.2}$	**91.3**

Outline

Prompting

- Alternative approach: **prompting**
  - Use a generative **foundation model** = a powerful general-purpose model trained on large amounts of data and suitable for a wide variety of tasks
  - No fine-tuning
- Example: GPT-3 and successors
  - Model input: description of data/task in textual form (**prompt**)
  - Model output: answer in textual form

Prompt

```
Classify the sentiment in these tweets:

1. "I can't stand homework"
2. "This sucks. I'm bored 😠"
3. "I can't wait for Halloween!!!"
4. "My cat is adorable ❤️❤️"
5. "I hate chocolate"

Tweet sentiment ratings:
```

Sample response

```
1. Negative
2. Negative
3. Positive
4. Positive
5. Negative
```

Prompt

```
##### Translate this function  from Python into Haskell
### Python

    def predict_proba(X: Iterable[str]):
        return np.array([predict_one_probas(tweet) for tweet in X])

### Haskell
```

Sample response

```
predict_proba :: [String] -> [Probability]
predict_proba = map predict_one_probas
```

OpenAI examples

Prompting (multi-modal)

Example: Kosmos-1 multimodal large language model

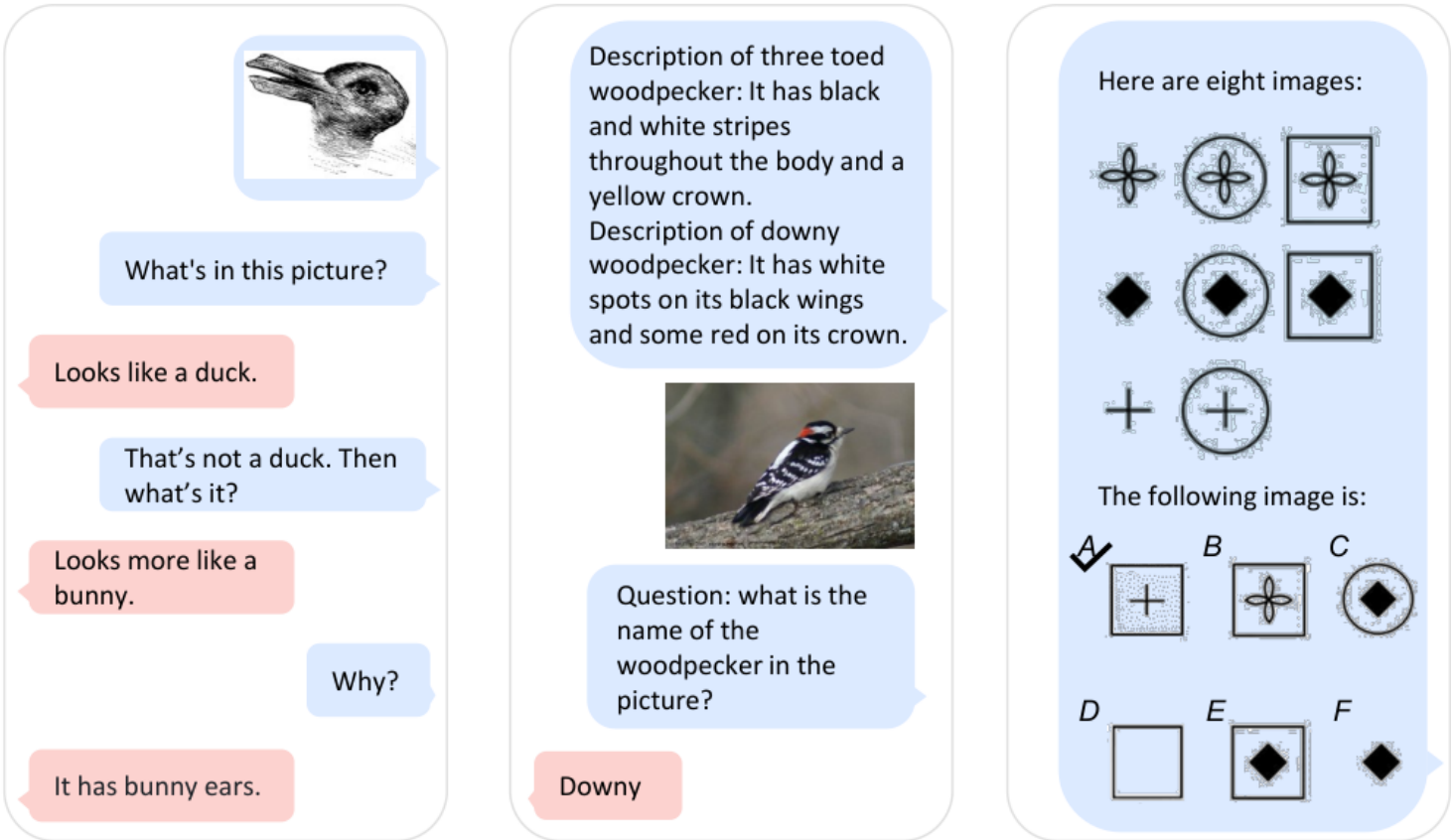


Figure 1: KOSMOS-1 is a multimodal large language model (MLLM) that is capable of perceiving multimodal input, following instructions, and performing in-context learning for not only language tasks but also multimodal tasks. In this work, we align vision with large language models (LLMs), advancing the trend of going from LLMs to MLLMs.

Huang et al. (2023)

Prompt/answer engineering

- Performance can heavily depend on how prompts and answers are written $\rightarrow$ **prompt/answer engineering**
- General approach:

Name	Notation	Example	Description
Input	$\boldsymbol{x}$	I love this movie.	One or multiple texts
Output	$\boldsymbol{y}$	++ (very positive)	Output label or text
Prompting Function	$f_{\text{prompt}}(\boldsymbol{x})$	`[X]` Overall, it was a `[Z]` movie.	A function that converts the input into a specific form by inserting the input $\boldsymbol{x}$ and adding a slot `[Z]` where answer $\boldsymbol{z}$ may be filled later.
Prompt	$\boldsymbol{x}'$	I love this movie. Overall, it was a `[Z]` movie.	A text where `[X]` is instantiated by input $\boldsymbol{x}$ but answer slot `[Z]` is not.
Filled Prompt	$f_{\text{fill}}(\boldsymbol{x}', \boldsymbol{z})$	I love this movie. Overall, it was a bad movie.	A prompt where slot `[Z]` is filled with any answer.
Answered Prompt	$f_{\text{fill}}(\boldsymbol{x}', \boldsymbol{z}^*)$	I love this movie. Overall, it was a good movie.	A prompt where slot `[Z]` is filled with a true answer.
Answer	$\boldsymbol{z}$	"good", "fantastic", "boring"	A token, phrase, or sentence that fills `[Z]`

Table 2: Terminology and notation of prompting methods. $\boldsymbol{z}^*$ represents answers that correspond to true output $\boldsymbol{y}^*$.

- More in survey of Lie et al. (2021)

Examples of prompt engineering

- **In-context learning** (see next slide)
  - Provide examples and/or background information for task with prompt
  - Typically just a few examples → a form of few-shot learning
  - Choice of examples matter → selection can be key
  - More in survey of Dong et al. (2023)
- **Chain-of-thought prompting**
  - Decompose multi-step problems into individual steps
  - Provide individual steps with demonstrations
  - Generate steps along with actual answer for new inputs
  - More in survey of Chu et al. (2023)
  - Modern "reasoning models" do this directly (via a *scratchpad*)
- Much more in: IE 868 Large Language Models and Agents

Poor English input: I eated the purple berries.
Good English output: I ate the purple berries.
Poor English input: Thank you for picking me as your designer. I'd appreciate it.
Good English output: Thank you for choosing me as your designer. I appreciate it.
Poor English input: The mentioned changes have done. or I did the alteration that you requested. or I changed things you wanted and did the modifications.
Good English output: The requested changes have been made. or I made the alteration that you requested. or I changed things you wanted and made the modifications.
Poor English input: I'd be more than happy to work with you in another project.
Good English output: I'd be more than happy to work with you on another project.

Poor English input: Please provide me with a short brief of the design you're looking for and that'd be nice if you could share some examples or project you did before.
Good English output: Please provide me with a brief description of the design you're looking for and that would be nice if you could share some examples or projects you have done before.

Poor English input: The patient was died.
Good English output: The patient died.

Poor English input: We think that Leslie likes ourselves.
Good English output: We think that Leslie likes us.

Poor English input: Janet broke Bill on the finger.
Good English output: Janet broke Bill's finger.

Figure 3.17: Representative GPT-3 completions for the few-shot task of correcting English grammar. Boldface is GPT-3's completions, plain text is human prompts. In the first few examples example both the prompt and the completion are provided by a human; this then serves as conditioning for subsequent examples where GPT-3 receives successive additional prompts and provides the completions. Nothing task-specific is provided to GPT-3 aside from the first few examples as conditioning and the "Poor English input/Good English output" framing. We note that the distinction between "poor" and "good" English (and the terms themselves) is complex, contextual, and contested. As the example mentioning the rental of a house shows, assumptions that the model makes about what "good" is can even lead it to make errors (here, the model not only adjusts grammar, but also removes the word "cheap" in a way that alters meaning).

Example: chain-of-thought prompting

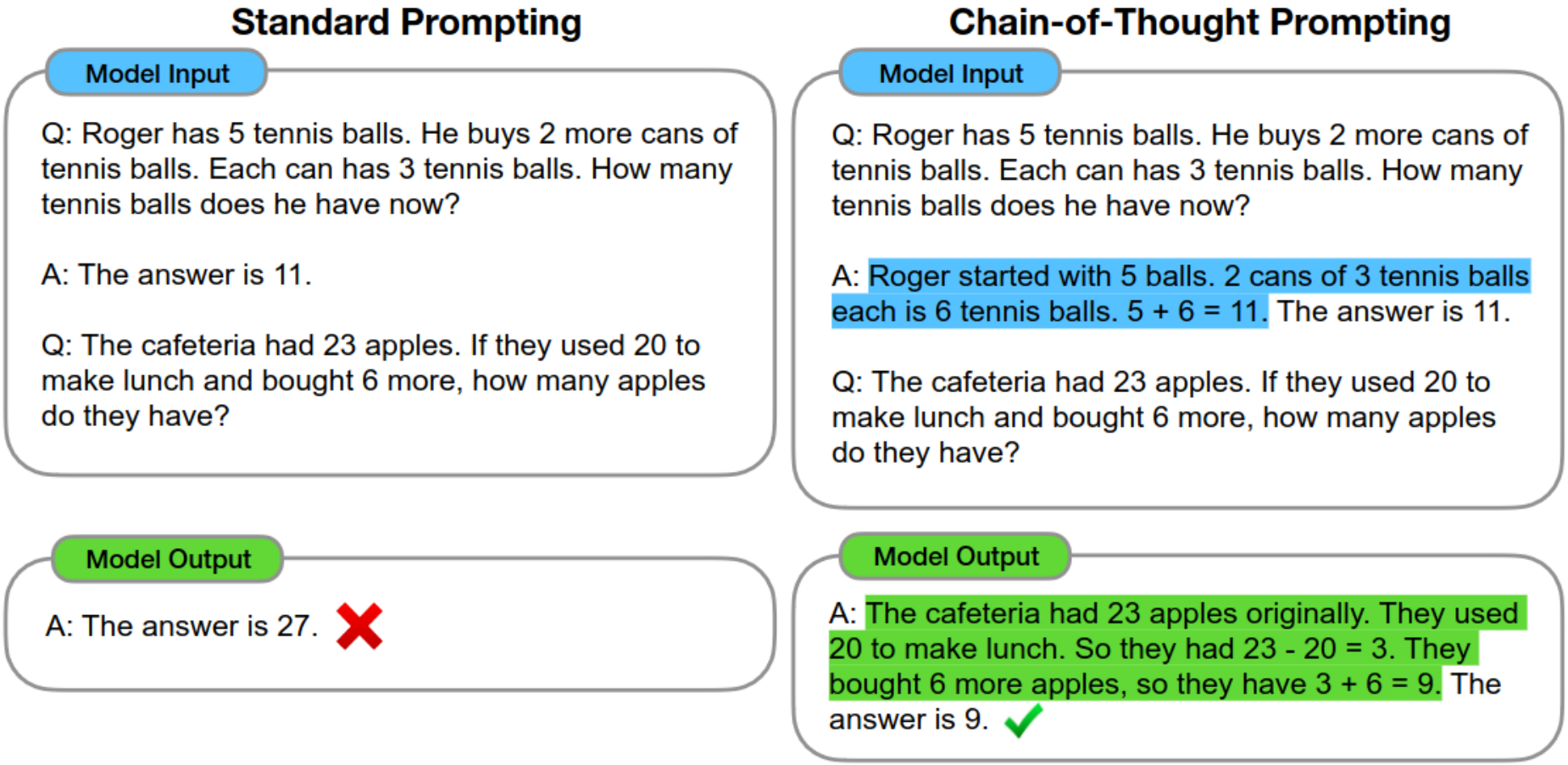


Figure 1: Chain-of-thought prompting enables large language models to tackle complex arithmetic, commonsense, and symbolic reasoning tasks. Chain-of-thought reasoning processes are highlighted.

Discussion

- Pretraining is often instrumental to good performance
  - Esp. in NLP (exploit large textual corpora) and CV (exploit resources such as ImageNet) or both
  - Generally, **hope that pretrained model useful for actual task**
  - Strong empirical evidence
  - Pretraining task/data very important (e.g., avoid spurious correlations)
  - Used via fine-tuning for task or prompting
- Related: **transfer learning**
  - Transfer learning: exploit model trained on different (but related) domain or task for actual task
  - E.g., pretrain model on related domain, then fine-tune on actual domain
- Related: **multi-task learning**
  - Simultaneously train for multiple tasks using a shared model
  - E.g., shared feature detector, task-specific prediction heads
  - Can improve data efficiency, reduce overfitting, and speed up learning