Content

1 Introduction

Artificial neural networks (ANNs) are a family of models inspired by biological neural networks. Usually, ANNs have an *input layer* and an *output layer*. Deep learning consists of ANNs with multiple *hidden layers,* that perform intermediate computations.

Embeddings are learned dense, continuous, low-dimensional vector representations of objects. They are useful to represent complex objects (image data, graph data, tabular data, textual data).

In practice, neural networks are usually trained using deep learning frameworks such as *PyTorch*. When using training data, the framework collects operations and their outputs to build a *computation graph*. This allows automatic gradient computation from this graph using *backpropagation*. The optimizer uses this gradient to update the model parameters in order to minimize some cost function.

Common Problems

We have large and complex models with many parameters. Training takes time and is costly.

We may face limited training data as large, labeled datasets are generally not available. The supervision signal alone may be insufficient to achieve reasonable performance.

Overfitting is a severe concern. Universal approximation theorem states that with sufficiently many hidden neurons, FNNs can perform arbitrarily well on the *training* set.

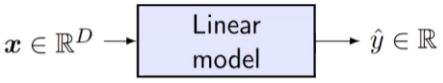
2 Feed-Forward Neural Networks

2.1 Embeddings

2.1.1 Linear Models

Consider a prediction task with inputs $x \in \mathcal{X}$ and outputs $y \in \mathcal{Y}$. The goal is to learn a function from $\mathcal{X}$ to $\mathcal{Y}$. Perhaps the simplest approach is to use a (generalized) linear model.

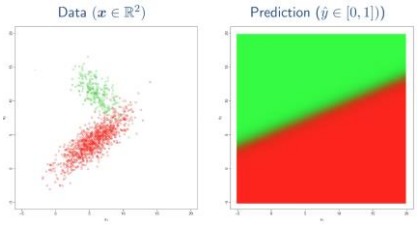
- Inputs must be real-valued feature vectors $\boldsymbol{x} \in \mathbb{R}^D$
- Outputs are a real value (e.g. lin./log. regression)



A linear model computes a weighted sum (inner product) of the feature vector $\boldsymbol{x}$ with the model weights $\boldsymbol{w}$:

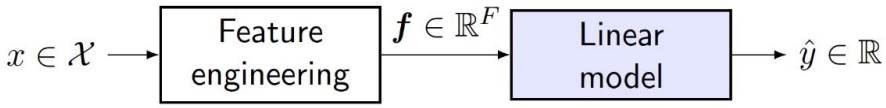
$$\hat{y} = \phi(\boldsymbol{w}^\top \boldsymbol{x} + b)$$

where $\boldsymbol{w} \in \mathbb{R}^D$ is a weight vector (one weight per feature), $b \in \mathbb{R}$ is a bias term and ϕ is a mean function.



❗ The problem of these models is the *low representational capacity* due to the linearity assumption. This means that the decision boundary is a linear hyperplane.

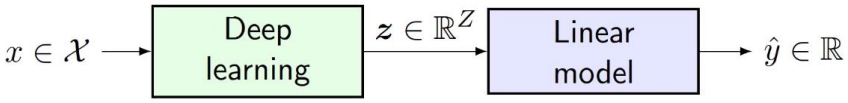
One approach to solve this problem is to perform *feature engineering*. Design some form of function that maps arbitrary input spaces to real-valued vectors $f: \mathcal{X} \to \mathbb{R}^F$



But this is hard to get right, usually requires expert domain knowledge and extensive experimentation.

2.1.2 Use of Deep Learning

📄 Deep Learning methods can be interpreted as an approach to learning features. Input objects $x \in \mathcal{X}$ are transformed into dense, low-dimensional representations called *embeddings* $\boldsymbol{z} \in \mathbb{R}^Z$.

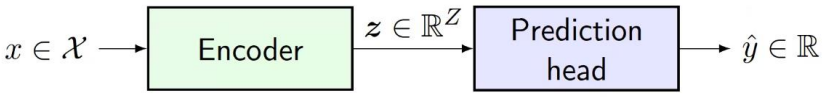


Instead of engineering features manually, embeddings are learned from data. They are also called latent code or distributed representations. The *embedding space* is also called *latent space*.

💡 Functions that transform objects $x \in \mathcal{X}$ to embeddings $\boldsymbol{z} \in \mathbb{R}^Z$ are known as *encoders*.

2.1.3 Terminology

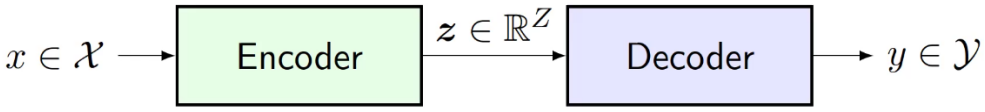
📄 Functions that transform embeddings $\boldsymbol{z} \in \mathbb{R}^Z$ to predictions $y \in \mathcal{Y}$ are known as *prediction heads*. They can be arbitrary simple/linear or complex.



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

2.1.4 Encoder-Decoder

Functions that decompress embeddings $\boldsymbol{z} \in \mathbb{R}^Z$ to obtain (complex) outputs $y \in \mathcal{Y}$ (or reconstructions $\hat{\boldsymbol{x}}$) are known as *decoders*. They generate a structured output.

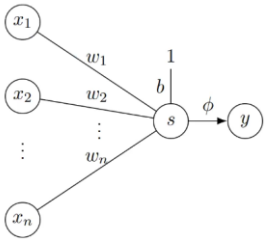


Decoders may be a reversed architecture of the encoder, but not necessarily.

2.2 Artificial Neurons

An artificial neuron (AN) is a function $f: \mathbb{R}^n \to \mathbb{R}$ that takes as some real-valued vector $\boldsymbol{x} \in \mathbb{R}^n$ an input and produces a single real value $y \in \mathbb{R}$ as an output, called *activation*.

$$y = \phi(\boldsymbol{w}^\top \boldsymbol{x} + b)$$



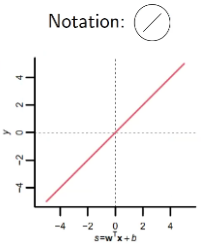
There are many types of neurons that only differ in their *activation* (*transfer*) function ϕ.

The simplest type of neuron is a **constant neuron**

- No inputs; output fixed value $x \in \mathbb{R}$
- Notation (from now on): x

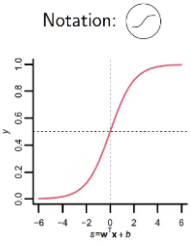
2.2.1 Linear Neuron / Identity

The linear neuron uses $\phi(s) = s$ as an activation function. This is very simple but computationally limited. We often (not always) want non-linear transfer functions



2.2.2 Logistic Neuron

The logistic neuron uses $\phi(s) = \sigma(s) \stackrel{\text{def}}{=} \frac{1}{1+\exp(-s)}$ as an activation function. This gives a real-valued output that is smooth and bounded in [0,1].

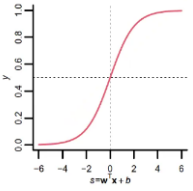


- negative activations mapped to $\phi(s) < 0.5$
- 0 activation mapped to $\phi(s) = 0.5$
- positive activations mapped to $\phi(s) > 0.5$

This gives us non-linearity and is often used in the output layer!

2.2.3 Stochastic Binary Neuron

Also uses the logistic function, but the output is treated as a probability of producing a spike:



$$\phi(s) = \begin{cases} 1 & \text{with probability } \sigma(s) \\ 0 & \text{otherwise} \end{cases}$$

This defines a probability distribution over outputs.

2.3 Artificial Neural Networks

A network of artificial neurons has a set of ANs with (directed or undirected) connections between these neurons. There are many architectural choices to be made:

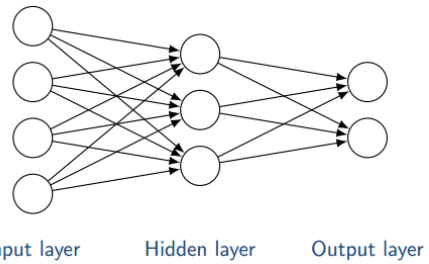
- How many neurons, which type?
- Output neurons? Hidden Neurons?
- Which are connected?
- Directed or undirected connections?

Picking the right architecture for the problem at hand is important → *architecture engineering*.

2.3.1 Feedforward Neural Network (FNN)

A feedforward neural network (FNN) is an ANN in which

- all connections are directed
- 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)

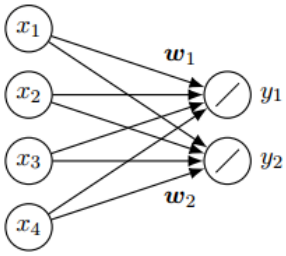
Layers do not need to be fully connected. Traditionally edges are only between subsequent layers (but: edges that skip layers are allowed, too)

💡 FFNs are discriminative models; given an input they compute an output but don't allow going from out- to inputs.

💡 As hidden layers outputs are the inputs of the next layer, we may think of hidden layers as intermediate features for the next layer. These are not provided upfront but learned.

2.3.2 Linear Layers

Layers in which all layer inputs are connected with all layer's outputs are called *dense* or *fully connected layers*. A dense *linear layer* is a layer consisting of only linear neurons: n layer inputs ($\boldsymbol{x} \in \mathbb{R}^n$) and m layer outputs ($\boldsymbol{y} \in \mathbb{R}^m$).



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

They are parameterized by weight vectors $\boldsymbol{w}_1, \dots, \boldsymbol{w}_m \in \mathbb{R}^n$ and optionally: biases $b_1, \dots, b_m \in \mathbb{R}$. Neuron outputs are given by:

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

💡 Let $\boldsymbol{W} \in \mathbb{R}^{n \times m}$ be a *weight matrix* in which the j-th column equals the weights $\boldsymbol{w}_j$ of the j-th layer output:

$$\boldsymbol{W} = (\boldsymbol{w_1} \quad \boldsymbol{w_2} \quad \dots \quad \boldsymbol{w}_m)$$

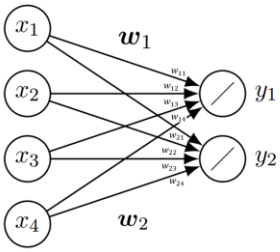
Linear layers compute a matrix-vector product:

$$\boldsymbol{y} = \boldsymbol{W}^\top \boldsymbol{x}$$

For our example $\boldsymbol{W} = (\boldsymbol{w_1} \quad \boldsymbol{w_2})$, and:

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

Example



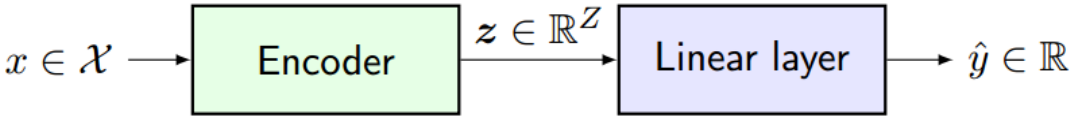
$$\boldsymbol{W} = (\boldsymbol{w_1} \quad \boldsymbol{w_2}) = \begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \\ w_{14} & w_{24} \end{pmatrix} \quad \boldsymbol{x}^\top = (x_1 \quad x_2 \quad x_3 \quad x_4)$$

$$\boldsymbol{y} = \boldsymbol{W}^\top \boldsymbol{x} = \underset{\in \mathbb{R}^{2\times 4}}{\begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix}} \underset{\in \mathbb{R}^{4\times 1}}{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}} = \underset{\in \mathbb{R}^{2\times 1}}{\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}}$$

❗ TODO Refer to batch processing.

Linear layers are typically used as:

- an output layer for regression tasks



- a hidden layers to perform dimensionality reduction ($m < n$)
- a hidden layer to increase dimensionality ($m > n$)

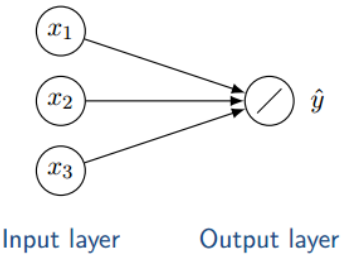
2.3.3 Linear Regression as FNN

In a linear FNN, all neurons/layers are linear. The simplest linear FNN has a single linear layer with one output. Its output is:

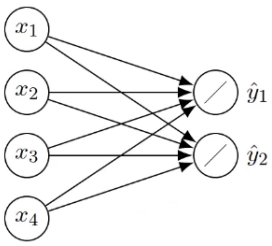
$$\hat{y} = \langle \boldsymbol{w}, \boldsymbol{x} \rangle + b$$

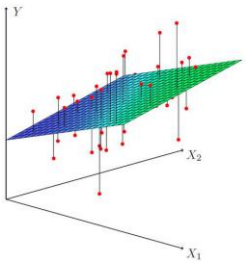
Suppose we train this network using ERM with squared loss:

$$\frac{1}{N}\sum_i (y_i - \hat{y}_i)^2$$



Then we obtain the ordinary least squares (OLS) estimate for linear regression. With multiple outputs, we obtain *multiple linear regression*.

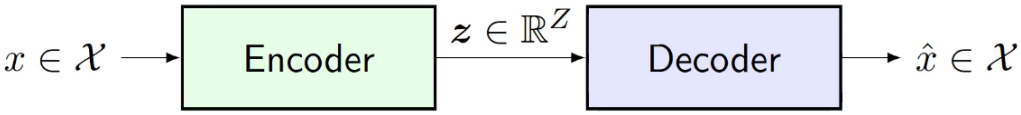




❗ The output of a linear FFN remains linear even when adding hidden layers. That's why we often want non-linear transfer functions.

2.3.4 Autoencoders

FNNs are useful for unsupervised learning as well. Given an unlabeled dataset $\mathcal{D} = \{\boldsymbol{x}_i\}_{i=1}^N$ we want to find structure, patterns or reduce dimensionality. The way autoencoders do this, is by training the FNN to predict its input, i.e., set $\boldsymbol{y}_i = \boldsymbol{x}_i$.



Autoencoders are a technique to learn embeddings. We don't really care about the decoder.

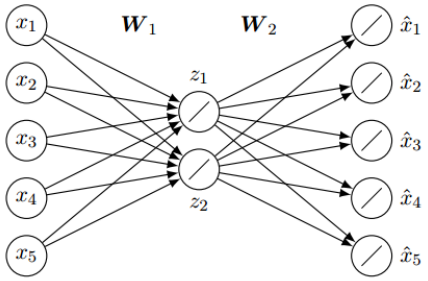
💡 Further, e.g. in semi-supervised learning, we can train the autoencoder on all inputs (unsupervised) and can then only train the prediction head in a supervised fashion using the labeled data. This allows the model to benefit from both, especially when labeled data is scarce.

Other uses are:

- Clustering: use embeddings as inputs to, say, K-means
- Denoising: use $\hat{x}$ (noisy input) instead of x but still try to predict x
- Visualization: visualize $\boldsymbol{z}$ (e.g., using Z=2 as in SVD)

2.3.5 Linear Autoencoders

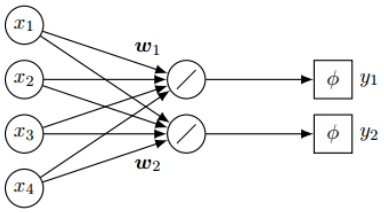
A linear autoencoder uses only linear layers (in both encoder and decoder). Consider a linear autoencoder with $\boldsymbol{x} \in \mathbb{R}^D$ and one hidden layer with $Z < D$ hidden neurons.



💡 A layer with few neurons is referred to as a *bottleneck, i.*e., fewer neurons than the surrounding layers. This forces FNN to compress information → *dimensionality reduction*

Since autoencoders need 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 ($\boldsymbol{z}$).

2.4 Non-Linear Layers

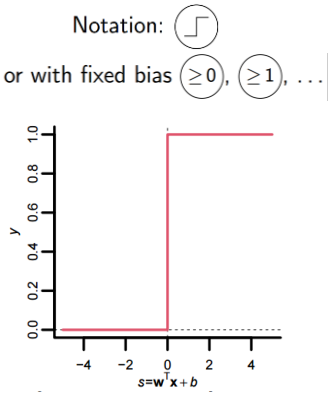


We can also interpret a fully connected layer as a (learned) linear layer followed by a (fixed) non-linearity. The action of the layer (without bias) is:

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

where we take the convection that ϕ is applied elementwise on vector inputs.

2.4.1 Binary Threshold Neuron

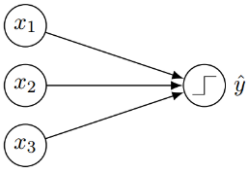


One of the (seemingly) simplest non-linear neurons is the binary threshold neuron (also called *McCulloch-Pitts neuron*). It uses the binary threshold function as transfer function:

$$\phi(s) = \mathbb{I}(s \geq 0) = \begin{cases} 1 & \text{if s} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

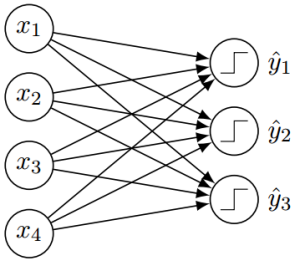
outputs fixed "spike" if input s is non-negative, else "nothing".

2.4.2 Perceptron



Corresponds to an FNN without hidden layers and binary threshold units for outputs (*single-layer perceptron*). 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.

2.4.3 Perceptron's with multiple output units

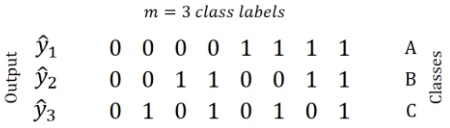


💡 Consider a perceptron with m binary outputs for classification tasks.

There are mutliple ways to interpret/use the output of such a network for classification.

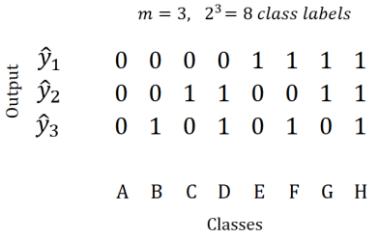
1. **multi-label classification** → works.

- 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), …



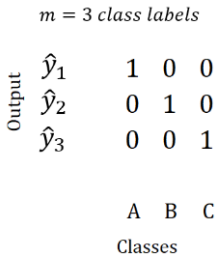
2. **multi-class classification** → problematic

- Each input is associated with one out of 2^m class labels
- We associate each label (A-H) with one output vector $(\hat{y}_1 \quad \hat{y}_2 \quad \hat{y}_3)^\top$ of the perceptron
- ❗ Problem: Which class label corresponds to which output vector? → choice matters!



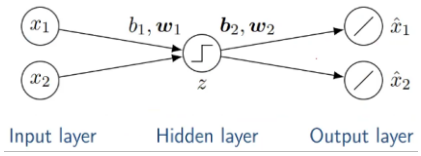
3. **multi-class classification** → problematic

- 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?



2.4.4 Autoencoders with Binary Threshold Unit

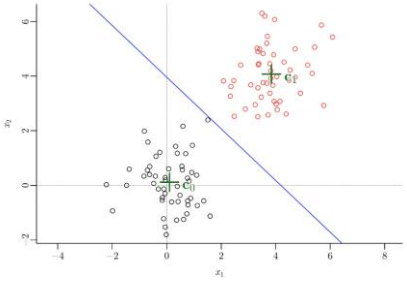
Consider the following autoencoder (image on the right). Here, the output z is constrained to be binary, taking values from the set {0,1}, creating a binary embedding.



💡 When minimizing the squared error over the training data, the autoencoder effectively computes the centroids of *"K=2"* clusters as embeddings.

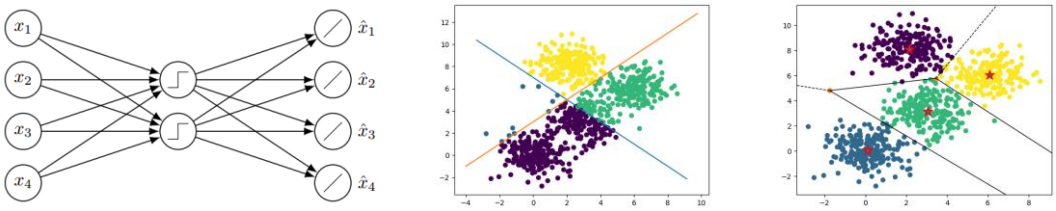
This happens as the binary threshold unit acts as a linear classifier. The input x is either mapped to $z = 0$ or to $z = 1$.

The weights b_1 and $\boldsymbol{w}_1$ will define a hyperplane separating the two possible outputs.



This is due to the binary threshold unit acting as a lin. classifier. Input $\boldsymbol{x}$ is mapped either to $z = 0$ or $z = 1$, depending on the weights b_1 and $\boldsymbol{w}_1$, effectively defining a hyperplane separating the two potential outputs. This is akin to the decision boundary in traditional linear classification.

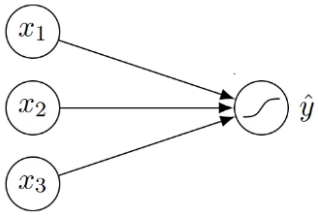
What happens if we have multiple binary threshold units?



Output produced from the autoencoder (left) vs. the K-Means counterpart (right).

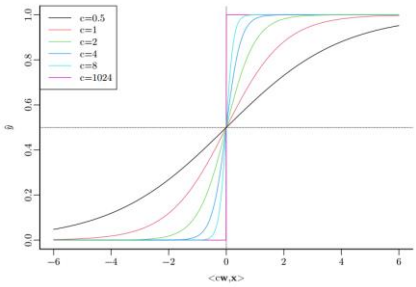
2.4.5 FNN with single Logistic Neuron

Recall the logistic neuron, *cf. 2.2.2*, and consider the FNN to the right.



If the output of the logistic unit $\hat{y} = \sigma(\langle \boldsymbol{w}, \boldsymbol{x} \rangle)$ is rounded to the nearest integer ($\in \{0,1\}$), we obtain the output of the corresponding perceptron (has binary threshold unit). Hence, we can see the logistic unit as a smooth version of a binary threshold unit.

If we scale the weights $\boldsymbol{w}$ by some constant $c > 0$, we change the degree of smoothing. This means that the norm of the weight vector $\|\boldsymbol{w}\|$, essentially determines the behavior of the logistic neuron.



Now 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):

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

which is *equivalent to the perceptron*. While the outputs $\hat{y}$ are related to the distance from decision boundary, there is no probabilistic interpretation possible.

We can also maximize the log-likelihood of the provided labels:

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

which is *equivalent to logistic regression*. Inputs $\langle \boldsymbol{w}, \boldsymbol{x} \rangle$ to logistic transfer function can be interpreted as estimate of the log odds of positive class. The output $\hat{y}_i$ can be interpreted as confidence for positive class.

> 💡 Hence, the output layer of FNNs for binary classification tasks is typically a logistic neuron.

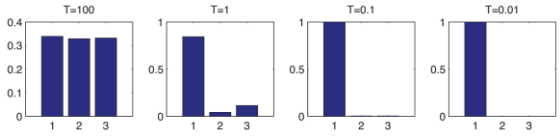
2.4.6 Softmax Function

The *Softmax function* takes as input a vector $\boldsymbol{\eta} = (\eta_1, \dots, \eta_C)^{\mathrm{T}} \in \mathbb{R}^C$ consisting of C real numbers and transforms these real values into a *probability vector* $S(\boldsymbol{\eta})$.

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

It exaggerates differences in the input vector. Below is the result for $\boldsymbol{\eta} = (3, 0, 1)^{\mathrm{T}}$ with $S\left(\frac{\boldsymbol{\eta}}{T}\right)$ at different *temperatures* T.

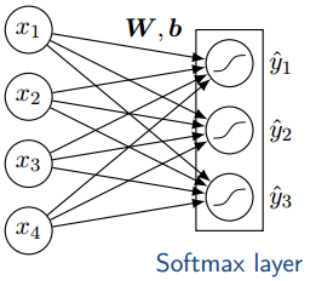
When the temperature T is high (left), the distribution is rather uniform, whereas when the temperature T is low (right), the distribution is "spiky", with all its mass on the largest element.



2.4.7 Softmax Layer

A *softmax layer* computes $\hat{\boldsymbol{y}} = S\left(\frac{\boldsymbol{W}^{\mathrm{T}}\boldsymbol{x}+\boldsymbol{b}}{T}\right)$, where $\hat{\boldsymbol{y}} \in \mathcal{S}_C$ is a probability vector, T is the temperature hyperparameter that controls smoothness of distribution (assume T = 1 for now).

A FNN with single softmax layer, trained with MLE / ERM + log loss, allows us to interpret $\hat{y}_c$ as the model's confidence in label c. This is *equivalent to multinomial logistic regression* (Softmax regression).



The output of a softmax layer with C neurons is an element of the *probability simplex* $\mathcal{S}_C$, with the following properties:

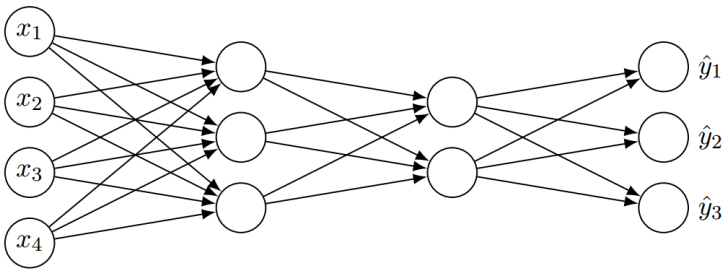
$$\mathcal{S}_C = \left\{ \boldsymbol{y}_c \in \mathbb{R}^C, \quad 0 < y_c \leq 1, \quad \sum_c y_c = 1 \right\}$$

this simply is the set of all vectors of size C, such that they are probability vectors.

2.4.8 Multi-Layer FNN / Perceptron (MLP)

💡 We can improve performance by engineering better features: by including *hidden layers*, we let the network perform feature engineering (we can interpret them as features for the next layers).

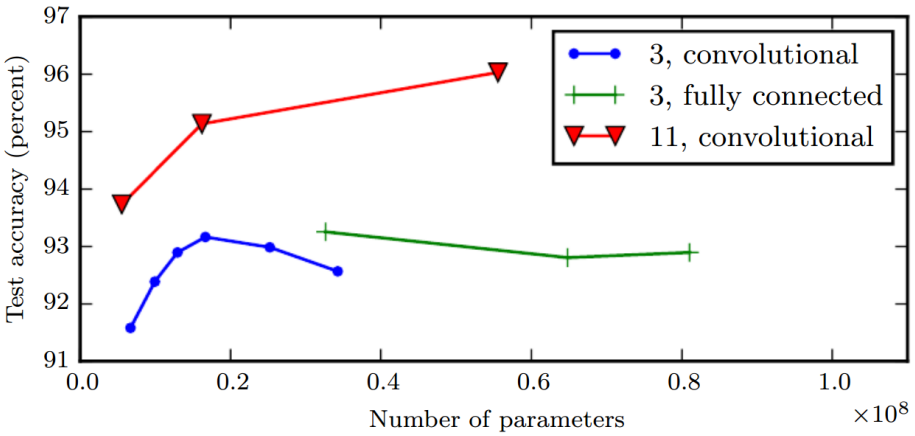
📄 If there is at least one hidden layer ($L \geq 1$) , the network is called a *multi-layer FNN* (or also *multi-layer perceptron: MLP*). If $L > 1$, we call this network *deep*.



❗ The *Universal Approximation Theorem* states that "any" function (on $[0,1]^D$) can be represented either via sufficient *width* or sufficient *depth* -- but that doesn't mean that we can learn it!

- Training methods may fail to find good parameterization
- Overfitting may occur
- Number of required units can be exponential in the input dimensionality

In general, empirical evidence shows that deep models tend to show better generalization performance than wide models.



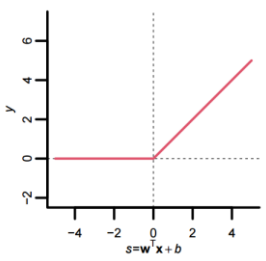
The legend indicates the *depth* of network used to make each curve. Increasing the number of parameters in layers of convolutional networks *without increasing their depth* is not nearly as effective at increasing test set performance.

2.4.9 Rectified Linear Unit (ReLU)

📄 Also called *linear threshold neuron* or *rectifier*. It outputs the weighted sum s if s is non-negative, else "nothing".

$$\phi(s) = \max\{0, s\} = \begin{cases} s & \text{if } s \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

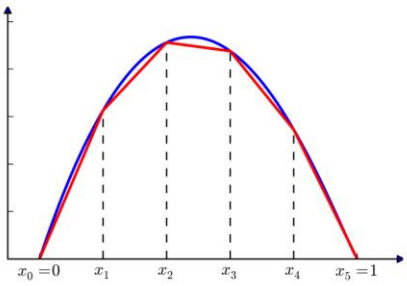
Notation:



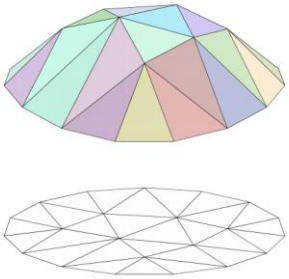
💡 Common non-linearity for intermediate layers in deep NNs.

2.4.10 Rectifier Networks

Rectifier networks are MLPs with only ReLU in hidden and output layers. The function computed by rectifier network is *piecewise linear it* → approximates a function by decomposing it into linear regions. The more linear regions, the more flexible/expressive (roughly).



1D example



2D example

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 $Z \geq D$ (→ *wide*)

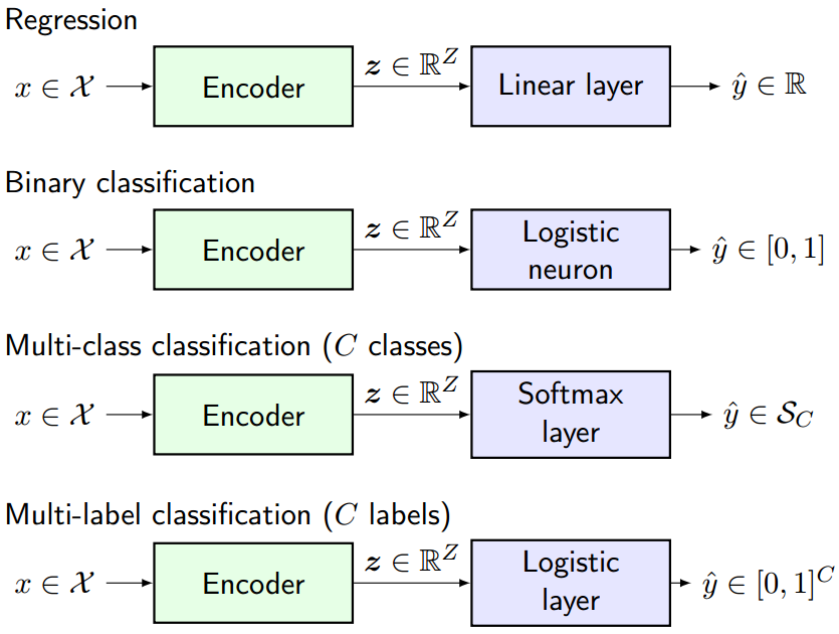
Then, the number of linear regions is *at most* 2^H → not more than exponentially many linear regions are possible! ("bounded from above")

The number of *attainable* linear regions *at least* $\Omega\left(\left(\frac{Z}{D}\right)^{(L-1)D} Z^D\right)$ → "bounded from below", with *attainable* meaning: assuming an optimal parameterization for a given architecture/form.

- Polynomial in Z (width)
- Exponential in L (depth)

Exponentially many linear regions are indeed possible!

2.5 Summary



3 Gradient-Based Training

Training FNNs in a supervised fashion involves minimizing a chosen *cost function*, typically through techniques like gradient descent or its variants. *Empirical risk minimization* (ERM) is a common approach in machine learning, with the goal to *minimize the average loss over the training data* $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)_{i=1}^N\}$. *Also see* *3.4.2 Effect of Batch Size*:

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

where $\boldsymbol{\theta}$ refers to the FNN's parameters. In the context of FNNs, this often involves minimizing the difference between the network's predictions $\hat{\boldsymbol{y}}_i = f(\boldsymbol{x}_i; \boldsymbol{\theta})$ and the true labels $\boldsymbol{y}_i$ of the training examples. The choice of cost function is crucial in determining the behavior and performance of the trained model. It defines what the network is trying to optimize during the training process.

> **Loss vs. Cost**
>
> A *cost function* typically refers to the objective function that the model aims to minimize during training. It encompasses not only the loss incurred on the training data but also potentially other regularization terms that penalize complex models or encourage certain properties. It is with respect to a single training example.
>
> A *loss function* specifically refers to the part of the cost function that quantifies the discrepancy between the model's predictions and the actual targets on the training data. Its is with respect to the entire training set.

Some common loss functions include:

- 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)

3.1 Backpropagation

Backpropagation is an algorithm to compute gradients. Given a compute graph *CG*, it performs:

1. Forward pass to compute (all) outputs → *forward propagation*
2. Backward pass to compute (all) gradients → *backward propagation*

For us the *comp. graph* typically represents:

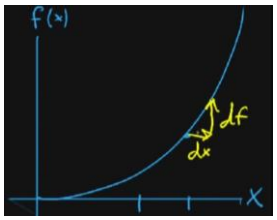
- Output $\hat{y}$ of an FNN, given $\boldsymbol{x}, \boldsymbol{\theta}$
- Loss L of an FNN, given $(\boldsymbol{x}, y), \boldsymbol{\theta}$
- Cost function J for an FNN, given $\{(\boldsymbol{x}_i, y_i)\}, \boldsymbol{\theta}$

We are also interested in gradients ∇:

- w.r.t. weights ($\nabla_{\boldsymbol{\theta}} J$) for gradient-based training
- w.r.t. intermediate outputs ($\nabla_{\boldsymbol{z}} L$) for model debugging
- w.r.t. inputs ($\nabla_{\boldsymbol{x}} L$ of $\nabla_{\boldsymbol{x}} \hat{y}$) for sensitivity analysis or adversarial training

3.2 Recap: Basics

3.2.1 Ordinary Derivatives



Ordinary derivatives are notated like this: $\frac{df}{dx}$ or simply $\frac{d}{dx}f$

They measure how a function $f(x)$ changes with respect to changes in x.

3.2.2 Partial Derivatives

The notation $\frac{\partial f}{\partial x}$ or $\frac{\partial}{\partial x}f$ is used for partial (∂) derivatives, where f is a function of several variables. Here we're finding the derivative with respect to one of those variables (e.g.: ∂x) while holding the others constant. For functions with multiple inputs, there are multiple partial derivatives.

$$f(x_1, x_2) = x_1^2 + 5x_1x_2 \qquad \frac{\partial}{\partial x_1}f = 2x_1 + 5x_2 \qquad \frac{\partial}{\partial x_2}f = 5x_1$$

3.2.3 Gradient

The gradient $\nabla_{\boldsymbol{x}^\top}$ of any function f gathers all its partial derivatives in a (row) vector:

$$\nabla_{\boldsymbol{x}^\top} f \overset{\text{def}}{=} \left(\frac{\partial}{\partial x_1}f \quad \frac{\partial}{\partial x_2}f \quad \dots \quad \frac{\partial}{\partial x_n}f\right)$$

For the example in 3.2.2, we obtain:

$$\nabla_{\boldsymbol{x}^\top} f = (2x_1 + 5x_2 \quad 5x_1)$$

Numerator layout (row)

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

Denominator layout (column)

3.2.4 Chain Rule

The chain rule tells us how to differentiate composite functions; it states:

$$\frac{d}{dv}f\big(g(v)\big) = \frac{df}{dg} \cdot \frac{dg}{dv} = f'\big(g(v)\big) * g'(v)$$

A function is composite if you can write it as $f\big(g(v)\big)$. In other words, it is a function within a function. For example, $f = \cos(v^2)$ is composite and $\frac{\partial}{\partial v}f = -\sin(v^2) * 2v$.

This can be generalized to any number of composite functions, for example:

$$\frac{d}{dx}f\Big(g\big(h(x)\big)\Big) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{dx} = f'\Big(g\big(h(x)\big)\Big) * g'\big(h(x)\big) * h'(x)$$

3.2.5 Product Rule

In calculus, the product rule is a formula used to find the derivatives of *products of two or more functions*. For two functions $u(x)$ and $v(x)$, it states:

$$(u * v)' = u' * v + u * v'$$

in Lagrange's notation

$$\frac{d}{dx}(u * v) = \frac{du}{dx} * v + \frac{dv}{dx} * u$$

in Leibniz's notation

The product rule can be generalized to products of more than two factors, e.g., for three factors:

$$\frac{d}{dx}(u * v * w) = \frac{du}{dx}vw + u\frac{dv}{dx}w + uv\frac{dw}{dx}$$

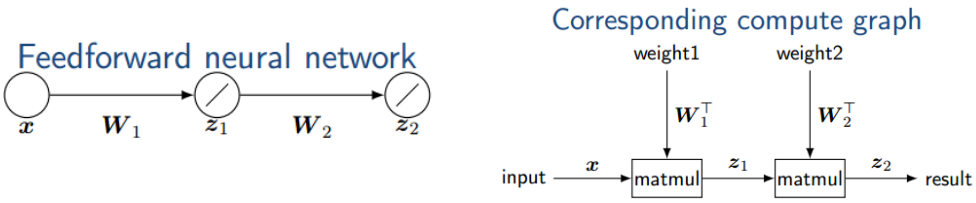
3.2.6 Multivariate Chain Rule

Consider a composite multivariable function $f\big(x(t), y(t)\big)$, its ordinary derivate is:

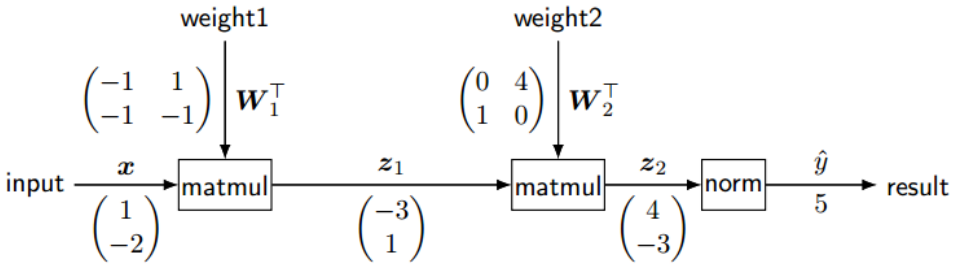
$$\frac{d}{dt}f\big(x(t), y(t)\big) = \frac{\partial f}{\partial x} \cdot \frac{dx}{dt} + \frac{\partial f}{\partial y} \cdot \frac{dy}{dt}$$

3.3 Compute Graphs

Backpropagation generally operates on a *compute graph*; a directed, acyclic graph that models a computation. Vertices (□) correspond to operations and edges (→) correspond to data passed between operations.



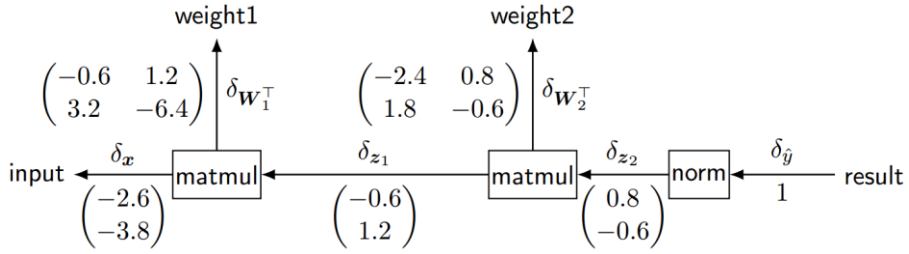
Example: Forward Pass



Operators are evaluated in topological order ("forwards"). Whenever an operator is evaluated, all its inputs must be available. Furthermore, the 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., $matmul(\boldsymbol{A}, \boldsymbol{B}) = \boldsymbol{AB}$ takes two 2D tensors and produces a 2D tensor.

💡 Intermediate results may need to be kept in order to evaluate subsequent operators and to enable gradient computation with backpropagation. Parallel processing is possible → transformer encoders.

Example: Backward Pass



We now want to compute gradients of the result for every edge in the compute graph.

$\delta_e \overset{\text{def}}{=}$ **gradient of result $\hat{y}$ wrt. values on edge e**

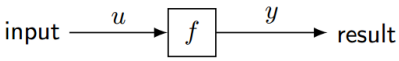
Example: $\delta_{\boldsymbol{z}_2} = \nabla_{\boldsymbol{z}_2}\hat{y} = \begin{pmatrix} 0.8 \\ -0.6 \end{pmatrix}$. This gradient tells us, that increasing the first value in $\boldsymbol{z}_2$ causes the result to *increase*; similarly, increasing the second value in $\boldsymbol{z}_2$ causes the result to *decrease*.

Note that the shape of each gradient in the backward pass is exactly the same as the shape of the respective inputs in the forward pass; generally: tensor valued.

Gradients δ_e can be computed incrementally akin to forward pass. Operators are evaluated in reverse topological order ("backwards"). When an operator is 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). Intermediate outputs of forward pass are required → memory consumption (or re-computation). Parallel Processing is possible.

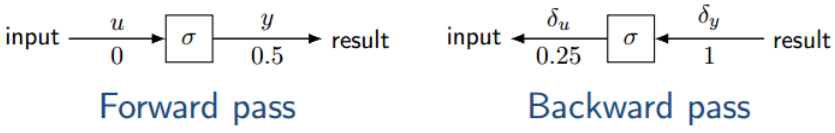
3.3.1 Gradient: Single Univariate Function

- Output $y = f(u)$
- Gradient $\delta_y \overset{\text{def}}{=} \nabla_y y = 1$
- Gradient $\delta_u \overset{\text{def}}{=} \nabla_u y = \nabla_u f(u) = \frac{\partial}{\partial u} f(u) = f'(u)$



Take-away: The gradient depends on the input u from the forward pass → we need to keep it!

Example: Consider the logistic function $y = \sigma(u)$ with input $u = 0$.



Gradient $\delta_u = \sigma'(u) = \sigma(u)\big(1 - \sigma(u)\big)$, evaluated at input $u = 0$ from forward pass:

$$\sigma(0)\big(1 - \sigma(0)\big) = 0.5 * (1 - 0.5) = 0.25 = \delta_u$$

3.3.2 Gradient: Composition of Two Univariate Functions

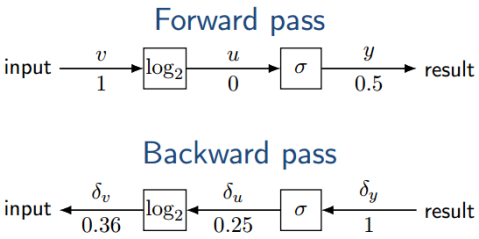
Output $y = f(u) = f\big(g(v)\big) \rightarrow$ function composition

Gradients can be obtained:

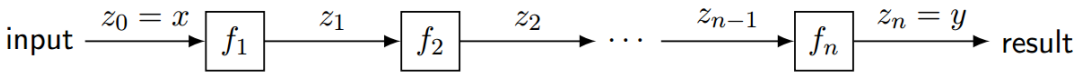
$$\delta_u \overset{\text{def}}{=} \nabla_u y = \frac{\partial}{\partial u} f(u) = f'(u) = f'\big(g(v)\big)$$

$$\delta_v \overset{\text{def}}{=} \nabla_v y = \frac{\partial}{\partial v} f\big(g(v)\big) = g'(v) * f'\big(g(v)\big)$$

$$= g'(v) * \delta_u \qquad \text{(Chain Rule)}$$



This generalizes, e.g., consider n operators:



We have: $y = f_n\Big(f_{n-1}\big(\dots\big(f_1(x)\big)\dots\big)\Big)$

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

$$\delta_{z_{i-1}} \overset{\text{def}}{=} \nabla_{z_{i-1}} y = \frac{\partial y}{\partial_{z_{i-1}}} = \frac{\partial y}{\partial_{z_i}} \cdot \frac{\partial z_i}{\partial_{z_{i-1}}}$$

$$= f_i'(z_{i-1}) \cdot \delta_{z_i}$$

Let's derive an expression for the gradients individually:

$$\begin{aligned}
\delta_{z_n} &= 1 \\
\delta_{z_{n-1}} &= f_n'(z_{n-1}) \cdot \delta_{z_n} &&= f_n'(z_{n-1}) \\
\delta_{z_{n-2}} &= f_n'(z_{n-2}) \cdot \delta_{z_{n-1}} &&= f_{n-1}'(z_{n-2}) \cdot f_n'(z_{n-1}) \\
\delta_{z_{n-3}} &= f_n'(z_{n-3}) \cdot \delta_{z_{n-2}} &&= f_{n-2}'(z_{n-3}) \cdot f_{n-1}'(z_{n-2}) \cdot f_n'(z_{n-1}) \\
&\dots
\end{aligned}$$

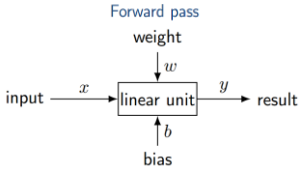
💡 The gradient is a product of local gradients along the path from the result to the resp. edge.

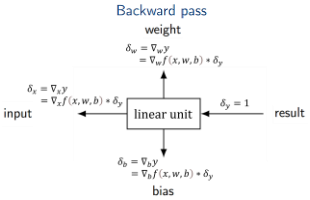
3.3.3 Gradient: Multiple Inputs

Operators often have multiple inputs, e.g., a simple linear unit. In the forward pass, the operator computes: $f(x, w, b) = wx + b = y$

In the backward pass, we simply compute gradients of result w.r.t. each edge using the chain rule:

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

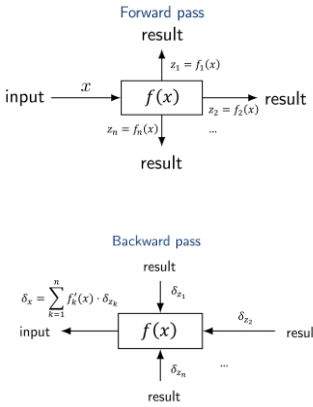




💡 We consider each input separately and reuse the δ-value.

3.3.4 Gradient: Multiple Outputs

Operators may have multiple outputs: multivariate operator $f(x)$ (single input) may output n values, say, $z_1 = f_1(x), \dots, z_n = f_n(x)$. During backpropagation, we obtain $\delta_{z_1}, \dots, \delta_{z_n}$. We are interested in:



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

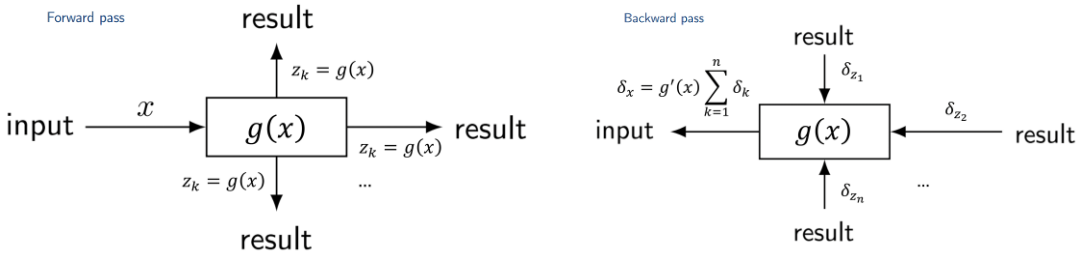
💡 We consider each output independently and sum up.

3.3.5 Gradient: Multiple Use

Sometimes an operator's output is "used" multiple times. E.g., the output of an operator $g(x)$ is used n times. That's equivalent to a single operator f with n identical outputs (i.e., $z_k = f_k(x) = g(x)$), each being used once.

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

💡 We sum up all incoming δ-values and proceed as before.



3.3.6 Gradient Computation with Tensors

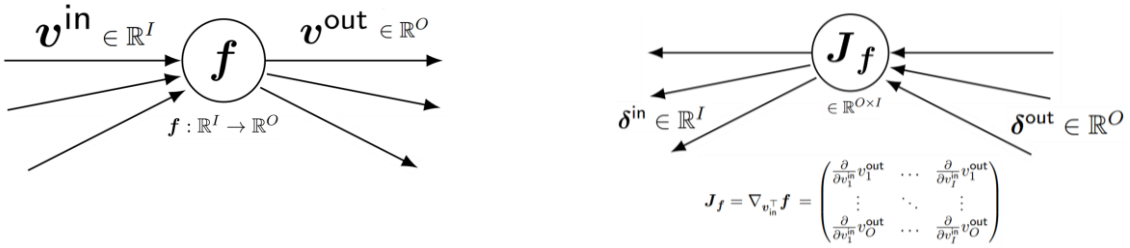
📄 In practice, we usually pass along *tensors* along the edges of the compute graph. Consider an arbitrary operator $\boldsymbol{f}: \mathbb{R}^I \to \mathbb{R}^O$ that performs on multiple inputs and outputs multiple values.

Forward pass

$$\boldsymbol{v}^{out} = \boldsymbol{f}(\boldsymbol{v}^{in})$$

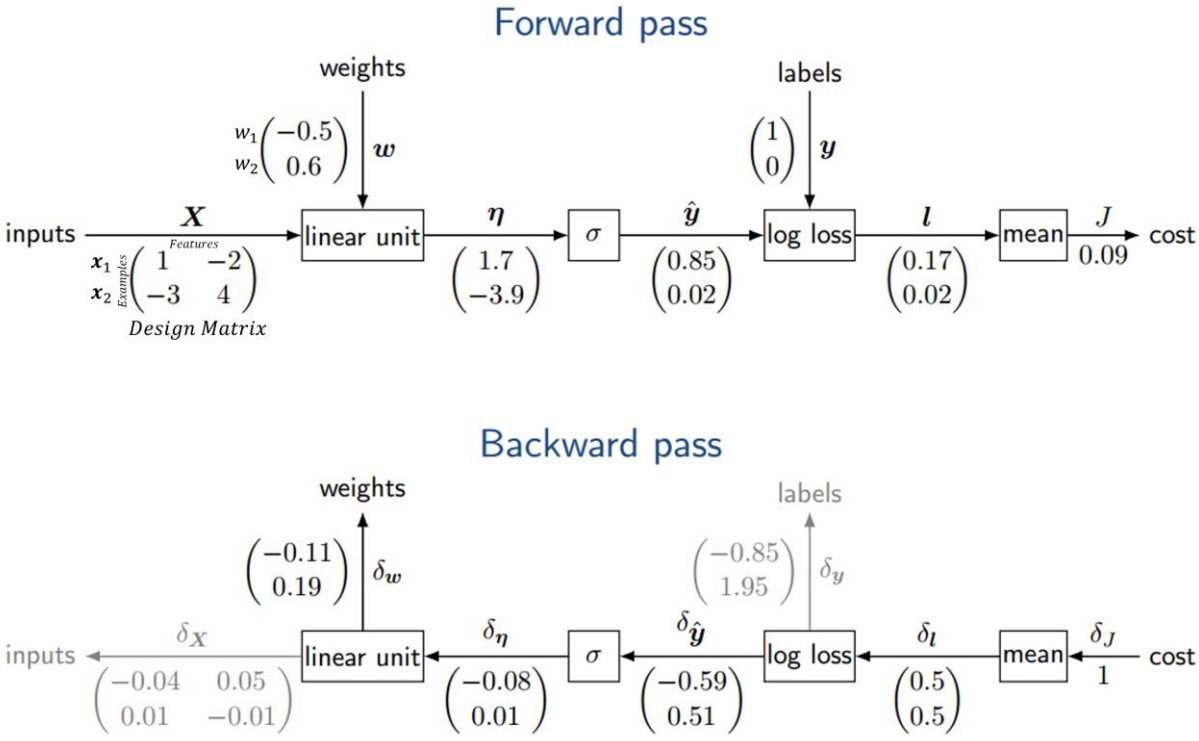
Backward pass

$$\boldsymbol{\delta}^{in} = \boldsymbol{J_f}(\boldsymbol{v}^{in})^{\mathsf{T}} \cdot \boldsymbol{\delta}^{out}$$

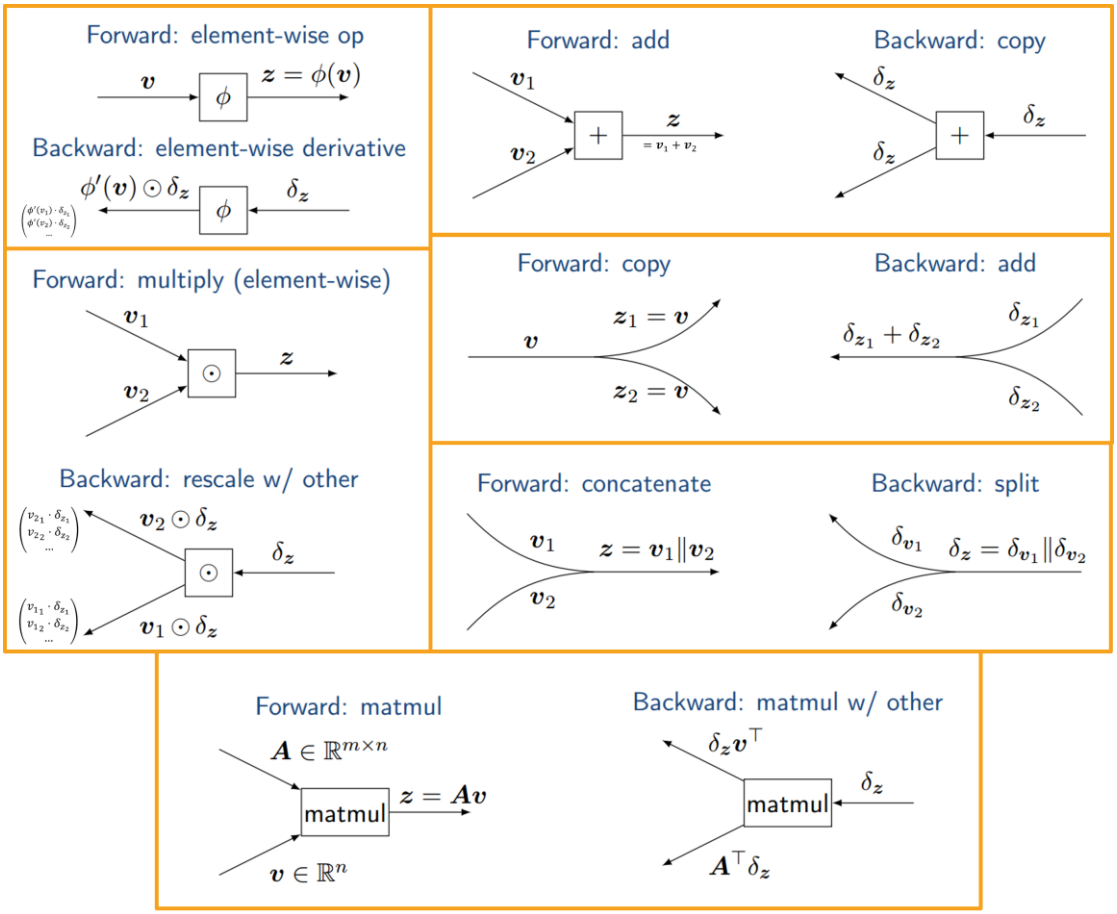


💡 The *Jacobian Matrix* $\boldsymbol{J_f}$ contains all partial derivatives of each output (columns) of $\boldsymbol{f}$ with respect to each input of $\boldsymbol{f}$ (rows). In the backward pass, these will be evaluated at the respective input value v^{in} from the forward pass, to obtain the real-valued matrix $\boldsymbol{J_f}(\boldsymbol{v}^{in})^{\mathsf{T}}$.

3.3.7 Example: Logistic Regression (2D), *N=2*



3.3.8 Backpropagation for Selected Operators



3.4 Optimizers

3.4.1 Gradient Descent

Recall *vanilla gradient descent*. This optimizer aims to minimize an objective function. It iteratively updates the current estimate $\boldsymbol{\theta}_t$ until some stopping criterion is met.

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

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

There are multiple variants for supervised learning:

- **Batch** gradient descent: computes exact gradient using *all training examples*
  - high cost - easy to parallelize - exact gradient
- **Stochastic** gradient descent: estimates gradient using *a single random training example*
  - low cost - noisy gradient - hard to parallelize
- **Mini-batch** gradient descent: estimates gradient using *some training examples*
  - middle ground between cost and parallelizability
  - number of examples called batch size

GD is suitable for large datasets/models but can converge slowly and gets stuck in local minima.

During training, we aim to minimize a potentially highly non-convex cost function $J(\boldsymbol{\theta})$ → difficult!

3.4.2 Effect of Batch Size

> Empirical Risk Minimization is a theoretic principle that guides the design of a variety of learning algorithms. Core idea:we cannot know exactly how well a predictive algorithm will work in practice (true "risk") because we do not know the true data distribution. We can instead *estimate* and optimize the performance of the algorithm on a known set of training data → minimize the *empirical risk* $R_{emp}(h)$. *Also see 3 Gradient-Based Training.*

Consider a *cost function* $J(\boldsymbol{\theta})$ over training examples $\mathcal{D}$ in the form of *empirical risk* $R_{emp}(h)$:

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

where $L_i(\boldsymbol{\theta}) = L(\hat{y}_i, y_i)$ is the loss on example i. Suppose we construct each batch $\mathcal{B}$ by sampling a fixed number of examples (uniformly & iid) and average losses:

$$J_{\mathcal{B}}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{B}|} \sum_{z \in \mathcal{B}} L_z(\boldsymbol{\theta})$$

Then, since $E[L_z] = \sum_{i \in \mathcal{D}} p_i \cdot L_i(\boldsymbol{\theta})$ with $p_i = \frac{1}{|\mathcal{D}|}$ (uniformly sampled) $E[L_z] = J$ follows; and:

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

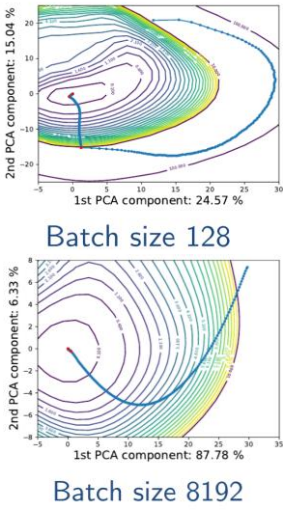
$$E[\nabla_{\boldsymbol{\theta}} J_{\mathcal{B}}(\boldsymbol{\theta})] = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

$$\text{var}[\nabla_{\boldsymbol{\theta}} J_{\mathcal{B}}(\boldsymbol{\theta})] = \frac{1}{|\mathcal{B}|} \text{var}[\nabla_{\boldsymbol{\theta}} L_z(\boldsymbol{\theta})]$$

Conclusion: gradient is correct in expectation; variance decreases with increasing batch size.
(this holds for this type of cost function)

3.4.3 Learning Rate ϵ and Batch Size $|\mathcal{B}|$

Red dot → learning rate reduced



For small batch sizes $|\mathcal{B}|$ → high variance gradients
- trajectory takes "detours"
- **regularizing effect** (escapes local minima)

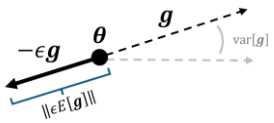
For large batch sizes $|\mathcal{B}|$ → low variance gradients
- trajectory attracted by local minima
- empirically often lower generalization performance

For low learning rates ϵ → small steps (slow trajectory)
- high variance gradient estimates average out

For high learning rates ϵ → large steps (fast trajectory)
- noisy gradient may be problematic

Consider $\boldsymbol{\theta}$ and the update to $\boldsymbol{\theta} - \epsilon \boldsymbol{g}$ during a mini-batch GD step.

- Step length $L = \|\epsilon E[\boldsymbol{g}]\|$
- Gradient variance $V = \mathrm{var}[\hat{\boldsymbol{g}}]$



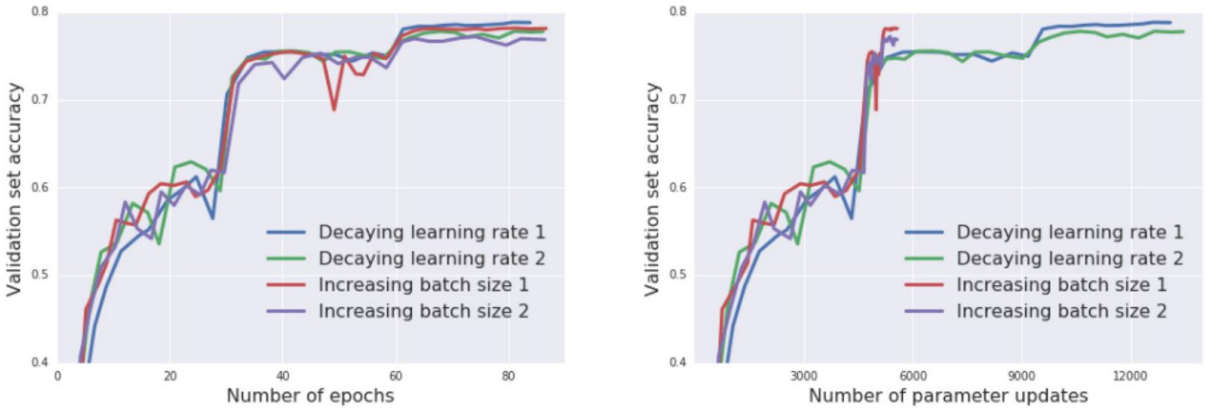
Both are affected by learning rate ϵ, batch size $|\mathcal{B}|$ and cost function J.

	$J_{\mathcal{B}}$=**mean loss**		$J_{\mathcal{B}}$=sum loss	
	L	V	L	V
$2\times$ step size	$2\times$	stays	$2\times$	stays
$2\times$ batch size	stays	$\frac{1}{2}\times$	$2\times$	$2\times$
$2\times$ step size, $\frac{1}{2}$ batch size	$2\times$	$2\times$	stays	$\frac{1}{2}\times$

Sum-Loss does not average examples in batch but sums them up.

💡 Learning rate and batch size are selected during hyperparameter tuning. Often a le*arning rate scheduler* is used to decrease an initially large ϵ to slowly approach the optimum once in vicinity.

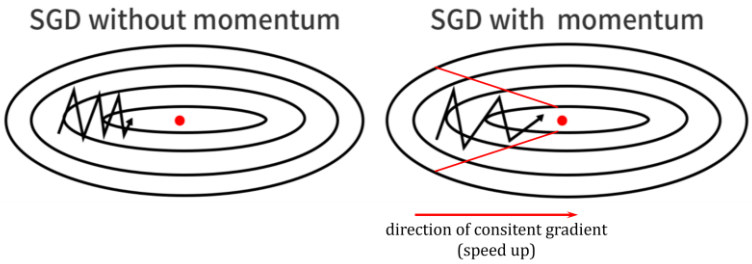
Similarly, a *batch size scheduler* can also be used to increase the batch size later during the training to slowly approach the optimum. This also eases parallelization.



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. Each experiment ran twice to illustrate the variance.

3.4.4 Momentum

A key idea to accelerate the GD algorithm is to build up velocity in directions that have consistent gradient.



This solves two problems: poor conditioning of the Hessian matrix (narrow valleys, see figure) and variance in the stochastic gradient.

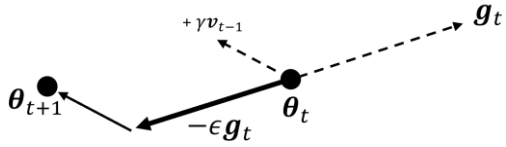
Vanilla/Basic *Momentum* (also: *heavy-ball method*) uses an exponentially-decaying moving average of the negative gradient.

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

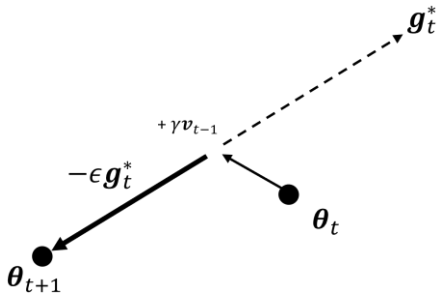
where we can think of $\boldsymbol{v}$ as *velocity*; hyperparameter $\gamma \in [0, 1)$ is referred to as *momentum.*

- The speed (norm of update) increased up to $\frac{1}{1-\gamma} \times$ w.r.t. GD step.
- $\frac{\epsilon}{1-\gamma}$ is called the *effective learning rate*, in the direction of a consistent gradient

Vanilla Momentum Gradient



Nesterov Momentum Gradient



Another variant is the *Nesterov momentum*, where the momentum update is applied *before* computing the gradient:

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

For convex functions (unique global optimum) and batch gradients, convergence rate improves from $O(1/t)$ to $O(1/t^2)$. Note that:

- this does not apply to SGD
- cost functions in DL are generally not convex
- but, in practice leads to much better performance

Requires additional memory to store velocity $\boldsymbol{v}$ (same size as model parameters $\boldsymbol{\theta}$).

3.4.5 Adaptive Learning Rates

The idea of adaptive learning rates is to use individual, per-parameter learning rates

- smaller learning rate for sensitive parameters (large derivative)
- larger learning rate for insensitive parameters (small derivative)

Example: Adagrad

Computes the sum of squared gradients to quantify parameter sensitivity:

$$\boldsymbol{r}_t \leftarrow \boldsymbol{r}_{t-1} + \boldsymbol{g}_t \odot \boldsymbol{g}_t$$

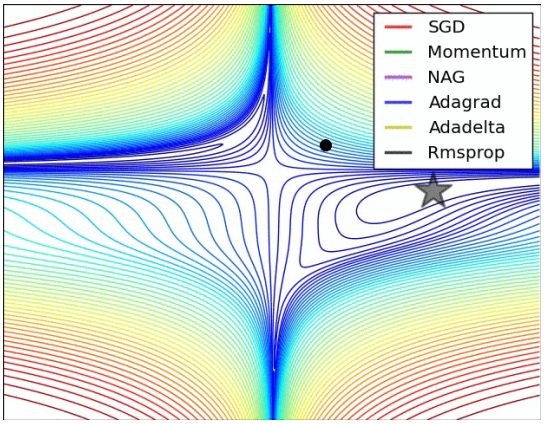
Note: operator $\odot$ is the elementwise multiplication.

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}_t}} \odot \boldsymbol{g}_t$$

Learning rate gets reduced over time, more so for parameters with larger derivatives.

- Good theoretical properties for convex functions
- For deep learning the learning rate reduction can happen too quick initially

Requires additional memory to store $\boldsymbol{r}$ (same size as model parameters $\boldsymbol{\theta}$).



Adadelta and *RMSProp* are variants that use an exponentially-decaying moving average of squared derivatives.

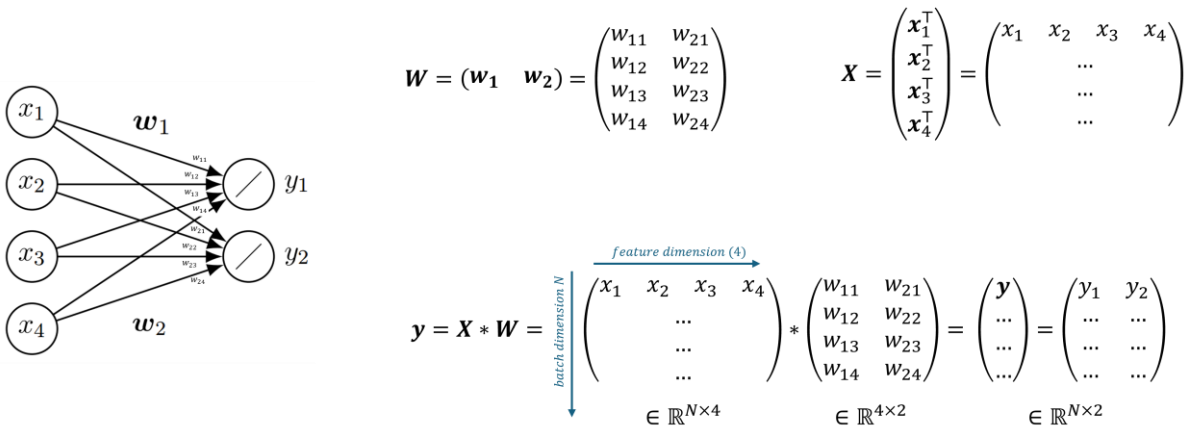
3.4.6 Discussion

- 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, …
  - Yet higher memory consumption (velocity and per-parameter LR)
- No consensus on best optimizer → hyperparameter tuning

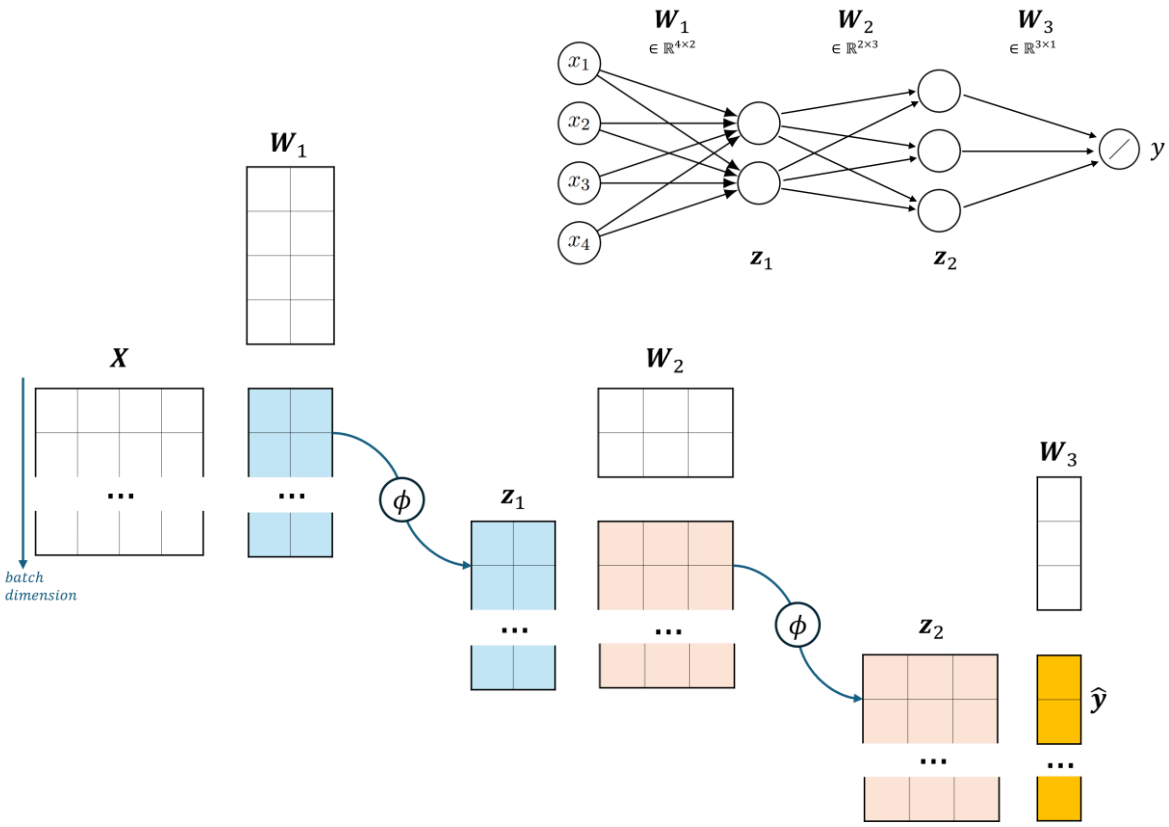
3.5 Forward Pass: Batch Processing

While we often use $\boldsymbol{y} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$ to describe the operation of a single layer during the forward pass, this is not quite accurate in practice. The reason why this notation is so common is to follow the existing notation of the rest of the models in the literature.

As we often want to pass batches (multiple $\boldsymbol{x}$'s), we'd like to make use of a design-matrix-like input structure. Consider the following example (omitting bias):

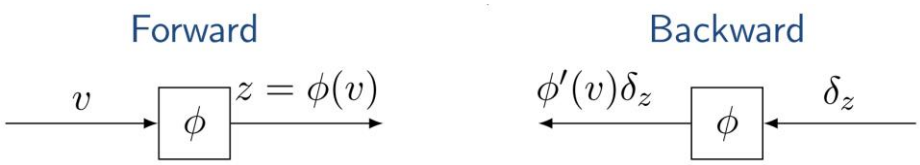


💡 Here we instead perform $\boldsymbol{y} = \boldsymbol{X} * \boldsymbol{W} + \boldsymbol{b}$ as the structure of our input has changed:



3.6 Vanishing/Exploding Gradient Problem (VGP/EGP)

Consider an activation function ϕ and recall the gradient computation in the backwards pass:



❗ If $\phi' < 1$, then the gradient becomes smaller. If this happens in consecutive layers, the gradient *vanishes* exponentially fast with depth, since local gradients multiply.

- If $\phi' \approx 0$, the gradient barely passes through → *saturated unit*
- If $\phi' = 0$, the gradient is gone right away → *dead unit*

This is problematic since prior layers do not receive a useful gradient signal. For $\nabla_{\boldsymbol{w}} J \approx 0$, gradient-based learning fails as weights are not updated in prior layers. For $\nabla_{\boldsymbol{x}} \hat{y} \approx 0$, model output becomes insensitive to input changes.

❗ Similarly, if $\phi' > 1$, then the gradient becomes larger. If this happens in consecutive layers, the gradient *explodes* exponentially fast with depth, since local gradients multiply.

This is problematic since the output is extremely sensitive to prior layers. For very large $\nabla_{\boldsymbol{w}} J$, gradient-based learning fails as weights diverge. For large $\nabla_{\boldsymbol{x}} \hat{y}$, model predictions are unstable.

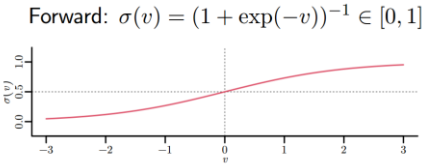
Note that this does not occur because of the activation functions, but rather due to other layers!

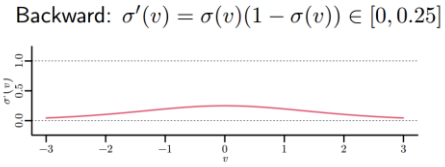
Example: Logistic Unit

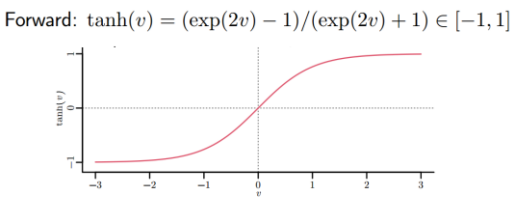
The logistic unit is not well suited as a hidden layer unit:

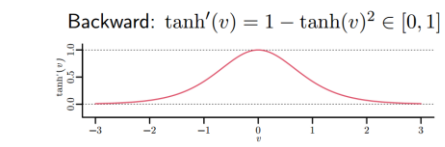
- $\sigma(0) = 0.5 \rightarrow$ zeros not passed through
- $\sigma'(v) \leq 0.25 \rightarrow$ gradients reduce by at least ¼
- The unit becomes saturated when $|v| \geq 5$.

💡 For these reasons we typically do not use logistic units as hidden neurons.









Example: $tanh(\,)$ activation function

The $tanh()$is a good alternative:

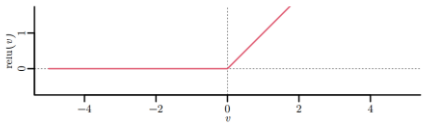
- $tanh(0) = 0 \rightarrow$ zeros passed through
- $tanh'(0) = 1 \rightarrow$ gradients are retained for small inputs
- The unit becomes saturated when $|v| \geq 3$.

Example: $ReLU(\,)$ function

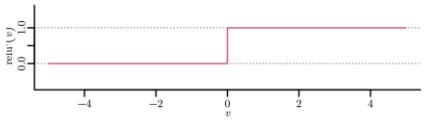
The $relu(\,)$ function is a good alternative:

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

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



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



Desirable Properties of Activation Functions

Non-Linearity → needed for expressiveness	**Differentiability** → enables gradient-based learning	**Zero's pass through** $\phi(0) = 0$ → avoids need to learn zero-outputs
Approximates linear unit around 0 → mitigates vanishing gradient	**Gradients bounded from above** → stability, mitigates expl. grad.	**Low computational cost** → (Leaky) ReLU wins

3.7 Architecture Design

Degradation problem: performance of plain MLPs tends to deteriorate beyond certain depth.

- Due to complicated optimization landscape
- Gradients may vanish/explode exponentially fast with increasing depth in MLP

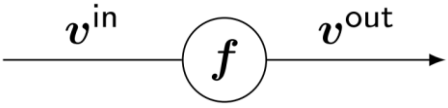
This can be mildened by choice of suitable activation function, but not by much. Instead, we can modify the *network architecture* to mitigate training challenges → better empirical performance.

Goals include

- Improve gradient flow through network
- Facilitate training and effectiveness of deep networks → mitigate the degradation problem
- Simplify the optimization landscape

3.7.1 Representation View

Consider a parameterized layer $\boldsymbol{f}: \mathbb{R}^Z \rightarrow \mathbb{R}^Z$ somewhere in an FNN, e.g.: $\boldsymbol{f} = \phi\left(\boldsymbol{W}_l^\top \cdot \boldsymbol{v}^{in} + \boldsymbol{b}_l\right)$ for a layer l:



How can we interpret what a layer is doing in an FNN? → *Representation view:*

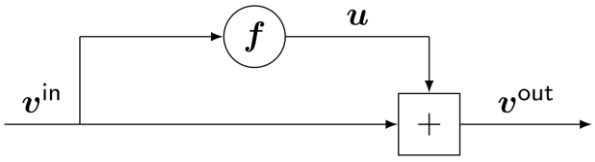
- $\boldsymbol{v}^{in}$ is representation/embedding of input (obtained from previous layer)
- $\boldsymbol{v}^{out}$ is updated representation of input (for next layer)

Each layer is considered a "step" in a multi-step computation

- How many steps? ≈ *network depth* (L)
- How much memory? ≈ *network width* (Z)

If $\boldsymbol{v}^{in}$ is already a good representation, the model needs to learn to preserve it. The deeper the network is, the more this matters → one reason for the degradation problem.

3.7.2 Residual Connections



Residual Connections are an architectural design pattern that change the layer:

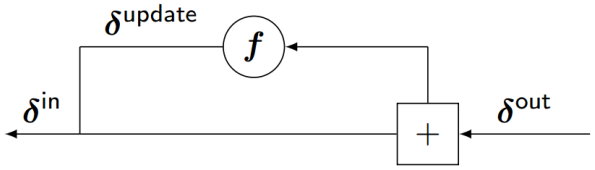
$$\boldsymbol{v}^{out} = \boldsymbol{v}^{in} + \boldsymbol{f}(\boldsymbol{v}^{in})$$

We think of $\boldsymbol{f}(\boldsymbol{v}^{in})$ not as a representation that should be used downstream but rather as an *update* or *residual*. How much should the representation change – not how should it be.

💡 If $\boldsymbol{v}^{in}$ already is a good representation, then learning to preserve this is now easier to do. This reduced degradation of performance that would happen due to "too many" layers.

In the backward pass we obtain (cf. *3.3.8*):

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



After L of such layers, the gradient is:

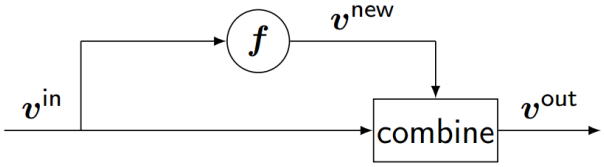
$$\boldsymbol{\delta}^{in} = \boldsymbol{\delta}^{out} + \boldsymbol{\delta}_L^{update} + \cdots + \boldsymbol{\delta}_1^{update}$$

Therefore, the original gradient $\boldsymbol{\delta}^{out}$ passes through → addresses vanishing gradient problem.

3.7.3 Skip Connections

More generally, *skip connections* (also: *shortcut connections*) skip one or more layers. Residual Connections are just one example of these.

The original representation $\boldsymbol{v}^{in}$ and the new representation $\boldsymbol{v}^{new}$ are ultimately combined in some form.



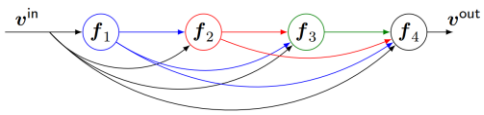
💡 Common combine-options include:

- Concatenation
- Averaging
- Max-Pooling
- Attention
- Addition (→ *3.7.2*Residual Connections)

Concatenation

In an extreme case, we concatenate *all* representations of previous layers. This makes preservation of information across layers trivial. Each layer merely enhances the current representation (increases effective dimensionality). Note that:

- Input size per layer increases linearly
- Amount of weight matrices increases linearly
- Total number of weights increases quadratically



💡 But we can get away with a much smaller output size Z per layer, reducing cost.

After L of such layers, the gradient is:

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

In a similar manner, the gradient from later layers $\boldsymbol{\delta}_L$ passes through.

3.7.4 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 → ignored by gradient-based methods
2. Gradient magnitudes may vary wildly across layers → complicates gradient-based learning

For BN we need to run batch or mini-batch gradient descent. It normalizes the input features to zero mean and unit variance within each batch. I.e., input $\boldsymbol{z} \in \mathbb{R}^Z$ is normalized to:

$$\tilde{\boldsymbol{z}} = \frac{(\boldsymbol{z} - \boldsymbol{\mu})}{\sqrt{\boldsymbol{\sigma}^2}}$$

where $\boldsymbol{\mu}, \boldsymbol{\sigma}^2 \in \mathbb{R}^Z$ are computed from the entire batch. This normalization is part of the layer, and the gradient is backpropagated through these operations. At test time, we use running average of $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ that we kept from training.

Variant: normalize to mean β and variance γ, where β and γ are learned parameters

Sometimes BN is problematic:

- During training, batch size must be sufficiently large (e.g., no online learning possible).
- When the number of layers is not fixed but depends on input (e.g., RNN, Transformer), where the number of required mean/variance statistics varies with input lengths.

3.7.5 Layer Normalization

Layer normalization (LN) is an alternative, where we normalize each input vector individually:

$$norm(\boldsymbol{z}) = \frac{\boldsymbol{z} - mean(\boldsymbol{z})}{\sqrt{std(\boldsymbol{z})}}$$

Mean/Standard deviation are computed across the elements of $\boldsymbol{z}$.

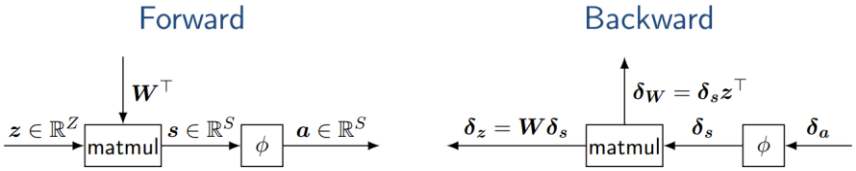
	Weight matrix re-scaling	Weight matrix re-centering	Weight vector re-scaling	Dataset re-scaling	Dataset re-centering	Single training case re-scaling
Batch norm	Invariant	No	Invariant	Invariant	Invariant	No
Weight norm	Invariant	No	Invariant	No	No	No
Layer norm	Invariant	Invariant	No	Invariant	No	Invariant

3.8 Initialization

Initialization refers to choosing starting values for all parameters of a NN. This is important since it affects the *performance* as well as *solutions found* by gradient-based optimizers.

3.8.1 Zero Initialization

Initializing weight matrices with zeros ($\boldsymbol{W} = \mathbf{0}$) is problematic as $\boldsymbol{\delta_z}$ in turn also is zero.



If $\boldsymbol{\delta_a} = c \cdot \mathbf{1}$ (only constant values), all vectors receive the same gradients. As a consequence, all units at each layer always have the same output and learning fails no matter for how long we train (→ *co-adaptation*: output neurons are highly correlated no matter the output).

3.8.2 Normal Initialization

When initializing $\boldsymbol{W}$ using iid. samples from a normal distribution $\mathcal{N}(0, \boldsymbol{\sigma}^2)$, the variance of the first layer output increases with increased *input* dimensionality Z. Depending on activation function ϕ this can lead to vanishing/exploding gradients. A solution is to initialize with samples from $\mathcal{N}(0, \boldsymbol{\sigma}^2/Z)$ → variance retained.

Similarly in the backward pass, when sampling from a standard normal distribution, the variance increases with increased *output* dimensionality S. Depending on activation function ϕ this can lead to exploding gradients. A solution is to initialize with samples from $\mathcal{N}(0, \boldsymbol{\sigma}^2/S)$ → variance retained.

❗ If input and output dimensionality differ ($S \neq Z$) a tradeoff is necessary. Generally, for MLPs, vanishing/exploding gradients can be mitigated at the start when variances are retained across layers. As forward and backward pass are affected differently, we need a compromise.

3.8.3 Xavier/Kaiming Initialization

For $\phi = \tanh(\,)$ or linear( ) we can select *Xavier initialization* (also: *Glorot init.*). Samples from:

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

where $D = mean(Z, S)$.

For other activation functions, can multiply by gain, e.g., for relu: $\sqrt{2}$ (*Kaiming/He initialization*)

3.8.4 Discussion

- Suitable scale matters for standard MLPs and depends on dimensionality
- Initialization differs based on architecture,
  - e.g., residual layers typically need a small weight matrix
  - scaling is less influential if layer/batch/weight normalization is used

4 Layers for Categorical Data

❗ *One-hot encoding* for categorical inputs, e.g.:

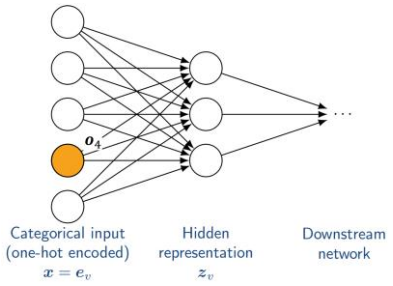
$$x \in \{\text{red}, \text{green}, \text{blue}\} \text{ then for } x = \text{green}, \text{ we encode it as } x = (0 \quad 1 \quad 0)^\top$$

is not efficient regarding compute cost and limited in parameter sharing.

4.1 Embedding Layers

Consider an FNN with a single categorical input.

- The set of categories (i.e. {red, green, blue}) is referred to as *vocabulary* $\mathcal{V} = \{1, \dots, V\}$.
- The input $v \in \mathcal{V}$ is one-hot encoded to $\boldsymbol{e}_v \in \{0,1\}^V$, e.g. $v = (0 \quad 1 \quad \dots \quad 0)^\top \rightarrow$ v[th] standard basis vector



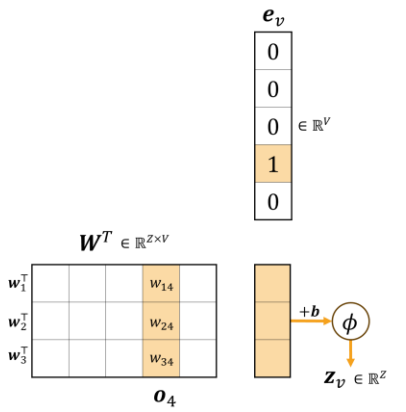
The downstream network only sees $\boldsymbol{z}_v$ and not $\boldsymbol{e}_v$.

If we view the weight matrix $\boldsymbol{W}$ as

$$\boldsymbol{W} \in \mathbb{R}^{V \times Z} = (\boldsymbol{w}_1 \quad \boldsymbol{w}_2 \quad \dots \quad \boldsymbol{w}_Z) = \begin{pmatrix} \boldsymbol{o}_1^\top \\ \boldsymbol{o}_2^\top \\ \dots \\ \boldsymbol{o}_V^\top \end{pmatrix}$$

where $\boldsymbol{o}_V^\top = (w_{1V} \quad w_{2V} \quad \dots \quad w_{ZV})$ are the weights coming from input neuron V to hidden neuron Z. Then we can express the output of the first layer in the following way:

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



📄 An *embedding layer* stores a representation of each category (*embeddings*) → no computation. It maps each category $v \in \mathcal{V}$ to a vector $\text{emb}(v) \in \mathbb{R}^Z$, the *embedding* of v. Such a layer is parameterized an *embedding matrix* $\boldsymbol{E} \in \mathbb{R}^{V \times Z}$ with categories as rows. This is more efficient than modelling via a fully connected layer. When training such a network, we can use the obtained gradient to update the embedding matrix and learn representations.

💡 Embedding layers are used for *non-divisible* objects (i.e. words, atomic objects); for divisible objects an encoder is used such that exploiting the structure of the object is possible (e.g., document embeddings, image embeddings, …)

Discussion

- Embedding layers may use large vocabularies $V \gg Z \rightarrow$ dimensionality reduction
- Two categories should be similar in embedding space when they have similar impact on final output → Embeddings expose similarities
- When multiple inputs use the same categories (e.g. RNNs) the embedding layer is typically shared → this is an example of *parameter sharing.*

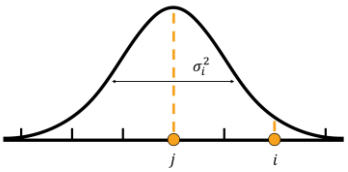
4.2 t-SNE

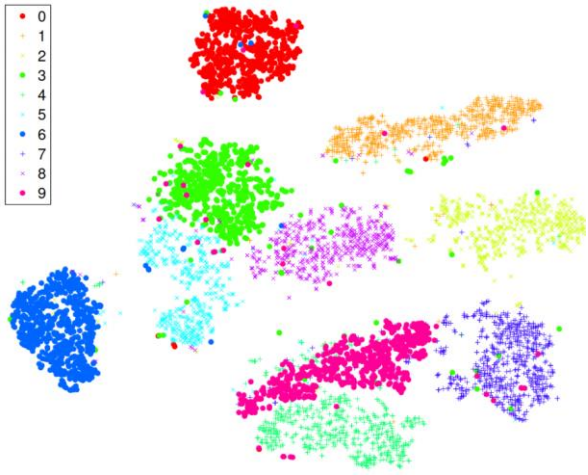
t-SNE (*t-distributed Stochastic Neighbor Embedding*) is a popular approach to visualize embedding spaces. It approximates the neighborhood of data points in the embedding space, or any other high-dimensional space, and outputs these neighborhoods in 2D/3D space.

High-Level-Idea: Data points close in the embedding space should also be close in the low-dimensional space → *small distances matter*

The similarity in the embedding space is measured with an isotropic Gaussian distribution with data-point-specific variance:

$$p(\boldsymbol{z}_j|\boldsymbol{z}_i) = \mathcal{N}(\boldsymbol{z}_j|\boldsymbol{z}_i, \sigma_i^2 \boldsymbol{I})$$



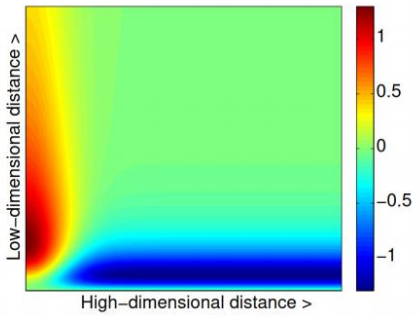


6000 MNIST handwritten digits; 28 × 28 → 784D

The data-point-specific variance σ_i^2 controls the number of effective neighbors. t-SNE sets this automatically such that we have small variance in dense areas and wide variance in sparsely populated areas (→ in embedding space).

In the low-dimensional space, t-SNE uses the Cauchy distribution s.t. the model accepts (*green area in plot*) data points with moderate similarity in the original data space to be far away in the low-dimensional space and vice-versa.

This makes *large* distances and cluster sizes in the t-SNE plot meaningless, i.e., the distance between number zero (**red**) and number one (**orange**) cannot be compared to the distance between number zero (**red**) and number four (**mint**). As they are far away in low-dimensional space we cannot reason about distance in the embedding space.



Gradient of t-SNE

4.3 Softmax Layers for Similarity

💡 Softmax layers also compute similarity. By computing the inner product between weight vector for each class and input to obtain the softmax score telling us the class probability p_c.

$$p(y = c|\boldsymbol{z}) = S(\boldsymbol{W}^\top \boldsymbol{z})_c = \frac{\exp(\langle \boldsymbol{w}_c, \boldsymbol{z} \rangle)}{\sum_{c'} \exp(\langle \boldsymbol{w}_{c'}, \boldsymbol{z} \rangle)} = p_c$$

Class probabilities are implicitly determined by softmax scores: $\eta_c = \langle \boldsymbol{w}_c, \boldsymbol{z} \rangle$. Using the geometric interpretation of the inner product: being proportional to the cosine similarity, the softmax layer normalizes these similarity scores to produce probabilities. We may thus think of a softmax layer as measuring the similarity between input $\boldsymbol{z}$ and each class vector $\boldsymbol{w}_c$. *Also see 2.4.6 and 2.4.7*.

> 💡 Cosine similarity only cares about angle difference, while the inner product cares about angle and magnitude. If you normalize your data to have the same magnitude, the two are indistinguishable.

4.4 Word Vectors

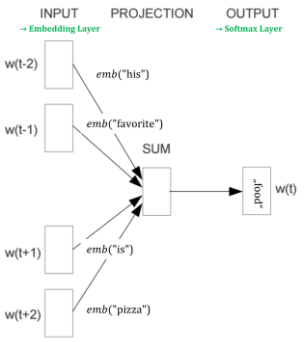
Word vectors, a type of word embeddings, use embedding layers and softmax layers to represent words in NLP tasks. While less relevant in current research, they are instructive. These vectors map words to continuous representations, aiming to *capture semantic similarity* and *compositionality*. By using word vectors, downstream models require fewer parameters to train and can handle unseen words in the training data by focusing on the meaning rather than the words themselves.

💡 They are guided by the *distributional hypothesis* (linguistics), which posits that words with similar meanings tend to occur in similar contexts. Thus, word vectors are trained to ensure that words with similar contexts have similar representations, thereby encoding similar meanings.

Example: *continuous bag-of-words* model (*CBOW*)

Predicts the missing word given a set of surrounding words (→ *context*).

- Hyperparameter Z: Size of the word vector.
- Hyperparameter W: Size of the left/right context.



The input layer maps words to word vectors (each of the $2W$ words separately) via an embedding layer. The embedding matrix $\boldsymbol{E} \in \mathbb{R}^{V \times Z}$ is shared across context words → *parameter sharing*

The sum layer takes $2W$ embeddings and sums them elementwise $\sum_i \text{emb}(w_i)$ (*composition*) producing a Z-dimensional *continuous* representation of the context.

The output layer (Softmax, trained to predict w_t with $C = V$ classes) will be ignored for downstream models, i.e., only $\boldsymbol{E}$ used).

💡 The standard BOW model would simply sum the word counts $\boldsymbol{e}_{w_i}$ (one-hot encoded).

"This"	0	1	0	0
"is"	0	0	1	0
$\sum_i \boldsymbol{e}_{w_i}$	0	1	1	0

4.5 Extreme Classification

Classification with many classes is called *extreme classification.*

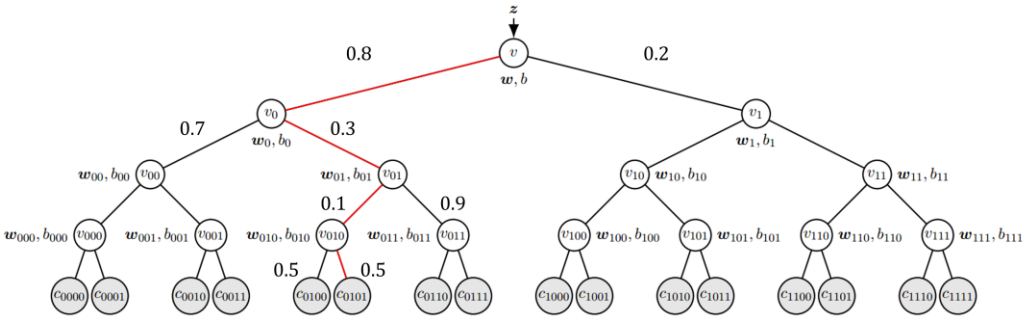
Training with a softmax layers is very costly due to computing the normalization term $\exp(\langle \boldsymbol{w}_{c'}, \boldsymbol{z} \rangle)$ for every class (cf. *2.4.6*, *2.4.7 and 4.3*). Hence, the *plain softmax* is commonly avoided.

4.5.1 Hierarchical Softmax

The hierarchical softmax computes it in multiple steps instead of a single one with the goal of reducing cost. For this, we arrange classes in a decision tree like manner.

- Input is $\boldsymbol{z}$, the ouput of the layer before the softmax
- Output is $\boldsymbol{p}$, the probability for each class

The leaves of our trees are classes, interior vertices are decision points, and each possible choice is associated with a probability. The probability of each class is obtained by computing the product along the corresponding path.



To model the probability distribution over the children of each interior vertex v_i, we still use a plain softmax. We have to store one weight matrix $\boldsymbol{w}$ and bias vector $\boldsymbol{b}$ per v_i. The probability distribution over the children of any v_i is $S(\boldsymbol{W}_i^\top \boldsymbol{z} + \boldsymbol{b}_i)$. Note that $\boldsymbol{z}$ is used at each v_i → influences all decisions. For example, the probability p_y of leaf $y = 0101$ is:

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

→ only depends on 4 (out of 15) weight vectors and bias terms! All other weight vectors / biases have zero gradient during backprop.

For balanced binary trees with C classes, we generally access $\log_2 C$ weight vectors & biases.

Discussion

- Produces good predictions if the classes in the "right" subtree are easy to discriminate from the classes of the "wrong" subtrees → similar classes are close together
- The choice of tree matters for training speed
  - Flat: as slow as softmax
  - Balanced: logarithmic cost
  - Fastest: *Huffman* tree → minimizes expected path lengths
- No/limited runtime improvement during prediction as we still need to compute all probabilities to get distribution over labels

4.5.2 Sampling-Based Approximate Softmax

During training, we approximate the softmax function in the last layer. The log-likelihood for a single example $(\boldsymbol{x}, y)$ is:

$$\ell = \log p_y = \log S(\boldsymbol{W}^\top \boldsymbol{z} + \boldsymbol{b})_y$$

$$= \log \frac{\exp(\eta_y)}{\sum_c \exp(\eta_c)} = \eta_y - \log \sum_c \exp(\eta_c)$$

The respective gradient is:

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

where $\nabla_{\boldsymbol{\theta}} \eta_y$ is the *positive reinforcement* for the current class y,

and $E_{c \sim \mathrm{Cat}(\boldsymbol{p})}[\nabla_{\boldsymbol{\theta}} \eta_c]$ is the (weighted) *negative reinforcement* for other classes

where $\boldsymbol{z}$ depends on $\boldsymbol{x}$ and parameters $\boldsymbol{\theta}$.

We want to maximize ℓ during training. Sampling-based approaches approximate the *negative reinforcement* term. But we can't sample from $\mathrm{Cat}(\boldsymbol{p})$ → computing $\boldsymbol{p}$ is to avoid in the first place!

Instead, we perform *Negative sampling* → sample *uniformly* instead of using $\boldsymbol{p}$.

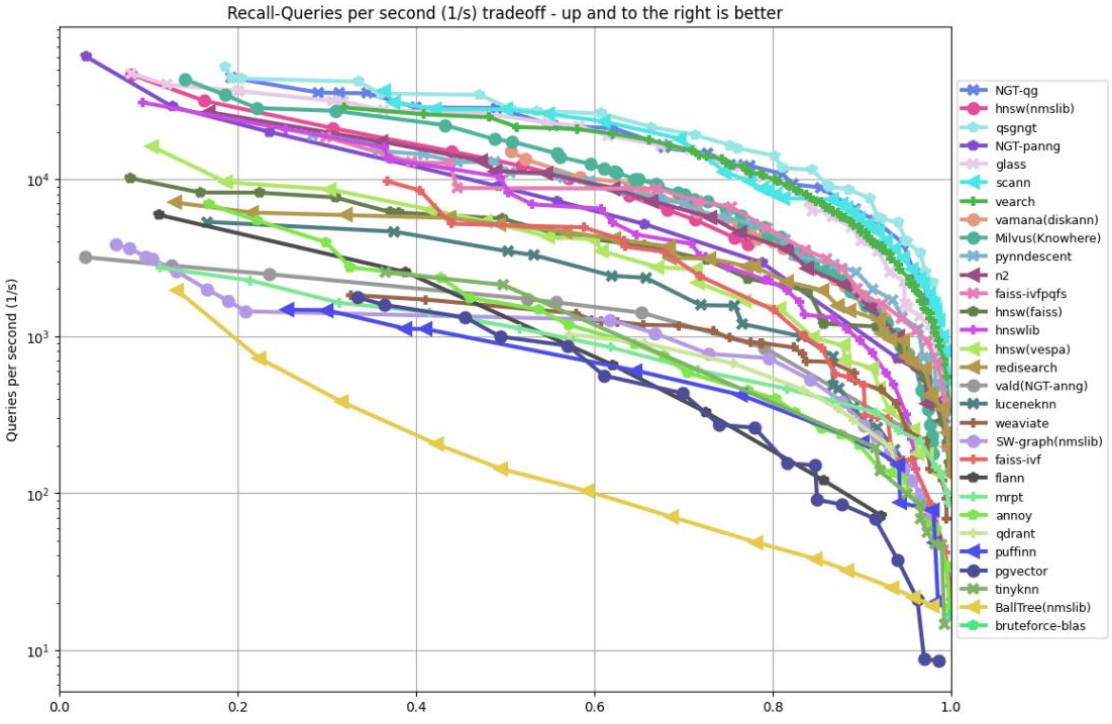
→ we could also perform *Adaptive Importance Sampling* or *Noise Contrastive Estimation*

4.5.3 Maximum Inner Product Search

For prediction tasks we often only care about the most-likely prediction and not necessarily its probability. Since the most-likely prediction is the one with the highest softmax score $\eta_c = \boldsymbol{w}_c^\top \boldsymbol{z}$.

Hence, in *maximum inner product search*: Given an indexed set of vectors $\mathcal{W} = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_C\}$ upfront as well as another vector $\boldsymbol{z}$, output the top-k inner-products (of vectors in $\mathcal{W}$ with z).

- Slow to do exactly, but many fast approximate methods exist
- Special case of *approximate nearest neighbor search*



5 Sequence Models

Sequence models operate on sequential data. *Sequential data* is any data with meaningful order:

- Time Series
- Natural Language
- Audio Signals
- Images
- Videos
- Processes

Data that is *not* naturally sequential is sometimes *sequentialized*, to use a sequence model on it. Formally, sequential data consist of:

An *input sequence* $\boldsymbol{x} \in \mathcal{X}^*$, where such a sequence consists of $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(\tau)}$ of length τ. This length may differ for different inputs $\boldsymbol{x}$, e.g. document length for classification. An element of the sequence $\boldsymbol{x}^{(t)} \in \mathcal{X}$ is the input at *times step* $t \in \mathbb{N}^+$.

An output sequence $y \in \mathcal{Y}^*$, where such a sequence consists of $\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, \ldots, \boldsymbol{y}^{(\tau_{out})}$ of length τ_{out}. An element of the output sequence is $\boldsymbol{y}^{(t)} \in \mathcal{Y}$. τ_{out} may or may not depend on input $\boldsymbol{x}$. τ_{out} may be *deterministic* or *random* (for a 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

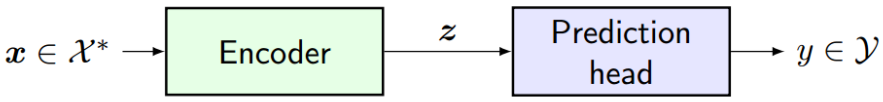
Examples from time series domain, where $\boldsymbol{x}$ is a time series

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

Types of **Sequence Models**

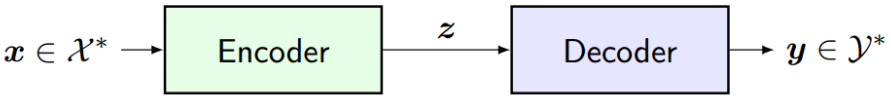
Encoder-only models compute useful representations/embeddings

- E.g., BERT for text data



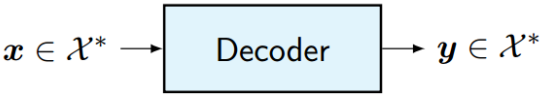
Encoder-decoder models add a decoder to generate sequences

- E.g., T5 for sequence-to-sequence models



Decoder-only models drop the encoder (and use decoder instead)

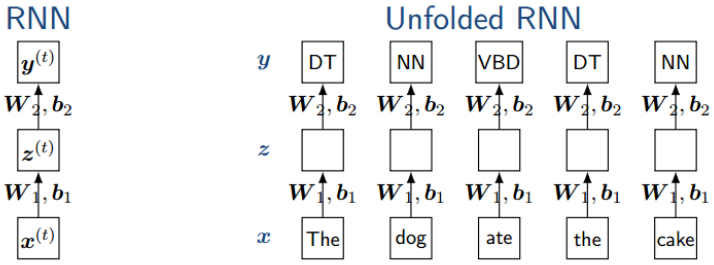
- E.g., GPT-1/2/3/4 for language modelling



5.1 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are a type of neural network designed for handling sequential data. Unlike traditional feedforward neural networks, RNNs don't have a fixed number of inputs and outputs. Instead, they can be thought of as a blueprint for constructing feedforward neural networks → a process known as *unfolding* or *unrolling* the RNN.

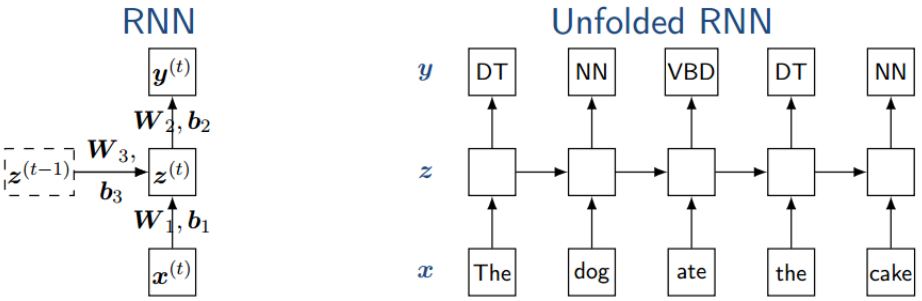
Key features of RNNs include *template modeling*, where a feedforward neural network is constructed based on a *template* describing computations at each time step, and *parameter sharing*, where weights associated with the template are shared across different time steps in the unrolled RNN. A *very simple* RNN example:



- Each node/vertex represents a subnetwork
- Each arrow/edge represents directed connections between corresponding subnetworks: from output neurons to input neurons, but *not necessarily fully connected*
- Specific details of the architecture are simplified or abstracted

5.1.1 Unidirectional RNNs

RNNs are *template models*. This means that the network is constructed by repeating the same *template* (left subfigure) for every time step → *unfolding*. Parameters are *shared* across time steps!



In a *unidirectional* RNN, connections between time steps go from left to right (right subfigure). The networks can pass along information to subsequent time steps → the hidden states are $\boldsymbol{z}^{(t)}$'s.

The hidden state representation serves two purposes:

- For prediction: providing good features to output layer $\boldsymbol{z} \rightarrow \boldsymbol{y}$ (at current timestep)
- For Memory: provide useful information to subsequent time step(s) $\boldsymbol{z}^{(t)} \rightarrow \boldsymbol{z}^{(t+1)}$

The unfolded network is called *deep in time*; even when the template is shallow.

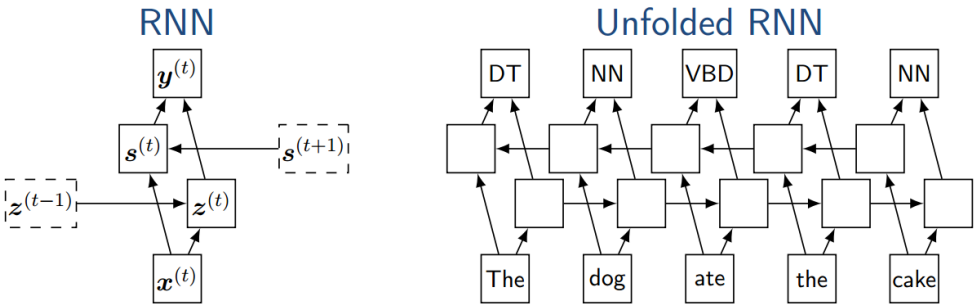
The RNN uses parameter sharing; this corresponds to an inductive bias: *position independence*

- Computation performed at each time step does not depend on position, but different data
- The selection of useful input features does not depend on position
- The selection of useful information to pass along does not depend on position

Advantages	Disadvantages
• No need to unfold the network during prediction	• Can be slow since computataion cannot be parallelized over time
• Can predict $\boldsymbol{y}^{(t)}$ as soon as we see $\boldsymbol{x}^{(t)}$ → real time prediction	• Valuable data in subsequent time steps ignored (unidirectional

5.1.2 Bidirectional RNNs

Bidirectional RNNs also have backward connections. Information about past *and* future gets captured in hidden units. Every output depends on every input → better predictions!



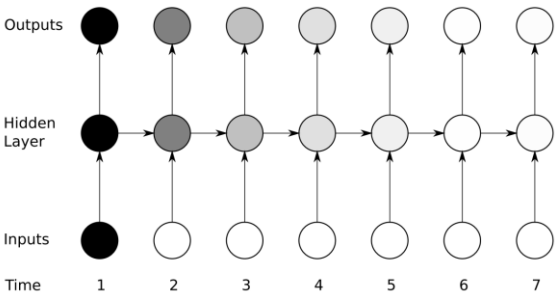
The main drawbacks are that no real-time predictions are possible as well as being more resource-intensive during prediction.

5.1.3 Vanishing Gradients

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

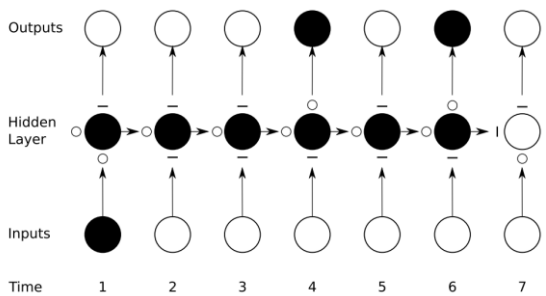


RNNs are deep in time, thus: Gradient of past input $x^{(t_{past})}$ w.r.t. to current output $y^{(t)}$ quickly decreases over time $(t - t_{past})$. The network becomes *insensitive* to changes in past input and may even forget past input (→ *catastrophic forgetting*). Common approaches to mitigate this:

- Use gating mechanisms in hidden layers (e.g., *LSTM*, *GRU*)
- Let the network directly access past information via *attention*

Preservation of gradient information by LSTM: For simplicity, all gates are either entirely open ('O') or closed ('—').

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



5.2 Long Short-Term Memory Networks (LSTMs)

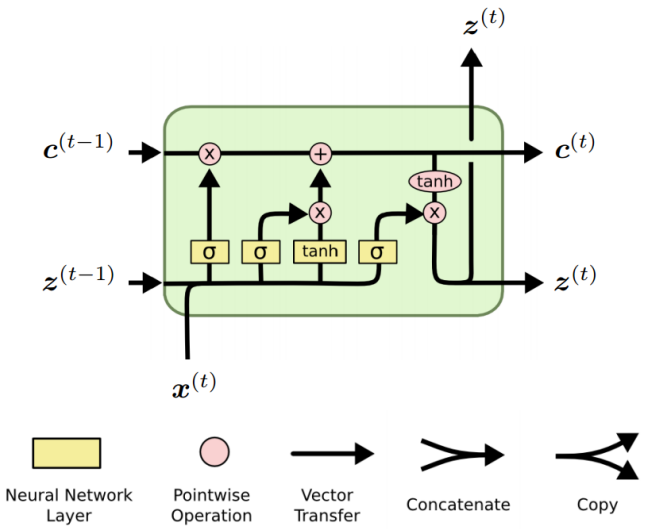
Long Short-Term Memory Networks (LSTMs) are a (sub-) architecture for the hidden layer of an RNN. Key ideas:

A D-dimensional LSTM unit (or 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:

- The *forget gate* controls to what extent cell state is retained
- The *input gate* controls to what extent cell state is updated
- The *output gate* controls outputs



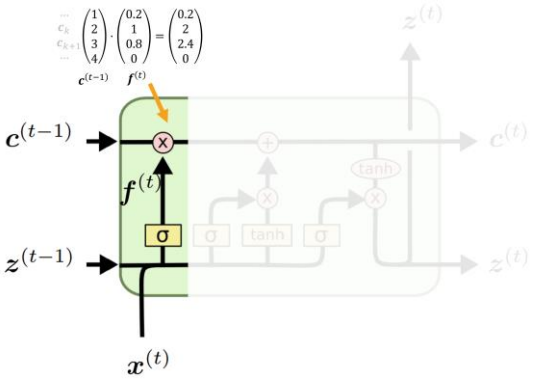
All gates are controlled by the unit's cell state $\boldsymbol{c}$. Again, hidden representations $\boldsymbol{z}$ serve multiple purposes. LSTMs are carefully designed such that gradient information can flow backwards:

- cell state $\boldsymbol{c}$ updated, but not replaced/recomputed
- gradient cannot explode, but can be kept high (when $\boldsymbol{f}^{(t)} \approx \mathbf{1}$)

5.2.1 Forget Gate

The *forget gate* controls to what extent a state is kept $\boldsymbol{f}^{(t)} \in [0,1]^D$:

- When $\boldsymbol{f}_k^{(t)} = 1$, the gate is *open* and element $\boldsymbol{c}_k$ passes *unmodified.*
- When $\boldsymbol{f}_k^{(t)} = 0$, the gate is *closed* and element $\boldsymbol{c}_k$ is *zeroed out.*



This decision is made based on the input $\boldsymbol{x}^{(t)}$ and the hidden state $\boldsymbol{z}^{(t-1)}$ via:

$$\boldsymbol{f}^{(t)} = \sigma\left(\boldsymbol{W}_f^\top \begin{pmatrix} \boldsymbol{z}^{(t-1)} \\ \boldsymbol{x}^{(t)} \end{pmatrix} + \boldsymbol{b}_f\right)$$

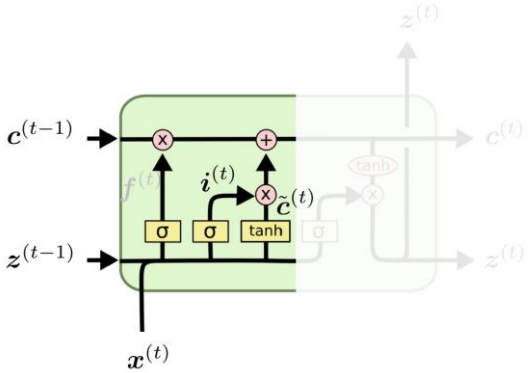
Where the weights $\boldsymbol{W}_f^\top$ and biases $\boldsymbol{b}_f$ are learned during training.

5.2.2 Input Gate

The *input gate* decides to what extent the cell state gets updated. The *proposed update* is $\tilde{\boldsymbol{c}}^{(t)} \in [-1,1]^D$, which gets scaled by factor $\boldsymbol{i}^{(t)} \in [0,1]^D$.

Both their parameters are learned during training.

This *additive* nature of the update allows the gradient to flow better, mitigating the VGP.

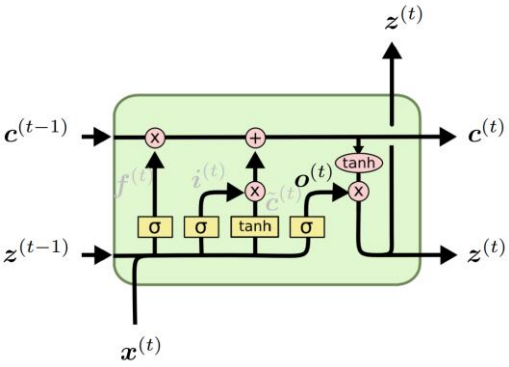


Example updates:

$\boldsymbol{f}_k^{(t)} = 1, \boldsymbol{i}_k^{(t)} = 0 \rightarrow$ old value fully retained $\quad \boldsymbol{f}_k^{(t)} = 0, \boldsymbol{i}_k^{(t)} = 1 \rightarrow$ fully new value

$\boldsymbol{f}_k^{(t)} = 1, \boldsymbol{i}_k^{(t)} = 1 \rightarrow$ updates value $\quad \boldsymbol{f}_k^{(t)} = 0, \boldsymbol{i}_k^{(t)} = 0 \rightarrow$ fully clears value

5.2.3 Output Gate



The *output gate* decides what to output. The output value can be considered a squashed & filtered version of new cell state $\boldsymbol{c}^{(t)} \rightarrow$ the *new hidden state* $\boldsymbol{z}^{(t)}$

The cell state gets pushed into the $[-1,1]$ range by the tanh *function* (not layer), with the extent of this adjustment controlled by $\boldsymbol{o}^{(t)} \in [0,1]^D$.

Other variants include:

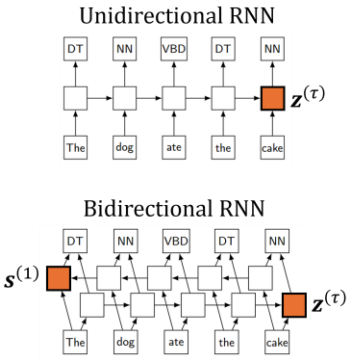
- *Peephole connections*: gate layers (sigmoids) take cell state as additional input
- *Coupled input/forget gates*: take $\boldsymbol{i}^{(t)} = \mathbf{1} - \boldsymbol{f}^{(t)}$
- *Gated recurrent units* (*GRU*): additionally combine cell state and hidden state (pushes filtering to obtain hidden state to next time step)

5.3 Discussion

So far, we have focused on RNN *encoders* only.

They provide a fixed-dimensional sequence representation → known as a *thought vector.*

- For unidirectional RNNs: the last state $\boldsymbol{z}^{(\tau)}$
- For bidirectional RNNs: the first state $\boldsymbol{s}^{(1)}$ in backwards direction and $\boldsymbol{z}^{(\tau)}$



These representations are useful for sequence-level tasks, i.e. sequence classification.

They also provide contextualized representations $\boldsymbol{z}^{(t)}$ for each input $\boldsymbol{x}^{(t)}$. Meaning that the representation $\boldsymbol{z}^{(t)}$ depends on the surrounding inputs. This is useful for element-level tasks, such as sequence labeling.

5.4 Deep Autoregressive (DAR) Models

5.4.1 Deep Generative Models

Recall that *generative models* usually also model the distribution of inputs $p(\boldsymbol{x})$ for $\boldsymbol{x} \in \mathcal{X}$.

A *conditional generative model* models the conditional distribution $p(\boldsymbol{x}|\boldsymbol{c})$. Where we may think of $\boldsymbol{c} \in \mathcal{C}$ as an input, a condition, a prompt, or context. In contrast to discriminative models, there usually exist multiple "correct" outputs.

Deep generative models use *deep neural networks* to define a generative model for complex data distributions (e.g., text, audio, image, graph, …).

Common tasks of interest include:

- Train a generative model $p(\boldsymbol{x})$ from samples of data distribution $p_D(\boldsymbol{x})$
- Sample from $p(\boldsymbol{x})$
- Determine the top-k highest-probability outputs
- Given $\boldsymbol{x}$, compute density $p(\boldsymbol{x})$
- Sometimes: obtain latent codes $\boldsymbol{z}$ for a given data point $\boldsymbol{x}$

Goals of sampling data:

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

We now look at deep generative models for sequence data, meaning that the distribution we care about, $p(\boldsymbol{x}) \in \mathcal{X}^*$, is over sequences for elements from $\mathcal{X}$. For example:

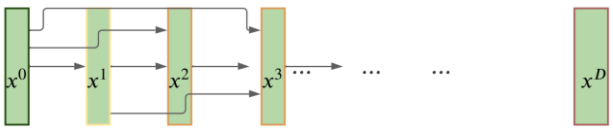
- a language model, where $\mathcal{X}$ is a discrete set of characters/tokens.
- A time series model, where $\mathcal{X} = \mathbb{R}$ (univariate) or $\mathcal{X} = \mathbb{R}^D$ (multivariate)

and then at conditional generative models.

5.4.2 Autoregressive Generative Models

Autoregressive models decompose the joint distribution into next-element distributions using the product rule. The parameters of these distributions depend on past outputs $x_{1:t-1}$:

$$p(x_{1:\tau}) = p(x_1)p(x_2|x_1)p(x_3|x_1,x_2)\cdots$$
$$= \prod_{t=1}^{\tau} p(\underbrace{x_t}_{\text{next element}} | \underbrace{x_{1:t-1}}_{\text{past elements}})$$



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.

To generate, sample repeatedly from $p(x_t\|x_{1:t-1})$	Example
1. Generate x_1 by sampling from $p(x_1)$	*The*
2. Generate x_2 by sampling from $p(x_2\|x_1)$	*dog*
3. Generate x_3 by sampling from $p(x_3\|x_{1:2})$	*ate*
4. Generate x_4 by sampling from $p(x_4\|x_{1:3})$	*the*
5. Generate x_5 by sampling from $p(x_5\|x_{1:4})$	*cake*
6. …	

Assumptions

In principle, any distribution $p(\boldsymbol{x})$ can be modeled. But the next-token distributions become more and more complex. Hence, we need to make assumptions to reduce complexity. For example, a *Markov Chain of order k* (→ window/context size k) makes the *Markov assumption*:

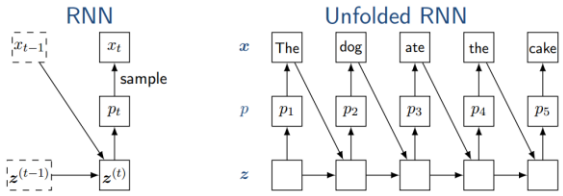
$$p(x_t|x_{1:t-1}) = p(x_t|x_{t-k:t-1}) = p(x_{k+1} = x_t|x_{1:k} = x_{t-k:t-1})$$

Here, we only look at k most recent outputs and use *the same distribution p* across these timesteps. In other words, knowing the last k states already contain all the information needed to predict the current state $k+1$, and the history before that is redundant given the Markov assumption. For binary data this requires 2^k parameters in total.

Using an RNN Decoder

We can use an RNN decoder (no input, only hidden states $\boldsymbol{z}$) to specify the next-element distribution.

The RNN has to be unidirectional as for autoregressive models we cannot access future outputs. Furthermore, the outputs $\boldsymbol{p}$ have to be stochastic, defining a probability distribution so we can sample $\boldsymbol{x}$ from it.



$$p(x_t|x_{1:t-1}) = p(x_t|x_{t-1}, z_{t-1})$$

Generally, $\boldsymbol{z}_t = f_{\boldsymbol{\theta}}(\boldsymbol{z}_{t-1}, \boldsymbol{x}_{t-1})$ is computed by the past hidden state $\boldsymbol{z}_{t-1}$ and resp. output $\boldsymbol{x}_{t-1}$. The output $x_t \sim p_{\boldsymbol{\theta}}(x_t|\boldsymbol{z}_t)$ is then sampled from the hidden state. The parameters $\boldsymbol{\theta}$ determine how exactly these operations will be done. For example, when using LSTM cells, then $\boldsymbol{z}_t = (\boldsymbol{c}_t, \boldsymbol{h}_t)$.

We can also provide additional information for each timestep, e.g. for temporal data: date, weekday, time, etc., or *attention* to prior outputs, or a condition → *conditional generative models*.

Adding Input: DAR Models as Conditional Generative Models

Deep autoregressive models can also be used as conditional generative models. Recall that we model $p(\boldsymbol{x}|\boldsymbol{c})$, where $\mathbf{c}$ is an input. The general approach is to use *input-dependent next-element distributions* of form:

$$p(x_t|x_{1:t-1}, \boldsymbol{c})$$

There are two key approaches to do this:

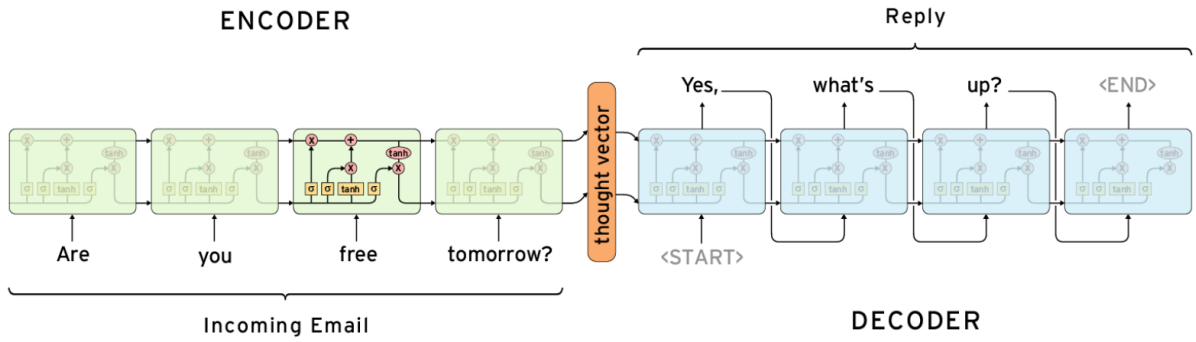
- Encoder-decoder models
- Decoder-only models

5.4.3 Encoder-Decoder Models

Encoder-decoder models generally use an encoder to compute the representation $\boldsymbol{z}$ of input $\boldsymbol{c}$. They then use $\boldsymbol{z}$ to condition the decoder on this representation from the encoder, i.e.:

$$p_{\boldsymbol{\theta}}(x_t|x_{1:t-1}, \boldsymbol{z})$$

Generally, the input and output may be of different types (A-to-B models) → e.g. text-to-image. In an *encoder-decoder RNN* this is not the case → in- and output are sequences. *Encoder-decoder RNNs* feed the thought vector of encoder as the initial hidden state into the decoder. Note that the encoder can be bi-directional.



5.4.4 Decoder-Only Models

Decoder-Only Models treat this condition on $\boldsymbol{c}$ as a "prior output":

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

where $\boldsymbol{c} \parallel \boldsymbol{x}$ is the concatenation of those sequences. This is equal to the probability of outputting $\boldsymbol{x}$ after $\boldsymbol{c}$ has already been generated. Logically, input and output must be of the same type.

Example: Time Series

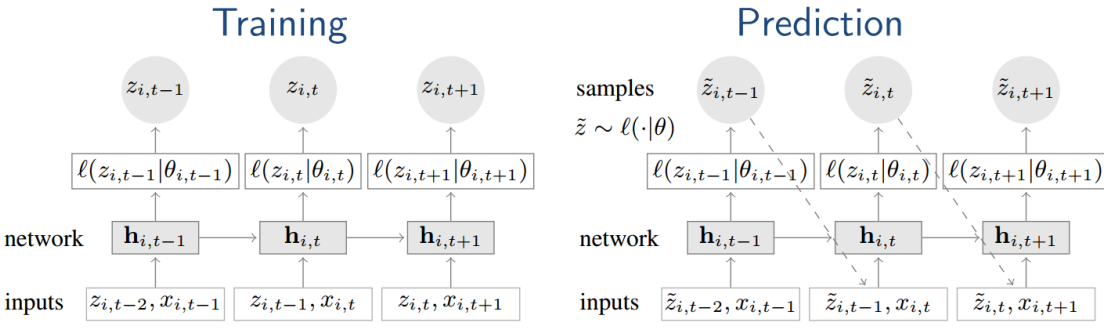
- $\boldsymbol{c} = observed\ past\ values$
- $\boldsymbol{x} = future\ values$

Example: LLM Prompts

- $\boldsymbol{c} = (Are, you, free, tomorrow?, END\_OF\_INPUT)$
- $\boldsymbol{x} = (Yes, what's, up?)$
- $\boldsymbol{c} \parallel \boldsymbol{x} = (Are, you, free, tomorrow?, END\_OF\_INPUT, Yes, what's, up?)$

Example: DeepAR (Amazon)

Probabilistic forecasting based on RNNs with stochastic units → e.g. for inventory management. Focuses on scenarios with many related/correlated time series (energy consumption of individual households, demand of products).



It learns a generative model for all of the time series jointly. Allows to fit more complex models, little feature engineering.

5.4.5 Discussion

Encoder-Decoder Models

- In- & output may be of different types
- Input can be processed bi-directionally
- No info-sharing between in- & output
- Decision on in- /output during training

Decoder-Only Models

- Input and output are of same type
- Completely uni-directional
- Info-sharing between in- & output
- No need to decide on what data is used as input and output during training

Intermediate: *Non-Causal Decoders-Only Models*

- Bi-directional on input, uni-directional on output → same as Encoder-Decoder Models
- Uses the same parameters in encoder and decoder → information sharing

5.4.6 Training DAR Models

The parameters $\boldsymbol{\theta}$ of deep autoregressive models can be determined using training data. For training example $\boldsymbol{x}^*$, its next-element probability is obtained by

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}^*) = \prod_t p_{\boldsymbol{\theta}}(x_t^* | x_{1:t-1}^*)$$

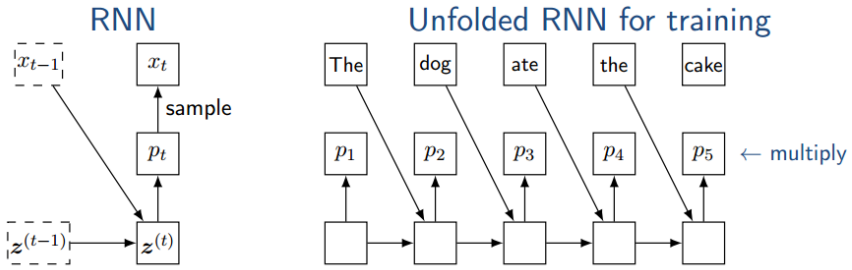
where quantities $p_{\boldsymbol{\theta}}(x_t^* | x_{1:t-1}^*)$ correspond to the probability that model generates correct element x_t^* at position t, given that it generated all previous elements $x_{1:t-1}^*$ correctly.

E.g., when using categorical data, we may use log loss or MLE, i.e. minimize:

$$-\log p_{\boldsymbol{\theta}}(\boldsymbol{x}^* | \boldsymbol{\theta}) = -\sum_t \log p_{\boldsymbol{\theta}}(x_t^* | x_{1:t-1}^*)$$

Observe that the model is trained for next-element prediction, which is sometimes referred to as the (full) *language modelling objective*.

For RNNs, we compute $p_{\boldsymbol{\theta}}(\boldsymbol{x}^*)$, the probability of the input sequence in the forward pass, by unrolling the RNN with training data instead of sampling:



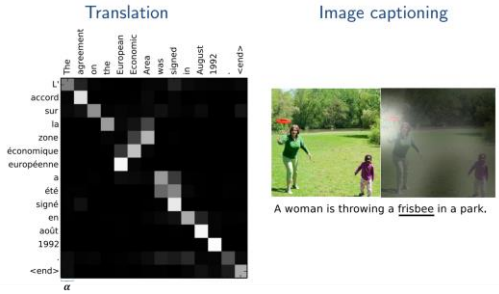
In *teacher forcing*, during the training phase, instead of feeding the network its own output from the previous time step (as it does during generation), we provide the true target output from the training data. This means that at each step of the training process, the input to the model is the actual correct output at the previous step, rather than the predicted output from the model itself. This helps stabilize and expedite training by providing the network with accurate context during the learning process.

5.5 Attention

Recall the encoder-decoder architecture from *5.4.3*: No matter the input length that the encoder gets, the decoder always receives relevant information in a continuous, fixed-size vector representation (*thought vector*). The decoders output is then solely based on this thought vector.
💡 The attention mechanism essentially allows the decoder to access the encoder's input again.

📄 Attention groups multiple inputs into a *fixed-length* representation. In this representation the inputs are *unordered*. The number of inputs may vary but attention still produces a fixed-length representation. This grouping is:

- Simple, e.g. a weighted average of the input
- Often focused on a small subset of the inputs
- Dynamic → depends on the network state and values of inputs



The decoder RNN generates token-by-token. In order to produce the first output, in addition to the thought vector, it will look at the inputs and *attend to* the relevant parts of the input sequence, weighting them according to their importance in context, before generating the initial token.

💡 Attention "provides access" to all inputs (→ soft memory). This is via arbitrary-length representations, i.e., longer inputs (to encoder) → larger representation (to decoder).

Attention is per se general, meaning it works on arbitrary multisets and not necessarily sequence data. Positional information can be added: $\{(1, The), (2, dog), (3, ate), (\dots)\}$.

5.5.1 Associative Memory

We can view attention as a form of accessing the input set via associative memory. We can retrieve a *value* via a *query*; e.g. $query = 3$ would output "ate". In essence, a key k_i describes input i, a value v_i represents input i and a query q_i *expresses* what is *considered relevant*.

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

💡 Attention can be viewed as a *soft form* of associative memory, where queries only describe "roughly" what we want to retrieve. We use neural representations to represent keys, values, and query → all these are learned.

- *Keys* $\boldsymbol{k}_1, \dots, \boldsymbol{k}_\tau \in \mathbb{R}^{d_K}$ are learned *descriptions* of the input (instead of addresses)
- *Values* $\boldsymbol{v}_1, \dots, \boldsymbol{v}_\tau \in \mathbb{R}^{d_V}$ learned *representations* of the input elements (instead of elements)
- *Query* $\boldsymbol{q} \in \mathbb{R}^{d_K}$ is a learned *description* of the *information need* (instead of an "address")

where dimensionalities of query/keys d_K and values d_V may differ.

5.5.2 Attention Model & Context Vector

To answer a query, we use an ***attention model*** $a(\boldsymbol{q}, \boldsymbol{k})$ that measures the compatibility of each input key $\boldsymbol{k}_i$ to the query $\boldsymbol{q}$ via an attention score:

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

his forms an attention score vector $\boldsymbol{e} \in \mathbb{R}^{\tau}$, where higher scores belong to more relevant keys. E.g. we could use the dot product $a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{k}^{\top}\boldsymbol{q}$ as such model, or also a small MLP for example.

We then compute a ***context vector*** $\boldsymbol{c} \in \mathbb{R}^{d_V}$ (has same shape as a *value*) by weighting:

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

where α_i are query-dependent *attention weights*. E.g.:

- *Soft Attention*: $\boldsymbol{\alpha} = S(\boldsymbol{e})$
- *Hard Attention*: $\alpha_i = 1$ for largest score e_i, else $0 \rightarrow$ one-hot
- *Linear Attention*: $\boldsymbol{\alpha} = \boldsymbol{e}$

Hence, we combine different values $\boldsymbol{v}_i$ via a weighted sum to form the context vector $\boldsymbol{c}$.

Example: Dot-Product (Soft) 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)	$(0 \quad 0 \quad 0)$	$(0 \quad 0 \quad 0 \quad 0 \quad 0)$	$(0.2 \quad 0.2 \quad 0.2 \quad 0.2 \quad 0.2)$

Example (i):

Query for Nouns

$\boldsymbol{q} = (5 \quad 0 \quad 0)$

$$\sum_i \alpha_i \boldsymbol{v}_i = \begin{matrix} \textit{The} & (\cdots & \cdots & \cdots) * 0 \\ \textit{dog} & +(\cdots & \cdots & \cdots) * 0.5 \\ \textit{ate} & +(\cdots & \cdots & \cdots) * 0 \\ \textit{the} & +(\cdots & \cdots & \cdots) * 0 \\ \textit{cake} & +(\cdots & \cdots & \cdots) * 0.5 \end{matrix} = (\cdots \quad \cdots \quad \cdots) = \boldsymbol{c}$$

$\boldsymbol{v}_i$ α_i

What the values exactly are depends on the architecture (e.g. $\rightarrow$ hidden states of an RNN).

5.5.3 Dot-Product Attention as a Layer

Stacking the keys and values into matrices gives us: $\boldsymbol{q} \in \mathbb{R}^{d_K}$, $\boldsymbol{K} \in \mathbb{R}^{\tau \times d_K}$, $\boldsymbol{V} \in \mathbb{R}^{\tau \times d_V}$

The dot-product attention scores are: $\boldsymbol{e} = \boldsymbol{K}\boldsymbol{q} \in \mathbb{R}^{\tau}$

Soft attention: $\boldsymbol{\alpha} = S(\boldsymbol{K}\boldsymbol{q})$

In order to obtain the context vectors, we need to compute the weighted averages:

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

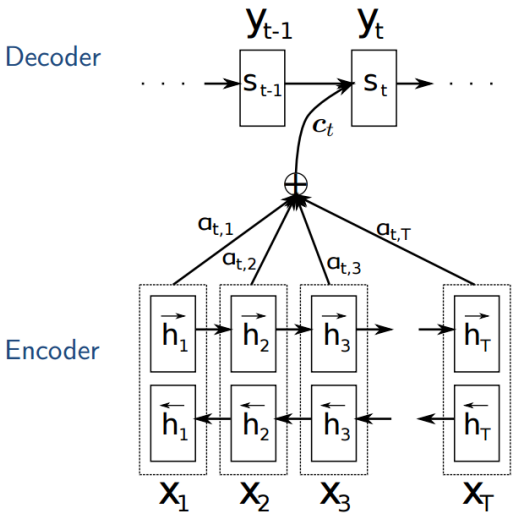
For large values d_K, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counteract this effect, we can scale the dot products by $\sqrt{d_K}$: $\mathrm{ScaledDPA}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = S\left(\boldsymbol{K}\boldsymbol{q}/\sqrt{d_K}\right)^{\top}\boldsymbol{V} \in \mathbb{R}^{d_V}$

5.5.4 Attention for Encoder-Decoder RNNs

The encoder RNN for machine translation from Bahdanau and Cho (2015) is a bidirectional architecture.

At each timestep t, the decoder RNN utilizes the preceding state $\boldsymbol{s}_{t-1}$ to retain a memory of the previously generated output, and it incorporates the context vector $\boldsymbol{c}_t$, which encapsulates relevant information from the input sequence, tailored to the current timestep t.

Thus, the decoder's previous state serves as a query ($\boldsymbol{q} = \boldsymbol{s}_{t-1}$) to extract time-dependent information, with the encoder states $\boldsymbol{h}_t$ functioning as both keys and values ($\boldsymbol{k}_t = \boldsymbol{h}_t = \boldsymbol{v}_t$).



💡 Training such a network encourages the hidden state $\boldsymbol{h}_t$ to capture information that is useful as a summary of what has been output from the decoder, but also be useful to query $\boldsymbol{q}$ the input.

This integration of the context vector enriches the decoder's understanding of the input sequence, ensuring that the translation process adapts contextually to the evolving output generation.

❓ So far, we discussed attention in the context of an RNN, but do we actually need RNNs?

5.6 Transformers

💡 RNNs are slow to train due to their sequential nature, but transformer architectures make use of non-sequential, easily parallelizable attention which reduces the time to train the model.

> Recall, that τ refers to the *length of* a sequence $\boldsymbol{x} \in \mathcal{X}^*$ consisting of $\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(\tau)}$ elements.

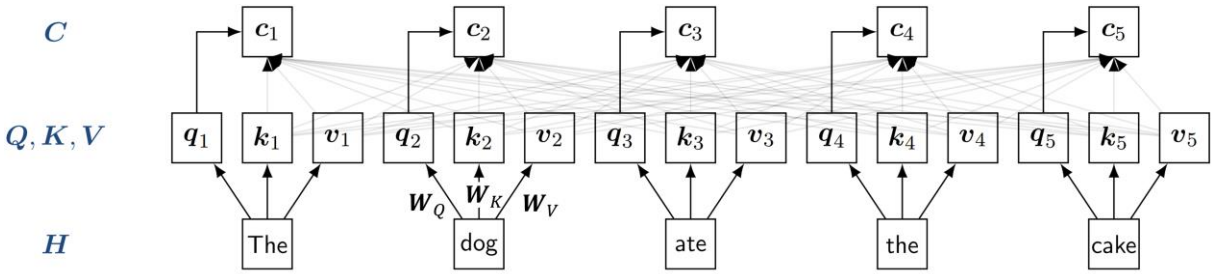
5.6.1 Self-Attention Layer

Self-attention refers to a type of layer where:

- inputs are feature representations $\boldsymbol{h}_1, \dots, \boldsymbol{h}_\tau \in \mathbb{R}^d$
- outputs are higher-level feature representations $\boldsymbol{c}_1, \dots, \boldsymbol{c}_\tau \in \mathbb{R}^{d_V}$
- sequence length τ may vary between inputs

Per input i, we have one query $\boldsymbol{q}_i$, a key $\boldsymbol{k}_i$ and a value $\boldsymbol{v}_i$. They are computed from $\boldsymbol{h}_i \rightarrow$ motivating the name *self-attention*. An output $\boldsymbol{c}_i = \text{Attention}(\boldsymbol{q}_i, \boldsymbol{K}, \boldsymbol{V})$, where $\boldsymbol{K}$ and $\boldsymbol{V}$ are the sets of keys and values computed from the entire input sequence $\boldsymbol{H}$.

Then, to form the higher-level feature representation for a single $\boldsymbol{c}_i$, the compatibility of *each* key of the input sequence ($\rightarrow \boldsymbol{K}$) to the query $\boldsymbol{q}_i$ needs to be computed, to determine the relevance or importance of each other input in $\boldsymbol{H}$ to the current input $\boldsymbol{h}_i$ (cf. 5.5.2).



💡 Intuitively, the query $\boldsymbol{q}_i$ summarizes how $\boldsymbol{c}_i$ should look like, based on the relationship between the input $\boldsymbol{h}_i$ and all the other inputs in the sequence $\boldsymbol{H}$.

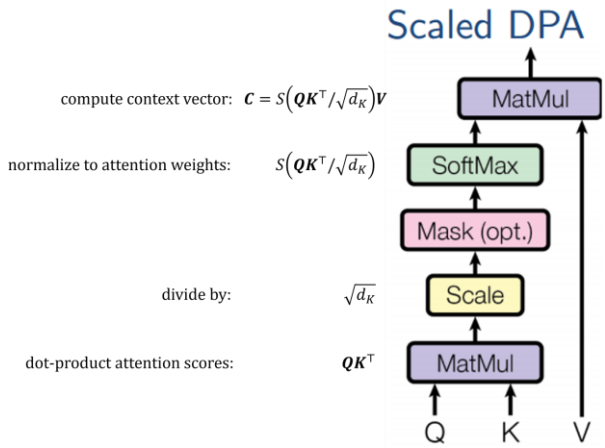
Practical Implication of Attention

Consider the word "bank" appearing in a sequence. Words may carry multiple different meanings; in this case it may refer to the financial institution or also to a riverbank. This disambiguation can be made by considering the context in which the word appears in.

The original word embedding thus may capture both meanings of the word. After performing attention and incorporating information from the other words in the sequence, the intended meaning in the given context can now be captured, yielding a more accurate representation.

Positional equivariance means that changing the order of words in a sequence doesn't change the output. This property is vital because self-attention operates on multisets, so rearranging inputs should yield the same results. It ensures the model captures relationships between words regardless of their positions, making it robust in understanding natural language text.

5.6.2 Single-Head Attention



Compact notation: $\boldsymbol{C} = \text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V})$

- $\boldsymbol{K} \in \mathbb{R}^{\tau \times d_K}$ contains keys
- $\boldsymbol{V} \in \mathbb{R}^{\tau \times d_V}$ contains values
- $\boldsymbol{Q} \in \mathbb{R}^{\tau \times d_K}$ contains queries as rows
- $\boldsymbol{C} \in \mathbb{R}^{\tau \times d_V}$ contains outputs as rows

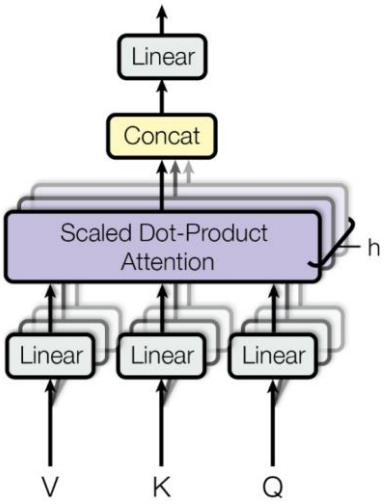
We define $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ by learning linear projections:

- $\boldsymbol{Q} = \boldsymbol{H}\boldsymbol{W}_Q$, where $\boldsymbol{W}_Q \in \mathbb{R}^{d \times d_K}$
- $\boldsymbol{K} = \boldsymbol{H}\boldsymbol{W}_K$, where $\boldsymbol{W}_K \in \mathbb{R}^{d \times d_K}$
- $\boldsymbol{V} = \boldsymbol{H}\boldsymbol{W}_V$, where $\boldsymbol{W}_V \in \mathbb{R}^{d \times d_V}$

plus their respective biases. Matrices $\boldsymbol{W}_Q, \boldsymbol{W}_K, \boldsymbol{W}_V$ are parameters/weights, while d_K and d_V are hyperparameters. Typically, we choose $d_K = d_V = d$.

5.6.3 Multi-Head Attention

In multi-head attention, instead of relying on a single attention mechanism to capture relationships between words in a sequence, multiple attention mechanisms (or *heads*) are used in parallel.



Each head $h \in \{1, \ldots, H\}$ has *its own set* of learnable parameters $\boldsymbol{W}_Q^{(h)}$ for queries, $\boldsymbol{W}_K^{(h)}$ for keys, and $\boldsymbol{W}_V^{(h)}$ for values. These parameters are learned during training and *allow each head to focus on different aspects* or relationships within the input sequence. By concatenating the outputs $\boldsymbol{C}^{(h)}$ of each attention head, the model can capture a richer representation of the relationships between words in the input sequence. The final representation is obtained after projecting the concatenation to the input space by using parameter $\boldsymbol{W}_O \in \mathbb{R}^{Hd_V \times d}$. The original transformer used: $d_K = d_V = d/H$, with $d = 512$ and $H = 8$.

This architecture is beneficial because the model can choose which representation subspaces to attend to in a position-dependent way.

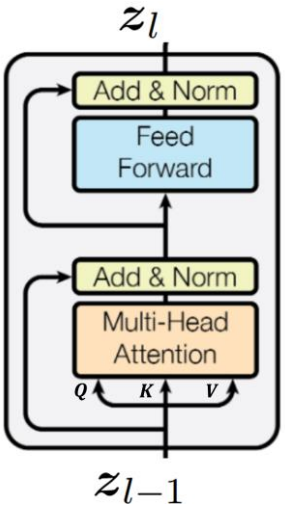
Computation and Path Lengths

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)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Computational complexity: Self-attention scales well with dim. d but not sequence length n!

5.6.4 Transformer Encoder Layer



Similarly to self-attention, as input we have lower-level element representations $\boldsymbol{z}^{l-1} = \{\boldsymbol{z}_1^{l-1}, \dots, \boldsymbol{z}_\tau^{l-1}\}$ and produce higher-level element representations $\boldsymbol{z}^{l} = \{\boldsymbol{z}_1^{l}, \dots, \boldsymbol{z}_\tau^{l}\}$. We now perform two steps:

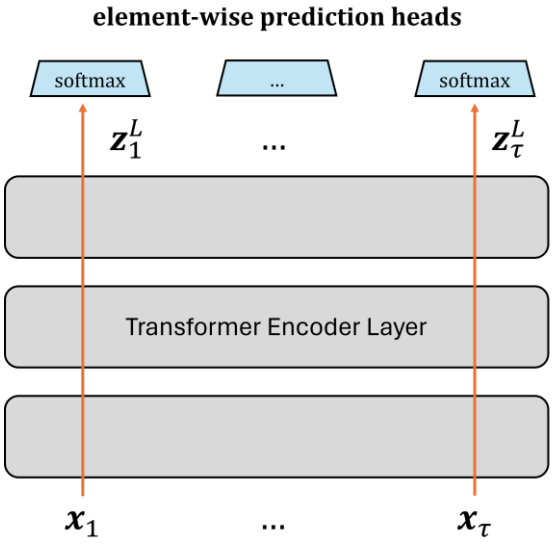
- Multi-head attention across elements $\boldsymbol{z}^{l-1} \rightarrow$ incorporate information from other elements of the sequence
- Local MLP for each element $\boldsymbol{c}_i \rightarrow$ update each element individually

Residual connections and layer normalization in between encourage both attention and MLP updates to not overwrite but update the element representation additively.

The layer makes use of *parameter sharing* in a position-independent way: re-using the same weight matrices for queries/keys/values across all input elements in a head (but not all heads).

The contextualized representations $\boldsymbol{z}_i^L$ of inputs elements $\boldsymbol{x}_i$ produced by the transformer are used as input representation for subsequent modules. Depending on the task we may:

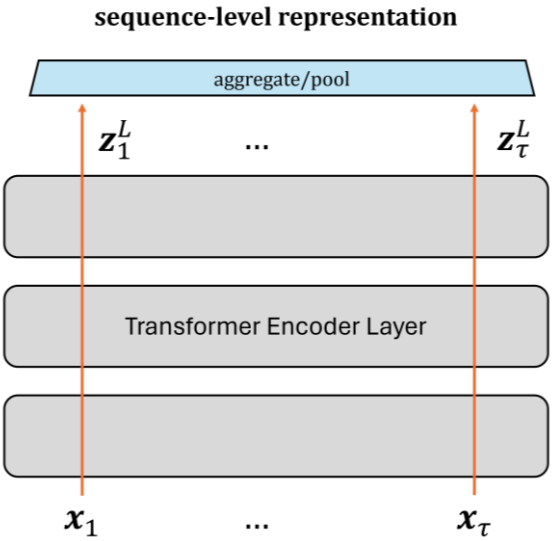
add an *element-wise prediction head* h on each $\boldsymbol{z}_i^L$ and train supervised using some element-level loss $L(h(\boldsymbol{z}_i^L), y_i)$, e.g. to perform part-of-speech tagging.



aggregate/*pool* the element-level representations (e.g. by sum, mean, attention) to obtain sequence-level representation:

$$\boldsymbol{s} = \text{aggregate}(\{\boldsymbol{z}_i^L\})$$

then add any prediction head h and train supervised using some loss $L(h(\boldsymbol{s}), y)$, e.g. to perform sentence classification

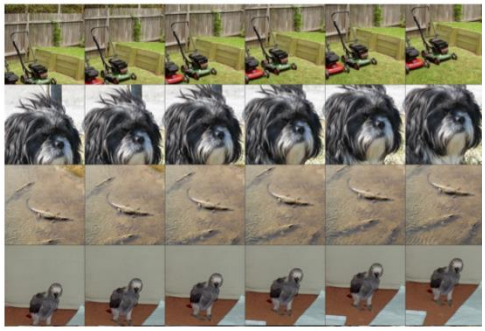


Alternatively, for sequence-level tasks, we can also use a special *classification token CLS*. The input to the encoder now is $CLS, x_1, \dots, x_\tau$. We add a prediction head only on $\boldsymbol{z}_{CLS}^L$ serving as a sequence/level representation. We train supervised using $L(h(\boldsymbol{z}_{CLS}^L), y)$.

5.7 …

Lecture #11 on thu. 2.5.

6 Convolutional Neural Networks (CNNs)



Convolutional neural networks (CNNs) are a family of neural networks tailored for processing stationary data.

Stationarity refers to a stochastic process where its unconditional joint probability distribution remains unchanged when shifted in time or space.

For example, in image classification tasks, the goal is to *classify patterns* consistently, regardless of their absolute position within the image.

Fully connected networks have some drawbacks that make them hard to use for these tasks:

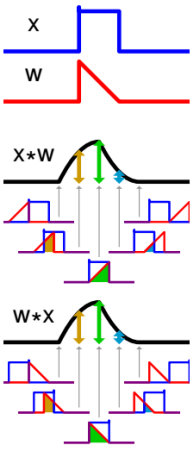
- Locality (neighborhood relationships) are not exploited
- No translation invariance → not robust against differently positioned objects

Main application areas are computer vision, natural language processing and signal processing.

6.1 Convolution

A convolution ($*$ operator) is linearly combining two functions x and w:

$$s(t) = (x * w)(t) = \int x(\tau)w(t-\tau)\ d\tau$$



In the context of CNNs,

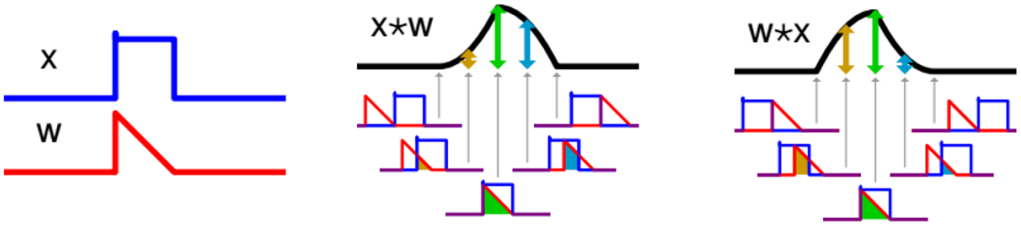
- we refer to x as *input*,
- w is called *kernel* or *filter*
- output s is referred to as *feature map* (single channel)

Intuitively, this computes a *weighted average* of inputs $x(\tau)$. At time t, the weight of $x(\tau)$ depends on its age $a = t - \tau$ via $w(a)$. We're shifting the kernel over the signal and then at each position calculating the weighted average.

Note: $*$ is commutative, hence the order of the arguments doesn't matter, the result stays the same.

6.1.1 Cross-Correlation

Cross-Correlation ($\star$ operator) is a closely related operation. The only difference being that the *kernel/filter* is not being flipped before getting shifted over the signal.



Cross-correlation is not commutative! The result will be different if order of arguments changed.

6.1.2 Discrete Convolution

For discrete t, for example images recorded on a discrete grid (e.g.: 1920×1080 pixels), we can use the discrete convolution operation:

$$s(t) = (x * w)(t) = \sum_{\tau=-\infty}^{\infty} x(\tau)w(t-\tau)$$

In ML, input x and kernel w are often represented by multidimensional arrays. E.g. $\boldsymbol{w} = (10 \quad 20 \quad 15)^{\mathsf{T}}$ may correspond to function with $w(-1) = 10$, $w(0) = 20$, $w(1) = 15$ and otherwise zero. The discrete convolution then only needs a finite sum to compute.

Different boundary conditions are possible, in DL frameworks often referred to as *padding:*

- *Reflection padding* mirrors data at the boundaries of the array
- *Valid convolution* only executes in area with full overlap (→ feature map reduces in size)

Choosing which one to use depends on the domain-specific application.

6.1.3 Discrete Cross-Correlation

In CNNs, cross-correlation is often used but referred to as *convolution without kernel flipping*.

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

Since the filter weights are learned, there is no conceptual difference though.

6.2 Convolution in CNNs

Convolutional neural networks (CNN) mostly operate on discrete, regular grid data, meaning that the distance between two timesteps (pixels, frames, etc.) is usually the same.

- 1D grid → sequential data (e.g., time series, text, audio, . . .)
- 2D grids (images)
- 3D grids (movies, CT scans) → referred to as *volume*

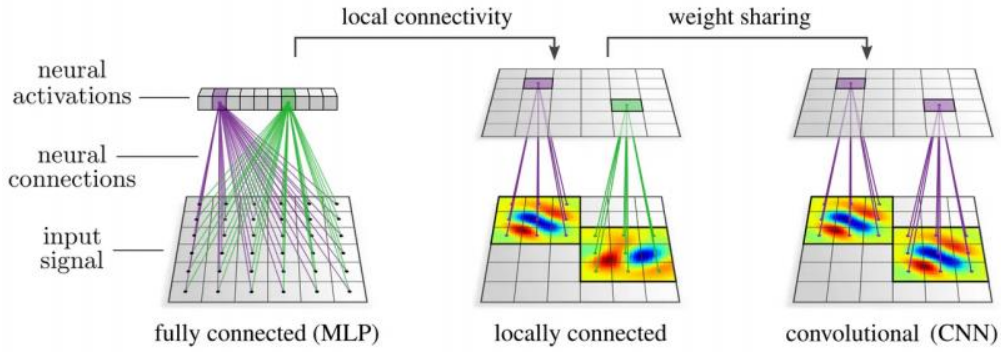
In this context, we can safely assume that neighboring data points are related. Hence, random permutation along each grid dimension would lead to loss of information. E.g.:

- in time series: neighboring points are temporally close
- in images: neighboring points are spatially close
- in videos: neighboring points are spatially/temporally close

Convolution in CNNs involves applying a filter to many local regions, where a local region comprises a set of related grid points, often neighboring points in space or time.

The size of the region is termed the *kernel width/size*. The filter operation itself is linear, taking as input the data points within the region and producing a single value, such as an average.

Filters can be represented by *kernel matrices* or basis matrices, which compute a weighted sum of the local region. The kernel matrix contains *one weight per data point* in the local region and shares the same shape as the region.



When trying to capture locality of a certain size, then the filter should relate to this size.

For example, in a region of width 3 over a sequence, the filter operation can be represented as

$$(x_{t-1}, x_t, x_{t+1}) \rightarrow x_{t-1} - x_{t+1}$$

where the corresponding kernel matrix reflecting this operation would be $\boldsymbol{K} = (1 \quad 0 \quad -1)$.

6.2.1 Stride & Padding

The (same) kernel is shifted systematically over the data, e.g. a sliding window/rectangle. The amount by which it is shifted is called *stride*.

$$\text{output size} \approx \frac{\text{input size}}{\text{stride}}$$

Example

Kernel $\boldsymbol{K} = (1 \quad 0 \quad -1)$ with stride 1

Input	0	1	2	4	8	4	2	1	0
Convolution		-2	-3	-6	0	6	3	2	

Here: zero padding such that output size = input size (called **same convolution**)

Generally, stride & padding control both output size and how many outputs are influenced by each input. (Here: at borders: 2, for the rest: 3)

Example 3×3 filters and convolutions

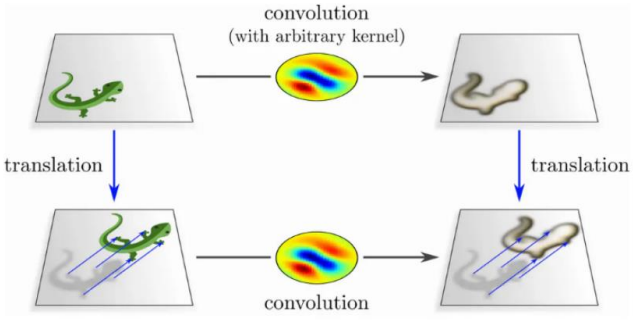
6.2.2 Key Concepts

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

💡 We can think of the kernel as a *feature detector,* as the operation extracts the same pattern in many locations/time steps of the data. A feature is computed via the inner product of input and kernel while the kernel is learned → *learned* feature detector. Hence, we may interpret such a feature as a measure of similarity between kernel and input. In CNNs, convolutions are followed by a nonlinearity.

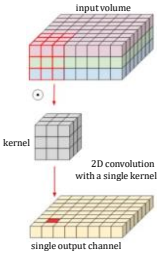
6.2.3 Translation Equivariance

❗ In CNNs we introduce an inductive bias, namely *translation equivariance*. When the input is translated by some amount, the output is too. Note that that's different to translation invariance.



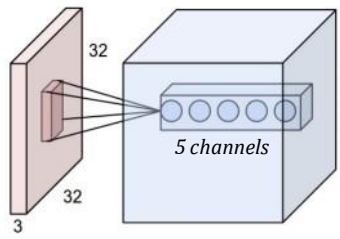
6.3 Common Layer Types

6.3.1 Convolutional Layers



📄 CNNs are composed of convolutional layers plus nonlinearity. Each layer performs multiple convolutions, each with a different kernel, and produces multiple feature maps (also: *channels*). Generally, conv. layers: take in input volume and produce an output volume.

Example



Input: image width (32) × image height (32) × RGB (3)

Output: image width (32) × image height (32) × feature maps (5)

The conv. layer is parameterized by 5 kernel matrices $\boldsymbol{K}_1, \ldots, \boldsymbol{K}_5 \in \mathbb{R}^{7\times7\times3}$ that are shifted across the first *two* dimensions, because this is the dimensionality in which we would assume the data to be strongly correlated → *locality*.

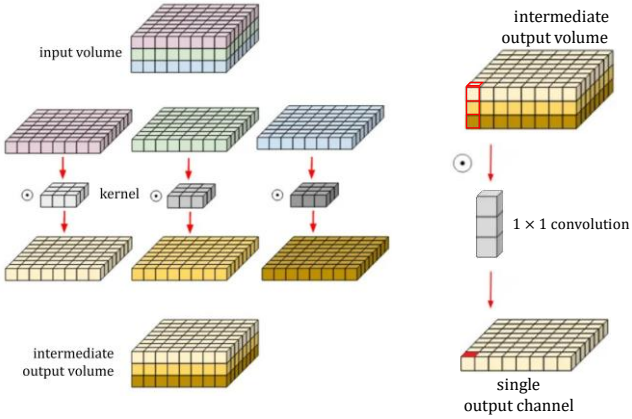
💡 Generally, the different kernel matrices aim to capture various patterns present in the data.

6.3.2 Depth-Wise Separable Convolution Layer

Depth-wise separable convolution layers utilize the fact that translation equivariance is not needed across different channels, i.e. that information across channels is not strongly correlated.

1. Apply *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 → H channels out)
2. Apply 1×1 convolution, i.e. a linear projection of individual element representations.

Example



Input:

width (32) × height (32) × RGB (3)

Intermediate output:

width (32) × height (32) × RGB (3)

Output:

width (32) × height (32) × channels (5)

E.g., for a 7×7 convolution the layer is parameterized by 3 kernel matrices $\boldsymbol{K}_1, \boldsymbol{K}_2, \boldsymbol{K}_3 \in \mathbb{R}^{7\times7}$ to produce the intermediate output plus 5 "kernel matrices" $\boldsymbol{K}'_1, \ldots, \boldsymbol{K}'_5 \in \mathbb{R}^{1\times1\times3}$ for the final output feature map. This reduces the amount of model parameters significantly:

$$(7 \times 7) \times 5 \times 3 \quad \text{vs.} \quad (7 \times 7) \times 3 + (5 \times 3)$$

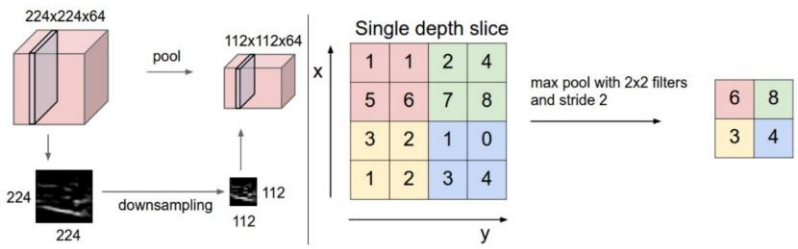
6.3.3 Pooling Layers

Convolutional layers are periodically followed by *pooling layers,* to (spatially) aggregate information more efficiently. These operations group together (*pool*) the values in a region.

Like a convolution the filter is shifted along certain dimensions, but is typically hard-coded (e.g., max pooling, avg pooling) and possibly a non-linear operation. It can be performed along spatial dimension and/or feature dimensions.

- Pooling reduces size: stride = number of pooled values (per dimension)
- Pooling introduces inductive bias: increased spatial/feature invariance

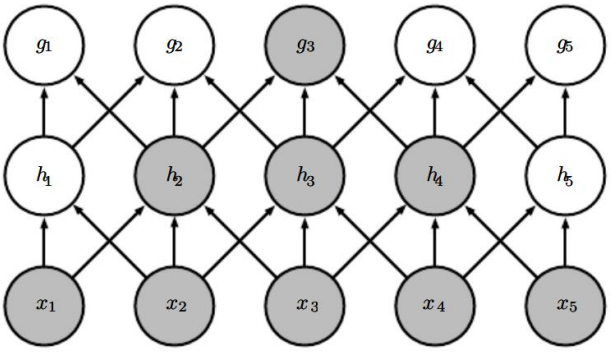
Example: spatial max-pooling



6.4 Receptive Field

Each output neuron in a CNN has a receptive field, i.e., the set of inputs that have influenced this feature or the region of the input data that the neuron is sensitive to. The *receptive field* increases via convolution or pooling operations. This means that neurons in higher layers are influenced by a larger portion of the input data, allowing them to capture more complex patterns and features. For example, lower layers might detect simple features like edges, while higher layers might detect more complex features like objects. This is an important design consideration: if we want to detect an object of a certain size, we'd want a receptive field of that size.

Example: receptive field after two convolutions of width 3



The *effective* receptive field is often smaller and focused on the center region.

6.5 …

Lecture #11 on thu. 2.5.

7 …

Lecture #12 on 7.5.