**Content**

1 Introduction .......... 2

2 Feed-Forward Neural Networks .......... 3

2.1 Embeddings .......... 3

2.2 Artificial Neurons .......... 4

2.3 Artificial Neural Networks .......... 5

2.4 Non-Linear Layers .......... 7

2.5 Summary .......... 12

3 Gradient-Based Training .......... 13

3.1 Backpropagation .......... 13

3.2 Recap: Basics .......... 14

3.3 Compute Graphs .......... 15

3.4 Optimizers .......... 19

# 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.

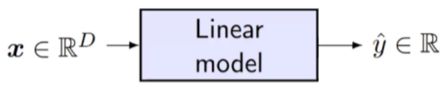
# 2 Feed-Forward Neural Networks

## 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.

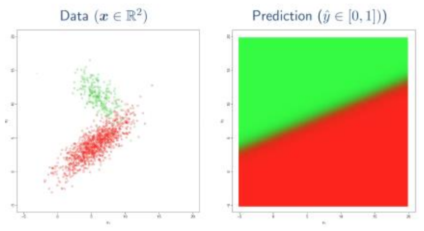
- Inputs must be real-valued feature vectors $\boldsymbol{x} \in \mathbb{R}^D$
- Outputs are a real value (e.g. lin./log. regression)



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

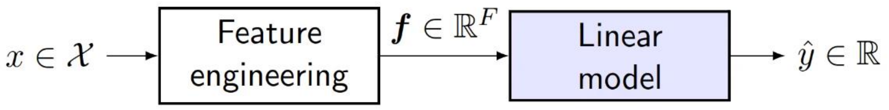
$$\hat{y} = \phi(\boldsymbol{w}^\top \boldsymbol{x} + b)$$

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

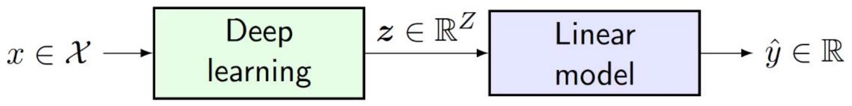
One approach to solve this problem is to perform *feature engineering*. Design some form of function that maps arbitrary input spaces to real-valued vectors $f : \mathcal{X} \to \mathbb{R}^F$



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* $\boldsymbol{z} \in \mathbb{R}^Z$.

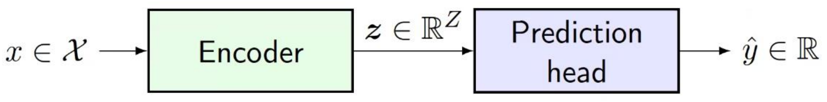


Instead of engineering features manually, embeddings are learned from data. They are also called latent code or distributed representations. The *embedding space* is also called *latent space*.

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

### 2.1.3 Terminology

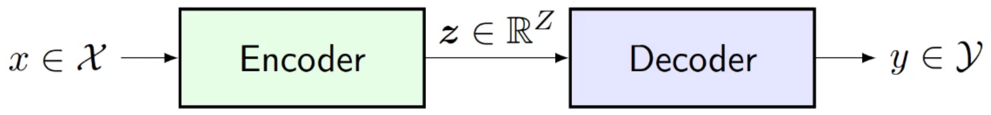
📝 Functions that transform embeddings $\boldsymbol{z} \in \mathbb{R}^Z$ to predictions $y \in \mathcal{Y}$ are known as *prediction heads*. They can be arbitrary simple/linear or complex.



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

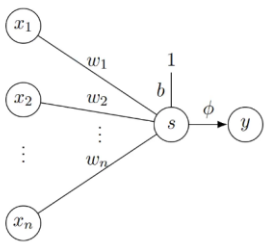
### 2.1.4 Encoder-Decoder

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.

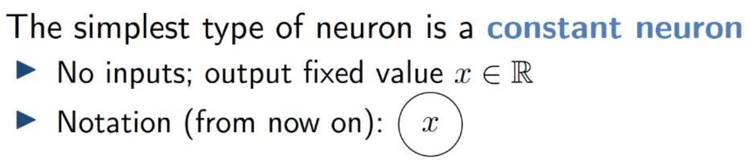
## 2.2 Artificial Neurons



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*.

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

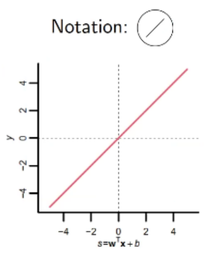
There are many types of neurons that only differ in their *activation* (*transfer*) function $\phi$.



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

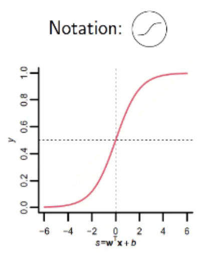
- No inputs; output fixed value $x \in \mathbb{R}$
- Notation (from now on): $\bigcirc\!\!\!\!x$

### 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

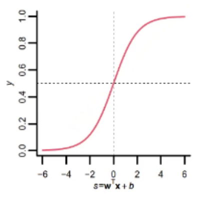


The logistic neuron uses $\phi(s) = \sigma(s) \overset{\text{def}}{=} \frac{1}{1+\exp(-s)}$ as an activation function. This gives a real-valued output that is smooth and bounded in [0,1].

- 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:

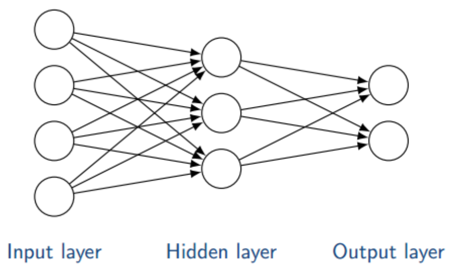
- How many neurons, which type?
- Output neurons? Hidden Neurons?
- Which are connected?
- Directed or undirected connections?

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)

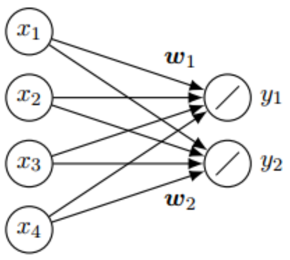
Layers do not need to be fully connected. Traditionally edges are only between subsequent layers (but: edges that skip layers are allowed, too)

💡 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:

$$\boldsymbol{W} = (\boldsymbol{w}_1 \quad \boldsymbol{w}_2 \quad \dots \quad \boldsymbol{w}_m)$$

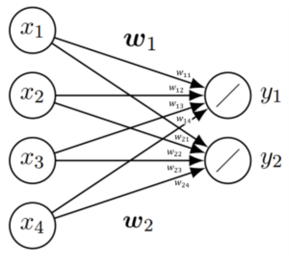
Linear layers compute a matrix-vector product:

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

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**



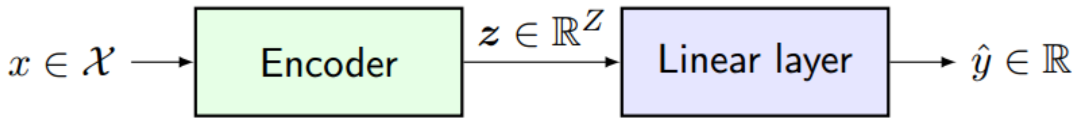
$$W = \begin{pmatrix} w_1 & w_2 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \\ w_{14} & w_{24} \end{pmatrix}$$

$$x^\top = (x_1 \quad x_2 \quad x_3 \quad x_4)$$

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

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 ($m > n$)

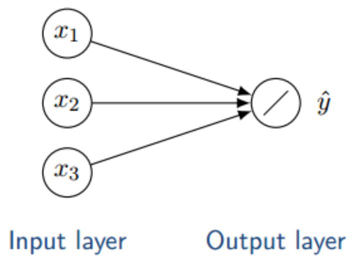
### 2.3.3 Linear Regression as FNN

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

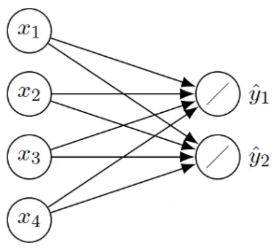
$$\hat{y} = \langle w, x \rangle + b$$

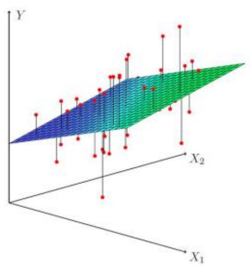
Suppose we train this network using ERM with squared loss:

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



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

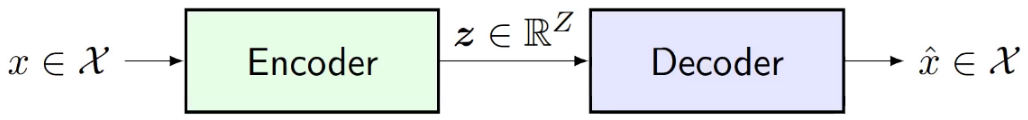




❗The output of a linear FFN remains linear even when adding hidden layers. That's why we often want non-linear transfer functions.

### 2.3.4 Autoencoders

📄 FNNs are useful for unsupervised learning as well. Given an unlabeled dataset $\mathcal{D} = \{x_i\}_{i=1}^N$ 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 $y_i = x_i$.



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

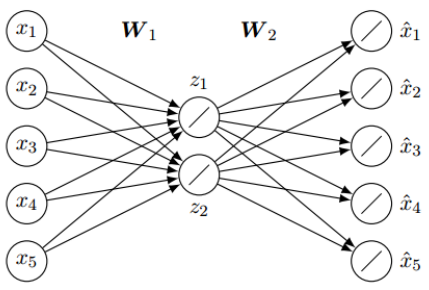
💡 Further, e.g. in semi-supervised learning, we can train the autoencoder on all inputs (unsupervised) and can then only train the prediction head in a supervised fashion using the labeled data. This allows the model to benefit from both, especially when labeled data is scarce.

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.

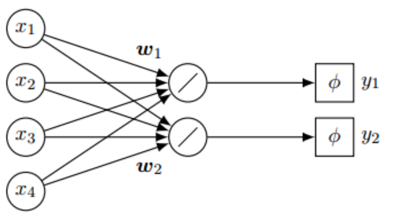


💡 A layer with few neurons is referred to as a *bottleneck, i*.e., fewer neurons than the surrounding layers. This forces FNN to compress information → *dimensionality reduction*

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:

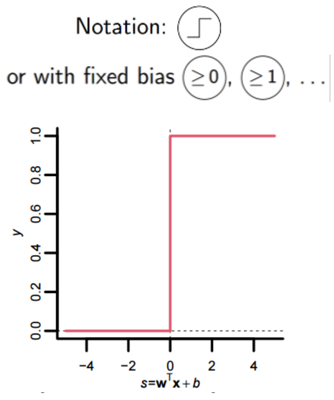


$$\boldsymbol{y} = \phi(\boldsymbol{W}^\top \boldsymbol{x})$$

where we take the convection that $\phi$ 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:

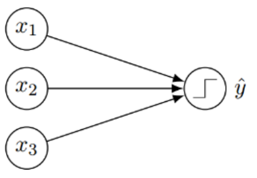


$$\phi(s) = \mathbb{I}(s \geq 0) = \begin{cases} 1 & \text{if s} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

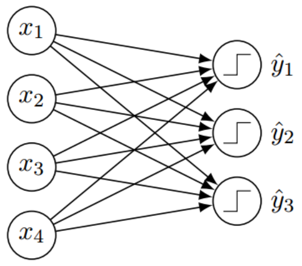
outputs fixed "spike" if input $s$ is non-negative, else "nothing".

### 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), ...

$m = 3$ *class labels*

$$\begin{array}{c|cccccccc|c} \text{Output} & & & & & & & & & \text{Classes} \\ \hat{y}_1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \text{A} \\ \hat{y}_2 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \text{B} \\ \hat{y}_3 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & \text{C} \end{array}$$

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!

$m = 3,\ \ 2^3 = 8$ *class labels*

$$\begin{array}{c|cccccccc} \text{Output} & & & & & & & & \\ \hat{y}_1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \hat{y}_2 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ \hat{y}_3 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ \hline \text{Classes} & \text{A} & \text{B} & \text{C} & \text{D} & \text{E} & \text{F} & \text{G} & \text{H} \end{array}$$

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

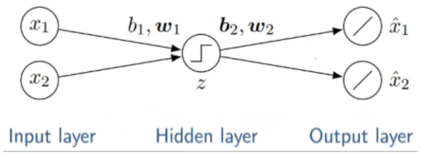
   - Each input is associated with one out of $m$ class labels
   - We associate each label with its indicator vector (one-hot encoding)
   - ❗ Problem: What if the network outputs less/more than a single 1?

$m = 3$ *class labels*

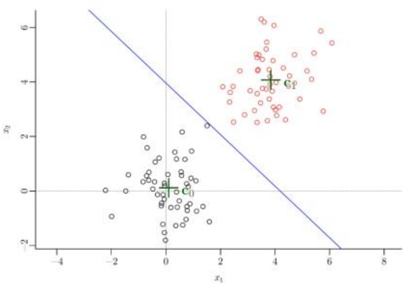
$$\begin{array}{c|ccc} \text{Output} & & & \\ \hat{y}_1 & 1 & 0 & 0 \\ \hat{y}_2 & 0 & 1 & 0 \\ \hat{y}_3 & 0 & 0 & 1 \\ \hline \text{Classes} & \text{A} & \text{B} & \text{C} \end{array}$$

### 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.

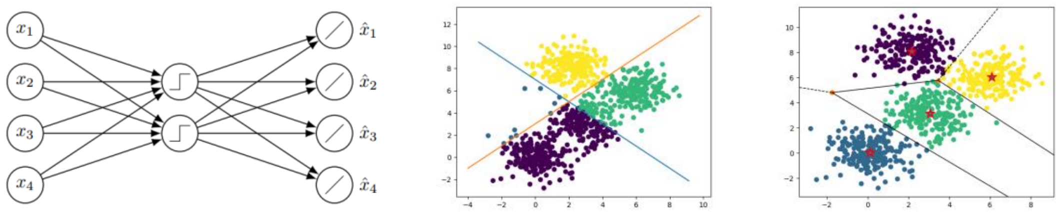


This happens as the binary threshold unit acts as a linear classifier. The input $x$ is either mapped to $z = 0$ or to $z = 1$.

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.

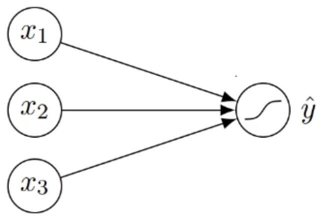
What happens if we have multiple binary threshold units?



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

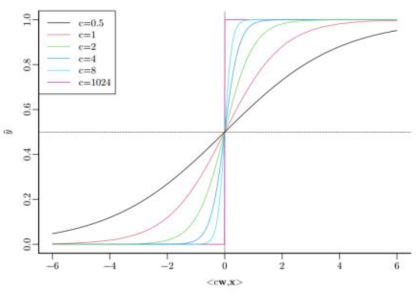
### 2.4.5 FNN with single Logistic Neuron

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.

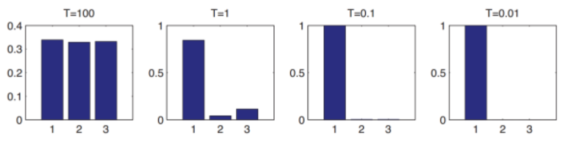
## 2.4.6 SoftMax Function

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* $T$.

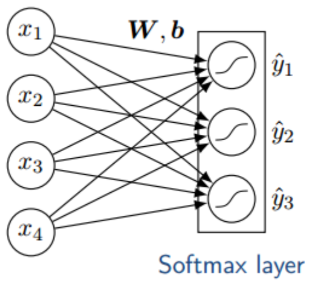
When the temperature $T$ is high (left), the distribution is rather uniform, whereas when the temperature $T$ is low (right), the distribution is "spiky", with all its mass on the largest element.



## 2.4.7 SoftMax Layer

A *SoftMax layer* computes $\hat{\boldsymbol{y}} = S\left(\frac{\boldsymbol{W}^{\top}\boldsymbol{x}+\boldsymbol{b}}{T}\right)$, 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:

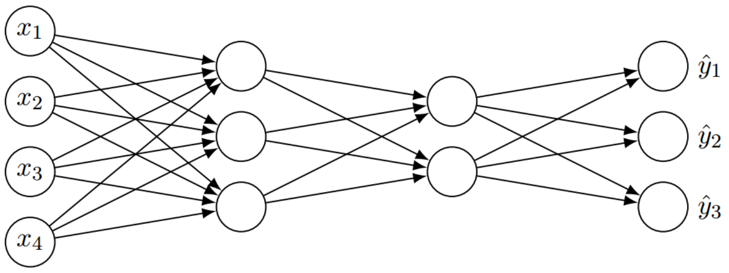
$$\mathcal{S}_C = \left\{ \boldsymbol{y}_c \in \mathbb{R}^C, \quad 0 < y_c \leq 1, \quad \sum_c y_c = 1 \right\}$$

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).

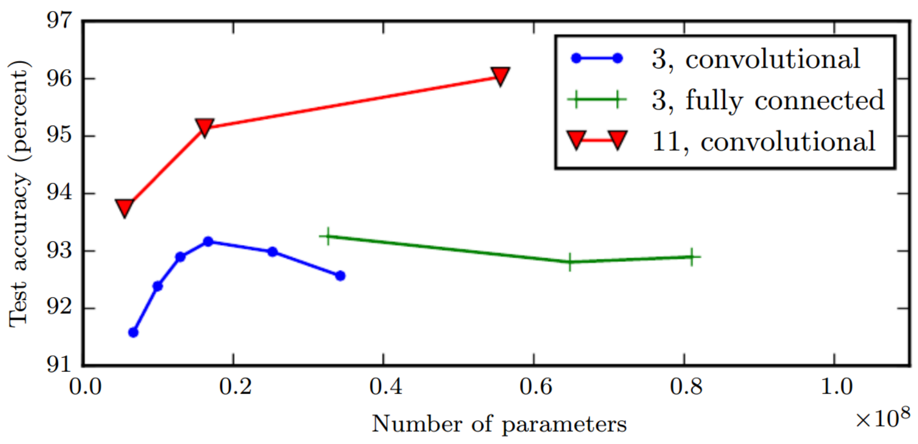
📝 If there is at least one hidden layer ($L \geq 1$) , the network is called a *multi-layer FNN* (or also *multi-layer perceptron: MLP*). If $L > 1$, we call this network *deep*.



❗ 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.

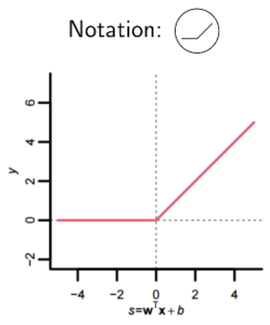


The legend indicates the *depth* of network used to make each curve. Increasing the number of parameters in layers of convolutional networks *without increasing their depth* is not nearly as effective at increasing test set performance.

### 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".

Notation:

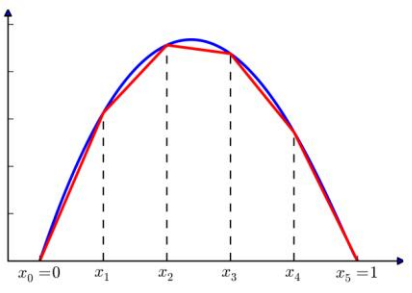


$$\phi(s) = \max\{0, s\} = \begin{cases} s & \text{if } s \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

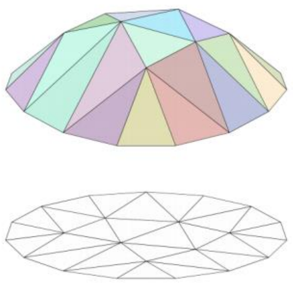
💡 Common non-linearity for intermediate layers in deep NNs.

### 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*)

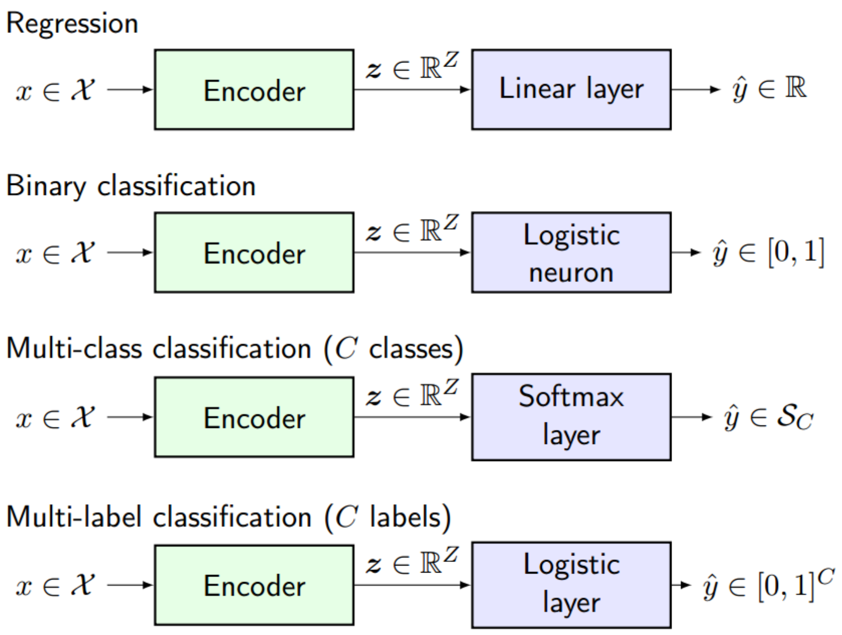
Then, the number of linear regions is *at most* $2^H$ → not more than exponentially many linear regions are possible! (“bounded from above”)

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:

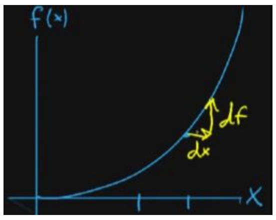
- Output $\hat{y}$ of an FNN, given $\boldsymbol{x}, \boldsymbol{\theta}$
- Loss $L$ of an FNN, given $(\boldsymbol{x}, y), \boldsymbol{\theta}$
- Cost function $J$ for an FNN, given $\{(\boldsymbol{x}_i, y_i)\}, \boldsymbol{\theta}$

We are also interested in gradients $\nabla$:

- 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 ($\partial$) derivatives, where $f$ is a function of several variables. Here we're finding the derivative with respect to one of those variables (e.g.: $\partial x$) 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 $\nabla_{x^\top}$ of any function $f$ gathers all its partial derivatives in a (row) vector:

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

For the example in 3.2.2, we obtain:

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

Numerator layout (row) Denominator layout (column)

### 3.2.4 Chain Rule

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

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

A function is composite if you can write it as $f(g(v))$. 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:

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

### 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:

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

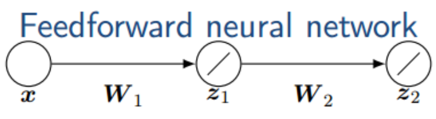
### 3.2.6 Multivariate Chain Rule

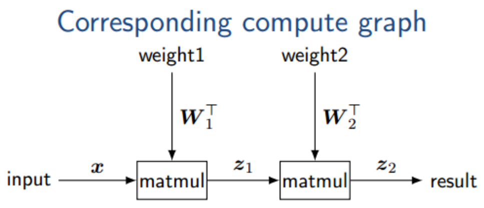
Consider a composite multivariable function $f\big(x(t), y(t)\big)$, its ordinary derivate is:

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

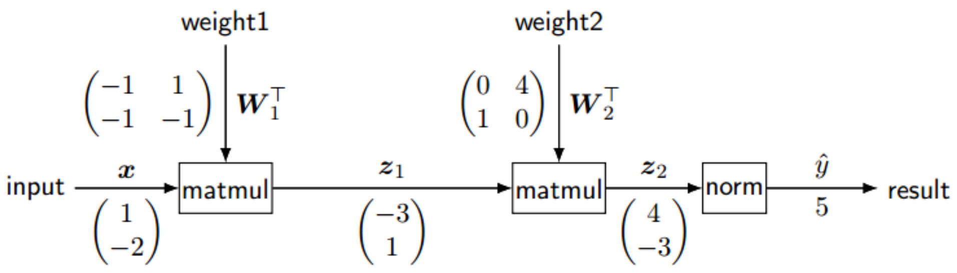
## 3.3 Compute Graphs

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.





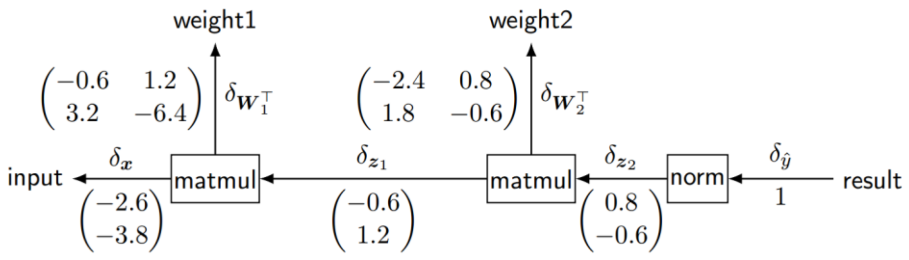
**Example**: Forward Pass


weight1
$\begin{pmatrix} -1 & 1 \\ -1 & -1 \end{pmatrix}$
$\boldsymbol{W}_1^\top$
weight2
$\begin{pmatrix} 0 & 4 \\ 1 & 0 \end{pmatrix}$
$\boldsymbol{W}_2^\top$
input
$\boldsymbol{x}$
$\begin{pmatrix} 1 \\ -2 \end{pmatrix}$
matmul
$z_1$
$\begin{pmatrix} -3 \\ 1 \end{pmatrix}$
matmul
$z_2$
$\begin{pmatrix} 4 \\ -3 \end{pmatrix}$
norm
$\hat{y}$
5
result

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


weight1
weight2
$\begin{pmatrix}-0.6 & 1.2\\ 3.2 & -6.4\end{pmatrix}$
$\delta_{W_1^\top}$
$\begin{pmatrix}-2.4 & 0.8\\ 1.8 & -0.6\end{pmatrix}$
$\delta_{W_2^\top}$
input
$\delta_x$
matmul
$\delta_{z_1}$
matmul
$\delta_{z_2}$
norm
$\delta_{\hat{y}}$
result
1
$\begin{pmatrix}-2.6\\ -3.8\end{pmatrix}$
$\begin{pmatrix}-0.6\\ 1.2\end{pmatrix}$
$\begin{pmatrix}0.8\\ -0.6\end{pmatrix}$

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 $e$**

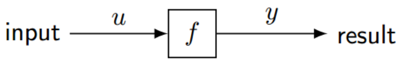
Example: $\delta_{z_2} = \nabla_{z_2}\hat{y} = \begin{pmatrix}0.8\\ -0.6\end{pmatrix}$. This gradient tells us, that increasing the first value in $\boldsymbol{z}_2$ causes the result to *increase*; similarly, increasing the second value in $\boldsymbol{z}_2$ causes the result to *decrease*.

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 $\delta_e$ 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.

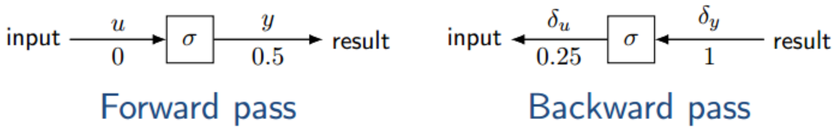
## 3.3.1 Gradient: Single Univariate Function

- 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 $u = 0$.



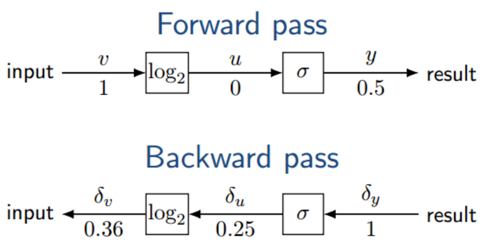
Gradient $\delta_u = \sigma'(u) = \sigma(u)\big(1 - \sigma(u)\big)$, evaluated at input $u = 0$ from forward pass:

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

### 3.3.2 Gradient: Composition of Two Univariate Functions

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

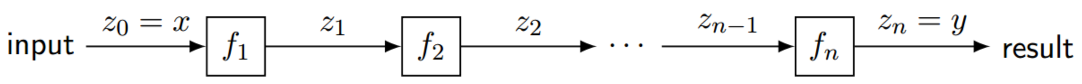
Gradients can be obtained:



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

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

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:

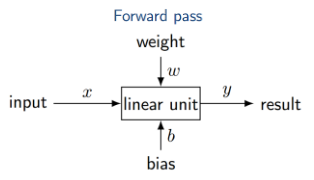
$$\delta_{z_{i-1}} \overset{\text{def}}{=} \nabla_{z_{i-1}} y = \frac{\partial y}{\partial z_{i-1}} = \frac{\partial y}{\partial z_i} \cdot \frac{\partial z_i}{\partial z_{i-1}}$$
$$= f_i'(z_{i-1}) \cdot \delta_{z_i}$$

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.

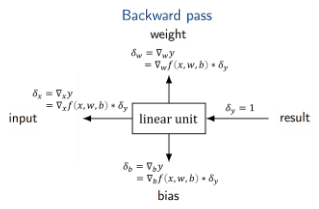
### 3.3.3 Gradient: Multiple Inputs



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

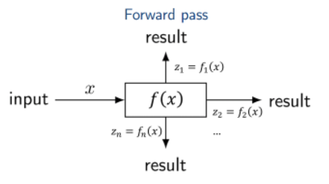
In the backward pass, we simply compute gradients of result w.r.t. each edge using the chain rule:

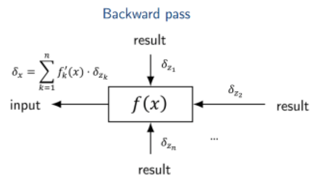
- $\delta_y = 1$
- $\delta_x = \nabla_x y = \nabla_x f(x, w, b) * \delta_y = x * 1$
- $\delta_w = \nabla_w y = \nabla_w f(x, w, b) * \delta_y = w * 1$
- $\delta_b = \nabla_b y = \nabla_b f(x, w, b) * \delta_y = 1 * 1$


Backward pass
weight
$\delta_w = \nabla_w y = \nabla_w f(x,w,b) * \delta_y$
$\delta_x = \nabla_x y = \nabla_x f(x,w,b) * \delta_y$
input
linear unit
$\delta_y = 1$
result
$\delta_b = \nabla_b y = \nabla_b f(x,w,b) * \delta_y$
bias

💡 We consider each input separately and reuse the $\delta$-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:

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

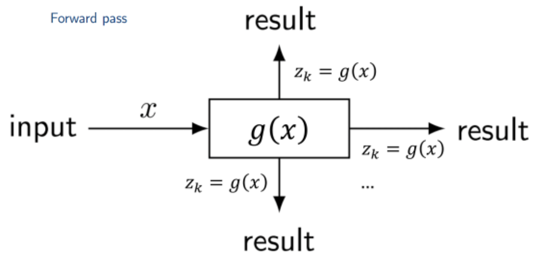
💡 We consider each output independently and sum up.

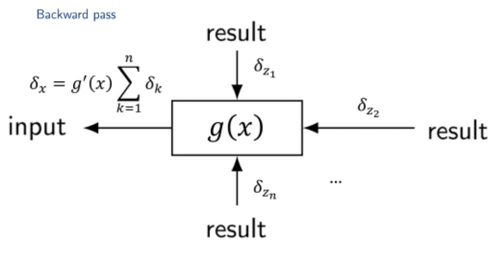
### 3.3.5 Gradient: Multiple Use

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 $\delta$-values and proceed as before.



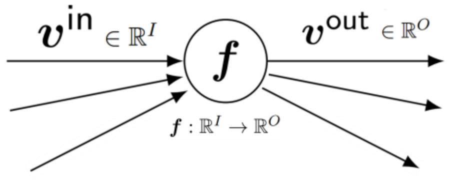


### 3.3.6 Gradient Computation with Tensors

📝 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.

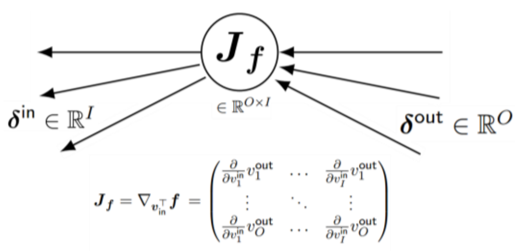
**Forward pass**

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


$\boldsymbol{v}^{\text{in}} \in \mathbb{R}^I$
$\boldsymbol{f}$
$\boldsymbol{v}^{\text{out}} \in \mathbb{R}^O$
$\boldsymbol{f}: \mathbb{R}^I \to \mathbb{R}^O$

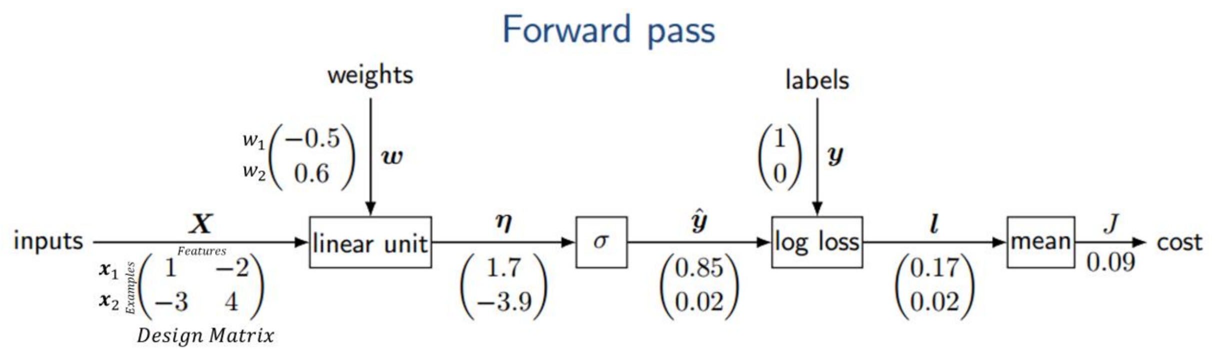
**Backward pass**

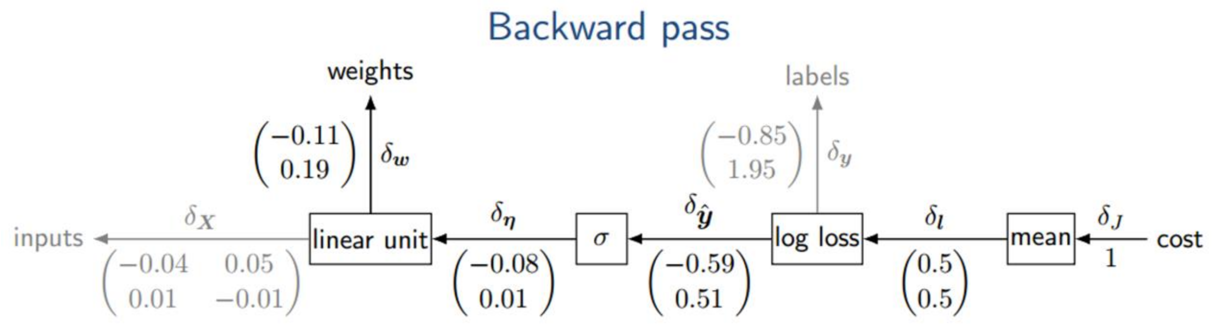
$$\boldsymbol{\delta}^{in} = \boldsymbol{J_f}(\boldsymbol{v}^{in})^\top \cdot \boldsymbol{\delta}^{out}$$


$\boldsymbol{J_f}$
$\in \mathbb{R}^{O \times I}$
$\boldsymbol{\delta}^{\text{in}} \in \mathbb{R}^I$
$\boldsymbol{\delta}^{\text{out}} \in \mathbb{R}^O$
$\boldsymbol{J_f} = \nabla_{\boldsymbol{v}_{\text{in}}^\top} \boldsymbol{f} = \begin{pmatrix} \frac{\partial}{\partial v_1^{\text{in}}} v_1^{\text{out}} & \cdots & \frac{\partial}{\partial v_I^{\text{in}}} v_1^{\text{out}} \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial v_1^{\text{in}}} v_O^{\text{out}} & \cdots & \frac{\partial}{\partial v_I^{\text{in}}} v_O^{\text{out}} \end{pmatrix}$

💡 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 $\boldsymbol{J_f}(\boldsymbol{v}^{in})^\top$.

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




Backward pass
weights
−0.11
0.19
$\delta_{\boldsymbol{w}}$
labels
−0.85
1.95
$\delta_{\boldsymbol{y}}$
inputs
$\delta_{\boldsymbol{X}}$
−0.04 0.05
0.01 −0.01
linear unit
$\delta_{\boldsymbol{\eta}}$
−0.08
0.01
$\sigma$
$\delta_{\hat{\boldsymbol{y}}}$
−0.59
0.51
log loss
$\delta_{\boldsymbol{l}}$
0.5
0.5
mean
$\delta_J$
1
cost

## 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 $J(\boldsymbol{\theta}) \rightarrow$ difficult! To make gradient-based methods work, need well-chosen *training parameters* and a suitable *network architecture design*.

### 3.4.2 Effect of Batch Size

💡 Empirical Risk Minimization (ERM) is a theoretic principle that guides the design of a variety of learning algorithms. The core idea is based on an application of the *law of large numbers*; more specifically, we cannot know exactly how well a predictive algorithm will work in practice (i.e. the true "risk") because we do not know the true distribution of the data. We can instead *estimate* and optimize the performance of the algorithm on a known set of training data, i.e. 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:

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

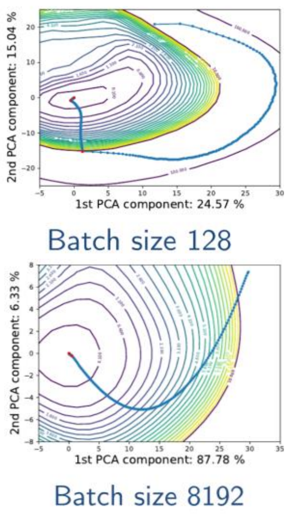
$$E[\nabla_{\boldsymbol{\theta}} J_{\mathcal{B}}(\boldsymbol{\theta})] = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

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

💡 **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 $\epsilon$ and Batch Size $|\mathcal{B}|$

*Red dot → learning rate reduced*



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

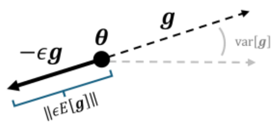
For large batch sizes $|\mathcal{B}| \rightarrow$ low variance gradients
- trajectory attracted by local minima
- empirically often lower generalization performance

For low learning rates $\epsilon \rightarrow$ small steps (slow trajectory)
- high variance gradient estimates average out

For high learning rates $\epsilon \rightarrow$ 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 $V = \text{var}[\hat{\boldsymbol{g}}]$



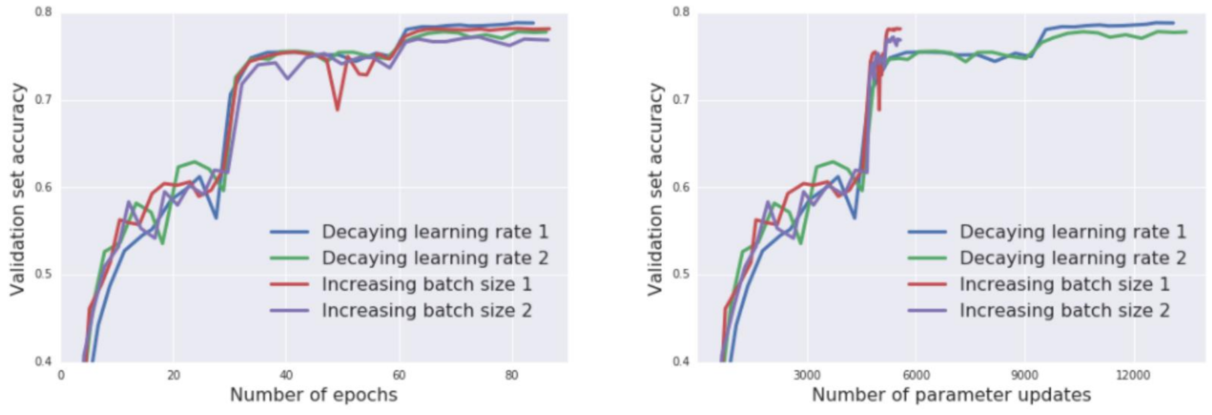
Both are affected by learning rate $\epsilon$, batch size $|\mathcal{B}|$ and cost function $J$.

| | $J_\mathcal{B}$=**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 $\epsilon$ to slowly approach the optimum once in vicinity.

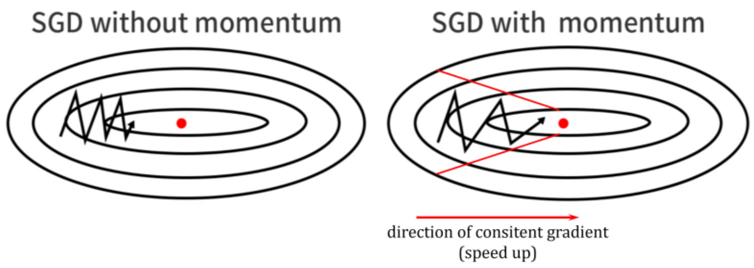
Similarly, a *batch size scheduler* can also be used to increase the batch size later during the training to slowly approach the optimum. This also eases parallelization.



*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.



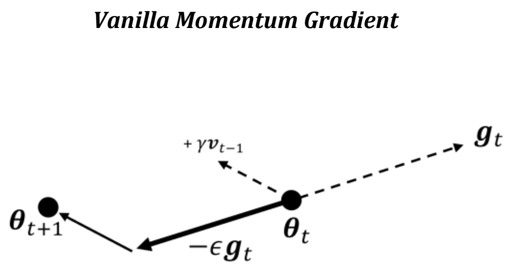
This solves two problems: poor conditioning of the Hessian matrix (narrow valleys, see figure) and variance in the stochastic gradient.

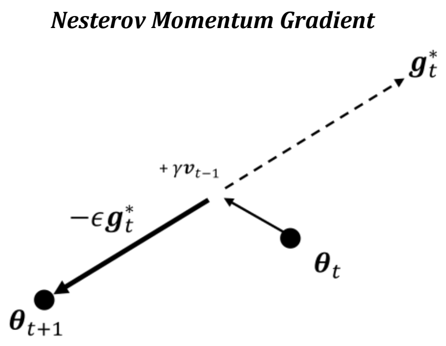
Vanilla/Basic *Momentum* (also: *heavy-ball method*) uses an exponentially-decaying moving average of the negative gradient.

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

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


*Nesterov Momentum Gradient*
$\boldsymbol{g}_t^*$
$+\gamma \boldsymbol{v}_{t-1}$
$-\epsilon \boldsymbol{g}_t^*$
$\boldsymbol{\theta}_t$
$\boldsymbol{\theta}_{t+1}$

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:

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

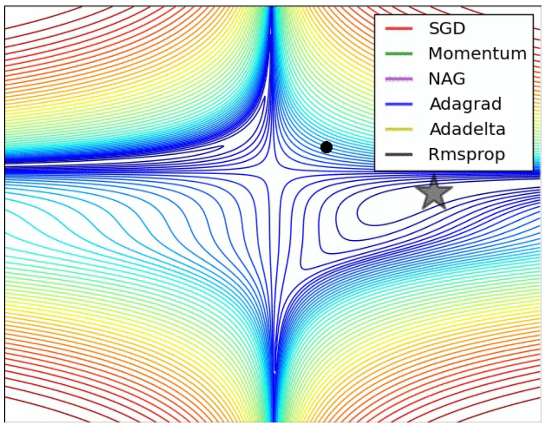
Note: operator $\odot$ is the elementwise multiplication.

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

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

To be continued after lect. 5

- … 💡
- … 📄
- … ❗
- … ❓
- … →