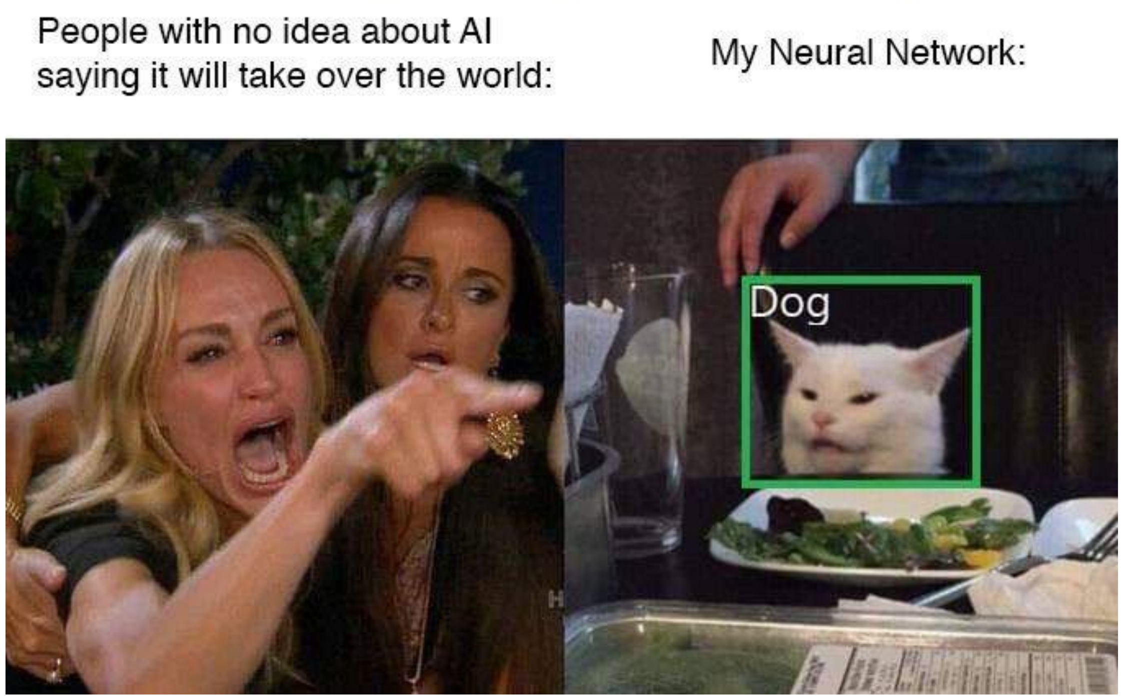
OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG

INFORMATIK UND MATHEMATIK

Lecture 4
Neural Networks

Dr.-Ing. Maike Stern | 04.11.2021


People with no idea about AI saying it will take over the world:
My Neural Network:
Dog

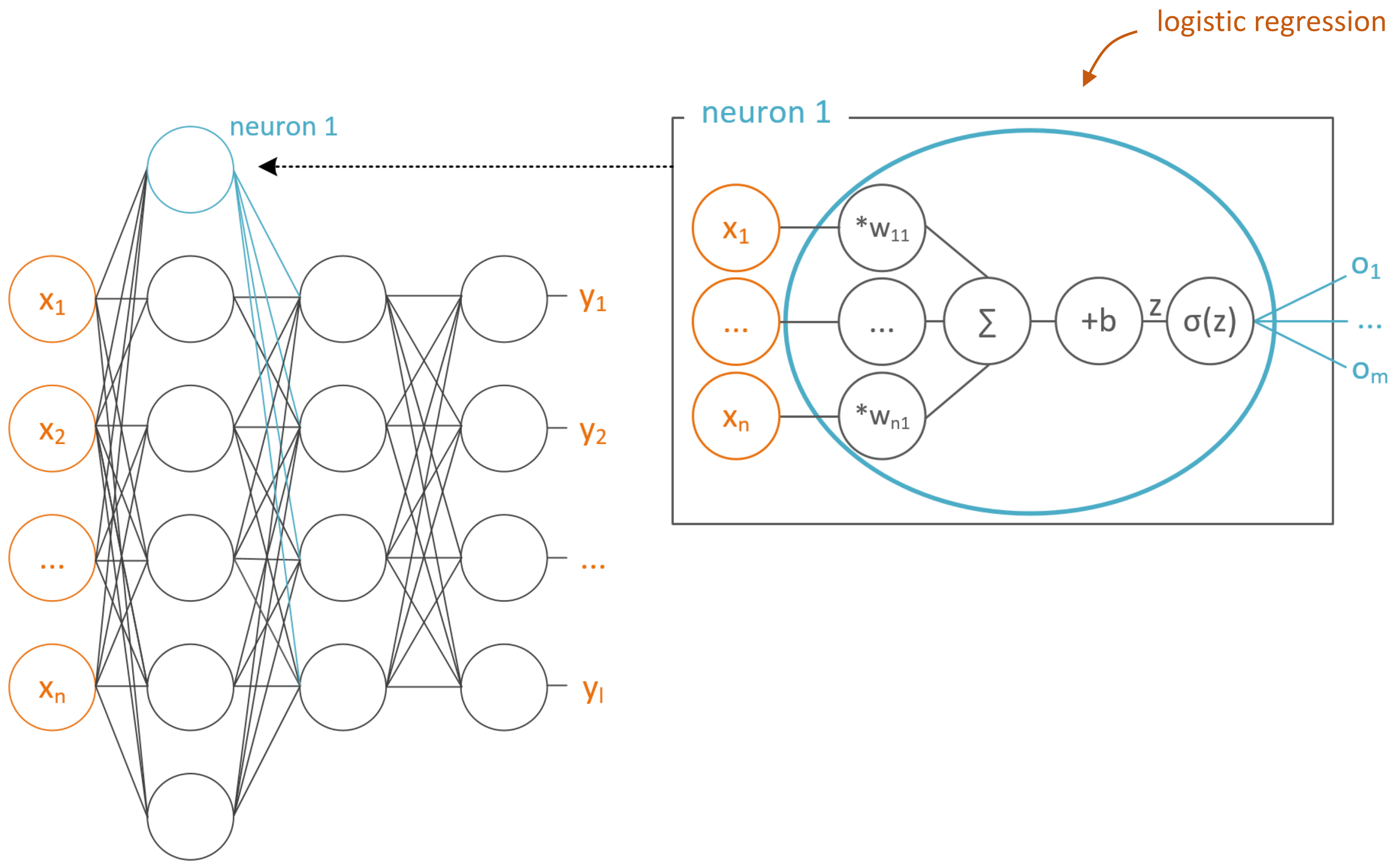
Tutorial

Please install TensorFlow and keras tuner for the tutorial on Monday.

- https://www.tensorflow.org/install (I am working with an Anaconda virtual environment but still use pip for TensorFlow and Keras tuner)
- https://keras.io/keras_tuner/

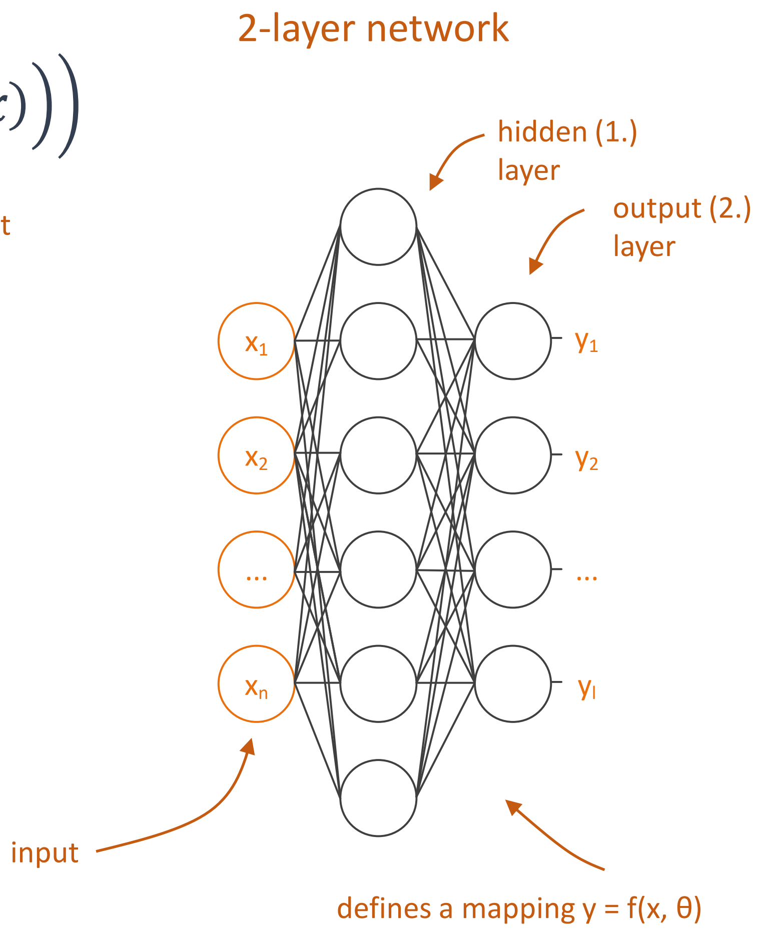
Feedforward neural networks

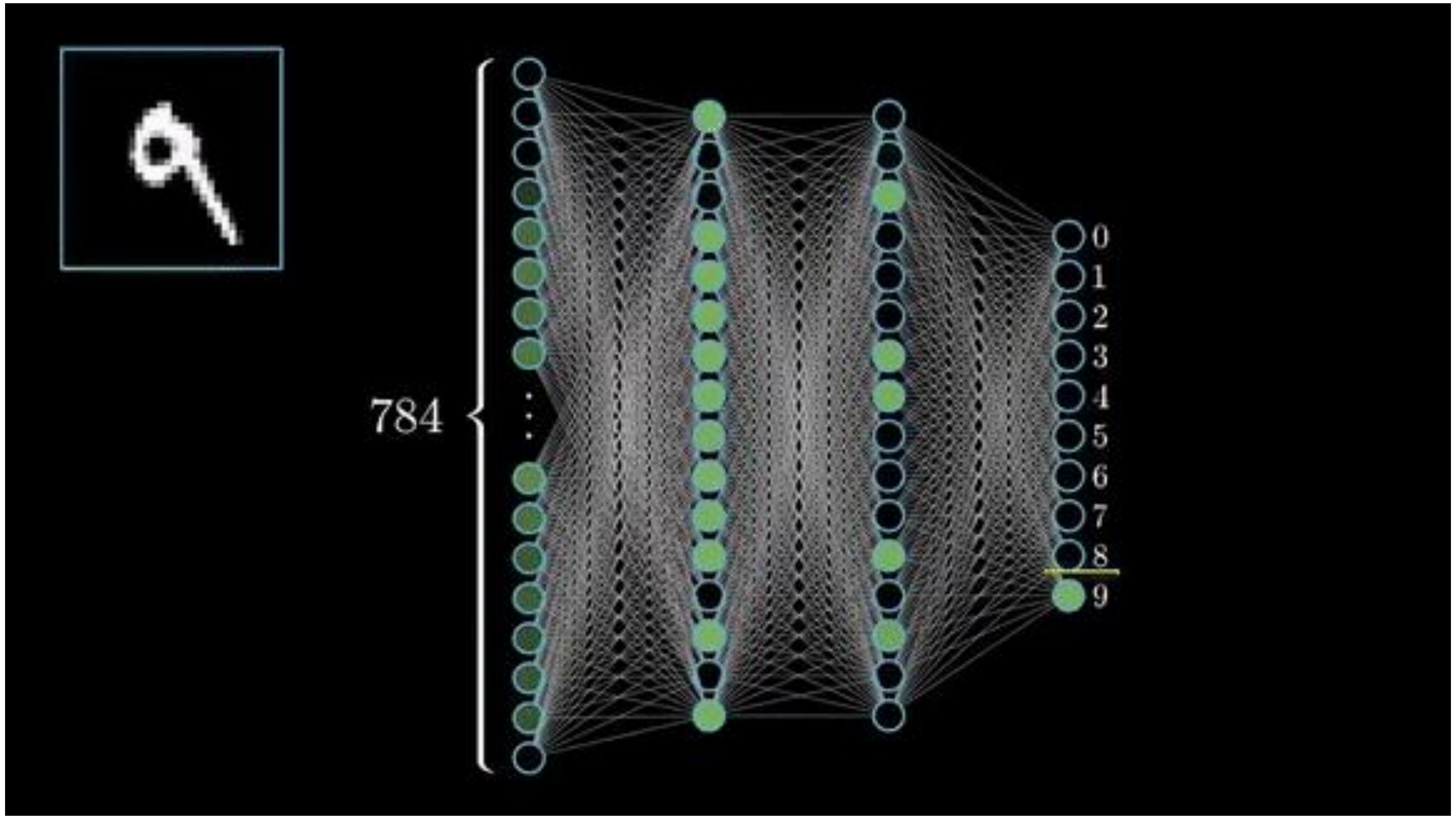
Feedforward neural networks or multi-layer perceptrons (MLP) are parallel and sequential combinations of neurons, e.g. of sigmoid neurons



Feedforward neural networks

- Goal: approximate some function f^*
- NNs are a composition of many different functions $f(\boldsymbol{x}) = \overset{\text{third layer}}{f^{(3)}}\left(\overset{\text{second layer}}{f^{(2)}}\left(\overset{\text{first layer}}{f^{(1)}}(\underset{\text{input}}{\boldsymbol{x}})\right)\right)$
- During network training we drive $f(\boldsymbol{x})$ to match $f^*(\boldsymbol{x})$

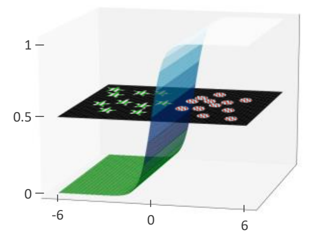


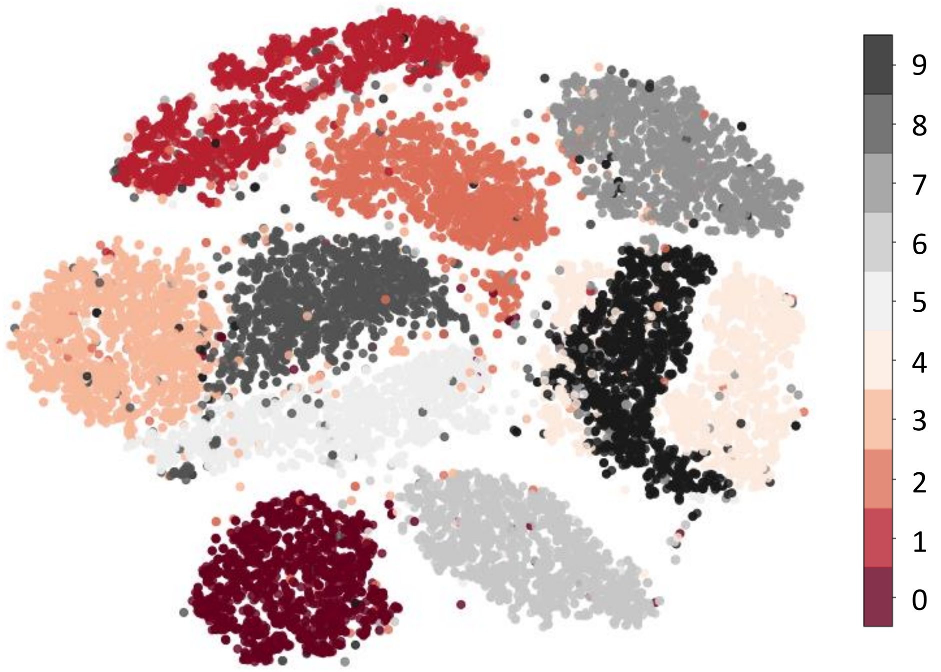

784
0
1
2
3
4
5
6
7
8
9

Feedforward neural networks

Feedforward networks with hidden layers provide a universial approximation framework

→ A large network will be able to represent the function we are trying to learn – however, it is not guaranteed that the training algorithm is able to learn that function





t-SNE visualisation of the MNIST dataset

Feedforward neural networks

But why do we then need different network architectures, anyway? The „no free lunch“ theorem!

Averaged over **all** possible data-generating distributions, every classification algorithm has the same average performance when classifying previously unobserved points

Feedforward neural networks

But why do we then need different network architectures, anyway? The „no free lunch" theorem!

Averaged over **all** possible data-generating distributions, every classification algorithm has the same average performance when classifying previously unobserved points

- No machine learning algorithm is universally better than any other
- By making assumptions about the data we want to classify, we can design algorithms that perform well on these tasks → we build a set of preferences into the algorithm
- Larger datasets require less skill to get a good performance

Feedforward neural networks What we can influence

Network architecture
Building a set of preferences into the algorithm

also hyperparameters

- E.g. number of layers and neurons, layer (architecture) type, activation function…

Hyperparameters
Ease network training

- E.g. learning rate, optimiser, regularisation, skip connections…

Data
What we tell the network to learn

- Data selection and preprocessing, data augmentation, …

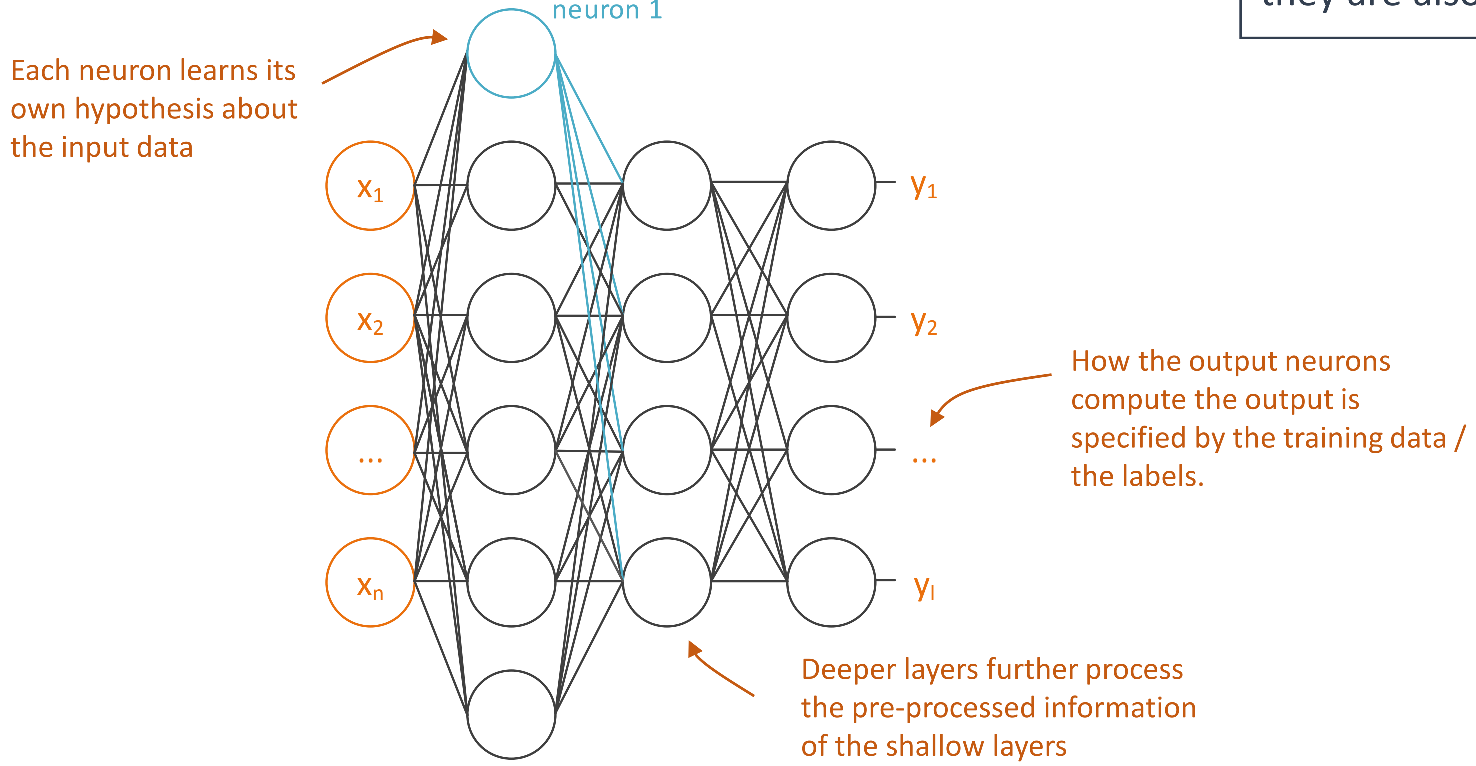
Feedforward network architecture

Feedforward neural networks
The basic network architecture

Hyperparameters:

- Number of neurons per layer
- Number of layers

Design rule:
Deeper networks (more layers) require fewer neurons per layer and tend to generalise better. But, they are also harder to train.


neuron 1
Each neuron learns its own hypothesis about the input data
x_1
x_2
...
x_n
y_1
y_2
...
y_l
How the output neurons compute the output is specified by the training data / the labels.
Deeper layers further process the pre-processed information of the shallow layers

Network architecture

- Feedforward networks are the simplest network architecture
- For computer vision, we use convolutional neural networks (CNNs)
  - ResNet
  - Encoder-Decoder
  - Fully convolutional networks
  - Generative adversarial networks
  - …
- For natural language processing, we use recurrent neural networks (RNNs)
  - Long short term memory networks (LSTM)
  - Gated recurrent unit networks (GRU)
  - Transformer (BERT, GPT-x, …)

Build-in preferences for images

Build-in preferences for sequential data

OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

INFORMATIK UND
MATHEMATIK

Activation Functions

Machine
Learning
Research

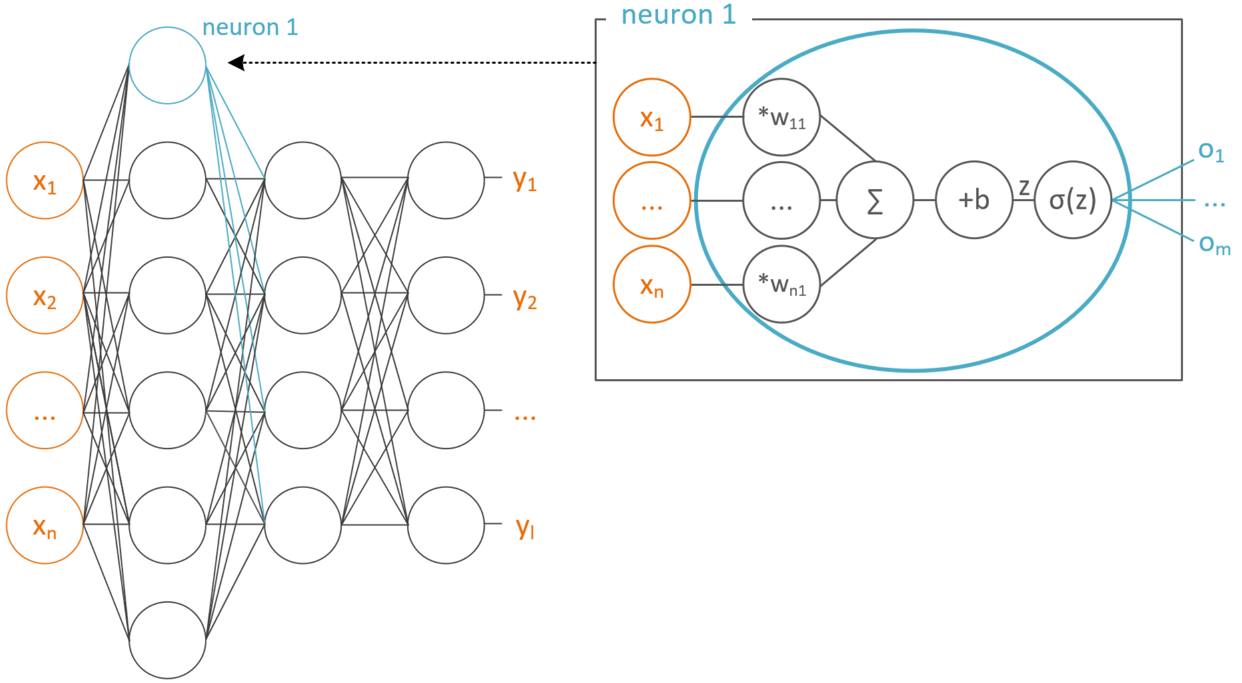
OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG

INFORMATIK UND MATHEMATIK

Feedforward neural networks Hyperparameters

Activation function g

- First layer: $\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$
- Second layer: $\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$
- …



Machine Learning Research

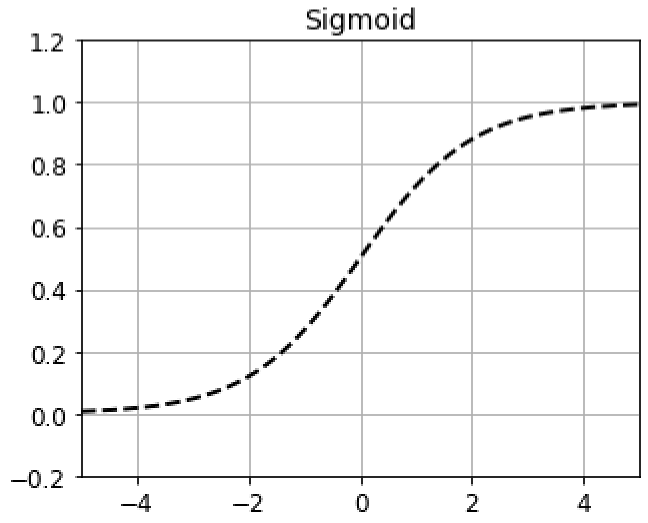
Feedforward neural networks
Hyperparameters | Activation functions

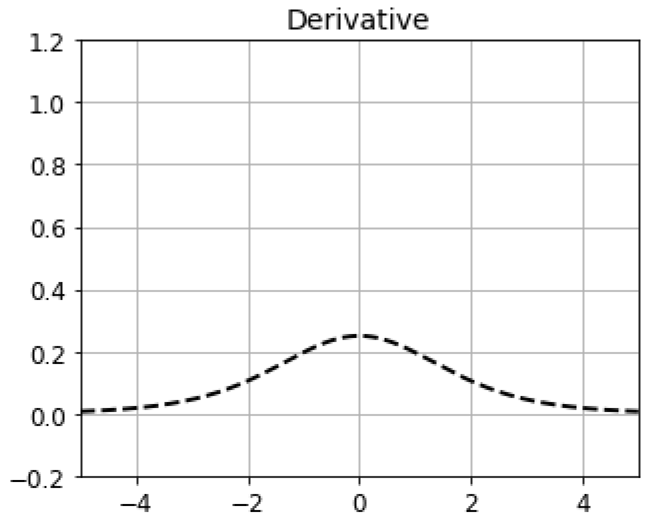
Sigmoid activation function: $\sigma(x) = \frac{1}{1+e^{-x}}$

Derivative: $\sigma'^{(x)} = \sigma(\text{x})(1 - \sigma(\text{x}))$

- Historically popular

- Saturates: very high or low values of x „kill“ the gradient
- Not zero-centered: can cause zig-zagging dynamics in the gradient updates





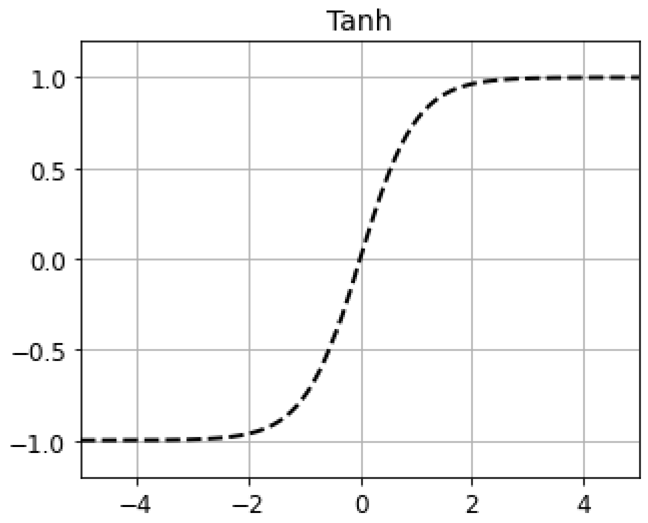
https://www.tensorflow.org/api_docs/python/tf/keras/activations

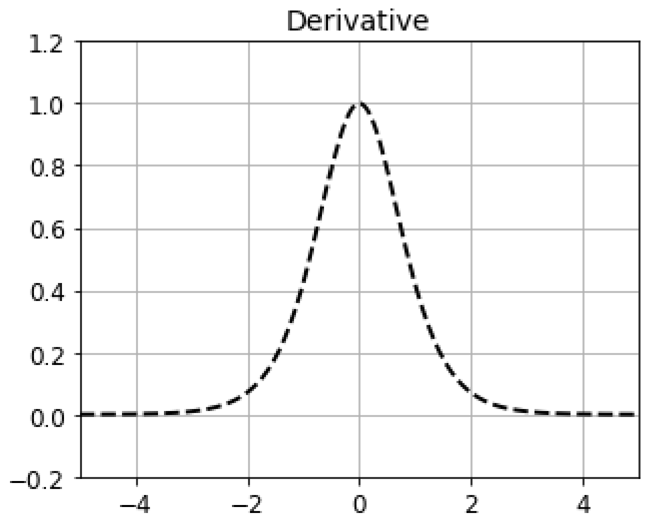
Feedforward neural networks
Hyperparameters | Activation functions

Tanh: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Derivative: $\tanh'(x) = 1 - \tanh(x)^2$

+ Zero-centered → better gradient updates
- Still saturates





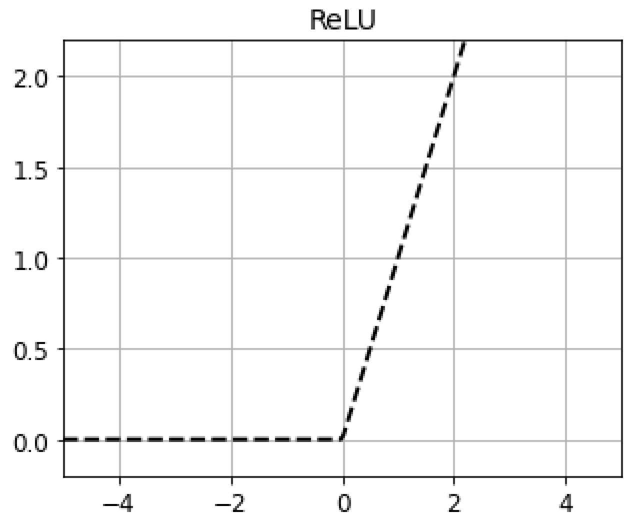
Machine Learning Research

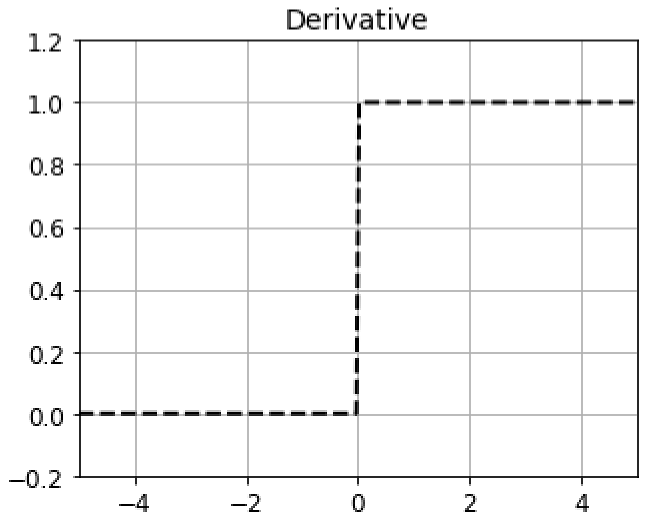
Feedforward neural networks
Hyperparameters | Activation functions

Rectified Linear Unit: $f(z) = \max(0, z)$

Derivative: $f'(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$

+ Greatly accelerates the convergence of Stochastic Gradient Descent, esp. compared to sigmoid/tanh
+ Computationally cheap
- Dying ReLUs (esp. with high learning rates), but then again sparse activations (to a certain degree) seem to be beneficial





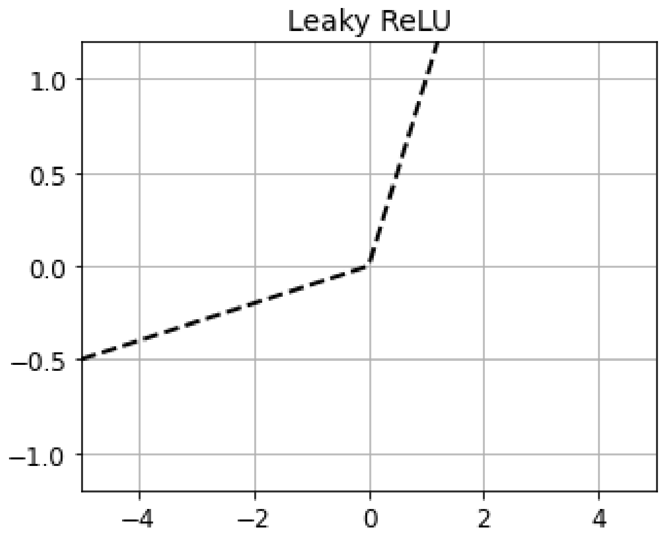
OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG
INFORMATIK UND MATHEMATIK

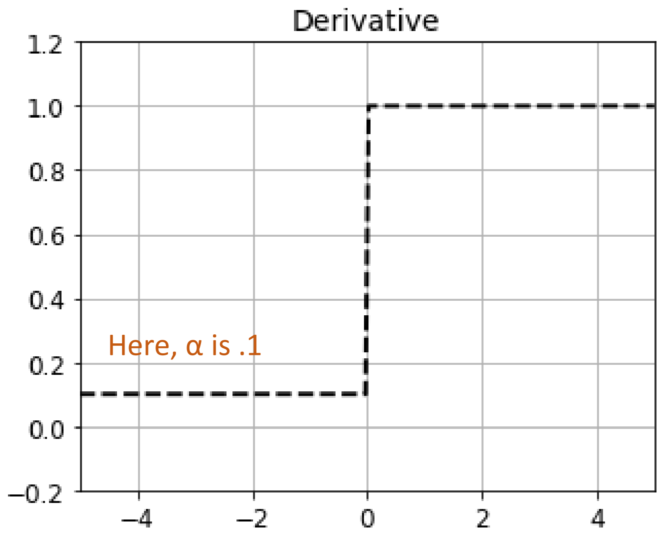
Feedforward neural networks
Hyperparameters | Activation functions

Leaky ReLU: $f(x) = \max(\alpha x, x)$

Derivative: $f'(z) = \begin{cases} \alpha, & z < 0 \\ 1, & z \geq 0 \end{cases}$

+ Non-zero gradient for $x < 0$, which avoids dead neurons





Machine Learning Research

OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG
INFORMATIK UND MATHEMATIK

Feedforward neural networks
Hyperparameters | Activation functions

Exponential Linear Unit: $f(x) = \begin{cases} \alpha(e^x - 1), & x < 0 \\ x, & x \geq 0 \end{cases}$

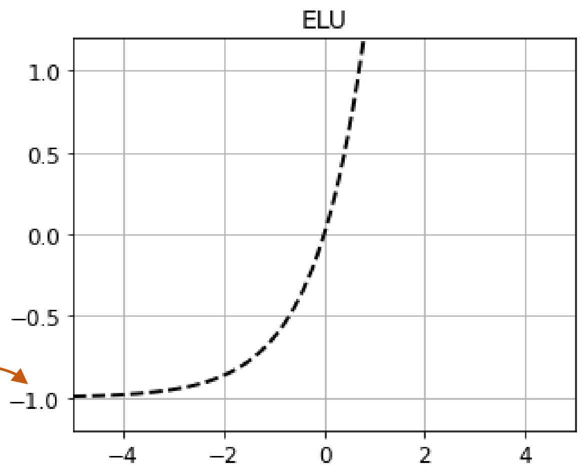
Determines the value f approaches when x is a large negative number

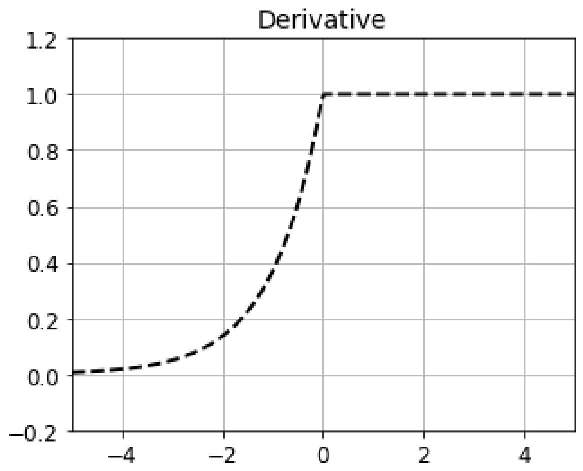
Design rule:
ELU > leaky ReLU > ReLU > tanh > sigmoid, for the hidden units, but ReLU as default is ok as well.

Derivative: : $f'(z) = \begin{cases} \mathrm{ELU}(x) + \alpha, & z < 0 \\ 1, & z \geq 0 \end{cases}$

+ Push mean unit activations closer to zero
+ Saturation to a negative value decreases the forward propagated variation & information → noise robustness
- More computationally expensive than ReLU

Here, α is 1.0





Machine Learning Research

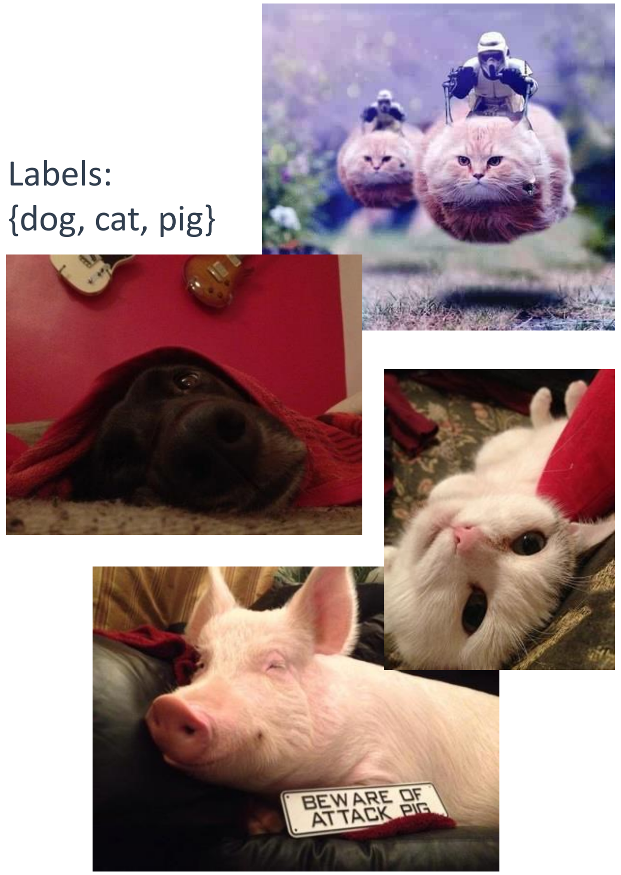
Feedforward neural networks
Hyperparameters | Activation functions

Multi-class / multinomial output

For classification, we use either a sigmoid activation function, for a binary classification problem, or, in case of more than two class categories, softmax:

- Softmax: $f(x)_i = \frac{e^{x_i}}{\sum_j^K e^{x_j}}$,
  where K is the number of classes and x is the logit of each output
- Allows us to transform the logits into the probability that an instance x belongs to class k
- But: regularisation (next lecture) leads to more diffuse softmax outputs
  → interpret the ordering of the score rather than the absolute numbers

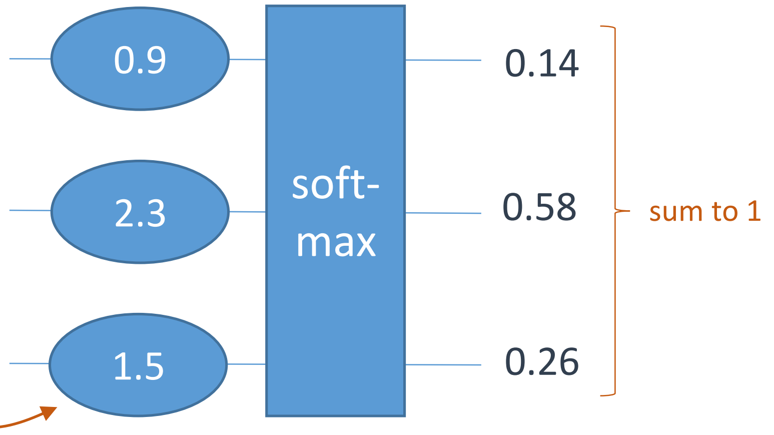
Labels:
{dog, cat, pig}



Feedforward neural networks
Hyperparameters | Activation functions

Multi-class / multinomial output

- Softmax: $f(x)_i = \frac{e^{x_i}}{\sum_j^K e^{x_j}}$,
  with K number of classes and x is the logit of each output
- Allows us to transform the logits into the probability that an instance x belongs to class k



The network output, which is then fed to a softmax or sigmoid function is also called *logit*

Feedforward neural networks
Hyperparameters

Number of layers: 3

Number of neurons first layer: 6

Number of neurons second & output layer: 4

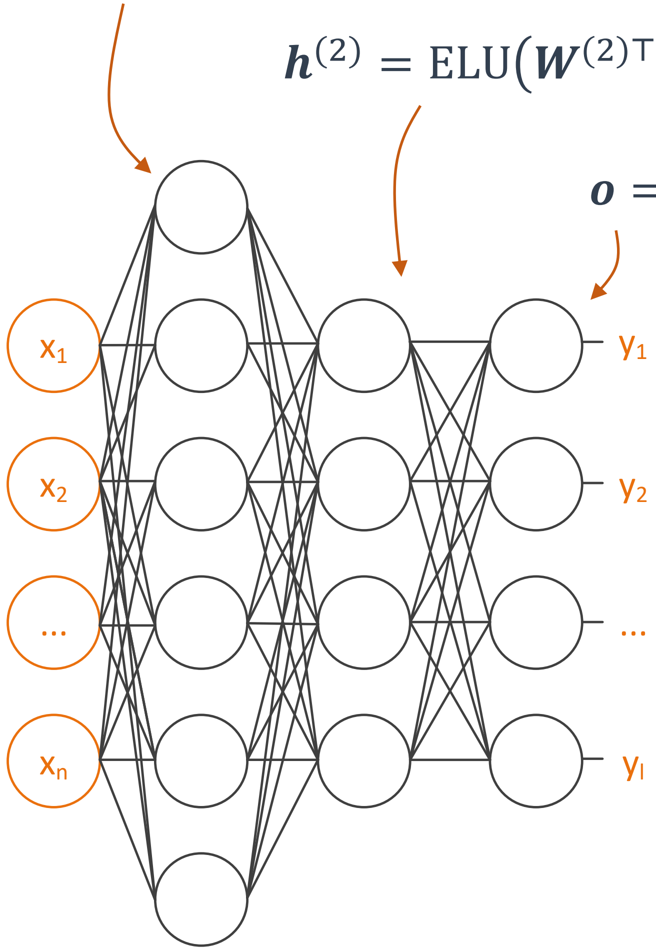
Activation function hidden layers: ELU

Activation function output: Softmax

$$\boldsymbol{h}^{(1)} = \mathrm{ELU}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$

$$\boldsymbol{h}^{(2)} = \mathrm{ELU}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

$$\boldsymbol{o} = \mathrm{softmax}\left(\boldsymbol{W}^{(3)\top}\boldsymbol{h}^{(2)} + \boldsymbol{b}^{(3)}\right)$$



Hyperparameters

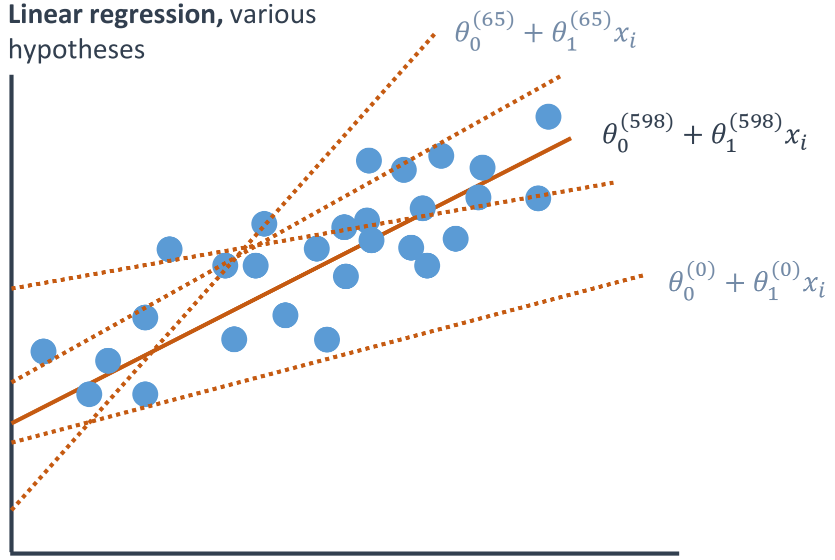
Learning as optimisation

What are hyperparameters for?

Finding an optimal network architecture can be seen as an optimisation task, where

- we have a hypothesis space of candidate models (networks with different parameter values)

Linear regression, various hypotheses



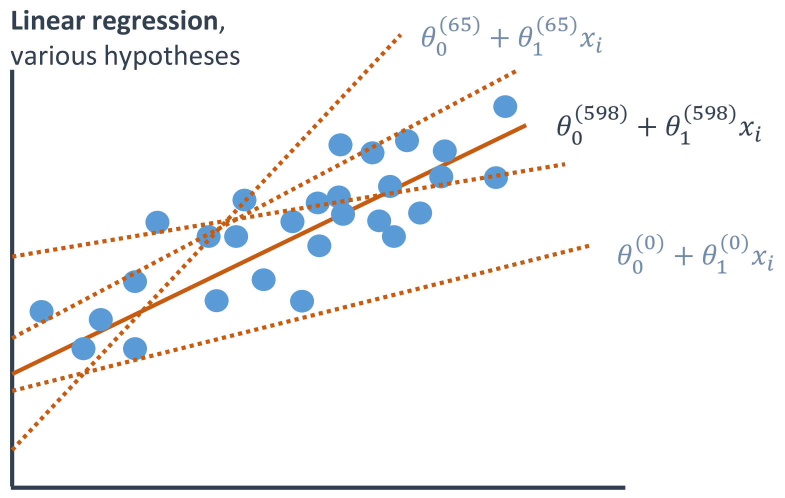
Learning as optimisation

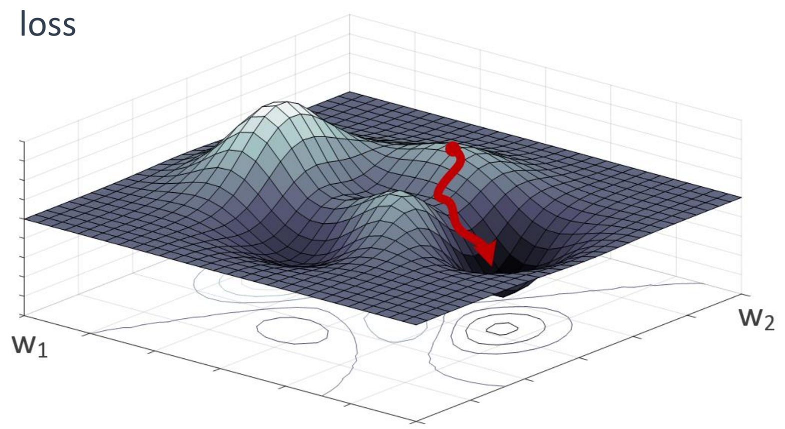
What are hyperparameters for?

Finding an optimal network architecture can be seen as an optimisation task, where

- we have a hypothesis space of candidate models (networks with different parameter values)
- and an objective function (the loss), which quantifies our preference for different models

→ Find a high-scoring model among the candidate models through network training



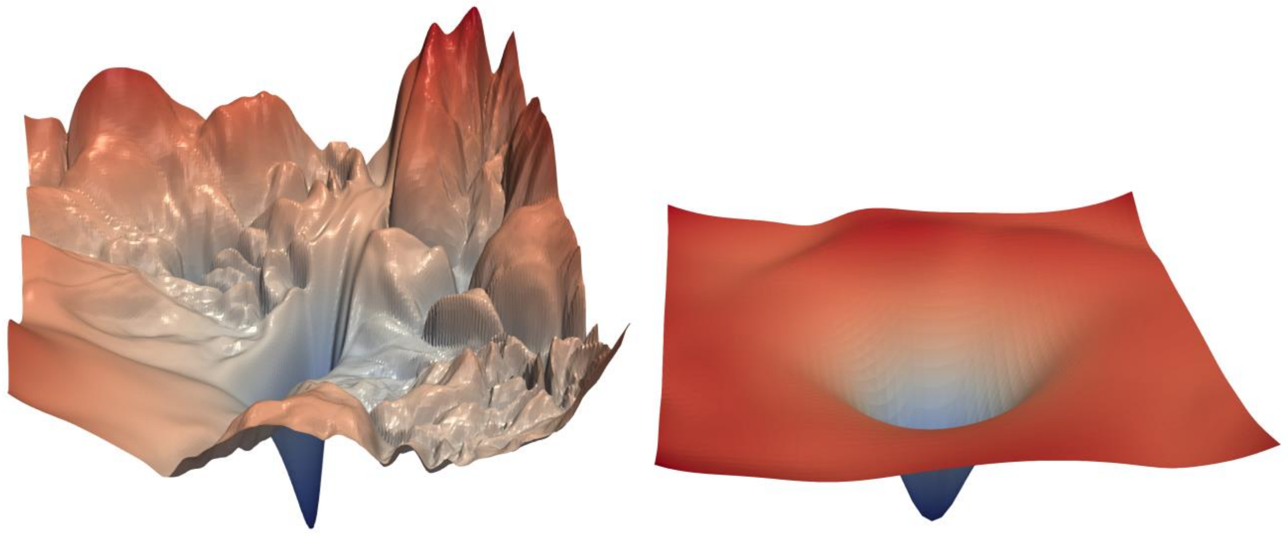


Learning as optimisation

What can we do to accelerate the search / reach the global minimum / a better local minimum?

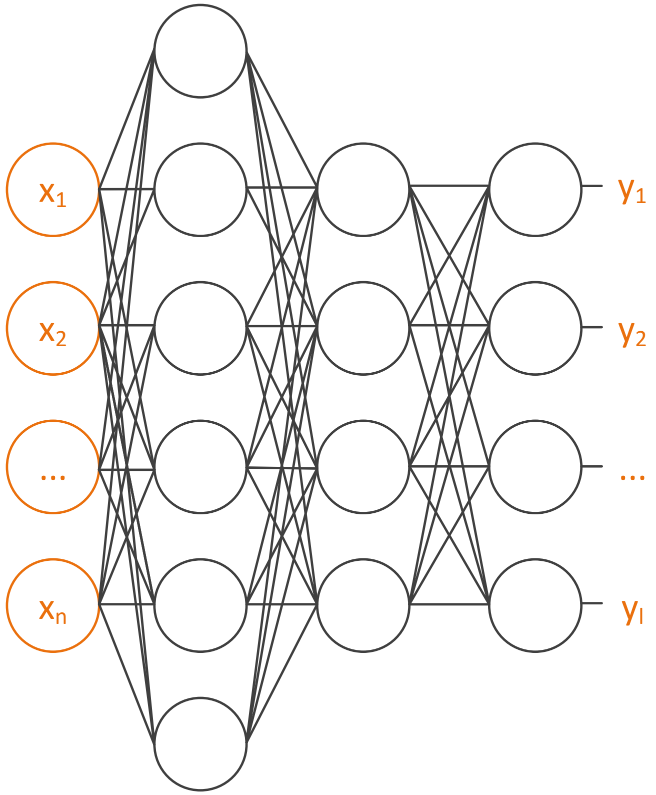
- Choose better steps towards the minimum
  - Parameter initialisation
  - Learning rate
  - Optimiser method, e.g. stochastic gradient descent
- Manipulate the loss landscape
  - Regularisation
- Soften the loss landscape
  - Batch norm
  - Skip connections

Loss landscape of a neural network



Hao Li et al. Visualizing the Loss Landscape of Neural Nets. (2018) https://arxiv.org/pdf/1712.09913.pdf

Parameter Initialisation



Feedforward neural networks
Hyperparameters | Parameter initialisation

Neural networks are strongly affected by the choice of parameter initialisation

- Initialisation must break symmetry, otherwise units will be updated in the same way
- Trade-off between optimisation (large weights) and generalisation (small weights)
- The initialisation choice depends on other hyperparameter choices

General initialisation strategy:

- Initialise weights randomly, drawn from a Gaussian or uniform distribution
- Initialise biases with heuristical chosen constants

Feedforward neural networks
Hyperparameters | Parameter initialisation

The problem: Vanishing gradients, especially in good-old fashioned sigmoid networks

- Sigmoid has a maximum derivative of 0.2 (and $0.2 * 0.2 = 0.04$), thus gradients get smaller and smaller during backpropagation

Glorot & Bengio: Understanding the difficulty of training deep feedforward neural networks. (2010)
He et al.: Diving deep into rectifiers: Surpassing human-level performance on ImageNet. (2015)

Feedforward neural networks
Hyperparameters | Parameter initialisation

The problem: Vanishing gradients, especially in good-old fashioned sigmoid networks

- Sigmoid has a maximum derivative of 0.2 (and $0.2 * 0.2 = 0.04$), thus gradients get smaller and smaller during backpropagation

Glorot / Xavier Initialisation:

- Initialise each layer with random numbers, drawn from a:
  - Gaussian distribution with mean 0 and variance $\sigma^2 = \frac{1}{fan_{\text{avg}}}$, with $fan_{avg} = \frac{fan_{\text{in}}+fan_{\text{out}}}{2}$
  - Uniform distribution between $\pm r$, with $r = \sqrt{\frac{3}{fan_{\text{avg}}}}$

number of inputs to a neuron

number of neurons

Glorot & Bengio: Understanding the difficulty of training deep feedforward neural networks. 2010

Feedforward neural networks
Hyperparameters | Parameter initialisation

The problem: Vanishing gradients, especially in old fashioned sigmoid networks

- Sigmoid has a maximum derivative of 0.2 (and $0.2 * 0.2 = 0.04$), thus gradients get smaller and smaller during backpropagation

Glorot / Xavier Initialisation:

- Initialise each layer with random numbers, drawn from a:
  - Gaussian distribution with mean 0 and variance $\sigma^2 = \frac{1}{fan_{\mathrm{avg}}}$, with $fan_{avg} = \frac{fan_{\mathrm{in}} + fan_{\mathrm{out}}}{2}$
  - Uniform distribution between $\pm r$, with $r = \sqrt{\frac{3}{fan_{\mathrm{avg}}}}$

number of inputs to a neuron

number of neurons

He initialisation:

- Gaussian distribution with mean 0 and variance $\sigma^2 = \frac{2}{fan_{\mathrm{in}}}$
- Uniform distribution between $\pm r$, with $r = \sqrt{3\sigma^2}$

Design rule:
Sigmoid, tanh, softmax, or linear activation functions: Glorot and biases with zero
ReLU and its variants: He and biases with 0.01

He et al.: Diving deep into rectifiers: Surpassing human-level performance on ImageNet. (2015)

Feedforward neural networks
Hyperparameters

Number of layers: 3

Number of neurons 1. layer: 6

Number of neurons 2. & output layer: 4

Activation function hidden layers: ELU

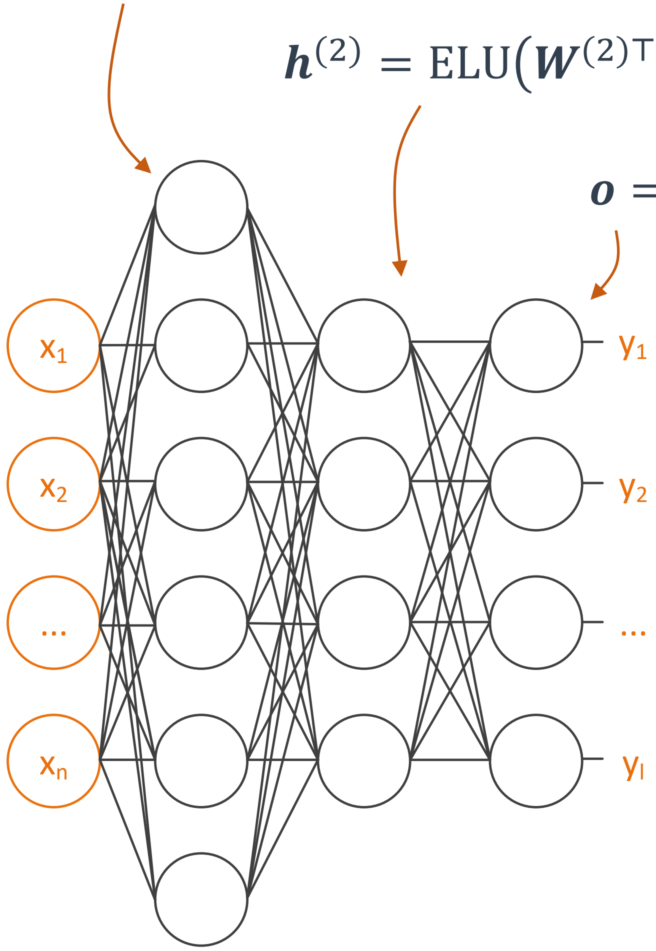
Activation function output: Softmax

Initialisation: He, biases with 0.1

$$\boldsymbol{h}^{(1)} = \mathrm{ELU}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$

$$\boldsymbol{h}^{(2)} = \mathrm{ELU}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

$$\boldsymbol{o} = \mathrm{softmax}\left(\boldsymbol{W}^{(3)\top}\boldsymbol{h}^{(2)} + \boldsymbol{b}^{(3)}\right)$$

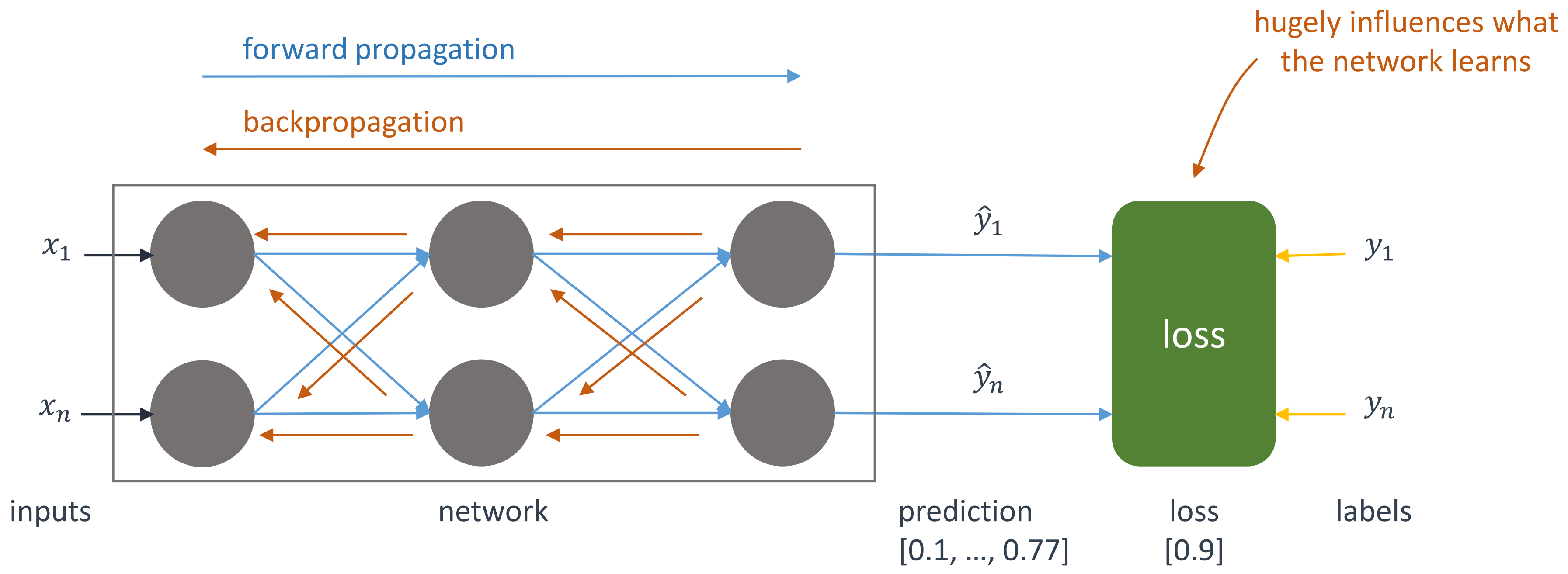


OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

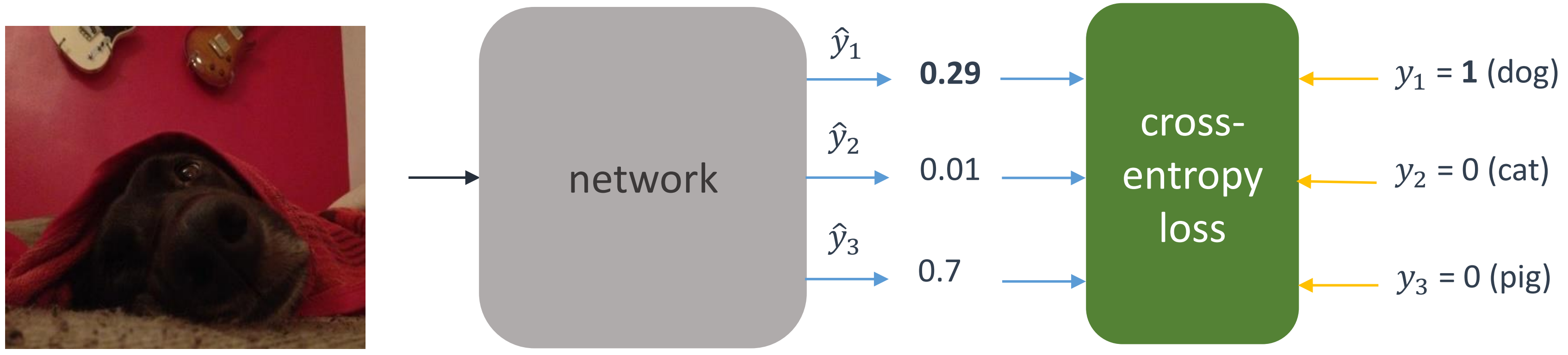
INFORMATIK UND
MATHEMATIK

Gradient Descent

Gradient descent: Update the network parameters (weights & biases) using backpropagation

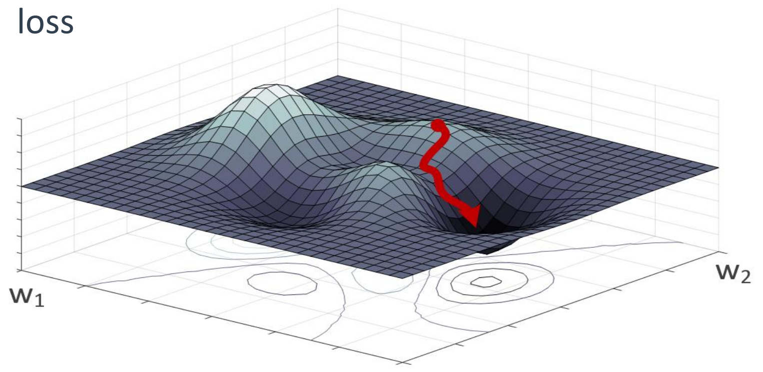


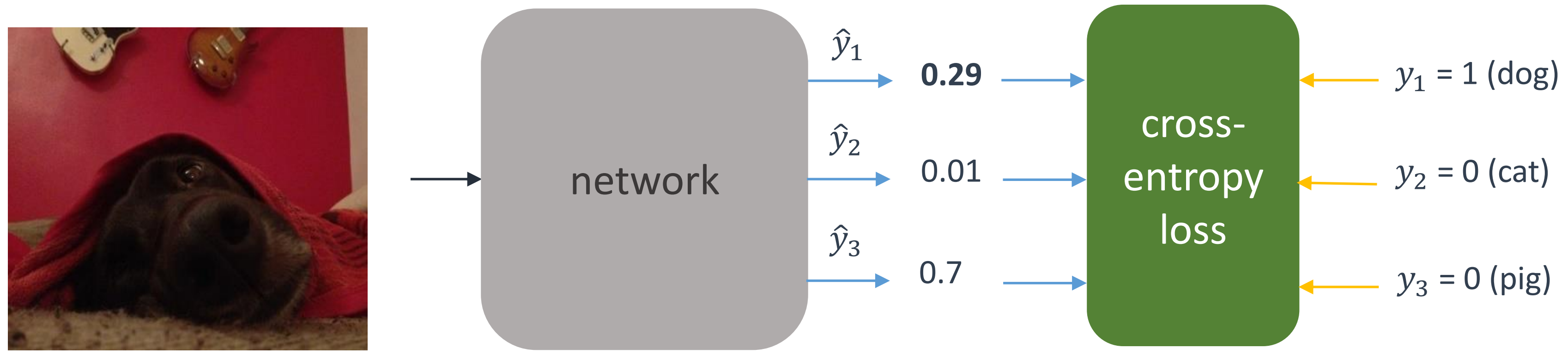
Feedforward neural networks
Gradient descent



input

Feedforward neural networks
Gradient descent





input

Feedforward neural networks
Backpropagation | Chain rule

Recap from lecture 3 | the parameter view:

- Gradient descent optimisation: To update the network parameters, we repeatedly move into the direction of the steepest descent, that is the direction of the negative gradient
  - $w \leftarrow w - \eta \Delta w$
  - $\mathrm{b} \leftarrow b - \eta \Delta b$

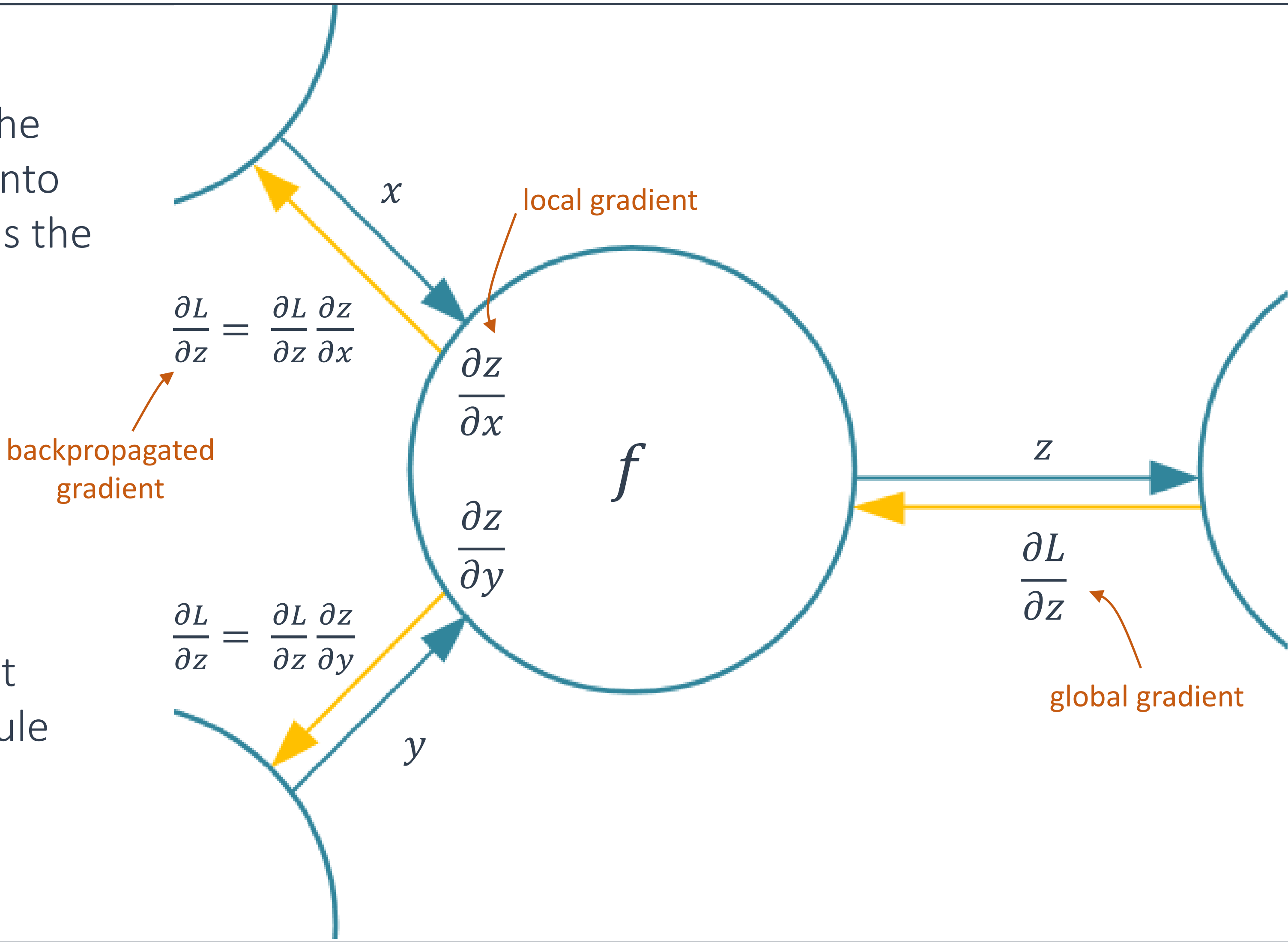
new estimate

old parameter estimate

learning rate

desirable direction to minimise the error

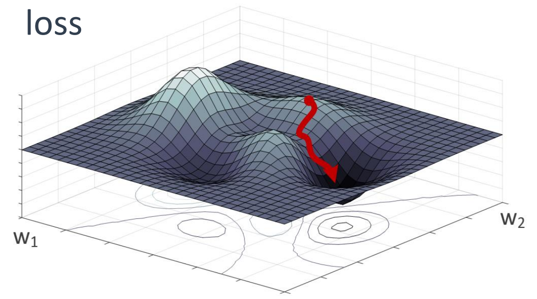
- Computationally, we propagate the gradient backwards through the network via chain rule

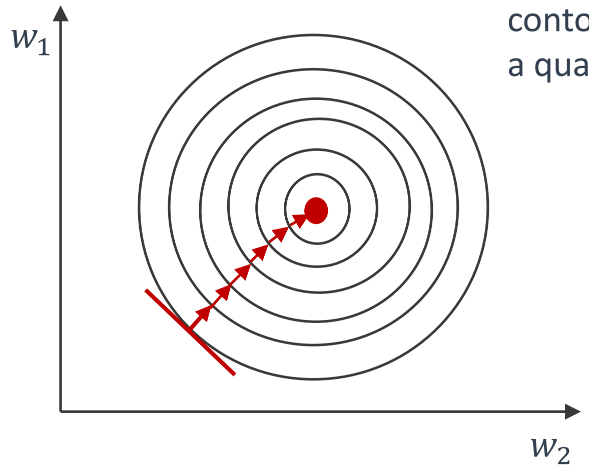

x
local gradient
$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial x}$
f
z
$\frac{\partial z}{\partial y}$
backpropagated gradient
$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial z}$
global gradient
y

Feedforward neural networks
Gradient descent

The loss landscape view:

- Gradient descent iteratively approaches the loss function‘s minimum by moving into the perpendicular direction with respect to the contours of the loss function



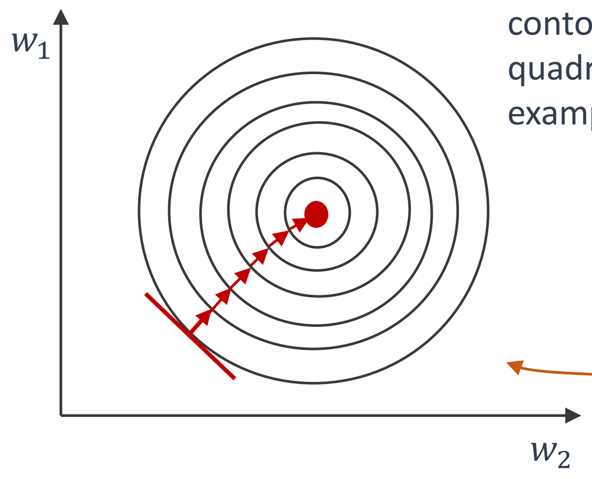


contour lines of a quadratic loss

Feedforward neural networks
Gradient descent

Neural networks are usually trained with hundreds to millions of examples

- Batch gradient descent: Computes the gradients based on the loss of all examples at once
  - + Stable convergence: Guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces
  - - Slow learning: Very long training time (if the training set does fit into memory anyway)
  - - Can get stuck in local minima and saddle points



contour lines of the quadratic loss of all examples in the dataset

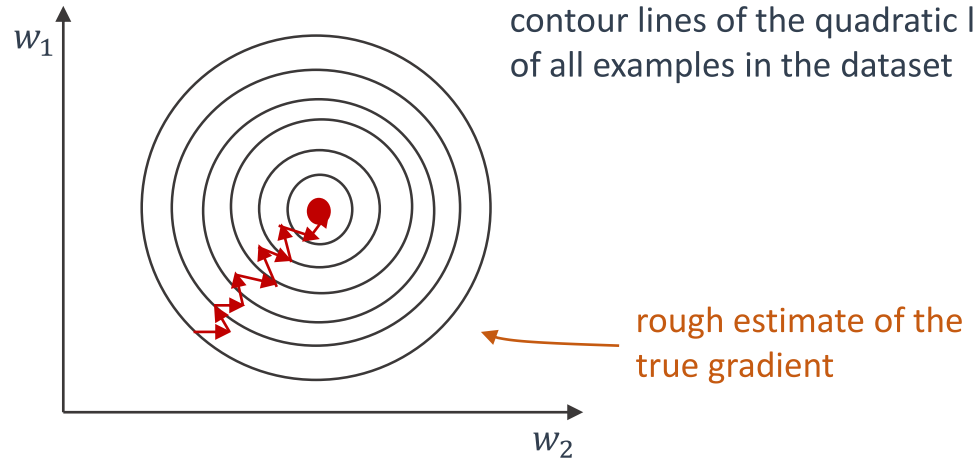
averaged gradient of all examples in the dataset

$$\theta_j \leftarrow \theta_j - \eta \frac{1}{n} \sum_{i=1}^{n} \nabla_{\theta_j} \mathcal{L}(y_i, \hat{y}_i)\,,$$
n is the number of examples

Feedforward neural networks
Gradient descent

Neural networks are usually trained with hundreds to millions of examples

- Batch gradient descent: Computes the gradients based on the loss of all examples at once
- Stochastic gradient descent (SGD) / Online learning: one random example at a time
  - + Very fast & enables the training on very large datasets
  - + Noisy gradients: Avoid local minima and saddle points
  - - Noisy gradients: Zig-zaging optimisation steps
  - - Reaches good but not optimal parameter values


w_1
w_2
contour lines of the quadratic loss of all examples in the dataset
rough estimate of the true gradient

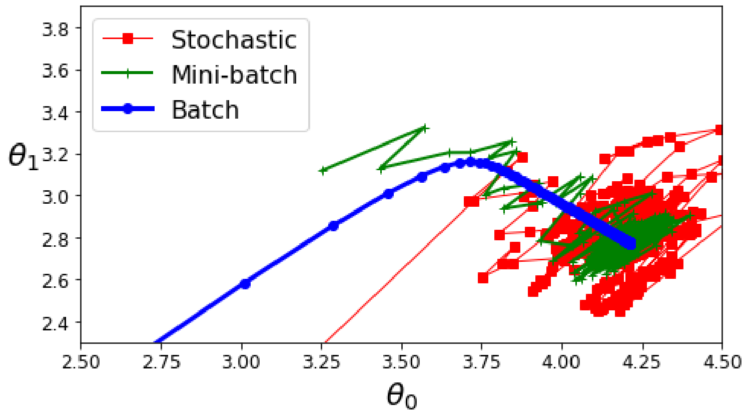
$$\theta_j \leftarrow \theta_j - \eta \nabla_{\theta_j} \mathcal{L}(y_i, \hat{y}_i)$$

Machine Learning Research

Feedforward neural networks
Mini-batch gradient descent

Neural networks are usually trained with hundreds to millions of examples

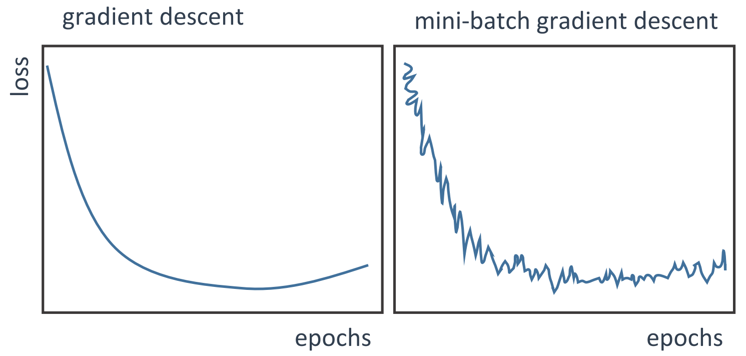
- Batch gradient descent: Computes the gradients based on the loss of all examples at once
- Stochastic gradient descent (SGD) / Online learning : one random example at a time
- Mini-batch gradient descent: small, random sets of examples
  - + Also very fast
  - + Noisy gradients: Avoid local minima and saddle points
  - + Stable convergence: Less erratic than SGD and thus reaches more optimal values
  - + Using batch sizes of 2^n exploits performant GPU matrix operations
  - - Batch size as a hyperparameter



$$\theta_j \leftarrow \theta_j - \eta \frac{1}{m} \sum_{i=mk}^{(k+1)m} \nabla_{\theta_j} \mathcal{L}(y_i, \hat{y}_i)\,,$$
m is the batch size,
$k = \{1, \frac{n}{m}\}$ is the number of batches

Feedforward neural networks
Mini-batch gradient descent

- Mini-batch gradient descent: Computes the gradients based on the loss of small, random sets of examples
  - typically 2 – 1024 examples per mini-batch
  - Each iteration of a mini-batch is called a step
  - One iteration of all mini-batches (that is the whole dataset) is called an episode
  - The number of episodes depends on your dataset, network, hyperparameters,…

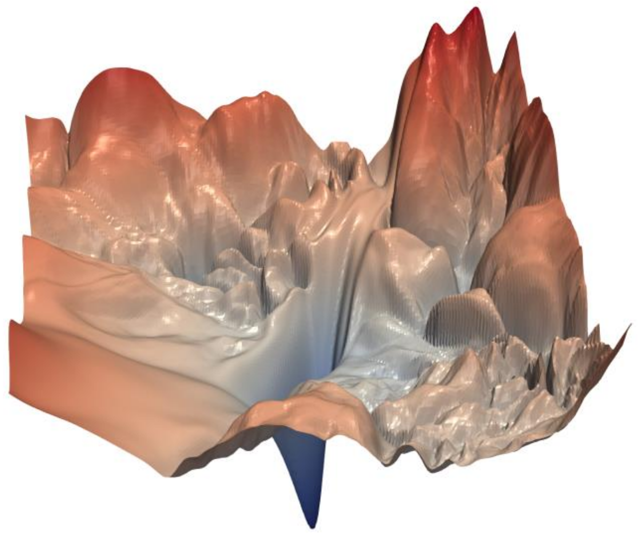


Design rule:
Typically 2 – 1024 examples per mini-batch, more or less depending on the data dimensions and memory.

Feedforward neural networks Optimisation

The real loss landscape view:

- Neural network loss landscapes typically have millions of parameters, not just two, and are badly conditioned*

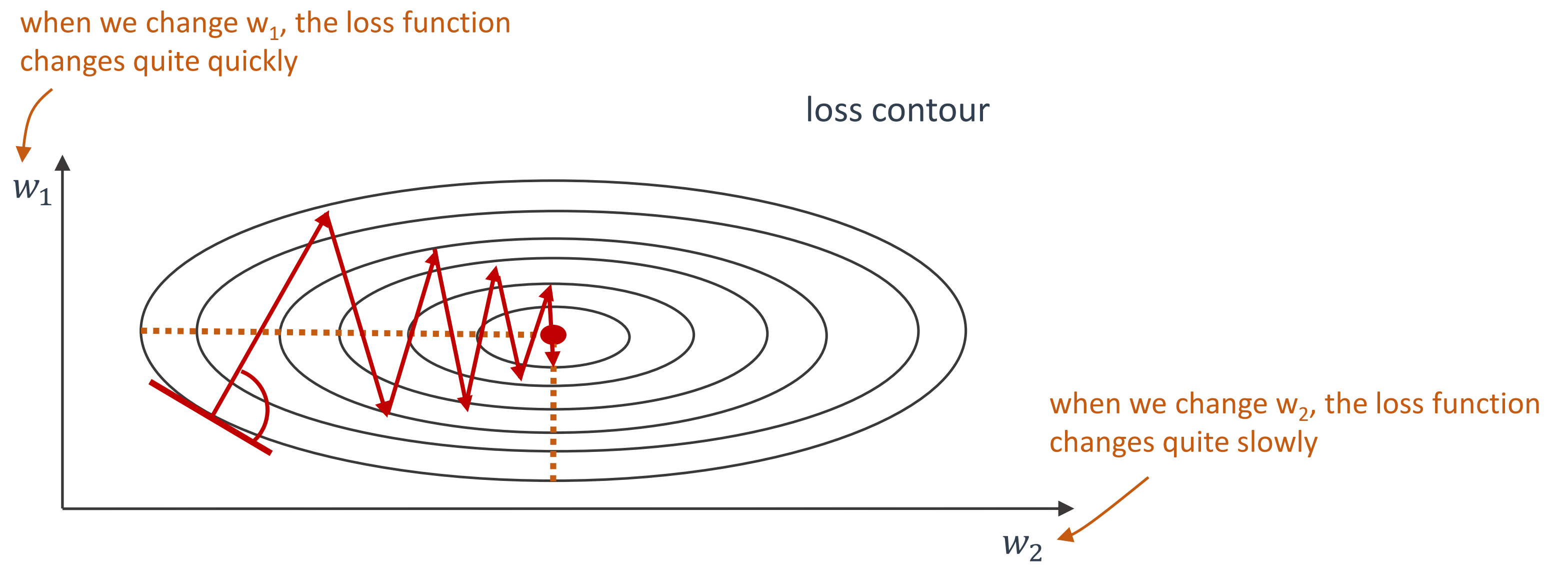


Hao Li et al. Visualizing the Loss Landscape of Neural Nets. (2018) https://arxiv.org/pdf/1712.09913.pdf
*See 2nd derivatives / Hessian, e.g. in Goodfellow et al.: Deep Learning, pages 83 pp

Feedforward neural networks
Optimisation

The real loss landscape view:

- Neural network loss landscapes typically have millions of parameters, not just two, and are badly conditioned
- Gradient descent if the loss changes quickly in one direction and slowly in the other:
  - Slow progress along the slowly changing direction
  - Zig-zagging progress along the quickly changing direction

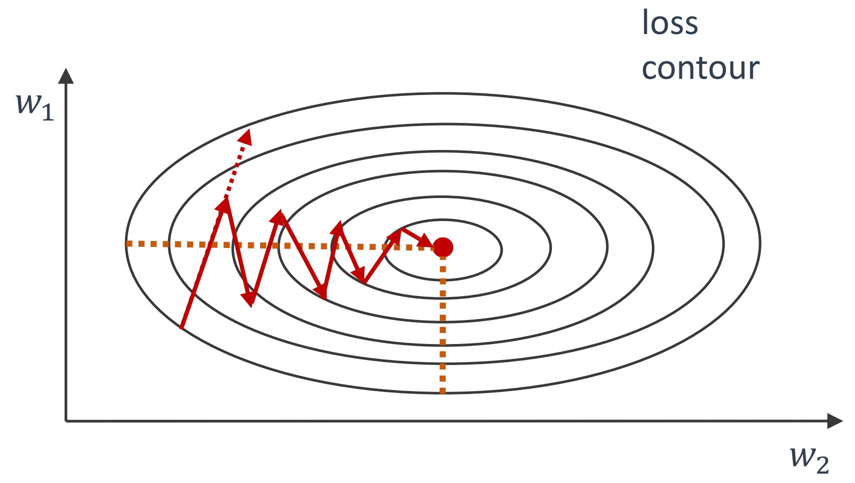

when we change w_1, the loss function changes quite quickly
loss contour
w_1
w_2
when we change w_2, the loss function changes quite slowly

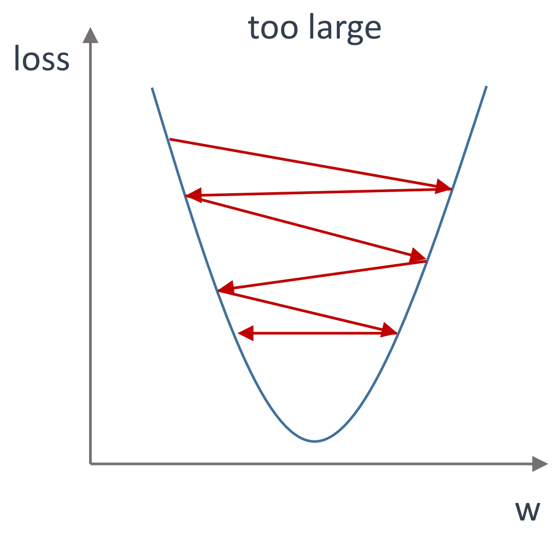
Learning Rate

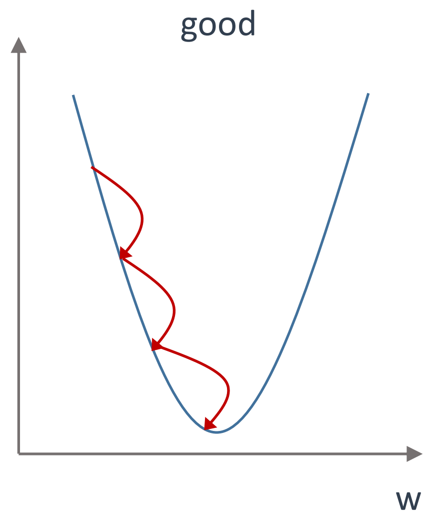
Learning rate

One solution: Adjust the learning rate: $\theta_j \leftarrow \theta_j - \eta \nabla_{\theta_j} \mathcal{L}(y_i, \hat{y}_i)$

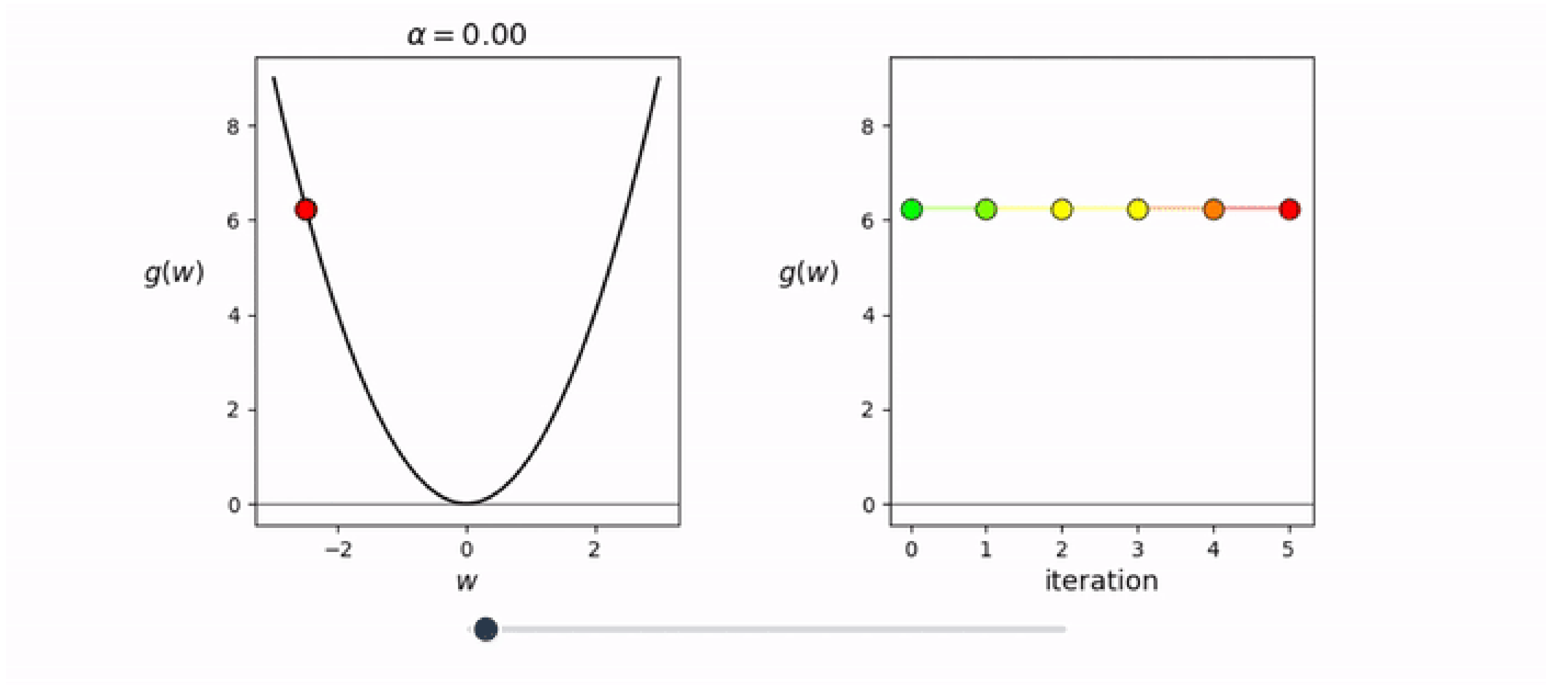
- A smaller learning rate diminishes the overshooting







Learning rate

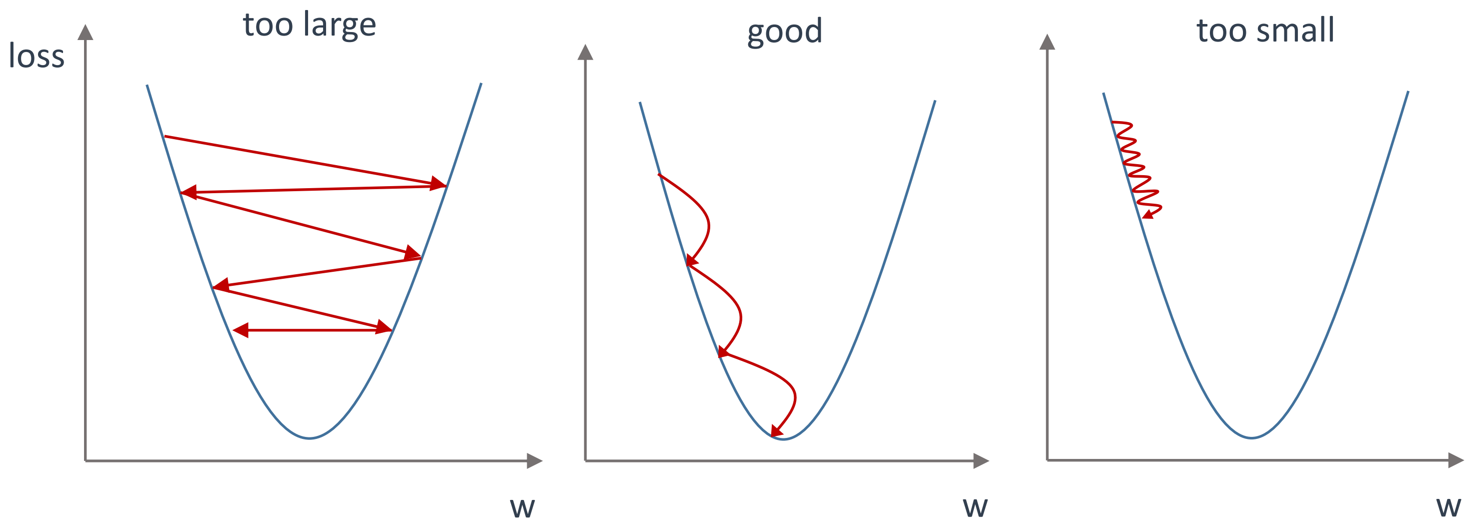


Watt et al. Machine Learning Refined. (2020) https://github.com/jermwatt/machine_learning_refined

Learning rate

Adjust the learning rate

- A smaller learning rate diminishes the overshooting
- However: the smaller the learning rate the longer the training process

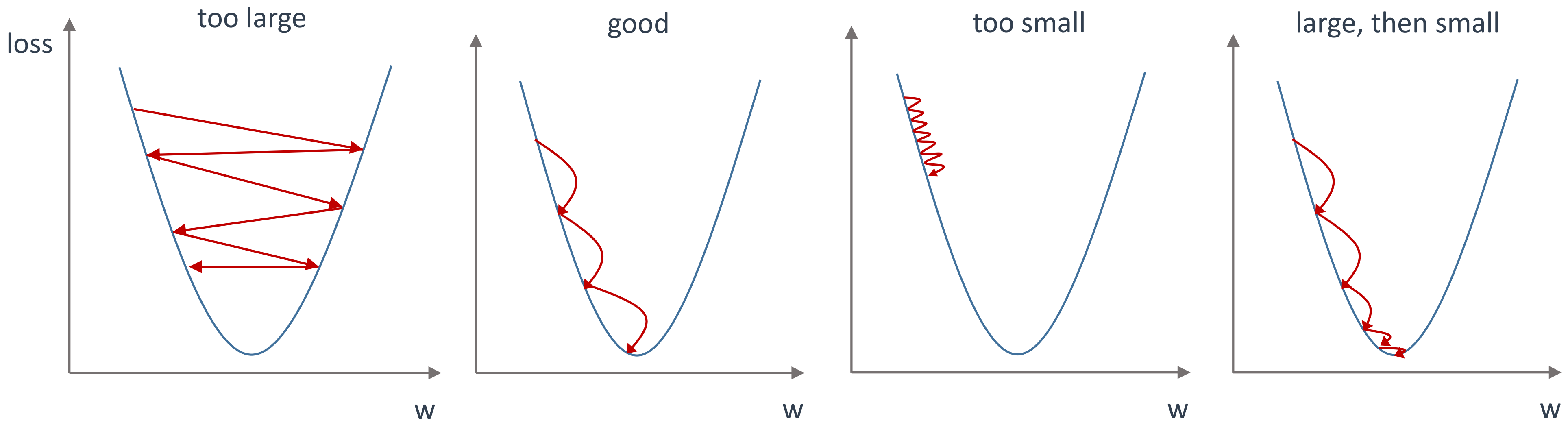


Learning rate

One solution: Adjust the learning rate

- A smaller learning rate diminishes the overshooting
- However: the smaller the learning rate the longer the training process

→ learning rate schedules

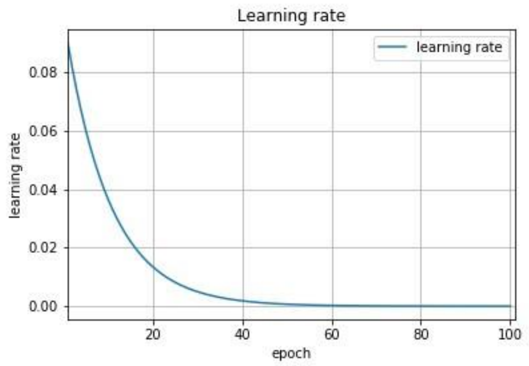


Learning rate schedules

- Piecewise constant scheduling
  1. Constant learning rate for a number of epochs, e.g. $\eta_0 = 0.01$ for 5 epochs
  2. Then, lower the learning rate and use it for another number of epochs, e.g. $\eta_1 = 0.0001$ for 30 epochs
  3. And so on
  - Hyperparameters: learning rates, number of epochs for each learning rate

Learning rate schedules

- Piecewise constant scheduling
    1. Constant learning rate for a number of epochs, e.g. $\eta_0 = 0.01$ for 5 epochs
    2. Then, lower the learning rate and use it for another number of epochs, e.g. $\eta_1 = 0.0001$ for 30 epochs
    3. And so on
    - Hyperparameters: learning rates, number of epochs for each learning rate
- Exponential scheduling
    - $\eta(t) = \eta_0\ 0.1^{\frac{t}{s}}$, where the learning rate will gradually drop by a factor of 10 every s steps. t is the current step (iteration) iteration of the network training.
    - Hyperparameters: decay steps s, initial learning rate η_0, decay rate (here 0.1)



Learning rate schedules

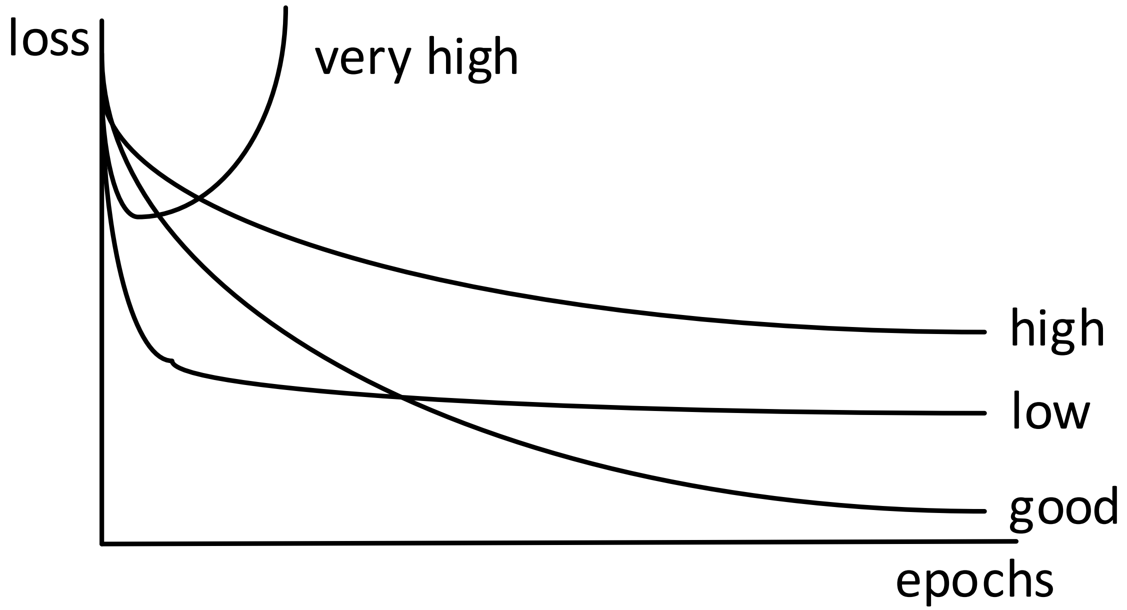
- Piecewise constant scheduling
  1. Constant learning rate for a number of epochs, e.g. $\eta_0 = 0.01$ for 5 epochs
  2. Then, lower the learning rate and use it for another number of epochs, e.g. $\eta_1 = 0.0001$ for 30 epochs
  3. And so on
  - Hyperparameters: learning rates, number of epochs for each learning rate
- Exponential scheduling
  - $\eta(t) = \eta_0\, 0.1^{\frac{t}{s}}$, where the learning rate will gradually drop by a factor of 10 every s steps. t is the current step (iteration) iteration of the network training.
  - Hyperparameters: decay steps s, initial learning rate η_0, decay rate (here 0.1)
- Polynomial decay
- Inverse time decay
- 1 cycle
- …

Design rule:
The learning rate is one of the most important hyperparameters.
LR values: from 0.1 to $1 \cdot 10^{-6}$

https://keras.io/api/optimizers/learning_rate_schedules/
Leslie Smith, Cyclical learning rates for training neural networks (2015) https://arxiv.org/abs/1506.01186

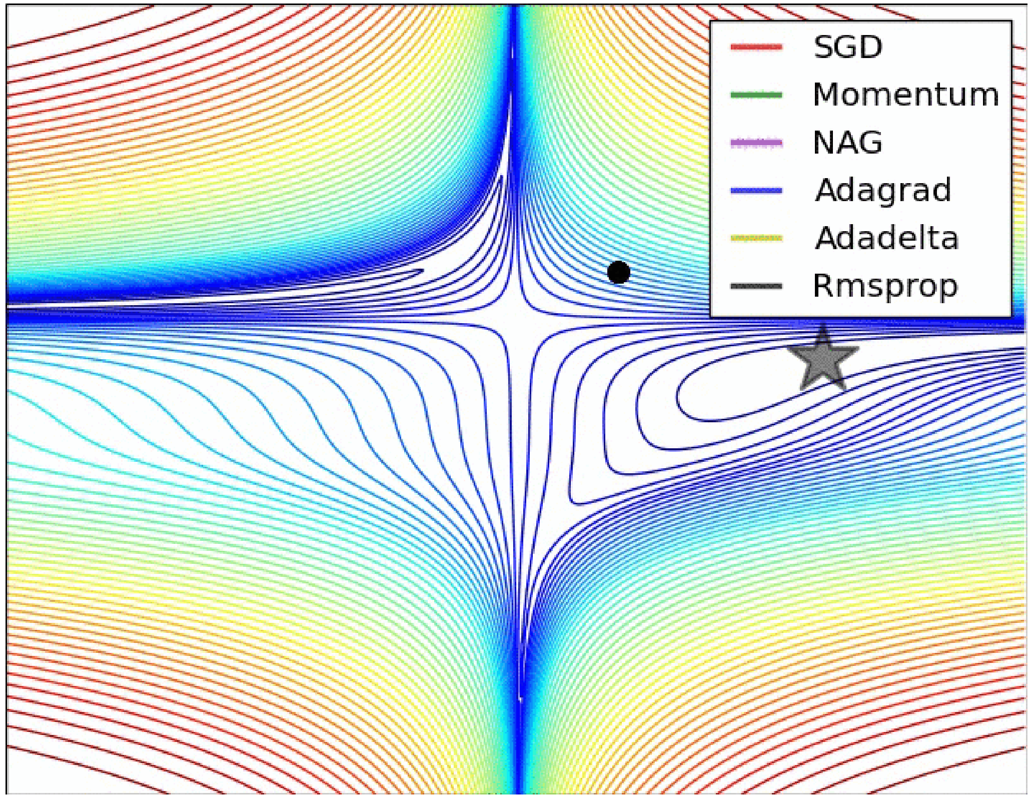
Learning rate and network training

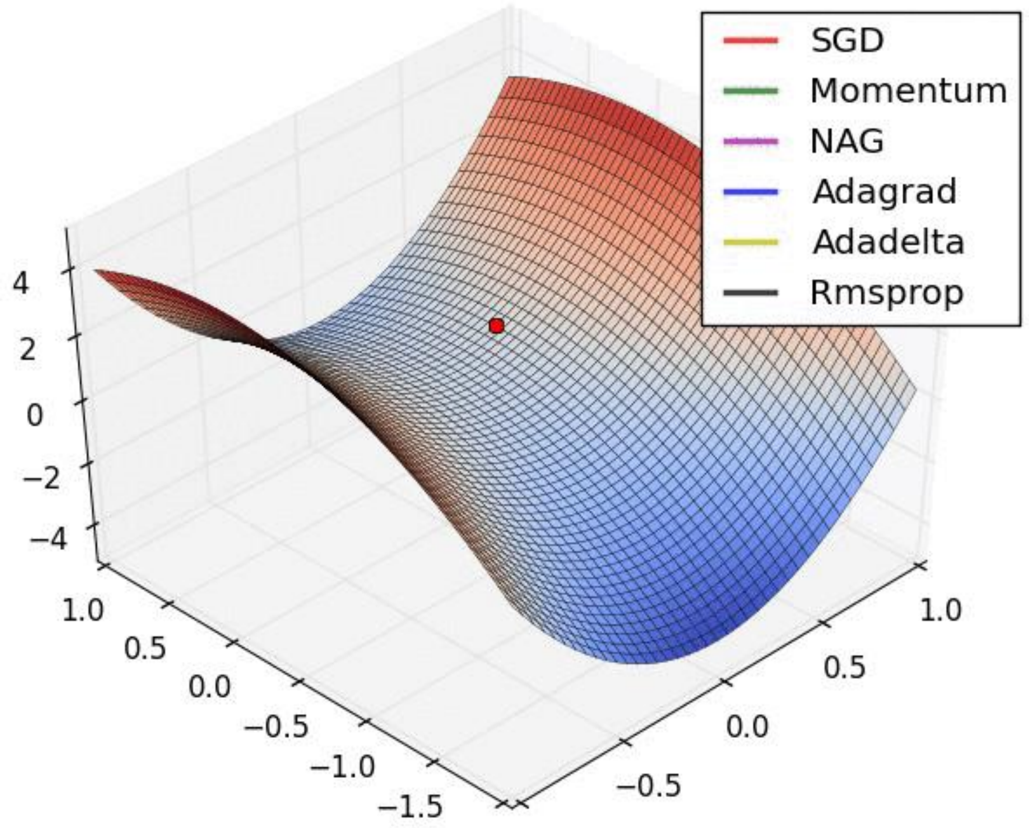
Observing the training loss over network training can give some indication on how good the learning rate value is



Optimiser

Optimiser

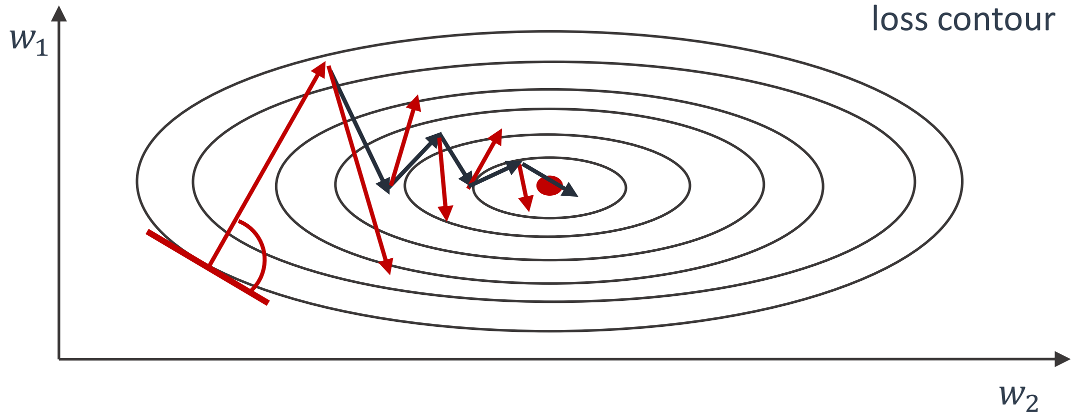




Sebastian Ruder. An overview of gradient descent optimization algorithms. https://ruder.io/optimizing-gradient-descent/

OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG
INFORMATIK UND MATHEMATIK

Optimiser Momentum



Momentum

- Accumulates the exponentially decaying moving sum of past gradients and continues to move in their direction
- $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} + \eta\nabla_{\theta_j}\mathcal{L}(y_i, \hat{y}_i)$, with $\alpha \in [0,1)$
- $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \boldsymbol{v}$

Typically 0.9
0: high friction
1: no friction

The larger α relative to η the more affect previous gradients the current direction

Velocity
$\boldsymbol{v}$ is largest, if many successive gradients point in the same direction

+ Diminishes the zig-zagging behaviour

- Overshoots

Machine Learning Research

Optimiser
AdaGrad & RMSprop

AdaGrad (Adaptive Gradient)

$$v_i \leftarrow v_i + (\Delta\theta_i)^2$$

$$\theta_i \leftarrow \theta_i - \frac{\eta}{\sqrt{v_i + \epsilon}} \Delta\theta_i$$

- Optimises the layer-specific partial derivative magnitutes by individually decaying the learning rate
- Faster decay for steep dimensions than dimensions with gentler slopes points the resulting updates more directly towards the optimum

+ More robust

- Dividing the learning rate η by the accumulated partial derivatives v_i may prematurely slow down learning

Optimiser
AdaGrad & RMSprop

AdaGrad (Adaptive Gradient)

$$v_i \leftarrow v_i + (\Delta\theta_i)^2$$

$$\theta_i \leftarrow \theta_i - \frac{\eta}{\sqrt{v_i + \epsilon}} \Delta\theta_i$$

- Selectively scales down the gradient vector via the learning rate along the steepest dimensons
- Points the resulting updates more directly towards the optimum

+ More robust

- Dividing the learning rate η by the accumulated partial derivatives v_i may prematurely slow down learning

RMSProp

decay rate, typically 0.9

$$v_i \leftarrow \beta v_i + (1 - \beta)(\Delta\theta_i)^2$$

$$\theta_i \leftarrow \theta_i - \frac{\eta}{\sqrt{v_i + \epsilon}} \Delta\theta_i$$

+ Fixes AdaGrad by using an exponentially decaying average to discard history from the extreme past

Optimiser ADAM

$m_i \leftarrow \beta_1 m_i + (1-\beta_1)\Delta\theta_i,$ Momentum

with $\widehat{m} = \frac{m}{(1-\beta_1^t)}$ Corrects the bias introduced by initialising m and b with 0

$v_i \leftarrow \beta_2 v_i + (1-\beta_2)(\Delta\theta_i)^2,$ Adaptive learning rate

with $\hat{v} = \frac{v}{(1-\beta_2^t)}$

$\theta_i \leftarrow \theta_i - \frac{\eta_\beta}{\sqrt{\hat{v}_i + \epsilon}}\widehat{m}_i$ Update rule

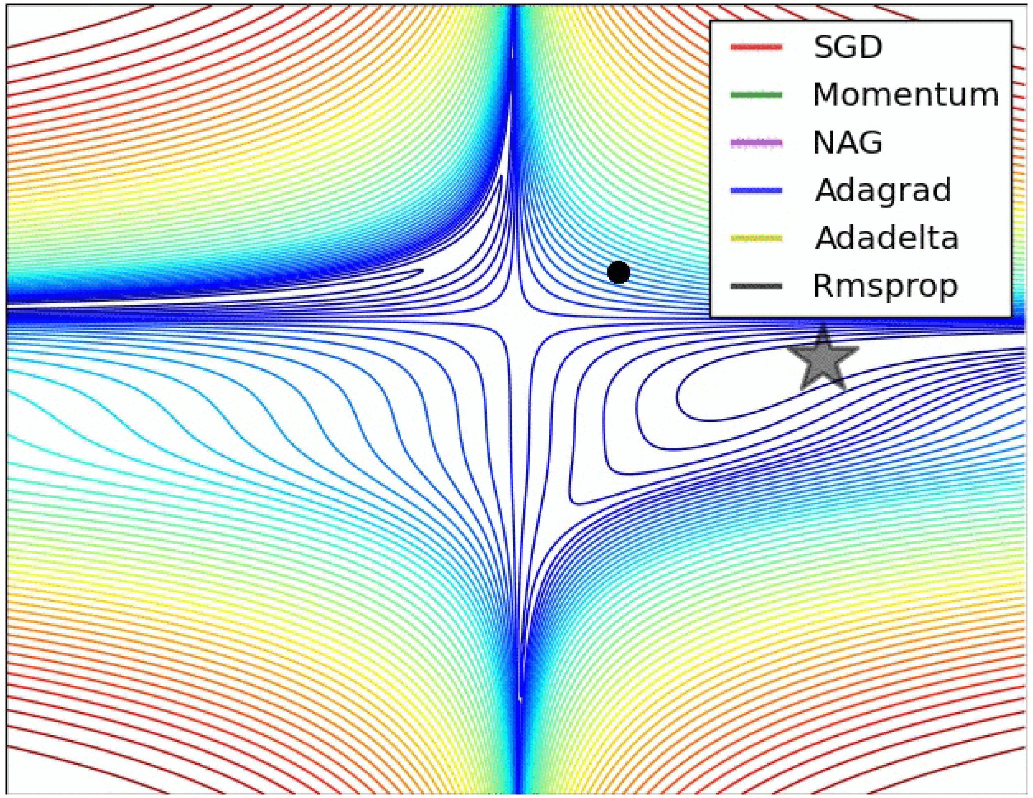
- Combines the ideas of momentum optimisation and RMSProp
- Momentum: accelerate the gradient based on an exponentially decaying average
- RMSProp: Selectively adapt the learning rate

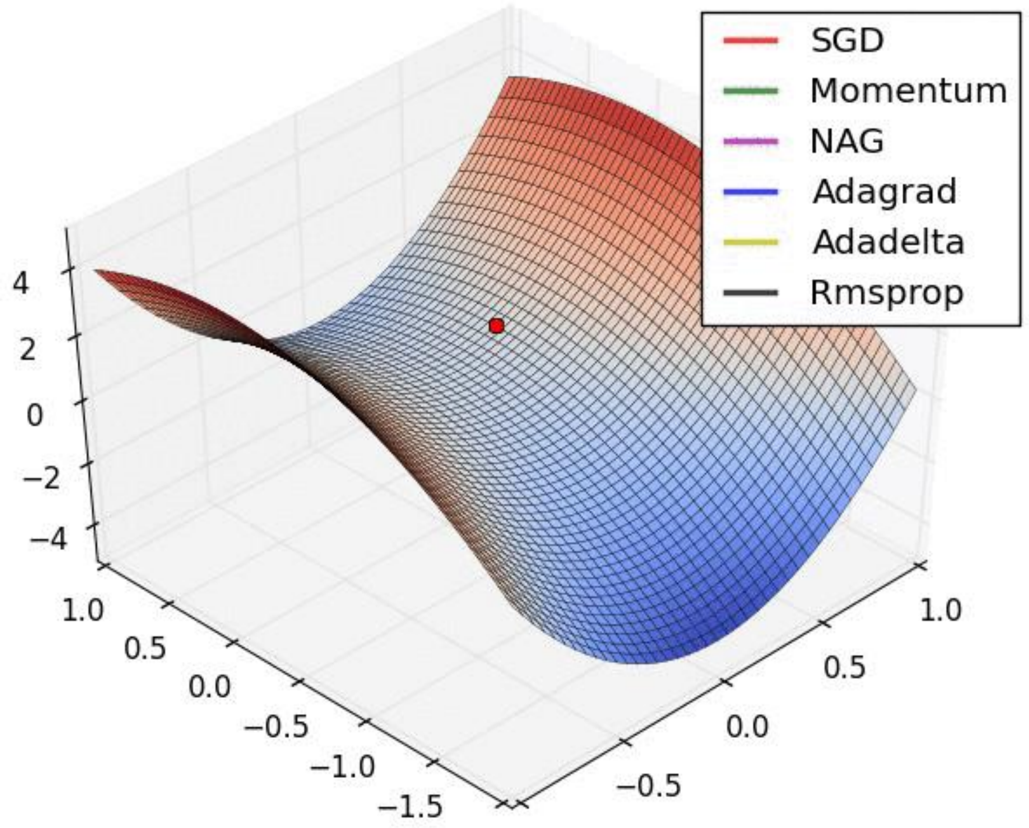
Momentum decay hyperparameter β_1 is typically 0.9
Learning rate decay hyperparameter β_2 is typically 0.999

Design rule:
Feed forward nets: Momentum,
Convolutional Neural Nets:
Adam optimiser with an initial learning rate $\eta = 0.001$ is a good start

Machine Learning Research

Optimiser





Sebastian Ruder. An overview of gradient descent optimization algorithms. https://ruder.io/optimizing-gradient-descent/

Feedforward neural networks
Architecture & some hyperparameters | Conclusion

Number of layers: 3

Number of neurons 1. layer: 6

Number of neurons 2. & output layer: 4

Activation function hidden layers: ELU

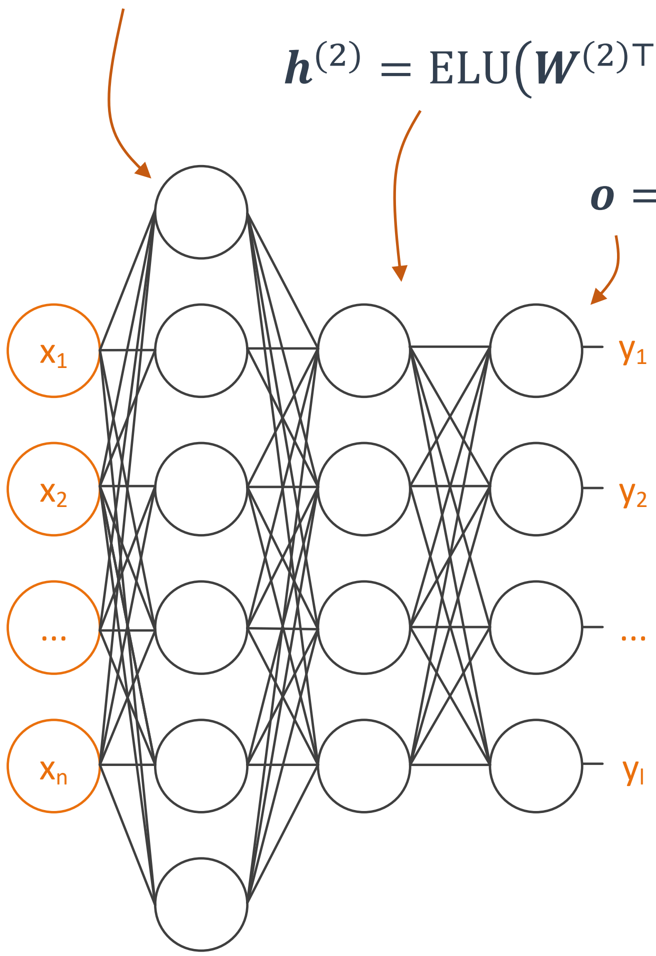
Activation function output: Softmax

Initialisation: He, biases with 0.1

$$\boldsymbol{h}^{(1)} = \mathrm{ELU}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$

$$\boldsymbol{h}^{(2)} = \mathrm{ELU}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

$$\boldsymbol{o} = \mathrm{softmax}\left(\boldsymbol{W}^{(3)\top}\boldsymbol{h}^{(2)} + \boldsymbol{b}^{(3)}\right)$$



Feedforward neural networks
Architecture & some hyperparameters | Conclusion

Number of layers: 3

Number of neurons 1. layer: 6

Number of neurons 2. & output layer: 4

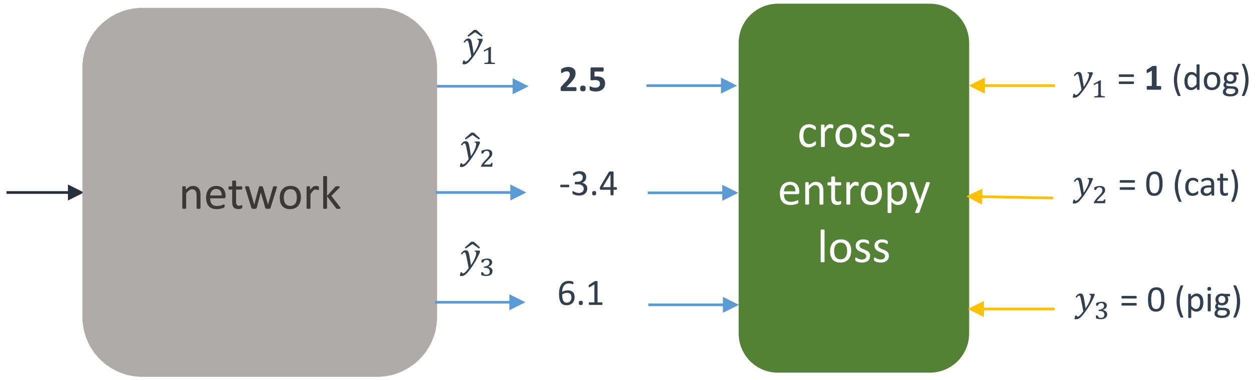
Activation function hidden layers: ELU

Activation function output: Softmax

Initialisation: He, biases with 0.1

Learning rate schedule: Exponential decay

Optimiser: Momentum



input

Feedforward neural networks
Resources

- Ian Goodfellow, Yoshua Bengio and Aaron Courville, Deep Learning
  https://www.deeplearningbook.org/
- Charu Aggarwal, Neural Networks and Deep Learning
  https://www.springer.com/de/book/9783319944623
- Michael Nielsen, Neural Networks and Deep Learning
  http://neuralnetworksanddeeplearning.com/index.html (online book)
- Nando de Freitas, Deep Learning Lecture
  https://www.youtube.com/watch?v=PlhFWT7vAEw&list=PLE6Wd9FR--EfW8dtjAuPoTuPcqmOV53Fu&index=16
- Hugo Larochelle, Neural Networks, online lecture
  https://www.youtube.com/watch?v=SGZ6BttHMPw&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH
- Sebastian Raschka, Intro to Machine Learning
  https://www.youtube.com/watch?v=OgK8JFjkSto&list=PLTKMiZHVd_2KyGirGEvKlniaWeLOHhUF3
- Allison George, Neural Networks from Scratch (nice interactive playground)
  https://aegeorge42.github.io/