Machine Learning in Bioinformatics

Artificial Neural Networks

Philipp Benner
philipp.benner@bam.de

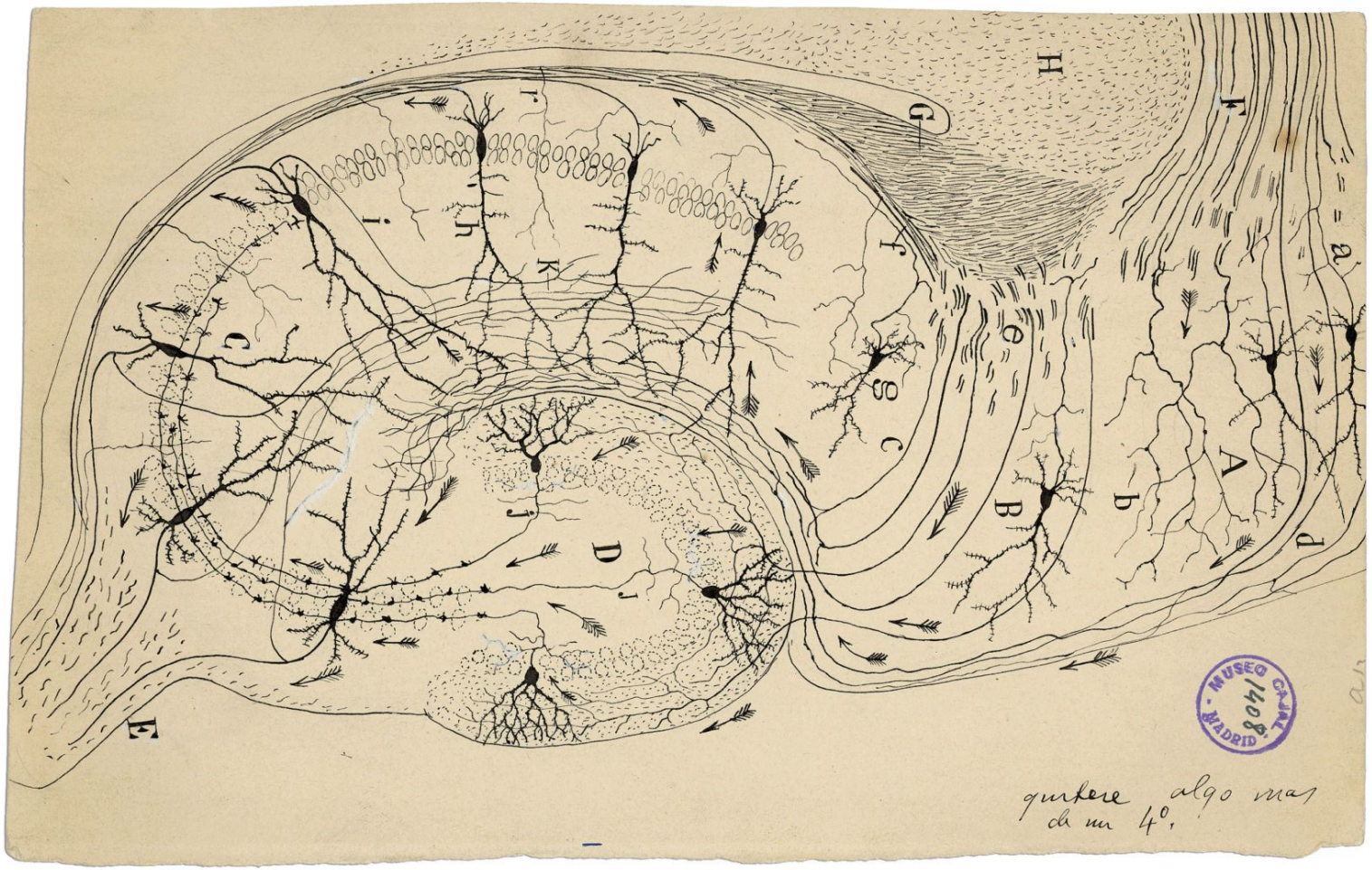
S.3 - eScience
Federal Institute for Materials Research and Testing (BAM)

July 22, 2023

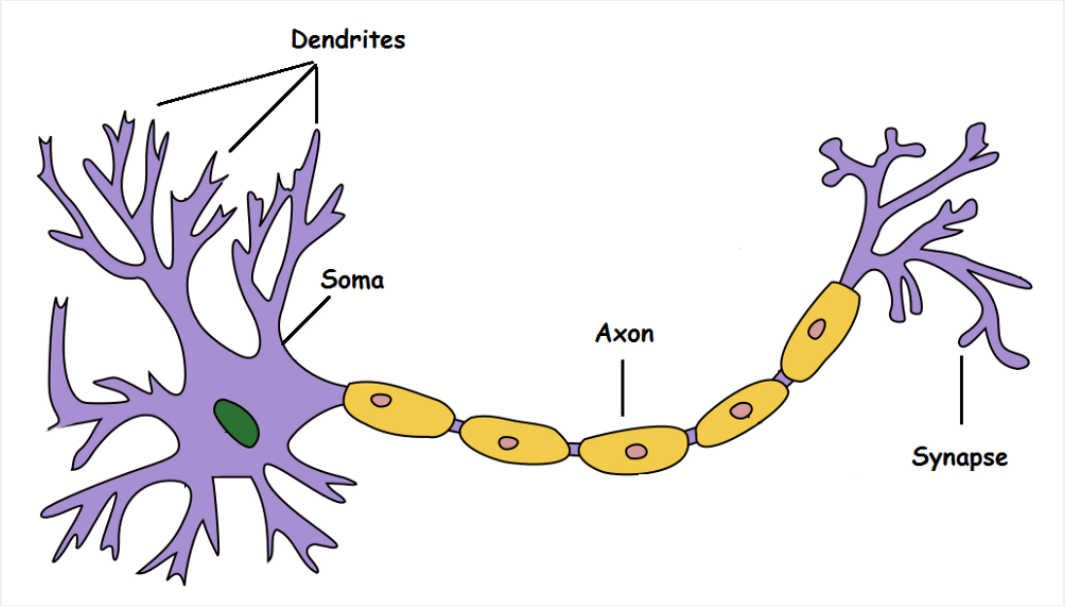
Outline

- We will discuss a particular class of *Artificial Neural Networks (ANNs)* called *Multilayer Perceptrons (MLPs)*
- MLPs are feed-forward networks, i.e. without any loops (directed acyclic graphs)
- We motivate MLPs from logistic regression

Drawings by Ramon y Cajal (∼ 1900)



A BIOLOGICAL NEURON

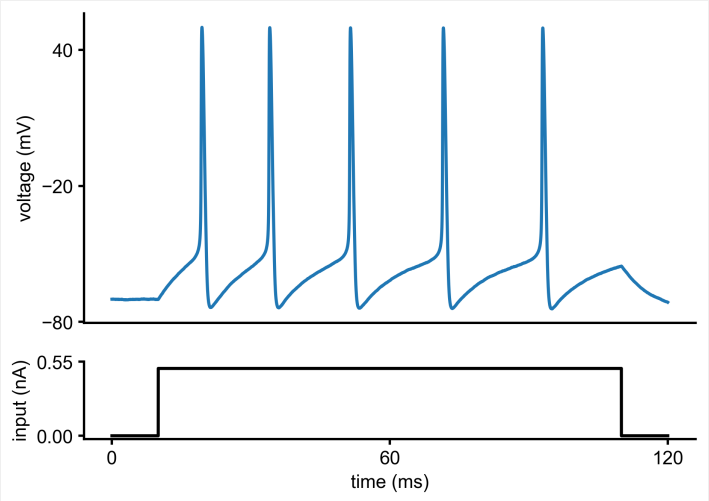


Hodgkin-Huxley model [Hodgkin and Huxley, 1952]

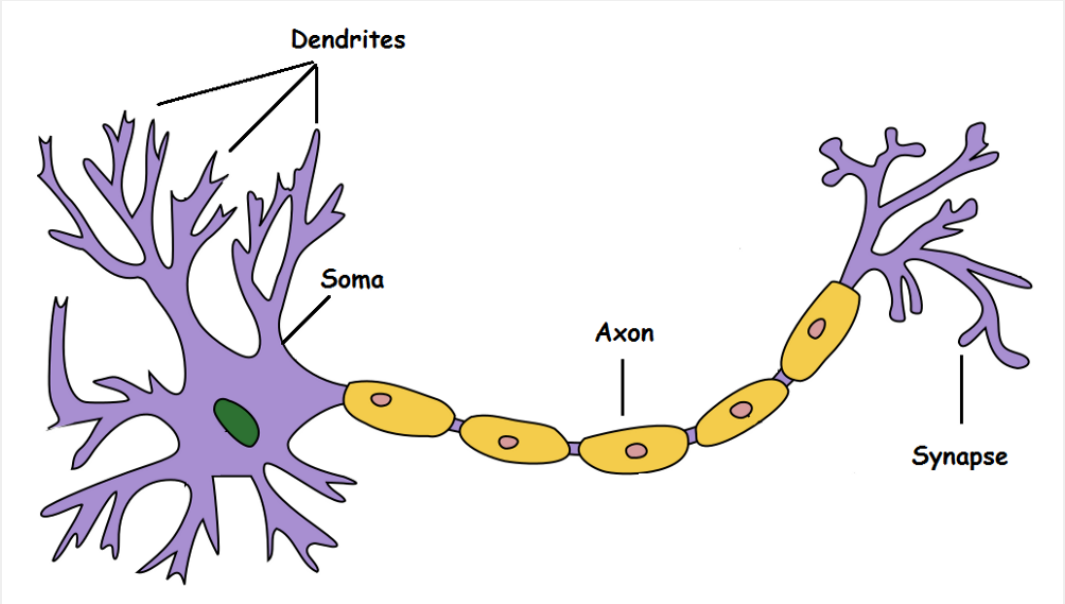
$$I = C_m \frac{\mathrm{d}V_m}{\mathrm{d}t} + \bar{g}_K n^4 (V_m - V_K) + \bar{g}_{Na} m^3 h (V_m - V_{Na}) + \bar{g}_l (V_m - V_l)$$

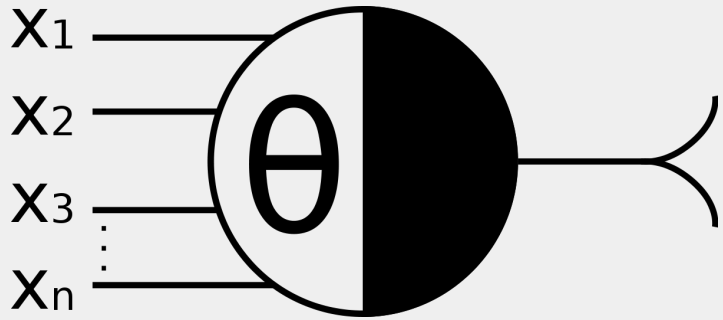
$$\frac{\mathrm{d}n}{\mathrm{d}t} = \alpha_n(V_m)(1-n) - \beta_n(V_m)n\,, \qquad \frac{\mathrm{d}m}{\mathrm{d}t} = \alpha_m(V_m)(1-m) - \beta_m(V_m)m$$

$$\frac{\mathrm{d}h}{\mathrm{d}t} = \alpha_h(V_m)(1-h) - \beta_h(V_m)h$$



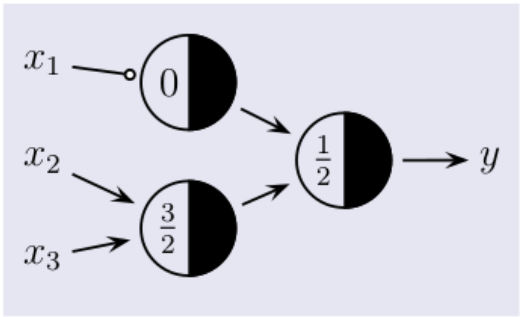
First mathematical models [McCulloch and Pitts, 1943]



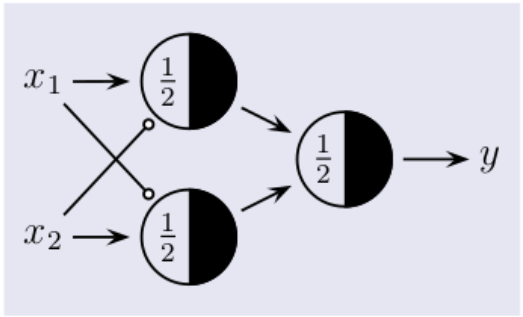


First mathematical models [McCulloch and Pitts, 1943]

- combining several McCullogh/Pitts neurons, we can build more complex boolean functions, e.g.:
  - $y = \neg x_1 \vee (x_2 \wedge x_3)$:



  - $y = XOR(x_1, x_2) = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$:



- Theorem: Networks of McCullogh/Pitts neurons can realize all boolean functions

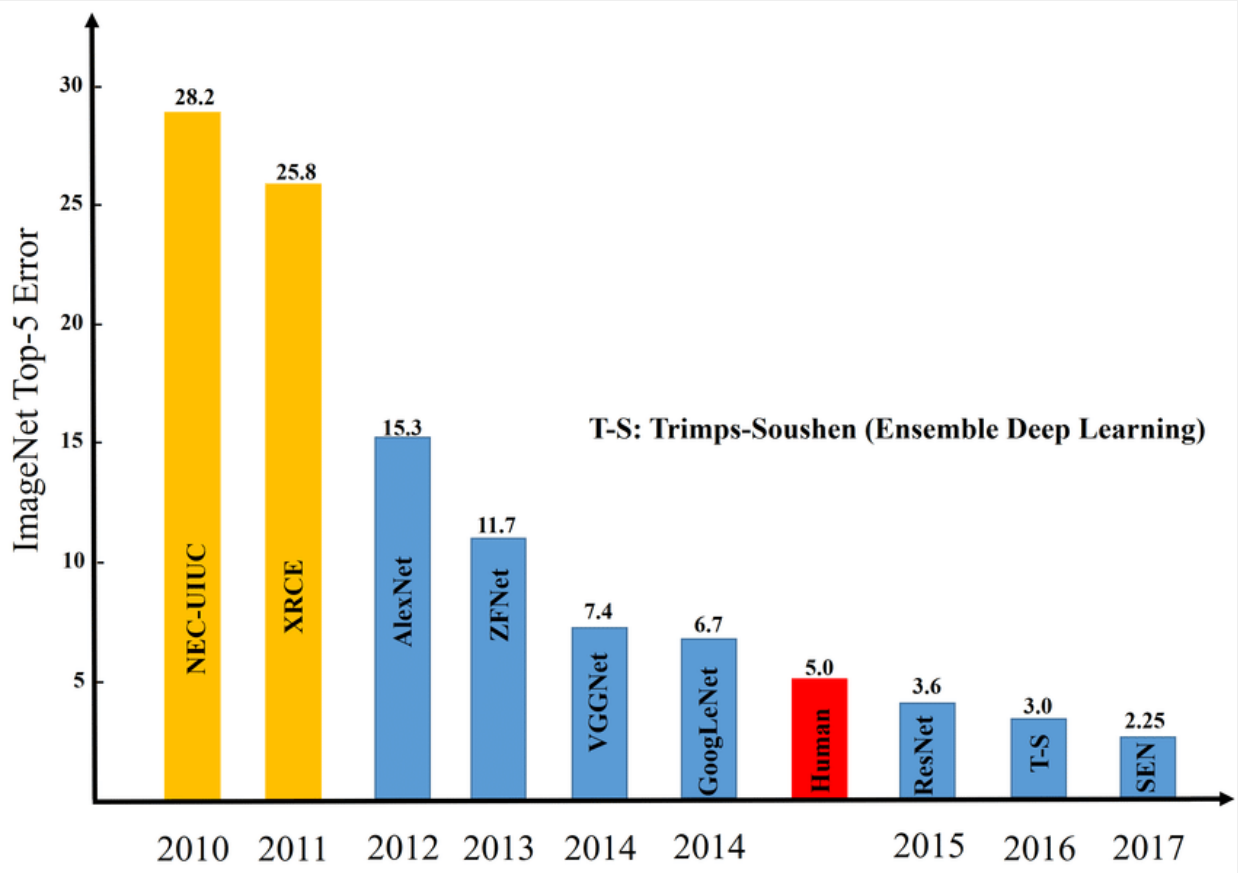
- McCulloch-Pitts networks must be designed, i.e. there exists no training algorithm

Mathematical models of neural networks

Short and incomplete history of neural network models:

- McCulloch-Pitts networks [McCulloch and Pitts, 1943]
- The perceptron [Rosenblatt, 1957]
- Multilayer perceptrons [Werbos, 1974, McClelland et al., 1986]
- Backpropagation algorithm [LeCun et al., 1988]

AlexNet - The breakthrough



- AlexNet [Krizhevsky et al., 2012] is the first neural network that performs better than traditional feature extraction and classification methods on images

From Logistic Regression to ANNs

Logistic regression

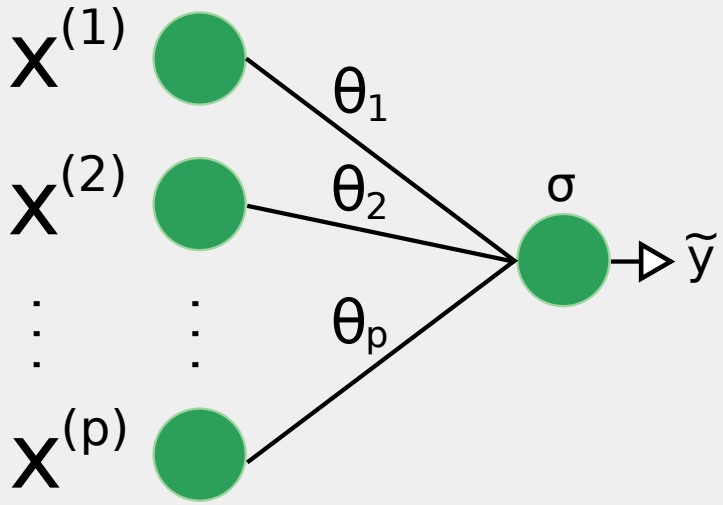
- We convert $x^\top \theta$ to probabilities

$$\mathrm{pr}(Y = 1 \mid \theta) = \sigma(x^\top \theta)$$

- The function σ denotes the sigmoid function

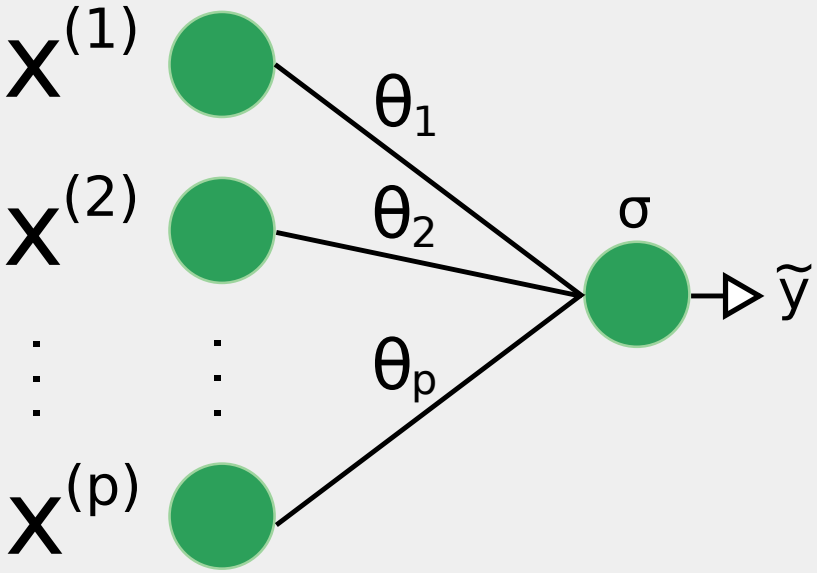
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- Visual respresentation of logistic regression:

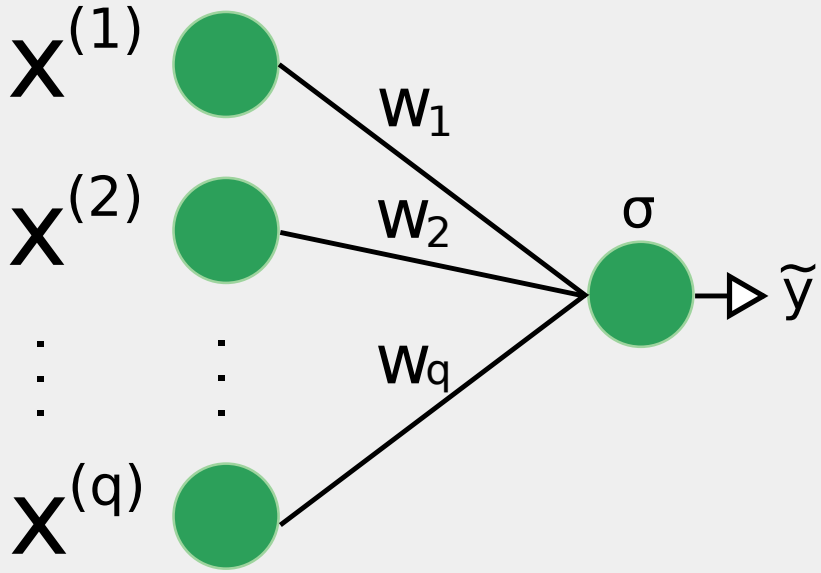


where $x^{(j)}$ is the j-th value of the input vector $x \in \mathbb{R}^p$

Generalizing logistic regression

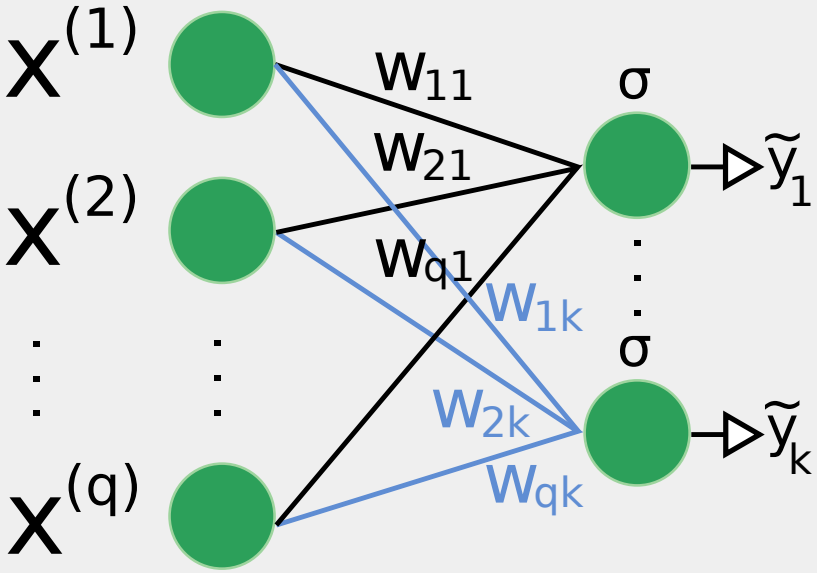


Generalizing logistic regression



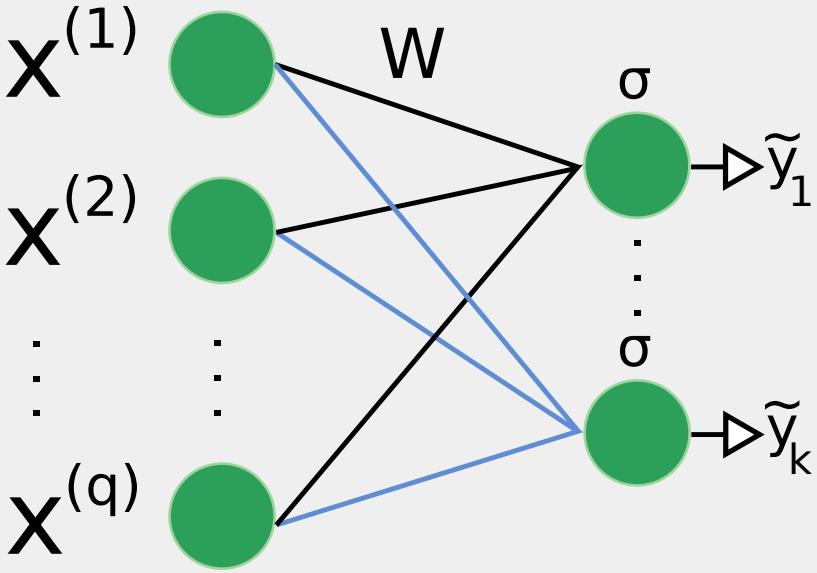
- Instead of θ we use w for weights. In addition, q refers to the number of inputs or features, which for neural networks is much lower than the number of parameters p

Generalizing logistic regression



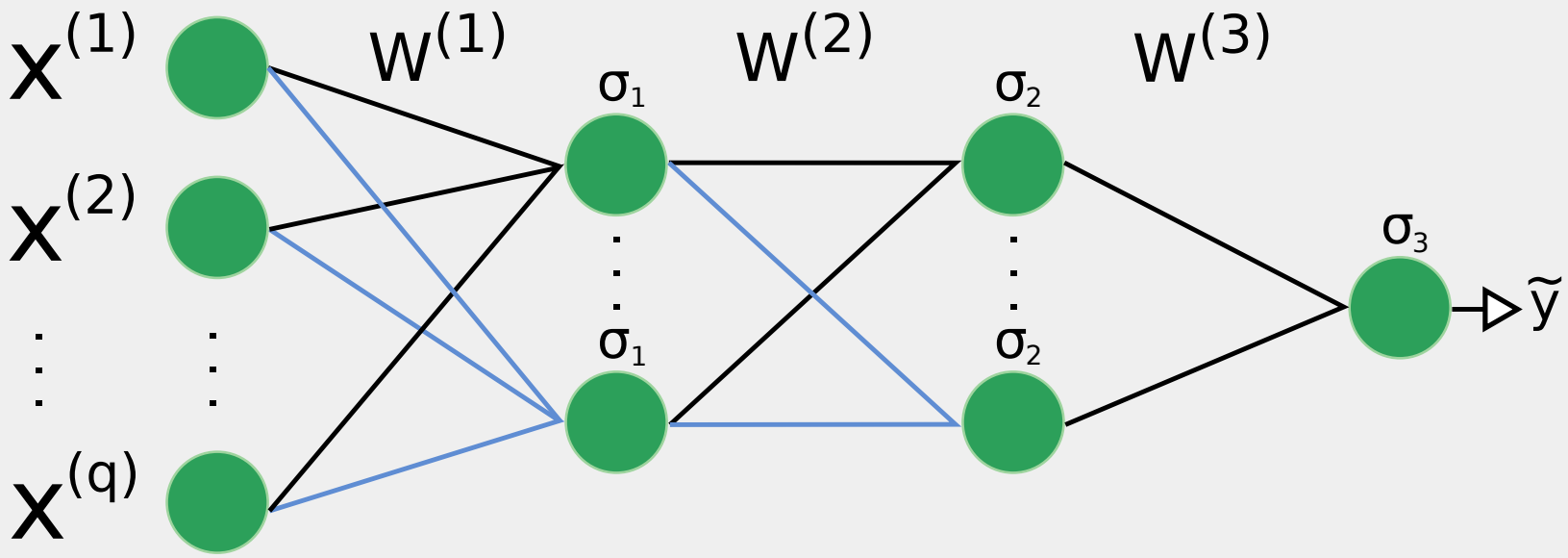
- Instead of θ we use w for weights. In addition, q refers to the number of inputs or features, which for neural networks is much lower than the number of parameters p
- We introduce multiple output neurons $\tilde{y}_1, \ldots, \tilde{y}_k$

Generalizing logistic regression



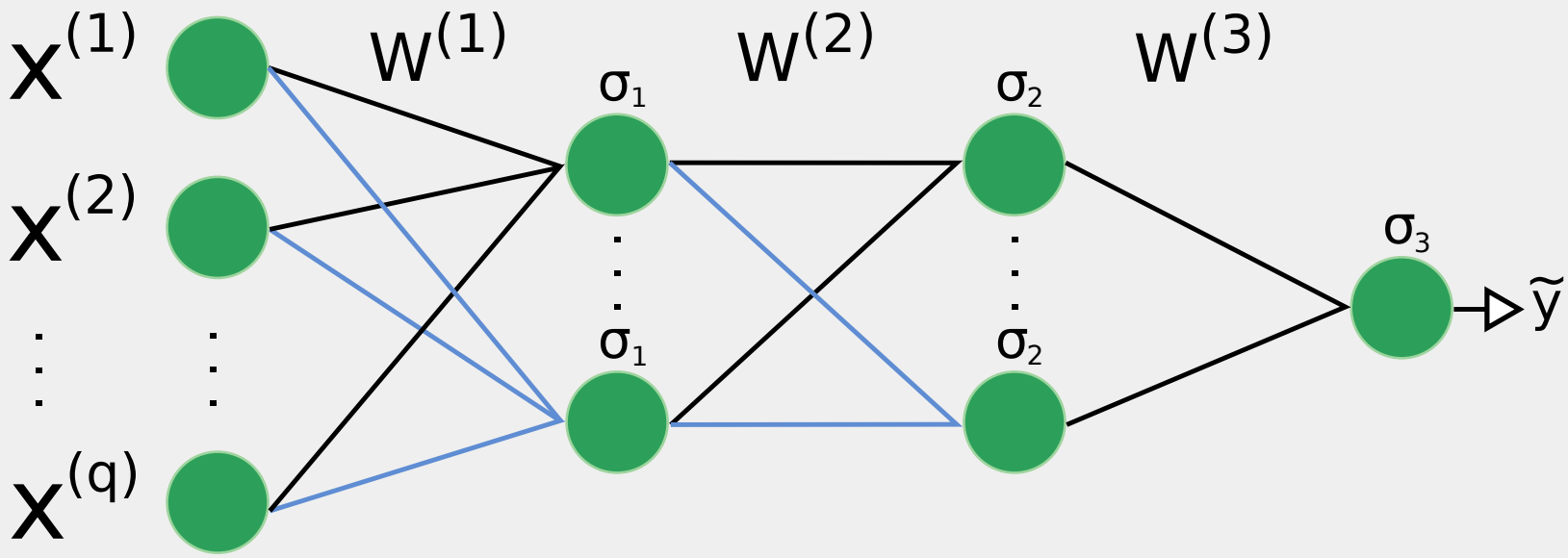
- Instead of θ we use w for weights. In addition, q refers to the number of inputs or features, which for neural networks is much lower than the number of parameters p
- We introduce multiple output neurons $\tilde{y}_1, \ldots, \tilde{y}_k$
- Weights are compiled into a single weight matrix $W \in \mathbb{R}^{p \times k}$

Generalizing logistic regression



- Instead of θ we use w for weights. In addition, q refers to the number of inputs or features, which for neural networks is much lower than the number of parameters p
- We introduce multiple output neurons $\tilde{y}_1, \ldots, \tilde{y}_k$
- Weights are compiled into a single weight matrix $W \in \mathbb{R}^{p \times k}$
- We stack several layers, where each layer ℓ has its own weight matrix $W^{(\ell)}$ and activation function σ_ℓ

Artificial neural network



Feedforward Artificial Neural Network

A feedforward neural network consists of a variable number of layers. Each layer ℓ has its own weight matrix $W^{(\ell)} \in \mathbb{R}^{k_\ell \times k_{\ell+1}}$, where k_ℓ denotes the number of nodes within layer ℓ. The input layer has weight matrix $W^{(1)} \in \mathbb{R}^{k_1 \times k_2}$, where $q = k_1$ is the dimension of the input data. The activation function σ_ℓ is a layer-specific function and applied independently to each node.

Artificial neural network - Notation

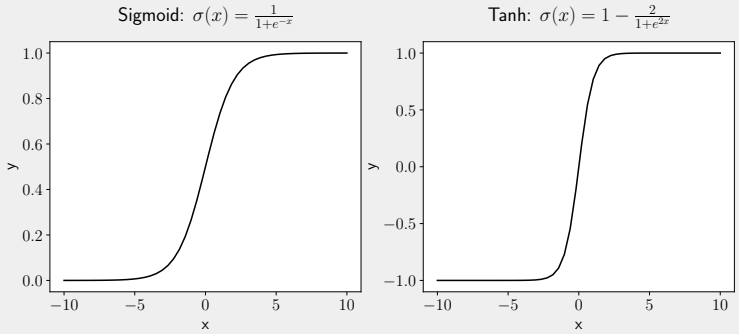
- Let $X = X^{(1)} \in \mathbb{R}^{n \times q}$ denote a data matrix with n samples or observations, each of dimension q
- The output of the ℓ-th layer of a neural network is denoted

$$X^{(\ell+1)} = \sigma_\ell(X^{(\ell)} W^{(\ell)})$$

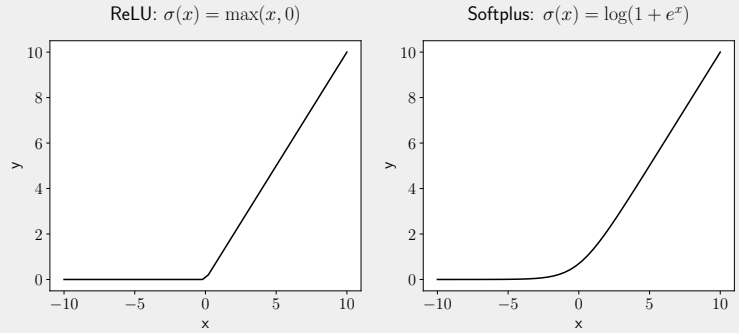
- The activation function σ_ℓ is applied to each cell of the matrix $X^{(\ell)} W^{(\ell)}$
- The prediction $\tilde{y}$ of a neural network with L layers is the output $X^{(L+1)}$ of the last layer
- *Deep neural networks* have a large number of layers L

Activation functions

- Linear functions $\sigma(x) = ax + b$ are not very useful, except for output neurons[1]
- Traditionally there were mainly two activation functions:



- Rectifier linear units (ReLU) [Glorot et al., 2011] and Softplus show better performance for training deep neural networks



[1] Stacking of linear functions results in a linear function

Activation functions

- Especially important is the activation function of the output layer
- Classification:
  - For binary classification problems we use one output node with sigmoid activation
  - The softmax activation $\sigma : \mathbb{R}^k \to (0,1)^k$ is used for multiclass problems

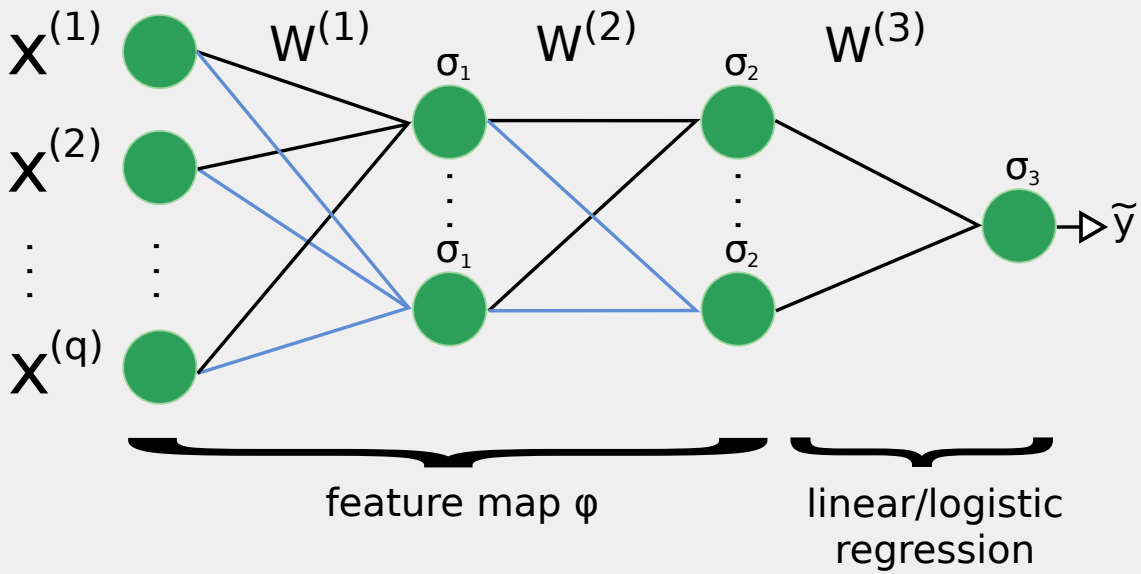
$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$$

  - The output of the softmax is interpreted as class probabilities
- Regression:
  - In most cases simply the identity function $\sigma(x) = x$
  - ReLU can be used for only positive targets

Architectures

- How do we choose the architecture? How many hidden layers? How many neurons in each layer?
- Neural networks with one hidden layer and an abritrary amount of neurons are universal approximators [Hornik et al., 1989]
- Deep neural networks seem to perform better in practice than shallow networks with many neurons
- Each application domain has its own Architectures
  - Image processing: convolutional neural networks (CNNs)
  - Materials: Graph neural networks
  - Machine translation: transformers / attention

ANN INTERPRETATION



Training ANNs

Training ANNs

- Similar to logistic regression, the weights of ANNs are computed using gradient descent
- Gradient descent is an iterative algorithm to minimize a given loss function $\mathcal{L}_W$, i.e.

$$\hat{W} = \arg\min_{w} \mathcal{L}_W(X, y)$$

  where (X, y) denotes the training data and $W = (W^{(1)}, W^{(2)}, \ldots, W^{(L)})$ the weights of the neural network
- Logistic regression uses the negative log-likelihood as loss function
- What loss functions do we use for neural networks?

Training ANNs - Loss functions

- Let f_W denote the neural network with weights W and (X, y) a training data set
- Regression:
  - Mean squared error (similar to OLS):

$$\mathcal{L}_W = \frac{1}{n} \|y - f_W(X)\|_2^2$$

  - Mean absolute error:

$$\mathcal{L}_W = \frac{1}{n} \|y - f_W(X)\|_1$$

    Useful when targets y contain outliers

Training ANNs - Loss functions

- Classification:
  - Binary cross-entropy ($y_i \in \{0, 1\}$):

$$\mathcal{L}_W = -\frac{1}{n} \sum_{i=1}^{n} y_i \log(f_W(x_i)) + (1 - y_i) \log(1 - f_W(x_i))$$

    Equivalent to maximum likelihood of logistic regression

  - Multiclass cross-entropy with k classes ($y_i \in \{0, 1\}^k$):

$$\mathcal{L}_W = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} y_{i,j} \log(f_W(x_i)_j) = -\frac{1}{n} \sum_{i=1}^{n} y_i^\top \log(f_W(x_i))$$

    where $y_{i,j}$ is 1 if the i-th sample belongs to class j and $f_W(x_i)_j$ denotes the output of the j-th node of the last layer of the neural network f_W

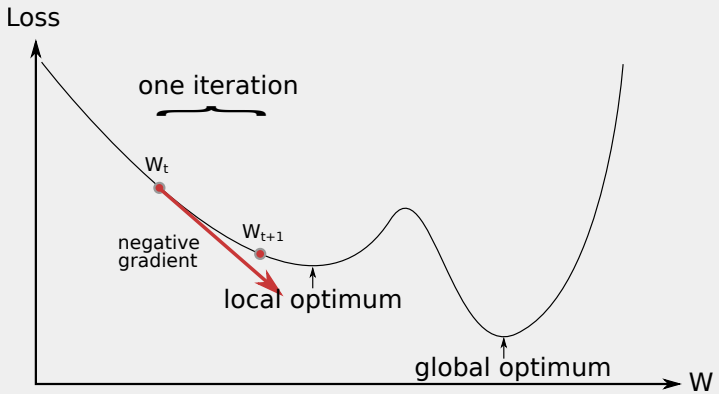
- The loss function is typically not convex!

Gradient descent

Gradient descent updates the weights at each iteration t according to the following update rule:

$$W_{t+1} = W_t - \gamma \nabla_{W_t} \mathcal{L}_{W_t}(X, y)$$

where γ is a parameter that determines the step size



- The larger γ the more likely the algorithm jumps over local optima, but the higher the chance of divergence

Stochastic gradient descent (SGD)

Stochastic gradient descent (SGD) selects at each iteration t a single training sample (x_t, y_t) with $t \in \{1, \ldots, n\}$ at random and updates the weights based on this one sample:

$$W_{t+1} = W_t - \gamma \nabla_{W_t} \mathcal{L}_{W_t}(x_t, y_t)$$

- The stochastic nature of SGD can help to bypass local optima [Masters and Luschi, 2018]
- SGD is slow, because we have to update weights for each sample and cannot utilize parallel computation

Mini-batch SGD

Mini-batch SGD selects at each iteration t a random subset (X_t, y_t) of m samples $X_t = (x_{t_1}, \ldots, x_{t_m})$, $y_t = (y_{t_1}, \ldots, y_{t_m})$ with $t_k \in \{1, \ldots, n\}$ and updates the weights accordingly:

$$W_{t+1} = W_t - \gamma \nabla_{W_t} \mathcal{L}_{W_t}(X_t, y_t)$$

(X_t, y_t) is called a mini-batch and m controls the mini-batch size

- Practice has shown that (mini-batch) SGD seems to improve generalization [Hoffer et al., 2017]
- We typically select $m = 32$ or $m = 64$

Gradient descent with momentum

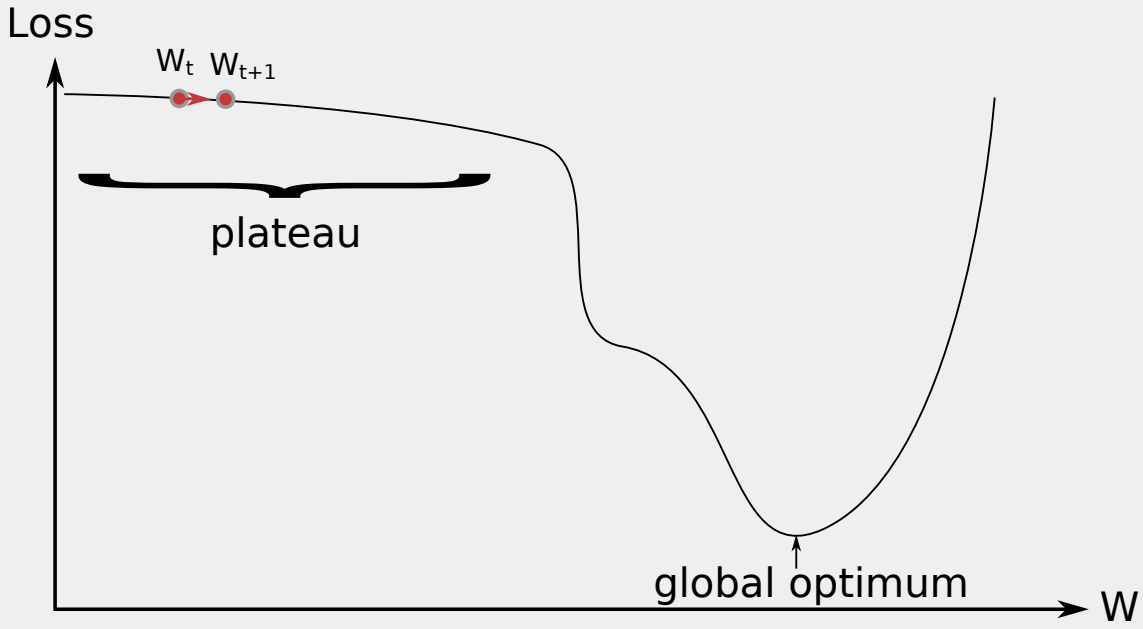
The update rule of gradient descent with momentum is

$$M_{t+1} = \beta M_t + \nabla_{W_t} \mathcal{L}_{W_t}(X_t, y_t)$$
$$W_{t+1} = W_t - \gamma M_{t+1}$$

where γ is the usual step size and β determines the weight of previous updates

- For $\beta = 0$ we obtain vanilla gradient descent
- Intuitive explanation: The mass of a stone rolling downhill adds momentum
- In some cases gradient descent with momentum can help to jump out of local optima
- It makes SGD or mini-batch SGD more stable

Training ANNs - Gradient descent



- Gradient descent converges slowly when being stuck on plateaus
- The gradient itself is not very informative
- Some gradient descent methods only use the sign of the gradient (i.e. Rprop [Riedmiller and Braun, 1992])
- Rprop does not work for mini-batches

Root Mean Square Propagation (RMSProp)

RMSProp uses the following update rule:

$$V_{t+1} = \beta V_t + (1-\beta)\left(\nabla_{W_t}\mathcal{L}_{W_t}(X_t, y_t)\right)^2$$
$$W_{t+1} = W_t - \frac{\gamma}{\sqrt{V_{t+1}}}\nabla_{W_t}\mathcal{L}_{W_t}(X_t, y_t)$$

where γ denotes the step size and β decay rate for averaging over previous gradients

- For $\beta = 0$ we obtain

$$W_{t+1} = W_t - \frac{\gamma}{|\nabla_{W_t}\mathcal{L}_{W_t}(X_t, y_t)|}\nabla_{W_t}\mathcal{L}_{W_t}(X_t, y_t)$$

  i.e. we only consider the sign of the gradient

Adaptive Moment Estimation (Adam)

Adam uses the following update rule:

$$M_{t+1} = \beta_1 M_t + (1-\beta_1)\nabla_{W_t}\mathcal{L}_{W_t}(X_t, y_t) \quad | \text{ momentum}$$

$$V_{t+1} = \beta_2 V_t + (1-\beta_2)\nabla_{W_t}\mathcal{L}_{W_t}(X_t, y_t)^2 \quad | \text{ adaptive } \gamma$$

$$\hat{M}_{t+1} = M_{t+1}/(1-\beta_1^{t+1}), \quad \hat{V}_{t+1} = V_{t+1}/(1-\beta_2^{t+1}) \quad | \text{ bias correction}$$

$$W_{t+1} = W_t - \frac{\gamma}{\sqrt{\hat{V}_{t+1} + \epsilon}}\hat{M}_{t+1}$$

where γ denotes the step size and β decay rate for averaging over previous gradients

- $M_0 = 0$ and $V_0 = 0$, hence early estimates are biased towards zero, $\hat{M}$ and $\hat{V}$ correct this bias

Adaptive Moment Estimation (Adam)

Adam uses the following update rule:

$$M_{t+1} = \beta_1 M_t + (1-\beta_1)\nabla_{W_t}\mathcal{L}_{W_t}(X_t, y_t) \quad | \text{ momentum}$$

$$V_{t+1} = \beta_2 V_t + (1-\beta_2)\nabla_{W_t}\mathcal{L}_{W_t}(X_t, y_t)^2 \quad | \text{ adaptive } \gamma$$

$$\hat{M}_{t+1} = M_{t+1}/(1-\beta_1^{t+1}), \quad \hat{V}_{t+1} = V_{t+1}/(1-\beta_2^{t+1}) \quad | \text{ bias correction}$$

$$W_{t+1} = W_t - \frac{\gamma}{\sqrt{\hat{V}_{t+1} + \epsilon}}\hat{M}_{t+1}$$

where γ denotes the step size and β decay rate for averaging over previous gradients

- Adam is the default for training neural networks
- RMSProp with bias-correction and momentum

Vanishing and exploding gradient problem

- Suppose we want the gradient of the first weight matrix of a neural network with L layers
- The derivative of the first weight matrix is defined by the *chain rule*:

$$\frac{\partial}{\partial W^{(1)}} \mathcal{L}_W(X, y) = \frac{\partial \mathcal{L}_W}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial X^{(L)}} \frac{\partial X^{(L)}}{\partial X^{(L-1)}} \cdots \frac{\partial X^{(2)}}{\partial W^{(1)}}(X, y)$$

- We use the partial derivative $\partial / \partial W^{(1)}$ to denote that all other weight matrices

$$W^{(2)}, W^{(3)}, \ldots, W^{(L)}$$

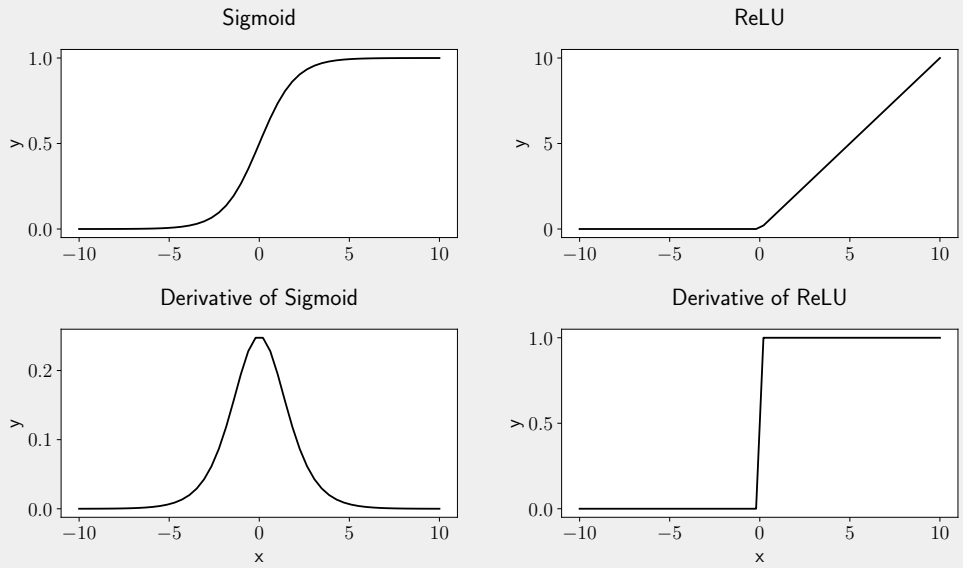
  are treated as constants at their current values
- In deep neural networks, the gradient of the first layers will behave very differently than the gradient of the last layers

Vanishing and exploding gradient problem

- In deep neural networks, the gradient of the first layers will behave very differently than the gradient of the last layers
- For the first layers, the gradient easily vanishes or explodes
- Strategies to counter this problem
  - Normalization of training data
  - Proper weight initialization
  - Activation functions such as ReLU
  - Normalization of layer outputs $X^{(\ell)}$ (batch normalization)
  - Skip connections: $X^{(\ell+1)} = \sigma_\ell(X^{(\ell)}W^{(\ell)}) + X^{(\ell)}$

Vanishing and exploding gradient problem

- Sigmoid and Tanh activations cause vanishing gradients:



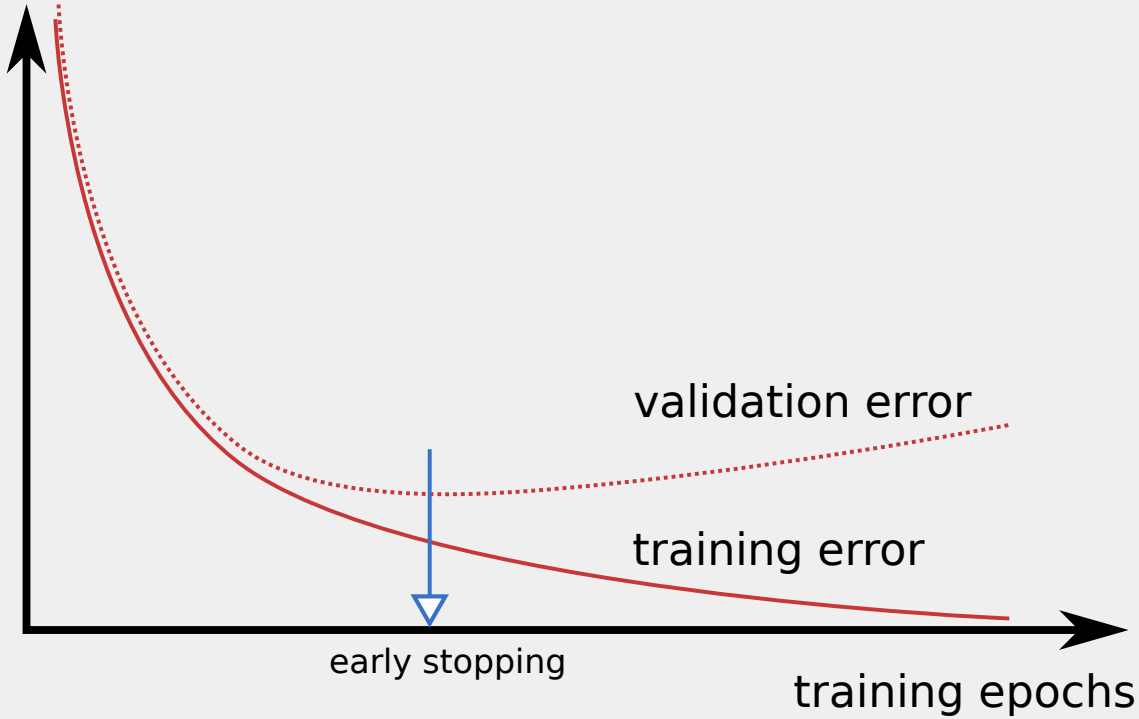
- ReLU is less prone to vanishing gradients
- ReLU often leads to inactive nodes during training (zero gradient)
- LeakyReLU is often used instead

Training ANNs - Remarks

- How do we compute the gradient for training neural networks?
- Backpropagation algorithm for neural networks
- Libraries such as Tensorflow or PyTorch use automatic differentiation (AD)
- With AD it is possible to compute the gradient of arbitrary functions, including neural networks
- Developing deep neural networks requires much testing (watch the gradient!)
  - Weights must stay within a reasonable range
  - Training data must be normalized
  - Local optima must be bypassed using SGD
  - Overfitting must be prevented

Training ANNs - Remarks

- Early stopping prevents overfitting
- We use a small portion of the training data for validation



Training ANNs - Remarks

- Statistics:
  - Define the optimization problem (e.g. maximum likelihood)
  - Use explicit regularization, especially for overparameterized models
  - Check that there is a unique solution
  - Develop numerical methods for finding the solution
- Modern machine learning:
  - Define the optimization problem (e.g. minimize MSE) using an overparameterized model
  - Use a variety of tricks to compute solutions that generalize well
  - The gradient method itself is considered a method for regularization

References I

Glorot, X., Bordes, A., and Bengio, Y. (2011).
Deep sparse rectifier neural networks.
In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings.

Hodgkin, A. L. and Huxley, A. F. (1952).
A quantitative description of membrane current and its application to conduction and excitation in nerve.
The Journal of physiology, 117(4):500.

Hoffer, E., Hubara, I., and Soudry, D. (2017).
Train longer, generalize better: closing the generalization gap in large batch training of neural networks.
Advances in neural information processing systems, 30.

Hornik, K., Stinchcombe, M., and White, H. (1989).
Multilayer feedforward networks are universal approximators.
Neural networks, 2(5):359–366.

References II


Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012).
Imagenet classification with deep convolutional neural networks.
Advances in neural information processing systems, 25.


LeCun, Y., Touresky, D., Hinton, G., and Sejnowski, T. (1988).
A theoretical framework for back-propagation.
In *Proceedings of the 1988 connectionist models summer school*, volume 1, pages 21–28.


Masters, D. and Luschi, C. (2018).
Revisiting small batch training for deep neural networks.
arXiv preprint arXiv:1804.07612.


McClelland, J. L., Rumelhart, D. E., and Hinton, G. E. (1986).
The appeal of parallel distributed processing.
MIT Press, Cambridge MA, pages 3–44.

References III


McCulloch, W. S. and Pitts, W. (1943).
A logical calculus of the ideas immanent in nervous activity.
The bulletin of mathematical biophysics, 5(4):115–133.


Riedmiller, M. and Braun, H. (1992).
Rprop - a fast adaptive learning algorithm.
In *Proc. of ISCIS VII), Universitat*. Citeseer.


Rosenblatt, F. (1957).
The perceptron, a perceiving and recognizing automaton Project Para.
Cornell Aeronautical Laboratory.


Werbos, P. (1974).
Beyond regression:" new tools for prediction and analysis in the behavioral sciences.
Ph. D. dissertation, Harvard University.