*** * * Artificial Intelligence and Machine Learning * * ***

Neural Networks and Deep Learning

Daniel Wehner, M.Sc.

SAP SE / DHBW Mannheim

Summer term 2025

Find all slides on GitHub (DaWe1992/Applied_ML_Fundamentals)

Lecture Overview

Agenda for this Unit

Section:

Introduction to Deep Learning

What is Deep Learning?
History of Deep Learning, biological and psychological Motivation
Perceptron Model Function
Perceptron Criterion / Error Function
Perceptron Training

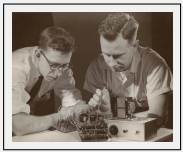
What is Deep Learning?

- **Deep Learning** is an umbrella term for all methods related to artificial neural networks
- It is a **supervised** method and **model based**
- Artificial neural networks are inspired by the human brain
  (but are much simpler)
- Lots of different architectures exist. Most commonly used are:
  - **Multi-Layer perceptrons (MLPs)**
  - **Convolutional neural networks (CNNs)**
  - **Recurrent neural networks (RNNs)**, e. g. LSTMs, GRUs

History of Deep Learning

Early booming (1950s – early 1960s):

FRANK ROSENBLATT suggests the **Perceptron** learning algorithm: Click ⇒ here for the original paper



Setback I (mid 1960s – late 1970s):

MINSKY and PAPERT (1969): The Perceptron algorithm cannot learn the **XOR problem**

History of Deep Learning (Ctd.)

Renewed enthusiasm (1980s):

Better optimization techniques are available, e. g. **backpropagation** for deep neural networks

Setback II (1990s – mid 2000s):

- Other techniques were considered superior, e. g. Support Vector Machines (SVMs)
- Computer science journals even rejected papers on neural networks

History of Deep Learning (Ctd.)

Deep Learning (since mid 2000):

Today we have

- a lot more data,
- faster computers, and
- better optimization techniques

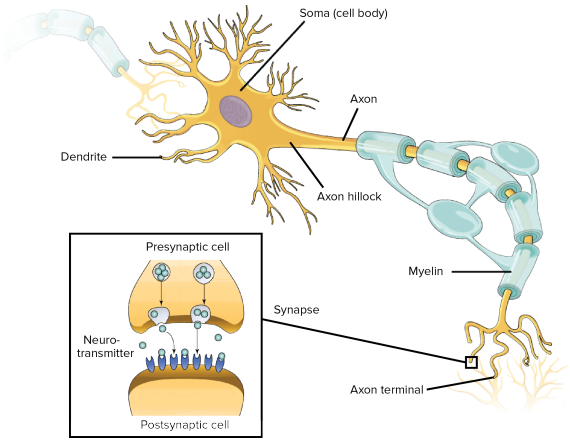
All of these factors make deep neural networks **one of the most popular machine learning techniques** today.

The adjective *'deep'* refers to the use of **multiple (hidden) layers** in the network.

Biological Motivation

- All neurons are connected and form a complex **network**
- **Transmitter chemicals** within the fluid of the brain influence the **electrical potential** inside the body of the neurons
- If the **membrane potential** reaches some threshold, the neuron **fires** and a pulse of fixed length is sent down the **axon**
- The axon connects the neuron with other neurons via **synapses**
- Probably there are 100 trillion **(!!!)** synapses in the human brain
- There is a **refractory period** after a neuron has fired

Sketch of a Brain Cell



How do Humans and Animals learn?

- **Idea:** Mechanism of learning is **association**
- **HEBBian learning:** If the firing of one neuron repeatedly assists in firing another neuron, their **synaptic connection will be strengthened**

*'When an axon of cell A is near enough to excite a cell B and repeatedly or persistently **takes part in firing it**, some growth process or **metabolic change** takes place in one or both cells such that A's **efficiency**, as one of the cells firing B, is **increased**.'*

*'The general idea is an old one, that any two cells or systems of cells that are **repeatedly active at the same time** will tend to become **'associated'**, so that activity in one facilitates activity in the other.'*

HEBB

Classical / PAVLOVian Conditioning

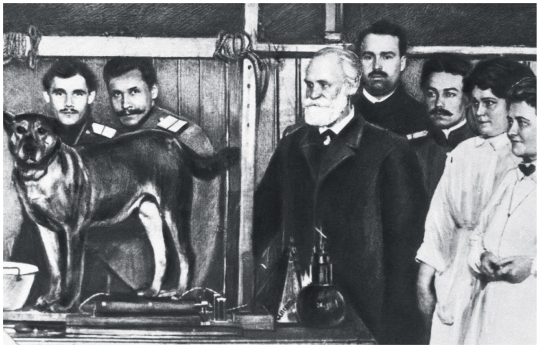
Before conditioning:

- Dog salivates when given food
- We refer to food as the **unconditioned stimulus (US)**
- Salivation in response to food is the **unconditioned response (UR)**

After conditioning:

- Food is paired with the sound of a bell
- Bell is **conditioned stimulus (CS)**
- Bell will eventually elicit salivation, even without food
- Now, salivation is referred to as the **conditioned response (CR)**

Classical / PAVLOVian Conditioning (Ctd.)



Blocking

- CS is a light (L), a noise (N), or a combination of both (LN)
- US is a mild shock that is paired with the CS in the training phase (+)
- Fear response is tested after training when only L is presented without shock (-)

Group A	train N+	train LN+	test L-	$\Rightarrow$ no conditioning
Group B		train LN+	test L-	$\Rightarrow$ conditioning

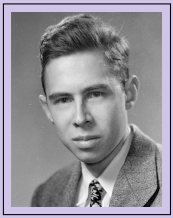
- Group B shows conditioning, group A does not $\Rightarrow$ **N blocks L**
- This is hard to explain with HEBBian learning
- **Idea:** Learning only happens, if there is a **prediction error**

Portrait: FRANK ROSENBLATT

FRANK ROSENBLATT's (1928 – 1971) perceptron played an important role in the history of machine learning. Initially, ROSENBLATT simulated the perceptron on an IBM 704 computer at Cornell in 1957, but by the early 1960s he had built special-purpose hardware that provided a direct, parallel implementation of perceptron learning.

Many of his ideas were encapsulated in $\Rightarrow$ Principles of Neurodynamics published in 1962. ROSENBLATT died in July 1971 on his 43rd birthday in a boating accident in Chesapeake Bay. Although MINSKY was an opponent of ROSENBLATT's perceptron, MINSKY continued ROSENBLATT's research on neural networks after his death.

(Wikipedia)



Perceptron Model Function

- The **Perceptron** was originally proposed by FRANK ROSENBLATT in 1957
- The Perceptron is a **binary classification model**
- The (linear) model function is given by ($\boldsymbol{w} \in \mathbb{R}^M$ and $b \in \mathbb{R}$):

$$h_{\boldsymbol{w},b}(\boldsymbol{x}) := \text{sign}\left(\boldsymbol{w}^\top \boldsymbol{x} + b\right) \tag{1}$$

- The non-linear activation function g is given by a **step function** of the form

$$\text{sign}(z) := \begin{cases} +1 & \text{if } z \geqslant 0 \\ -1 & \text{if } z < 0 \end{cases} \tag{2}$$

Perceptron Criterion

- The labels y_n are given by either +1 *(positive class)* or -1 *(negative class)*
- We seek a weight vector $\boldsymbol{w} \in \mathbb{R}^M$ and an offset $b \in \mathbb{R}$ such that

$$y_n\left(\boldsymbol{w}^\top \boldsymbol{x}^n + b\right) > 0 \qquad \forall n = 1, 2, \ldots, N$$

Perceptron criterion: *(error function)*

$$\mathfrak{J}_P(\boldsymbol{w}, b) = -\sum_{n \in \mathcal{M}} y_n\left(\boldsymbol{w}^\top \boldsymbol{x}^n + b\right) \tag{3}$$

Remark: $\mathcal{M}$ is the set of indices of misclassified examples

Perceptron Criterion (Ctd.)

- The elements of the sum in the objective function (3) depend on the set of misclassified feature vectors $\mathcal{M}$
- **Please note:** In each iteration step the cardinality of $\mathcal{M}$ might change
- The gradient of the error function is given by:

$$\frac{\partial}{\partial \boldsymbol{w}} \mathfrak{J}_P(\boldsymbol{w}, b) = -\sum_{n \in \mathcal{M}} y_n \boldsymbol{x}^n \tag{4}$$

$$\frac{\partial}{\partial b} \mathfrak{J}_P(\boldsymbol{w}, b) = -\sum_{n \in \mathcal{M}} y_n \tag{5}$$

Perceptron Learning

- The algorithm applies a **stochastic gradient descent scheme**, i. e. it goes through all training examples sequentially
- If the current example is classified correctly, **do nothing!**
- If the classification is incorrect, **update the parameters** according to:

$$\begin{pmatrix} \boldsymbol{w}^{t+1} \\ b_{t+1} \end{pmatrix} := \begin{pmatrix} \boldsymbol{w}^{t} \\ b_{t} \end{pmatrix} + \alpha \begin{pmatrix} y_n \boldsymbol{x}^n \\ y_n \end{pmatrix} \tag{6}$$

- The hyperparameter α is the **learning rate**

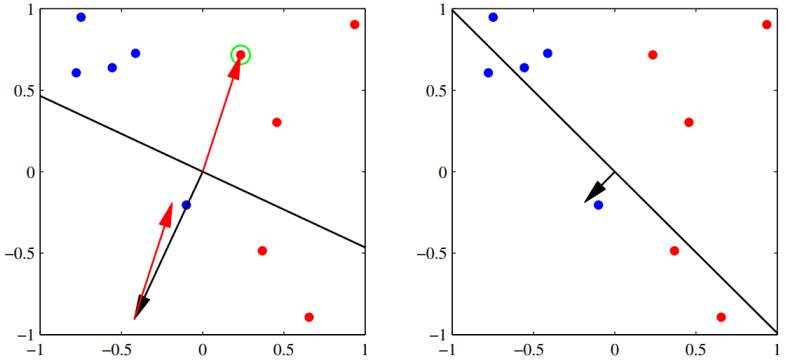
Perceptron Learning (Ctd.)

- Without loss of generality we can set $\alpha := 1$, as **scaling the parameters does not change the decision boundary**
- Effectively, the perceptron algorithm does the following:
- Let $\boldsymbol{x}^n$ be a misclassified example, then

$$\boldsymbol{w}^{t+1} := \begin{cases} \boldsymbol{w}^t + \boldsymbol{x}^n \\ \\ \boldsymbol{w}^t - \boldsymbol{x}^n \end{cases} \quad \text{and} \quad b_{t+1} := \begin{cases} b_t + 1 & \text{if } y_n = +1 \\ \\ b_t - 1 & \text{if } y_n = -1 \end{cases} \tag{7}$$

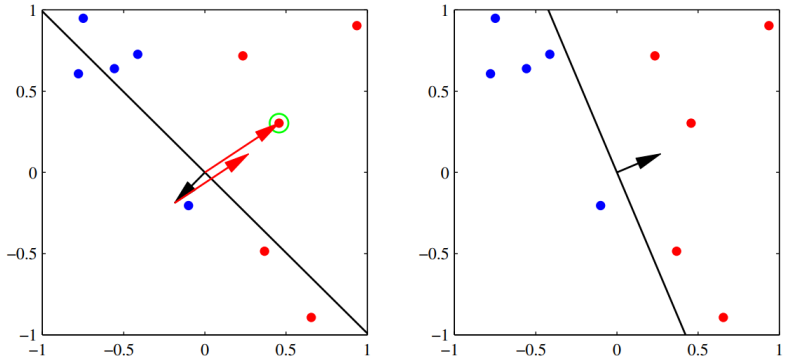
- This means we move the direction of $\boldsymbol{w}$ towards the misclassified data point $\boldsymbol{x}^n$

Perceptron Algorithm Example



cf. [BISHOP.2006], page 195

Perceptron Algorithm Example (Ctd.)



cf. [BISHOP.2006], page 195

Geometric Interpretation

- Assume we classified a positive example as being negative:

$$\begin{aligned}\left\langle \boldsymbol{w}^{t+1}, \boldsymbol{x}^n \right\rangle &= \left\langle \boldsymbol{w}^t + \boldsymbol{x}^n, \boldsymbol{x}^n \right\rangle \\ &= \left\langle \boldsymbol{w}^t, \boldsymbol{x}^n \right\rangle + \left\langle \boldsymbol{x}^n, \boldsymbol{x}^n \right\rangle \\ &= \left\langle \boldsymbol{w}^t, \boldsymbol{x}^n \right\rangle + \left\| \boldsymbol{x}^n \right\|^2 \\ &\geqslant \left\langle \boldsymbol{w}^t, \boldsymbol{x}^n \right\rangle\end{aligned}$$

- **Observation:** $\boldsymbol{w}^{t+1}$ is closer to $\boldsymbol{x}^n$ than $\boldsymbol{w}^t$
- $\langle \cdot, \cdot \rangle$ denotes the scalar product

Mark 1 Perceptron Hardware

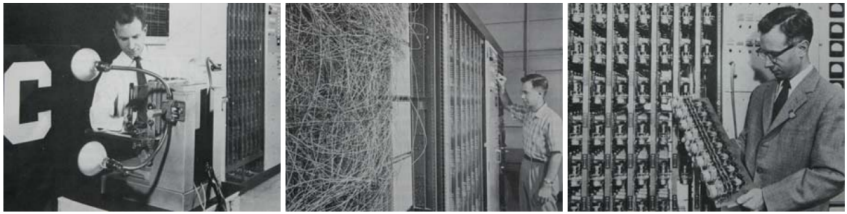


Figure 4.8 Illustration of the Mark 1 perceptron hardware. The photograph on the left shows how the inputs were obtained using a simple camera system in which an input scene, in this case a printed character, was illuminated by powerful lights, and an image focussed onto a 20×20 array of cadmium sulphide photocells, giving a primitive 400 pixel image. The perceptron also had a patch board, shown in the middle photograph, which allowed different configurations of input features to be tried. Often these were wired up at random to demonstrate the ability of the perceptron to learn without the need for precise wiring, in contrast to a modern digital computer. The photograph on the right shows one of the racks of adaptive weights. Each weight was implemented using a rotary variable resistor, also called a potentiometer, driven by an electric motor thereby allowing the value of the weight to be adjusted automatically by the learning algorithm.

cf. [BISHOP.2006], page 196

Perceptron Convergence Theorem

Perceptron Convergence Theorem:

If the training data is **linearly separable**, then the perceptron learning algorithm will **converge after a finite amount of time** and classifies **all training data examples correctly**.

- **The number of steps can be substantial** *(until convergence we do not know if the problem is nonseparable or simply slow to converge)*
- **The solution is not unique** and depends on the initialization of $\boldsymbol{w}$ and b and on the order of presentation of the data points!

Section:

Neural Network Architectures

Introduction
Network with one Neuron
Network with multiple Sigmoid Output Neurons
Network with multiple Softmax Output Neurons
Network with hidden Layers, Multi-Layer Perceptrons

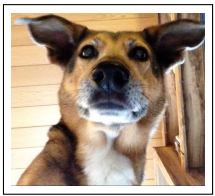
Roadmap

- **Use case:** We want to detect different animals in images *(one animal per image)*
- Starting from **logistic regression**, we gradually increase the complexity of the model to finally obtain a deep neural network
- **Steps:**
  1. We start with a **single neuron** and sigmoid activation *(binary classification)*
  2. We **add more output neurons** with sigmoid activation *(multi-class classification)*
  3. We change the activation function to **softmax** to allow for a probabilistic interpretation
  4. We **add hidden layers** to obtain highly non-linear classifiers
  5. (We **add convolution** to obtain better features from images)
- The basic building block of neural networks is a single neuron

Detection of Animals from Images: Three Classes

Let us consider the three classes:

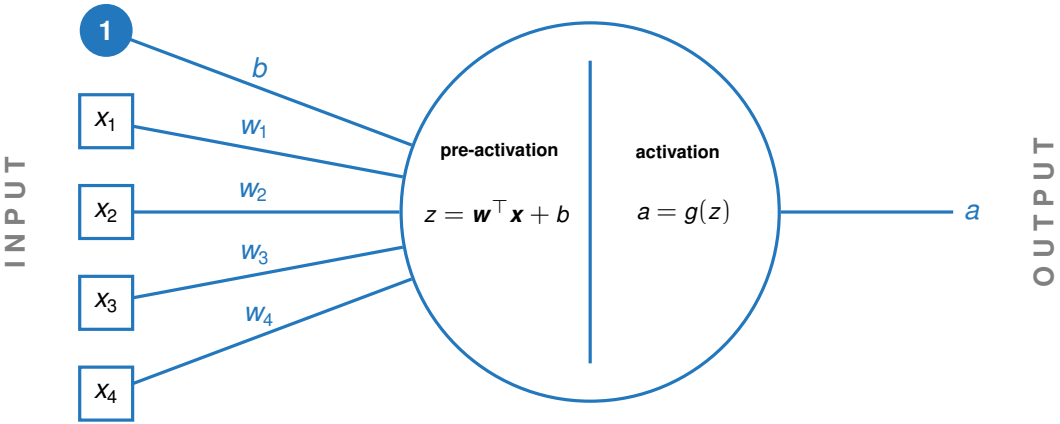
Dog



Cat

Rabbit

Architecture of a single Neuron



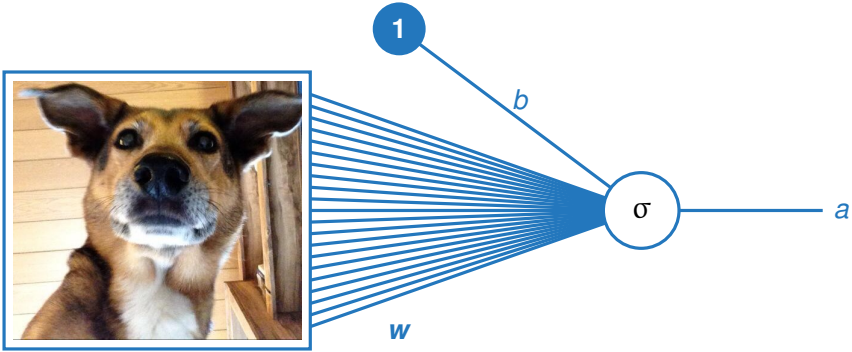
NW1: Model Architecture and Model Function

- Let us start with a simple network: **Single neuron with sigmoid activation (NW1)**
- **Task:** Predict whether a dog is in the image or not
- Remember the model function for logistic regression:

$$h_{\boldsymbol{w},b}(\boldsymbol{x}) := a := \sigma\left(\boldsymbol{w}^\top \boldsymbol{x} + b\right) \tag{8}$$

- **Note:** $\boldsymbol{w} \in \mathbb{R}^M$ and $b \in \mathbb{R}$ are the parameters of the network
- **Observation:** The neuron performs logistic regression when choosing the sigmoid function $\sigma(z) := \frac{1}{1+e^{-z}}$ as the activation function g **(logistic unit)**

NW1: Visualization of the Model Architecture



NW1: Loss Function

- The model minimizes the **binary cross entropy loss** which is given by

$$\mathfrak{J}(\boldsymbol{w}, b) := \frac{1}{N} \sum_{n=1}^{N} \ell^{\text{BCE}}\left(a_n, y_n\right) \tag{9}$$

with:

$$\ell^{\text{BCE}}\left(\widehat{y}, y\right) := -y \log(\widehat{y}) - (1 - y) \log(1 - \widehat{y}) \tag{10}$$

- We derived this cost function using **maximum likelihood estimation**

NW1: Some Remarks

- The quantity $z := \boldsymbol{w}^\top \boldsymbol{x} + b$ is referred to as the **pre-activation** of the neuron
  (it is comparable to the electrical potential in a brain cell)
- The quantity $a := g(z)$ is called the **activation** of the neuron
  (this is the output of the model comparable to the impulse sent down the axon)
- When using the sigmoid function as the activation function, i. e. $g(z) = \sigma(z)$, the output a lies in the interval $[0, 1]$ and allows for a probabilistic interpretation

Remark: A single neuron is a **linear classifier** and can only be used in a **binary classification** setting

NW2: Model Architecture

- Let us now turn towards the **multi-class classification problem**
- **New task:** Predict if there is a dog, a cat, or a rabbit in the image
  (assume there is exactly one of these animals present in the image)
- **Multiple output neurons with sigmoid activation (NW2):**
  - We add one output neuron for each of the K classes (here: $K = 3$)
  - For each class we maintain a separate set of parameters $\boldsymbol{w}^k$ and a bias b_k
  - Again, we decide to use the sigmoid activation function σ

Basically we train three logistic regression models in parallel!

NW2: Model Architecture (Ctd.)

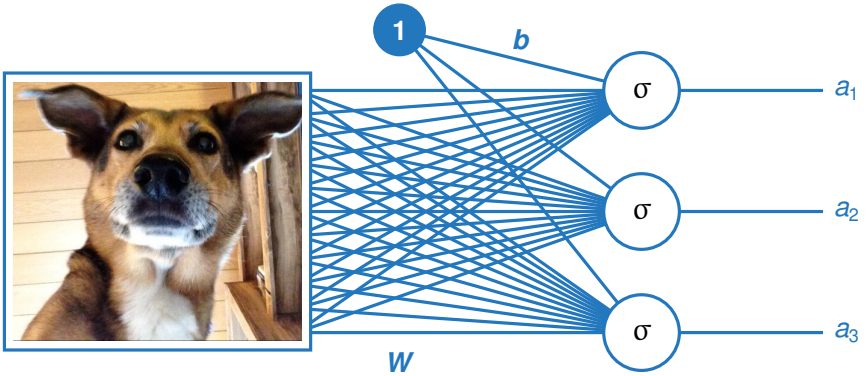
- We stack the parameters $\boldsymbol{w}^k$ into a matrix $\boldsymbol{W} \in \mathbb{R}^{3 \times M}$:

$$\boldsymbol{W} := \begin{pmatrix} — & \boldsymbol{w}^1 & — \\ — & \boldsymbol{w}^2 & — \\ — & \boldsymbol{w}^3 & — \end{pmatrix} \in \mathbb{R}^{3 \times M} \tag{11}$$

- Similarly, we construct a column vector $\boldsymbol{b} \in \mathbb{R}^3$:

$$\boldsymbol{b} := \begin{pmatrix} b_1 & b_2 & b_3 \end{pmatrix}^\top \in \mathbb{R}^3 \tag{12}$$

NW2: Visualization of the Model Architecture



NW2: Model Function

- The model function takes the form:

$$h_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{x}) := \boldsymbol{a} := \boldsymbol{\sigma}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{13}$$

- Please note that $\boldsymbol{\sigma}$ is a vector-valued function, i. e.:

$$\boldsymbol{\sigma} : \mathbb{R}^3 \to \mathbb{R}^3, \qquad \boldsymbol{z} \mapsto \boldsymbol{\sigma}(\boldsymbol{z}) := \begin{pmatrix} \sigma(z_1) \\ \sigma(z_2) \\ \sigma(z_3) \end{pmatrix} \tag{14}$$

- In other words: It applies the sigmoid function to each component of the vector $\boldsymbol{z}$

NW2: Loss Function

- We have to sum the loss over all training examples and output neurons
- Also, we have to apply **one-hot encoding** for the labels *(multi-class classification)*
- The loss function becomes:

$$\mathfrak{J}(\boldsymbol{W}, \boldsymbol{b}) := \frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{3} \ell^{\text{BCE}}\left(a_k^{(n)}, y_k^{(n)}\right) \tag{15}$$

- Again, we define $\ell^{\text{BCE}}(\widehat{y}, y) := -y \log(\widehat{y}) - (1-y) \log(1-\widehat{y})$

Question: Do you see any issues with this model?

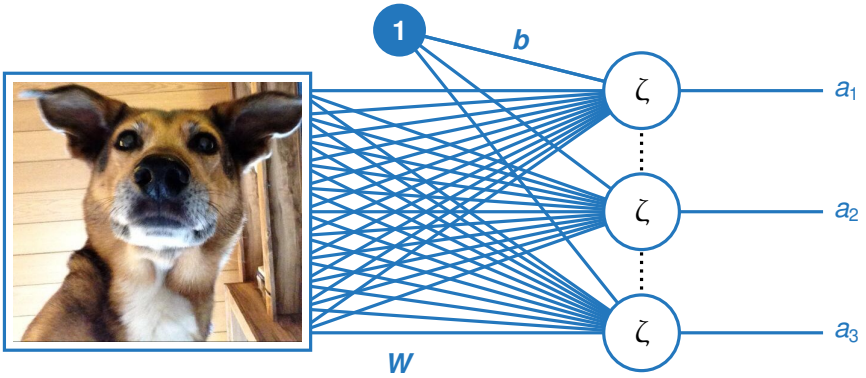
NW3: Switch to Softmax Activations

Problem with the previous approach: Several neurons can output values close to 1 *(i.e. the sum of the output vector components can be larger or less than 1)*, but we assumed that only one animal is shown in the image!

We replace the sigmoid activations with **softmax activations** to enforce that the sum of the output is equal to 1 **(NW3)**:

$$\zeta : \mathbb{R}^3 \to \mathbb{R}^3, \qquad \boldsymbol{z} \mapsto \zeta(\boldsymbol{z}), \qquad \zeta_k(\boldsymbol{z}) := \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \tag{16}$$

NW3: Visualization of the Model Architecture



NW3: Loss Function

- We have to adjust the loss function
- In softmax regression we had already introduced the general **cross entropy loss** given by

$$\mathfrak{J}(\boldsymbol{W}, \boldsymbol{b}) := \frac{1}{N} \sum_{n=1}^{N} \ell^{\mathrm{CE}}\left(\boldsymbol{a}^n, \boldsymbol{y}^n\right) \tag{17}$$

- Again, we define

$$\ell^{\mathrm{CE}}\left(\widehat{\boldsymbol{y}}, \boldsymbol{y}\right) := -\sum_{k=1}^{K} y_k \log(\widehat{y}_k)$$

- We have set $\widehat{y}_k := \zeta_k(\boldsymbol{z})$

NW4: Model Architecture

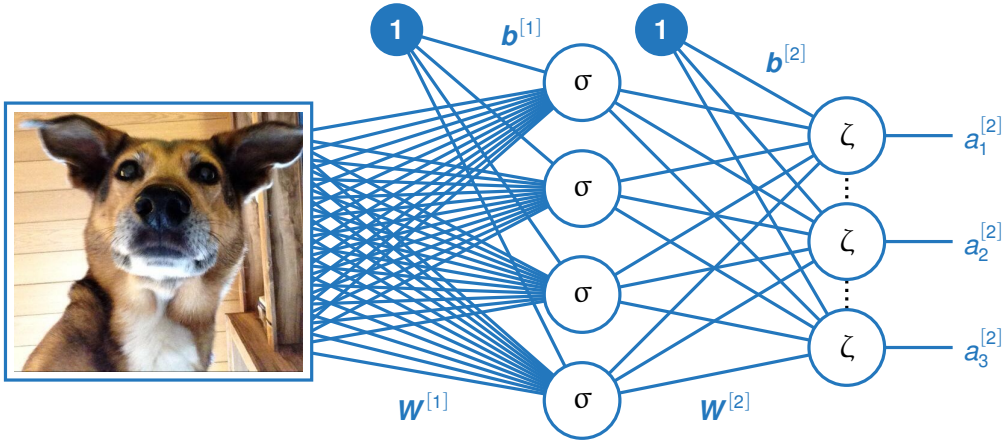
- The previous models (NW1, NW2, and NW3) are **still linear classifiers**
- We add a hidden layer comprising four neurons to NW3 to obtain NW4
- Hidden layers turn neural networks into **highly non-linear models**
- Each layer will be parameterized by separate network parameters:

$$\textbf{Hidden layer:} \quad \boldsymbol{W}^{[1]} \in \mathbb{R}^{4\times M}, \qquad \boldsymbol{b}^{[1]} \in \mathbb{R}^{4}$$

$$\textbf{Output layer:} \quad \boldsymbol{W}^{[2]} \in \mathbb{R}^{3\times 4}, \qquad \boldsymbol{b}^{[2]} \in \mathbb{R}^{3}$$

- Such networks are generally known as **Multi-Layer Perceptrons (MLPs)**

NW4: Visualization of the Model Architecture



Section:

Training of Multi-Layer Perceptrons

Brief Introduction to Matrix Calculus
MLP Forward Pass
MLP Backward Pass and the Backpropagation Algorithm
MLP Batch Training
Classification of exemplary Datasets

Matrix Calculus

- We will derive the gradients for neural networks in **matrix/vector form** which is more efficient from a computational point of view
- The basic building block of vectorized gradients is the **Jacobian matrix**
- Suppose we have a function $\boldsymbol{f} : \mathbb{R}^L \to \mathbb{R}^D$, i. e.

$$\boldsymbol{f}(\boldsymbol{x}) := \begin{pmatrix} f_1(x_1, \ldots, x_L) \\ f_2(x_1, \ldots, x_L) \\ \vdots \\ f_D(x_1, \ldots, x_L) \end{pmatrix} \tag{18}$$

JACOBIan Matrix

- The JACOBIan matrix *(matrix of first-order derivatives)* of $\boldsymbol{f}$ is given by

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} := \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_L} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_D}{\partial x_1} & \cdots & \frac{\partial f_D}{\partial x_L} \end{pmatrix} \in \mathbb{R}^{D \times L} \tag{19}$$

- This means that $\left(\partial \boldsymbol{f} / \partial \boldsymbol{x}\right)_{ij} = \partial f_i / \partial x_j$ *(a standard non-vector derivative)*
- We can apply the **chain rule** to a vector-valued function just by multiplying JACOBIans

Example: Chain Rule with JACOBIans

- Consider the functions $\boldsymbol{f}(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \end{pmatrix}$ and $\boldsymbol{g}(\boldsymbol{x}) = \begin{pmatrix} g_1(x_1, x_2) \\ g_2(x_1, x_2) \end{pmatrix}$
- Suppose we want to compute $\frac{\partial}{\partial x}\boldsymbol{g}\big(\boldsymbol{f}(x)\big) = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{f}} \cdot \frac{\partial \boldsymbol{f}}{\partial x}$
- We receive using the chain rule:

$$\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{f}} \cdot \frac{\partial \boldsymbol{f}}{\partial x} = \underbrace{\begin{pmatrix} \frac{\partial g_1}{\partial f_1} & \frac{\partial g_1}{\partial f_2} \\ \frac{\partial g_2}{\partial f_1} & \frac{\partial g_2}{\partial f_2} \end{pmatrix}}_{2\times 2} \cdot \underbrace{\begin{pmatrix} \frac{\partial f_1}{\partial x} \\ \frac{\partial f_2}{\partial x} \end{pmatrix}}_{2\times 1} = \underbrace{\begin{pmatrix} \frac{\partial g_1}{\partial f_1} \cdot \frac{\partial f_1}{\partial x} + \frac{\partial g_1}{\partial f_2} \cdot \frac{\partial f_2}{\partial x} \\ \frac{\partial g_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial x} + \frac{\partial g_2}{\partial f_2} \cdot \frac{\partial f_2}{\partial x} \end{pmatrix}}_{2\times 1} \tag{20}$$

Matrix Calculus – Identity 1

Derivative of a matrix times a column vector with respect to the vector:

Let $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}$, where $\boldsymbol{W} \in \mathbb{R}^{D \times L}$ and $\boldsymbol{x} \in \mathbb{R}^{L}$. Then the derivative of $\boldsymbol{z}$ with respect to the vector $\boldsymbol{x}$ is given by:

$$\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{z} = \frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{W}\boldsymbol{x} = \boldsymbol{W} \tag{21}$$

Similarly, we have the identity: $\frac{\partial \boldsymbol{x}\boldsymbol{W}}{\partial \boldsymbol{x}} = \boldsymbol{W}^{\top}$, where $\boldsymbol{x} \in \mathbb{R}^{1 \times D}$ and $\boldsymbol{W} \in \mathbb{R}^{D \times L}$

Matrix Calculus – Proof of Identity 1

Proof: We can think of $\boldsymbol{z}$ as a function of $\boldsymbol{x}$ which transforms an L-dimensional input into a D-dimensional output. Therefore, the JACOBIan matrix of $\boldsymbol{z}$ has the shape $D \times L$ by definition. The i-th entry of the output vector $\boldsymbol{z}$ is computed as $z_i = \sum_{k=1}^{L} w_{ik} x_k$ and the entry $\left(\partial \boldsymbol{z} / \partial \boldsymbol{x}\right)_{ij}$ of the JACOBIan matrix is given by:

$$\left(\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}\right)_{ij} = \frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} \sum_{k=1}^{L} w_{ik} x_k = \sum_{k=1}^{L} w_{ik} \frac{\partial}{\partial x_j} x_k = w_{ij} \tag{22}$$

Thus, $\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{z} = \boldsymbol{W}$, as claimed. ■

Matrix Calculus – Identity 2

Derivative of a vector with respect to itself:

Let $\boldsymbol{z} = \boldsymbol{x}$. Then the derivative of $\boldsymbol{z}$ with respect to $\boldsymbol{x}$ is the identity matrix $\boldsymbol{I}$, i. e. we have:

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \boldsymbol{I} \tag{23}$$

Matrix Calculus – Proof of Identity 2

Proof: We have $z_i = x_i$ for all i. Therefore, we obtain

$$\left(\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}\right)_{ij} = \frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} x_i = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} \tag{24}$$

We see that the JACOBIan of $\boldsymbol{z}$ is a diagonal matrix whose entries $(i, i) = 1$ for all i. This is the definition of the identity matrix $\boldsymbol{I}$. ■

Matrix Calculus – Identity 3

Derivative of an elementwise function applied to a vector:

Let $\boldsymbol{x}, \boldsymbol{z} \in \mathbb{R}^D$, where $\boldsymbol{z} = \boldsymbol{f}(\boldsymbol{x}) = \big(f(x_1), \ldots, f(x_D)\big)^\top$. Then we have

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \operatorname{diag}\big(f'(x_1), f'(x_2), \ldots, f'(x_D)\big). \tag{25}$$

Remark: Multiplication by a diagonal matrix corresponds to elementwise multiplication by the entries on the diagonal. Elementwise multiplication is usually denoted by $\odot$ **(HADAMARD product)**, e. g. for $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^D$ we have $\boldsymbol{x} \odot \boldsymbol{y} := \big(x_1 y_1, x_2 y_2, \ldots, x_D y_D\big)^\top \in \mathbb{R}^D$

Matrix Calculus – Proof of Identity 3

Proof: Since $\boldsymbol{f}$ is a vector-valued function which applies the function f elementwise to $\boldsymbol{x}$, we have $z_i = f(x_i)$. Similarly to above we obtain

$$\left(\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}\right)_{ij} = \frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} f(x_i) = \begin{cases} f'(x_i) & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} \tag{26}$$

We see that in this case the JACOBIan is a diagonal matrix for which the entry at position (i, i) is the derivative of f applied to x_i. Therefore, we obtain the result $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \operatorname{diag}\big(f'(x_1), f'(x_2), \ldots, f'(x_D)\big)$. ■

Matrix Calculus – Identity 4

Derivative of the loss function with respect to a weight matrix:

Let ℓ denote the loss function, let further $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}$ and $\boldsymbol{\delta} = \frac{\partial \ell}{\partial \boldsymbol{z}}$. Then:

$$\frac{\partial \ell}{\partial \boldsymbol{W}} = \frac{\partial \ell}{\partial \boldsymbol{z}} \cdot \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}} = \boldsymbol{\delta} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}} = \boldsymbol{\delta}^\top \boldsymbol{x}^\top \tag{27}$$

Remark: $\boldsymbol{x} \in \mathbb{R}^L$ and $\boldsymbol{z} \in \mathbb{R}^D$ are column vectors, $\boldsymbol{\delta} \in \mathbb{R}^{1 \times D}$ is a row vector, and $\boldsymbol{W} \in \mathbb{R}^{D \times L}$, $\boldsymbol{\delta}^\top \boldsymbol{x}^\top \in \mathbb{R}^{D \times L}$ are matrices

Matrix Calculus – Proof of Identity 4

Proof: We want the gradient $\frac{\partial \ell}{\partial \boldsymbol{W}}$ to have the same shape as $\boldsymbol{W} \in \mathbb{R}^{D \times L}$, so that we can easily subtract it from $\boldsymbol{W}$ in the gradient descent update formula. However, $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$ is a tensor! We will solve this issue by taking the gradient with respect to a single weight w_{ij} instead.

Let us consider $\frac{\partial \boldsymbol{z}}{\partial w_{ij}}$ which is a vector. By definition we have:

$$z_k = \sum_{l=1}^{L} w_{kl} x_l \qquad \text{and therefore:} \qquad \frac{\partial z_k}{\partial w_{ij}} = \sum_{l=1}^{L} x_l \frac{\partial}{\partial w_{ij}} w_{kl} \tag{28}$$

Matrix Calculus – Proof of Identity 4 (Ctd.)

Note that the partial derivative $\frac{\partial}{\partial w_{ij}} w_{kl} = 1$ if $i = k$ and $j = l$. Otherwise $\frac{\partial}{\partial w_{ij}} w_{kl} = 0$. This means that the gradient of $\boldsymbol{z}$ is equal to $\mathbf{0}$ if $k \neq i$.

If, however, $k = i$, then the only non-zero element has index $l = j$ and we get x_j as a result. Thus we find:

$$\frac{\partial \boldsymbol{z}}{\partial w_{ij}} = \begin{cases} \begin{pmatrix} 0 & \dots & 0 & x_j & 0 & \dots & 0 \end{pmatrix}^\top & \text{if } k = i \\ \mathbf{0} & & \text{if } k \neq i \end{cases} \tag{29}$$

Matrix Calculus – Proof of Identity 4 (Ctd.)

Let us now compute $\frac{\partial \ell}{\partial w_{ij}}$. We get

$$\frac{\partial \ell}{\partial w_{ij}} = \frac{\partial \ell}{\partial \boldsymbol{z}} \cdot \frac{\partial \boldsymbol{z}}{\partial w_{ij}} = \boldsymbol{\delta} \frac{\partial \boldsymbol{z}}{\partial w_{ij}} = \sum_{k=1}^{L} \delta_k \frac{\partial z_k}{\partial w_{ij}} = \delta_i x_j \tag{30}$$

as the only non-zero term in the sum is $\delta_i \frac{\partial z_i}{\partial w_{ij}}$. We want $\frac{\partial \ell}{\partial \boldsymbol{W}}$ to be a matrix whose entry (i, j) is equal to $\delta_i x_j$. This matrix is equal to the outer product $\frac{\partial \ell}{\partial \boldsymbol{W}} = \boldsymbol{\delta}^\top \boldsymbol{x}^\top$ and the proof is complete. ■

Numerator Layout versus Denominator Layout

- In definition (19) we had stacked the gradients of the component functions row-wise to form the JACOBI matrix which leads to the **numerator layout**
- An alternative is to stack the gradients column-wise which gives rise to the **denominator layout**
- The shapes of the results differ based on the layout used
- Have a look at $\Rightarrow$ this Wikipedia page for details

In this slide deck we will stick to the numerator layout!

Numerator Layout versus Denominator Layout (Ctd.)

The shape of the result differs based on the layout used:

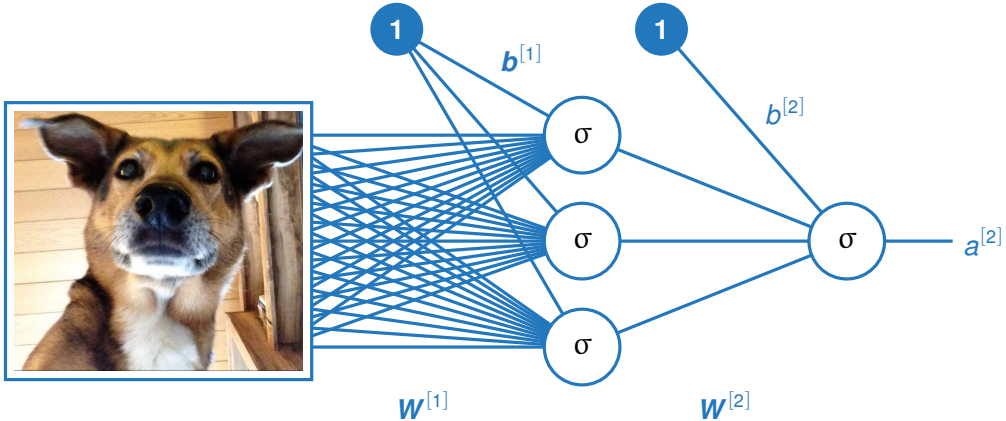
Let $x, y \in \mathbb{R}$, $\boldsymbol{x} \in \mathbb{R}^{D\times 1}$, $\boldsymbol{y} \in \mathbb{R}^{L\times 1}$, $\boldsymbol{X} \in \mathbb{R}^{P\times Q}$

Derivative	Numerator Layout	Denominator Layout
$\frac{\partial y}{\partial x}$	$\mathbb{R}$	$\mathbb{R}$
$\frac{\partial y}{\partial \boldsymbol{x}}$	$\mathbb{R}^{1\times D}$	$\mathbb{R}^{D\times 1}$
$\frac{\partial \boldsymbol{y}}{\partial x}$	$\mathbb{R}^{L\times 1}$	$\mathbb{R}^{1\times L}$
$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$	$\mathbb{R}^{L\times D}$	$\mathbb{R}^{D\times L}$
$\frac{\partial y}{\partial \boldsymbol{X}}$	$\mathbb{R}^{Q\times P}$	$\mathbb{R}^{P\times Q}$
$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{X}}$	Tensor!	Tensor!

General Training Procedure

- The training of a neural network can be devided into the following steps:
  1. **Initialization** of the parameters $\boldsymbol{W}^{[1]}$, $\boldsymbol{b}^{[1]}$, $\boldsymbol{W}^{[2]}$, $\boldsymbol{b}^{[2]}$, ...
  2. Computation of the **forward pass**
     (i. e. compute the activations of the network for a batch of training examples)
  3. Evaluate the **loss function** based on the true labels
  4. **Propagate the error gradients backwards** through the network and update the parameters *(this step is referred to as **backpropagation**)*
  5. **Update the weights** using the gradient descent update rule
- Steps (2) to (5) are repeated until the network performs satisfactorily or until a fixed number of iterations specified in advance has been performed

Exemplary Network



Computation of the Forward Pass

- Let an input $\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ with label $y \in \{0, 1\}$ be given
- Each layer is parameterized by a weight matrix and a bias vector *(square brackets indicate the layer index)*:

$$\boldsymbol{W}^{[1]} \in \mathbb{R}^{3\times 2}, \qquad \boldsymbol{b}^{[1]} \in \mathbb{R}^{3\times 1}, \qquad \boldsymbol{W}^{[2]} \in \mathbb{R}^{1\times 3}, \qquad b^{[2]} \in \mathbb{R}^{1\times 1}$$

- The output of the network is computed according to:

$$a^{[2]} = \sigma\left(\boldsymbol{W}^{[2]}\boldsymbol{\sigma}\left(\boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}\right) + b^{[2]}\right) \tag{31}$$

Computation of the Forward Pass (Ctd.)

Let us decompose the single elements in equation (31):

$\boldsymbol{x}$	$\in \mathbb{R}^{2\times 1}$	network input
$\boldsymbol{z}^{[1]} := \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$	$\in \mathbb{R}^{3\times 1}$	preactivation in hidden layer
$\boldsymbol{a}^{[1]} := \boldsymbol{\sigma}\left(\boldsymbol{z}^{[1]}\right)$	$\in \mathbb{R}^{3\times 1}$	activation in hidden layer
$z^{[2]} := \boldsymbol{W}^{[2]}\boldsymbol{a}^{[1]} + b^{[2]}$	$\in \mathbb{R}^{1\times 1}$	preactivation in output layer
$a^{[2]} := \sigma\left(z^{[2]}\right)$	$\in \mathbb{R}^{1\times 1}$	network output

Error Function Gradients

- We consider the **binary cross entropy loss**

$$\ell^{\text{BCE}}(\widehat{y}, y) := -y \log(\widehat{y}) - (1 - y) \log(1 - \widehat{y}) \tag{32}$$

- **How to proceed?**
  - We have to compute the gradient of ℓ^{BCE} with respect to all parameters
  - Therefore, we have to compute:

$$\frac{\partial \ell^{\text{BCE}}}{\partial \boldsymbol{W}^{[2]}}, \quad \frac{\partial \ell^{\text{BCE}}}{\partial \boldsymbol{b}^{[2]}}, \quad \frac{\partial \ell^{\text{BCE}}}{\partial \boldsymbol{W}^{[1]}}, \quad \frac{\partial \ell^{\text{BCE}}}{\partial \boldsymbol{b}^{[1]}} \tag{33}$$

  - Let us have a closer look into these gradients on the following slides

Error Function Gradients in the Output Layer

- The gradients in the output layer are given by **(chain rule!)**:

$$\frac{\partial \ell^{\text{BCE}}}{\partial \boldsymbol{W}^{[2]}} = \overbrace{\frac{\partial \ell^{\text{BCE}}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}}}^{=: \delta^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial \boldsymbol{W}^{[2]}} = \delta^{[2]} \cdot \frac{\partial z^{[2]}}{\partial \boldsymbol{W}^{[2]}}$$

$$\frac{\partial \ell^{\text{BCE}}}{\partial \boldsymbol{b}^{[2]}} = \frac{\partial \ell^{\text{BCE}}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial \boldsymbol{b}^{[2]}} = \delta^{[2]} \cdot \frac{\partial z^{[2]}}{\partial \boldsymbol{b}^{[2]}}$$

- We have introduced the quantity $\delta^{[2]}$ because we need it several times
- δ is sometimes referred to as the **error gradient** *(error signal)*

Computation of the Weight Gradient in the Output Layer

Goal: Compute $\frac{\partial \ell^{\text{BCE}}}{\partial \boldsymbol{W}^{[2]}} = \frac{\partial \ell^{\text{BCE}}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial \boldsymbol{W}^{[2]}} = \delta^{[2]} \cdot \frac{\partial z^{[2]}}{\partial \boldsymbol{W}^{[2]}}$

$$\frac{\partial \ell^{\text{BCE}}}{\partial a^{[2]}} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}} = \frac{a^{[2]} - y}{a^{[2]} \cdot \left(1 - a^{[2]}\right)} \in \mathbb{R}^{1\times 1}$$

$$\frac{\partial a^{[2]}}{\partial z^{[2]}} = \sigma\left(z^{[2]}\right) \cdot \left(1 - \sigma\left(z^{[2]}\right)\right) = a^{[2]} \cdot \left(1 - a^{[2]}\right) \in \mathbb{R}^{1\times 1}$$

$$\delta^{[2]} = \frac{\partial \ell^{\text{BCE}}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} = a^{[2]} - y \in \mathbb{R}^{1\times 1}$$

Computation of the Weight Gradient in the Output Layer (Ctd.)

We can now apply identity (27) to compute the weight gradient:

Weight gradient in the output layer:

$$\frac{\partial \ell^{\text{BCE}}}{\partial \boldsymbol{W}^{[2]}} = \delta^{[2]} \cdot \left(\boldsymbol{a}^{[1]}\right)^{\top} \in \mathbb{R}^{1\times 3}$$

Remarks:

- We do not have to transpose $\delta^{[2]}$ because it is a scalar
- We have replaced $\boldsymbol{x}$ with $\boldsymbol{a}^{[1]}$ in equation (27), since $\boldsymbol{a}^{[1]}$ is the input to the output layer

Computation of the Weight Gradient in the Output Layer (Ctd.)

Goal: Compute $\frac{\partial \ell^{\text{BCE}}}{\partial b^{[2]}} = \frac{\partial \ell^{\text{BCE}}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial b^{[2]}} = \delta^{[2]} \cdot \frac{\partial z^{[2]}}{\partial b^{[2]}}$

- Similarly we can compute $\frac{\partial \ell^{\text{BCE}}}{\partial b^{[2]}}$
- We have already computed $\delta^{[2]}$ above

Bias weight gradient in the output layer:

$$\frac{\partial \ell^{\text{BCE}}}{\partial b^{[2]}} = \delta^{[2]} \frac{\partial z^{[2]}}{\partial b^{[2]}} = \delta^{[2]} \cdot 1 = \delta^{[2]} \in \mathbb{R}^{1 \times 1}$$

Error Function Gradients in the hidden Layer

- The gradients in the hidden layer are given by **(chain rule!)**:

$$\frac{\partial \ell^{\text{BCE}}}{\partial \boldsymbol{W}^{[1]}} = \overbrace{\underbrace{\frac{\partial \ell^{\text{BCE}}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}}}_{=:\,\delta^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial \boldsymbol{a}^{[1]}} \odot \frac{\partial \boldsymbol{a}^{[1]}}{\partial \boldsymbol{z}^{[1]}}}^{=:\,\boldsymbol{\delta}^{[1]}} \cdot \frac{\partial \boldsymbol{z}^{[1]}}{\partial \boldsymbol{W}^{[1]}} = \boldsymbol{\delta}^{[1]} \cdot \frac{\partial \boldsymbol{z}^{[1]}}{\partial \boldsymbol{W}^{[1]}}$$

$$\frac{\partial \ell^{\text{BCE}}}{\partial \boldsymbol{b}^{[1]}} = \frac{\partial \ell^{\text{BCE}}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial \boldsymbol{a}^{[1]}} \odot \frac{\partial \boldsymbol{a}^{[1]}}{\partial \boldsymbol{z}^{[1]}} \cdot \frac{\partial \boldsymbol{z}^{[1]}}{\partial \boldsymbol{b}^{[1]}} = \boldsymbol{\delta}^{[1]} \cdot \frac{\partial \boldsymbol{z}^{[1]}}{\partial \boldsymbol{b}^{[1]}}$$

- Please note that $\boldsymbol{\delta}^{[1]} := \delta^{[2]} \cdot \frac{\partial z^{[2]}}{\partial \boldsymbol{a}^{[1]}} \odot \frac{\partial \boldsymbol{a}^{[1]}}{\partial \boldsymbol{z}^{[1]}} \in \mathbb{R}^{1 \times 3}$

Computation of the Weight Gradient in the hidden Layer

Goal: Compute $\frac{\partial \ell^{\text{BCE}}}{\partial \boldsymbol{W}^{[1]}} = \frac{\partial \ell^{\text{BCE}}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial \boldsymbol{a}^{[1]}} \odot \frac{\partial \boldsymbol{a}^{[1]}}{\partial \boldsymbol{z}^{[1]}} \cdot \frac{\partial \boldsymbol{z}^{[1]}}{\partial \boldsymbol{W}^{[1]}} = \boldsymbol{\delta}^{[1]} \cdot \frac{\partial \boldsymbol{z}^{[1]}}{\partial \boldsymbol{W}^{[1]}}$

$$\frac{\partial z^{[2]}}{\partial \boldsymbol{a}^{[1]}} = \boldsymbol{W}^{[2]} \qquad \in \mathbb{R}^{1\times 3}$$

$$\frac{\partial \boldsymbol{a}^{[1]}}{\partial \boldsymbol{z}^{[1]}} = \left(\boldsymbol{a}^{[1]} \odot \left(\boldsymbol{1} - \boldsymbol{a}^{[1]}\right)\right)^\top \qquad \in \mathbb{R}^{1\times 3}$$

$$\boldsymbol{\delta}^{[1]} = \delta^{[2]} \cdot \boldsymbol{W}^{[2]} \odot \left(\boldsymbol{a}^{[1]} \odot \left(\boldsymbol{1} - \boldsymbol{a}^{[1]}\right)\right)^\top \qquad \in \mathbb{R}^{1\times 3}$$

Computation of the Weight Gradient in the hidden Layer

Again we apply identity (27) to obtain the weight gradient:

Weight gradient in the hidden layer:

$$\frac{\partial \ell^{\text{BCE}}}{\partial \boldsymbol{W}^{[1]}} = \left(\boldsymbol{\delta}^{[1]}\right)^{\top} \boldsymbol{x}^{\top} \in \mathbb{R}^{3\times 2}$$

(This time we have to apply the transposition because $\boldsymbol{\delta}^{[1]}$ is a vector)

Computation of the Weight Gradient in the hidden Layer (Ctd.)

Goal: Compute $\frac{\partial \ell^{\text{BCE}}}{\partial \boldsymbol{b}^{[1]}} = \frac{\partial \ell^{\text{BCE}}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial \boldsymbol{a}^{[1]}} \odot \frac{\partial \boldsymbol{a}^{[1]}}{\partial \boldsymbol{z}^{[1]}} \cdot \frac{\partial \boldsymbol{z}^{[1]}}{\partial \boldsymbol{b}^{[1]}} = \boldsymbol{\delta}^{[1]} \cdot \frac{\partial \boldsymbol{z}^{[1]}}{\partial \boldsymbol{b}^{[1]}}$

Bias weight gradient in the hidden layer:

$$\frac{\partial \ell^{\text{BCE}}}{\partial \boldsymbol{b}^{[1]}} = \boldsymbol{\delta}^{[1]} \frac{\partial \boldsymbol{z}^{[1]}}{\partial \boldsymbol{b}^{[1]}} = \left(\boldsymbol{\delta}^{[1]} \boldsymbol{I}\right)^{\top} = \left(\boldsymbol{\delta}^{[1]}\right)^{\top} \in \mathbb{R}^{3 \times 1}$$

Remark: Actually, $\frac{\partial \ell^{\text{BCE}}}{\partial \boldsymbol{b}^{[1]}}$ is a row vector when using numerator layout. However, we want the shape of the gradient to match the shape of $\boldsymbol{b}^{[1]}$ which is why we additionally apply transposition!

Backpropagation with Softmax Activations

- Let L denote the index of the output layer
- For a single **sigmoid unit** in the output layer we had:

$$\delta^{[L]} := \frac{\partial \ell^{\text{BCE}}}{\partial z^{[L]}} = \frac{\partial \ell^{\text{BCE}}}{\partial a^{[L]}} \cdot \frac{\partial a^{[L]}}{\partial z^{[L]}} = \frac{a^{[L]} - y}{a^{[L]} \cdot \left(1 - a^{[L]}\right)} \cdot a^{[L]} \cdot \left(1 - a^{[L]}\right) = a^{[L]} - y$$

- We get a similar result for **softmax activations**

$$\boldsymbol{\delta}^{[L]} := \frac{\partial \ell^{\text{CE}}}{\partial \boldsymbol{z}^{[L]}} = \left(\boldsymbol{a}^{[L]} - \boldsymbol{y}\right)^{\top} \tag{34}$$

Backpropagation with Softmax Activations – Proof

Proof: Let $\boldsymbol{a} = (a_1 \ \dots \ a_K)^\top \in \mathbb{R}^K$ with $a_k = \zeta_k(\boldsymbol{z})$ be the output of the last layer of the neural network and let $\boldsymbol{z} \in \mathbb{R}^K$ be the respective preactivations. We want to compute the error gradient $\boldsymbol{\delta} := \frac{\partial \ell^{\text{CE}}}{\partial \boldsymbol{z}} = \frac{\partial \ell^{\text{CE}}}{\partial \boldsymbol{a}} \cdot \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{z}}$, where the cost function ℓ^{CE} is given by

$$\ell^{\text{CE}}(\boldsymbol{a}, \boldsymbol{y}) := -\sum_{k=1}^{K} y_k \log\big(\zeta_k(\boldsymbol{z})\big) = -\sum_{k=1}^{K} y_k \log(a_k).$$

For the first factor produced by the chain rule we obtain

$$\frac{\partial \ell^{\text{CE}}}{\partial \boldsymbol{a}} = -\begin{pmatrix} \frac{y_1}{a_1} & \frac{y_2}{a_2} & \dots & \frac{y_K}{a_K} \end{pmatrix} \in \mathbb{R}^{1 \times K}. \tag{35}$$

Backpropagation with Softmax Activations – Proof (Ctd.)

Concerning the second factor we had seen that the derivative of softmax function is

$$\frac{\partial a_k}{\partial z_j} = \begin{cases} a_j(1 - a_j) & \text{if } k = j \\ -a_k a_j & \text{if } k \neq j. \end{cases}$$

Therefore, the JACOBIan matrix of the softmax function is given by

$$\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{z}} = \begin{pmatrix} a_1(1 - a_1) & -a_1 a_2 & \dots & -a_1 a_K \\ -a_2 a_1 & a_2(1 - a_2) & \dots & -a_2 a_K \\ \vdots & \vdots & \ddots & \vdots \\ -a_K a_1 & -a_K a_2 & \dots & a_K(1 - a_K) \end{pmatrix} \in \mathbb{R}^{K \times K}. \tag{36}$$

Backpropagation with Softmax Activations – Proof (Ctd.)

Putting the results (35) and (36) together, we obtain

$$\frac{\partial \ell^{\text{CE}}}{\partial \boldsymbol{z}} = \frac{\partial \ell^{\text{CE}}}{\partial \boldsymbol{a}} \cdot \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{z}} = -\begin{pmatrix} \frac{y_1}{a_1} & \dots & \frac{y_K}{a_K} \end{pmatrix} \begin{pmatrix} a_1(1-a_1) & \dots & -a_1 a_K \\ \vdots & \ddots & \vdots \\ -a_K a_1 & \dots & a_K(1-a_K) \end{pmatrix}$$

$$= \begin{pmatrix} -y_1 + a_1 \sum_{k=1}^{K} y_k & \dots & -y_K + a_K \sum_{k=1}^{K} y_k \end{pmatrix} = \begin{pmatrix} a_1 - y_1 & \dots & a_K - y_K \end{pmatrix}$$

$$= (\boldsymbol{a} - \boldsymbol{y})^\top \in \mathbb{R}^{1 \times K}.$$

Thus, $\boldsymbol{\delta} := (\boldsymbol{a} - \boldsymbol{y})^\top$ as claimed. ■

Backpropagation Algorithm (single Training Example)

Input: A single training example with features $\boldsymbol{x}$ and one-hot encoded label $\boldsymbol{y}$

Forward pass: Compute $\boldsymbol{z}^{[\lambda]}$ and $\boldsymbol{a}^{[\lambda]}$ for all L layers of the network $(\lambda = 1, \ldots, L)$

Compute the cost $\ell\left(\boldsymbol{a}^{[L]}, \boldsymbol{y}\right)$ based on the prediction and the gold label

Backward pass: Compute $\boldsymbol{\delta}^{[L]} := \frac{\partial \ell}{\partial \boldsymbol{z}^{[L]}} = \left(\boldsymbol{a}^{[L]} - \boldsymbol{y}\right)^\top$

for $\lambda = L - 1$ **to** 0 **do**

  if $\lambda \neq 0$ **then**

    Compute:

$$\boldsymbol{\delta}^{[\lambda]} := \frac{\partial \ell}{\partial \boldsymbol{z}^{[\lambda]}} = \left(\boldsymbol{\delta}^{[\lambda+1]} \boldsymbol{W}^{[\lambda+1]}\right) \odot \boldsymbol{g}'\left(\boldsymbol{z}^{[\lambda]}\right)^\top$$

  end

  Compute:

$$\frac{\partial \ell}{\partial \boldsymbol{W}^{[\lambda+1]}} = \left(\boldsymbol{\delta}^{[\lambda+1]}\right)^\top \left(\boldsymbol{a}^{[\lambda]}\right)^\top \quad \text{and} \quad \frac{\partial \ell}{\partial \boldsymbol{b}^{[\lambda+1]}} = \left(\boldsymbol{\delta}^{[\lambda+1]}\right)^\top$$

end

Backpropagation Algorithm – Remarks

Some remarks:

- In the backpropagation algorithm above we have set $\boldsymbol{a}^{[0]} := \boldsymbol{x}$, i. e. the 'activation' of the first layer is given by the network inputs
- The function $\boldsymbol{g}$ denotes a generic **activation function** *(see below for details)*
- For the moment consider:

$$\boldsymbol{g}(\boldsymbol{z}) := \boldsymbol{\sigma}(\boldsymbol{z}) \quad \Rightarrow \quad \boldsymbol{g}'(\boldsymbol{z}) = \boldsymbol{\sigma}'(\boldsymbol{z}) = \boldsymbol{\sigma}(\boldsymbol{z}) \odot \big(\mathbf{1} - \boldsymbol{\sigma}(\boldsymbol{z})\big) \tag{37}$$

- Backpropagation does not update the parameters, it only computes the gradients

Update of the Parameters

- We have computed the gradients $\frac{\partial \ell}{\partial \boldsymbol{W}^{[\lambda]}}$ and $\frac{\partial \ell}{\partial \boldsymbol{b}^{[\lambda]}}$ for all layers $\lambda = 1, 2, \ldots, L$
- We can use the **gradient descent update formula** to update the network parameters for all layers $\lambda = 1, 2, \ldots, L$

$$\left(\boldsymbol{W}^{[\lambda]}\right)^{t+1} \longleftarrow \left(\boldsymbol{W}^{[\lambda]}\right)^{t} - \alpha \frac{\partial \ell}{\partial \boldsymbol{W}^{[\lambda]}} \tag{38}$$

$$\left(\boldsymbol{b}^{[\lambda]}\right)^{t+1} \longleftarrow \left(\boldsymbol{b}^{[\lambda]}\right)^{t} - \alpha \frac{\partial \ell}{\partial \boldsymbol{b}^{[\lambda]}} \tag{39}$$

- In practice we use more sophisticated algorithms to optimize deep neural networks *(see below for more details)*

Training the Network in Batches

- Usually, we train the network in **batches of a certain size**
- The neural network formulas can be adjusted to deal with multiple training examples simultaneously (consider batches of size N_b)
- First of all, we stack all training examples column-wise into a matrix $\boldsymbol{X}$:

$$\boldsymbol{X} = \begin{pmatrix} | & | & | & | \\ \boldsymbol{x}^1 & \boldsymbol{x}^2 & \dots & \boldsymbol{x}^{N_b} \\ | & | & | & | \end{pmatrix} \in \mathbb{R}^{M \times N_b} \tag{40}$$

- Similarly we stack the one-hot encoded labels columnwise into a matrix $\boldsymbol{Y}$

Training the Network in Batches – Issue 1

- In the forward pass:

$$\boldsymbol{W}\boldsymbol{X} + \boldsymbol{b} = \lightning$$

- `Numpy` deals with this issue automatically for you by applying **broadcasting**
- This means it **implicitly** repeats $\boldsymbol{b}$ columnwise to get a matrix of matching shape:

$$\boldsymbol{B} = \begin{pmatrix} | & | & | & | \\ \boldsymbol{b} & \boldsymbol{b} & \dots & \boldsymbol{b} \\ | & | & | & | \end{pmatrix} \in \mathbb{R}^{H \times N_b}$$

where H is the size of the hidden layer

Training the Network in Batches – Issue 2

- When updating the bias parameters:

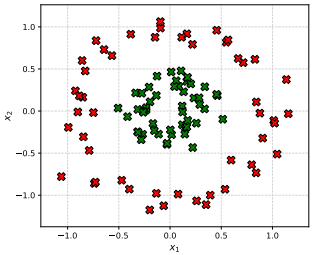
$$\left(\boldsymbol{b}^{[\lambda]}\right)^{t+1} \longleftarrow \left(\boldsymbol{b}^{[\lambda]}\right)^{t} \lightning - \alpha \frac{\partial \ell}{\partial \boldsymbol{b}^{[\lambda]}} = \left(\boldsymbol{b}^{[\lambda]}\right)^{t} \lightning - \alpha \left(\boldsymbol{\Delta}^{[\lambda]}\right)^{\top}$$

- The error gradient $\boldsymbol{\delta}^{[\lambda]}$ has turned into a matrix $\boldsymbol{\Delta}^{[\lambda]}$
- Before updating $\boldsymbol{b}^{[\lambda]}$, we first have to compute the sum of each row of $\boldsymbol{\Delta}^{[\lambda]}$
- How to do it in `Numpy`:

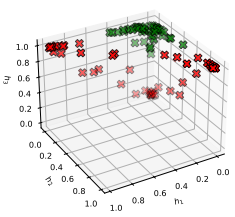
```
grad_b = np.sum(Delta.T, axis=1, keepdims=True)
```

Hidden Layers are Feature Learners!

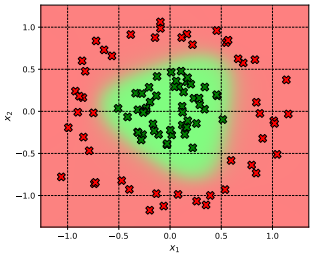
Input layer:



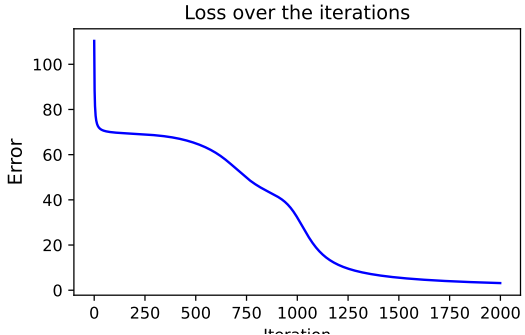
Hidden layer:



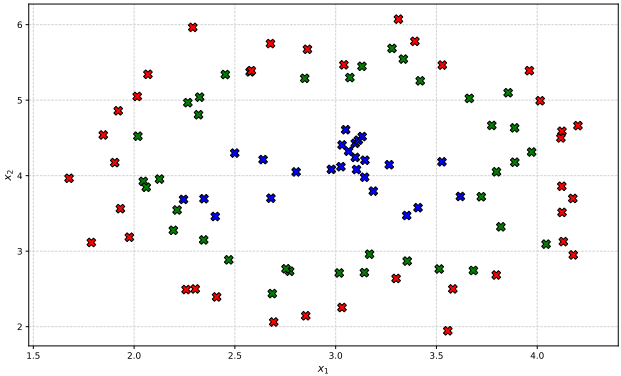
Output layer:



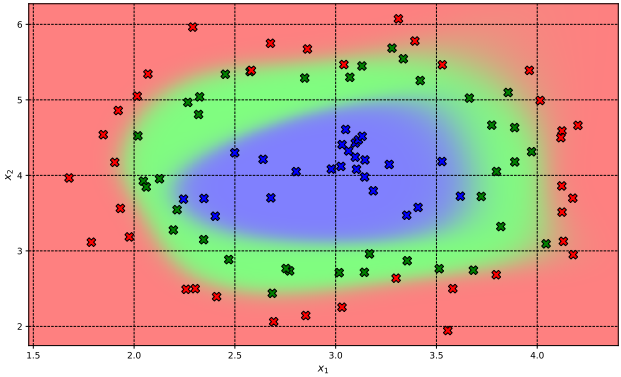
Error Reduction over the Iterations



Three-Class Problem



Three-Class Problem (Ctd.)



Section:

Neural Network Extensions and Improvements

Activation Functions
Parameter Initialization Techniques
Optimization Algorithms for Deep Learning Models
Convolutional Neural Networks (CNNs)
Training of Convolutional Neural Networks (CNNs)

Why Activation Functions?

Question: Why don't we use $g(z) = z$ (identity function)?

- For simplicity consider $\boldsymbol{b}^{[1]} = \mathbf{0}$ and $\boldsymbol{b}^{[2]} = \mathbf{0}$

$$a^{[2]} = g\Big(\boldsymbol{W}^{[2]}\boldsymbol{g}\big(\boldsymbol{W}^{[1]}\boldsymbol{x}\big)\Big) = \boldsymbol{W}^{[2]}\boldsymbol{W}^{[1]}\boldsymbol{x} = \widetilde{\boldsymbol{W}}\boldsymbol{x}$$

- Notice how $\boldsymbol{W}^{[2]}\boldsymbol{W}^{[1]}$ collapses into $\widetilde{\boldsymbol{W}}$
- Applying a linear function to a linear function results in a linear function
- **The neural network simply performs linear regression!**

Sigmoid Function (σ)

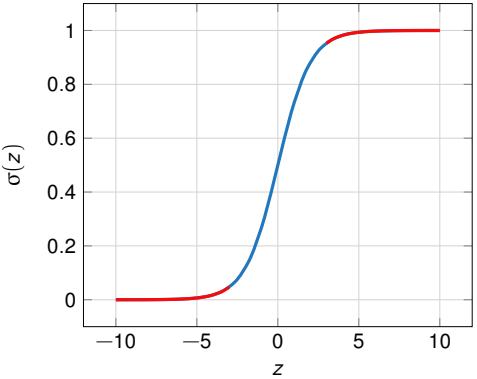
Sigmoid function: The output of the function lies in the interval $[0, 1]$

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \qquad \sigma'(z) = \sigma(z) \cdot \big(1 - \sigma(z)\big)$$

Problems:

- The gradients are small when $|z|$ is big **(vanishing gradient problem)**
- Output is not zero-centered **(makes optimization harder)**

Vanishing Gradients



There is almost no slope in the red areas!

$\Rightarrow$ No gradient, no learning!

Tangent Hyperbolic (tanh)

Tangent hyperbolic: The output of the function lies in the interval $[-1, 1]$

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \qquad \tanh'(z) = 1 - \tanh^2(z)$$

Advantage: The output of the function is **zero-centered**

Problem: The function also suffers from the **vanishing gradient problem**

Rectified Linear Unit (ReLU)

Rectified Linear Unit (ReLU):

$$\text{ReLU}(z) = \max(0, z), \qquad \text{ReLU}'(z) = \begin{cases} 0 & \text{if } z \leqslant 0 \\ 1 & \text{if } z > 0 \end{cases}$$

Advantage: No vanishing gradients and efficient to compute

Problem: Dying ReLUs always output 0, use **batch-normalization!**

Activation Functions Overview

Name	Function	Derivative
Sigmoid	$g(x) = \frac{1}{1+e^{-x}}$ 	$g'(x) = g(x) \cdot \big(1 - g(x)\big)$
Tanh	$g(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ 	$g'(x) = 1 - g^2(x)$
ReLU	$g(x) = \max(0, x)$ 	$g'(x) = \begin{cases} 0 & \text{if } x \leqslant 0 \\ 1 & \text{if } x > 0 \end{cases}$

Activation Functions Overview (Ctd.)

Name	Function	Derivative
Leaky ReLU	$g(x) = \begin{cases} \alpha x & \text{if } x \leqslant 0 \\ x & \text{if } x > 0 \end{cases}$ 	$g'(x) = \begin{cases} \alpha & \text{if } x \leqslant 0 \\ 1 & \text{if } x > 0 \end{cases}$
ELU	$g(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x \leqslant 0 \\ x & \text{if } x > 0 \end{cases}$ 	$g'(x) = \begin{cases} \alpha e^x & \text{if } x \leqslant 0 \\ 1 & \text{if } x > 0 \end{cases}$

Global Activation: Softmax Activation

Softmax:

$$\zeta : \mathbb{R}^K \to \mathbb{R}^K, \qquad \boldsymbol{z} \mapsto \zeta(\boldsymbol{z}), \qquad \zeta_k(\boldsymbol{z}) := \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \tag{41}$$

- ζ is a **global activation function** *(it also depends on the preactivations of the other units in the layer)*
- It is used to convert the output of the network into a probability distribution, i. e. **the activations sum to 1**

Parameter Initialization Schemes

- Neural networks often suffer from **vanishing** or **exploding gradients**
- We want the signal to flow properly through the network:
  1. In the forward direction when making predictions
  2. In the reverse direction when backpropagating the gradients

Question: Can we initialize the weights so as to avoid these issues?

In general, the network weights should be initialized to small random numbers around 0. **Do not set initial values of 0 for all weights! (Why?)**

XAVIER Initialization

- We use a zero-mean GAUSSian distribution with finite variance σ^2, i. e.

$$w \sim \mathcal{N}\left(0, \sigma^2\right)$$

- We want **the variance of the output of a layer to be roughly equal to the variance of the inputs to the layer**
- Suppose we have inputs $\boldsymbol{x} \in \mathbb{R}^M$ (zero mean) and a linear neuron with random weights $\boldsymbol{w} \in \mathbb{R}^M$; Further assume that the inputs $\boldsymbol{x}$ and the weights $\boldsymbol{w}$ are i. i. d.
- Ignoring the bias, the output a is computed according to

$$a = \boldsymbol{w}^\top \boldsymbol{x} = w_1 x_1 + w_2 x_2 + \ldots + w_M x_M$$

XAVIER Initialization (Ctd.)

- The variance of the output a is then given by (due to i. i. d. assumption):

$$\begin{aligned}\mathbb{V}\{a\} &= \mathbb{V}\{w_1 x_1 + w_2 x_2 + \ldots + w_M x_M\} \\ &= \mathbb{V}\{w_1 x_1\} + \mathbb{V}\{w_2 x_2\} + \ldots + \mathbb{V}\{w_M x_M\} \\ &= M \cdot \mathbb{V}\{w_m\}\mathbb{V}\{x_m\}\end{aligned}$$

- We see that the variance of the output is equal to the variance of the inputs **scaled by the factor**

$$M \cdot \mathbb{V}\{w\}$$

XAVIER Initialization (Ctd.)

- We said we want $\mathbb{V}\{a\} \approx \mathbb{V}\{x\}$
- Therefore, the scaling factor $M \cdot \mathbb{V}\{w\}$ has to be (roughly) equal to 1
- We can achieve this by setting the variance of the weights to be

$$\mathbb{V}\{w\} = \frac{1}{M} = \frac{1}{M_{\text{in}}}$$

XAVIER initialization scheme:

$$w \sim \mathcal{N}\left(0, 1/M_{\text{in}}\right) \tag{42}$$

GLOROT Initialization

- For the backward pass we obtain a similar result when applying the same steps:

$$\mathbb{V}\{w\} = \frac{1}{M_{\text{out}}}$$

- GLOROT and BENGIO suggest to use the average of M_{in} and M_{out}:

$$\mathbb{V}\{w\} = \frac{1}{M_{\text{avg}}}, \qquad \text{where} \qquad M_{\text{avg}} := \frac{M_{\text{in}} + M_{\text{out}}}{2}$$

GLOROT initialization scheme:

$$w \sim \mathcal{N}\left(0, 1/M_{\text{avg}}\right) \tag{43}$$

He Initialization

- Glorot and Bengio considered sigmoid activations
- In case of ReLU activations, another initialization scheme is advantageous

He initialization scheme:

$$w \sim \mathcal{N}\left(0, {}^{2}/_{M_{\text{avg}}}\right) \tag{44}$$

Please note that the Glorot and He initialization schemes only differs by a constant factor of 2

Optimization Algorithms with adaptive Learning Rates

- The learning rate α is one of the most difficult to set hyperparameters
- The cost function may be highly sensitive to some directions in parameter space and insensitive to others
- **Idea:**
  - Use a separate learning rate for each parameter
  - Automatically update the learning rates throughout the course of learning
- We will discuss two closely related algorithms:
  - **AdaGrad (Adaptive Gradient)**
  - **RMSProp (Root Mean Square Propagation)**

AdaGrad

- AdaGrad adapts the learning rates by scaling them **inversely proportional to the square root of the sum of all historical squared gradients**
- This means
  - The learning rates of parameters with **large** partial derivatives are decreased **more**
  - The learning rates of parameters with **small** partial derivatives are decreased **less**

Disadvantage: For highly non-convex functions (e. g. the cost function in deep learning), the accumulation of gradients **from the beginning of training** can result in a **premature decrease of the learning rate**

AdaGrad Algorithm

Input: Global learning rate α, initial parameters $\boldsymbol{\theta}$, small constant ε for numerical stability (e. g. 10^{-7})

Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$

while *stopping criterion not met* **do**

- Sample a minibatch of N_b examples from the training set
- Compute the loss gradient $\boldsymbol{g}$ with respect to the parameters
- Accumulate the gradient: $\boldsymbol{r} \longleftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
- Compute update: $\boldsymbol{\delta_\theta} \longleftarrow -\frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$
- Apply update: $\boldsymbol{\theta} \longleftarrow \boldsymbol{\theta} + \boldsymbol{\delta_\theta}$

end

RMSProp

- The RMSProp algorithm was proposed by GEOFFREY HINTON
  (he did this in his ⇒ Coursera class on neural networks)
- RMSProp modifies AdaGrad to perform better in the non-convex setting
- The gradient accumulation is changed into an **exponentially weighted moving average** by introducing another hyperparamter ρ (decay rate)
- Adam $\approx$ RMSProp + Momentum (Adam stands for ***Ada**ptive **M**oment Estimation*)

Advantage: Unlike AdaGrad, RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge more rapidly

RMSProp Algorithm

Input: Global learning rate α, initial parameters $\boldsymbol{\theta}$, small constant ε for numerical stability (e. g. 10^{-7}), decay rate ρ (default: 0.9)

Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$

while *stopping criterion not met* **do**

- Sample a minibatch of N_b examples from the training set
- Compute the loss gradient $\boldsymbol{g}$ with respect to the parameters
- Accumulate the gradient: $\boldsymbol{r} \longleftarrow \rho \boldsymbol{r} + (1 - \rho)\boldsymbol{g} \odot \boldsymbol{g}$
- Compute update: $\boldsymbol{\delta_\theta} \longleftarrow -\frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$
- Apply update: $\boldsymbol{\theta} \longleftarrow \boldsymbol{\theta} + \boldsymbol{\delta_\theta}$

end

Portrait: GEOFFREY HINTON

GEOFFREY HINTON (born 6 December 1947) is a British-Canadian computer scientist and Nobel Prize winner in Physics, known for his work on artificial neural networks which earned him the title as the 'Godfather of AI'. HINTON is University Professor Emeritus at the University of Toronto.

With DAVID RUMELHART and RONALD J. WILLIAMS, he was co-author of a highly cited ⇒ paper published in 1986 that popularised the backpropagation algorithm for training multi-layer neural networks, although they were not the first to propose the approach. HINTON is viewed as a leading figure in the deep learning community. He received the 2018 Turing Award, together with YOSHUA BENGIO and YANN LECUN, for their work on deep learning.

(Wikipedia)

Hyperparameter Optimization

Hyperparameter optimization is absolutely necessary:

- # hidden layers
- # hidden units
- activation functions
- learning rate
- batch size
- # training epochs
- regularization
- ...

Have a look at: `https://playground.tensorflow.org`

Convolutional Neural Networks (CNNs)

A fully connected network (MLP) has lots of parameters, i. e. it might be **too complex** or **computationally inefficient** for some tasks

- The **input position** does not matter in all cases *(e. g. word positions when classifying e-mails, pixels in images)*
- An MLP is fed with a fixed-size input vector, but we might have **variable-size input data** *(e. g. images in different resolutions, text sequences of different lengths)*

Solution: Use **Convolutional Neural Networks (CNNs)**

General Idea of CNNs

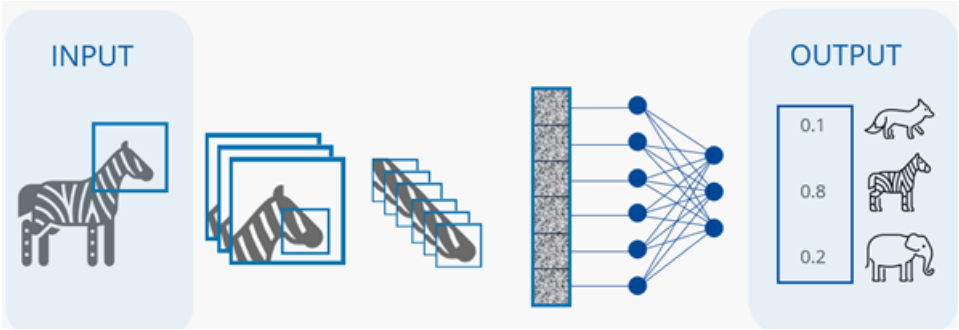
'A convolutional neural network is designed to identify indicative local predictors in a large structure, and combine them to produce a fixed size vector representation of the structure, capturing these local aspects that are most informative for the prediction task at hand.'

YOAV GOLDBERG

General Idea of CNNs (Ctd.)

- A CNN consists of multiple **convolutional layers**
  - Such a layer usually receives as input an image or the output of a previous convolutional layer
  - Each convolutional layer has one or more **filters** (also known as **kernels**) which contain the trainable parameters of the layer
  - It is possible to apply a non-linear activation function to the convolution output
- A convolutional layer is usually followed by a **pooling layer**
  - Pooling extracts the most important features **independent of their position**
  - It reduces the size of the input to the next layer
  - A pooling layer **does not** have trainable parameters

General Idea of CNNs (Ctd.)



Portrait: YANN LECUN

YANN LECUN (born 8 July 1960) is a French-American computer scientist working primarily in the fields of machine learning, computer vision, mobile robotics and computational neuroscience. He is the Silver Professor of the Courant Institute of Mathematical Sciences at New York University and Vice President, Chief AI Scientist at Meta.

He is well known for his work on optical character recognition and computer vision using convolutional neural networks (CNNs). He is also one of the main creators of the DjVu image compression technology. In 2018, LECUN, YOSHUA BENGIO, and GEOFFREY HINTON, received the Turing Award for their work on deep learning. The three are sometimes referred to as the 'Godfathers of AI' and 'Godfathers of Deep Learning'.

(Wikipedia)

Convolution Operator

Convolution Operator:

$$\big(\boldsymbol{I} \otimes \boldsymbol{F}\big)\big(i,j\big) := \sum_m \sum_n \boldsymbol{I}\big(i-m, j-n\big) \cdot \boldsymbol{F}\big(m,n\big)$$

- The symbol $\otimes$ denotes the **convolution operator**, i. e. $\big(\boldsymbol{I} \otimes \boldsymbol{F}\big)$ represents the convolution output $\boldsymbol{O}$
- $\boldsymbol{I}$ is the convolution input *(e. g. an image of a certain size)*
- $\boldsymbol{F}$ is a filter/kernel comprising the adjustable parameters

Convolutional Layer

- We shift the filter ***F*** across the input image ***I*** using a **stride** specified in advance and multiply the input with the filter weights at each position
- The size of the resulting feature map ***O*** depends on the stride and the convolution type (padding)

Input ***I***

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

Filter ***F***

0	1	2
2	2	0
0	0	1

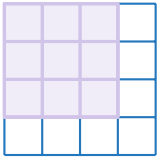
9	8	12
10	15	15
9	6	13

Output ***O***

Types of Convolution (Stride 1)

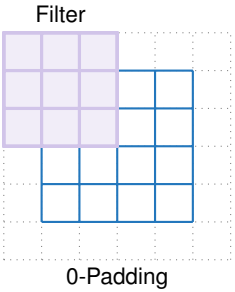
Valid

Output size < Input size



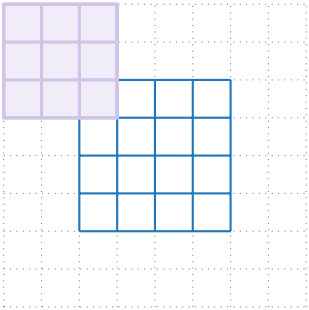
Same

Output size = Input size



Full

Output size > Input size



Size of the Convolution Output

- The output size can be computed according to the formula:

$$\text{Output size} = \frac{\text{Input size} - \text{Kernel size} + 2 \cdot \text{Padding}}{\text{Stride}} + 1 \tag{45}$$

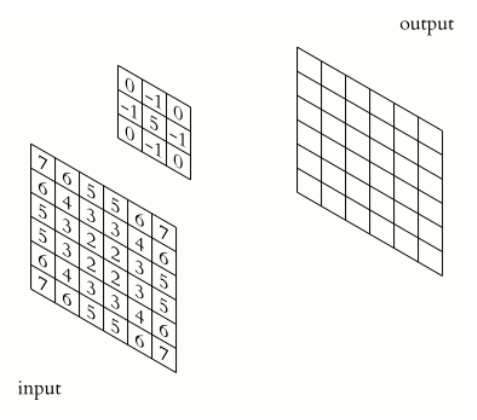
- In the examples above:

$$\text{Output size}_{\text{valid}} = \frac{4 - 3 + 2 \cdot 0}{1} + 1 = 2 \qquad \Rightarrow 2 \times 2$$

$$\text{Output size}_{\text{same}} = \frac{4 - 3 + 2 \cdot 1}{1} + 1 = 4 \qquad \Rightarrow 4 \times 4$$

$$\text{Output size}_{\text{full}} = \frac{4 - 3 + 2 \cdot 2}{1} + 1 = 6 \qquad \Rightarrow 6 \times 6$$

Convolution Animation



Pooling Layer

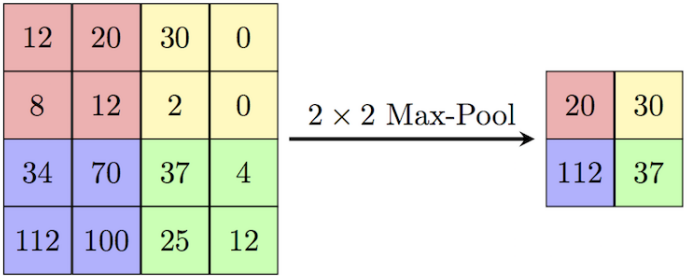
- The convolution output is passed through a **non-linear activation function**
- For pooling we have to specify a **pooling window size** and a **stride**
- **Popular types:**
  - **Max-pooling** *(extracts the maximum activation)*
  - **Mean-pooling** *(extracts the mean activation)*

There are no parameters involved in the pooling operation

In image processing, pooling is necessary to reduce the dimensionality!

Visualization of Pooling

Example: Max pooling with window of size 2×2 and a stride of 2



CNN Training

- CNNs are also trained using the backpropagation algorithm
- However, the derivations are more involved
- If you are interested, have a look at the following resources to get an idea:
  1. https://deeplearning.cs.cmu.edu/F21/document/recitation/Recitation5/CNN_Backprop_Recitation_5_F21.pdf
  2. https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c
  3. https://www.youtube.com/watch?v=z9hJzduHToc

Section:

Wrap-Up

Summary
Recommended Literature
Self-Test Questions
Lecture Outlook

Summary

- Neural Networks are powerful models for pattern recognition
- The perceptron can classify all training examples correctly, **if the training data is linearly separable**
- **MLPs are universal function approximators**
- Backpropagation is a recursive procedure based on the **chain rule of calculus** to obtain the gradients for neural network learning
- In deep learning we usually use more sophisticated optimization algorithms like RMSProp or Adam because the **objective function is not convex**
- Different neural network architectures like CNNs and RNNs exist for solving different kinds of problems

Recommended Literature

1. [BISHOP.2006], section 4.1.7
2. [BISHOP.2006], chapter 5
3. [GOODFELLOW.2016]

(For free PDF versions, see list in GitHub readme!)

Self-Test Questions

1. What is the relation between neural networks and logistic regression?
2. What is a perceptron? Which problems can it solve and which not?
3. Why do we often use multiple layers instead of a simple perceptron?
4. How does backpropagation work? How does a neural network learn?
5. What are advantages and disadvantages of using neural networks?
6. What are CNNs and RNNs? For which tasks are they suitable?

What's next...?

I	Machine Learning Introduction
II	Optimization Techniques
III	Bayesian Decision Theory
IV	Non-parametric Density Estimation
V	Probabilistic Graphical Models
VI	Linear Regression
VII	Logistic Regression
VIII	Deep Learning
IX	Evaluation
X	Decision Trees
XI	Support Vector Machines
XII	Clustering
XIII	Principal Component Analysis
XIV	Reinforcement Learning
XV	Advanced Regression

Thank you very much for the attention!

*** * * Artificial Intelligence and Machine Learning * * ***

Topic: Neural Networks and Deep Learning

Term: Summer term 2025

Contact:
Daniel Wehner, M.Sc.
SAP SE / DHBW Mannheim
daniel.wehner@sap.com

Do you have any questions?