UNIVERSITÄT PASSAU

Complex Systems Engineering

Neural Networks

Prof. Dr. Christin Seifert

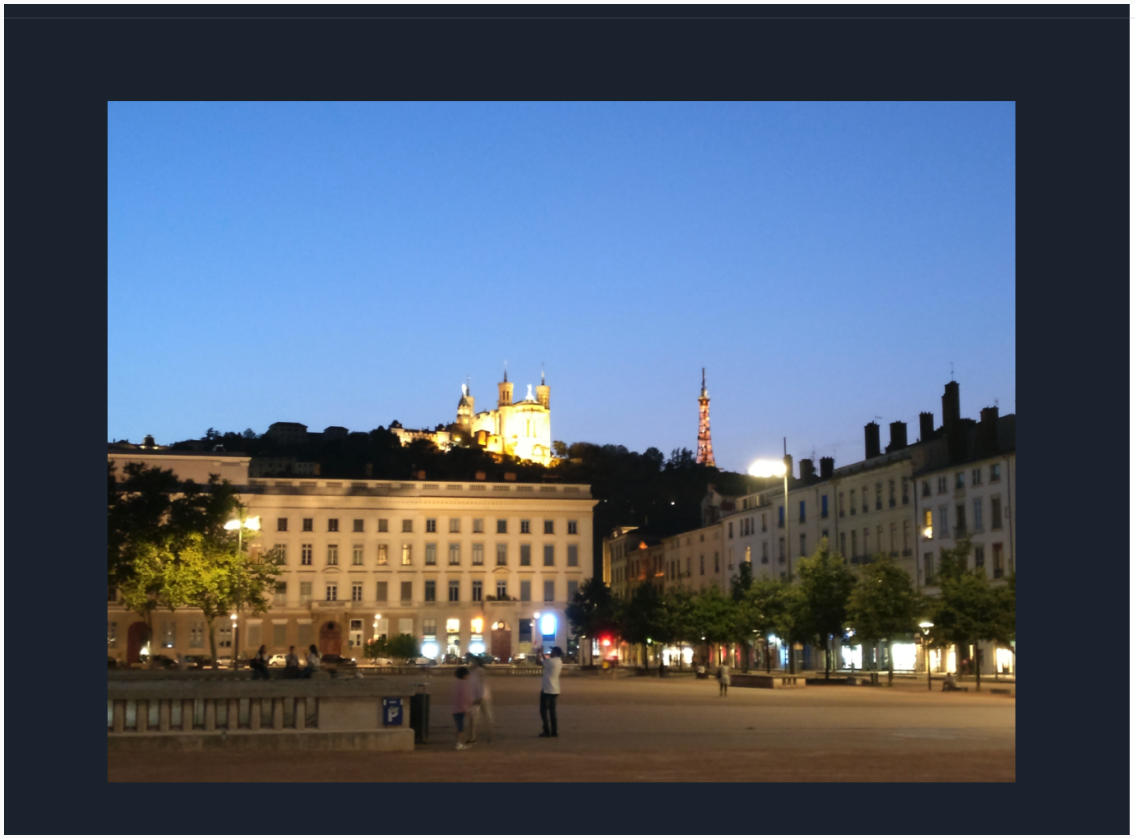
December 13, 2017

University of Passau, WS 2017/2018

Overview

Introduction

Neural networks learn understand the world [13]



outdoors	0.958
sky	0.949
administration	0.949
tourism	0.948
castle	0.931
daylight	0.922
museum	0.908
urban	0.897
water	0.896
tourist	0.894
town	0.889

Figure 1: Scene Understanding (Generated with `https://www.clarifai.com/demo`)

Neural networks writing by hand [4]

- Recurrent neural networks can be used to generated sequences, models can also include memory (Long short-term memory networks, LSTM)
- Network trained on corpus from 221 writers, learnt to produce new text in their handwriting

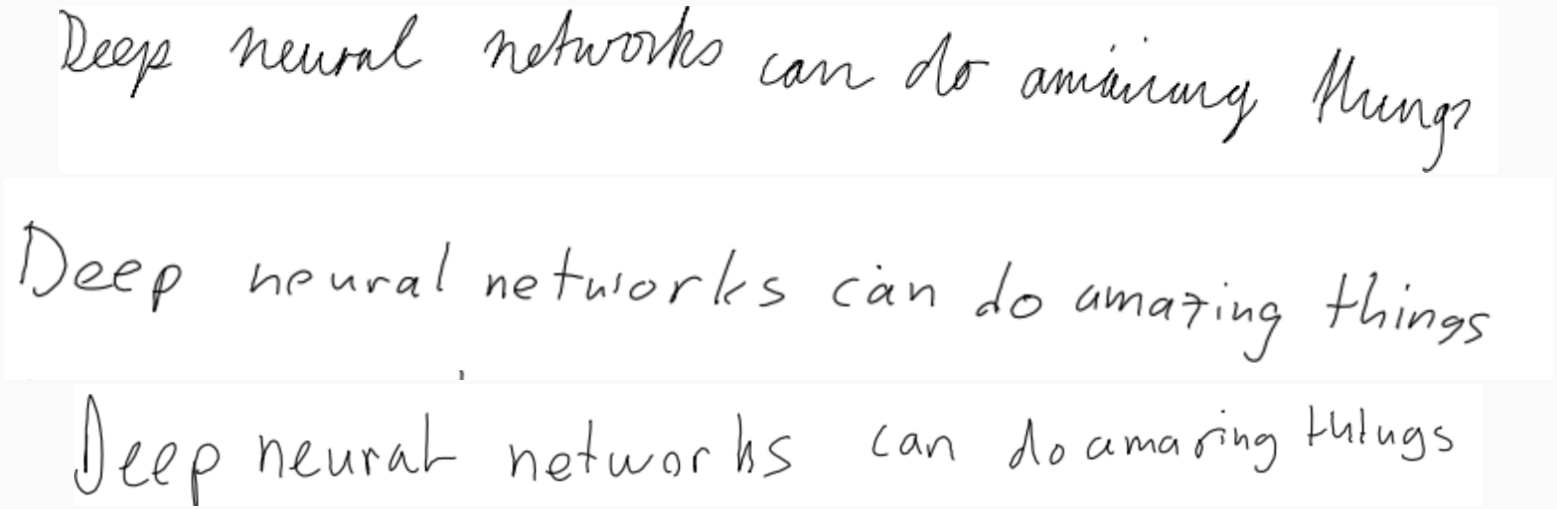


Figure 2: Three handwritten samples for the same text (generated with `http://www.cs.toronto.edu/~graves/handwriting.html`)

Neural networks as artists [1]

- Deep neural network (Convolutional Neural Network) trained on artistic images
- Network learnt to separate image content from image style
- Given a new image it can apply any (learnt) artistic style
- Demo available at `https://deepart.io/`

Figure 3: Input image (left) plus style (right)

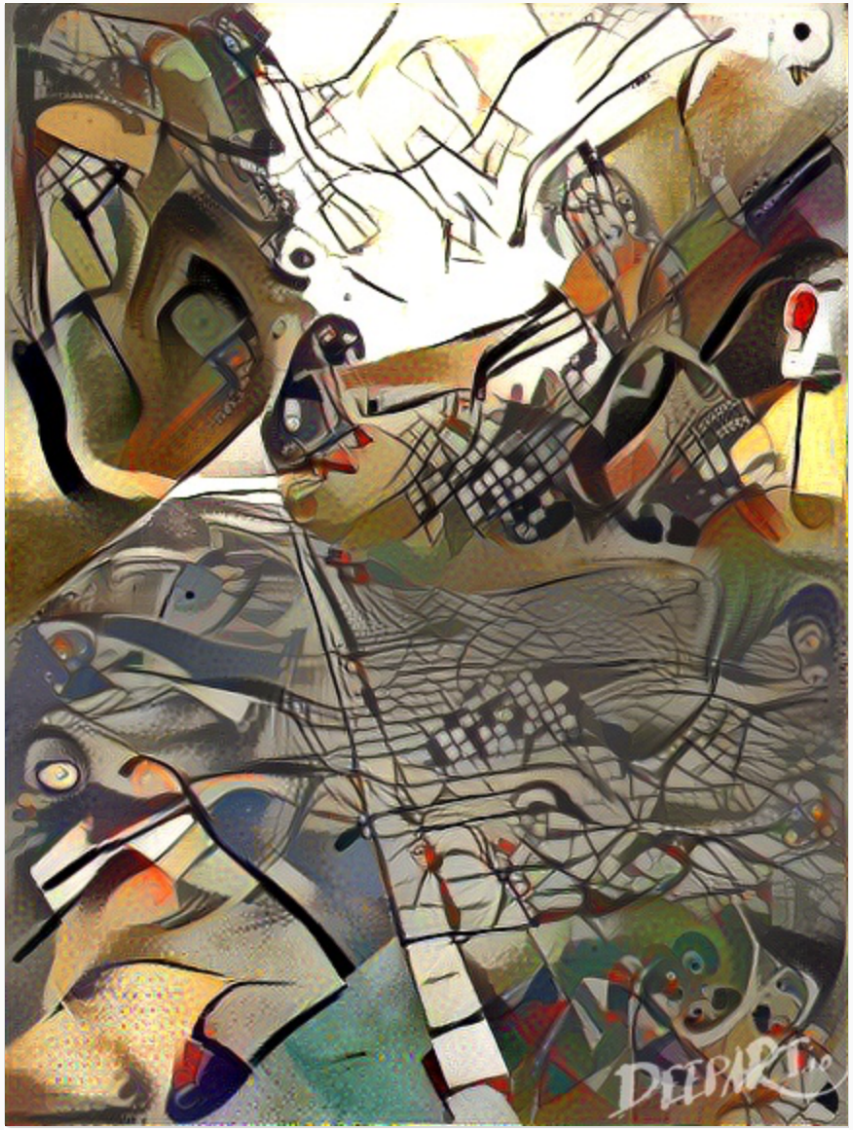


Figure 4: DeepArt.io Result

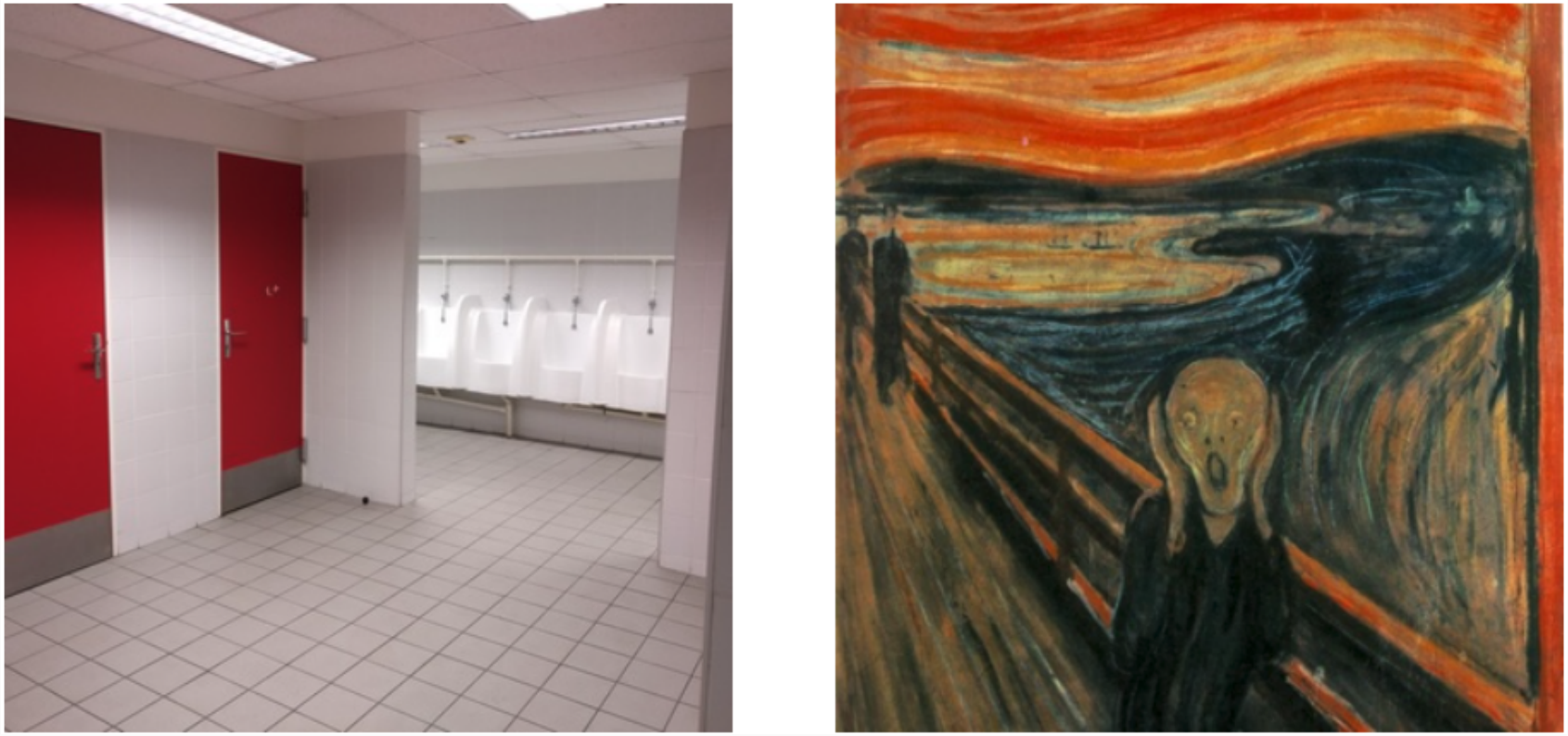


Figure 5: Input image (left) plus style (right)

Figure 6: DeepArt.io Result

Neural networks as strategy game players [8]

- Chinese game Go has 10^{170} board configurations (more than atoms in the universe, much more complex than chess)
- Artificial intelligence AlphaGo uses deep neural networks
- In October 2015 AlphaGo won against the European Go Master Fan Hui (using 1,202 CPUs and 176 GPUs)
- Webpage `https://deepmind.com/research/alphago/`

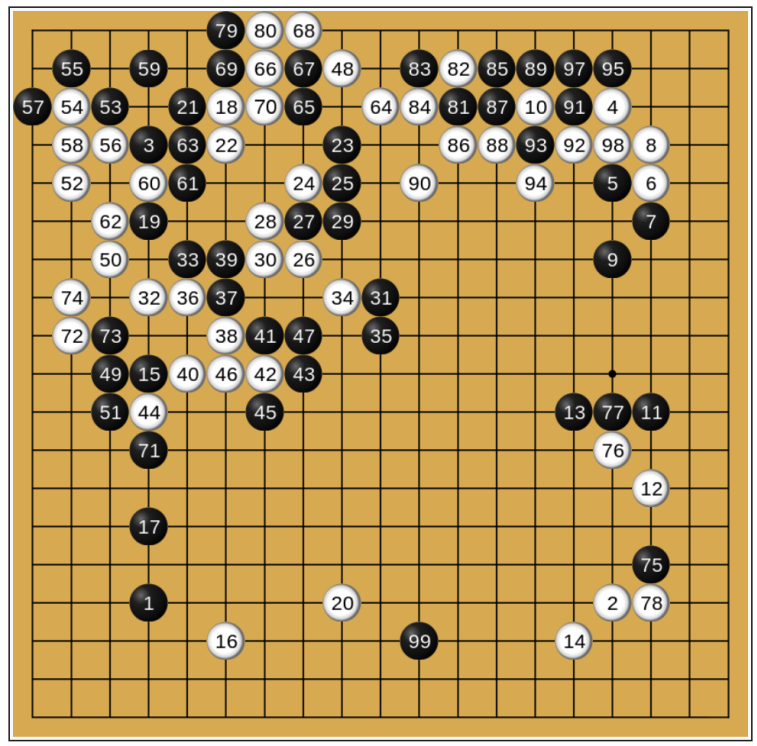


Figure 7: First 99 turn in tournament (Public Domain, via Wikimedia Commons)

Neural networks control robot movements [6]

- Including convolutional neural networks in robot control loops improves movement

Figure 8: Robot grasping office objects (Full video available `https://www.youtube.com/watch?v=H4V6NZLNu-c` (1:50 min))

History

- 1940s–1960s: Cybernetics
  - Theories about how the brain learns
  - Warren Sturgis McCulloch, Walter Pitts, Donald Olding Hebb (Hebbian learning), Frank Rosenblatt (perceptron)
  - Simple linear models, failed to solve easy task (XOR)
- 1980s–1990s: Connectionism
  - David Rummelhart (backpropagation algorithm)
  - Multi-layer networks
  - Outperformed by other methods and hard to train
- 2006 –now: Deep Learning
  - Networks with many layers (deep) achieving near human performance on some tasks
  - Depend on huge data sets (Big Data) and require great processing power (mostly GPUs are used)
  - Geoffrey Hinton, Yoshua Bengio, Yann LeCunn (Canadian Institute for Advanced Research)

History

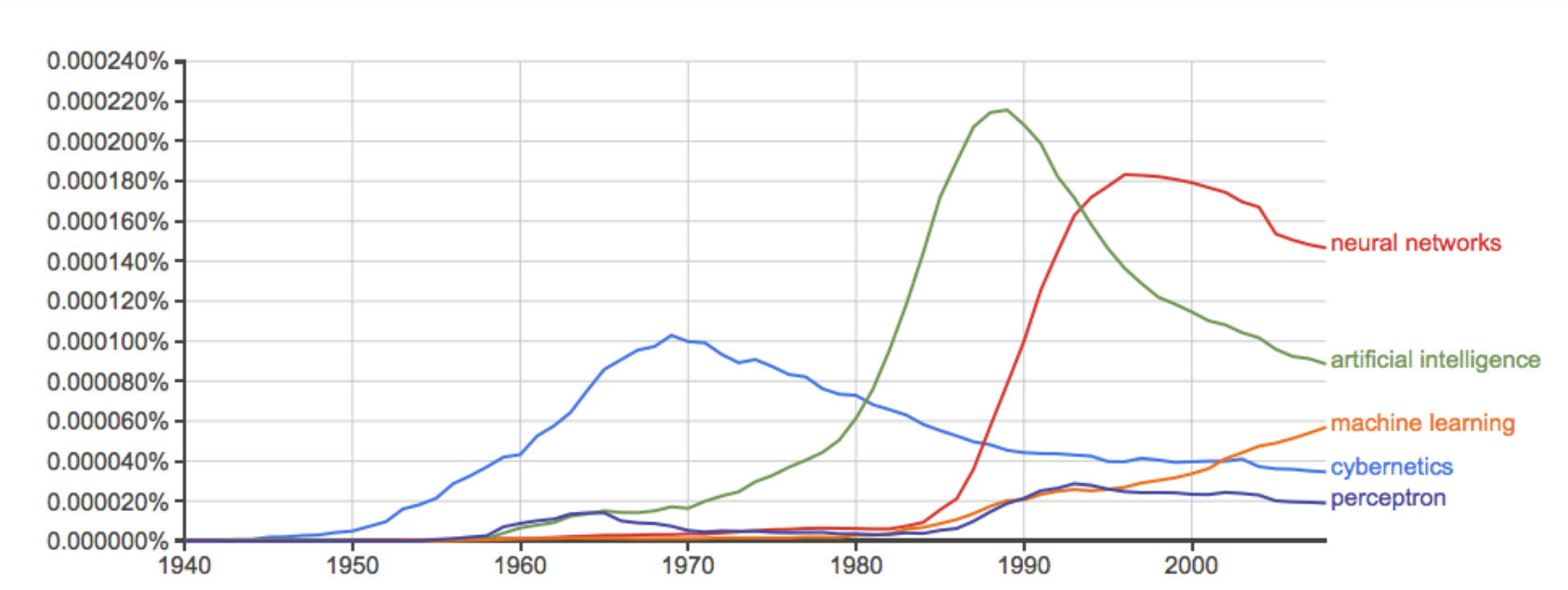


Figure 9: Phrases occurring in English books over time. No books on ”deep learning” or ”deep neural networks” in the books of the corpus. (generated with Google n-grams viewer `https://books.google.com/ngrams/`), data set ids 20120701 and 20090715

History

- Size of data sets increased dramatically over the years
- Some examples:

Data Set	Year	Number of Items
Iris Flower	1936	150
Cars	1990	1,728
MNIST Digits	1998	70,000
CIFAR10	2009	60,000
ImageNet	2009	14,197,122
Google News Texts	2013	3,000,000

- Deep Learning is only successful with Big Data (because deep networks have many parameters that need to be ”fixed”)
- Rule-of-thumb in 2016: supervised deep learning can do well with 5,000 items per category, can achieve near-human performance with 10,000,000 examples.

Biological Neural Networks

The Human Nervous System

- Life of multicellular organisms is steered by the nervous system
- The nervous system receives input from the environment (sensor signals) and creates output
- Sensor input examples
  - Smell
  - Vision
  - Audio signals
- Output
  - Behavior
  - Thoughts
  - Movements

The Human Nervous System

The human nervous system consists of two main parts

- Central nervous system (CNS)
  - Brain and spinal cord
  - Control center
- Peripheral nervous system (PNS)
  - Cranial and spinal nerves
  - Communication lines between body and CNS

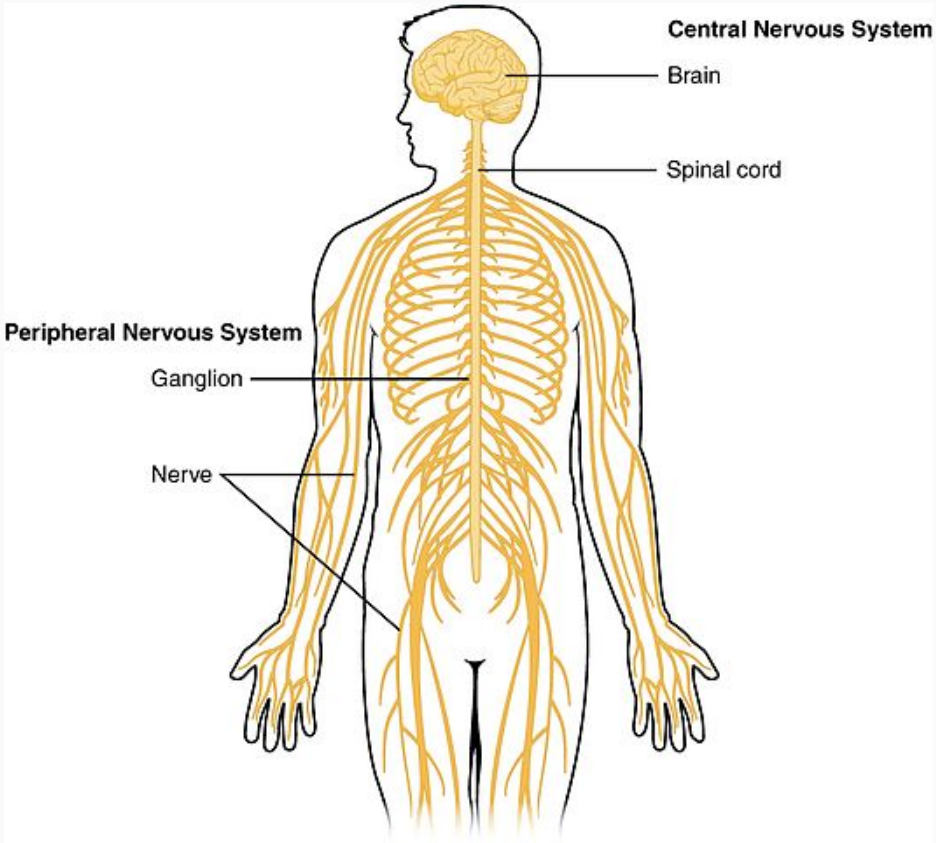


Figure 10: Human nervous system (CC-SA 4.0, OpenStax, via Wikimedia Commons)

The Human Nervous System

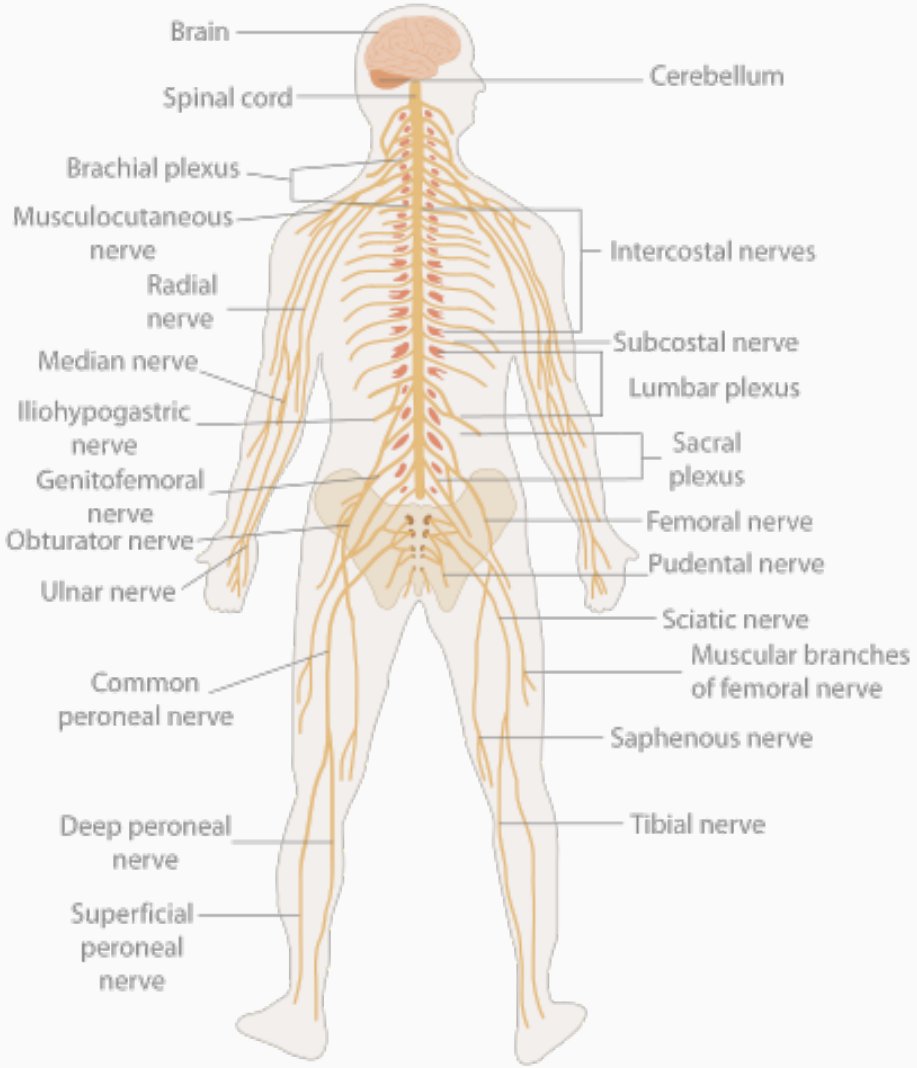


Figure 11: Human Nervous System (CC-SA 3.0, theEmirr, via Wikimedia Commons)

The Human Brain

Some statistics:

Weight	1.3 kg
Volume	1200 cm^3
Number of neurons	86 billion
Length of nerve fibres	5,8 million km
Main Areas	Cerebral cortex (Großhirn), Diencephalon (Zwischenhirn), Cerebellum (Kleinhirn), Brainstem

The Human Brain

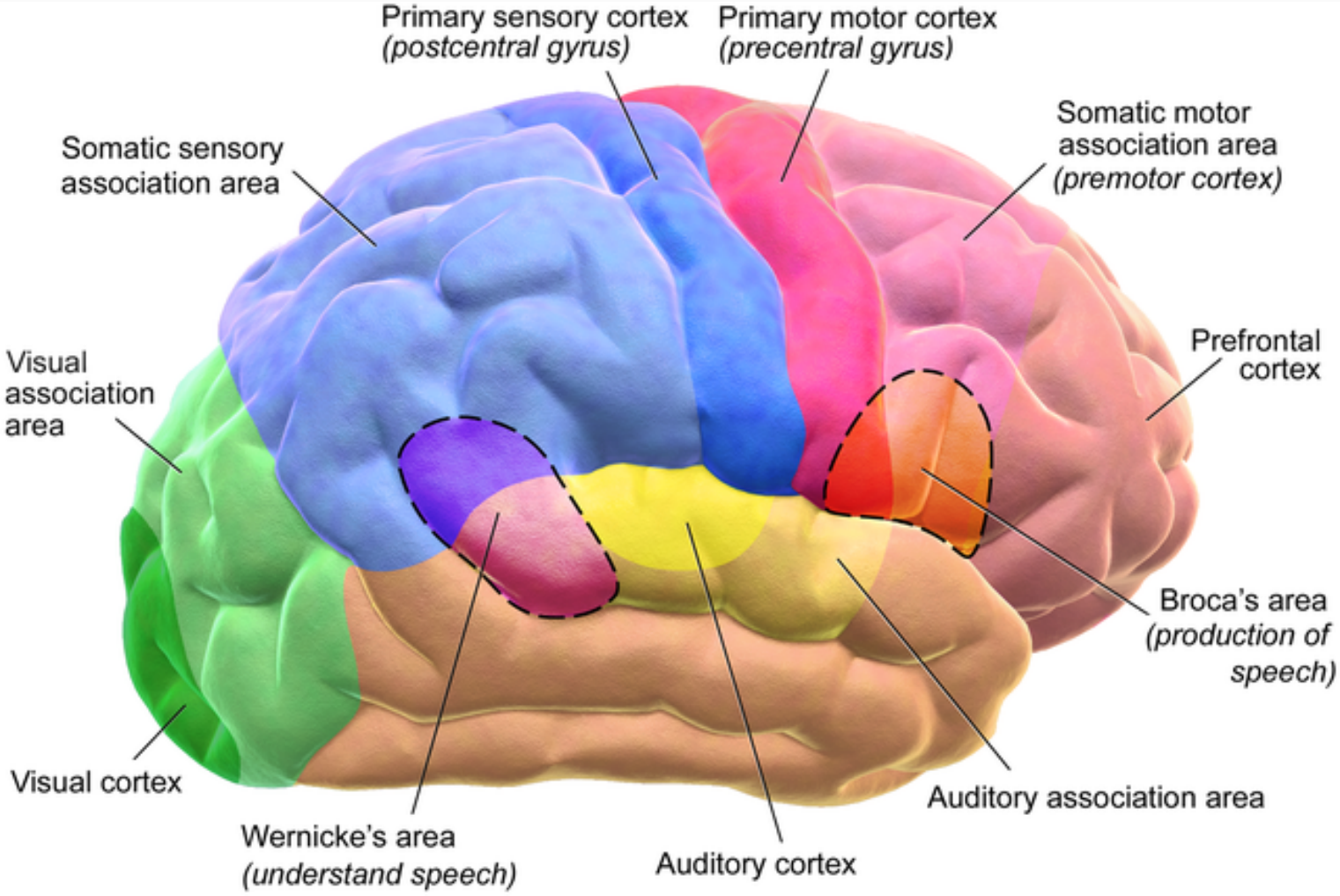


Figure 12: Functional areas (CC-BY 3.0, Blausen.com staff (2014). "Medical gallery of Blausen Medical 2014". WikiJournal of Medicine 1 (2). DOI:10.15347/wjm/2014.010)

Anatomy

- cells responsible for transmitting nerve impulses
- 3 types of neurons (sensory, motor and interneurons)

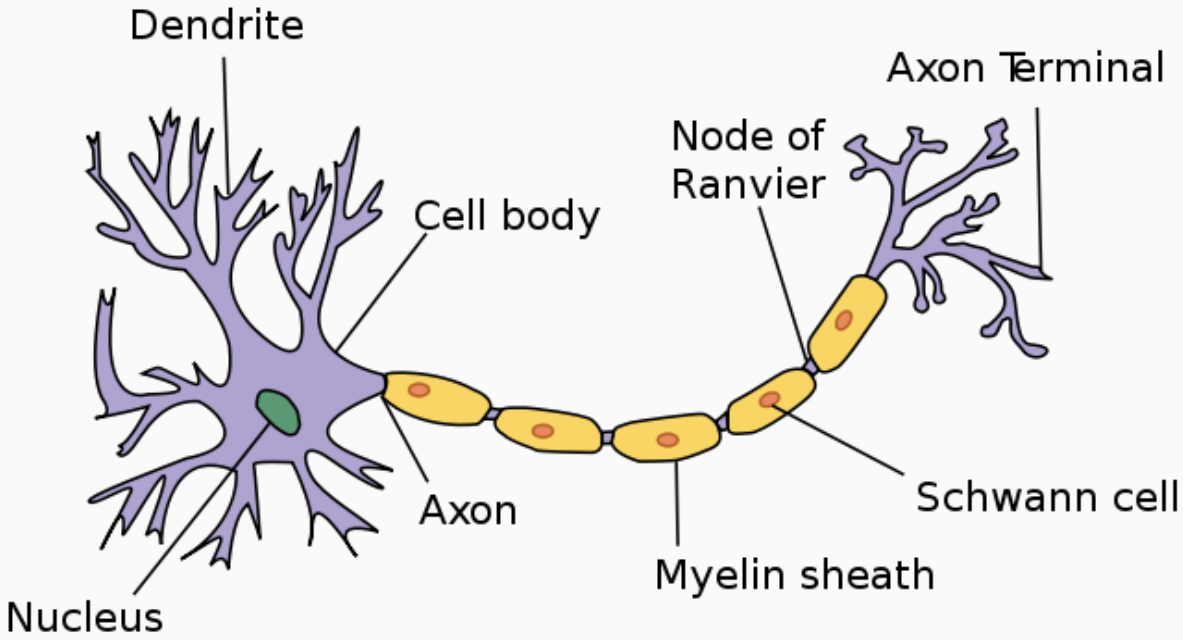


Figure 13: Anatomy of a neuron (CC-SA 3.0, Dhp1080 via Wikimedia Commons)

Neurons

Anatomy

- **Dendrite:** nerve endings for incoming signals, generally shorter than axons
- **Axon:** nerve ending for outgoing signal, can be up to 1 m in length; combination of Myelin sheath, Schwann cells and nodes of Ranvier support fast signal transmission across the length of the axon (the action potential ”jumps” from node to node)
- **Cell Body:** sums up the incoming signals and generates an outgoing signal if the aggregated incoming signal is above a certain threshold
- **Synapse:** Connection at the terminal end of axons or dentrites, transmits the signal from one neuron to another

Synapses

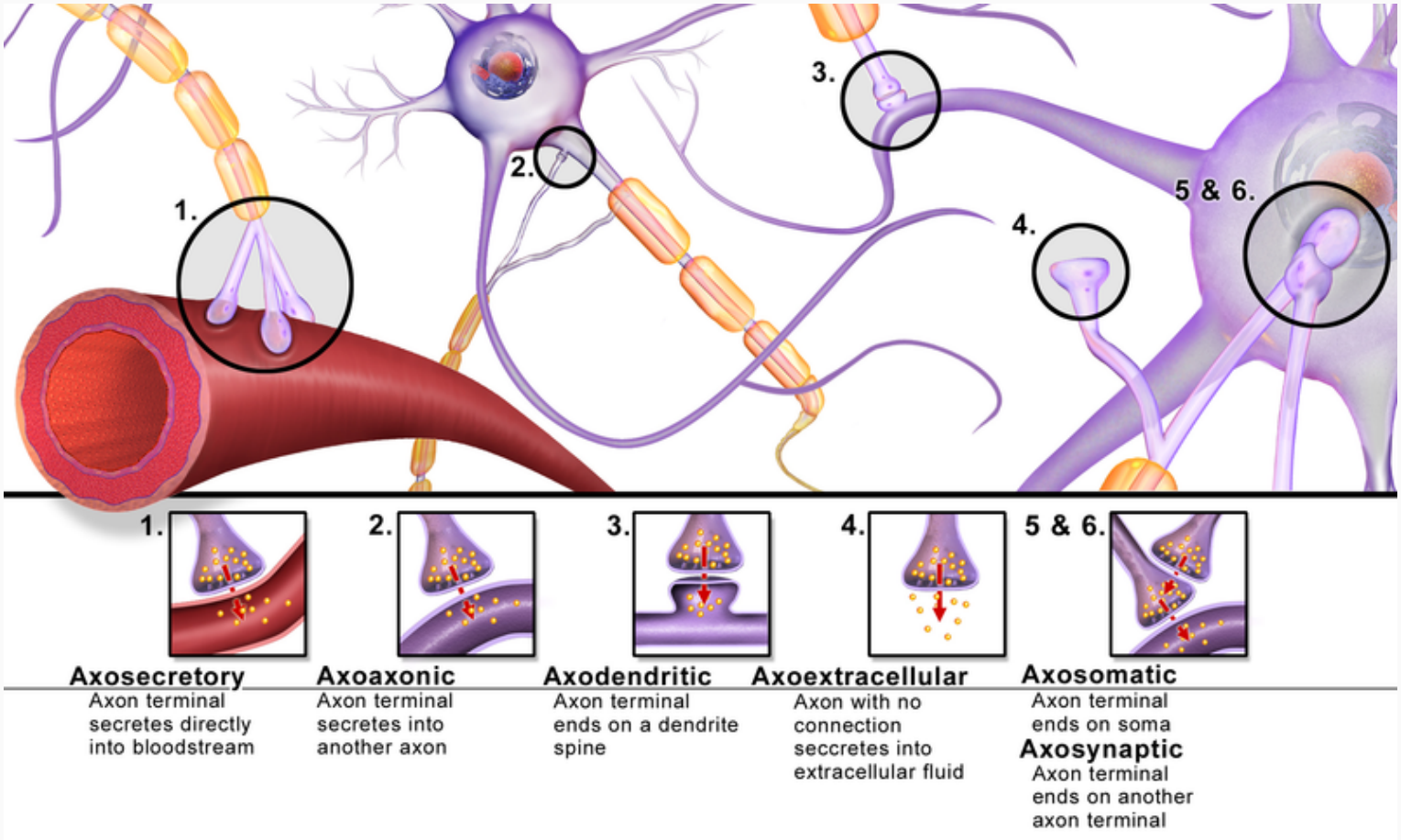


Figure 14: Types of synapses (CC-BY 3.0, Blausen.com staff (2014). "Medical gallery of Blausen Medical 2014". WikiJournal of Medicine 1 (2). DOI:10.15347/wjm/2014.010)

Signal Transmission

1. If a neuron gets excited it generates an electrical action potential in the cell body.
2. At a synaptic junction there is the *synaptic cleft* between the releasing and the receiving neuron.
3. The electrical signal is translated into a chemical signal. The synapse releases chemicals, so called neurotransmitters.
4. The neurotransmitters cross the synapic cleft and enter the receiving neuron through receptors.
5. The chemical signal is translated back to a electrical signal in the receiving neuron.
6. Neurotransmitters are transported back to the releasing neuron or degraded.

Explanatory video `https://en.wikipedia.org/wiki/File:Neuron_action_potential.webm` (10:00 min)

Post-Synaptic Signals

- An action potential is generated if the incoming signals are above a certain threshold.

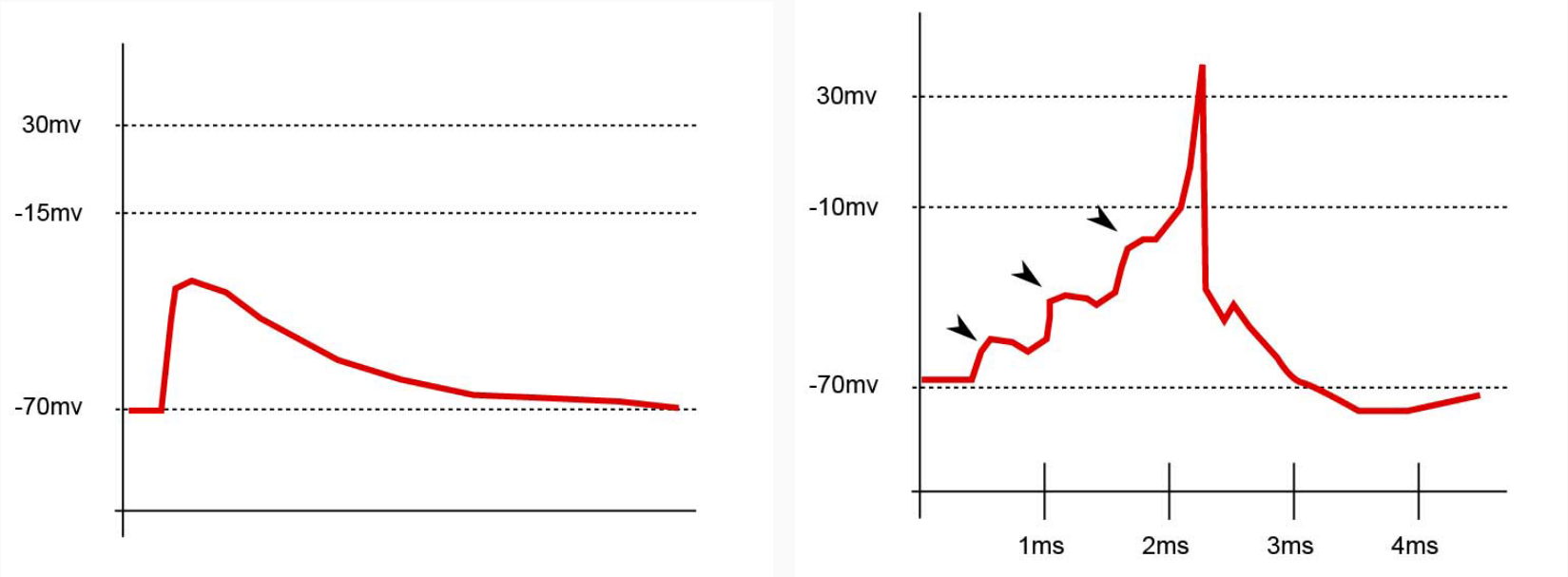


Figure 15: Incoming activation does not (left) and does create an action potential (right) (CC-SA 3.0, Dake via Wikimedia Commons)

Synaptic plasticity

- Synaptic strength (the strength of the signal transmitted over an active synapse) can vary over time.
- Changes in synaptic strength are the basis of learning and memory.
- Strength of the synapse can be altered by changing the number of released neurotransmitters, and the sensitivity of the receiving cell to those neurotransmitters.

Model of a neuron

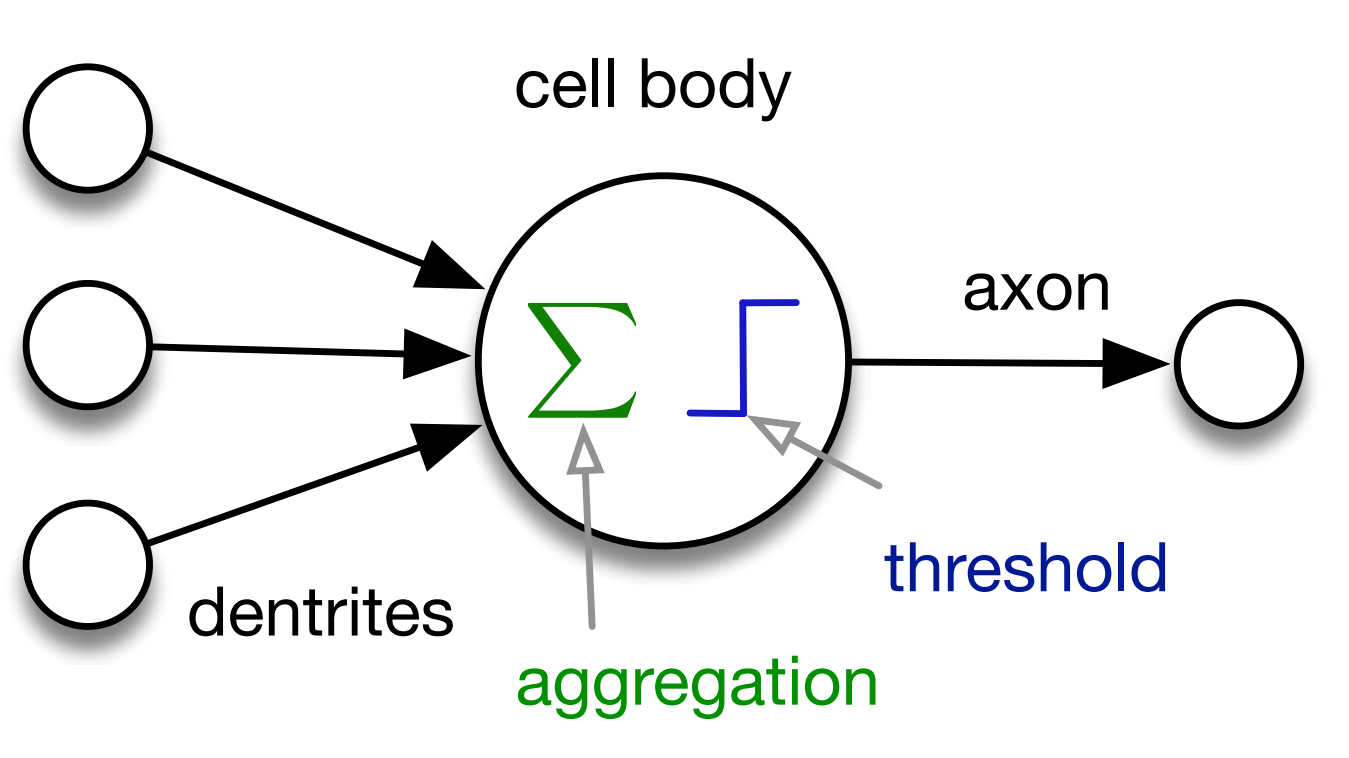


Figure 16: Schematic model of a neuron

Model of a network of neurons

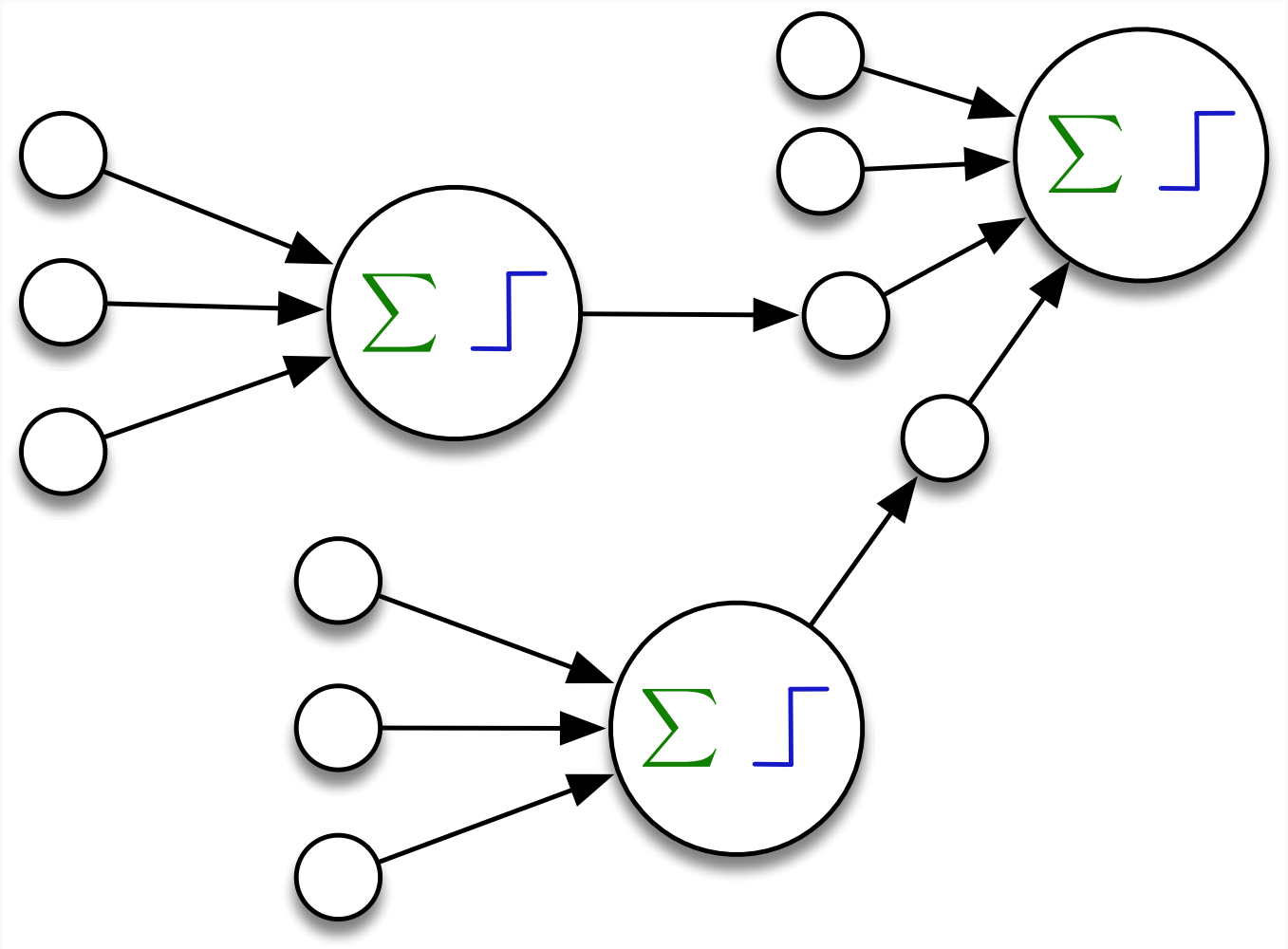


Figure 17: Schematic model of a network of neurons

The Human Brain

	Human Brain	Supercomputer 2011
Data Storage	3.5 quadrillion bytes	30 quadrillion bytes
Processing Speed	2.2 billion megaflops	8.2 billion megaflops
Power Consumption	20 watts	9.9 million watts
Storage Method	associative	address-based
Parallelisation	massively parallel	(mostly) serial

Notes:

- Numbers based on `https://www.scientificamerican.com/article/computers-vs-brains/`
- quadrillion 10^{15}, billion 10^{9}
- Ted Talk "What is so special about our brain?" `https://www.youtube.com/watch?v=_7_XH1CBzGw` (13:31 min)

Perceptron

Perceptron

The perceptron is the simplest neural network, with only one node that does computations.

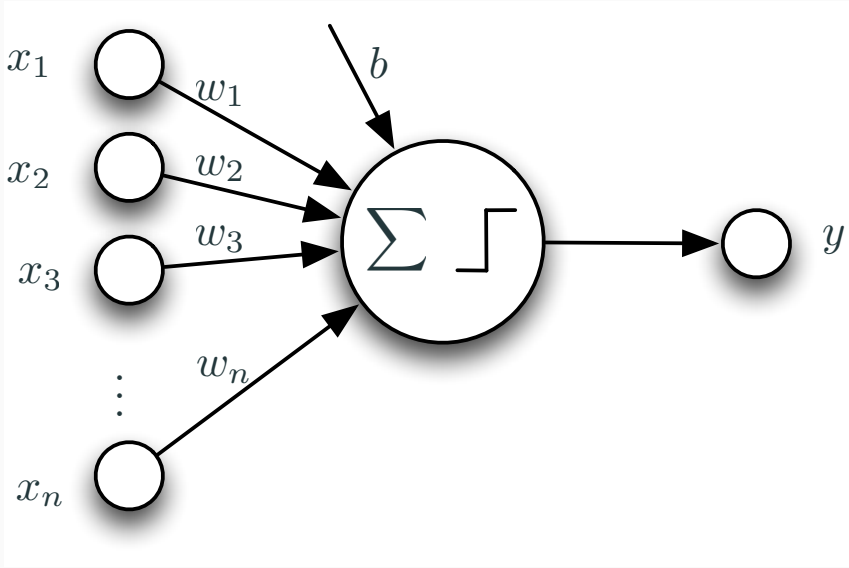


Figure 18: Perceptron

- $x = (x_1, \ldots, x_n)^T$ is the input vector
- $w = (w_1, \ldots, w_n)^T$ is the weight vector of the incoming edges
- b is a bias term
- $y \in \{0, 1\}$ is the output

Perceptron

The perceptron does the following computations

- First, it calculates the weighted sum s of the inputs

$$s = \sum_{i=1}^{n} w_i x_i + b$$

- If this sum is greater than zero, it outputs 1, otherwise 0

$y = h(s)$ with h being the heaviside function

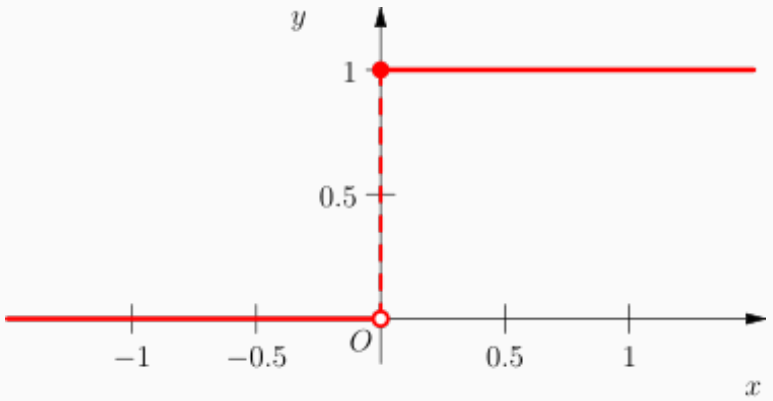
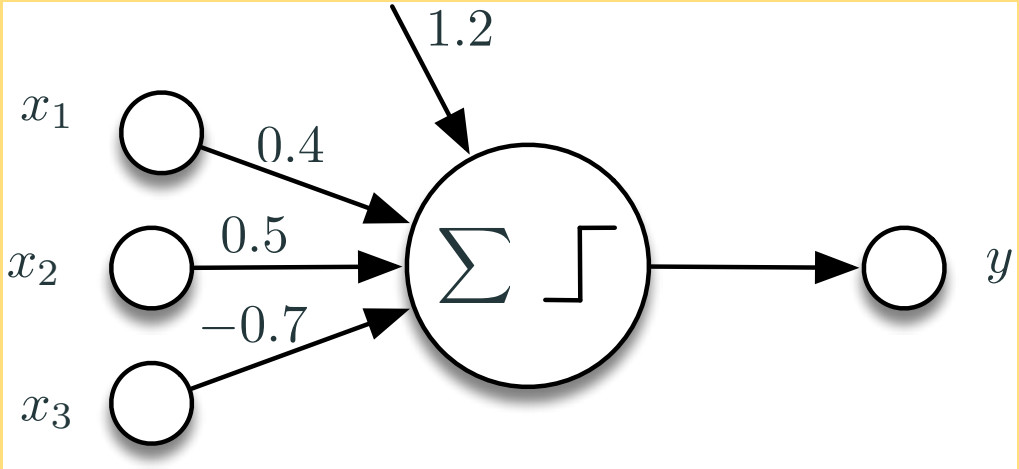


Figure 19: Heaviside function (Public Domain, via Wikimedia Commons)

Exercise

Calculate the output of the perceptron for the following input vectors:
$x^1 = (0, 0, 0)^T$, $x^2 = (-0.5, -1, 1)^T$



$y^1 = 1, y^2 = 0$

Perceptron

- Computation of perceptron

$$y = h(\sum_{i=1}^{n} w_i x_i + b)$$

- Which is equivalent to

$$y = h(w^T x + b)$$

Note:

- Sometimes the bias b is included in the weight vector as follows: The input vector is extended to $x = (x_0, x_1, \ldots, x_n)^T$ with $x_0 = 1$. The weight vector is then $w = (w_0, w_1, \ldots, w_n)^T$ with $w_0 = b$.
- This notation is equivalent and leads to a mathematically more compact notation. However, in terms of computation leaving out the bias is more efficient (in the future this means, less effort in terms of matrix multiplication).

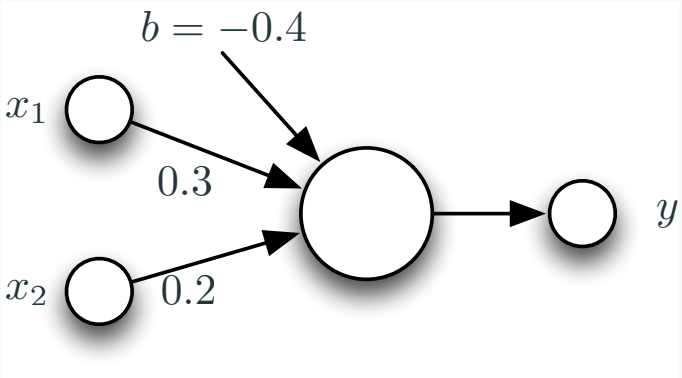
Exercise

Calculate $w^T x$ with $w = (0.4, 0.5, -0.7)^T$, $x = (-0.5, -1, 1)^T$

$w^T x = -1,4$

Perceptron – Example

- Perceptron with 2 binary input units.



- Represents the following function (logical AND)

x_1	x_2	s	y
0	0	-0.4	0
0	1	-0.2	0
1	0	-0.1	0
1	1	0.1	1

Perceptron – Limitations

- The computation function of a perceptron

$$y = h(w^T x + b)$$

  corresponds to a hyperplane in n-dimensional space
- $0 = w^T x + b = w_1 x_1 + w_2 x_2 + \ldots + b = 0$ is the equation for a hyperplane (e.g. $2x_1 + 3x_2 = 0$ in $\mathbb{R}^2$)
- Thus, a perceptron can learn to represent any hyperplane (by adapting the weights w).
- The perceptron is a linear classifier.

Perceptron – Limitations

- The perceptron is a linear classifier and can learn to separate linearly separable data.

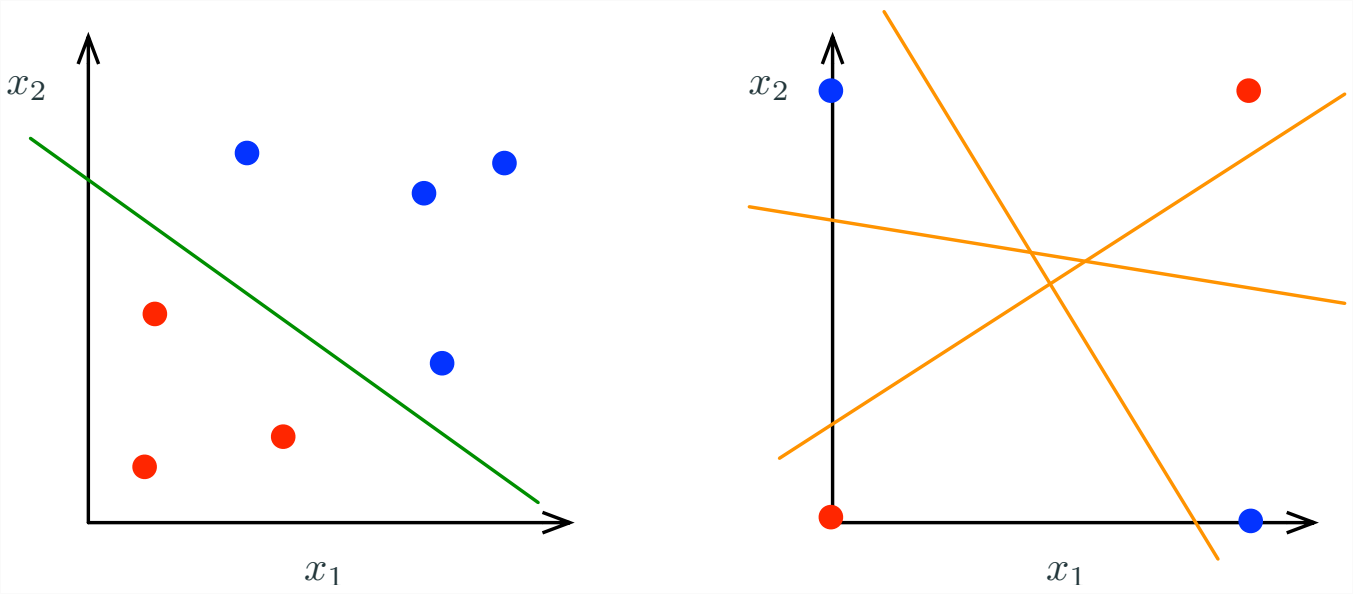


Figure 20: Linearly separable data (left) and inseparable data (right)

Perceptron – Limitations

- A perceptron can learn the logical functions AND, OR, NAND, NOR; but not XOR.
- Any logical function can be represented as a combination of AND, OR, NOT or NAND, or NOR.
- This means, any Boolean function can be learned by a neural network with at least 2 layers[1]
- This gives rise to the idea of using multi-layer neural networks for learning complex functions.

[1] Because every Boolean function can be represented in disjunctive normal form.

Perceptron – Training

Definition (Perceptron Training Rule)

For a given training example $(\mathbf{x}, y)$ the perceptron adapts its weights based on the following training rule:

$$error = y - h(\mathbf{w}^T \mathbf{x})$$
$$\Delta w_j = \eta \cdot error \cdot x_j$$
$$w_j \leftarrow w_j + \Delta w_j$$

with x_j being the j-th entry in the input feature vector w_j the weight of the j-th edge and η the learning rate.
In matrix form:

$$\mathbf{w} \leftarrow \mathbf{w} + \eta(y - h(\mathbf{w}^T \mathbf{x} + b))\mathbf{x}$$

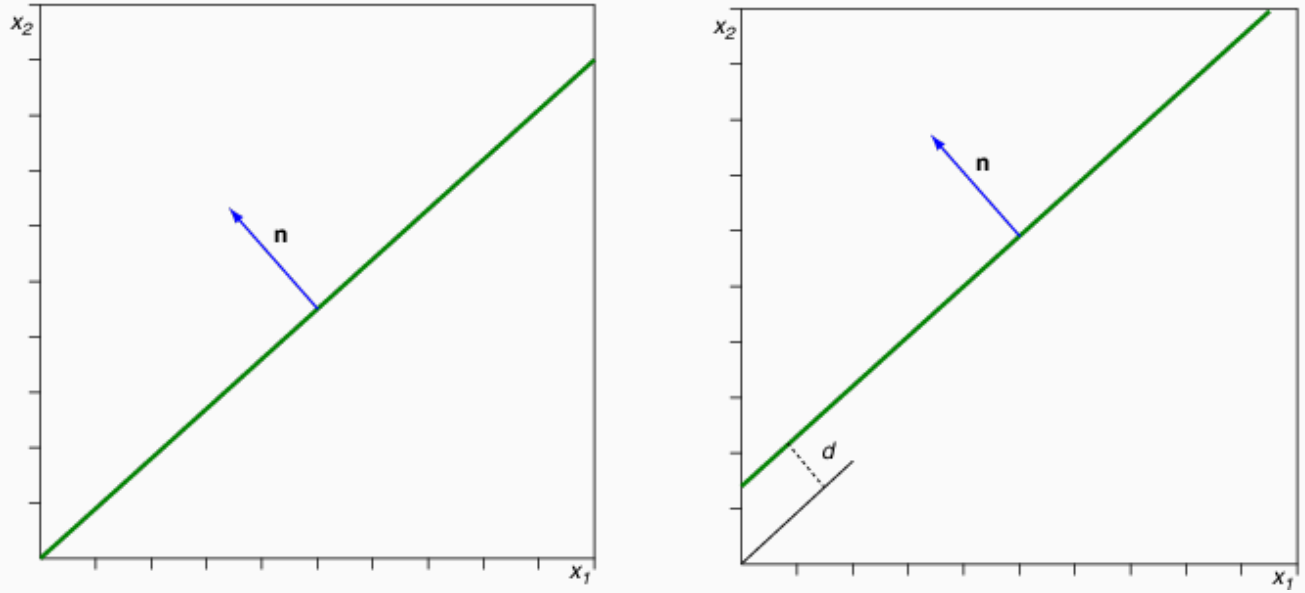
Perceptron – Training

Algorithm:	PT	Perceptron Training
Input:	D	Training examples of the form $(\mathbf{x}, y)$, $y \in \{0, 1\}$.
	η	Learning rate, a small positive constant.
Output:	$\mathbf{w}$	Weight vector.

$PT(D, \eta)$

1. *initialize_random_weights*($\mathbf{w}$), $t = 0$
2. **REPEAT**
3. $t = t + 1$
4. $(\mathbf{x}, y) = random\_select(D)$
5. $error = y - h(\mathbf{w}^T\mathbf{x} + b)$
6. **FOR** $j = 0$ **TO** p **DO**
7. $\Delta w_j = \eta \cdot error \cdot x_j$
8. $w_j = w_j + \Delta w_j$
9. **ENDDO**
10. **UNTIL**(*convergence*() **OR** $t > t_{\max}$)
11. *return*($\mathbf{w}$)

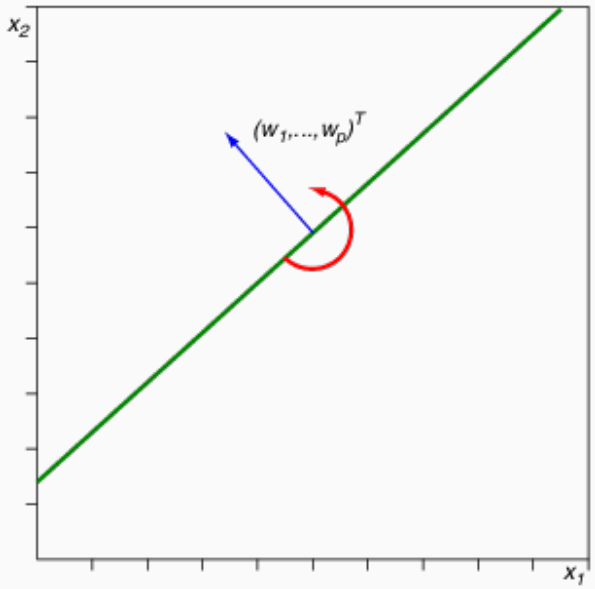
Perceptron – Training



Definition of an (affine) hyperplane: $\mathbf{n}^T\mathbf{x} = d$.

- $\mathbf{n}$ denotes a normal vector that is perpendicular to the hyperplane.
- If $||\mathbf{n}|| = 1$ then $|d|$ corresponds to the distance of the origin to the hyperplane.
- If $\mathbf{n}^T\mathbf{x} < d$ and $d \geq 0$ then $\mathbf{x}$ and the origin lie on the same side of the hyperplane.

Perceptron – Training



Definition of an (affine) hyperplane: $\mathbf{w}^T\mathbf{x} + b = 0 \Leftrightarrow \sum_{j=1}^{n} w_j x_j = -b.$

Perceptron – Training

- A perceptron defines a hyperplane that is perpendicular (= normal) to $(w_1, \ldots, w_n)^T$.
- $-b$ specify the offset of the hyperplane from the origin, along $(w_1, \ldots, w_n)^T$.
- The set of possible weight vectors $\mathbf{w} = (w_0, w_1, \ldots, w_n)^T$ form the hypothesis space H (that is everything the perceptron can learn).
- Weight adaptation means learning, and the shown learning paradigm is supervised.

Perceptron – Training

- The error for one training sample can either be 0, 1 or -1.

y	$h(\mathbf{w}^T\mathbf{x} + b)$	**error**
0	0	0
0	1	-1
1	0	1
1	1	0

- The computation of the weight difference Δw_j in Line 7 of the perceptron training algorithm (slide 38) considers a feature vector $\mathbf{x}$ component-wise. In particular, if some x_j is zero, Δw_j will be zero as well.

Perceptron – Training

- Weight update for example that is classified as 0, but should be 1.
- The hyperplane is rotated towards the example.

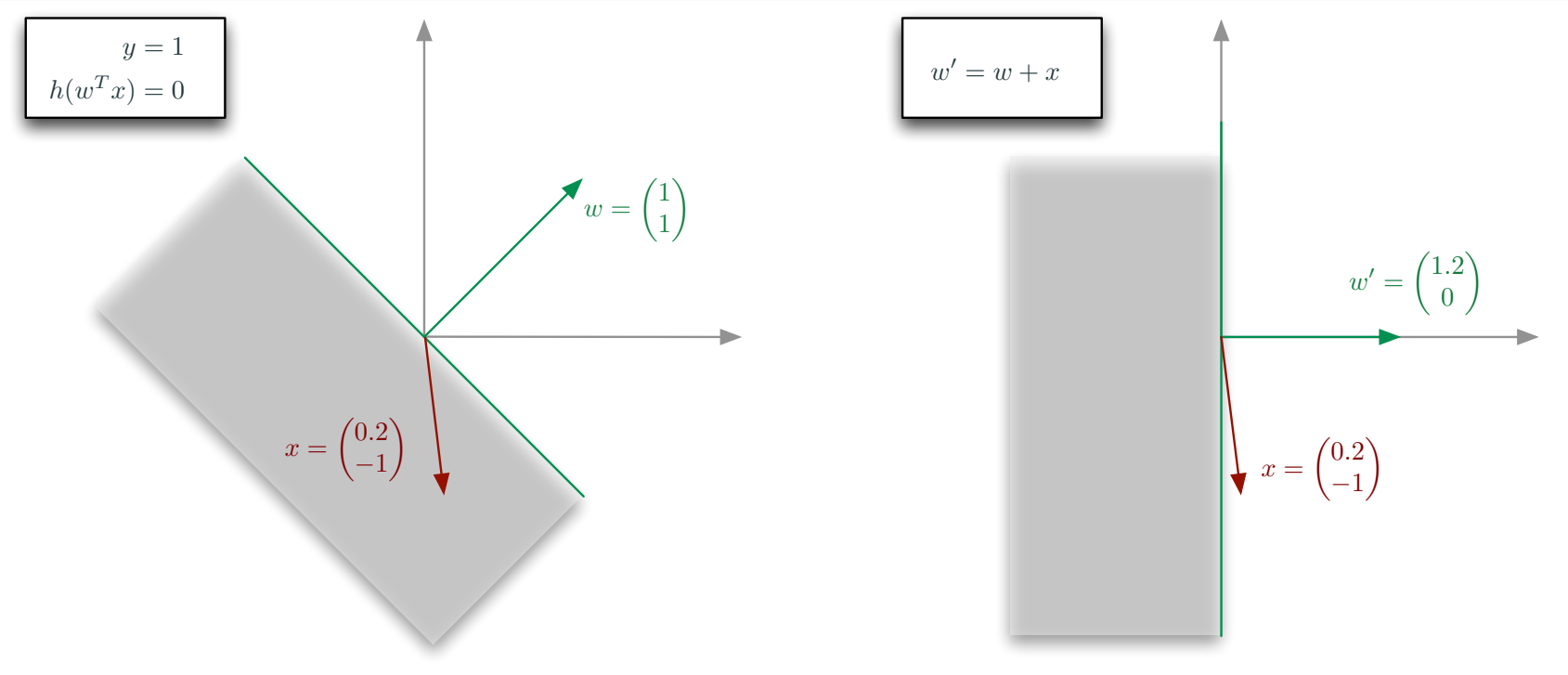


Figure 21: Perceptron weight update schema, case 1

Perceptron – Training

- Weight update for example that is classified as 1, but should be 0.
- The hyperplane is rotated away from the example.

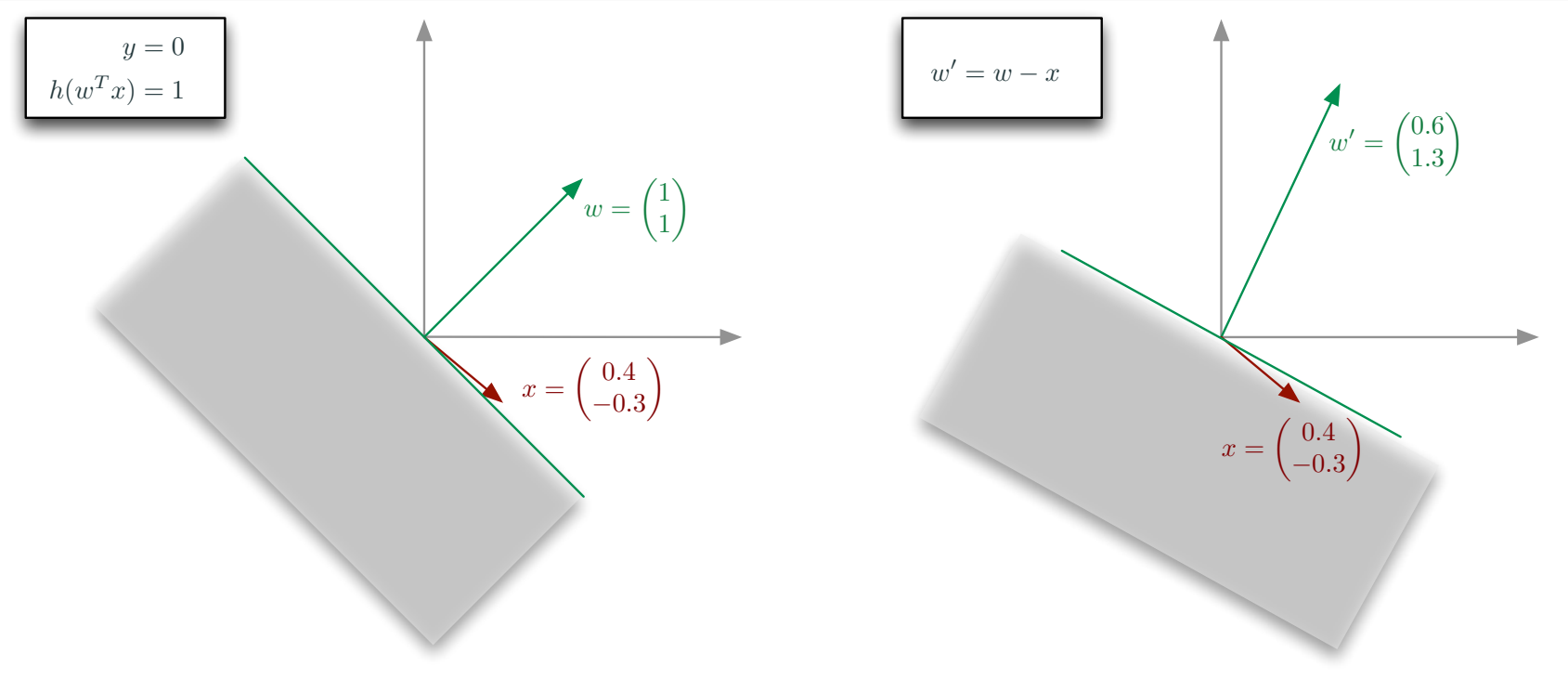


Figure 22: Perceptron weight update schema, case 2

Example

- Example images are presented to the perceptron.
- The perceptron has to decide whether the image contains the letter A or B.

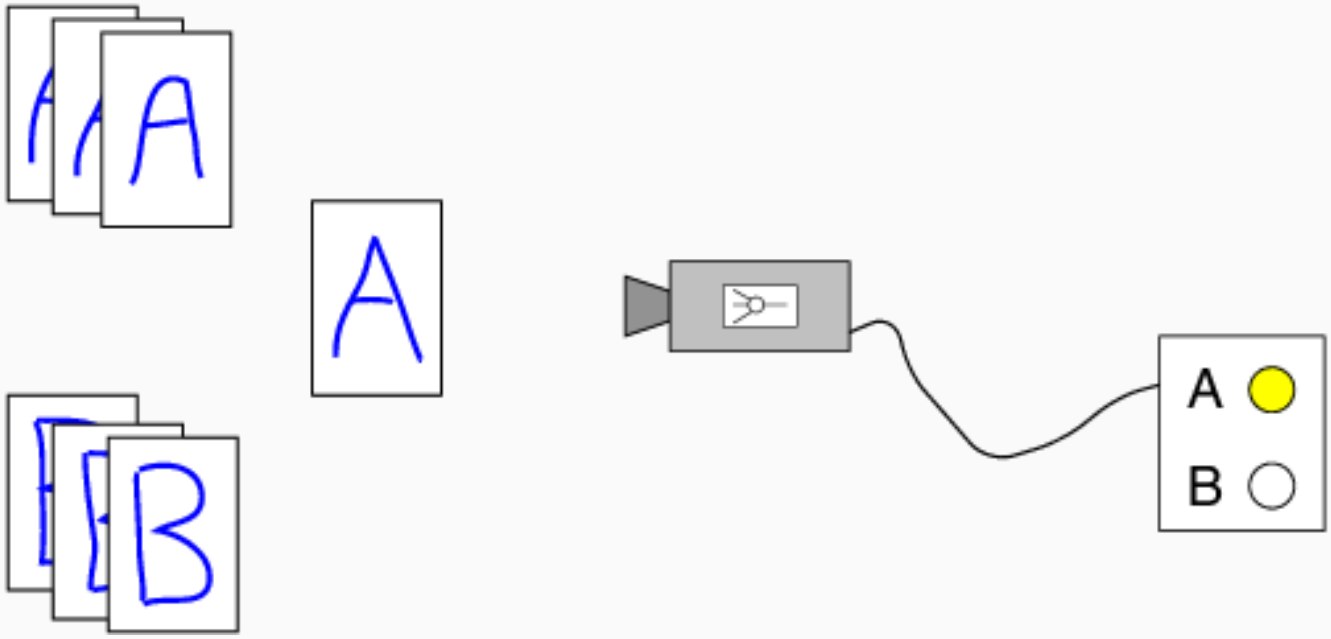


Figure 23: Simple classification task

Perceptron – Training

- The encoding of the examples is based on features: number of line crossings, most acute angle, longest line, etc.
- The class label, y, is encoded as a number. Examples from A are labeled with 1, examples from B are labeled with 0.

$$\underbrace{\begin{pmatrix} x_{1_1} \\ x_{1_2} \\ \vdots \\ x_{1_n} \end{pmatrix} \cdots \begin{pmatrix} x_{k_1} \\ x_{k_2} \\ \vdots \\ x_{k_n} \end{pmatrix}}_{\text{Class } A \;\simeq\; y = 1} \qquad \underbrace{\begin{pmatrix} x_{l_1} \\ x_{l_2} \\ \vdots \\ x_{l_n} \end{pmatrix} \cdots \begin{pmatrix} x_{m_1} \\ x_{m_2} \\ \vdots \\ x_{m_n} \end{pmatrix}}_{\text{Class } B \;\simeq\; y = 0}$$

Perceptron – Training

- Initial configuration of items in feature space (projected to 2D)

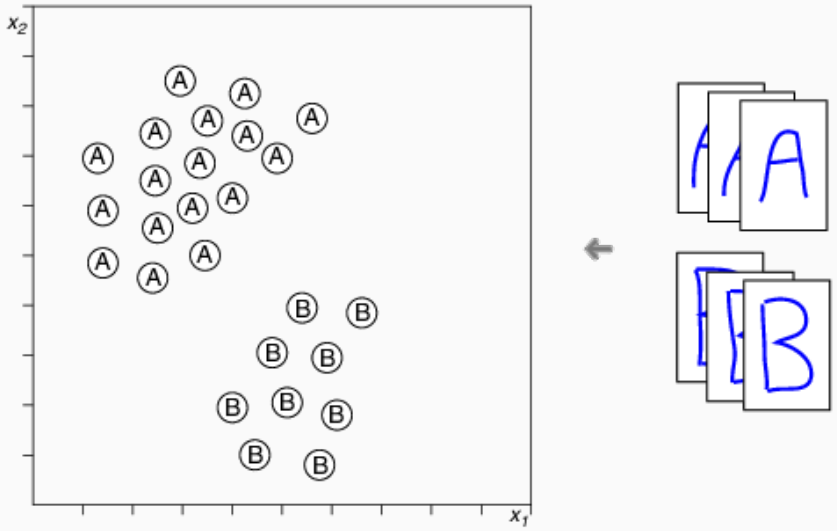


Figure 24: Feature space

Perceptron – Training

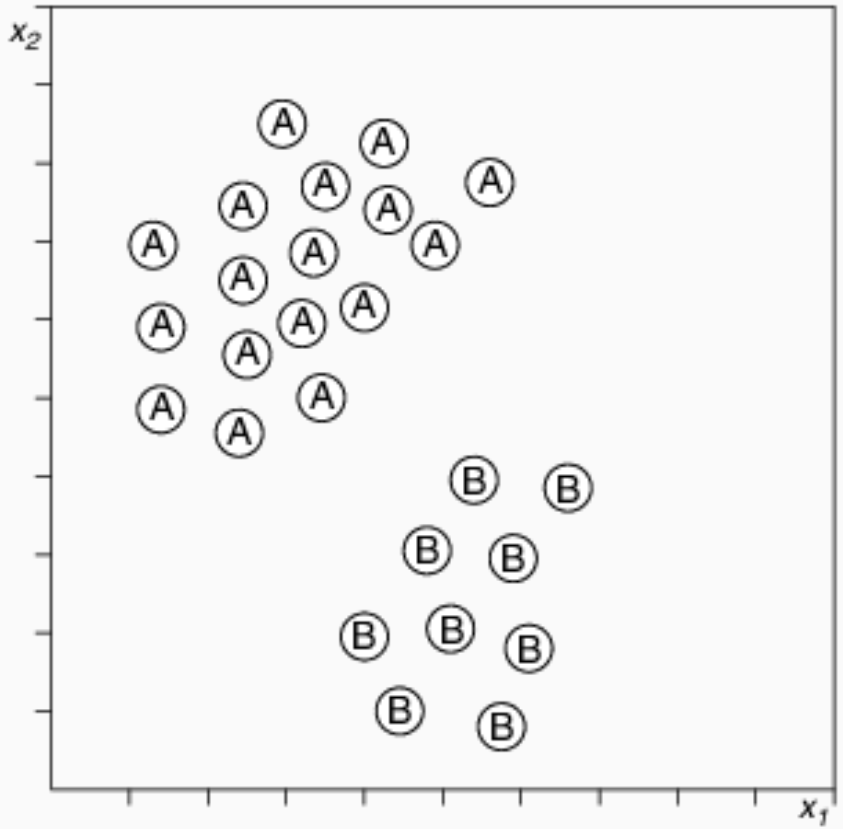


Figure 25: Feature space

Perceptron – Training

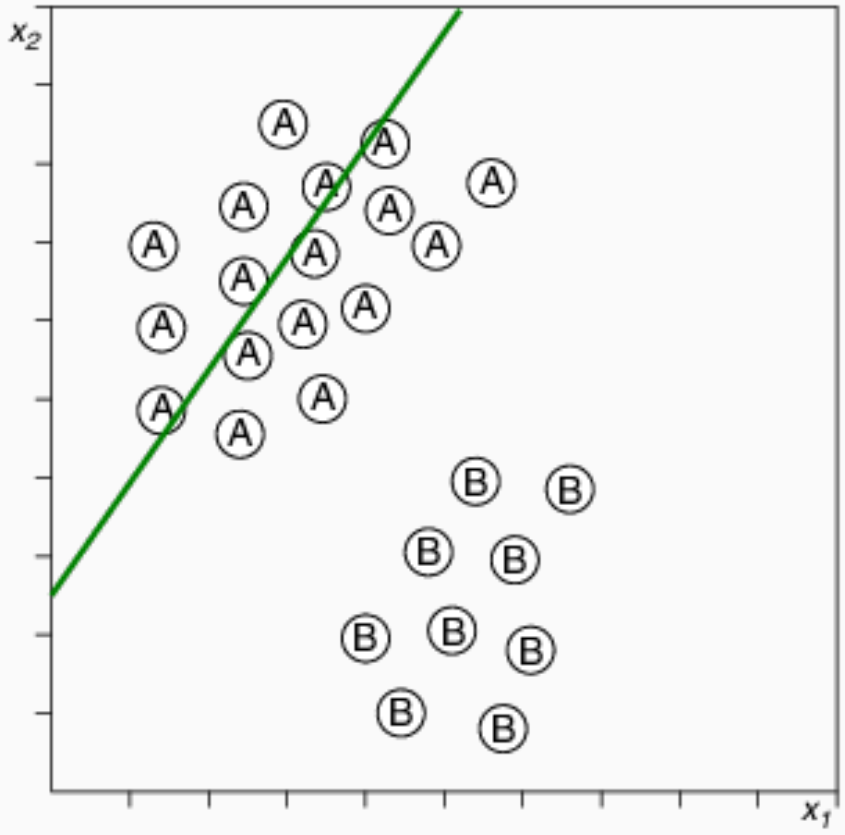


Figure 26: Feature space

- Initial (random) hyperplane

Perceptron – Training

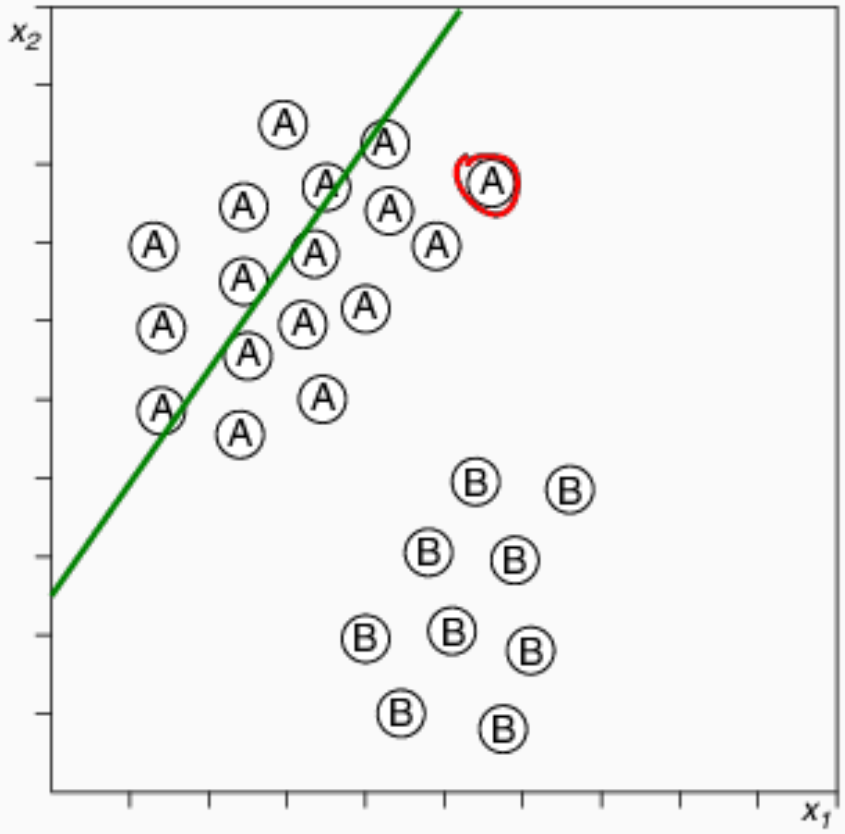


Figure 27: Feature space

- Item is classified as 0 (B), but should be 1 (A)

Perceptron – Training

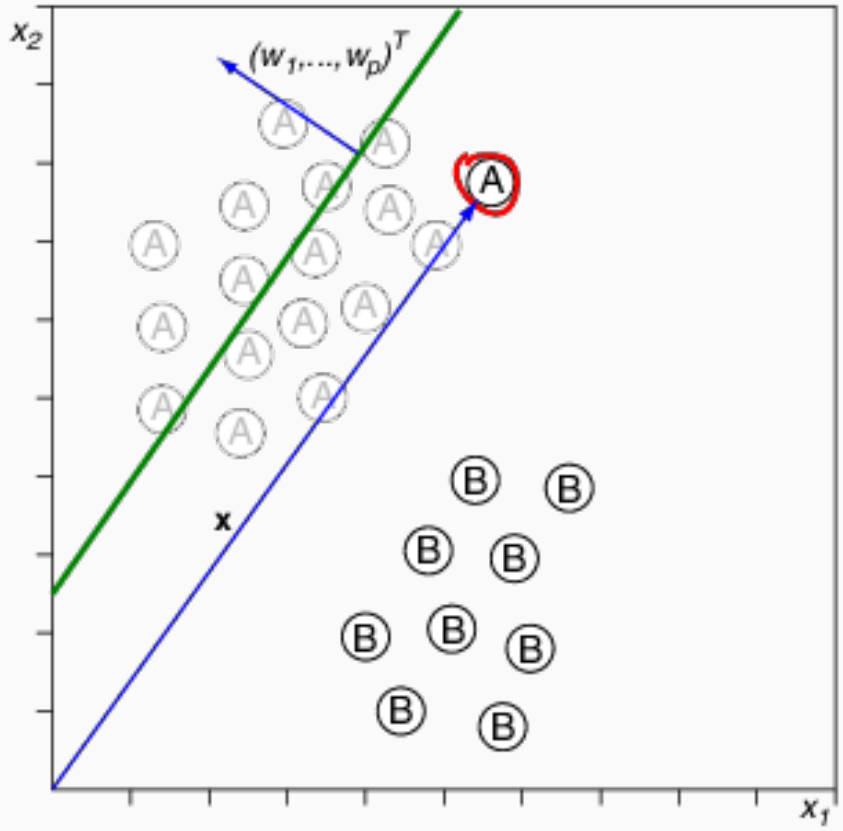


Figure 28: Feature space

- Item is classified as 0 (B), but should be 1 (A) (it lies not in the direction of the normal vector)

Perceptron – Training

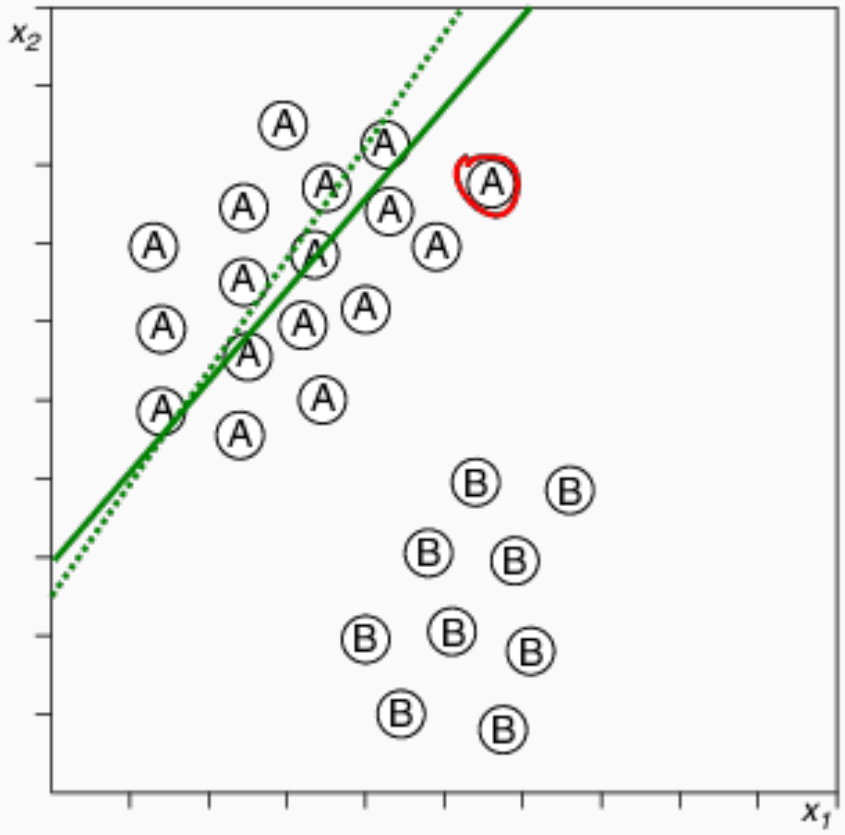


Figure 29: Feature space

- Hyperplane is rotated towards the example (dashed: before weight update, solid line: after weight update)

Perceptron – Training

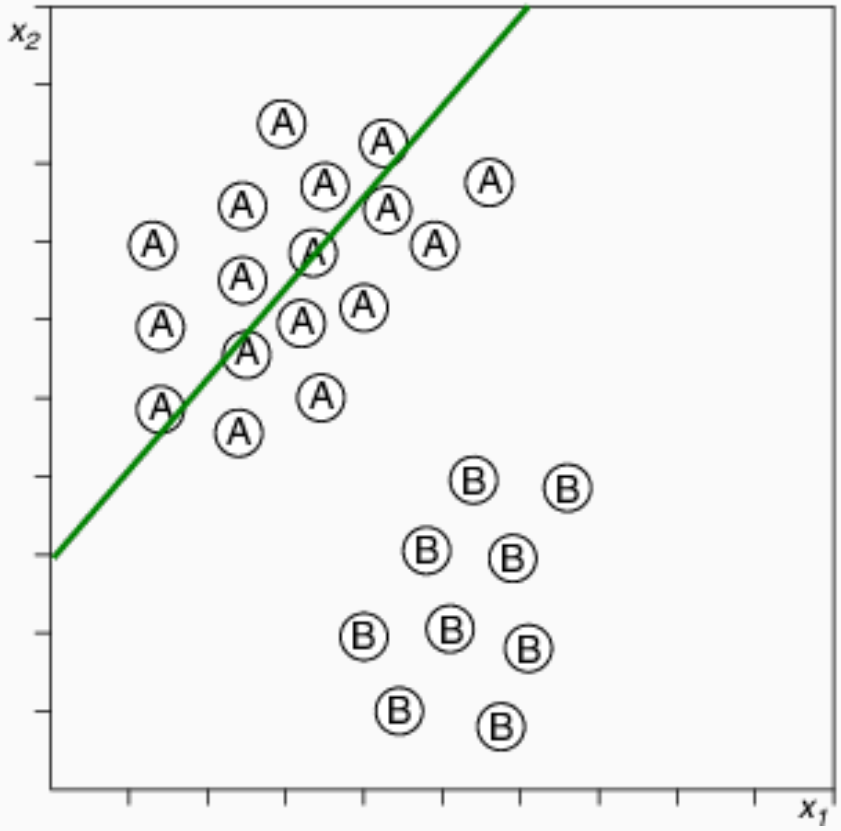


Figure 30: Feature space

- Classifier after training one example

Perceptron – Training

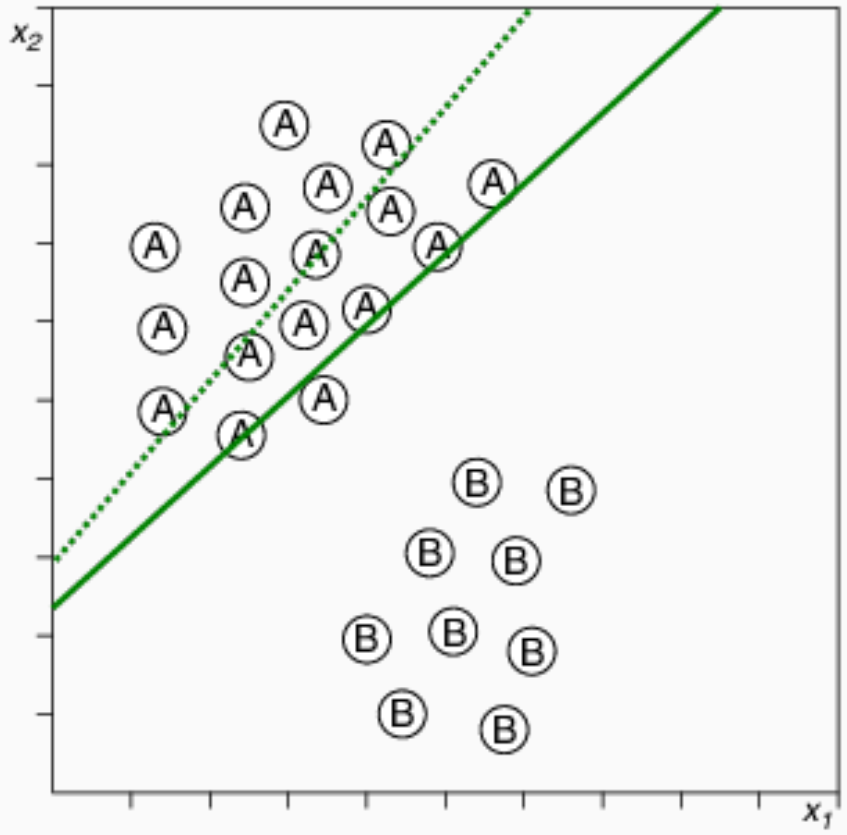


Figure 31: Feature space

- Next weight udpate

Perceptron – Training

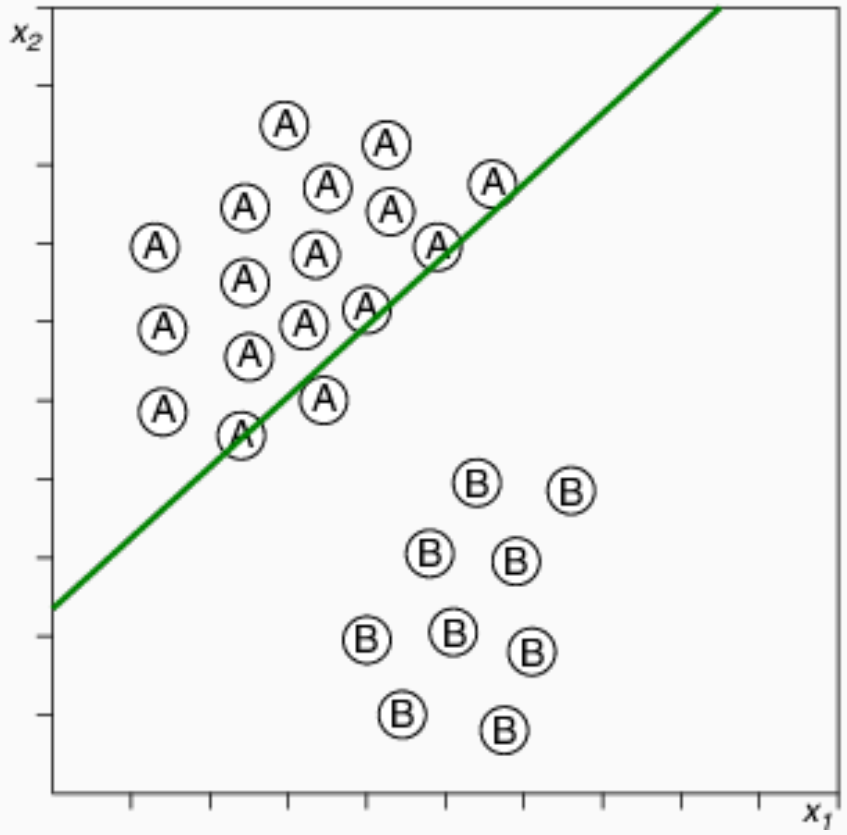


Figure 32: Feature space

- Classifier after two training examples

Perceptron – Training

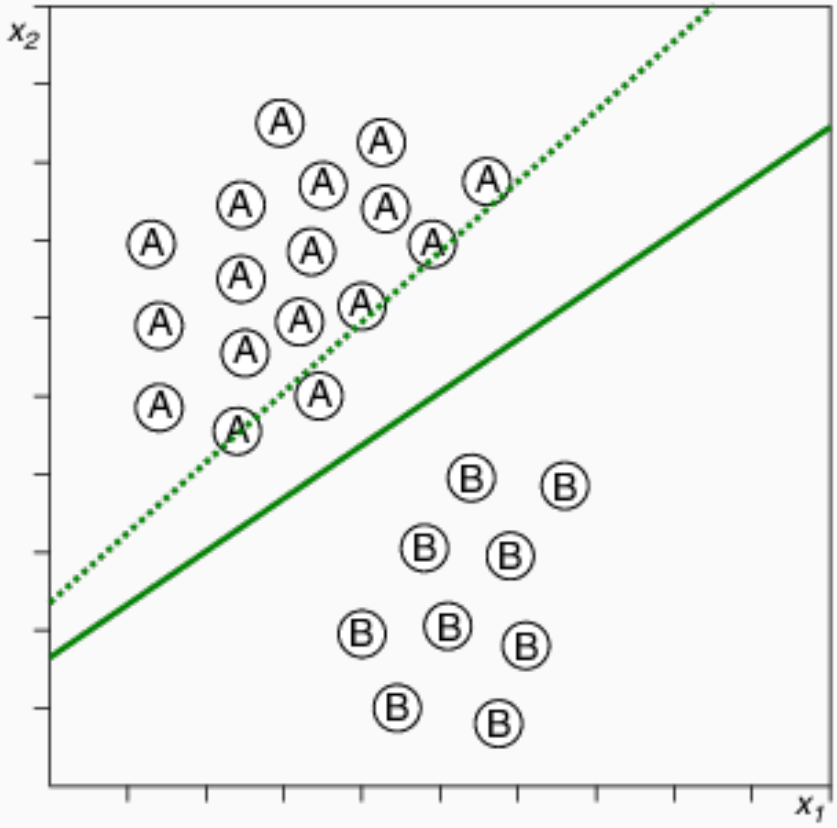


Figure 33: Feature space

- Next weight update leads to zero errors

Perceptron Convergence Theorem

Theorem (Perceptron Convergence)

Let X_0 and X_1 be two finite sets with vectors of the form $\mathbf{x} = (x_1, \ldots, x_n)^T$, let $X_1 \cap X_0 = \emptyset$, and let $\widehat{\mathbf{w}}$ define a separating hyperplane with respect to X_0 and X_1. Moreover, let D be a set of examples of the form $(\mathbf{x}, 0)$, $\mathbf{x} \in X_0$ and $(\mathbf{x}, 1)$, $\mathbf{x} \in X_1$. Then the following holds:

If the examples in D are processed with the perceptron training algorithm (cf. slide 38) the underlying weight vector $\mathbf{w}$ will converge within a finite number of iterations.

- If a separating hyperplane exists, i.e., the data is linearly separable, the perceptron training algorithm converges.

Perceptron Convergence Theorem

- The perceptron algorithm will not converge if a separating hyperplane does not exist.

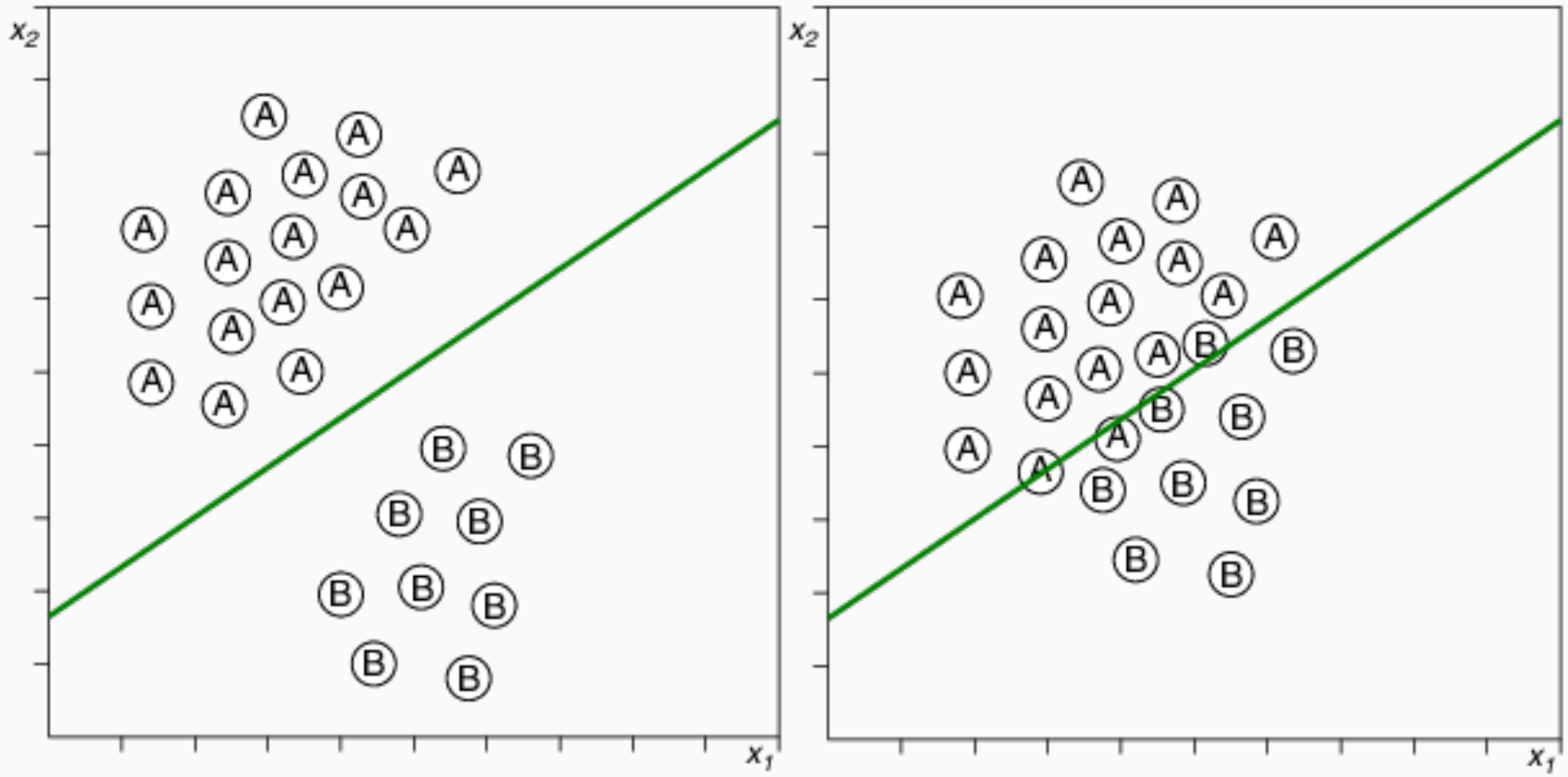


Figure 34: Linearly separable data left vs. linearly not separable data (right).

Machine Learning Basics

Machine Learning

Definition (Machine Learning)

A computer program is said to learn

- from experience
- with respect to some class of tasks and
- a performance measure,

if its performance at the tasks improves with the experience [7].

Examples:

- Robot navigation
  - Task: navigate its way
  - Performance measures: Length of the route / number of times the robot reached its goal (without the battery being empty before / number of accidents (wall bumps) / ..
  - Experience: navigations through training mazes
- Credit card fraud detection
  - Task: recognize fraudulent transactions
  - Performance measures: number of fraudulent transactions recognized / number of transactions correctly recognized / money saved
  - Experience: history of transactions with annotation whether they were fraudulent or not

Machine Learning Paradigms

Three basic types of machine learning algorithms, dependent on the type of feedback (experience) the learner receives.

- **Supervised Learning**
  Learner receives the desired output for an input from a ”teacher”.
- **Unsupervised Learning**
  Learner only has the input data and aims to detect patterns in this data.
- **Reinforcement Learning**
  Learner takes action in an environment and receives a reward. Reward might come only at the end of a very long sequence of actions and might also be very simple, such as +1 (success) and -1 (failure).

A machine learning system may also use combinations of paradigms (e.g. semi-supervised learning).

Machine Learning Paradigms

Examples:

- Supervised Learning
  - Optical character recognition
  - Credit card fraud detection
  - Object recognition (ReCaptchas are used for getting training data)
- Unsupervised Learning
  - Anomaly detection
  - Customer segmentation (special case of clustering)
- Reinforcement Learning
  - Robot navigation
  - Robot walk
  - Chess

Supervised Machine Learning

Supervised learning is further distinguished depending on the type of output variable.

- **Classification**
  The output of the learner (and the desired output presented by the teacher) is a *set of categories*.
  E.g. recognize hand-written digits. Categories are $0, 1, \ldots, 9$.
- **Regression** The output of the learner is a real-valued number.
  E.g. predict the prize of a house.

Taxonomy of Machine Learning

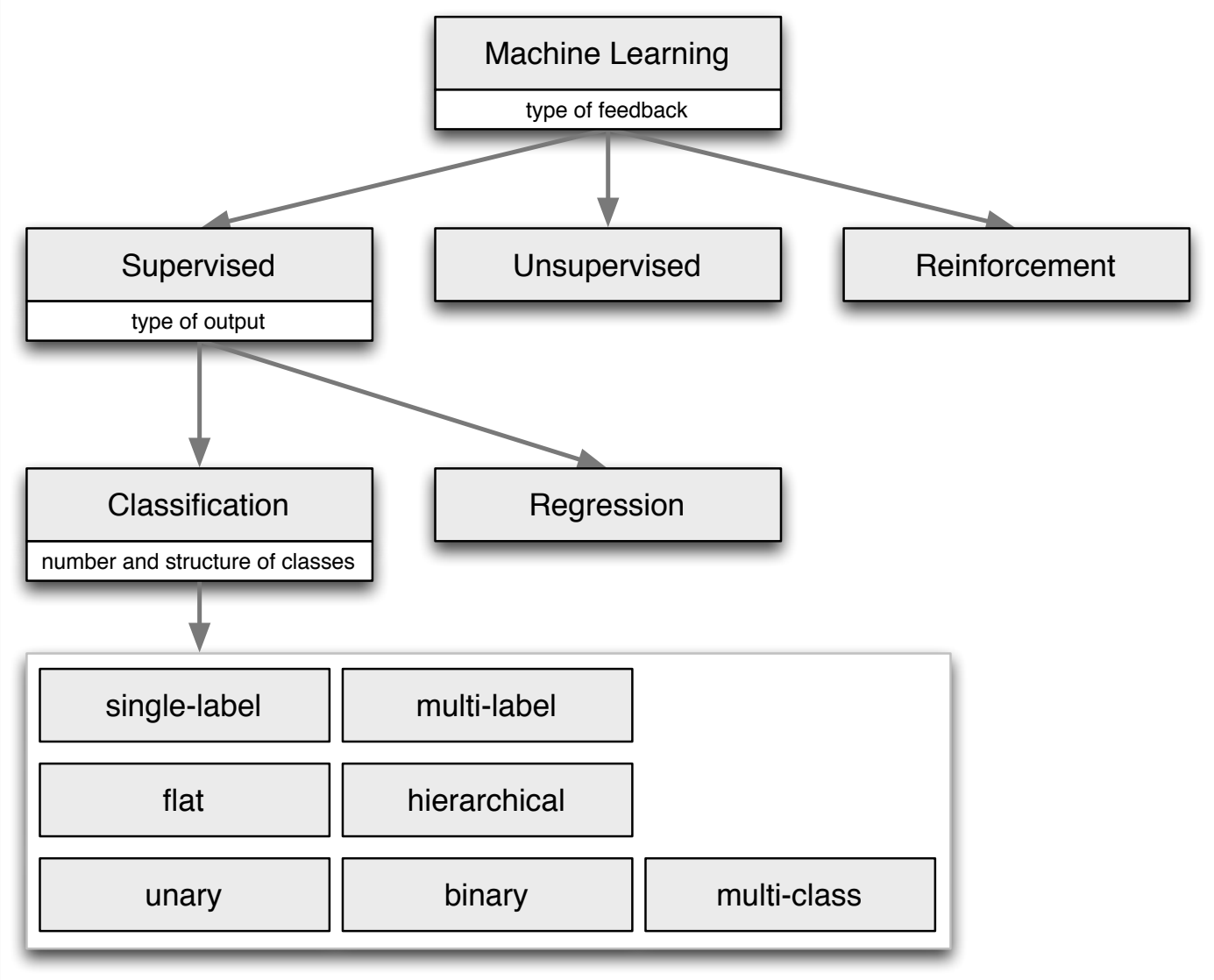


Figure 35: Taxonomy of machine learning algorithms (simplified)

Notation for Supervised Learning

Notation

- n_x – input feature vector size
- n_y – output size (number of classes)
- $x = (x_1, \ldots, x_{n_x})$ – input vector
- y – desired output (target)
- $\hat{y}$ – predicted output (what the learner produces)
- $(x^{(1)}, y^{(1)})$ – training item (first example from the training data)
- m – number of examples in the data set
- $X \in \mathcal{R}^{n_x \times m}$ – input matrix (each item is a column, each feature is a row)

Notation for Supervised Learning

Example hand-written digit classification[2]

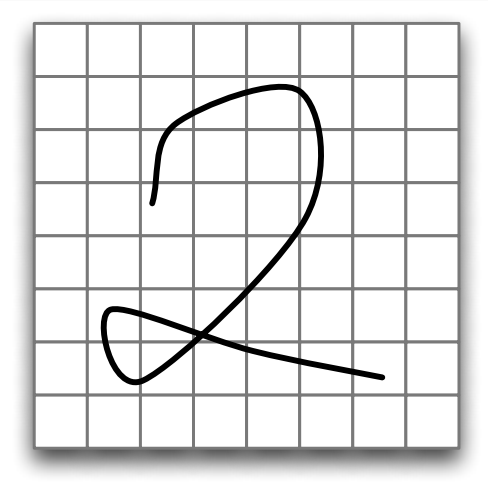


Figure 36: Example hand-written digit

- $n_x = 64$ (8x8 pixels)
- $n_y = 10$ (numbers from 0 to 9)
- $x^{(1)} = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, \ldots)^T$ (each pixel is either not-filled – 0 or filled – 1. Rows of the image are concatenated to form the feature vector)
- $y^{(1)} = 2$
- $X = (x^{(1)}, x^{(2)}, \ldots, x^{(m)})$

$$X = \begin{pmatrix} x_1^1 & \cdots & x_1^m \\ \vdots & & \vdots \\ x_{n_x}^1 & \cdots & x_{n_x}^m \end{pmatrix}$$

[2]Note, that many more feature representations are possible

Classification Pipeline

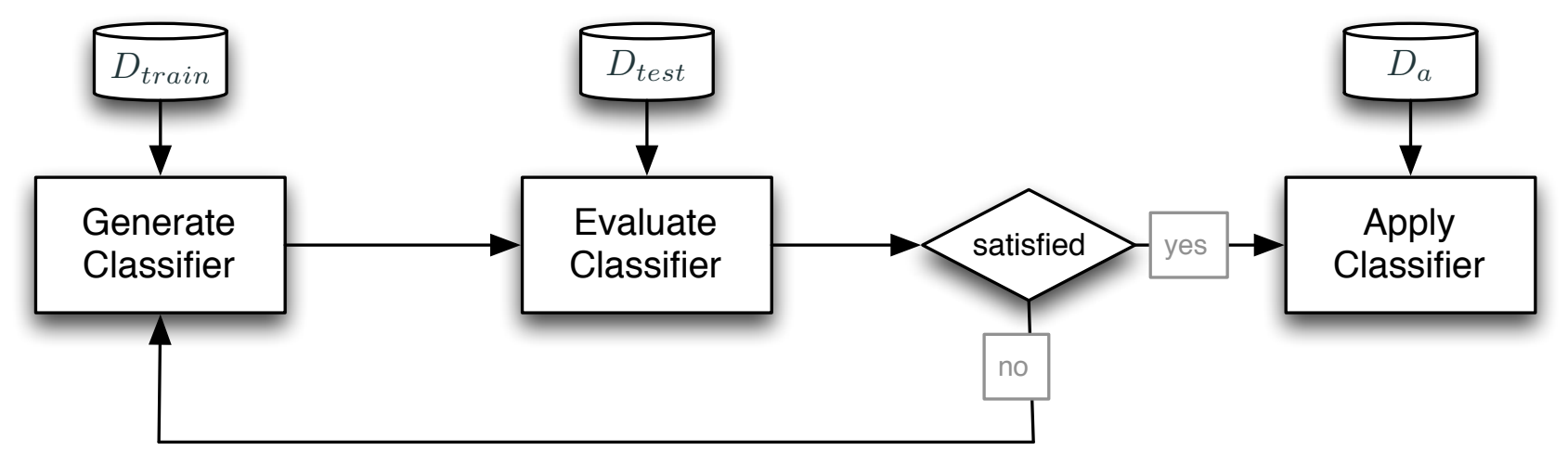


Figure 37: Classification Pipeline (simplified)

- A classifier is trained on the training data set D_{train} and then evaluated on the test data set D_{test}.
- If the performance is satisfactory, the classifier is applied to the unlabeled data from the application D_a. If not, the classifier has to be retrained (e.g. with more training data, with different parameters, or a different classifier).

Data Sets

Note

The classifier has to be evaluated on a different data set than has been used for training. Otherwise the classifier can ”just remember the training data” and can not be assured to generalize to unseen data. That means $D_{train} \neq D_{test}$, optimally $D_{train} \cap D_{test} = \emptyset$.

Assumption for learning:

- The data samples in D_{train}, D_{test} and D_a stem from the same population, i.e., have similar statistical properties.
- Counter-example: A classifier for hand-written digit recognition was trained on handwriting from school children and is applied to handwritings from adults.

Definition (Classification Error)

The classification error on a data set D is defined as follows:

$$E(D) = \frac{1}{m}\left|\{1 \leq i \leq m : y^{(i)} \neq \hat{y}^{(i)}\}\right|$$

with $\hat{y}$ being the predictions of the classifier.

- Classification error counts how many predictions are wrong. The error is then the rate of wrong predictions.
- Classification Accuracy $\mathcal{A} = 1 - Err$.

Definition (Loss Function)

A loss function defines the ”loss of quality” given a prediction and the desired output: $\mathcal{L}(y, \hat{y})$. Common loss functions are

- Squared loss $\mathcal{L}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$
- 0-1 loss $\mathcal{L}(y, \hat{y}) = \mathcal{I}(y \neq \hat{y})$ with $\mathcal{I}$ being an indicator function.

- The prediction error is the loss aggregated over the data set:

$$E = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(y^{(i)}, \hat{y}^{(i)})$$

- Classification error is the aggregation of the 0-1 loss.
- Summed Squared Error (SSE) is the aggregation of the squared loss.

Classifier Evaluation

Examples:

- Consider a data set on object recognition. In the data set are 20 objects, 10 images for each object. Total 200 images. The classifier correctly classifies 190 of them. The classification error is 0.05 (5%).
- Consider a data set on cancer detection. The data set contains 100 samples. 95% of all samples are labeled ”no cancer” only 5% are labeled ”cancer”. A classifier that assigns all samples to ”no-cancer” has a classification error of 5%.

Note

Classification error is not always the best evaluation measure, especially for problems with strong non-uniform class distributions (see cancer example). At least, the error has to be compared to naive baselines (trivial acceptor, trivial rejector, random classifier).

- Many more evaluation measures exist (true positive rate, precision, recall, sensitivity, classification cost,..)[3].

[3] Those are not part of this lecture

Holdout Estimation

- Labeled data set is *randomly* split into train and test data set. Classifier is trained on D_{train} and evaluated on D_{test}.
- Common split ratios are 60-40 and 70-30.
- We get two error estimations $Err_{D_{train}}$ and $Err_{D_{test}}$.

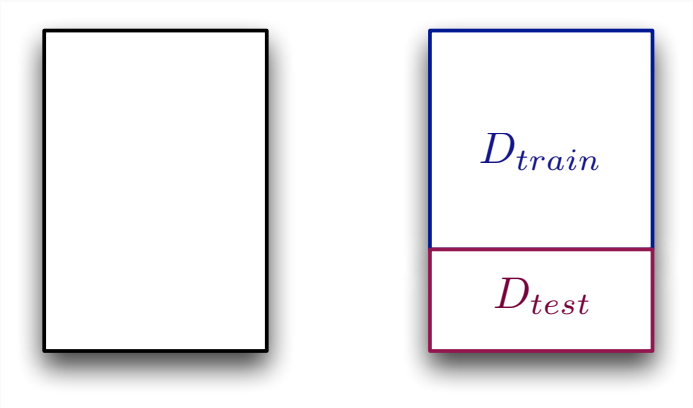


Figure 38: Train-test splits for holdout-error-estimation

Cross-Validation

- Labeled data set is *randomly* split into k disjoint subsets. Classifier is trained k times on respective D^j_{train} and evaluated on respective D^j_{test}. k is usually either 3, 5 or 10.
- We get $2k$ error estimations $Err_{D^j_{train}}$ and $Err_{D^j_{test}}$.
- Cross-validation error is then

$$Err_{cv} = \frac{1}{k} \sum_{j=1}^{k} Err_{D^j_{test}}$$

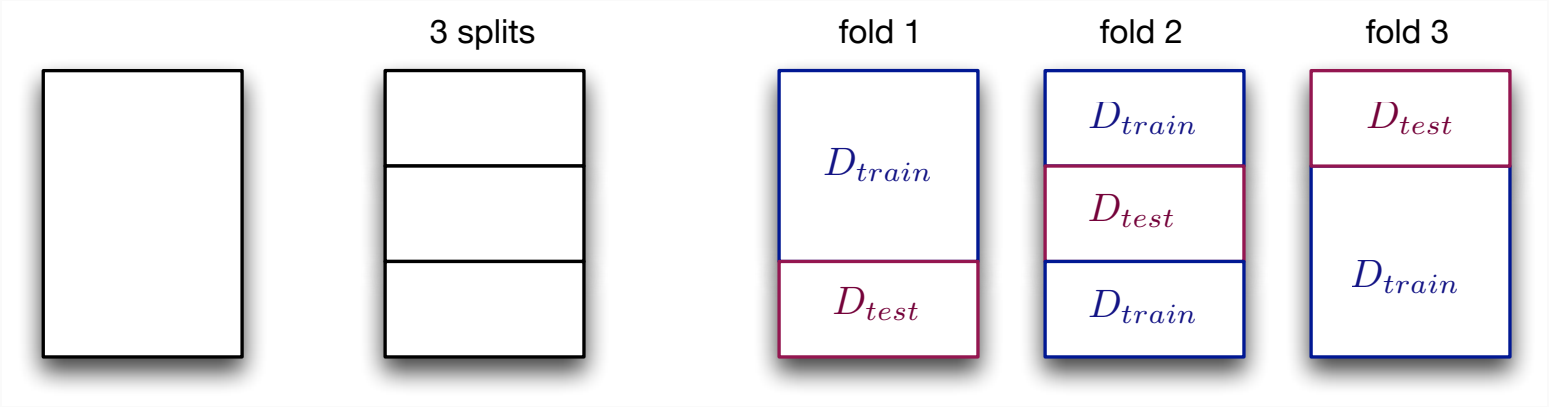


Figure 39: Train-test splits for cross-validation

Classifier Evaluation

- Cross-validation is computationally more expensive (classifier has to be trained k times) but provides the more realistic error estimation.
- Cross-validation only reasonably applicable to small data sets.
- Leave-on-out estimation is cross-validation with the test set containing exactly 1 data sample.
- When splitting the data set into subsets it has to be ensured that training an test sets retain the same properties (underlying distribution). Mostly, random splits is a reasonably good choice.

Learning curves

- Plot the error on the train and test split for different sizes of the training data set.
- The training error gets worse with more training data, because the classifier has to include more (and different) data points into its model.
- The test error (the error on unseen samples) gets better, because the classifier has been able to learn from more data.

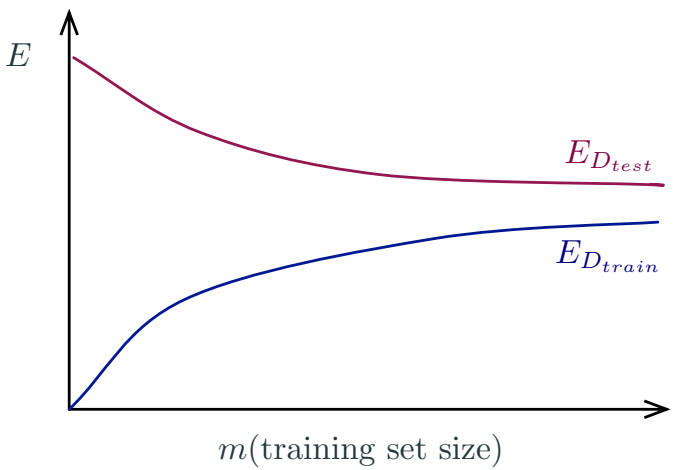


Figure 40: Sample learning curve

Bias vs. Variance

- Bias and variance are two common problems in machine learning
- Errors based on bias are errors stemming from wrong assumptions about the problem
  E.g. fitting a straight line to data point lying on a curve
- Errors based on variance stem from (unimportant) variances in the training data set to which the classifier adapts to.
  E.g. fitting a higher-order polynomial to a data set which samples lie on a quadratic curve

Bias vs. Variance

- Bias and variance for data points sampled from quadratic function
- Green curve is the current predictor (hypothesis)

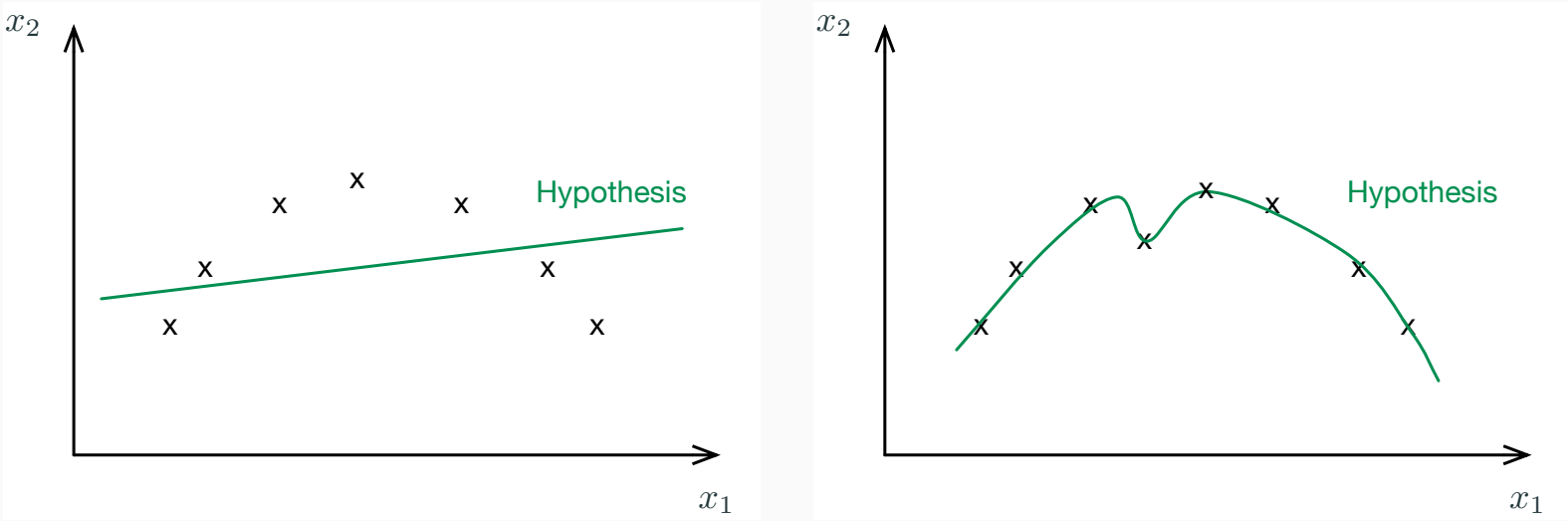


Figure 41: High bias (left) and high variance for which the classifier adapts to an outlier (right)

High Bias – Underfitting

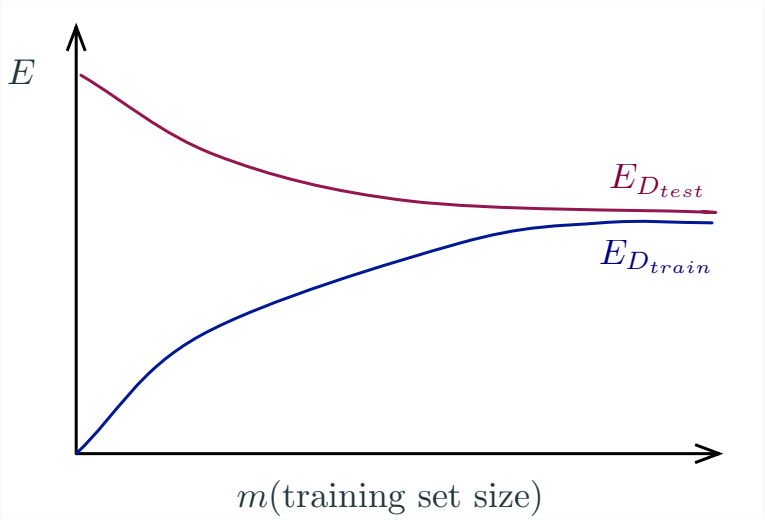


Figure 42: Learning curve indicating high bias)

- Error on train and test set are both high
- More training data does not improve the model

$\Rightarrow$ choose a more complex model (e.g. more layers in the neural network)

High Variance – Overfitting

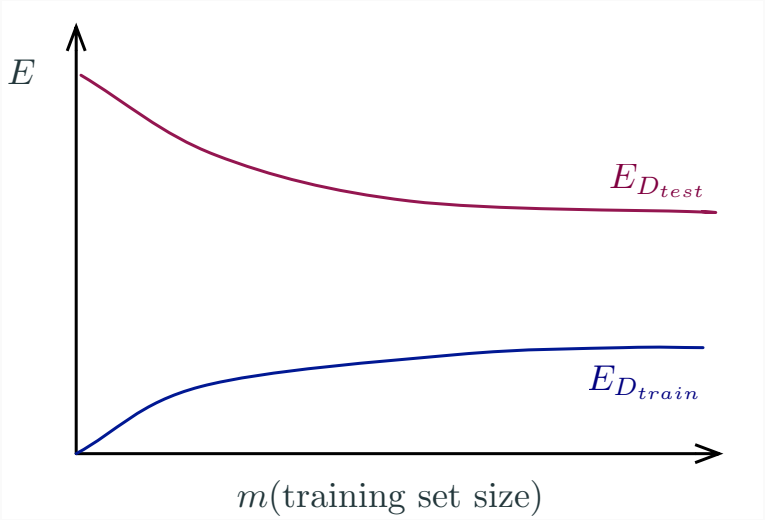


Figure 43: Learning curve indicating high variance)

- Error on train set is low, error on test set is high
- Classifier is well adapted to training data, but can not generalize to unseen data

$\Rightarrow$ choose a simpler model (e.g. less layers in the neural network) or get more training data

Classifier Evaluation

- Error on training and test set should be similar (it means that the classifier generalizes well).
- Error on both, D_{train} and D_{test} should be small (this means the model does not suffer from high bias).

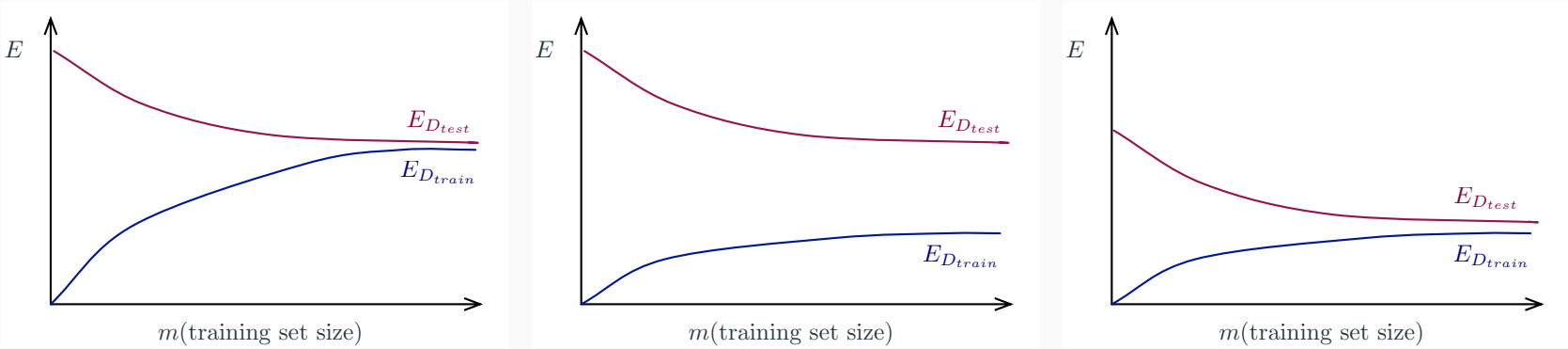


Figure 44: Learning curve indicating high bias (left), high variance (center), good fit (right)

Machine Learning Basics – Summary

- Learning systems adapt to experiences.
- In case of supervised learning, experiences are data samples together with the ”truth” given by a ”teacher”.
- A learning algorithm needs to know what to optimize for, thus we define a loss function and an error (which we want to minimize).
- Learning the model and evaluating its performance needs to be done on different data (sub-)sets.
- Plotting of learning curves helps to find common problems in machine learning, namely bias and variance.

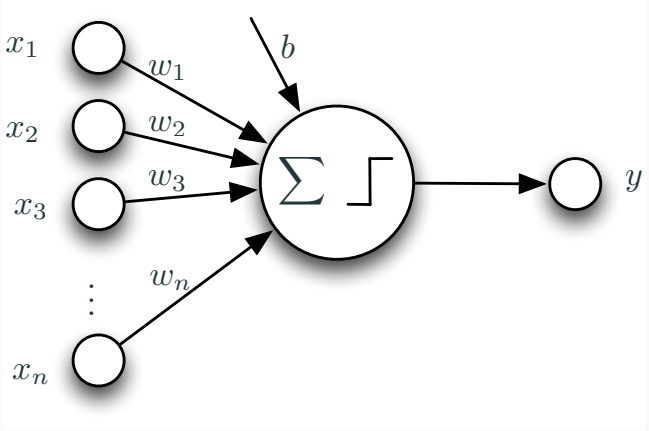
Machine Learning Basics – Summary

Important Concepts

- Supervised, unsupervised learning; reinforcement learning
- Cross-validation
- Learning Curve
- Loss function, classification error
- Train and test data set

Gradient Descent

Gradient Descent



- Perceptron training rule learns a separating hyperplane **if** the data is linearly separable[4].

$$w \leftarrow w + \eta(y - h(w^T x))x$$

- If data is not linearly separable the perceptron might fail to converge.

$\Rightarrow$ The delta rule for training perceptrons finds a best-fit approximation for the hyperplane if the data is not linearly separable.

[4] We use the notation from slide 65 here. Note that x is a vector, while y is a scalar.

Delta Rule and Gradient Descent

- The key idea behind the delta rule is gradient descent.
- Gradient descent is the basis for learning algorithms for multi-layered networks.

Gradient Descent

- Consider an untresholded perceptron, i.e., a perceptron that only takes the weighted sum of the inputs.
- The function σ the perceptron applies to a data point is then

$$\sigma(x) = w^T x$$

- Consider a measure of the training error (see slide 70) for a specific perceptron which is specified by its weights w

$$E(w) = \frac{1}{2}\sum_{i=1}^{m}(y^{(i)} - \sigma(x^{(i)}))^2 = \frac{1}{2}\sum_{i=1}^{m}(y^{(i)} - w^T x^{(i)})^2 \qquad (1)$$

- The error is then the squared difference between the intended output and the actual output summed over the training data

Gradient Descent

- Error in weight space is a convex function of the weights
- The optimal solution (weights that result in the smallest error) is at the bottom of the ”bowl”

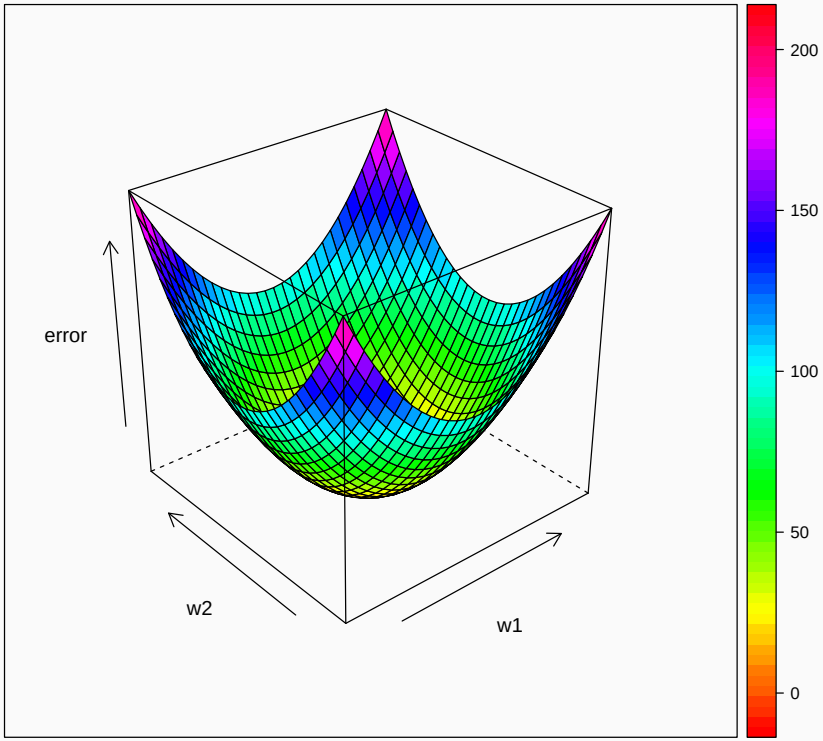
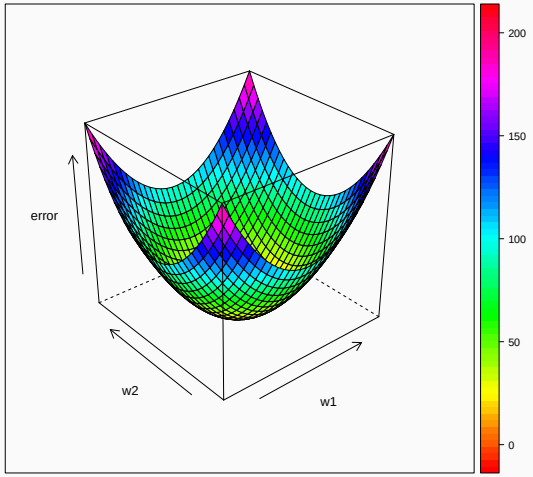


Figure 45: Convex error surface in weight space

Gradient Descent

Intuition

- The x-y plane spans all possible solutions (the hypothesis space).
- Start with any solution.
- Follow the direction of the steepest descent of the error surface to get new weights.
- Evaluate the new solution and again, follow the direction of steepest descent.
- Start with large steps towards the bottom and then make smaller steps so as not to overshoot the target.
- The direction of steepest descent is characterized by the gradient of the function at a particular point w.



Gradient Descent

- Direction of steepest ascent characterized by the gradient.

$$\nabla E(w) = [\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_{n_x}}]$$

- In weight space $\nabla E(w)$ is a vector pointing towards the direction of steepest ascent from a starting point w.
- Weight update is as follows:

$$w \leftarrow w + \Delta w \qquad \Delta w = -\eta \nabla E(w)$$

  - The minus sign ensures that we walk into the direction of steepest descent (towards the minimum).
  - η is the learning rate, determining the size of the step
- Formula can also be written component-wise

$$w_i \leftarrow w_i + \Delta w_i \qquad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Derivation of the gradient for perceptron training error of formula 1

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{k=1}^{m}(y^{(k)} - w^T x^{(k)})^2 \qquad \text{//sum rule}$$

$$= \sum_{k=1}^{m}\frac{\partial}{\partial w_i}\frac{1}{2}(y^{(k)} - w^T x^{(k)})^2$$

$$= \frac{1}{2}\sum_{k=1}^{m}2(y^{(k)} - w^T x^{(k)})\frac{\partial}{\partial w_i}(y^{(k)} - w^T x^{(k)})$$

$$= \sum_{k=1}^{m}(y^{(k)} - w^T x^{(k)})\frac{\partial}{\partial w_i}(y^{(k)} - w^T x^{(k)}) \qquad \text{//i-th component only}$$

$$\frac{\partial E}{\partial w_i} = \sum_{k=1}^{m}(y^{(k)} - w^T x^{(k)})(-x_i^{(k)})$$

Gradient Descent

Definition (Perceptron Delta Rule (Gradient Descent))

For a given training example $(x^{(k)}, y^{(k)})$ the perceptron adapts its weights based on the following training rule:

$$w_i \leftarrow w_i + \Delta w_i$$

$$\Delta w_i = \eta \sum_{k=1}^{m} (y^{(k)} - w^T x^{(k)}) x_i^{(k)}$$

with $x_i^{(k)}$ being the i-th entry in the k input feature vector w_i the weight of the i-th edge and η the learning rate.

- The update rule is tied to the concrete error function (which may in general not be convex).
- The perceptron converges to a optimal solution if η is sufficiently small.
- Converging can sometimes be rather slow.

Gradient Descent

Data: Training Data (x, y)
Result: Weight vector w
Initialize w_i to small random value;
repeat
  $\Delta w_i = 0$;
  /* sum up weight changes */
  for *each training example* $(x^{(k)}, y^{(k)})$ **do**
    $\hat{y}^{(k)} = w^T x^{(k)}$;
    $\Delta w_i \leftarrow \Delta w_i + \eta(y^{(k)} - \hat{y}^{(k)}) x_i^{(k)}$;
  end
  /* apply weight changes */
  for *each weight* w_i **do**
    $w_i \leftarrow w_i + \Delta w_i$;
  end
until *termination condition is met*;

Algorithm 1: Gradient Descent Algorithm

Gradient Descent

- Gradient descent searches through the hypothesis space and is generally applicable if
  1. parameter space is continuous
  2. error function is differentiable w.r.t. parameters
- If error surface has multiple minima gradient descent is not guaranteed to find the global minimum
- Convergence can be slow, at each iteration the model needs to be applied to the whole training data set
  $\Rightarrow$ Stochastic Gradient Descent approximates the gradient descent solution by updating weights after each data item

Stochastic Gradient Descent

Data: Training Data (x, y)
Result: Weight vector w
Initialize w_i to small random value;
repeat
    $\Delta w_i = 0$;
    /* apply weight changes */
    for *each training example* $(x^{(k)}, y^{(k)})$ **do**
        $\hat{y}^{(k)} = w^T x^{(k)}$;
        $w_i \leftarrow w_i + \eta(y^{(k)} - \hat{y}^{(k)}) x_i^{(k)}$;
    end
until *termination condition is met*;

Algorithm 2: Stochastic Gradient Descent Algorithm

Summary

- Gradient descent updates the weights after calculating the error for all training samples (the batch), also called batch gradient descent

$$w_i \leftarrow w_i + \Delta w_i$$

$$\Delta w_i = \eta \sum_{k=1}^{m} (y^{(k)} - w^T x^{(k)}) x_i^{(k)}$$

- Stochastic gradient descent updates the weights after each training sample, also called incremental gradient descent

$$w_i \leftarrow w_i + \eta (y^{(k)} - \hat{y}^{(k)}) x_i^{(k)}$$

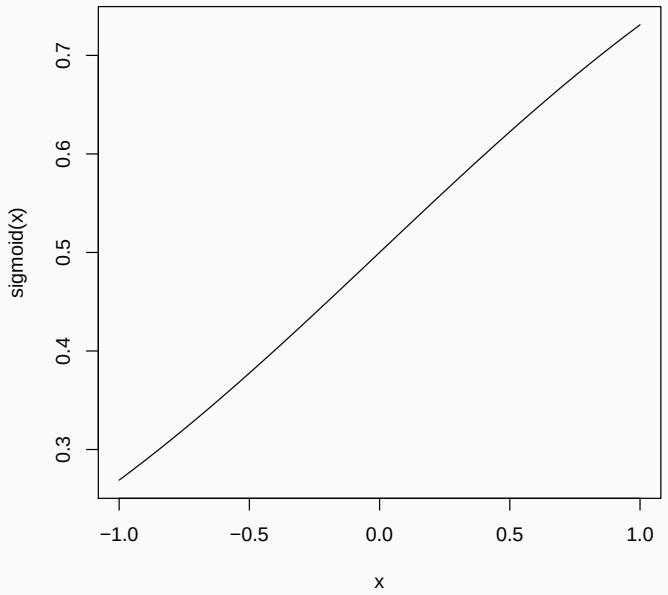
  - Also known as delta rule, Widrow-Hoff rule and Adaline rule
  - If η is sufficiently small stochastic gradient descent approximates gradient descent at arbitrary accuracy.

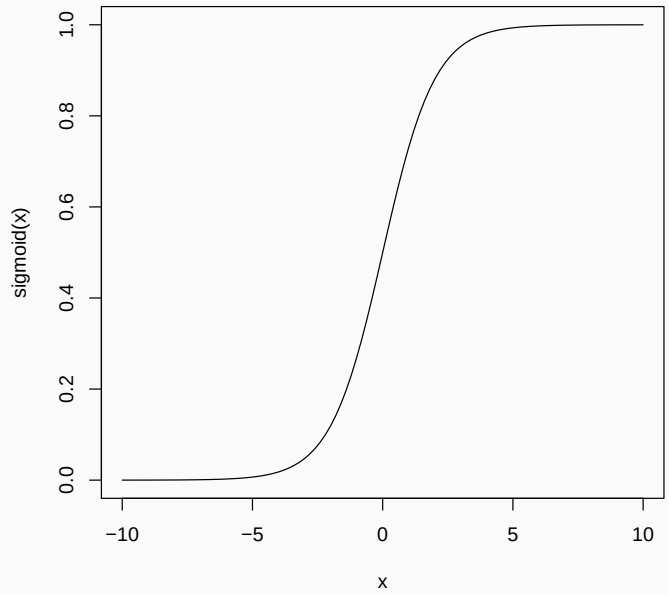
Types of Neurons

Sigmoid Units

- Values in $[0, 1]$, for large positive and large negative values of x, the gradient is nearly zero

$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad \frac{\partial}{\partial x}\sigma(x) = \sigma(x)(1 - \sigma(x))$$

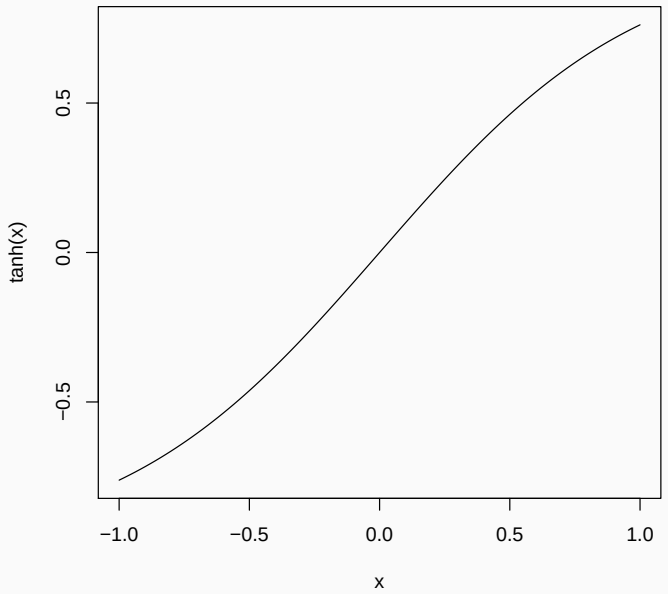


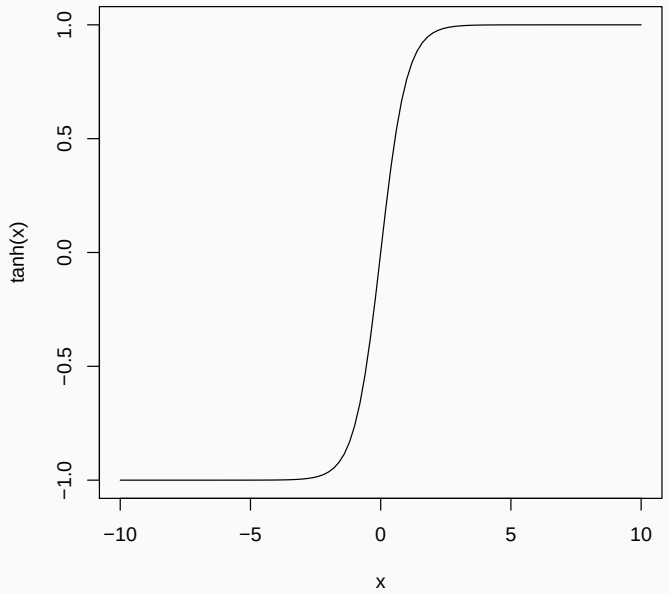


Tanh Units(Tangens hyperbolicus)

- Values in $[-1, 1]$, for large positive and large negative values of x, the gradient is nearly zero

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad \frac{\partial}{\partial x} \tanh(x) = 1 - \tanh(x)^2$$

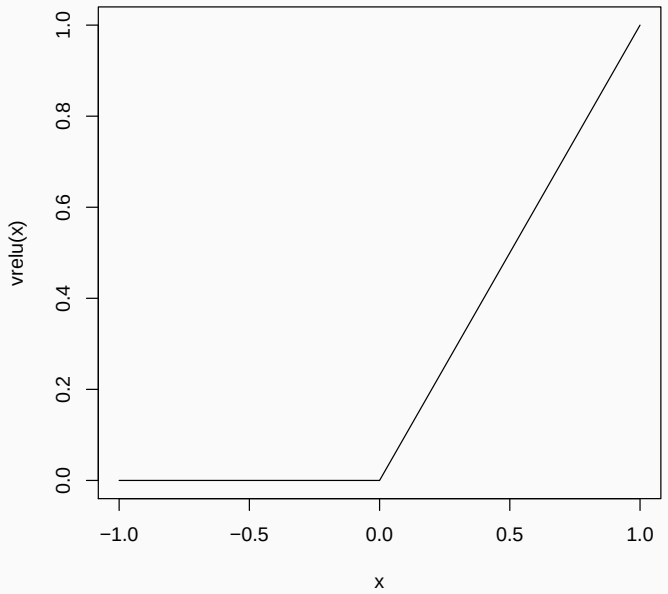


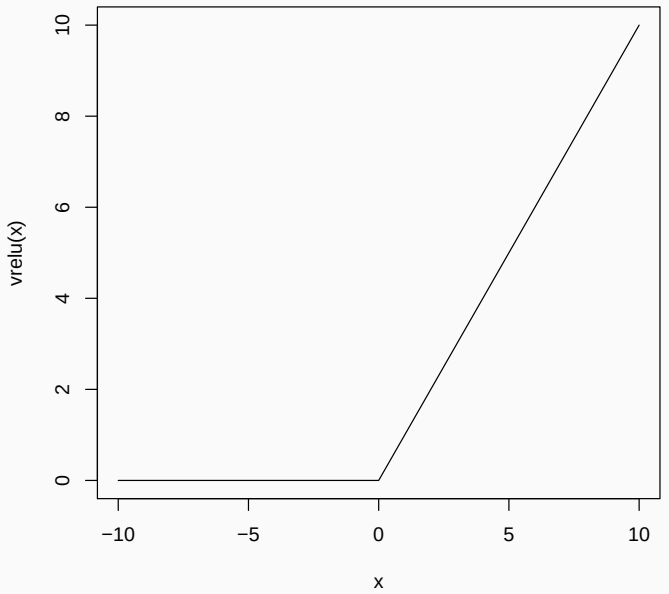


ReLU (Rectified Linear Units)

- Values in $[0, 1]$, gradient is 1 for all positive values, undefined for 0, but in practice it can be set to either 0 or 1

$$relu(x) = \max(x, 0) \qquad \frac{\partial}{\partial x} relu(x) = \begin{cases} 1 & \text{for } x > 0 \\ 0 & \text{for } x < 0 \\ \text{undefined} & x = 0 \end{cases}$$

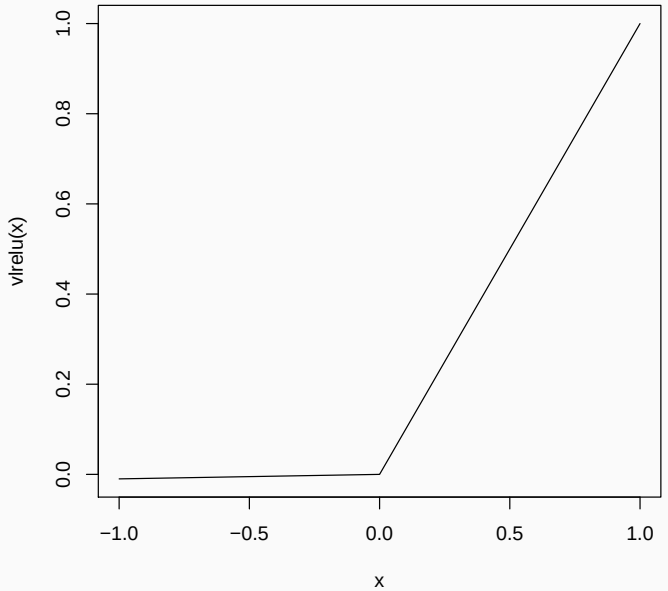


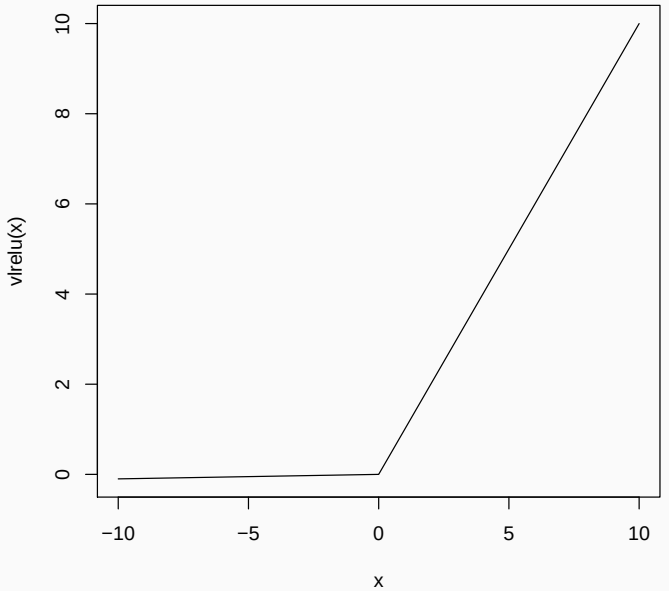


LReLU (Leaky Rectified Linear Units)

- Adaptation of ReLU, with small constant gradient for values smaller than 0

$$lrelu(x) = \max(x, 0.001) \qquad \frac{\partial}{\partial x} lrelu(x) = \begin{cases} 1 & \text{for } x > 0 \\ 0.001 & \text{for } x < 0 \\ \text{undefined} & x = 0 \end{cases}$$





Summary

- Neuron types usage in practise, rules of thumb (as of 2017)

Unit type	Comment
σ	in output layer for binary classification
tanh	mostly superior to σ
relu	similar usage to tanh, use when problems with vanishing gradients
lrelu	shown to be better than relu, but not much used in practise

Back-Propagation

Notation

- n_x – input feature vector size
- n_y – output size (number of classes)
- y – desired output (target)
- $\hat{y}$ – predicted output (what the learner produces)
- $(x^{(k)}, y^{(k)})$ – $k-th$ training item
- m – number of examples in the data set
- $X \in \mathcal{R}^{n_x \times m}$ – input matrix (each item is a column, each feature is a row)
- L – number of layers in the network
- n_h^l – number of units in the l-th hidden layer
- superscript [l] denotes the index of the layer
- $b^k \in \mathcal{R}^{n_h^k}$ – bias vector for $k-th$ layer
- $W^{[k]} \in \mathcal{R}^{n_h^k \times n_h^{k-1}}$ – weight matrix before $k-th$ layer

Notation

- By convention, a neural network with one input, one hidden and one output layer is called a 2-layer neural network (the input layer does not count as its units do not compute anything)

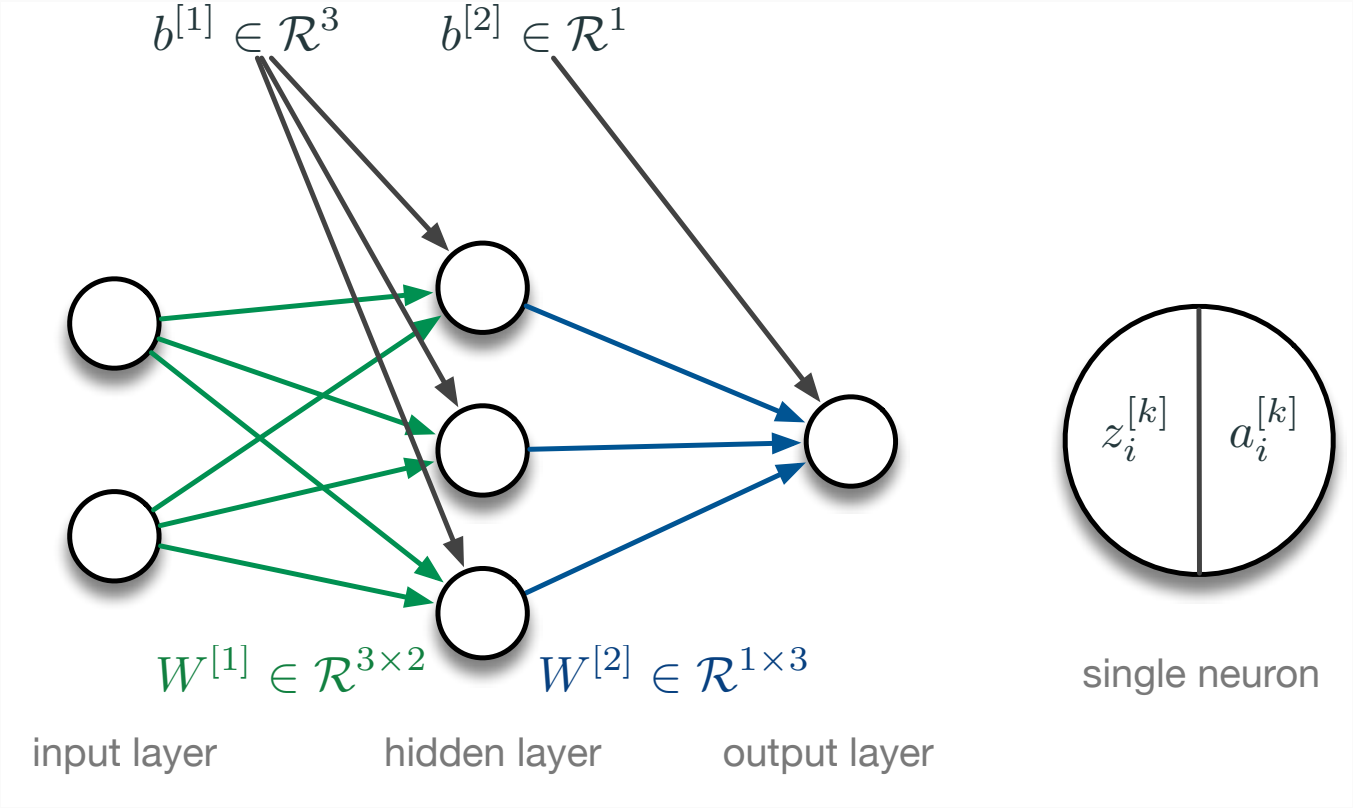


Figure 46: Simple 2-3-1 multi-layered network

Notation

- Vector-notation for the example network on the previous slide

$$a^{[0]} = x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

$$b^{[1]} = \begin{pmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{pmatrix}, \quad z^{[1]} = \begin{pmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{pmatrix}, \quad a^{[1]} = \begin{pmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{pmatrix}$$

$$b^{[2]} = \left(b_1^{[2]}\right), \quad z^{[2]} = \left(z_1^{[2]}\right), \quad a^{[2]} = y = \left(a_1^{[2]}\right)$$

$$W^{[1]T} = \begin{pmatrix} - w_1^{[1]T} - \\ - w_2^{[1]T} - \\ - w_3^{[1]T} - \end{pmatrix} \in \mathcal{R}^{3\times 2}, \quad W^{[2]T} = \left(- w_1^{[2]T} -\right) \in \mathcal{R}^{1\times 3}$$

- with $w_1^{[1]}$ being the vector of input weights for the first unit in the first hidden layer
- $w_{i,j}^{[k]}$ can be read as: weight from unit i to unit j in layer k

Foward Calculation

Forward Pass

$$z^{[1]} = W^{[1]T} x + b^{[1]}, \quad a^{[1]} = \sigma(z^{[1]})$$
$$z^{[2]} = W^{[2]T} a^{[1]} + b^{[2]}, \quad a^{[2]} = \sigma(z^{[2]}) = \hat{y}$$

Checking the dimensions:

$$z^{[1]} = \begin{pmatrix} w_{1,1}^{[1]} & w_{2,1}^{[1]} \\ w_{1,2}^{[1]} & w_{2,2}^{[1]} \\ w_{1,3}^{[1]} & w_{2,3}^{[1]} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{pmatrix}, \quad a^{[1]} = \sigma \begin{pmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{pmatrix}$$

$$z^{[2]} = \begin{pmatrix} w_{1,1}^{[2]} & w_{2,1}^{[2]} & w_{3,1}^{[2]} \end{pmatrix} \begin{pmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{pmatrix} + \begin{pmatrix} b_1^{[2]} \end{pmatrix}, \quad a^{[2]} = \sigma \begin{pmatrix} z_1^{[2]} \end{pmatrix} = \hat{y}$$

- Note: σ is applied element-wise to a vector

General Idea

- Calculate the output of the net for a training example
- Calculate the loss/error for this training example
- Calculate the gradient of the error surface w.r.t. the weights in each layer
- Adapt the weights in each layer

For the (simpler) derivation of the Back-Propagation algorithm we make the following assumptions:

1. The activation function of the hidden and output units is the sigmoid function σ (see slide 96).
2. We use the squared loss (see slide 70).

Loss function

- Forward pass to calculate $\hat{y}$

$$\hat{y} = a^{[2]} = \sigma(z^{[2]}),\ z^{[2]} = W^{[2]T} a^{[1]} + b^{[2]},$$
$$a^{[1]} = \sigma(z^{[1]}),\ z^{[1]} = W^{[1]T} x + b^{[1]}$$

- We use the squared loss

$$\mathcal{L}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

- To calculate the weight updates we need the partial derivatives of the loss function w.r.t. to the model parameters W_1, W_2, b_1, b_2: $\frac{\partial \mathcal{L}}{\partial W^{[1]}}$, $\frac{\partial \mathcal{L}}{\partial W^{[2]}}$, $\frac{\partial \mathcal{L}}{\partial b^{[1]}}$, $\frac{\partial \mathcal{L}}{\partial b^{[2]}}$

Weight update

- The update for a single weight $w_{ij}^{[l]}$ in any layer is negative derivative of the loss function w.r.t. to this weight
- We calculate the gradient and move in the direction of the steepest descent (minus sign)

$$w_{ij}^{[l]} \leftarrow w_{ij}^{[l]} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}^{[l]}}$$

Derivation for a single weight

- Derive equations for output units and hidden units separately
- Use a simplified notation as below

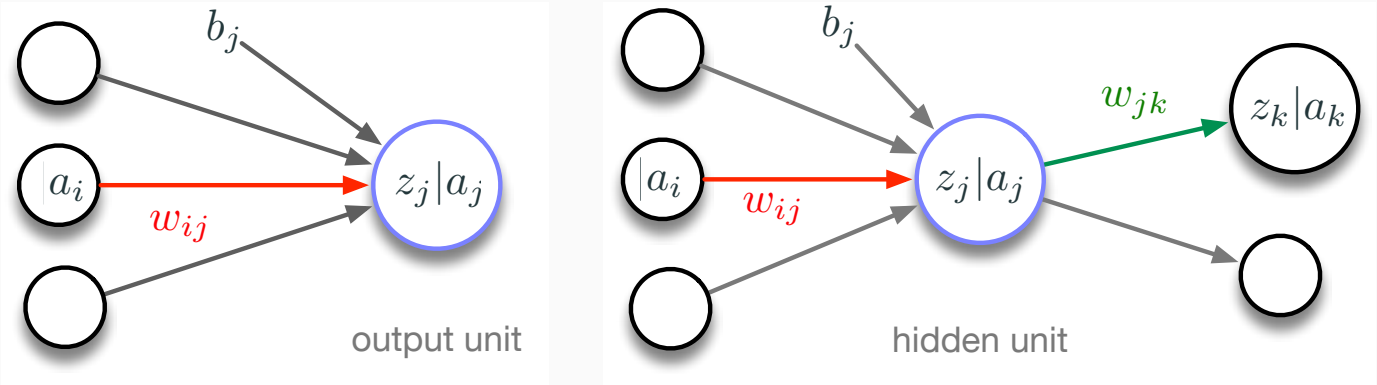
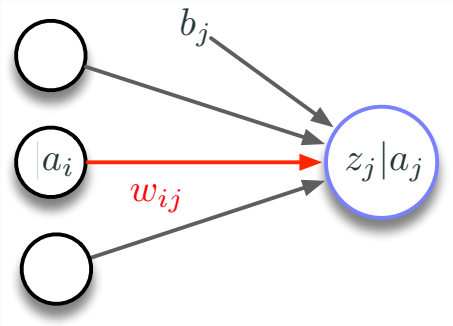


Figure 47: Simplified notation for output units (left) and input units (right)

For Output Units



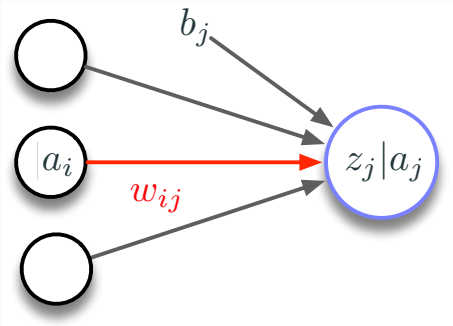
- Derive $\frac{\partial \mathcal{L}}{\partial w_{ij}}$, observe that w_{ij} influence $\mathcal{L}$ through z_j, and only through z_j (see figure). Using the chain rule for derivates we can write

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}$$

- Observe that z_j influence $\mathcal{L}$ through a_j, and only through a_j. We can write

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial a_j} \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}$$

For Output Units



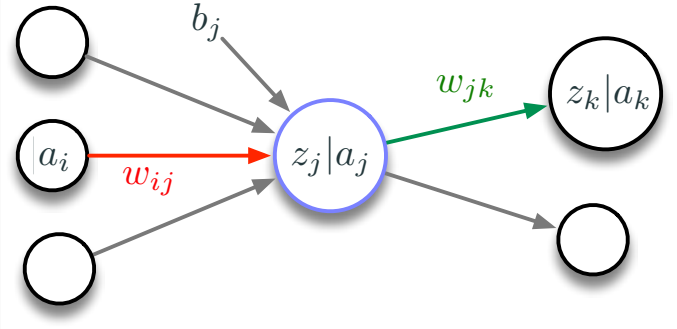
$$\frac{\partial \mathcal{L}}{\partial a_j} = \frac{\partial}{\partial a_j}\frac{1}{2}\sum_{k \in Output}(y_k - \hat{y}_k)^2 = \frac{\partial}{\partial a_j}\frac{1}{2}(y_j - \hat{y}_j)^2 = -(y_j - \hat{y}) = -(y_j - a_j)$$

$$\frac{\partial a_j}{\partial z_j} = \frac{\partial}{\partial z_j}\sigma(z_j) = \sigma(z_j)(1 - \sigma(z_j)) = a_j(1 - a_j)$$

$$\frac{\partial z_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\sum_i (w_{ij}a_i) + b_j = a_i$$

$$\Rightarrow \frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial a_j}\frac{\partial a_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}} = -(y_j - a_j)(a_j(1 - a_j))a_i$$

For Hidden Units



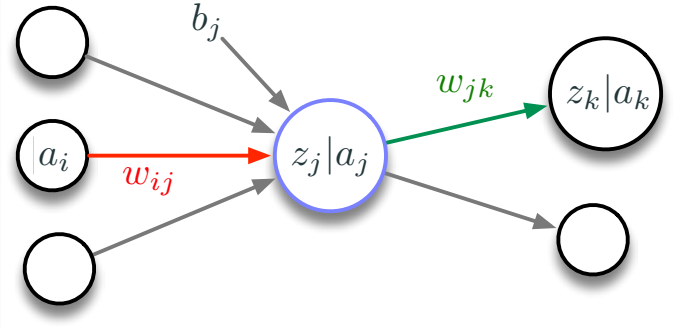
- Derive $\frac{\partial \mathcal{L}}{\partial w_{ij}}$, observe that w_{ij} influence $\mathcal{L}$ through z_j, and only through z_j (see figure). Using the chain rule for derivates we can write

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}$$

- Observe that z_j influence $\mathcal{L}$ through all z_k, that is through all nodes in the next layer. Therefore

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \sum_k \left(\frac{\partial \mathcal{L}}{\partial z_k} \frac{\partial z_k}{\partial z_j} \right) \frac{\partial z_j}{\partial w_{ij}}$$

For Hidden Units



- z_j influences z_k through a_j, and only through a_j. Using the chain rule we can write:

$$\sum_k \frac{\partial \mathcal{L}}{\partial z_k} \frac{\partial z_k}{\partial z_j} = \sum_k \frac{\partial \mathcal{L}}{\partial z_k} \frac{\partial z_k}{\partial a_j} \frac{\partial a_j}{\partial z_j}$$

- Getting the partial derivatives

$$\frac{\partial z_k}{\partial a_j} = \frac{\partial}{\partial a_j} \sum_j w_{jk} a_j + b_k = w_{jk}$$

$$\frac{\partial a_j}{\partial z_j} = \sigma(z_j)(1 - \sigma(z_j)) = a_j(1 - a_j)$$

For Hidden Units

- Substituting back in

$$\sum_k \frac{\partial \mathcal{L}}{\partial z_k}\frac{\partial z_k}{\partial z_j} = \sum_k \frac{\partial \mathcal{L}}{\partial z_k}\frac{\partial z_k}{\partial a_j}\frac{\partial a_j}{\partial z_j} = \sum_k \frac{\partial \mathcal{L}}{\partial z_k} w_{jk} a_j(1-a_j)$$

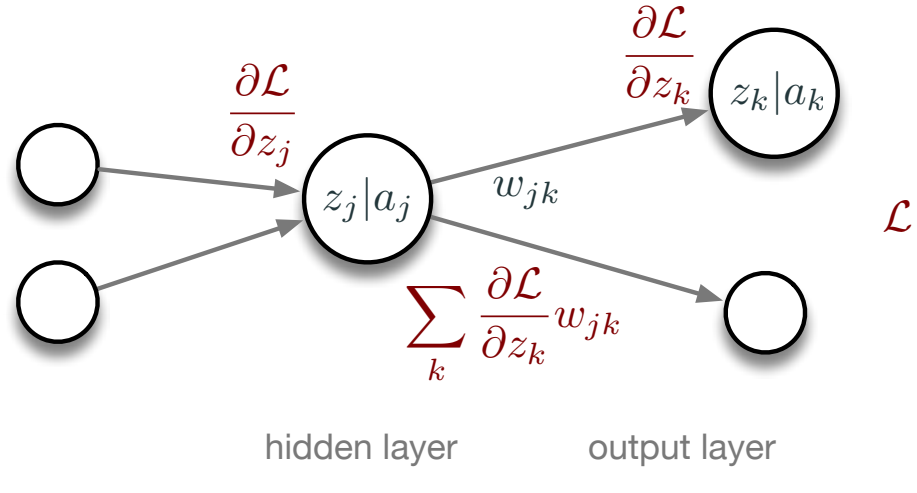
$$= a_j(1-a_j)\sum_k \frac{\partial \mathcal{L}}{\partial z_k} w_{jk}$$

- Calculating the loss w.r.t. the weights

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = a_j(1-a_j)\Big(\sum_k \frac{\partial \mathcal{L}}{\partial z_k} w_{jk}\Big) a_i$$

- Note, that there are still partial derivates in the formula. However, if we start from output units and iteratively apply the weight update, we know $\frac{\partial \mathcal{L}}{\partial z_k}$ because it can be directly computed for output units

Propagating through the network



- For output units
  $\frac{\partial \mathcal{L}}{\partial w_{ij}} = -(y_j - a_j)(a_j(1-a_j))a_i \quad \frac{\partial \mathcal{L}}{\partial z_{ij}} = -(y_j - a_j)(a_j(1-a_j))$
- For hidden units
  $\frac{\partial \mathcal{L}}{\partial w_{ij}} = a_j(1-a_j)\left(\sum_k \frac{\partial \mathcal{L}}{\partial z_k} w_{jk}\right) a_i$
- Weight update
  $w_{ij} \leftarrow w_{ij} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$

Back-Propagation

Data: Data set $\{(x, y)\}$, trained neural network
Result: Trained neural network
Initialize $w_{ij}^{[k]}$ to small random values;
while *termination criterion not met* **do**
  for *each* (x, y) **do**
    `/* forward pass */`
    calculate $\hat{y}$ and activation a_k of each unit;
    `/* backward pass */`
    for *each output unit L* **do**
      $\delta_L = -(y - \hat{y})(\hat{y}(1 - \hat{y}))$
    end
    for *each hidden unit h* **do**
      $\delta_h = (a_h(1 - a_h)) \sum_{k \in successors} w_{hk}\delta_k$;
    end
    for *each weight* w_{ij} **do**
      $w_{ij} \leftarrow w_{ij} + \eta\delta_j a_i$;
    end
  end
end

Algorithm 3: Backpropagation with stochastic gradient descent

Notes

- The weight update for the bias units can be similarly calculated (the formula only differs slightly from the formulas derived for w_{ij}).
- The formulas were derived for stochastic gradient descent, but are not much different for batch gradient descent (only the loss has to be replaced with the error).
- The derived formulas are tied to the specific choice of loss function; for each choice of loss function the update rules for the weights have to be derived separately.
- Because the error surface is in general non-convex, backpropagation is only guaranteed to converge to a local minimum, but works surprisingly well in practice.

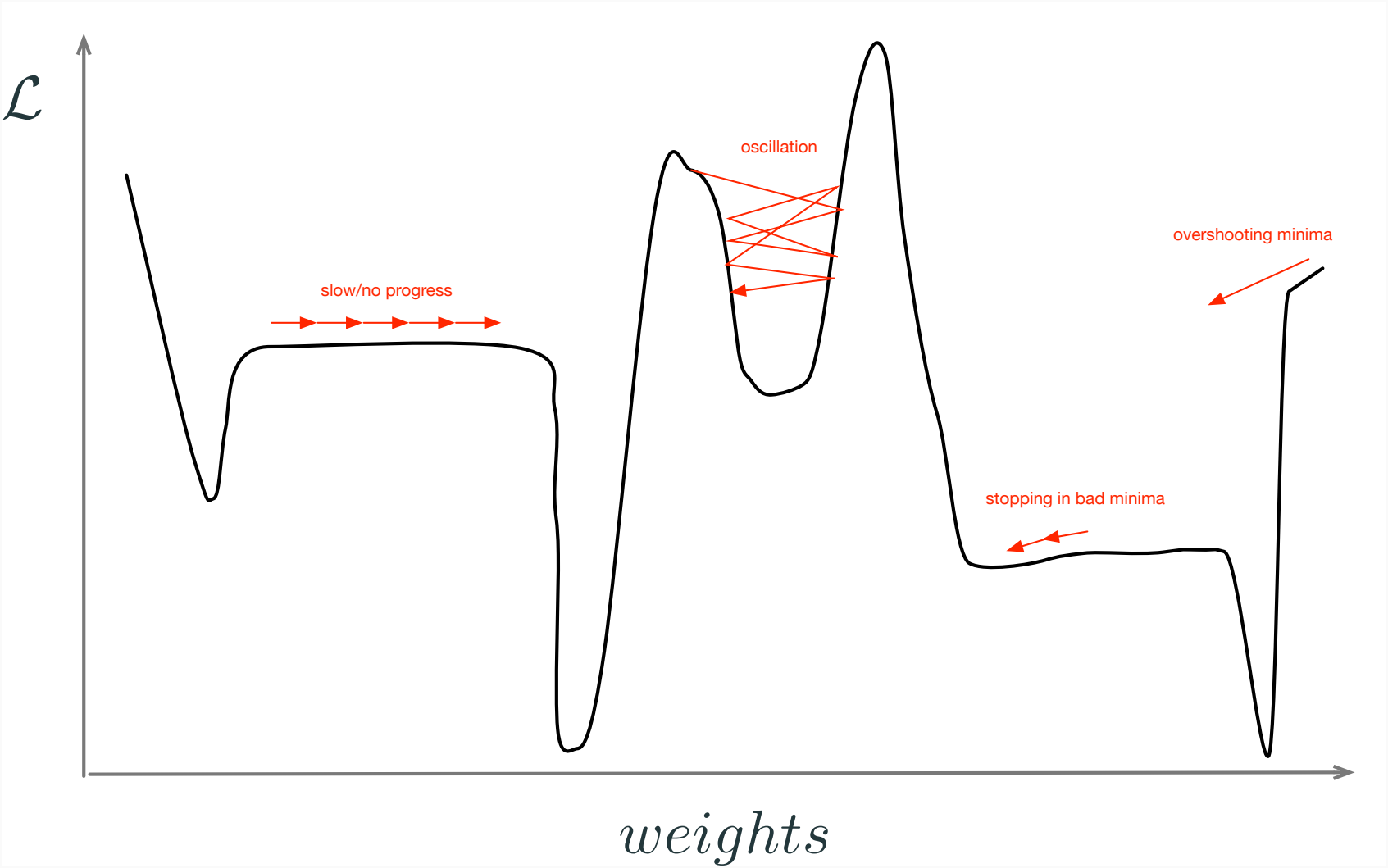


Figure 48: Possible backpropagation errors

Heuristics for avoiding typical backpropagation problems

- Weight initialization
  - Start with different weight initializations (somewhere else in weight space, this might lead to better minimum)
- Adaption of η
  - Vary η over time
  - Start with large learning rate e.g., 0.9, end with small learning rate, e.g. 0.01

- There is no optimal learning rate for all problems (all surfaces of the loss function)

	Learning Rate	
	large	small
advantages	faster changes in plateaus	less likely oscillations
	faster movement towards distant minima	small global minima can be reached
disadvantages	overshooting small minima	large training time
	oscillation more likely	stalling in plateaus or local minima

Training Heuristics

Training Heuristics

General problems when training Neural Networks

- Overfitting (model fits perfectly to a sample of the data, but not to unseen data points)
- Underfitting (model fits poorly to the data)
- Slow training
- Bad performance (model does not what it is supposed to do, or does it very poorly, for example worse than random guessing)

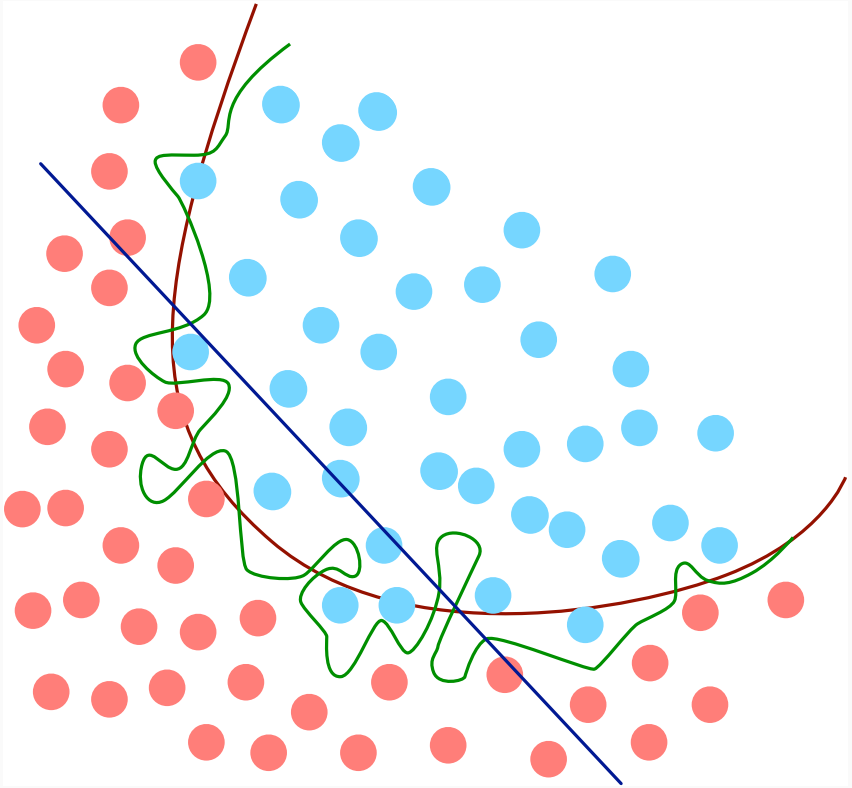


Figure 49: Examples of overfitting (green classifier), underfitting (blue classifier) and good fitting models (brown classifier) (After work from Chabacano, via Wikimedia Commons)

Random Initialization

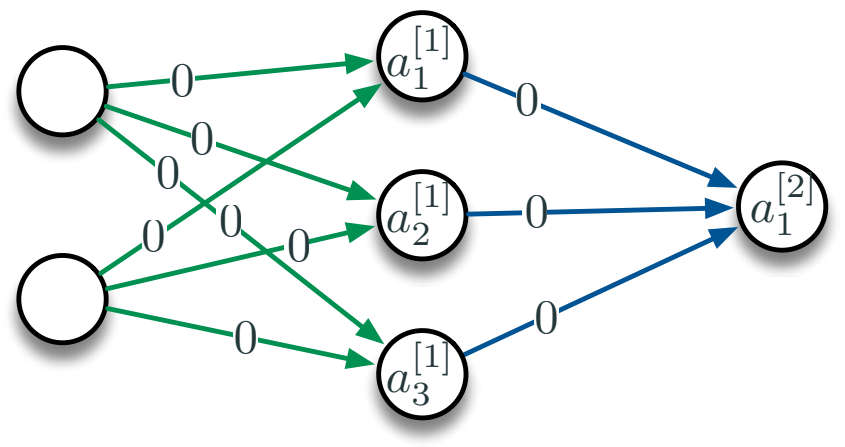


Figure 50: Network with weights initialized to zero

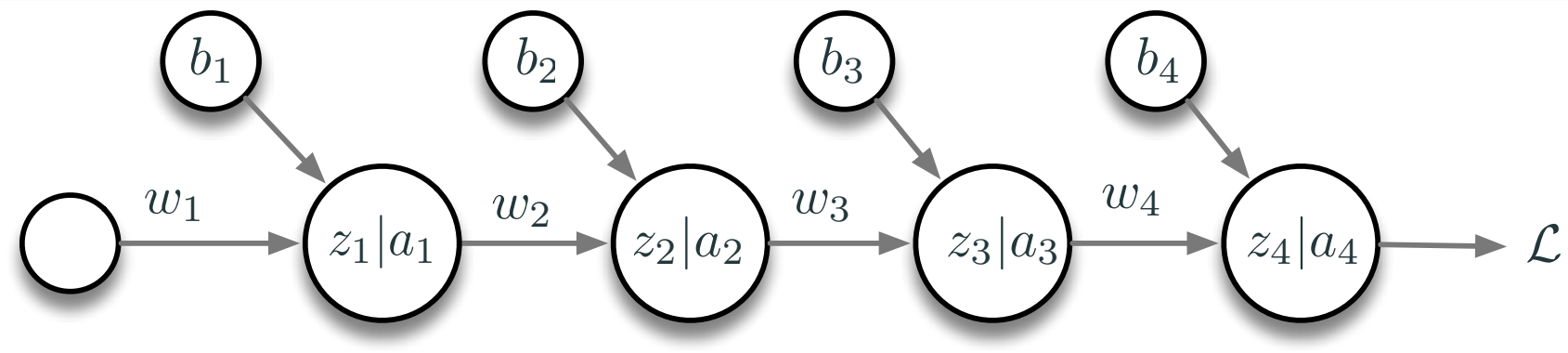
- Consider a network with all weights initialized to zero
- Forward pass for training sample x would lead to all activations in the hidden layer being the same, i.e., $a_1^{[1]} = a_2^{[1]} = a_3^{[1]}$.
- Backpropagating the error leads to the same update of all the weights of $W^{[2]}$ (see formulas derived on slide 110) and consequently all the weights of $W^{[1]}$ receive the same updates (see formulas derived on slide 111).

Random Initialization

- This is not only true for zero weights, but also for weights that are the same (e.g. if all weights were initialized to 0.001).
- Thus, we need to break the symmetry in order for the network to learn something.

$\Rightarrow$ Weights should be initialized with small random numbers.

Vanishing and Exploding Gradients



- Consider the simple network above
- The derivative of the loss w.r.t. to the first bias is

$$\frac{\partial \mathcal{L}}{\partial b_1} = \sigma'(z_1) \cdot w_2 \cdot \sigma'(z_2) \cdot w_3 \cdot \sigma'(z_3) \cdot w_4 \cdot \sigma'(z_4) \cdot \frac{\partial \mathcal{L}}{\partial a_4}$$

with σ' being the derivative of the σ activation function

Vanishing and Exploding Gradients

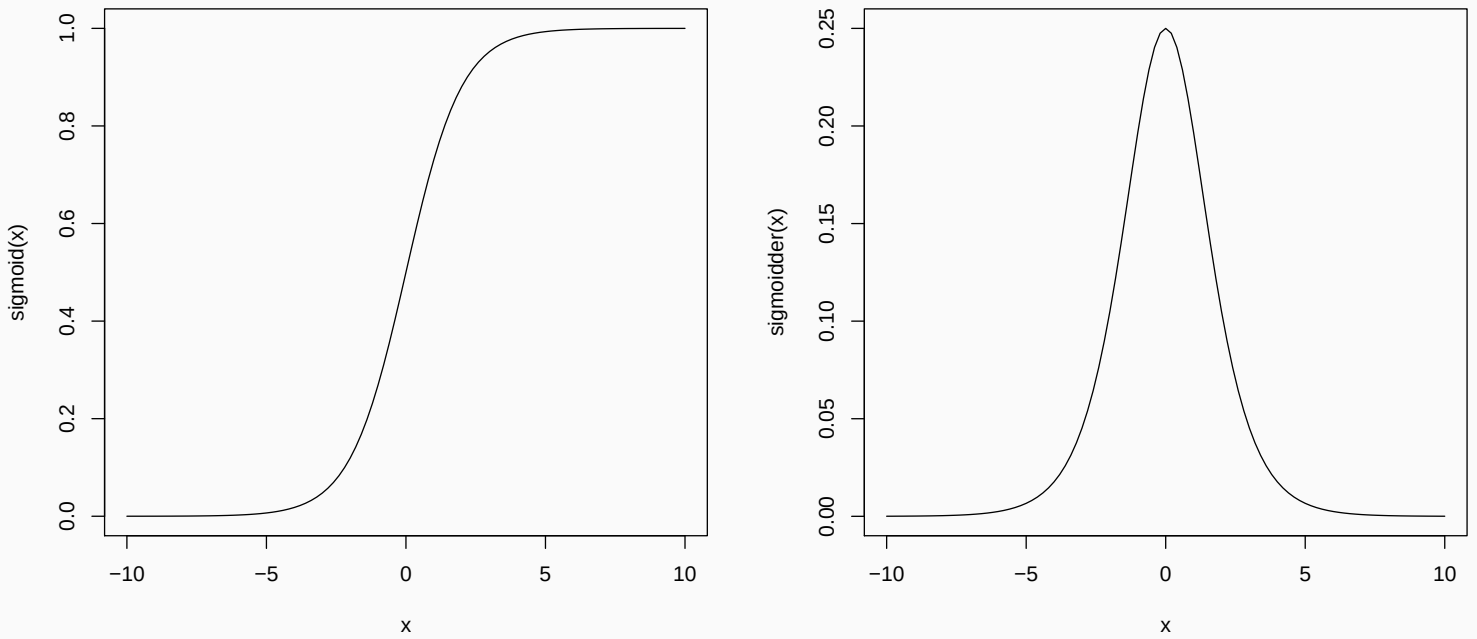


Figure 51: Sigmoid activation function (left) and its derivative (right)

Vanishing and Exploding Gradients

- If the argument of the sigmoid function is large (either positive or negative), the derivative $\sigma'(\cdot)$ is near zero
- $\sigma'(\cdot)$ is 0.25 at maximum
- For the derivative of the loss w.r.t. to weights and biases in lower layers of the network, the single derivatives $\sigma'(\cdot)$ get multiplied, and thus the values get even smaller.
- This phenomenon is called vanishing gradients, i.e., gradients that are nearly zero, which means there are no changes after the update of biases and weights and learning is very slow (and sometimes non-existent).
- Similarly, gradients can also explode (be very large) and cause the network to overstep minima.

Vanishing and Exploding Gradients

Things to try:

- $\Rightarrow$ Gradient clipping (normalize the gradient vectors to maximal length) - for exploding gradients
- $\Rightarrow$ Choose different activation functions (e.g. ReLU)
- $\Rightarrow$ For recurrent problems, other architectures, e.g. Long Short-Term Memory (LSTMs) networks

Early Stopping

- When models are trained too long (with not enough variance in training data) they might adapt to noise in the training data, that is, they might overfit
- This can be observed by plotting the changes in error over training time for both, training and validation set
- $\Rightarrow$ Stop when the error on the validation set starts to increse

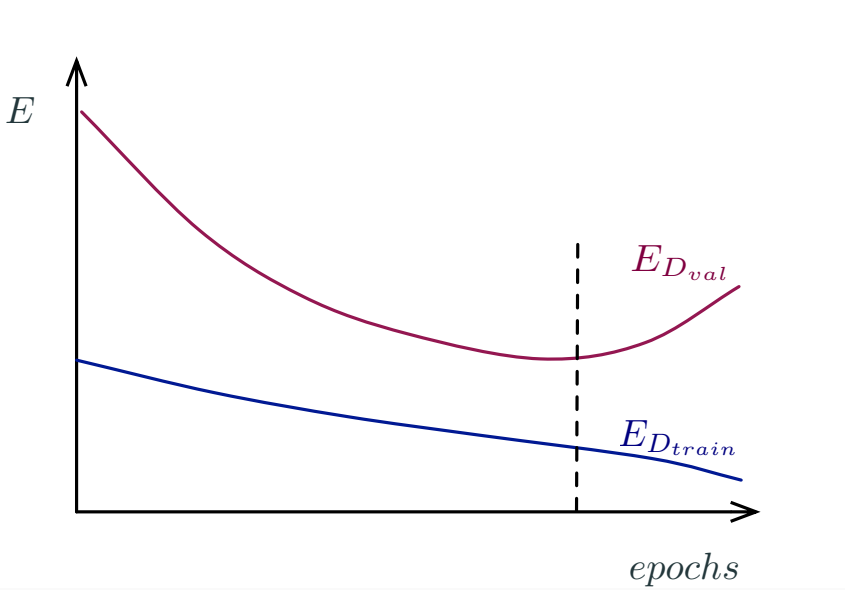
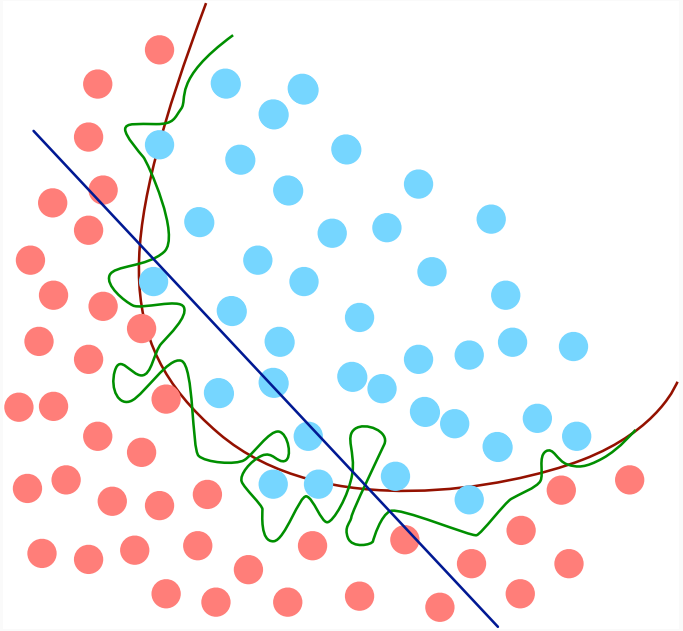


Figure 52: Overfitting can be prevented by stopping training at the time indicated by the dashed line

Reduce Number of Hyperparameters

- If the model does overfit, the reason might be, that there are too many free parameters that have to be fixed.
- In the example, a polynomial with degree $k > 2$ overfits, while a quadratic function provides a good fit
- In neural networks the number of parameters can be reduced by
  - Removing layers of the network
  - Removing nodes from layers of the network



Weight Regularization

- Regularization is a technique to prevent overfitting
- A term that ”regularizes” the weights is added to the loss function
- Squared loss

$$\mathcal{L}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

- L2-regularized squared loss, with m being the size of the training data set, λ regularization parameter

$$\mathcal{L}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2 + \frac{\lambda}{2m} \sum_w w^2$$

- In general, a L2-regularized loss function can be written as

$$\mathcal{L} = \mathcal{L}_0 + \frac{\lambda}{2m} \sum_w w^2$$

with $\mathcal{L}_0$ being the unregularized loss function

Weight Regularization

- The regularization term forces the weights to be small.
- λ expresses how much influence the regularization should have relative to the original loss function.
- Regularization can also improve performance if enough training data is available. Intuition: With unregularized loss functions the length of the weight vector is likely to grow. Changes due to gradients do not change the vector much (since the changes are added to the old vector, which is large). Thus, the vector more or less always points in the same direction. This means, that in an unregularized setting, the weight space can not properly explored.[5]

[5] This fact has not been proven, but it seems reasonable and empirical evidence exists.

Dropout [9]

- Complex models can learn complex relationships in the data.
- In case there is not enough training data, models learn patterns in the data that are results of sampling noise (if more data would be available, there would be more variance and the model could learn that those relations are actually NOT patterns).
- This leads to overfitting, i.e. adapting to patterns of the training data, that are not present in the general population or in the test data.
- One solution would be: train multiple models with different parameter settings and average their decision. The assumption is that while some models overfit on noisy patterns others will not (this is called *ensemble learning*).
- For deep neural networks, training multiple models is usually not feasible.

$\rightarrow$ Dropout is an approximate solution.

Dropout [9]

- Dropout prevents overfitting and combines multiple models efficiently.
- Idea:
  - For each training sample temporarily remove a hidden or visible unit (and all its connections) in the network with probability $1 - p$ (e.g. 0.5) during training.
  - This means, this training sample is trained on a thinned neural network that shares weights with the original network.
  - The network is trained with back-propagation (stochastic gradient descent).
  - At test time, the whole model is used, but the learned weights are multiplied by p.
  - This approximates averaging over all trained ”thinned” networks.

Dropout [9]

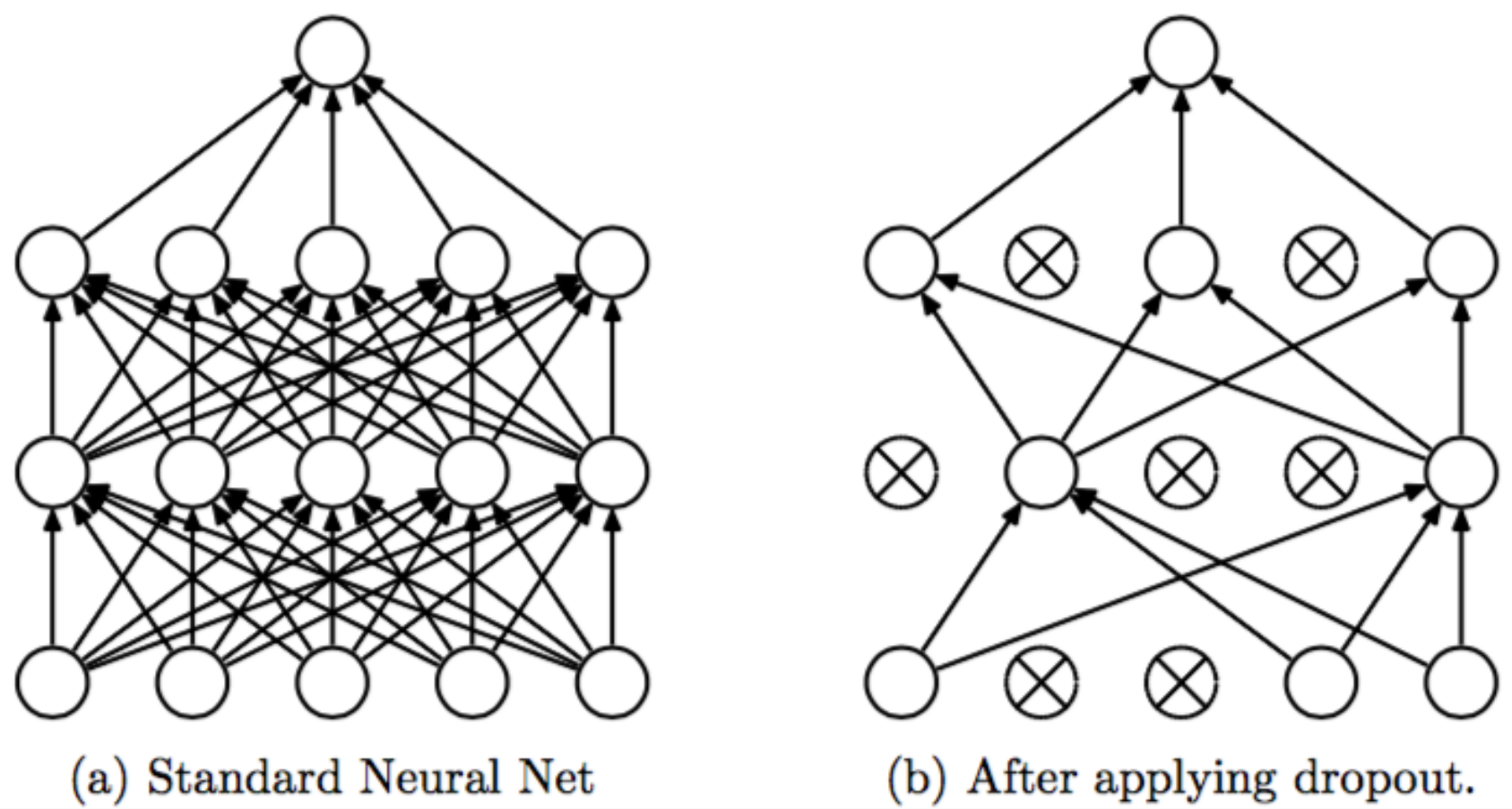


Figure 53: Droping out nodes during training results in thinned networks
(figure taken from [9])

Dropout [9]

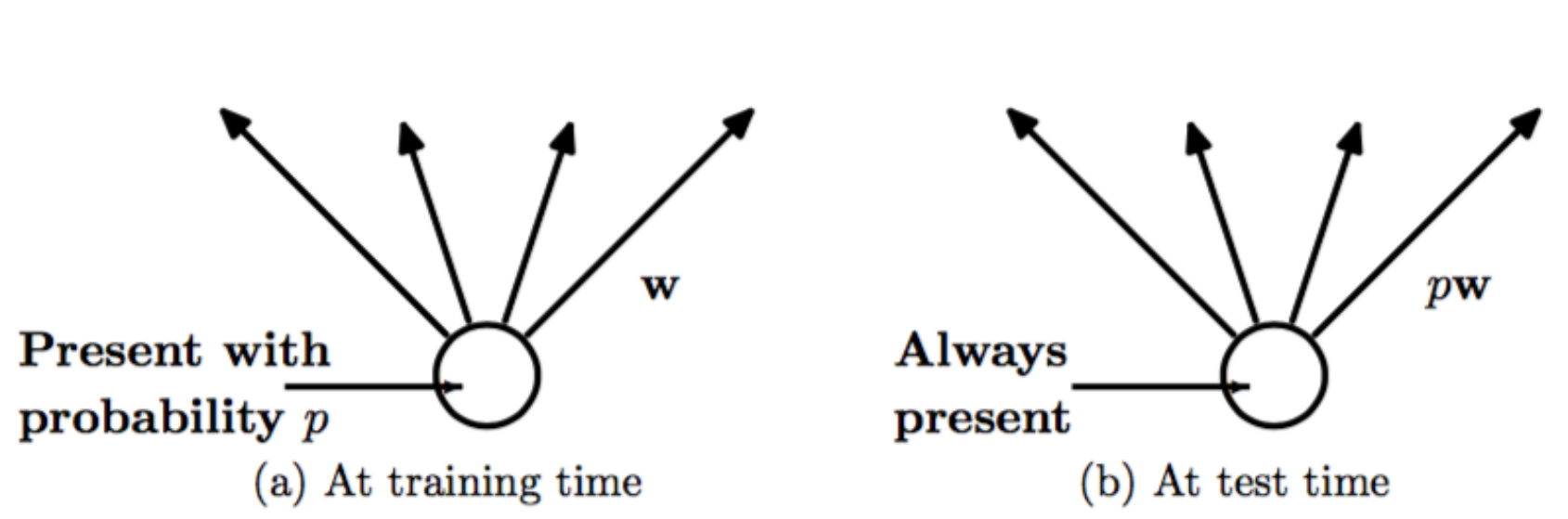


Figure 54: During testing, the full network is used. The weights for edges are multiplied by the probability that the node was present during training. (figure taken from [9])

Special Architectures

Special Architectures

Neural Networks differ in

- number and types of neurons
- types and direction of connections
- number of layers
- learning algorithm (aside from Backprop there is also Hebbian learning for instance)

We review some recent architecture types here. A nice (visual) overview can be found at `http://www.asimovinstitute.org/neural-network-zoo/`.

Feedforward Network (FF)

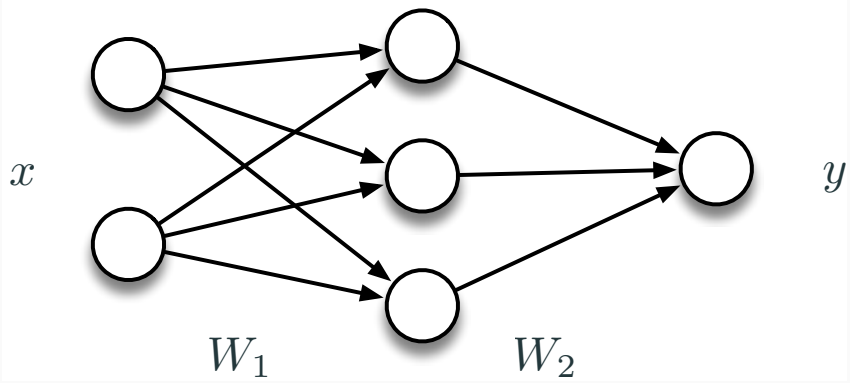


Figure 55: Simple 2 layer feedforward network

- Feedforward connections from input to output layer
- If there is more than 1 hidden layer, the network is called ”deep”
- Each hidden layer corresponds to a transformation of the incoming data, i.e., there are many sequential transformations of the input data.
- Activation function in the hidden layer need to be non-linear (otherwise the computed function from input to output is also just linear).

Recurrent Neural Network (RNN)

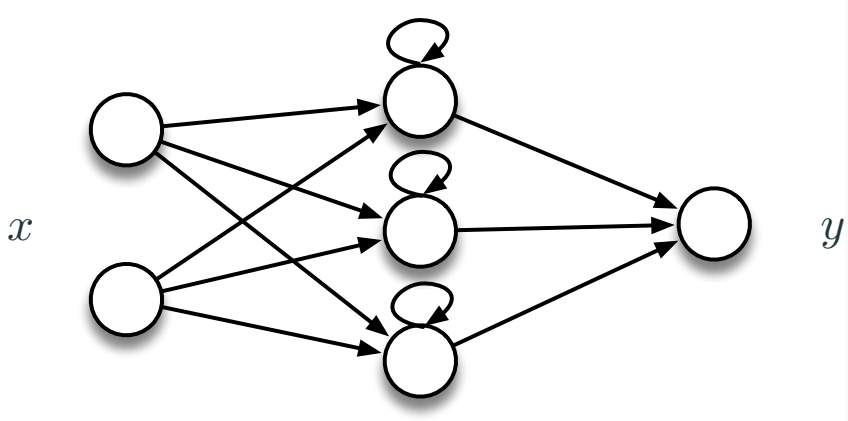


Figure 56: Simple recurrent neural network

- There are cyclic connections in the network.
- Good for modelling sequential data, but very hard to train.
- If hidden connections are recurrent, the network can remember information (about the input sequence).
- Training is done by unrolling the network over time using the back-propagation algorithm (back-propagation through time).

Autoencoders (AE)

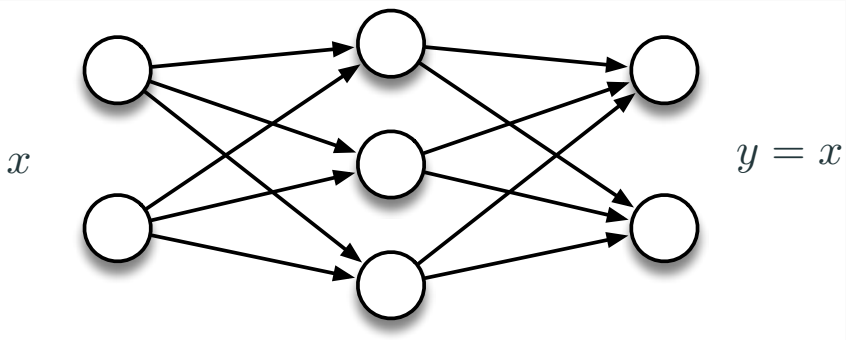


Figure 57: Simple autoencoder

- Architecture is similar to feedforward network.
- During training it is required that $y = x$, i.e., the network is trained to reproduce the input.
- Tied weights ($W_2 = W_1^T$) could be used to reduce the number of parameters to train.
- The hidden layer then corresponds to compressed representation of the data.

Autoencoders (AE)

Variations

- Sparse Autoencoder
  - The number of hidden units is larger than the number of inputs.
  - A sparsity constraint is added to the loss function forcing the network to learn a code dictionary for the data.
- Denoising Autoencoder
  - The number of hidden units is smaller than the number of input units.
  - The input to the network is a distorted version of the data $\tilde{x}$, and the network is trained to produce x.
  - The network learns robust representations of x.

Long short-term memory network (LSTM)

- Are recursive neural networks
- Provide a solution to vanishing gradient problem [5].
- Network contains information outside the normal information flow in special cells (*gated cells*).
- From gated cells information can be read / stored / modified based on the status of gates (output gate / input gate / forget gate).
- Gates are analog (based on multiplication with sigmoids).
- Gates act based on the strength (input) and the importance (weight) of the signal.
- Weights on gates are learned (backpropagation through time), thus, the cell learns when to store, forget or modify its content.

LONG SHORT-TERM MEMORY NETWORK (LSTM)

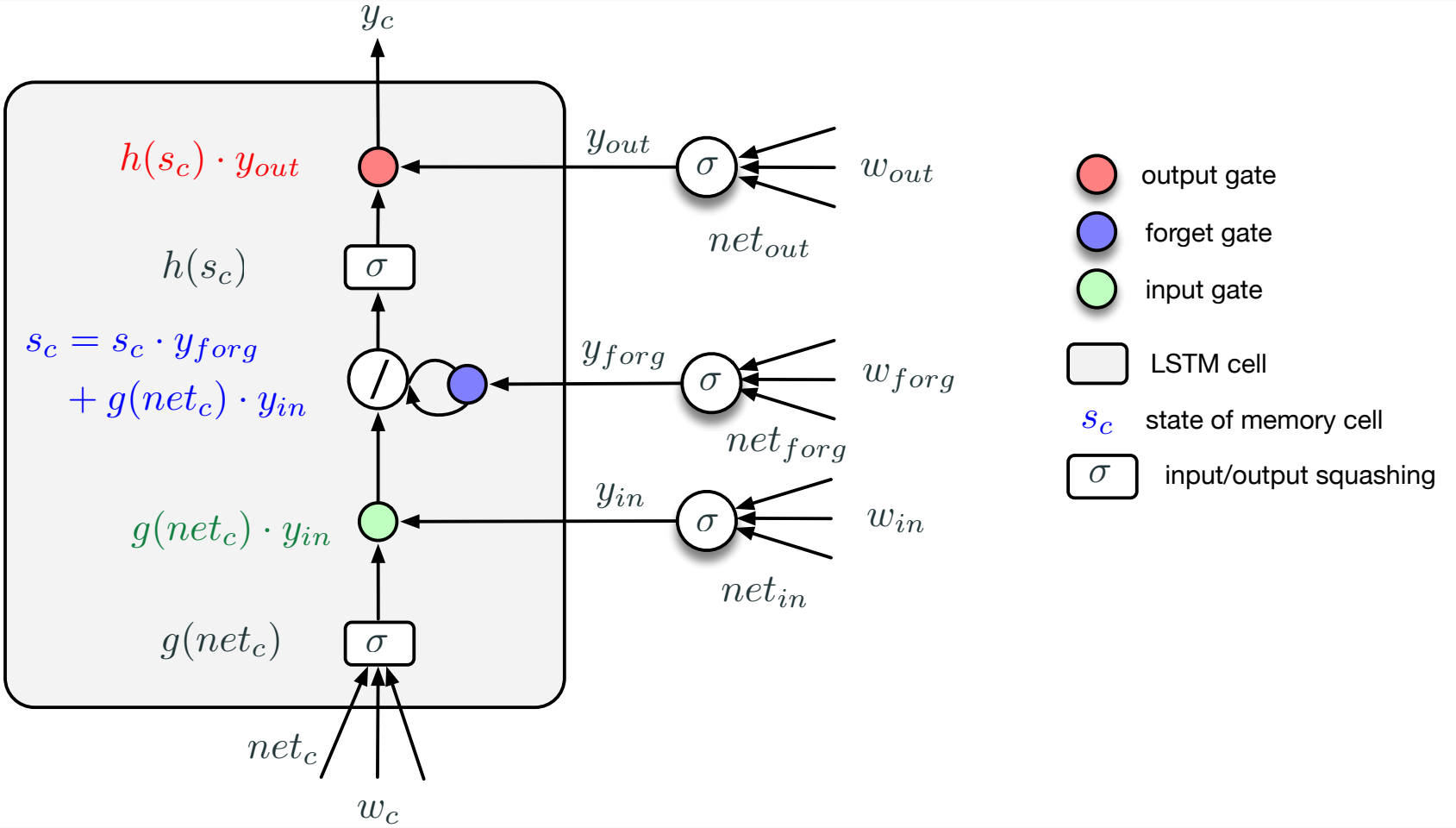
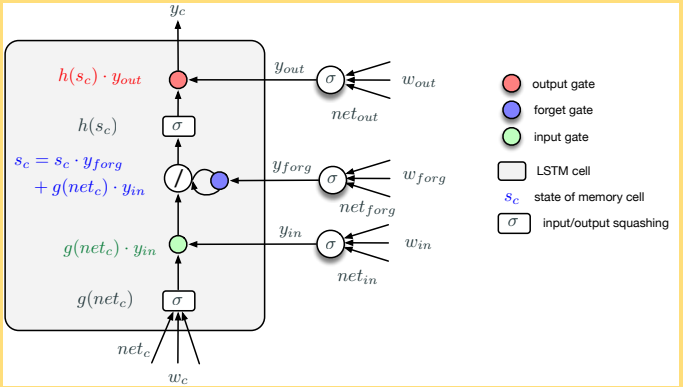


Figure 58: Long short-term memory network

Exercise



What happens in the LSTM cell for the following configurations:

1. $y_{out} = 1$, $y_{forg} = 1$, $y_{in} = 0$
2. $y_{out} = 0$, $y_{forg} = 0$, $y_{in} = 1$
3. $y_{out} = 0$, $y_{forg} = 1$, $y_{in} = 1$
4. $y_{out} = 1$, $y_{forg} = 1$, $y_{in} = 1$?

1. state of cell is read (output)
2. input is stored
3. input is combined with stored value (cell is modified)
4. input is combined with stored value (cell is modified) and presented as ouput

Applications

- Machine Translation (learn a sequence from a sequence) [10]

Type	Sentence
Our model	Ulrich UNK , membre du conseil d' administration du constructeur automobile Audi , affirme qu' il s' agit d' une pratique courante depuis des années pour que les téléphones portables puissent être collectés avant les réunions du conseil d' administration afin qu' ils ne soient pas utilisés comme appareils d' écoute à distance .
Truth	Ulrich Hackenberg , membre du conseil d' administration du constructeur automobile Audi , déclare que la collecte des téléphones portables avant les réunions du conseil , afin qu' ils ne puissent pas être utilisés comme appareils d' écoute à distance , est une pratique courante depuis des années .
Our model	" Les téléphones cellulaires , qui sont vraiment une question , non seulement parce qu' ils pourraient potentiellement causer des interférences avec les appareils de navigation , mais nous savons , selon la FCC , qu' ils pourraient interférer avec les tours de téléphone cellulaire lorsqu' ils sont dans l' air " , dit UNK .
Truth	" Les téléphones portables sont véritablement un problème , non seulement parce qu' ils pourraient éventuellement créer des interférences avec les instruments de navigation , mais parce que nous savons , d' après la FCC , qu' ils pourraient perturber les antennes-relais de téléphonie mobile s' ils sont utilisés à bord " , a déclaré Rosenker .

Figure 59: Example of translation result (Source [10])

Applications

- Handwriting Generation [4]

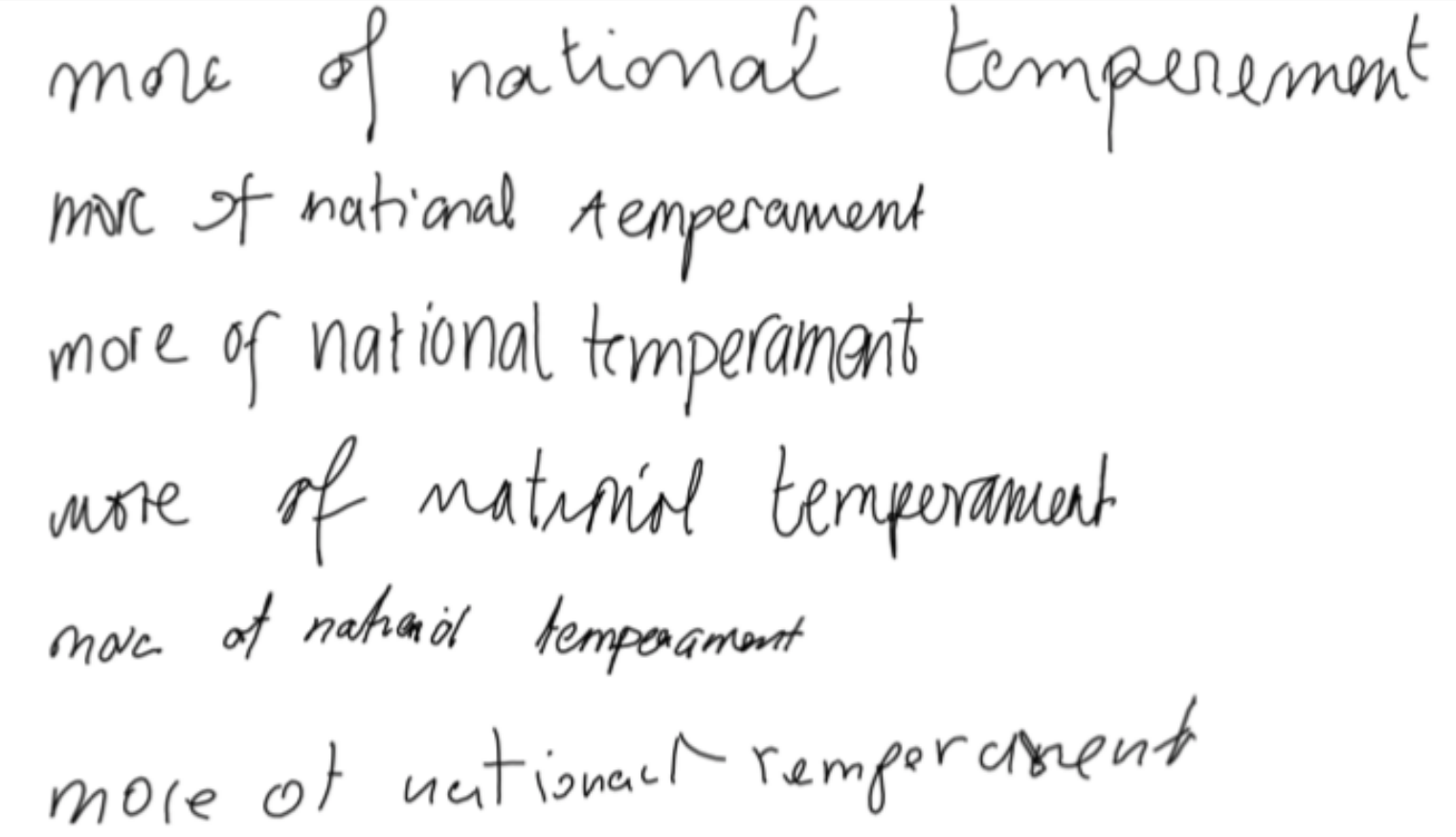


Figure 60: Example of generated handwriting (top example is actual handwriting) (Source [4])

Applications

- Image Captioning [11]

Figure 61: Example output (top example is actual handwriting) (Source [11])

Generative Adversarial Network (GAN)

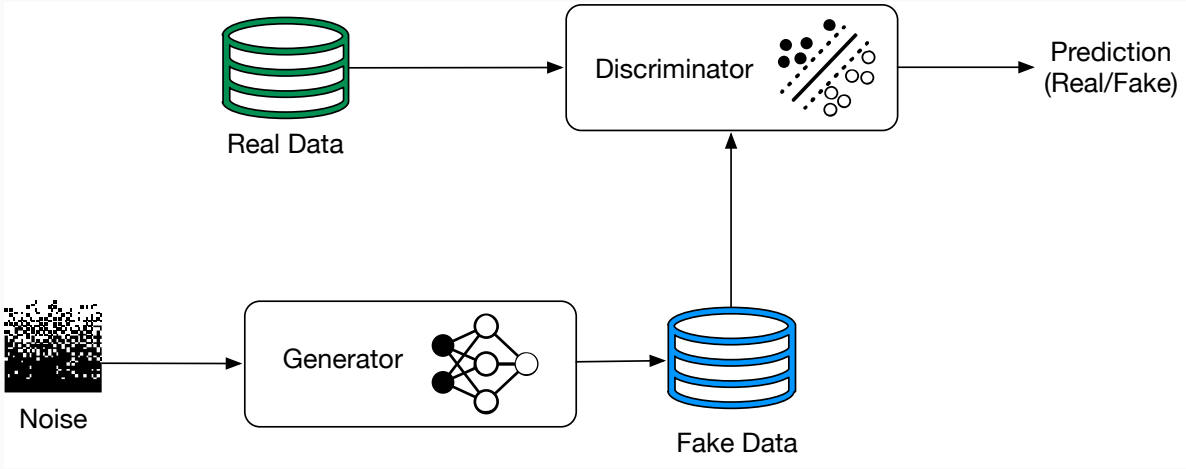


Figure 62: Generative Adversarial Network

- Consists of two networks that compete against each other
- The generator network generates data from a random distribution that looks like real data
- The discriminator network learns to distinguish real and fake
- The better the discriminator gets in distinguishing real from fake data, the better the generator gets to produce fake data and vice versa

Generative Adversarial Network (GAN)

- Generator produces samples $x = g(z, \theta^{(g)})$, with z being the distribution of the random seeds and $\theta^{(g)}$ the parameters of the generator network
- Discriminator produces a probability $p = d(x, \theta^{(d)})$ indicating that a sample belongs to the real distribution
- Formulation as zero-sum game, goal is to find Nash equilibrium
  - Payoff discriminator $v(\theta^{(g)}, \theta^{(d)})$
  - Payoff generator $-v(\theta^{(g)}, \theta^{(d)})$
  - Learning is trying to maximize payoff (each networks wants to gain as much as possible), GAN converges to the optimal generator

$$g^* = \arg\min_g \max_d v(g, d)$$

Generative Adversarial Network (GAN)

- Payoff function:

$$v(\theta^{(g)}, \theta^{(d)}) = \mathrm{E}_{real} \log(p) + \mathrm{E}_{fake}(1 - \log(p))$$

  with *real* being the real data distribution, *fake* the generated fake data distribution and E being the expected value

  - v is large if the probability given by the discriminator for real data is large and for fake data is small (discriminator can distinguish real and fake)
  - v is small if the probability given by the discriminator for fake data is large and for real data is small (discriminator thinks real is fake and vice versa)
- At convergence, the discriminator output $\frac{1}{2}$ for all samples (either real of fake)

Examples

- Predicting the next frame in a video. Left: actual frame of the video, center: model predicting based on mean squared error using the current frame, right: additional GAN that is forced to produce realistic images (cf. ears)

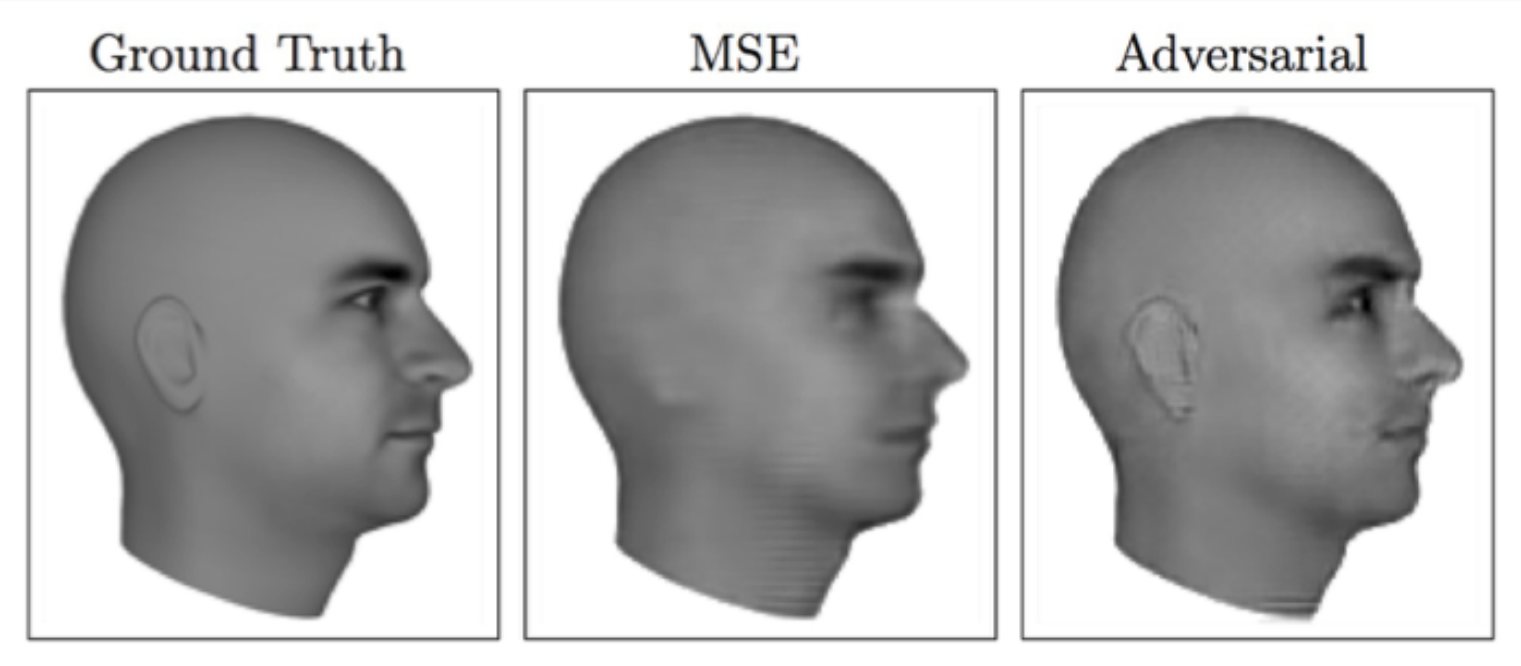


Figure 63: Predicting next frame in a video (Source [2])

Examples

- Generating realistic superresolution images from example images in small resolution

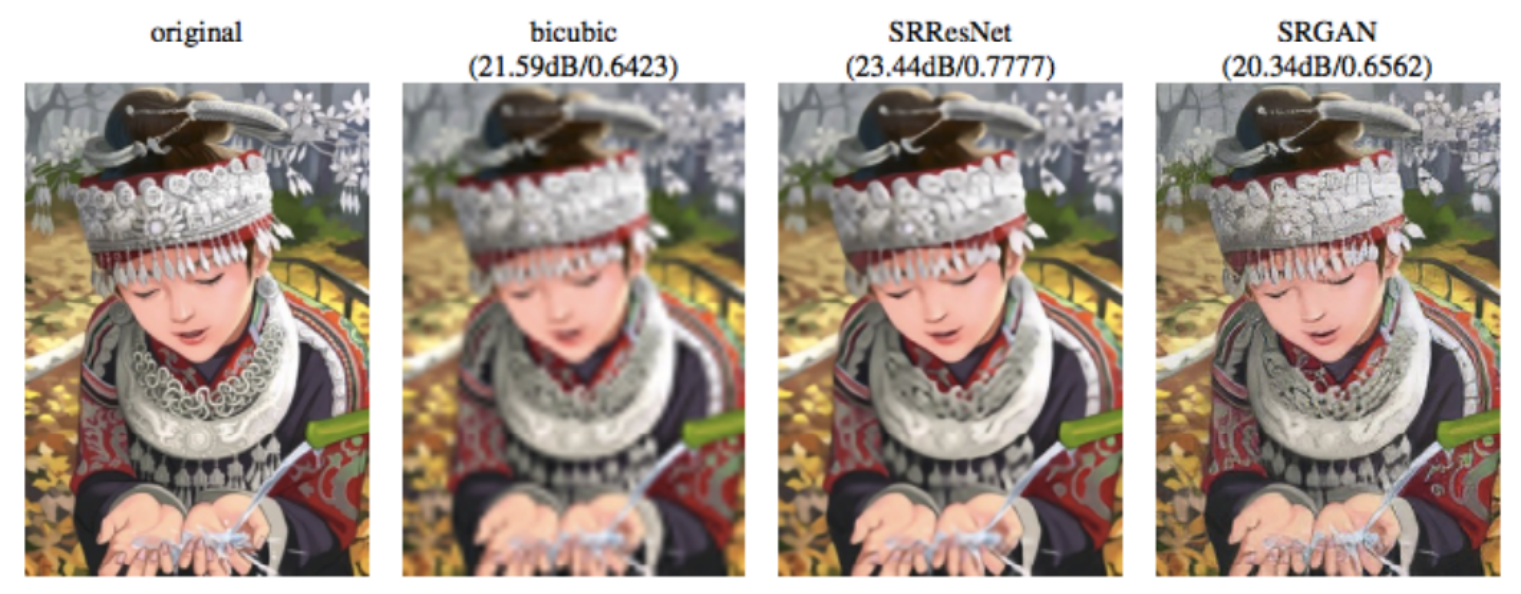


Figure 64: Generating realistic images (Source [2])

Examples

- Impainting of images (fixing holes realistically)

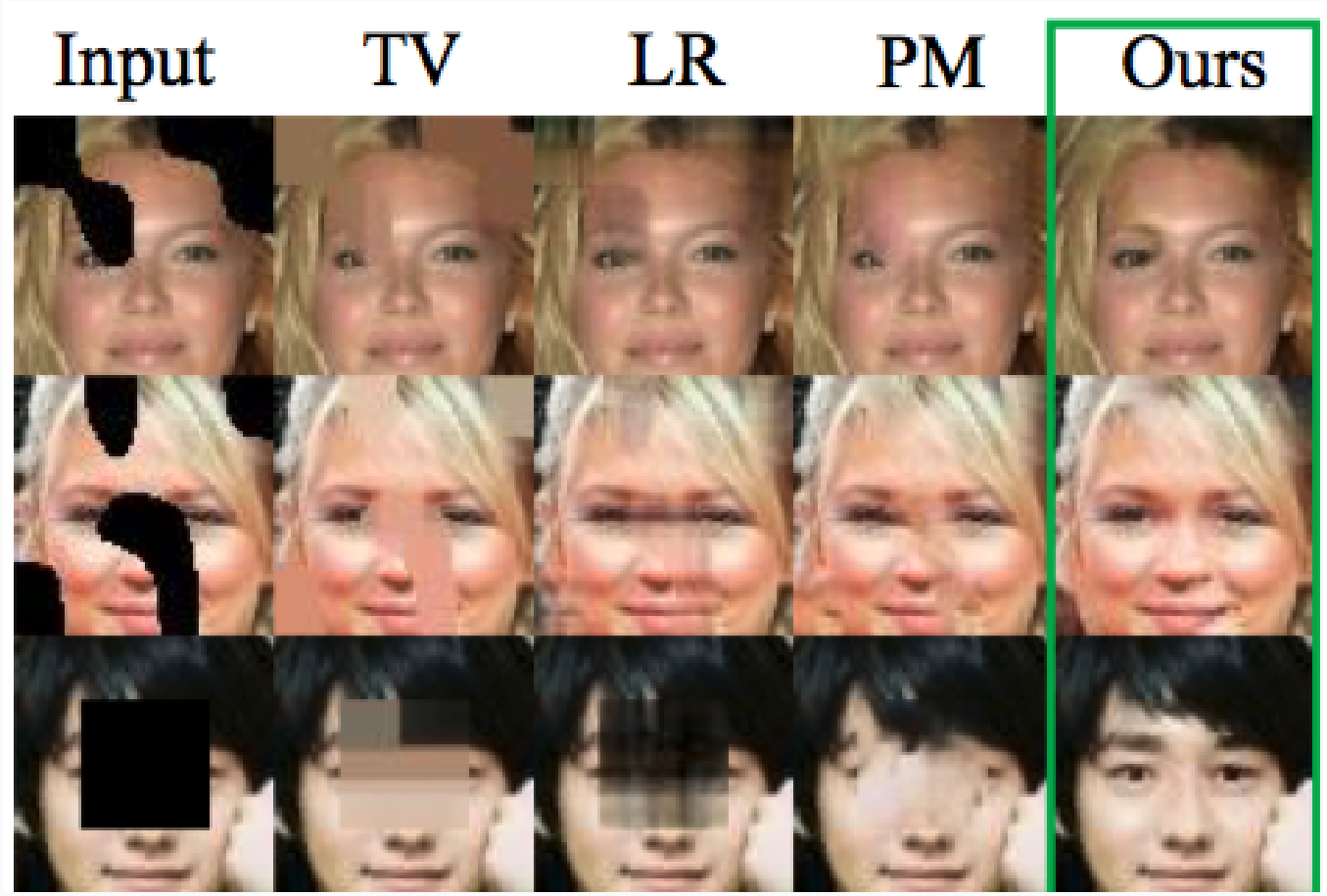


Figure 65: Image impainting (right GAN) (Source [12])

Self-Organising Maps (SOMs)

- Unsupervised learning, the network tries to learn pattern from the data (without a ground-truth)
- Can be used for dimensionality reduction
- Learning algorithm different, no error optimization based on gradients but *competitive learning*
  - when the network is presented with a data sample, all output neurons compete agains each other
  - the winning neuron's weights get updated, such that it would produce a stronger response to the same data sample next time
  - the winning neuron's neighbours also get updated similarly, but to a lower extent
- Self-organizing maps resemble some properties of how learning is done in the brain (and was used to study this)

Self-Organising Maps (SOMs)

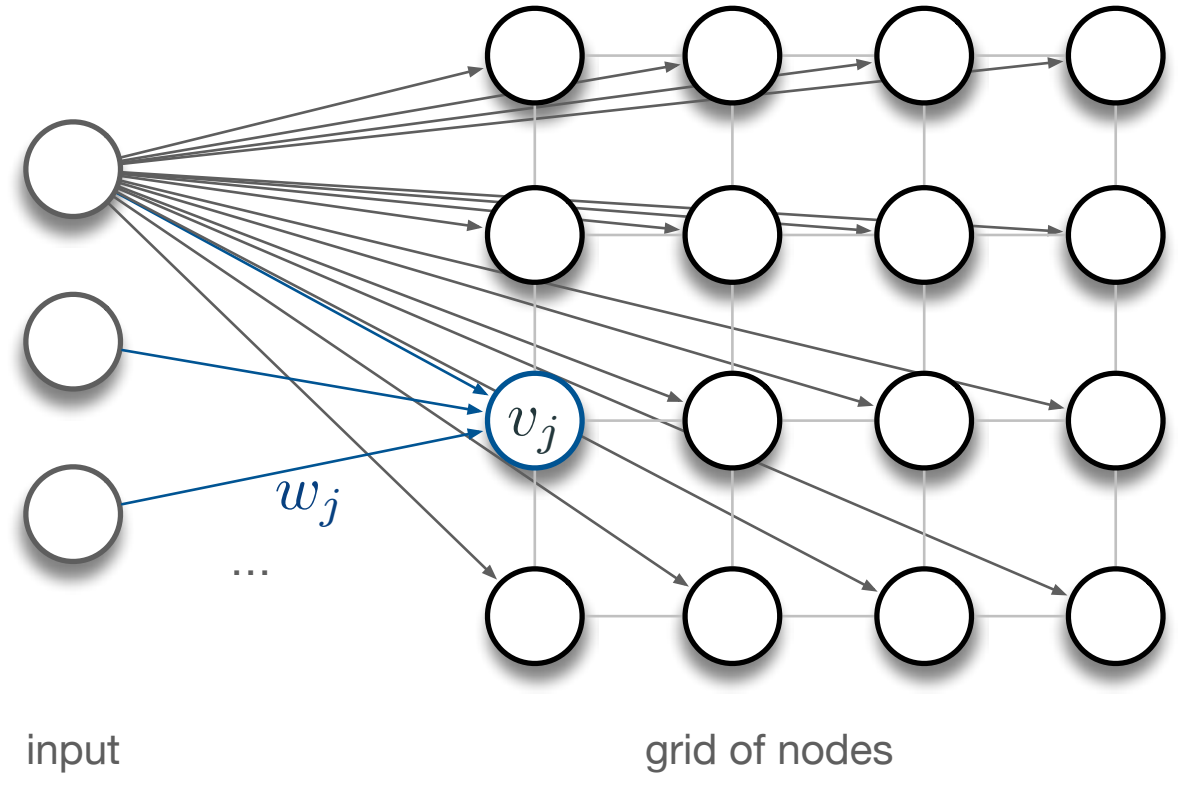


Figure 66: General architecture of a SOM

- output nodes are arranged in a grid, each node has a neighbourhood defined
- all input nodes are fully connected to output nodes

Self-Organising Maps (SOMs)

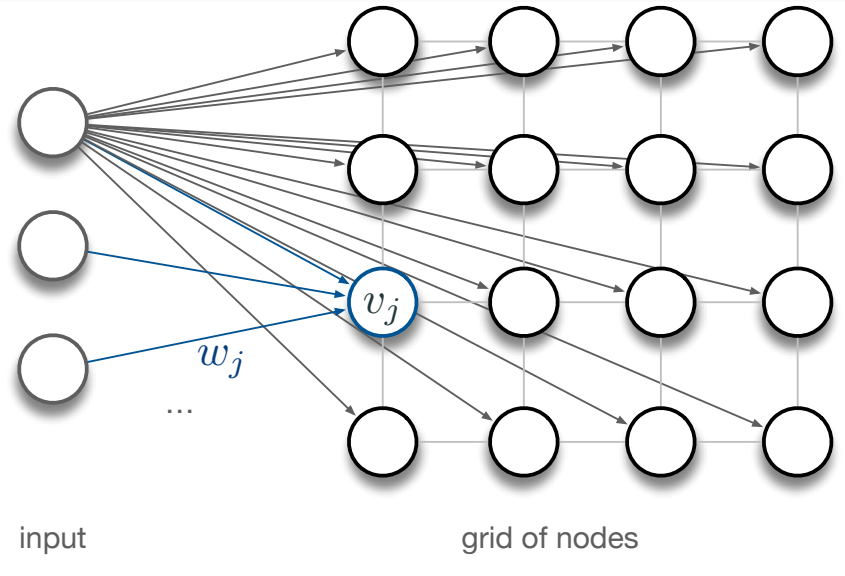


Figure 67: General architecture of a SOM

Notation:

- input data $x_i^k \in \mathcal{R}^m$ is the k-th data point
- v_j is the j-th output node
- w_j are the weights associated with node v_j

General Idea of Learning in SOMs

1. Initialize all weights randomly (**Initialization**)
2. Determine the neuron that is most similar to the current input vector (according to some similarity measure), this is the winning neuron (**Competition**)
3. Determine the neighborhood of the winning neuron (**Cooperation**)
4. Make the weights of the winning neuron and – to a lesser extent – its neighbours more similar to the current input, such that it will get even more activated when seeing the same data point again (**Update**)

Competition

- A similarity function defines how similar the input vector is to each node
- For instance, the Euclidean distance can be used

$$d(x_i, v_j) = \sum_{k=1}^{m} (x_i^k - w_{kj})^2$$

  with x_i^k being the k-th coordinate of sample i
- The winning neuron v_* is the one with the lowest Euclidean distance to x_i

$$v_* = argmin_j(d(x_i, v_j))$$

Cooperation

- A neighborhood function $N_{v_* u}$ defines the influence of the update on the neighbors u of the winning node v_*
- Simple choices are the Moore or von-Neumann neighborhoods, $N_{v_* u} = 1$ for all neighbors of v_* and $N_{v_* u} = 0$ for all other nodes
- Other choice is Gaussian neihborhood

$$N_{v_* u} = e^{\frac{-s_{v_*, u}}{2\sigma^2}}$$

  with σ being the variance and $s_{v_*, u}$ the grid distance between nodes u and v_*
- The neighborhood should get smaller over time, thus the neihborhood function is a function of time; for Gaussian σ can be made to decrease over time

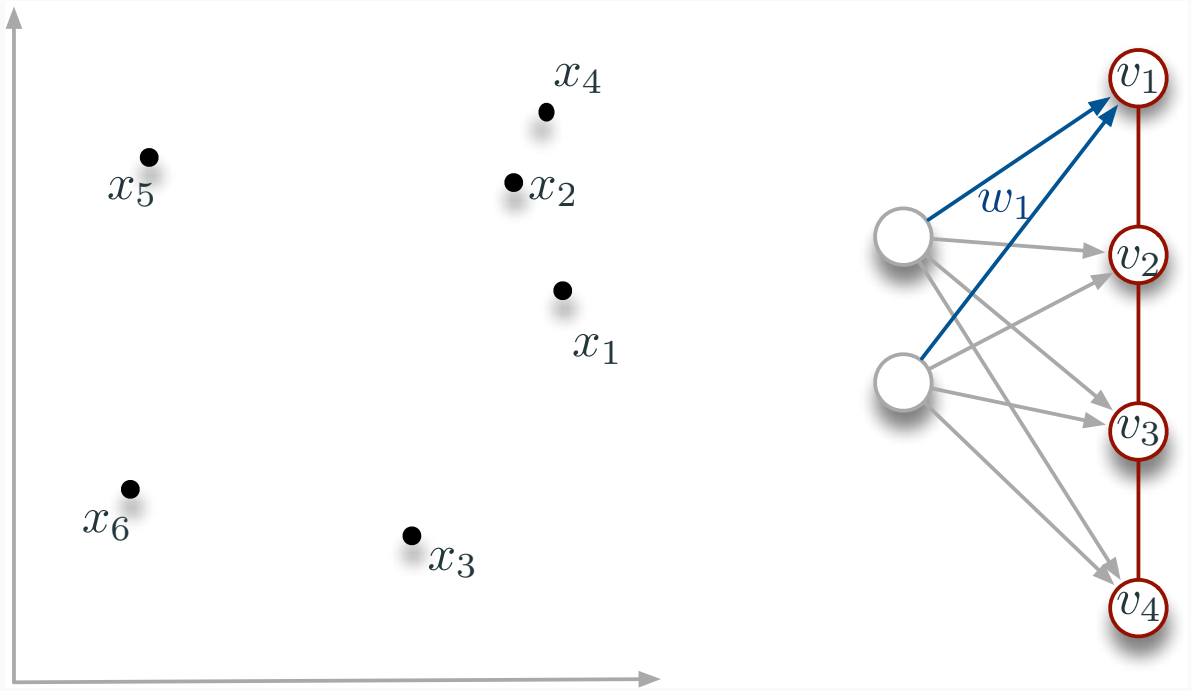
Update

- Using a neighborhood, the network only learns locally
- Update rule

$$\Delta w_{ij} = \eta(t) \cdot N_{v_* u}(t) \cdot (x_i - w_j)$$

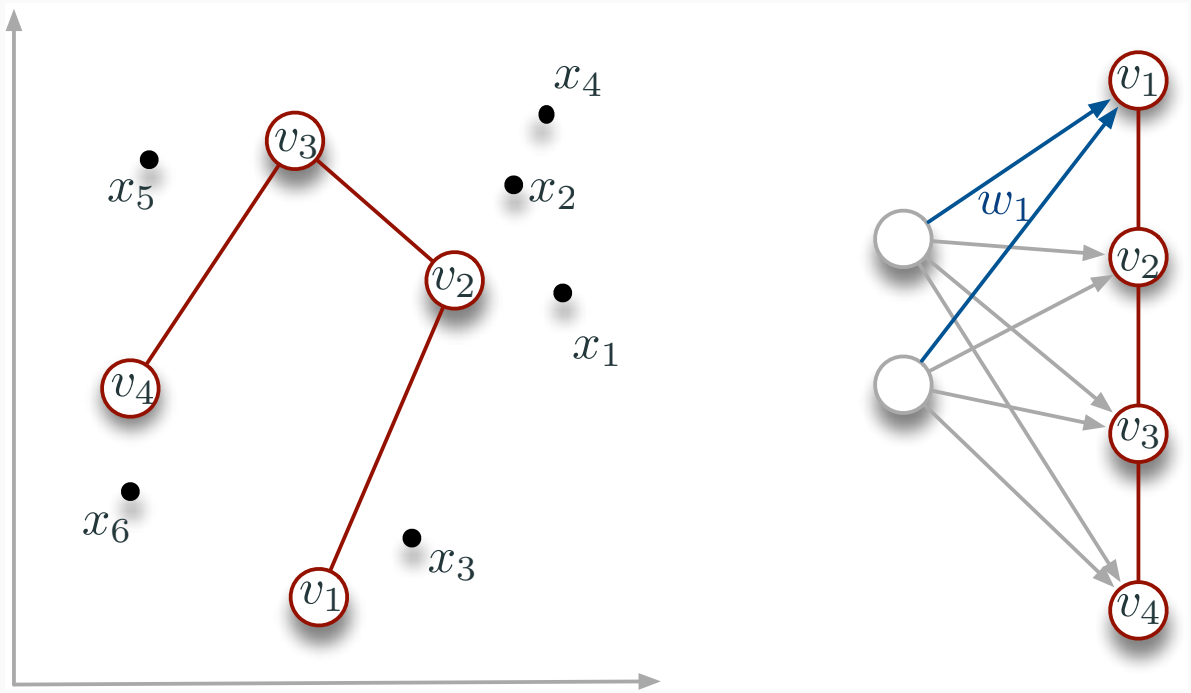
- The learning rate $\eta(t)$ decreases over time
- The neighborhood function $N_{v_* u}(t)$ depends on time, becoming more local
- The term $(x_i - w_j)$ arises from the derivative of the distance function (we wanted to minimize the distance between input and a node, which is represented by its weights)

Example



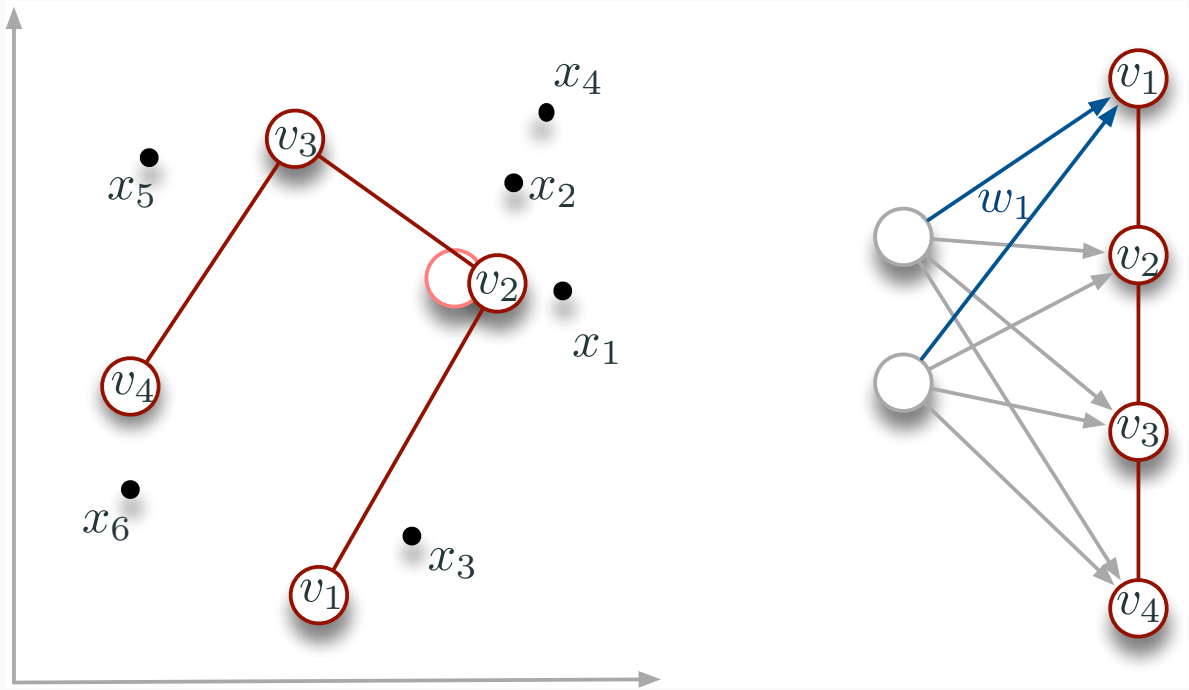
- Six data points x_i in 2-dimensional space
- SOM with four nodes v_j
- Input to SOM are coordinates of data points

Example



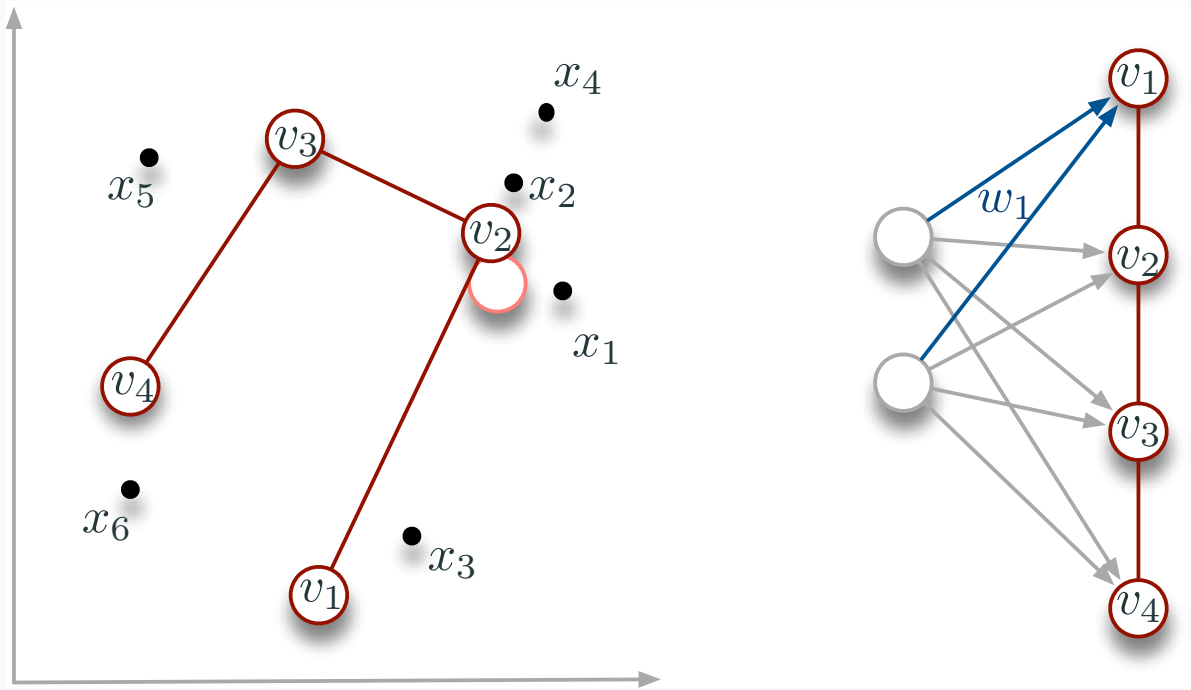
- After initialization of weights
- Each node v_j can be represented in feature space using its weights w_j

Example



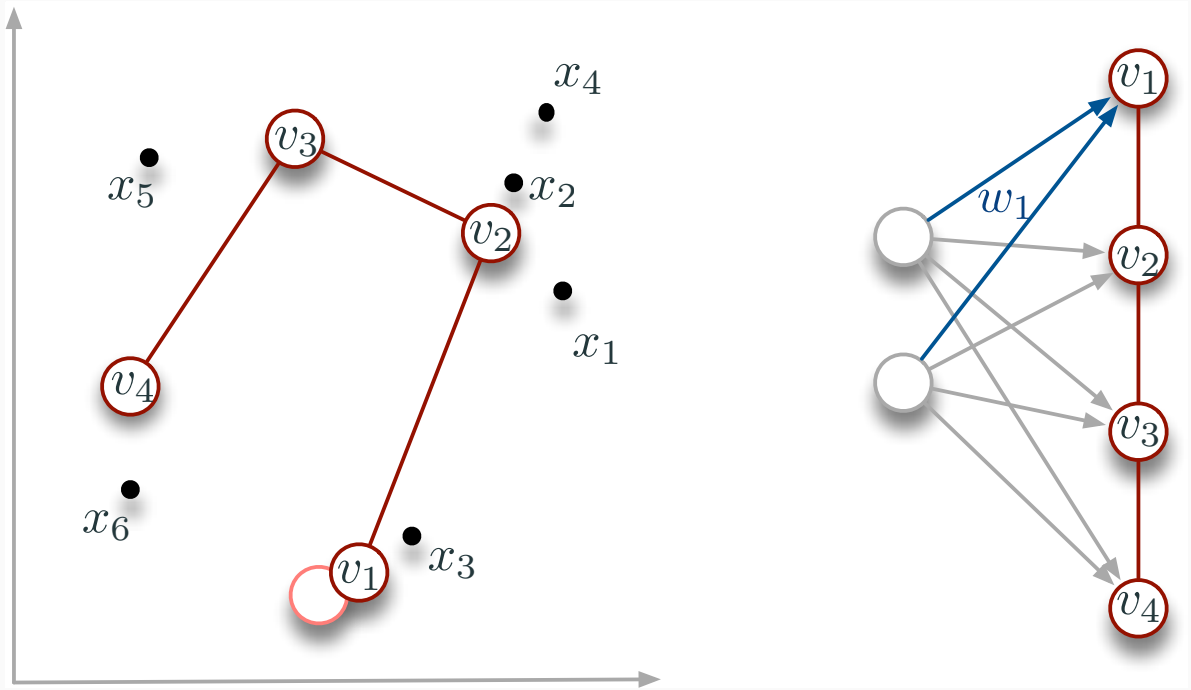
- Input of x_1
- Wining neuron is v_2
- v_2's weights w_2 get updated, v_2 moves closer to x_1 (for simplicity update of neighborhood is ignored)

Example



- Input of x_2
- Wining neuron is v_2
- v_2's weights w_2 get updated, v_2 moves closer to x_2 (for simplicity update of neighborhood is ignored)

Example



- Input of x_3
- Wining neuron is v_1
- v_1's weights w_1 get updated, v_1 moves closer to x_3 (for simplicity update of neighborhood is ignored)

Self-Organising Maps (SOMs)

- The data distribution's shape (grey dots) is well approximated by a SOM (red dots)
- For dimensionality reduction each data point can be represented by its nearest SOM node (the winner)

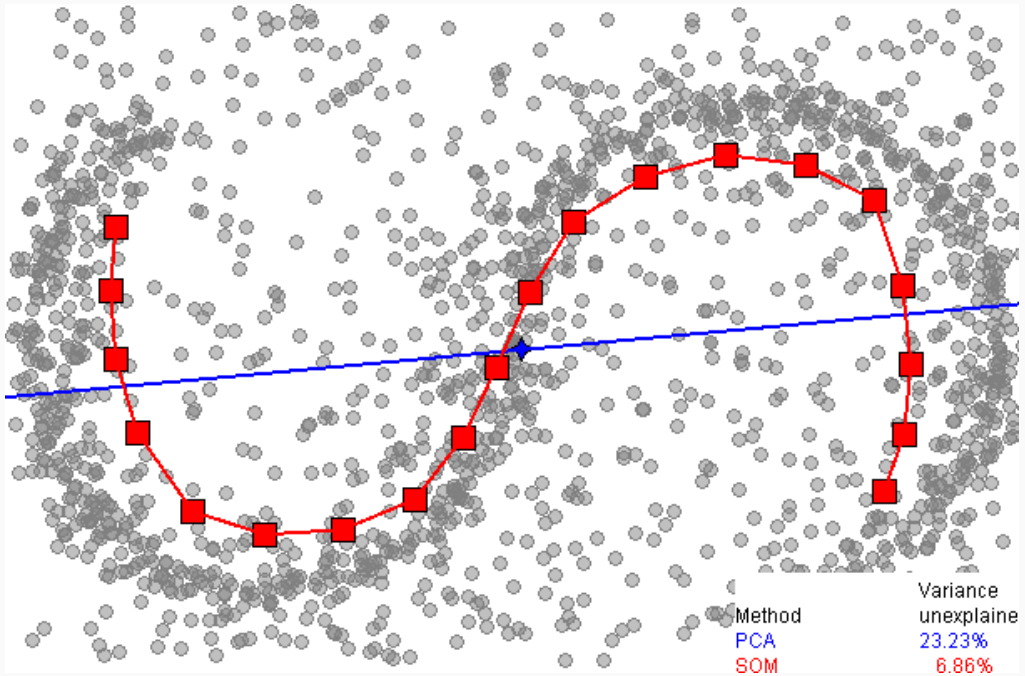


Figure 68: Dimensionality reduction with SOMs (CC-SA 3.0, Agor153, via Wikimedia Commons)

Summary

Summary

- Neural Networks with many hidden layers (Deep Neural Networks) trained on a lot of training data are capable of achieving near human performance on certain tasks (e.g. object recognition).
- Backpropagation is the de-facto standard training algorithm, implementing a gradient descent on the error surface.
- Neural networks differ in
  - their architecture (types of neurons, number of layers, structure of connections)
  - the number of hyperparameters
  - training method (algorithm, training parameters, data set)

Summary

Important Concepts

- Neuron
- Supervised, unsupervised, semisupervised learning
- Parameters, hyperparameters
- Perceptron
- Back-Propagation
- Feedforward and Recurrent Neural Networks and derivations

Acknowledgement

- Some slides (perceptron training) are adapted from the course on Machine Learning and Data Mining at the Bauhaus University Weimar (Prof. Dr. Benno Stein).
  - Overview: `webis.de/lecturenotes/overview/overview.html`
  - Slides: `webis.de/lecturenotes/slides/slides.html`

Further Readings

- Machine Learning (general ML introductory book)
  Thomas M. Mitchell. *Machine Learning*. 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997. ISBN: 0070428077, 9780070428072
- Deep Learning
  Chapter available online `http://www.deeplearningbook.org`
  Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016
- Neural Networks and Deep Learning (online book)
  `http://neuralnetworksanddeeplearning.com/`

References

[1] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. “A Neural Algorithm of Artistic Style”. In: *CoRR* abs/1508.06576 (2015). URL: http://arxiv.org/abs/1508.06576.

[2] Ian J. Goodfellow. “NIPS 2016 Tutorial: Generative Adversarial Networks”. In: *CoRR* abs/1701.00160 (2017). arXiv: 1701.00160. URL: http://arxiv.org/abs/1701.00160.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[4] Alex Graves. “Generating Sequences With Recurrent Neural Networks”. In: *CoRR* abs/1308.0850 (2013). URL: http://arxiv.org/abs/1308.0850.

[5] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

References II

[6] Sergey Levine et al. “Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection”. In: *CoRR* abs/1603.02199 (2016). URL: http://arxiv.org/abs/1603.02199.

[7] Thomas M. Mitchell. *Machine Learning*. 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997. ISBN: 0070428077, 9780070428072.

[8] David Silver et al. “Mastering the Game of Go with Deep Neural Networks and Tree Search”. In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961.

[9] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[10] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. “Sequence to Sequence Learning with Neural Networks”. In: *CoRR* abs/1409.3215 (2014). arXiv: 1409.3215. URL: http://arxiv.org/abs/1409.3215.

[11] Oriol Vinyals et al. “Show and Tell: A Neural Image Caption Generator”. In: *CoRR* abs/1411.4555 (2014). arXiv: 1411.4555. URL: http://arxiv.org/abs/1411.4555.

[12] Raymond A. Yeh et al. “Semantic Image Inpainting with Perceptual and Contextual Losses”. In: *CoRR* abs/1607.07539 (2016). arXiv: 1607.07539. URL: http://arxiv.org/abs/1607.07539.

References III

[13] Matthew D. Zeiler and Rob Fergus. “Visualizing and Understanding Convolutional Networks”. In: *Computer Vision – ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I*. Ed. by David Fleet et al. Cham: Springer International Publishing, 2014, pp. 818–833. ISBN: 978-3-319-10590-1. DOI: 10.1007/978-3-319-10590-1_53. URL: https://doi.org/10.1007/978-3-319-10590-1_53.