TEST FOR DEV: Introduction to Data Science

WS 23/24, RWTH Aachen

February 2, 2024

1 Naive Bayesian Classification

Next, we're gonna look at a classification based on stochastical formulas developed by Thomas Bayes (1701-1761, English statistician and philosopher). More concretely, the classification is based on Bayes' theorem, often used in probability-based learning.

First, we'll lay down some basic statistics:

$$
\begin{aligned}
A, \ldots, Z &: \text{Some event (can either be true or false)} \\
P(X) &: \text{Probability that } X \text{ holds} \\
P(X, Y) &: \text{Probability that both } X \text{ and } Y \text{ hold} \\
P(X|Y) &: \text{Probability that } X \text{ holds given } Y \text{ (conditional probability)} \\
P(\neg X) &= 1 - P(X) \\
P(\neg X|Y) &= 1 - P(X|Y) \\
P(Y) &= P(Y|X)P(X) + P(Y|\neg X)P(\neg X)
\end{aligned}
$$

Further, we have the product and chain rule:

$$
\begin{aligned}
P(X, Y) &= P(X|Y) \cdot P(Y) \\
P(A, B, \ldots, Y, Z) &= P(A, B, \ldots, Y|Z) \cdot P(Z) = P(A|B, \ldots, Y, Z) \cdots P(Y|Z) \cdot P(Z)
\end{aligned}
$$

If X and Y are independent, so $P(X|Y) = P(X)$, it further holds[1]:

$$
P(X, Y) = P(X) \cdot P(Y)
$$

And finally, we have **Bayes' theorem**, and so the generalized one, which can be simply derived from the product rule: Bayes' Theorem

$$
P(Y|X) = \frac{P(X|Y) \cdot P(Y)}{P(X)} P(Y = y|x_1, \ldots, x_m) \quad = \frac{P(x_1, \ldots, x_m|Y = y) \cdot P(Y = y)}{P(x_1, \ldots, x_m)}
$$

We can now apply the theorem to classification, by setting X as the event that the target feature has a specific value and Y that the descriptive features have specific values. The

[1] Extendable for any number of events

probabilities can then be estimated trivially. Formally, we have the Bayesian **maximize a posteriori (MAP)** prediction model:

MAP

$$\begin{aligned}\mathbb{M}_{MAP}(\vec{\mathbf{x}}) &= \arg\max_{\vec{\mathbf{y}}} P[y|x_1,\ldots,x_m] \\ &= \arg\max_{\vec{\mathbf{y}}} \frac{P(x_1,\ldots,x_m|y)\cdot P(y)}{P(x_1,\ldots,x_m)}\end{aligned}$$

Or without normalization:

$$\mathbb{M}_{MAP}(\vec{\mathbf{x}}) = \arg\max_{\vec{\mathbf{y}}} P(x_1,\ldots,x_m|y)\cdot P(y)$$

To avoid overfitting, and solve some computational problems due to the curse of dimensionality, we can assume independence, which then results in the **Naive Bayes' Classifier**:

Naive Bayes' Classifier

$$\mathbb{M}(\vec{\mathbf{x}}) = \arg\max_{\vec{\mathbf{y}}} \left(\prod_{i=1}^{m} P[x_i|y]\right)\cdot P(y)$$

- Many combinations of features don't appear in the training data, but we can't conclude that these cannot happen.
- We can use this classifier, to approximate them.

2 Neural networks

So far, we have looked at different **supervised learning** techniques:

- Decision trees (first categorical, then numerical)
- Regression (first numerical, then categorical)
- Support vector machines (SMV)
- Naive Bayesian classifier

All those techniques have the following things in common:

- They try to learn a **function predicting** a **target** feature in terms of its descriptive features.

$$f(\underbrace{\vec{\mathbf{x}}}_{\text{descriptive features}}) := \underbrace{\vec{\mathbf{y}}}_{\text{target feature}}$$

- Some way of measuring the **error** is provided, e.g. the sum of squared errors, or the number of misclassifications.

$$m_{error} : \{(\underbrace{f(\vec{\mathbf{x}}_i)}_{\text{predicted label}}, \underbrace{\vec{y}_i}_{\text{correct label}})\} \mapsto d \in \mathbb{R}$$

- Learning is now based on **training data**. The evaluation then requires **test data** (unseen) to address the problem of overfitting.

$$\mathcal{D}_{train}, \mathcal{D}_{test} \subseteq \{(\underbrace{\vec{\mathbf{x}}_i}_{\text{input}}, \underbrace{\vec{y}_i}_{\text{correct label}})\}, \quad \mathcal{D}_{train} \cap \mathcal{D}_{test} = \emptyset$$

For **classification** and specifically linear regression and SVMs, we derived the following formula:

$$y_{\vec{\mathbf{w}}}(\vec{\mathbf{x}}) = f\left(\sum_{j=1}^{M} \vec{\mathbf{w}}_j \phi_j(\vec{\mathbf{x}})\right)$$

- $\vec{\mathbf{x}}$ is the input vector containing the descriptive features
- $\vec{\mathbf{w}}$ is our weights vector
- The input vector can be lifted to a higher dimension by a mapping ϕ:
  - Simple linear classification: $\vec{\mathbf{x}}$ can be used directly
  - Nonlinear classification: apply feature space mapping or the "kernel trick"
    $\rightarrow$ requires lots of domain knowledge
    $\rightarrow$ increases the number of features M
- Finally, we see the actual categorizer f, which is either:
  - A simple threshold-based function with returning 0 or 1, or
  - A more sophisticated function like the sigmoid function (is used in logistic regression)

We can abstract that further to the following image:

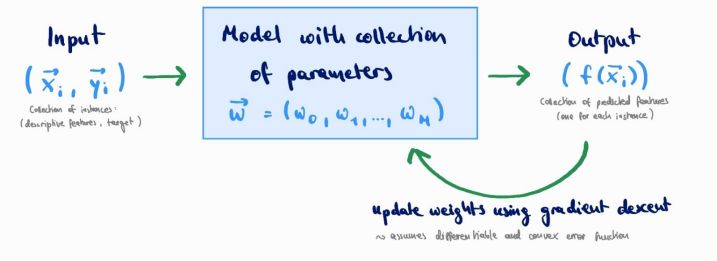


Figure 2.1: Abstraction on supervised learning

- **Gradient descent** is a basic principle to iteratively reduce the error by walking down the hill in the steepest direction.
- Important is the choice of the step size. If it's too small, the convergence is slow. If it is too large, a risk of overshooting or even divergence arises.

Now, the topic of this chapter comes into play: why do we need **neural networks**?

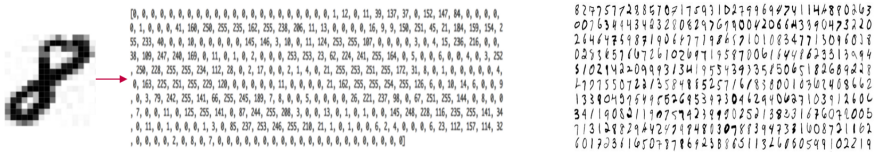
- The input for our classification problem can be anything, e.g. text, sound, images, videos, etc.
- This would mean, our classifier can no longer be described by a simple (linear) function.
- Our traditional linear approaches don't work anymore. SVM already tried to address this problem by introducing the mapping $\phi(\vec{\mathbf{x}})$ to lift the number of dimensions, but this mapping needed to be manually constructed.
- In the NN approach: $\phi(\vec{\mathbf{x}})$ may be based on other layers and can be learned.

Our before-seen function is therefore lifted to the following:

$$y_{\vec{\mathbf{w}}}(\vec{\mathbf{x}})[k] = f\left(\sum_{j=1}^{M} w_{jk}^{(2)} \underbrace{h\left(\sum_{i=1}^{D} w_{ij}^{(1)} x_i + w_{0j}^{(1)}\right)}_{\text{"}\phi_j(\vec{\mathbf{x}})\text{" is now a network}} + w_{0k}^{(2)}\right)$$

- We have input $\vec{\mathbf{x}}$ with D dimensions or features.
- The network uses the activation functions f and h.

Consider the classification of images with automatic dog detection. For that, many labeled samples (dog and non-dogs) are provided. The samples are pictures, so for the computer just a collection of pixels. This results in a huge number of features. The same can be seen for the example in 2.2. Generally, as the input for our NNs, we're only gonna consider **unstructured data** such as images, text, sound, or video.



Set of features | Training dataset

$$\text{Image} = \text{grid of numbers representing the darkness of each pixel (grayscale)}$$

$$\text{Numbers for MNIST are stored as } 18 \times 18 \text{ pixel images} \implies \text{Array of } 324 \text{ numbers}$$

Figure 2.2: MNIST dataset example (number recognition classifier)

After we saw the intuition of why we need NNs, what they roughly are, and some example applications, we'll now take a look at what an Artificial NN really is.

- Generally speaking, it is a computing system inspired by, but not identical to, biological NNs.
  - Since there are many differences between real NNs and artificial ones, the terms are sometimes criticized.
- It learns to perform tasks by considering examples without being explicitly programmed.
- Artificial NNs are collections of connected artificial neurons.
  - The connections correspond to weights that can be updated to make the error on the training data smaller.

2.1 Historical background

Throughout the history of developing NNs (see 2.3), we can see two paradigms for AI:

1. Logic-inspired based on **reasoning** "Good old-fashioned AI"
   - Symbolic rules over symbolic expressions

- Programmed using unambiguous language

⇝ 1943 – 2005: beat NNs with various other methods

2. Biologically-inspired based on **learning**
   - Large vectors representing neural activities
   - Vectors learned from data

   ⇝ 2005 – 2010: NNs show more promising results (improvements in backpropagation, more training material, computational power)

   ⇝ > 2010 : major investments (Google, IBM, Apple etc. for Alexa, Siri, autonomous driving, ...)

   ⇝ >2018: high on political agenda

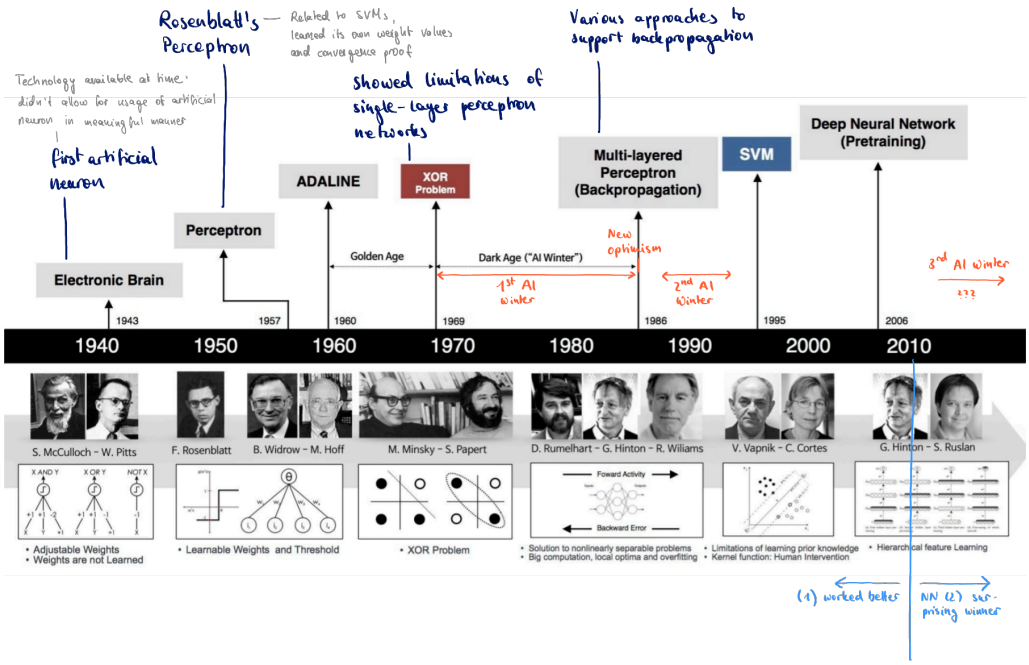


Figure 2.3: History of Neural Networks

In 2018, the Turing Award was given for the development of a convolutional NN, concretely the conceptual and engineering breakthroughs making deep neural networks a critical component of computing. Still, for a NN to work successfully, a lot of engineering and trial-and-error is necessary.

Pro and Con of NNs

- The main advantages of ANNs:
  - NNs are best at identifying patterns or trends, so they are well suited for:
    - Sales forecasting, logistics, customer behavior, security, medicine
    - Specific tasks: face recognition, traffic sign classification, sentiment analysis, diagnosis of hepatitis, speech recognition, hand-written text recognition, computer vision, pattern recognition, etc.
  - Can model complex (non-linear) functions
  - Generic and flexible, driven by data
  - Good performance on unseen noisy data
  - Can handle images, sound, text, video, etc.

  - After the model is learned: can be applied fast
- Main disadvantages:
  - Only works with lots of training examples
  - Time- and resource-consuming
  - Non-transparent (black-box, interpretation of hidden layers is difficult → keyword explainable AI)
    * What the Hidden layers are doing is feature extraction. However due to the nonlinearity, it can be hard to interpret what feature they're extracting,
  - Risk of overfitting
    * Large number of parameters allows more complex functions (with enough parameters, you can fit any data set exactly)
    * But they require lots of training data and are drawn to overfitting the model
    * So they "learn" the training data by heart, without abstracting
  - Performs worse on well-defined problems
  - Can be "hacked" (add noise, switch one pixel → wrong classification, etc.)

2.2 Human and Artificial Neurons

Now that we know some of the historical developments, we'll take a look at how the ideas to develop artificial NNs came up.

The "original" biological inspiration is the **Human brain** with around 100 billion neurons. The human brain learns in the way displayed in 2.4.

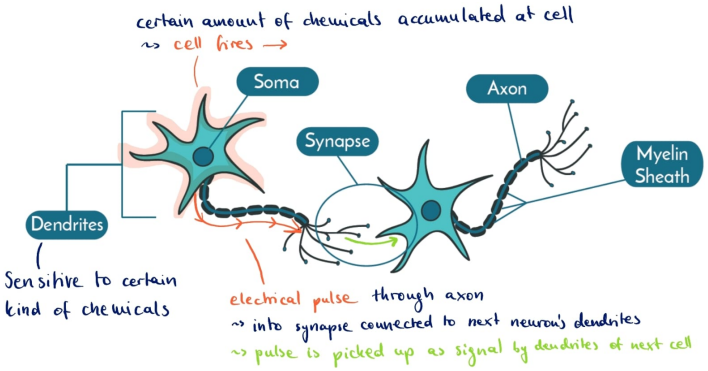


Figure 2.4: Biological neuron (learning)

So on a more abstract level: when a neuron receives excitatory input, learning occurs.

- The input needs to be sufficiently large compared to its inhibitory input, only then is a spike of electrical activity sent down the axon.
- Learning is then the change of the effectiveness of the synapses, and/or the influence of one neuron on other changes.

Single-layer/Rosenblatt's perceptron

This abstraction can be reduced to this simple (artificial) neuron shown in 2.5. This is also known as a **single-layer perceptron** which is the simplest feedforward neural

network and only works for binary classification.

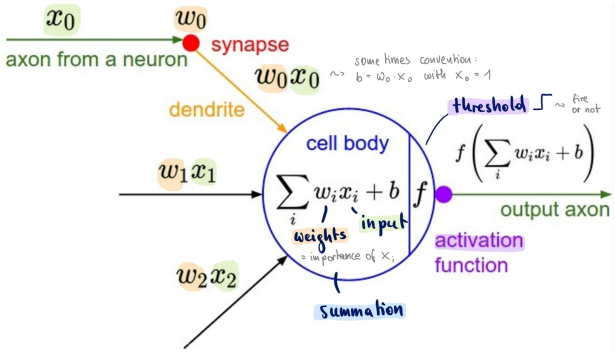


Figure 2.5: Simple artificial neuron (Rosenblatt's perceptron)

The **learning process** of this simple neuron is now the following:

- Randomly assign weights $w_i \in [0, 1]$
- Present inputs from training data $\vec{\mathbf{x}}$
- Get the output $\vec{\mathbf{y}} = f\left(\sum_i w_i x_i\right)$
- Compare the calculated output to the training label
  - $\rightsquigarrow$ nudge weights to get results towards the desired target output
  - Repeat until the error is small or the given number of epochs is completed

An important detail of the neuron, which needs to be set in the beginning and can't be changed throughout the learning process, is the **activation function**. The choice is free, but here are two typical choices: Activation function

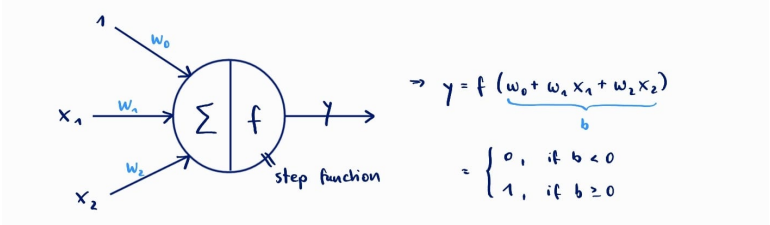
- Biological neurons use a threshold to decide when to fire. This simple **step function** can be imitated by Step function

$$f(x) = \begin{matrix} 0, & x < 0 \\ 1, & x \geq 0 \end{matrix}$$

- In NNs, the **sigmoid function** (just as in linear regression) is commonly used as the activation function f. Sigmoid function

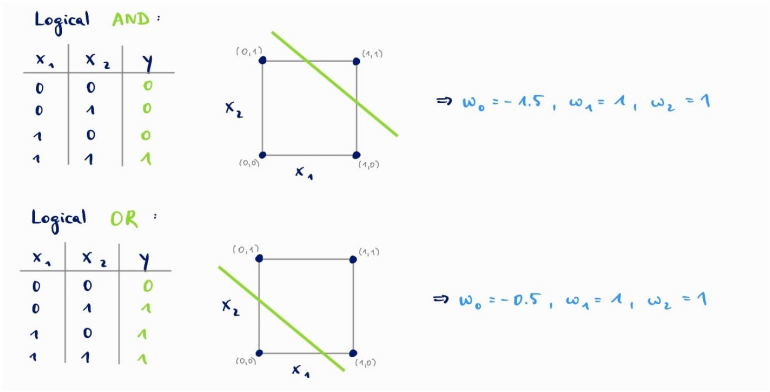
$$f(x) = sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Finally, for the simple one-layer perceptrons, we're gonna look at some examples to see what artificial neurons can learn.



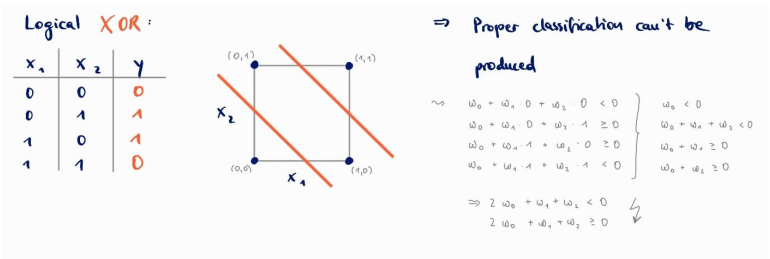
Simple NN architecture

AND- and OR-problem



Learnable problems

XOR-problem



Not-learnable problem

Figure 2.6: One-layer perceptron: Binary classification examples

The "XOR" example shows the limitations of single-layer perceptrons:

- Only a limited set of functions can be represented (e.g. even simple XOR not expressable)
- Decision boundaries must be hyperplanes (no non-linearity representable)
- Can only perfectly separate linearly separable data

2.3 Feedforwards Networks

Since the limitations of single-layer perceptrons are too strong, there is a need for more complex networks that also allow us to realize more complex problems. Different types of networks are depicted in 2.7.

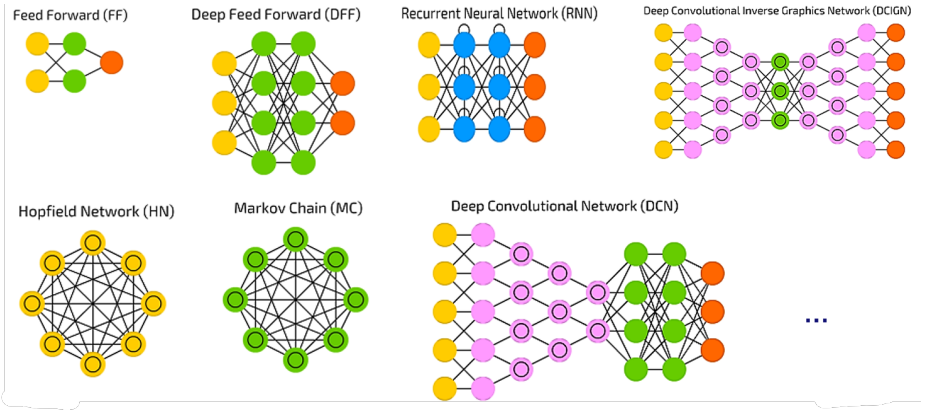


Figure 2.7: Different types and topologies of NNs

Convolutional Neural Network

One very often used architecture is the **convolutional NN** (CNN) as shown in 2.8. The intuitive idea is to have a hierarchy of visual elements that start by identifying edges using filters etc. and then move to more complex shapes. The extracted features then can be used for classification. CNNs are therefore best applied when the order of features matters, e.g. to successfully capture the spatial and temporal dependencies.

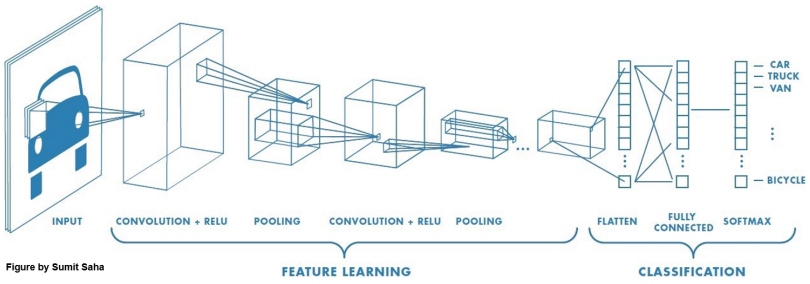


Figure 2.8: CNN architecture

Recurrent Neural Network

Another exemplary more complex architecture is the **recurrent NN** (RNN), also known as Long Short-Term Memory (LSTM). This network processes sequences of data such as speech or video. It can find the next-following most-fitting element for a sequence. The details of this network architecture are not discussed here.

- E.g.: "I lived in the Netherlands and speak perfectly Dutch"
- E.g.: "2, 4, 16, 32, 64, 128"

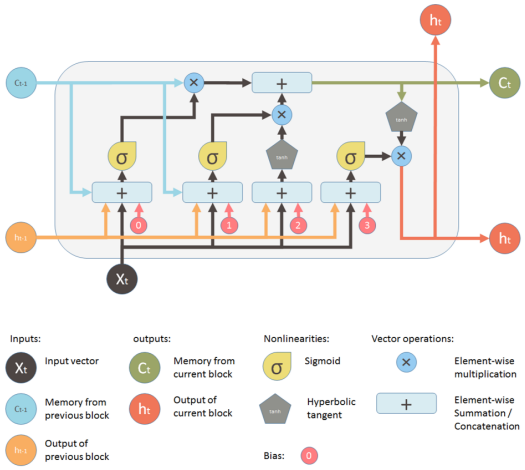


Figure 2.9: RNN architecture

Feedforward Neural Networks

For this course, we're only gonna look at **feedforward neural networks** (FNN), also called multi-layer perceptrons, that don't contain any loops[2] or special layers[3]. This means we have:

- Multiple simple perceptrons are arranged in a way that **layers** can be identified, and connections only exist to the next layer.
- The weights of the connection (in between two layers) can be changed or trained.
- The activation function (just as in a single-perceptron case) calculates whether the neuron fires or not.

To see, that FNNs really can express more functions than single-layer networks, we'll again consider the "XOR" example. As can be seen in 2.10. The result can be generalized, such that any boolean function with two input values can be represented by a two-layer perceptron.

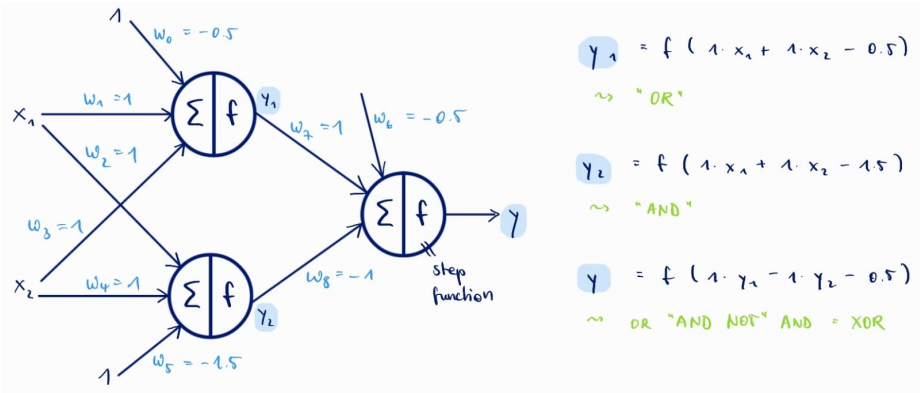


Figure 2.10: XOR with FNN

To lead us through the whole chapter, we'll introduce a general 2-layer FNN implementing

[2]Like in RNNs
[3]Like in CNNs

the function

$$y_k(\vec{\mathbf{x}}, \vec{\mathbf{w}}) = f\left(\sum_{j=0}^{M} w_{jk}^{(2)} \cdot \underbrace{h\left(\sum_{i=0}^{D} w_{ij}^{(1)} x_i\right)}_{=:z_j}\right), \forall k \in [N]$$

- Weights are notated as $w_{\text{from to}}^{(\text{layer})}$.
- f and h represent activation functions (can be different, but every layer is assumed to have the same activation function on all its neurons).
- We have input dimension D, M nodes in the hidden layer, and N output neurons.
- The hidden units with their activation functions can express non-linear functions.

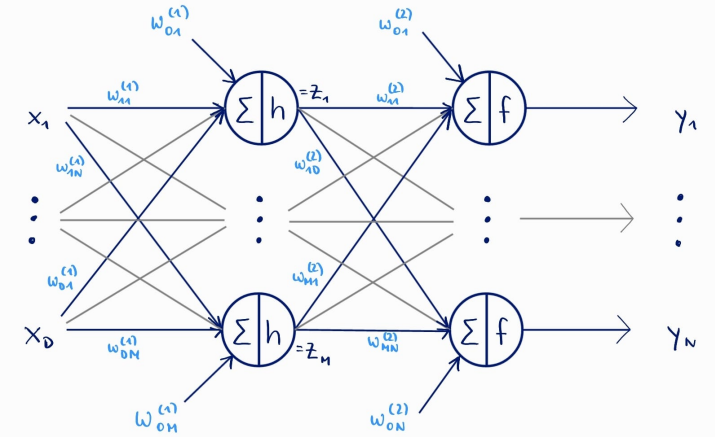


Figure 2.11: 2-layer FNN architecture

2.4 Network parameters

Generally, a **topology** of a network consists of the following network parameters that need to be decided before starting with training: Topology

- Number of **input** units (typically given)
- Number of **hidden** layers and number of neurons in each hidden layer (one or more layers)
- Number of **output** units (typically given)

Those parameters, but also generally how to train a network, raise the following questions:

- How are the inputs selected?
- How many hidden layers/neurons?
- How many neurons are in the output layer?
- How are the weights initialized? When and how are they updated?
- How many examples are (needed to be) in the training set?

We'll investigate those training parameters now in a bit more detail, starting with the **inputs**.

- Typically, we normalize our inputs such that the values fall in the range of $[0.0, 1.0]$

- Nominal or discrete-values attributes may be encoded s.t. there is one input unit per domain value
  - For that we can use "One Hot Encoding", which we already saw in a previous chapter.
  - E.g.: X can take on the possible or known values $\{v_1, v_2, v_3\}$
  - $\rightarrow$ Now encode these by separate inputs:

$$\begin{aligned} X = v_1 &\implies v_1 = 1, v_2 = 0, v_3 = 0 \\ X = v_2 &\implies v_1 = 0, v_2 = 1, v_3 = 0 \\ X = v_3 &\implies v_1 = 0, v_2 = 0, v_3 = 1 \end{aligned}$$

Next, we have the parameters influencing the **outputs**. NNs can be used for both classification and numeric prediction.

- For classification, the prediction of a class label given some input, we can either have one neuron to represent two classes (0 or 1), or one neuron per class, similar to One Hot encoding.
- Numeric prediction on the other hand has one contiuous-valued output neuron.

The next interesting parameter to investigate is the **weights**, which are the elements that are continuously updated to reduce the prediction error.

- For the initialization, we typically have random values assigned to each weight.
- Typical ranges for weights are: $w_{ij} \in [-1.0, 1.0]$ or $w_{ij} \in [-5.0, 5.0]$

For the **hidden layers** we first need to determine their amount and also the amount of neurons per hidden layer, as well as the activation functions.

- There is no clear rule as to the "best" number of hidden layers and neurons, but they may affect the accuracy of the resulting trained network.
- Network design is a trial-and-error process.

For the **training data** we have the following demands:

- The amount of training data is crucial for the correctness of the trained network.
- But, there is no robust way to indicate what the minimal size of the training set needs to be. As a rule of thumb, it goes:

$$\#\text{ training instances} \geq \underbrace{10}_{\text{some advocate 50}} \cdot \#\text{ weights in NN}$$

Important to mention: the parameters determining the hidden layers and the amount of training data are linked:

- A bigger NN allows to describe more sophisticated non-linear structures, BUT needs more training data
- Further, we have the constant battle between over- and under-fitting, as shown in a previous chpater (**??**).

2.5 Backpropagation

Now that we have the model and its basic structure and parameters, we'll look into how to train it.

Our **first step in training** is the random assignment of weights and then the measurement of the error.

$$Error\left(\vec{\mathbf{x}}, \vec{\mathbf{t}}, \vec{\mathbf{w}}\right) = \sum_{k=1}^{N} \frac{1}{2} \left(y_k - t_k\right)^2$$

With $\vec{\mathbf{x}}$: Input from training set
$\vec{\mathbf{t}}$: Labels (target values) — all vectors of size N
$\vec{\mathbf{y}}$: Calculated output with the help of $\vec{\mathbf{w}}$

Our **training goal** is to lower the error in each training episode by changing the weights. But in which direction should we change which weight?

- We have the same problem as before, and again we choose the steepest way down. The direction is known since we have the derivative of the function.
- But for our FNN, we have many weights. We need to pick a suitable step size, and we should do the update for all instances. For efficiency, this can be done in smaller batches.

So basically, we again apply gradient descent. But, when having thousands of neurons and millions of connections, the descent is performed in a space with millions or even billions of dimensions. To calculate how to nudge the weights, we therefore introduce the Neural Network Training Algorithm, better known as **Backpropagation** algorithm. It consists of the repetition of two passes:

Backpropagation

- First, we have the **Forward pass**:
  1. The network is activated on one example.
  2. Based on the calculated output, the *error of neurons of the output layer* is computed.

Forward pass

- Second, we have the **Backward pass**:
  1. Starting at the output layer, the *error is propagated backwards* through the network (layer by layer)
     - Recursively computing local deviation of error for each layer.
     - Use derivatives to preview the effect of a small change.
  2. Then the *weights are updated* (also for hidden layers).

Backward pass

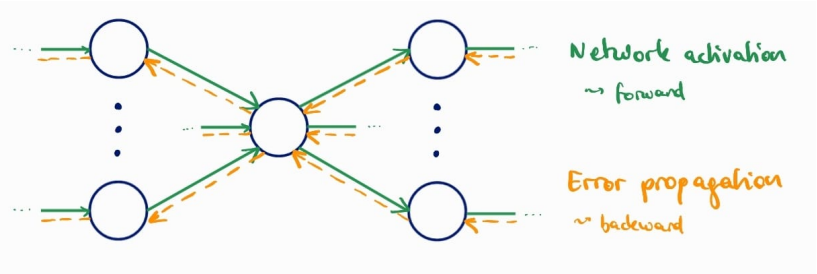


Figure 2.12: Backpropagation steps

To understand backpropagation, first, the concept of **derivatives** needs to be established:

$$\left.\frac{df(x)}{dx}\right|_{x=a} \text{ gives the slope of the curve } f \text{ at } x = a$$

Common Functions	$f(x)$	$\frac{d}{dx}f(x)$
Constant	c	0
Line	x	1
Line	ax	a
Square	x^2	$2x$
Square Root	$\sqrt{x}$	$\frac{1}{2}x^{-\frac{1}{2}}$
Exponential	e^x	e^x
Exponential	a^x	$\ln(a) \cdot a^x$
Logarithms	$\ln(x)$	$\frac{1}{x}$
Logarithms	$\log_a(x)$	$\frac{1}{x\ln(a)}$
Trigonometry	$\sin(x)$	$\cos(x)$
Trigonometry	$\cos(x)$	$-\sin(x)$
Trigonometry	$\tan(x)$	$\sec^2(x)$
Inverse Trigonometry	$\sin^{-1}(x)$	$\frac{1}{\sqrt{1-x^2}}$
Inverse Trigonometry	$\cos^{-1}(x)$	$-\frac{1}{\sqrt{1-x^2}}$
Inverse Trigonometry	$\tan^{-1}(x)$	$\frac{1}{1+x^2}$

Rules	$f(x)$	$\frac{d}{dx}f(x)$
Multiplication (const.)	cf	cf'
Power Rule	x^n	nx^{n-1}
Sum Rule	$f+g$	$f'+g'$
Difference Rule	$f-g$	$f'-g'$
Product Rule	fg	$fg'+f'g$
Quotient Rule	$\frac{f}{g}$	$\frac{f'g-g'f}{g^2}$
Reciprocal Rule	$\frac{1}{f}$	$-\frac{f'}{f^2}$
Chain Rule	$f \circ g$	$(f' \circ g) \cdot g'$
Chain Rule	$f(g(x))$	$f'(g(x)) \cdot g'(x)$
Chain Rule	$\frac{dy}{dx}$	$\frac{dy}{du} \cdot \frac{du}{dx}$

y_k, y_j, a_j, w_{ij} Consider the functions and variables we saw before for the two-layer FNN:

$$y_k(\vec{\mathbf{x}}, \vec{\mathbf{w}}) = f\left(\sum_{j=0}^{M} w_{ik}^{(2)} z_j\right) \quad \text{for } 1 \leq k \leq N$$

$$z_j = h(a_j) = \sigma(a_j) = \frac{1}{1+\exp(-a_j)}$$

$$a_j = \sum_{i=0}^{D} w_{ij}^{(1)} x_i$$

We'll calculate the weight update function for one single neuron j regarded in isolation.

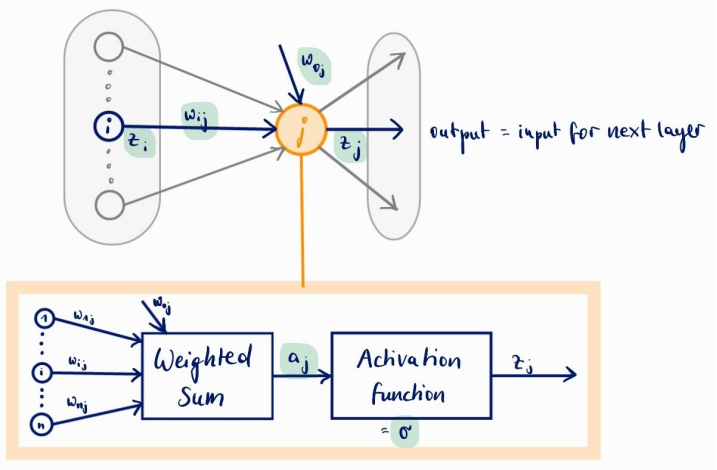


Figure 2.13: Isolated neuron components

Each neuron in the hidden layers as well as in the output layer takes its input, sums them up with weights, and then applies the activation function to it. Here, we chose the sigmoid (logistic) function. Generally, there are different activation functions h that can be used. But important for the backpropagation algorithm is that h is differentiable:

- Reason: we use the derivation of the error function, which is a variant of the gradient descent used for regression
- Figure 2.14 shows different activation functions, from which only the Sigmoid function is suitable for backpropagation

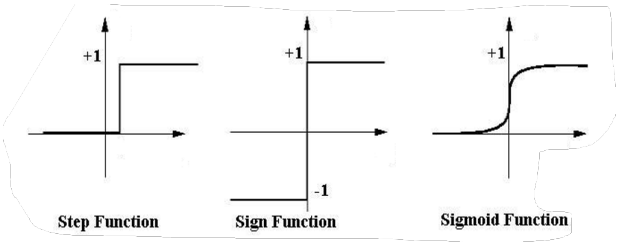


Figure 2.14: Activation functions (sigmoid)

What we now want to do with our isolated neuron j, whichever one this is, is to minimize the general error by nudging all its regarding weights w_{ij}, so we want to calculate $\frac{\partial Error}{\partial w_{ij}}$.

- Just quick note: for w_{0j} we have $z_0 = 1$ and hence $w_{0j}z_0 = 1$
- Our error is again defined as: $Error = \frac{1}{2}\sum_{\text{instances}}\sum_k (y_k - t_k)^2$ where the sum over instances is related to batch updating and epochs (in detail later)
- Important to mention about the indices: technically, we need to include the layer number (dropped here for simplicity)

Assume we notch the error definition a bit, and not regard the output of the whole NN but instead of a single layer. So we have:

$$Error = \frac{1}{2}\sum_k (z_k - t_k)^2$$

Further, we define $E_{ij} := -\frac{\partial Error}{\partial w_{ij}}$ to indicate the direction of the desired change for w_{ij}. With that, we define our update:

$$w_{ij}^{\text{new}} := w_{ij}^{\text{old}} + \partial w_{ij} = w_{ij}^{\text{old}} + \underbrace{l}_{\text{learning rate}} \cdot E_{ij}$$

- This introduces also a scaling parameter, precisely the learning rate l (similar to step size)
- The ∂w_{ij} aims to reduce the error

When we look at all our definitions, we can see that our w_{ij} only has effects "downstream" the network activation flow. We therefore can apply the chain rule. For that, we again introduce a new variable $E_j = -\frac{\partial Error}{\partial a_j}$.

$$\begin{aligned}
E_j = -\frac{\partial Error}{\partial a_j} &= -\frac{\partial Error}{\partial z_j}\frac{\partial z_j}{\partial a_j} = -\frac{\partial Error}{\partial z_j}\Big(\underbrace{\sigma(a_j)}_{=\frac{1}{1+e^{-a_j}}}\Big)' \\
&= -\frac{\partial Error}{\partial z_j} \cdot (-1)\frac{1}{(1+e^{-a_j})^2}\underbrace{e^{-a_j}\cdot(-1)}_{=(1+e^{-a_j})'} \\
&= -\frac{\partial Error}{\partial z_j}\underbrace{\frac{1}{1+e^{-a_j}}}_{=z_j}\underbrace{\frac{e^{-a_j}}{1+e^{-a_j}}}_{=\left(1-\frac{1}{1+e^{-a_j}}\right)} = -\frac{\partial Error}{\partial z_j} z_j(1-z_j)
\end{aligned}$$

$$\begin{aligned}
E_{ij} = -\frac{\partial Error}{\partial w_{ij}} &= -\frac{\partial Error}{\partial a_j}\frac{\partial a_j}{\partial w_{ij}} = E_j\frac{\partial a_j}{\partial w_{ij}} \\
&= E_j\frac{\partial(\sum_{i^*} w_{i^*j}z_{i^*})}{\partial w_{ij}} = E_j\left(\underbrace{\sum_{i^*\neq i}\frac{\partial w_{i^*j}z_{i^*}}{\partial w_{ij}}}_{=0} + \underbrace{\frac{\partial w_{ij}z_i}{\partial w_{ij}}}_{=z_i}\right) \\
&= E_j z_i
\end{aligned}$$

The last computation we need to do is $-\frac{\partial Error}{\partial z_j}$. For this case, we have two cases:

1. j is in the **output layer**

$$\begin{aligned}
\implies \quad & z_j = t_j \\
\implies \quad & -\frac{\partial Error}{\partial z_j} = -\frac{\partial\left(\frac{1}{2}\sum_{j^*}(z_{j^*}-t_{j^*})^2\right)}{\partial z_j} \\
& = -\frac{1}{2}\left(\underbrace{\sum_{j^*\neq j}(z_{j^*}-t_{j^*})^2}_{=0} + \underbrace{(z_j-t_j)^2}_{=z_j-t_j}\right) = t_j - z_j \\
\implies \quad & E_{ij} = E_j z_i = -\frac{\partial Error}{\partial z_j} z_j(1-z_j)z_i = (t_j-z_j)z_j(1-z_j)z_i \\
\implies \quad & \Delta w_{ij} = l\cdot z_i z_j(1-z_j)(t-z_j) \\
\text{And in particular} \quad & \Delta w_{0j} = l\cdot z_j(1-z_j)(t-z_j)
\end{aligned}$$

2. j is in a **hidden layer**, which means changes in z_j impact all a_k values in the next layer proportional to the connection weight.

$$\implies \quad -\frac{\partial Error}{\partial z_j} = -\sum_k \underbrace{\frac{\partial Error}{\partial a_k}}_{=-E_k} \frac{\partial a_k}{\partial z_j}$$

$$= -\sum_k -E_k \underbrace{\frac{\partial \sum_{j^*} w_{j^*k} z_{j^*}}{\partial z_j}}_{=\frac{\partial w_{jk} z_j}{\partial z_j} = w_{jk}} = \sum_k w_{jk} E_k$$

$$\implies \quad E_j = z_j(1-z_j)\sum_k w_{jk} E_k$$

$$\implies \quad E_{ij} = z_i z_j (1-z_j)\sum_k w_{jk} E_k$$

$$\implies \quad \Delta w_{ij} = l \cdot z_i z_j (1-z_j)\sum_k w_{jk} E_k$$

And in particular $\quad \Delta w_{0j} = l \cdot z_j (1-z_j)\sum_k w_{jk} E_k$

Summarized, we have explicit expressions for our updates:

Weight update, E_{ij}, E_j, Δw_{ij}

$w_{ij}^{\text{new}} = w_{ij}^{\text{old}} + \Delta w_{ij}$ — updated weight of connection from neuron i to neuron j based on some training instance

$\Delta w_{ij} = l E_{ij} = l E_j z_i$

$\Delta w_j = l E_j$

$E_j = z_j(1-z_j)(t-z_j)$ — **Case 1:** neuron j is in the output layer

$E_j = z_j(1-z_j)\sum_k w_{jk} E_k$ — **Case 2:** neuron j is in a hidden layer

With scaling parameter l as the learning rate

Next, we're gonna investigate the **learning rate**, which is usually a constant between 0.0 and 1.0.

Learning rate l

- As a rule of thumb: $l = \frac{1}{r}$ with r being the "round", so the number of iterations
- It can also help to start with bigger steps and end with smaller ones to help a quicker convergence while avoiding overshooting the target
- The choice of l is similar to the choice of the step size in regression and SVMs
- Typical problem:

  l too small : learning occurs at very slow pace

  l too large : oscillation between inadequate solutions can occur

An important thing to consider for backpropagation is **when to update**:

- In the case of **instance updating** we update the NN for each instance (sample) individually.

  Instance updating
- Contrary, when we apply **batch updating** the NN is updated for a subset (or all) of all training instances in one update action. So we have an error calculation based on the same weights for multiple instances

  Batch updating

- Best practice is using **mini batches**, where the size depends on the problem.

Epoch

- Another important term is **epoch**, which describes one complete consideration of all the training data.

Termination

Finally, there's also the consideration for the **termination** of the backpropagation algorithm. There are multiple termination criteria indicating to stop training.

- All Δw_{ij} in the previous epoch are small (below some specified threshold)
- Percentage of misclassified inputs in previous epoch is small
- Pre-specified number of epochs has expired
- In practice: Several hundreds of thousands of epochs may be required before weights are converging

2.6 Beyond Basic FNNs

The general application area for NNs is supervised learning. This has been the focus and most dominant usage.

- This creates the requirement of labeled data as input, potentially a large amount of those
- There is a huge variety of network types, as we saw before, also optimized for different input types:
  - CNNs use convolution to better deal with images
  - RNNs and LSTM (Recurrent NN, Long Short-Term Memory) are tailored for temporal or sequential data
- Generally, NNs require lots of engineering and trial and error (no magic bullet, no free lunch)

Beyond supervised learning, there are also **unsupervised learning** applications, where we don't use classes or similar things as our target.

Autoencoder

For example, in the case of **autoencoders**, the "label" is produced from the input such that the input is tried to be reproduced. The error is the difference between input and calculated output. The interesting part is the significantly more dense representation of the input data captured by the middle layer, as can be seen in 2.15.

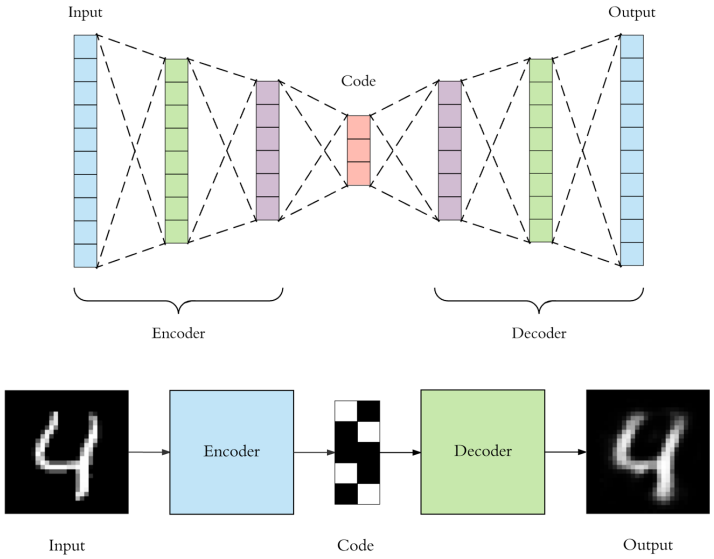


Figure 2.15: Unsupervised learning: autoencoder

Another application is **GANs** (generative adversarial networks). Here, we have two networks: Generative Adversarial Networks

- One network is the generator that tries to create "fake data" (e.g. fake art, fake people)
- The other is the discriminator that tries to distinguish "real" from "fake data"
- This means, they have opposing goals

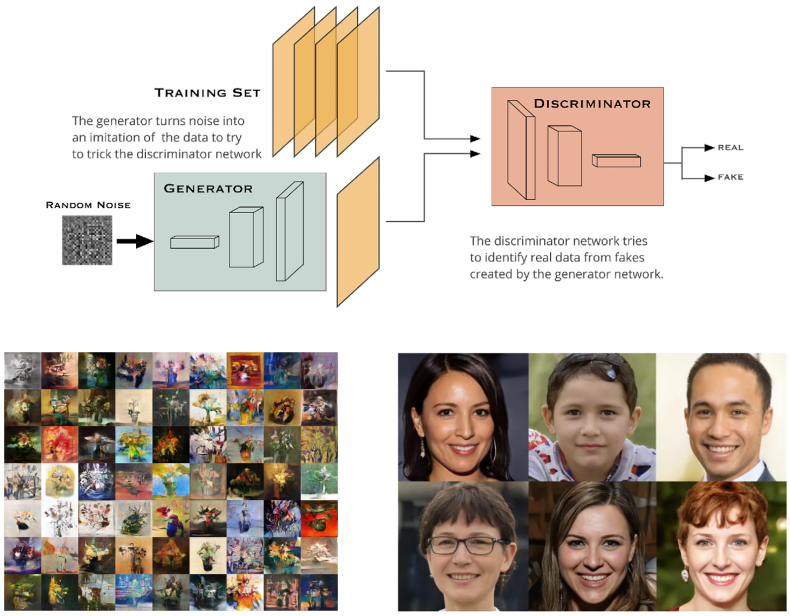
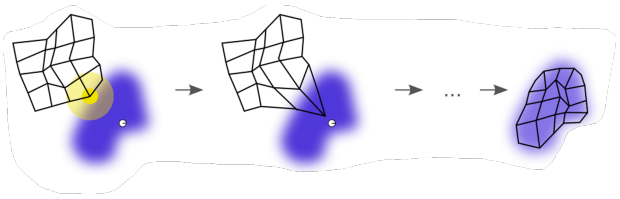


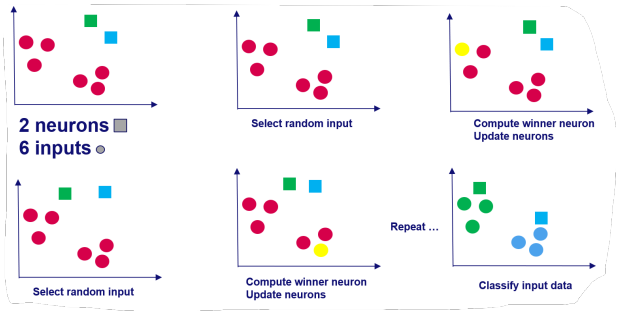
Figure 2.16: Unsupervised learning: GANs

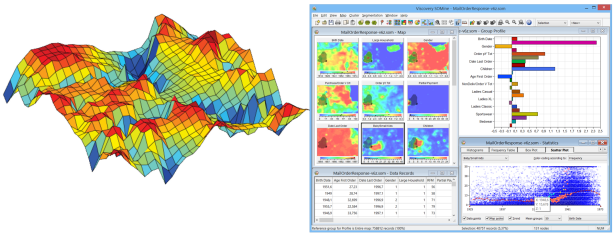
A final unsupervised application is **self-organizing maps** which is a directly unsuper- Self-organizing maps

vised neural network.

- We have a distribution of the training data (blue blob)
- In the beginning, the SOM nodes are arbitrarily positioned in the data space
- Then the following process is iterated until all input data has been seen:
  - Randomly select a training data point as input (white point)
  - The node nearest to the training node is selected as winner (yellow point, with neighbors in yellow area)
  - It is then moved toward the training data, just as the neighbors on the grid (just to a lesser extent), so update the neurons
- Then the input can be classified
  - The inputs are connected to the neurons with weights
  - The neurons are also related to each other







Natural grouping of instances in the two-dimensional grid (each cell colored using any of the features)

Figure 2.17: Unsupervised learning: SOMs