Machine Learning - Lesson 2

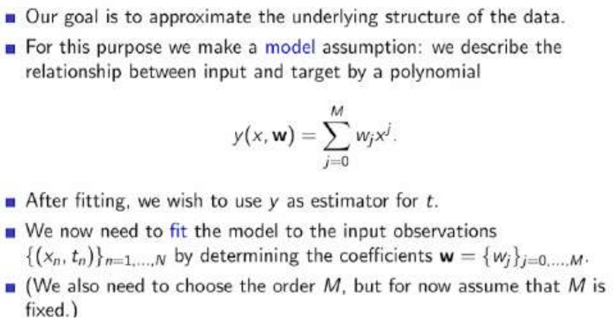
LINEAR REGRESSION

Solving the fitting task

As seen in the previous lesson, to solve every Machine Learning problem task, we need to define a model which "models" the relation tip between inputs and targets. Each model is an assumption, data may not follow that model, but it may be able to still detect an underlying structure.

NW. A polynomial models mans that the target is some polynomial function of the input.

Let's look this example model:



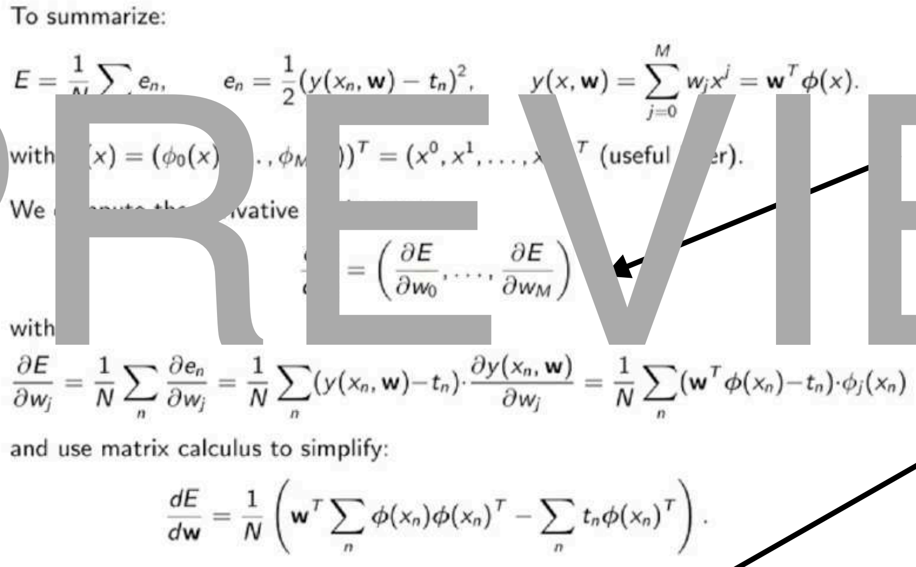
- Our goal is to approximate the underlying structure of the data.
- For this purpose we make a model assumption: we describe the relationship between input and target by a polynomial

$$y(x, \mathbf{w}) = \sum_{j=0}^{M} w_j x^j.$$

- After fitting, we wish to use y as estimator for t.
- We now need to fit the model to the input observations $\{(x_n, t_n)\}_{n=1,\dots,N}$ by determining the coefficients $\mathbf{w} = \{w_j\}_{j=0,\dots,M}$.
- (We also need to choose the order M, but for now assume that M is fixed.)

y is the estimator for t. This models depends on some parameters collected in the vector w. To solve the fitting task means that, out of the infinite amount of possible functions that could map the input to thtarget into t, we choose the functions which fits into this model template. In more academic words, instead of looking for an arbitrary function, now we look for a parameter vector. In machine Learning, to parametrize a model means to estimate a vector of parameters that can actually fit in that model.

How do we fit? In order to fit we need some criterion that, out of the infinite amount of possible w (parameters) in W (parameter vector), chooses the parameters that can fit in the model. In this case we need to find the vector that **minimises** the mean squared error.



To summarize:

$$E = \frac{1}{N}\sum_n e_n, \qquad e_n = \frac{1}{2}(y(x_n, \mathbf{w}) - t_n)^2, \qquad y(x, \mathbf{w}) = \sum_{j=0}^{M} w_j x^j = \mathbf{w}^T\phi(x).$$

with $\phi(x) = (\phi_0(x), \dots, \phi_M(x))^T = (x^0, x^1, \dots, x^M)^T$ (useful later).

We compute the derivative

$$\frac{dE}{d\mathbf{w}} = \left(\frac{\partial E}{\partial w_0}, \dots, \frac{\partial E}{\partial w_M}\right)$$

with

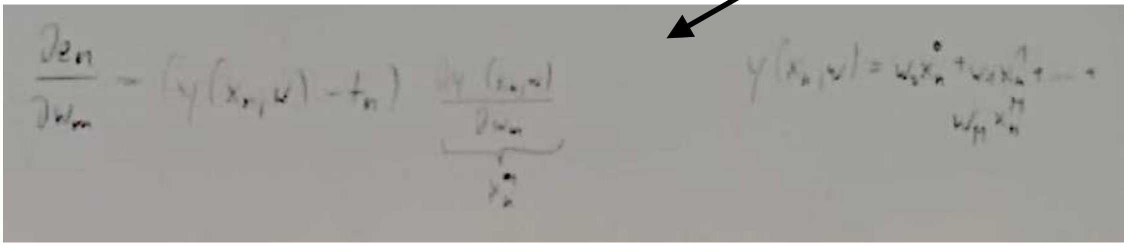
$$\frac{\partial E}{\partial w_j} = \frac{1}{N}\sum_n \frac{\partial e_n}{\partial w_j} = \frac{1}{N}\sum_n (y(x_n, \mathbf{w}) - t_n)\cdot\frac{\partial y(x_n, \mathbf{w})}{\partial w_j} = \frac{1}{N}\sum_n (\mathbf{w}^T\phi(x_n) - t_n)\cdot\phi_j(x_n)$$

and use matrix calculus to simplify:

$$\frac{dE}{d\mathbf{w}} = \frac{1}{N}\left(\mathbf{w}^T\sum_n \phi(x_n)\phi(x_n)^T - \sum_n t_n\phi(x_n)^T\right).$$

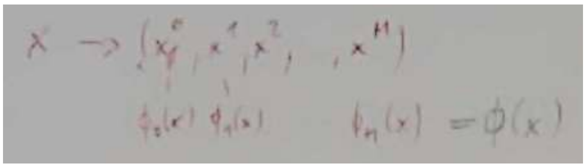
If we calculate the derivative of the error with respect to the weight vector this should be a vector containing the derivatives of E with respect to the single direction.

This derivative vector can be solved using the **chain rule**.

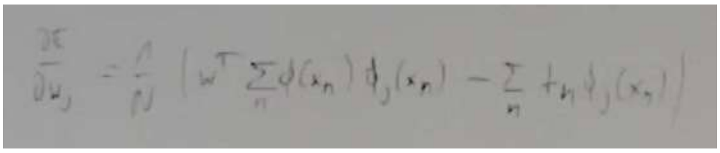

$\frac{\partial e_n}{\partial w_m} = (y(x_n, w) - t_n)\underbrace{\frac{\partial y(x_n, w)}{\partial w_m}}_{x_n^m}$
$y(x_n, w) = w_0 x_n^0 + w_1 x_n^1 + \dots + w_M x_n^M$

In this simple example x is a number (not a vector). In this calculation we have x^m power. From this 1-dimensional x, we can get a couple of different numbers, each one called $\Phi_n(x)$:

Please note that $\Phi(x_n)$ is a vector, while $\Phi_j(x_n)$ is the single element of that vector

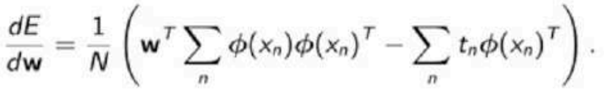

$x \rightarrow (x^0, x^1, x^2, \dots, x^M)$
$\phi_0(x)\ \phi_1(x)\ \dots\ \phi_M(x) = \phi(x)$

Since the derivative of a sum is a sum of derivatives, we can now compute the sum of the derivatives of single errors.



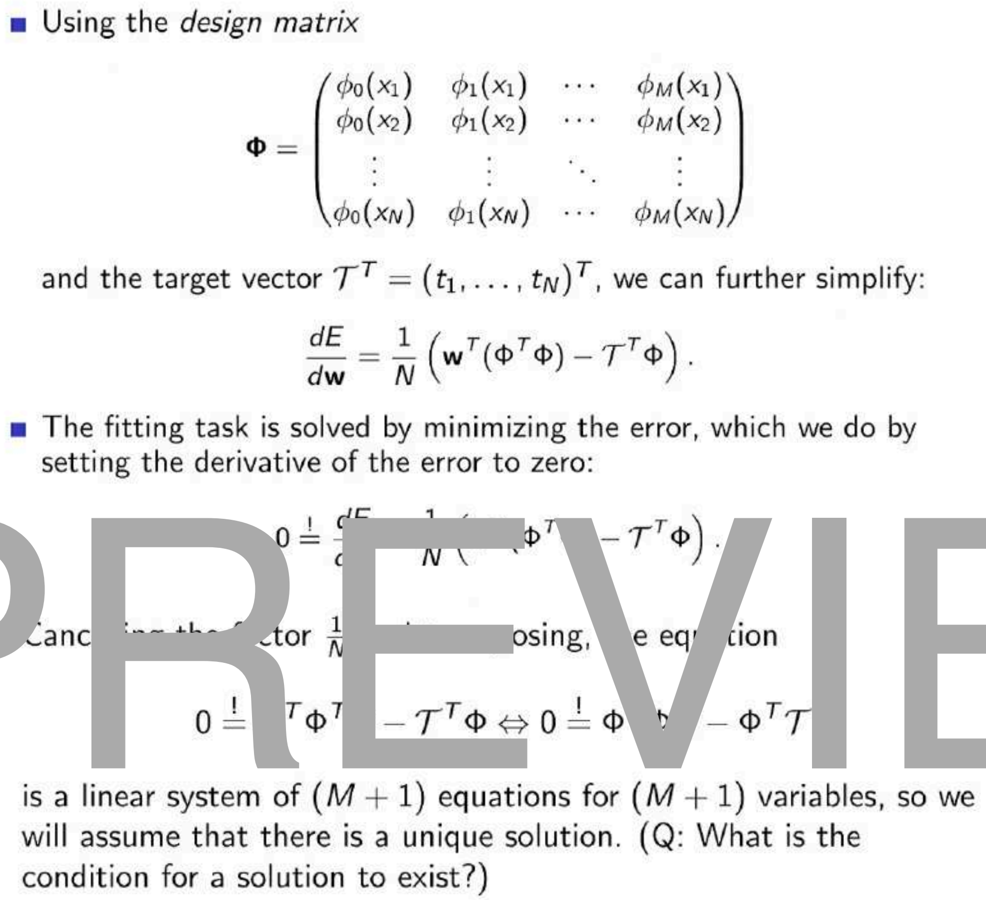
$$\frac{\partial E}{\partial w_j} = \frac{1}{N}\left(w^T \sum_n \phi(x_n)\,\phi_j(x_n) - \sum_n t_n \phi_j(x_n) \right)$$

This can be written in matrix form as follows:



$$\frac{dE}{d\mathbf{w}} = \frac{1}{N}\left(\mathbf{w}^T \sum_n \phi(x_n)\phi(x_n)^T - \sum_n t_n \phi(x_n)^T \right).$$

Then



- Using the *design matrix*

$$\mathbf{\Phi} = \begin{pmatrix} \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_M(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \cdots & \phi_M(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(x_N) & \phi_1(x_N) & \cdots & \phi_M(x_N) \end{pmatrix}$$

and the target vector $\mathcal{T}^T = (t_1, \ldots, t_N)^T$, we can further simplify:

$$\frac{dE}{d\mathbf{w}} = \frac{1}{N}\left(\mathbf{w}^T(\Phi^T\Phi) - \mathcal{T}^T\Phi \right).$$

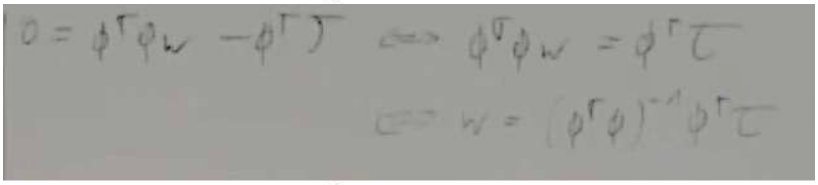
- The fitting task is solved by minimizing the error, which we do by setting the derivative of the error to zero:

$$0 \overset{!}{=} \frac{dE}{d\mathbf{w}} = \frac{1}{N}\left(\mathbf{w}^T(\Phi^T\Phi) - \mathcal{T}^T\Phi \right).$$

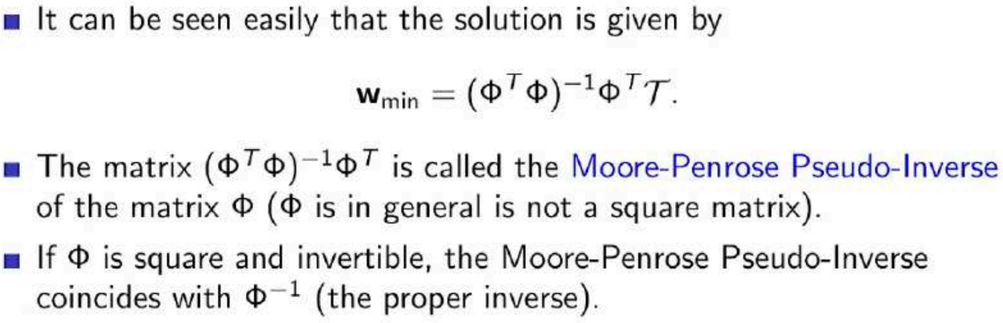
Cancelling the factor $\frac{1}{N}$ and transposing, the equation

$$0 \overset{!}{=} \mathbf{w}^T\Phi^T\Phi - \mathcal{T}^T\Phi \Leftrightarrow 0 \overset{!}{=} \Phi^T\Phi\mathbf{w} - \Phi^T\mathcal{T}$$

is a linear system of $(M+1)$ equations for $(M+1)$ variables, so we will assume that there is a unique solution. (Q: What is the condition for a solution to exist?)



$$0 = \Phi^T\Phi w - \Phi^T \mathcal{T} \Leftrightarrow \Phi^T\Phi w = \Phi^T \mathcal{T}$$
$$\Leftrightarrow w = (\Phi^T\Phi)^{-1}\Phi^T \mathcal{T}$$



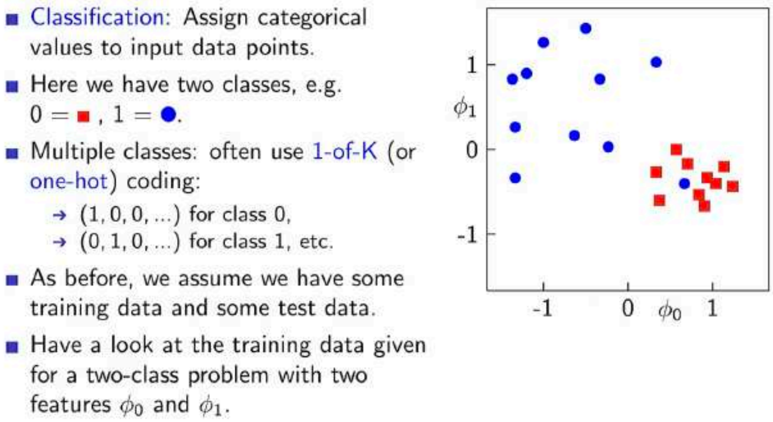
- It can be seen easily that the solution is given by

$$\mathbf{w}_{\text{min}} = (\Phi^T\Phi)^{-1}\Phi^T\mathcal{T}.$$

- The matrix $(\Phi^T\Phi)^{-1}\Phi^T$ is called the Moore-Penrose Pseudo-Inverse of the matrix Φ (Φ is in general is not a square matrix).
- If Φ is square and invertible, the Moore-Penrose Pseudo-Inverse coincides with Φ^{-1} (the proper inverse).

CLASSIFICATION

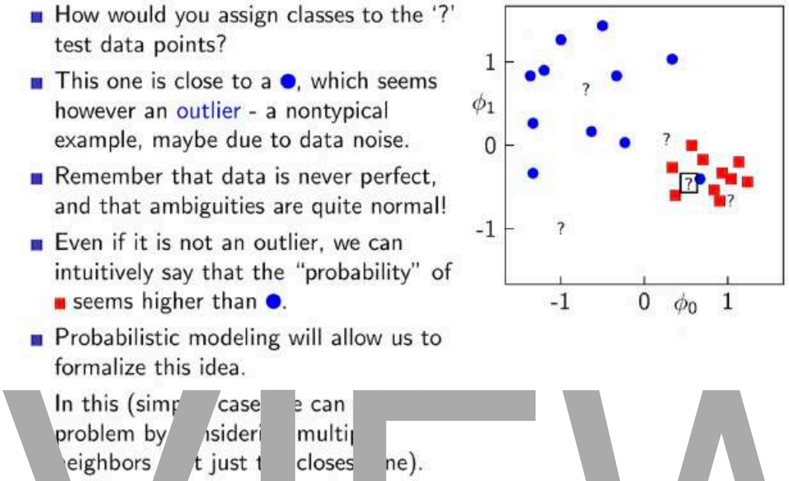
Classification is the other general task of supervised learning. While in regression I get real values targets (I.e. in a continuous space), in classification I get categorical targets (0 or 1).
So in a classification task, each class is represented by a vector. This vector maps all the dataset and is filled with 0 except for the elements contained in that class, which have value 1.



- Classification: Assign categorical values to input data points.
- Here we have two classes, e.g. 0 = 🟥 , 1 = ●.
- Multiple classes: often use 1-of-K (or one-hot) coding:
  - → $(1, 0, 0, \ldots)$ for class 0,
  - → $(0, 1, 0, \ldots)$ for class 1, etc.
- As before, we assume we have some training data and some test data.
- Have a look at the training data given for a two-class problem with two features ϕ_0 and ϕ_1.

Example of a Classification problem

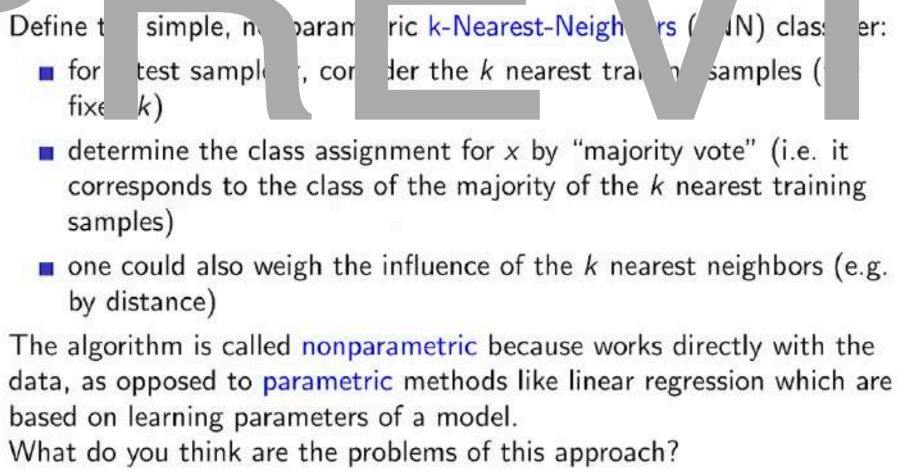
In this example, you have to assign the "?" Points to a class. Each time you decide where to assign a point, you are making an assumption (in this caused based on distance, as we'll see later). We frequently assume that for each sample there is a true class, but there may be sample that could be in both classes, in this case the model can reject the classification saying that he cannot decide.



- How would you assign classes to the '?' test data points?
- This one is close to a ●, which seems however an outlier - a nontypical example, maybe due to data noise.
- Remember that data is never perfect, and that ambiguities are quite normal!
- Even if it is not an outlier, we can intuitively say that the "probability" of 🟥 seems higher than ●.
- Probabilistic modeling will allow us to formalize this idea.

In this (simple) case we can solve the problem by considering multiple neighbors (not just the closest one).

kNN classifier

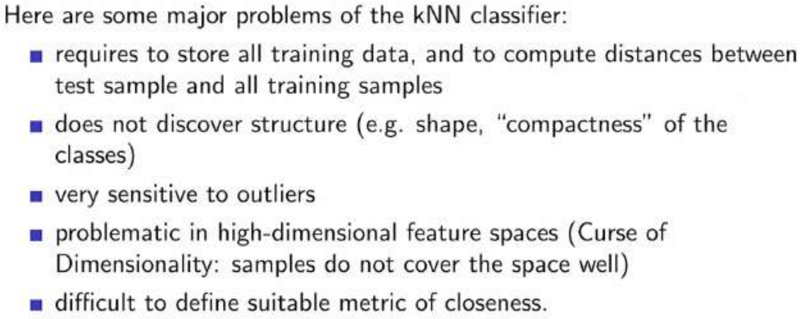


Define the simple, nonparametric k-Nearest-Neighbors (kNN) classifier:

- for a test sample x, consider the k nearest training samples (for fixed k)
- determine the class assignment for x by "majority vote" (i.e. it corresponds to the class of the majority of the k nearest training samples)
- one could also weigh the influence of the k nearest neighbors (e.g. by distance)

The algorithm is called nonparametric because works directly with the data, as opposed to parametric methods like linear regression which are based on learning parameters of a model.
What do you think are the problems of this approach?

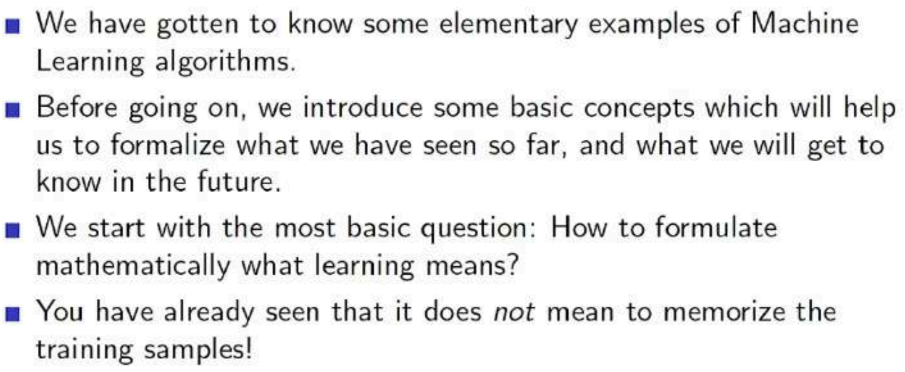
Since K-means doesn't learn structure of the data, it is extremely prone to outliers. **NW.** The classification task does not **assume a model**. It is the task to find the best way to classify data, not to find an underlying structure.



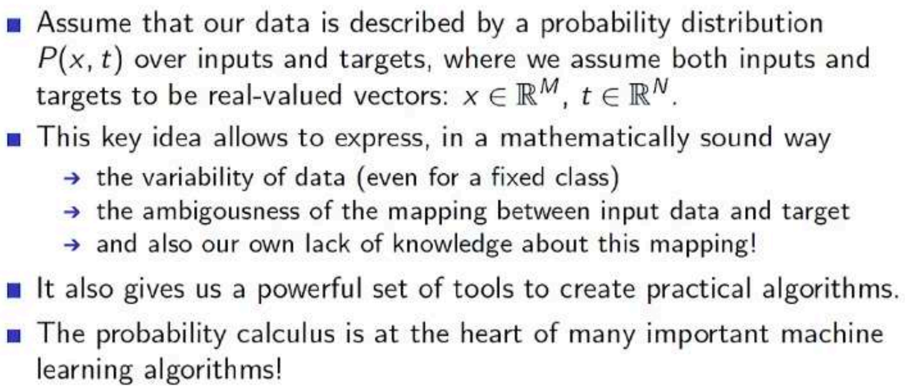
Here are some major problems of the kNN classifier:

- requires to store all training data, and to compute distances between test sample and all training samples
- does not discover structure (e.g. shape, "compactness" of the classes)
- very sensitive to outliers
- problematic in high-dimensional feature spaces (Curse of Dimensionality: samples do not cover the space well)
- difficult to define suitable metric of closeness.

ELEMENTS OF THE FOUNDATIONS OF MACHINE LEARNING



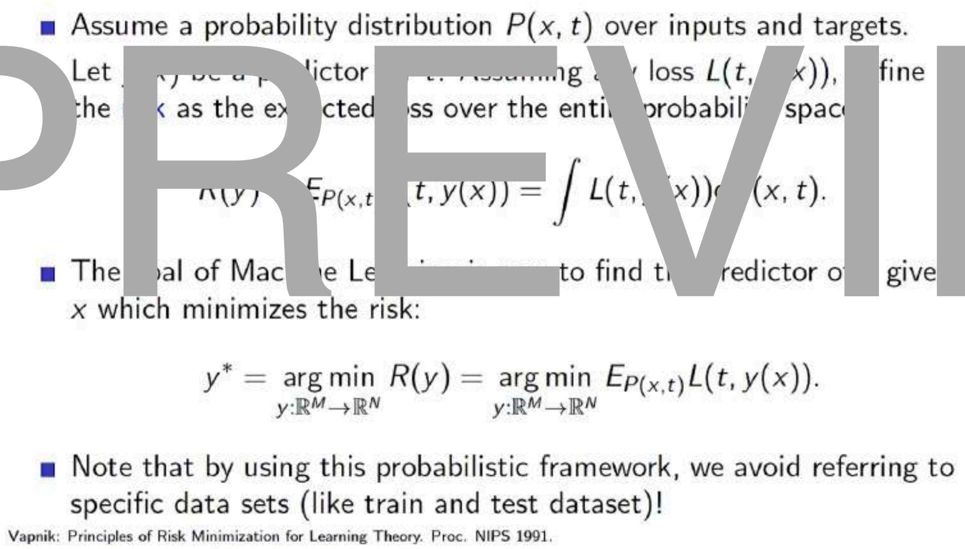
- We have gotten to know some elementary examples of Machine Learning algorithms.
- Before going on, we introduce some basic concepts which will help us to formalize what we have seen so far, and what we will get to know in the future.
- We start with the most basic question: How to formulate mathematically what learning means?
- You have already seen that it does *not* mean to memorize the training samples!



- Assume that our data is described by a probability distribution $P(x,t)$ over inputs and targets, where we assume both inputs and targets to be real-valued vectors: $x \in \mathbb{R}^M$, $t \in \mathbb{R}^N$.
- This key idea allows to express, in a mathematically sound way
  - → the variability of data (even for a fixed class)
  - → the ambigousness of the mapping between input data and target
  - → and also our own lack of knowledge about this mapping!
- It also gives us a powerful set of tools to create practical algorithms.
- The probability calculus is at the heart of many important machine learning algorithms!

Here x are the data and t the targets. M is the number of features and N the number of targets.

Machine Learning is all about minimising the so called **risk.**



- Assume a probability distribution $P(x,t)$ over inputs and targets. Let $y(x)$ be a predictor of t. Assuming any loss $L(t, y(x))$, define the risk as the expected loss over the entire probability space

$$R(y) = E_{P(x,t)}L(t,y(x)) = \int L(t,y(x))dP(x,t).$$

- The goal of Machine Learning is now to find the predictor of t given x which minimizes the risk:

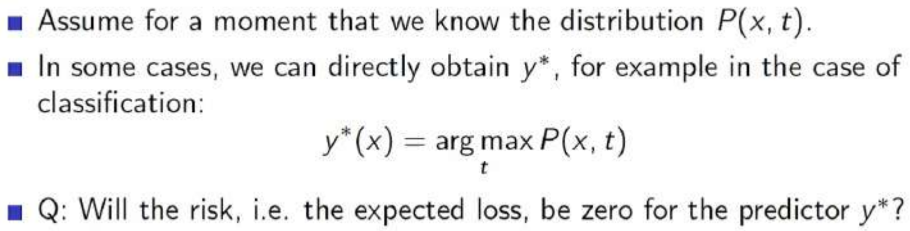
$$y^* = \underset{y:\mathbb{R}^M \to \mathbb{R}^N}{\arg\min}\ R(y) = \underset{y:\mathbb{R}^M \to \mathbb{R}^N}{\arg\min}\ E_{P(x,t)}L(t,y(x)).$$

- Note that by using this probabilistic framework, we avoid referring to specific data sets (like train and test dataset)!

Vapnik: Principles of Risk Minimization for Learning Theory. Proc. NIPS 1991.

This is impossible since you cannot integrate the whole probability.

The irreducible error
We demonstrate that the risk will never be zero by assuming that:



- Assume for a moment that we know the distribution $P(x,t)$.
- In some cases, we can directly obtain y^*, for example in the case of classification:

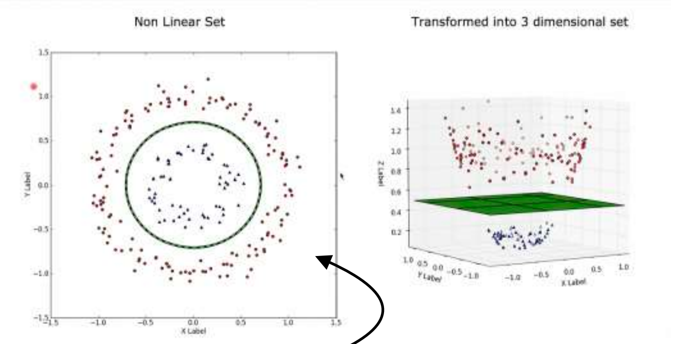
$$y^*(x) = \arg\max_t P(x,t)$$

- Q: Will the risk, i.e. the expected loss, be zero for the predictor y^*?

Machine Learning - Lesson 5

KERNEL METHODS

Kernel Methods are linear methods which operates in a feature space. In a broad way, Kernel methods are methods which map the feature of a dataset in a different dimension, making possible to separate values. Kernel methods find the hyperplane/boundary by bringing fetures/datapoints in a different dimensionality space.



- In the last part, we have talked about linear methods for classification and regression.
- The methods we have covered so far are all *parametric*: a model is computed from the training data, then the training data is discarded, and only the model is used for further calculations.
- There are, however, methods where at least a part of the training data is kept.
- Some of them are based on simple comparison of test samples and training samples (e.g. the kNN classifier).
- Kernel methods are more sophisticated examples of this category.
- Kernel methods are linear methods in a feature space. Yet, they offer a different view of both linear models, and the concept of features.
- We will cover in detail the best-known kernel algorithm, the Support Vector Machine.

In this dataset (which is 2-D), the hyperplane would be 1-D (so a line). Since it's impossible to find a line that divides correctly this dataset, you plot the data to a 3-dimensional space, and you find the hyperplane as a plane that divides the feature. If then you re-transpose this into a 1-D space, you get it as a circle.
NW. From here on, by "plot the data to a n-D space" I refer to the fact that kernel methods **can lead to a higher dimensional representation of data**, they don't actually plot anything, they are just able to compute fake input data that can lead to a representation higher in dimensionality).

Dual Representations

With Ke[illegible]ls, you [illegible]ar regression in a different way. We are expressing a solution of the linear classifier as a **linear combination of the features**.
We consider as a starting point a Linear Regression problem. This is the loss function (i.e. the function that minimises the Mean Squared Error).

$E(\mathbf{w}) = \sum_{n=1}^{N} \frac{1}{2}(\mathbf{w}^T\phi_n - t_n)^2$ This minimises the MSE

while,

$+\frac{\lambda}{2}||\mathbf{w}||_2^2$ This is the regularization term, which penalises large values of the weight vector **w**.

- Consider Linear Regression with training data $(\phi_n, t_n)_{n=1,\dots,N}$ with L_2 regularization, where $\phi_n = \phi(\mathbf{x}_n)$:

$$E(\mathbf{w}) = \sum_{n=1}^{N} \frac{1}{2}(\mathbf{w}^T\phi_n - t_n)^2 + \frac{\lambda}{2}||\mathbf{w}||_2^2$$

(note that for each calculation [illegible] regularized squared error, not the *mean* squared error)

The gradient of the right-hand side w.r.t. **w** is

$$\sum_{n=1}^{N} (\mathbf{w}^T\phi_n - t_n)\phi_n + \lambda\mathbf{w}$$

and setting it to zero yields

$$\mathbf{w} = \sum_{n=1}^{N} \underbrace{-\frac{1}{\lambda}(\mathbf{w}^T\phi_n - t_n)}_{a_n}\phi_n$$

with scalars a_n.

If u want to solve for this equation, you take gradient with respect to weight vector **w** and then you set it to 0 and solve for the weight vector **w**.
Basically you can express the solution for the weight vector **w** as a linear combination of your features. Basically can express the solution of the linear classifier with a linear combination of the data itself. *This is the basic idea of kernel methods*.

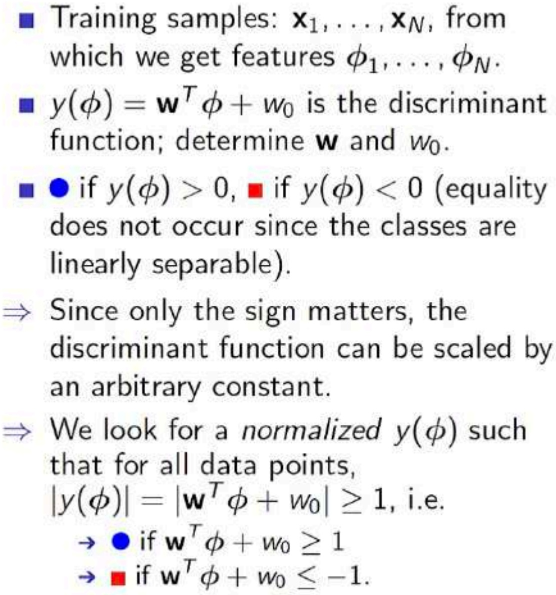
This expression $\mathbf{w} = \sum_{n=1}^{N} \underbrace{-\frac{1}{\lambda}(\mathbf{w}^T\phi_n - t_n)}_{a_n}\phi_n$ can be re-written as $\mathbf{w} = \Phi^T\mathbf{a}$.

So you have your weight vector **w** which is a function of your features, which are put tighter in the form of a matrix ϕ^T multiplied for vector **a** which stores the coefficients. This ϕ^T is a matrix having rows as features for one data point and columns are the different data points.
So basically, **a** contains, for each feature, this

$$\underbrace{-\frac{1}{\lambda}(\mathbf{w}^T\phi_n - t_n)}_{a_n}$$

Mathematical Formulation of this idea

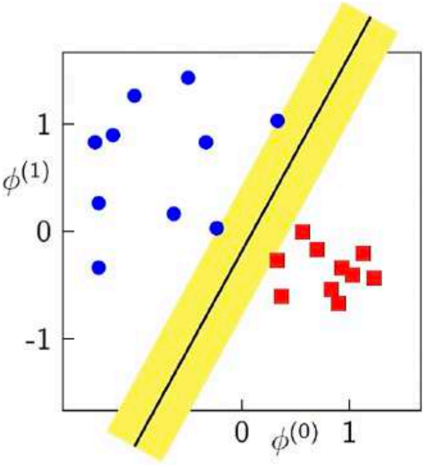
As seen before, the decision function has been parametrised as $y(\phi) = \mathbf{w}^T\phi + w_0$
Where $\mathbf{w_0}$ is a bias term.



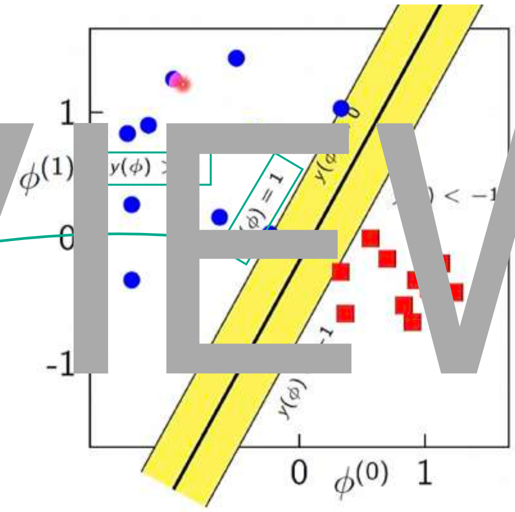
- Training samples: $\mathbf{x}_1, \ldots, \mathbf{x}_N$, from which we get features $\phi_1, \ldots, \phi_N$.
- $y(\phi) = \mathbf{w}^T\phi + w_0$ is the discriminant function; determine $\mathbf{w}$ and w_0.
- ● if $y(\phi) > 0$, ■ if $y(\phi) < 0$ (equality does not occur since the classes are linearly separable).

⇒ Since only the sign matters, the discriminant function can be scaled by an arbitrary constant.

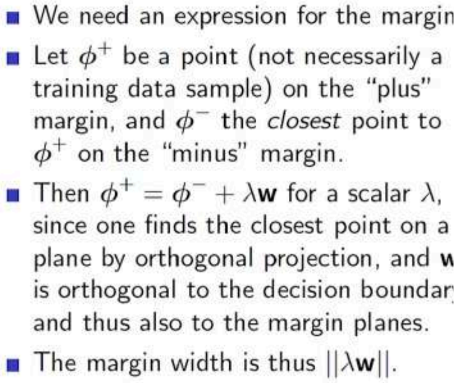
⇒ We look for a *normalized* $y(\phi)$ such that for all data points, $|y(\phi)| = |\mathbf{w}^T\phi + w_0| \geq 1$, i.e.
- → ● if $\mathbf{w}^T\phi + w_0 \geq 1$
- → ■ if $\mathbf{w}^T\phi + w_0 \leq -1$.



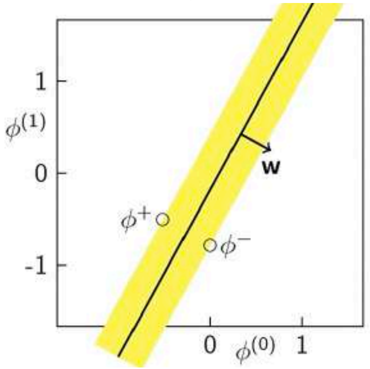
By scaling the discriminant function by an arbitrary function you get a normalised version of the decision boundary in which you can say that:
- you assign a point to the first class only if the prediction is larger then 1 i.e. you are beyond the margin
- You assign class 2 if you are < -1 (i.e. under the margin).

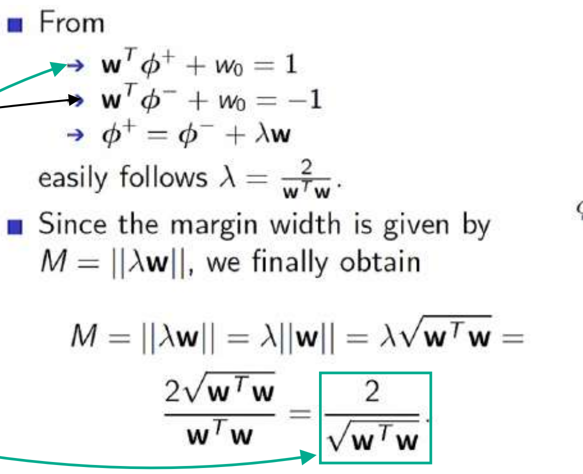


Then to calculate the width of the margin we can take two points on the two side of the margin **Φ** + and **Φ** - , and then we can define $\phi^+ = \phi^- + \lambda\mathbf{w}$ for a scalar λ,
This makes sense because the w vector is orthogonal to the decision boundary.



- We need an expression for the margin
- Let ϕ^+ be a point (not necessarily a training data sample) on the "plus" margin, and ϕ^- the *closest* point to ϕ^+ on the "minus" margin.
- Then $\phi^+ = \phi^- + \lambda\mathbf{w}$ for a scalar λ, since one finds the closest point on a plane by orthogonal projection, and $\mathbf{w}$ is orthogonal to the decision boundar and thus also to the margin planes.
- The margin width is thus $||\lambda\mathbf{w}||$.





- From
  - → $\mathbf{w}^T\phi^+ + w_0 = 1$
  - → $\mathbf{w}^T\phi^- + w_0 = -1$
  - → $\phi^+ = \phi^- + \lambda\mathbf{w}$

  easily follows $\lambda = \frac{2}{\mathbf{w}^T\mathbf{w}}$.
- Since the margin width is given by $M = ||\lambda\mathbf{w}||$, we finally obtain

$$M = ||\lambda\mathbf{w}|| = \lambda||\mathbf{w}|| = \lambda\sqrt{\mathbf{w}^T\mathbf{w}} = \frac{2\sqrt{\mathbf{w}^T\mathbf{w}}}{\mathbf{w}^T\mathbf{w}} = \boxed{\frac{2}{\sqrt{\mathbf{w}^T\mathbf{w}}}}.$$

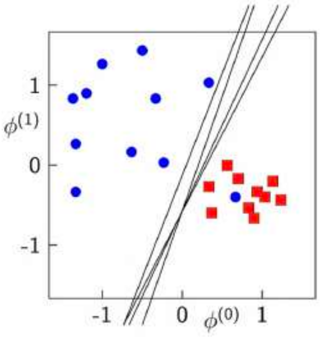
Then we can say:
When we are on one side off the margin we get 1, if we are on the other we get -1
This is the **final width** of the margin

NEURAL NETWORKS

Introduction

We have two classes that may or may not be linearly separable. So we are in a classification problem.
To issue this problem we have seen two different models: **SVM** and **Logistic Regression** (which is a probabilistic model).
Here we introduce another model to issue this task.

- Again consider a two-class problem for classification.
- The classes may or may not be linearly separable.
- We know the SVM and the maximum margin criterion, as well as Logistic Regression. Which other ways can we think of to
  1. parametrize a classifier,
  2. define a criterion for training it,
  3. actually compute the optimal solution?



The Perceptron Model

This is a **linear model.**
In this model we have:

- a linear model with **fixed feature Φ(x)**
- the activation function **f** which is equal to +1 if the argument is greater or equal than 0 and equal to -1 if it is lower than 0.

Below we are just explaining why in this formulation there is no bias term.

- We consider a linear model of the form

$$y(\mathbf{x}) = f(\mathbf{w}^T\phi(\mathbf{x}))$$

with the usual fixed feature transformation ϕ and an *activation function*

$$f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases}$$

- For simplicity, the bias is included in the feature transformation as a fixed value $\phi_0(\mathbf{x}) = 1$.
- The class targets are encoded as $t = \pm 1$ to match the possible values of $y(\mathbf{x}) = f(\mathbf{w}^T\phi(\mathbf{x}))$.

This just means that we call the two classes “class 1” and “class -1”

LINEAR BIAS

$$\boxed{w^T\phi} + \boxed{w_0} = 0 \quad \iff$$

NOW, WE DEFINE A NEW FEATURE VECTOR $\boxed{\tilde{\phi} = \begin{pmatrix}\phi \\ 1\end{pmatrix}}$

THIS IS CALLED MORE 1-COEFFICIENT VWARE. THIS MEANS THAT $\tilde{\phi}$ IS ϕ WITH 1 COEFFICIENT MORE (1 in this case)

SO,

w vector WITH ONE COEFFICIENT MORE

$$w^T\phi + w_0 \iff \tilde{w}^T\tilde{\phi} = 0, \text{ with } \tilde{\phi} = \begin{pmatrix}\phi\end{pmatrix}, \; \tilde{w} = \begin{pmatrix}w\end{pmatrix}$$

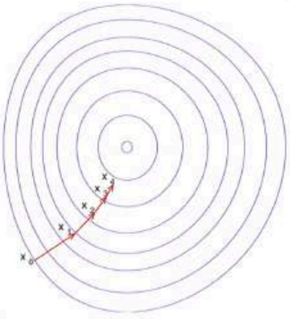
TH EQUAT MEANS THAT WE AN L E OUT THE BI TERM B T USI A 1-COEFF. MOR and φ V

In the a d we are s ply lo king for the weight ecto w so tha he fu ction matches the or ct class, s this is a **li ar m lel.**

Optimising a Function: Gradient Descent

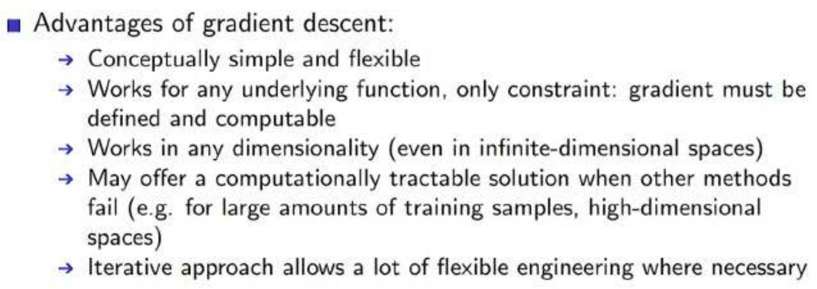
The most generic way to optimise functions that are far too complex is the Gradient Descent.
NW. Equivalently gradient descent can be used to maximise the function by looking, at each step, which is the direction of the positive gradient.
The cool feature of this solution is that you need only **local informations**, I.e. the information of the “position”/step you are in the moment, so you don’t need to compute all the parameter space, you don’t need to know what there will be on the next step, you just look for the negative gradient in your actual location.

- We wish to use Gradient Descent to minimize the loss of a classifier.
- Idea:
  1. Start at any place $x = x_0$ in the “parameter space”.
  2. Consider the *local* shape of the loss function by computing the gradient at the current position x. Note that the gradient points in the direction of steepest ascent.
  3. Take a “step” in the direction of the negative gradient to decrease the loss, arriving at a new position x.
  4. Repeat steps 2 and 3 until satisfied.



Img src: Wikipedia, *Gradient Descent*

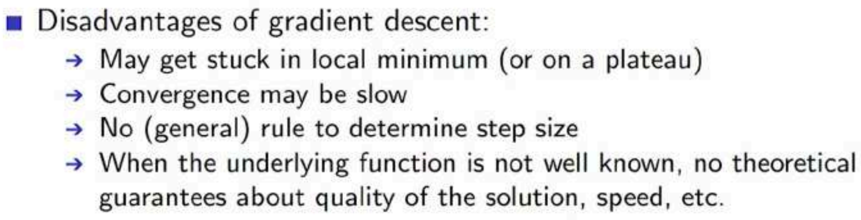
The only problem of the Gradient Descent is that if the function is very steep we make a large step, if the function is not steep you make a small step. We need to define an **hyperparameter** that optimises the “**step size**”.



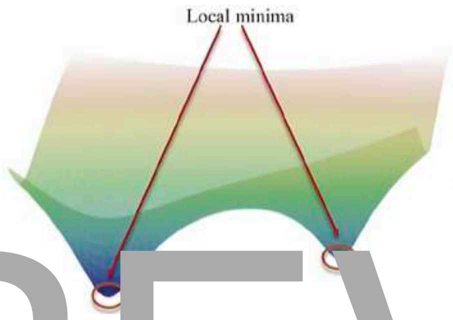
- Advantages of gradient descent:
  - → Conceptually simple and flexible
  - → Works for any underlying function, only constraint: gradient must be defined and computable
  - → Works in any dimensionality (even in infinite-dimensional spaces)
  - → May offer a computationally tractable solution when other methods fail (e.g. for large amounts of training samples, high-dimensional spaces)
  - → Iterative approach allows a lot of flexible engineering where necessary

To compute the gradient you just use derivatives.

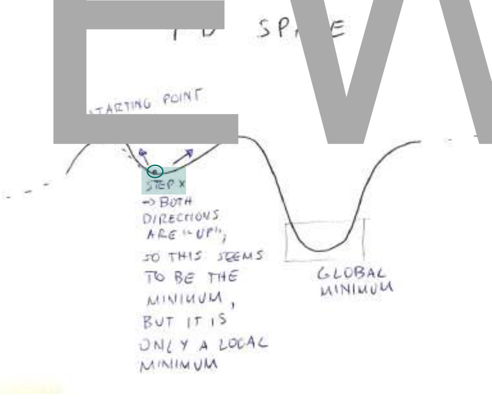
This is the first iterative approach we introduce in this course: I don't directly compute an exact solution, but I try to approximate the solution to an optimal one step by step. This is the concept of **iteratively improving a system**, which is almost always used in Machine Learning (this is what is called training).



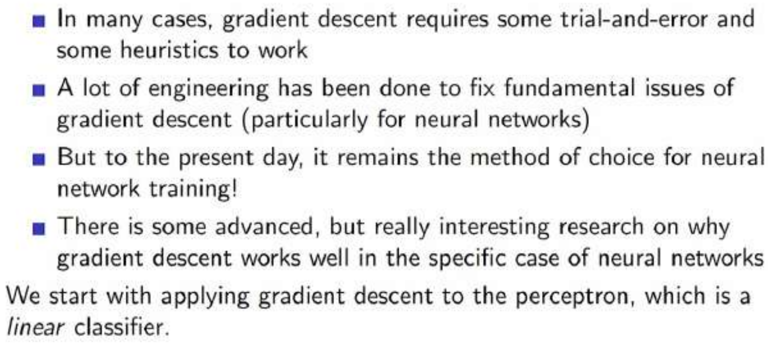
- Disadvantages of gradient descent:
  - → May get stuck in local minimum (or on a plateau)
  - → Convergence may be slow
  - → No (general) rule to determine step size
  - → When the underlying function is not well known, no theoretical guarantees about quality of the solution, speed, etc.



In gradient descent you may find the local minimum, but not the global one, which would be more significative.
For very high dimensional function is **improbable** to have local minima. As for recent studies, **neural networks** always find the global minimum. To understand this, think at the global minimum in a 1-dimensionality space -> A function in a1-d space is a curve, this curve may have several minima. Since we are in a 1-d space, in order to have a local minimum, we need to define **2 conditions**: a point


1 D SPACE
STARTING POINT
STEP X
-> BOTH DIRECTIONS ARE "UP", SO THIS SEEMS TO BE THE MINIMUM, BUT IT IS ONLY A LOCAL MINIMUM
GLOBAL MINIMUM

The local minimum [illegible] in both direction (before and after it) the line goes up in a 1-d space if both directions (before and after the point) go up, then that is a minimum. In a 2-d space the conditions become [illegible] and so on with the increase of dimensionality. In neural networks we are dealing with such high dimensionality that a local minima must satisfy an extremely high number of conditions.
With such high dimensionality, only the global minimum can satisfy all the conditions at the same time. This is why for neural network local minima seems to not exists, i.e. all these conditions are satisfied only in the global minima.



- In many cases, gradient descent requires some trial-and-error and some heuristics to work
- A lot of engineering has been done to fix fundamental issues of gradient descent (particularly for neural networks)
- But to the present day, it remains the method of choice for neural network training!
- There is some advanced, but really interesting research on why gradient descent works well in the specific case of neural networks

We start with applying gradient descent to the perceptron, which is a *linear* classifier.

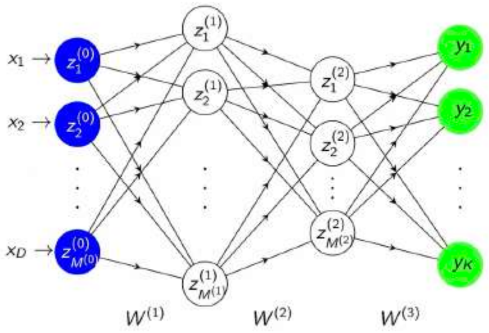
ML Lesson 10

Backpropagation: Backpropagation is just to implement gradient descent across neural networks. This lesson will introduce back propagation.

Recap of last lecture

Representation of a Neural Network

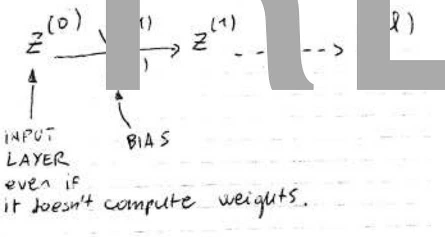
In the neural network each neuron aggregates the input to produce a scalar output. This aggregation is a weighted linear combination of the inputs and the bias (not drawn in the image) followed by a non linearity. The parameters that makes a Neural Network assume a certain behaviour are the weights of the connections. Please note that the computation is not performed independently for each neuron's, but it is a **matrix multiplication** followed by component-wise non-linearity.

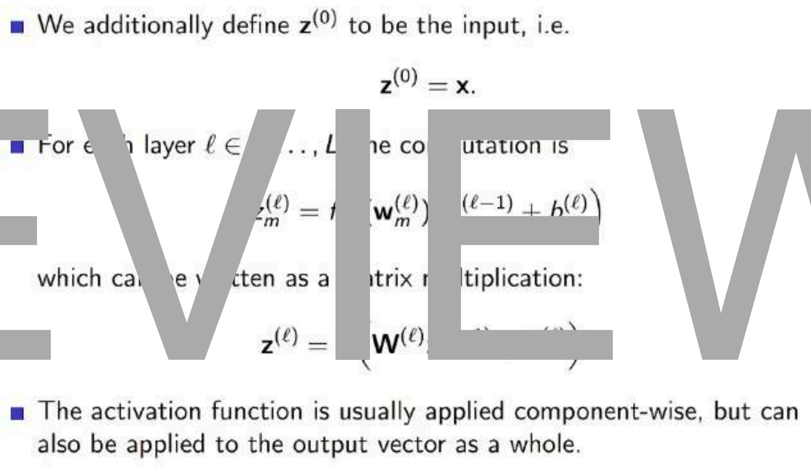


The image graphically represents a neural network with three layers, or two *hidden* layers. Computation runs from the left to the right. Note that $M^{(0)} = D$ and $M^{(3)} = K$

Feedforward Full-Connected Neural Networks

This is the description above in mathematical terms.
it's just a linear mapping + component-wise non-linearity.
NW. the [illegible] n hyperparameter. The operation can be simplified as





- We additionally define $\mathbf{z}^{(0)}$ to be the input, i.e.

$$\mathbf{z}^{(0)} = \mathbf{x}.$$

- For each layer $\ell \in \{1, \ldots, L\}$ the computation is

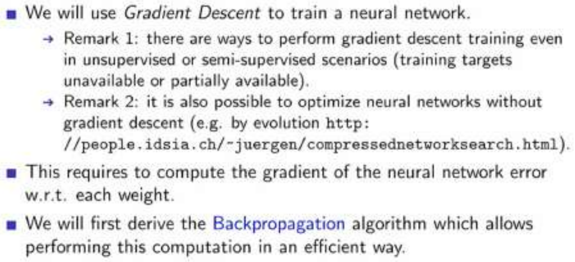
$$z_m^{(\ell)} = f\left((\mathbf{w}_m^{(\ell)})^T \mathbf{z}^{(\ell-1)} + b^{(\ell)}\right)$$

which can be written as a matrix multiplication:

$$\mathbf{z}^{(\ell)} = f\left(\mathbf{W}^{(\ell)} [illegible]\right)$$

- The activation function is usually applied component-wise, but can also be applied to the output vector as a whole.

Gradient Descent by Backpropagation

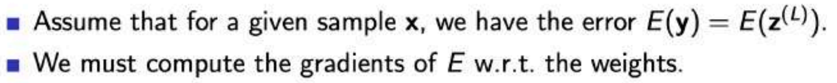


- We will use *Gradient Descent* to train a neural network.
  - Remark 1: there are ways to perform gradient descent training even in unsupervised or semi-supervised scenarios (training targets unavailable or partially available).
  - Remark 2: it is also possible to optimize neural networks without gradient descent (e.g. by evolution `http://people.idsia.ch/~juergen/compressednetworksearch.html`).
- This requires to compute the gradient of the neural network error w.r.t. each weight.
- We will first derive the Backpropagation algorithm which allows performing this computation in an efficient way.

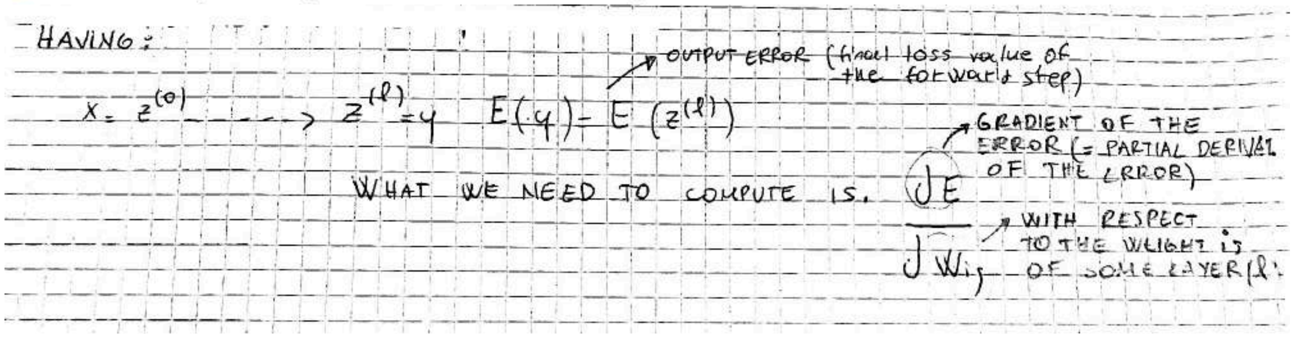
Gradient descent fundamentally you have a space of trainable parameters (or weights), and at each iteration you take a step into the direction of the steepest descent. Heading to the steepest descent of this space means to minimise the loss. Ideally if the descent is not so steep, we are closer to a minimum, so we should take a small step, otherwise, if it is very steep, we are far from the minimum and thus we just take a larger step. The stepwise is a hyper parameter.
If the step is too small it takes long to train, if its too large the error would explode to infinity since you cannot find the minimum and you just bounce around the space.
The problem is that if we are working with a neural network, we do not have a 2 or 3 dimension space, but we have up to billions of parameters to optimise, thus a billion-dimensional space of

trainables. Backrpopagation aims to compute gradient descent in a space with billions of possible steps directions.
The forward pass was to compute the loss (error) at each layer, from z⁰ to zᴸ . With backpropagation we take the error of the last error and proceed backward: we compute the error of the previous layer and so on. *The computation of the weight goes backwards.*

Backpropagation Training
In back propagation, what we have E(y) which is the error of the output produced by the neural network, i.e. the error of the last layer (so the error should be a scalar). What we need to compute is the gradient of the error with respect to some of the weights.



- Assume that for a given sample **x**, we have the error $E(\mathbf{y}) = E(\mathbf{z}^{(L)})$.
- We must compute the gradients of E w.r.t. the weights.



HAVING:

$x = z^{(0)} \dashrightarrow z^{(\ell)} = y \quad E(y) = E(z^{(\ell)})$

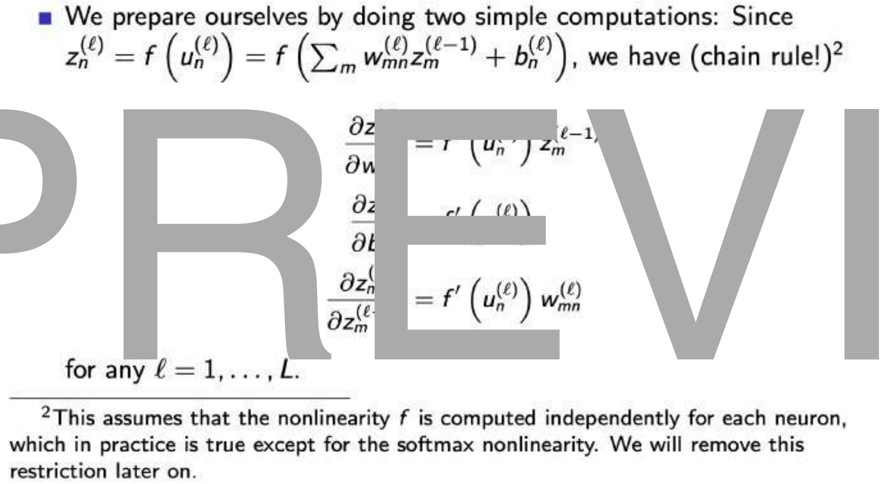
→ OUTPUT ERROR (final loss value of the forward step)

WHAT WE NEED TO COMPUTE IS. $\frac{\partial E}{\partial W_{ij}}$

→ GRADIENT OF THE ERROR (= PARTIAL DERIVAT. OF THE ERROR)

→ WITH RESPECT TO THE WEIGHT ij OF SOME LAYER (ℓ).

So, since we cannot compute each one of the billion weights one by one, we use the following formulation, which is based on the neural network computation using matrix multiplication:



- We prepare ourselves by doing two simple computations: Since $z_n^{(\ell)} = f\left(u_n^{(\ell)}\right) = f\left(\sum_m w_{mn}^{(\ell)} z_m^{(\ell-1)} + b_n^{(\ell)}\right)$, we have (chain rule!)[2]

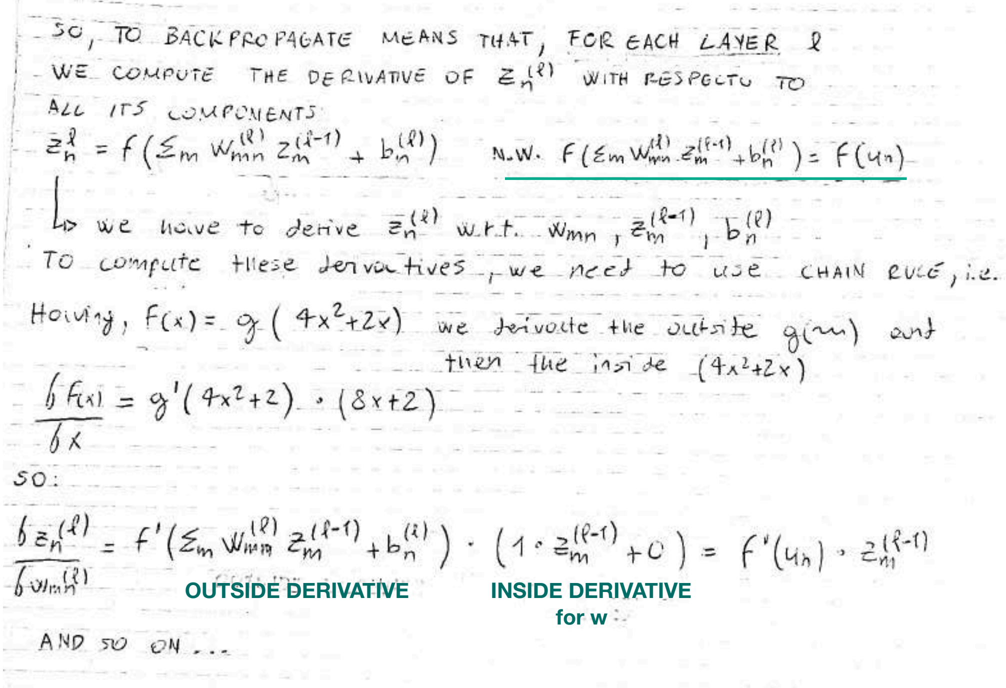
$$\frac{\partial z}{\partial w} = f'\left(u_n\right) z_m^{(\ell-1)}$$

$$\frac{\partial z}{\partial b} = f'\left(u^{(\ell)}\right)$$

$$\frac{\partial z_n^{(}}{\partial z_m^{(\ell}} = f'\left(u_n^{(\ell)}\right) w_{mn}^{(\ell)}$$

for any $\ell = 1, \ldots, L$.

[2] This assumes that the nonlinearity f is computed independently for each neuron, which in practice is true except for the softmax nonlinearity. We will remove this restriction later on.



SO, TO BACKPROPAGATE MEANS THAT, FOR EACH LAYER ℓ WE COMPUTE THE DERIVATIVE OF $z_n^{(\ell)}$ WITH RESPECTU TO ALL ITS COMPONENTS

$z_n^{\ell} = f(\Sigma_m W_{mn}^{(\ell)} z_m^{(\ell-1)} + b_n^{(\ell)})$ N.W. $f(\Sigma_m W_{mn}^{(\ell)} z_m^{(\ell-1)} + b_n^{(\ell)}) = f(u_n)$

↳ we have to derive $z_n^{(\ell)}$ w.r.t. W_{mn}, $z_m^{(\ell-1)}$, $b_n^{(\ell)}$

TO compute these derivatives, we need to use CHAIN RULE, i.e.

Having, $F(x) = g(4x^2+2x)$ we derivate the outside $g(\ldots)$ and then the inside $(4x^2+2x)$

$$\frac{\delta F(x)}{\delta x} = g'(4x^2+2) \cdot (8x+2)$$

SO:

$$\frac{\delta z_n^{(\ell)}}{\delta w_{mn}^{(\ell)}} = f'(\Sigma_m W_{mn}^{(\ell)} z_m^{(\ell-1)} + b_n^{(\ell)}) \cdot (1 \cdot z_m^{(\ell-1)} + 0) = f'(u_n) \cdot z_m^{(\ell-1)}$$

OUTSIDE DERIVATIVE

INSIDE DERIVATIVE for w

AND SO ON...

The idea of back propagation is to compute a **sub-gradient** between some output to any of the weights of the connection with the previous layer. This is a simpler solution since between two layers it is a linear mapping + some non linearity.

δW_{ij}

$z^{(l)} = \begin{pmatrix} z_1^{(l)} \\ \vdots \\ z_{M^{(l)}}^{(l)} \end{pmatrix}$ IS A VECTOR CONTAINING ALL THE NEURON'S OUTPUTS FOR LAYER l.

$$\begin{tikzcd} z^{(l-1)} \arrow[r] & z^{(l)} \\ W_{ij}^{(l)} \arrow[u] & \end{tikzcd}$$

$W_{ij}^{(l)}$ → THE CONNECTION BETWEEN EACH LAYER IS WEIGHTED BY $W_{ij}^{(l)}$, WHICH IS A VECTOR OF WEIGHTS WEIGHTING THE CONNECTION BETWEEN NEURON i AND j.

COMPUTING BACKPROPAGATION MEANS:

$$\begin{tikzcd} z^{(0)} \arrow[r, "W_{ij}^{(0)}"'] & z^{(1)} \arrow[r, "W_{ij}^{(1)}"'] & z^{(2)} \arrow[r, dashed, "W_{ij}^{(2)}"'] & \cdots \arrow[r, dashed] & z^{(l-1)} \arrow[r, "W_{ij}^{(l-1)}"'] & z^{(l)} \end{tikzcd}$$

$$\frac{\delta E}{\delta W_{ij}^{(1)}} \qquad \frac{\delta E}{\delta W_{ij}^{(l-1)}}$$

TO COMPUTE THE GRADIENT OF THE ERROR WITH RESPECT TO THE WEIGHT OF THE CONNECTION WITH THE PREVIOUS LAYER

How do we use this concept (and the calculation seen above) to compute more complex derivatives?

- For the *last* layer, we can now immediately compute the gradients:

$$\frac{\partial E}{\partial w_{mn}^{(L)}} = \frac{\partial E}{\partial z_n^{(L)}} \frac{\partial z_n^{(L)}}{\partial w_{mn}^{(L)}} = \frac{\partial E}{\partial z_n^{(L)}} f'\left(u_n^{(L)}\right) z_m^{(L-1)}$$

$$\frac{\partial E}{\partial b_n^{(L)}} = \frac{\partial E}{\partial z_n^{(L)}} \frac{\partial z_n^{(L)}}{\partial b_n^{(L)}} = \frac{\partial E}{\partial z_n^{(L)}} f'\left(u_n^{(L)}\right).$$

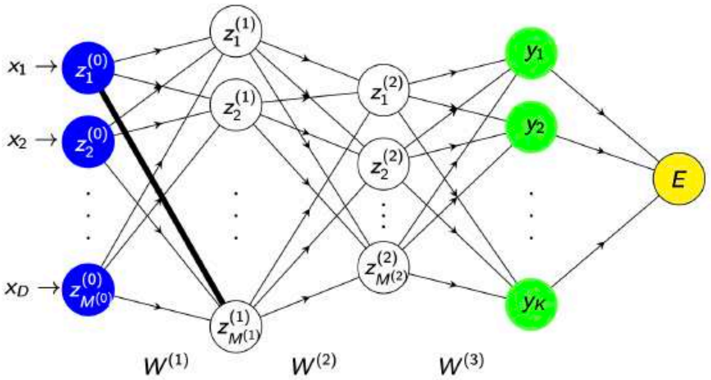
[illegible] IS [illegible] T[illegible] AS ABOV[illegible] INFACT IF [illegible]U [illegible]

$\frac{\delta E}{[illegible]} = [illegible] (u_n^{(L)}) \cdot [illegible]_m^{(L-[illegible]}$, where L [illegible] last [illegible]yer.

The fir[illegible] passa[illegible]e [illegible]s the conseque[illegible] of [illegible]he [illegible]HAIN RULE

→ if $f(x[illegible] = g(h[illegible])$ $[illegible] \frac{\delta g}{\delta x}\,\frac{[illegible]}{\delta g}\,\frac{\delta g}{\delta n}$

- This computation is easiest for the last layer, since there is only one "path" in which the weight $w_{mn}^{(L)}$ influences the error[3].
- Let us now consider the general case.



The situation is slightly more complicated for the lower layers, since we need to consider *all* paths which lead to a certain weight. In how many ways does the indicated weight influence the loss?

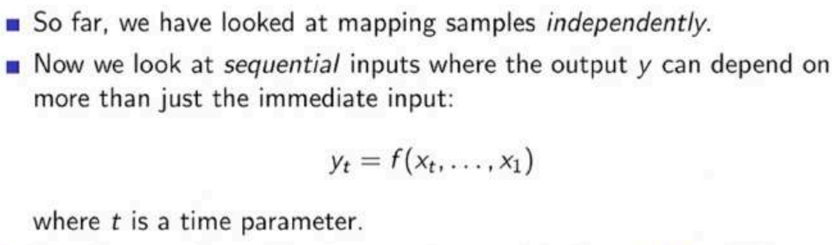
The calculation for the last layer is easy because this weights ($W^{(3)}$) influences the error in just one passway. To compute how previous weights (i.e. $W^{(1)}$) influence the final Error, is a nightmare.
By changing $W^{(1)}$, the E changes. The problem is to understand why it changes, since the influence of $W^{(1)}$ on the Error passes through a multitude of different pathways.

Machine Learning - Lesson 11

Advanced Architecture for Neural Networks and Sequence Modeling

Recurrent Neural Networks can be introduced in two different ways:

Introduction (1)



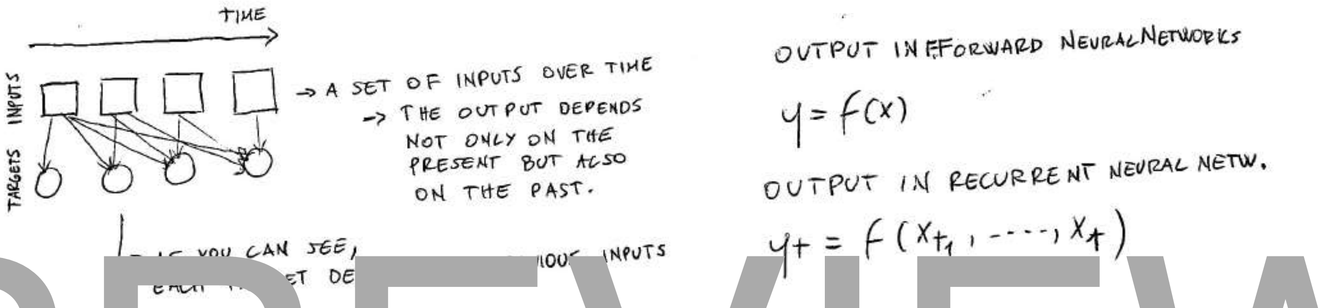
- So far, we have looked at mapping samples *independently*.
- Now we look at *sequential* inputs where the output y can depend on more than just the immediate input:

$$y_t = f(x_t, \dots, x_1)$$

where t is a time parameter.

Mapping a sample independently, means to define the relationship between sample (input data) and its own target. In the problems seen before, the sample was independent, i.e. it wasn't linked to other samples. Recurrent Neural networks are therefore NN where the sample are related together, and thus cannot be considered independently. *The output y can depend on more than just the immediate output*. This means, basically, to work with sequences as inputs. For example working with video -> each frame is linked to the previous. Recurrent NN look at **sequential inputs**.

In a simpler way we have that:

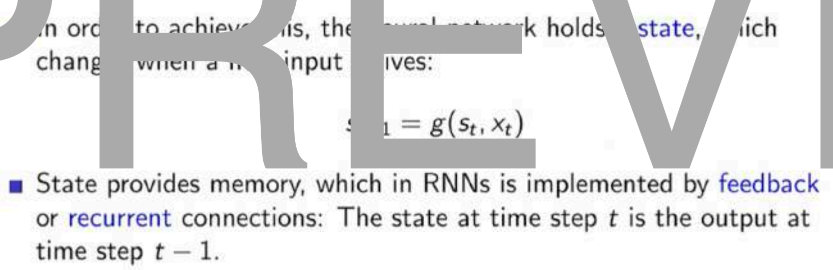

TIME
INPUTS
TARGETS
-> A SET OF INPUTS OVER TIME
-> THE OUTPUT DEPENDS NOT ONLY ON THE PRESENT BUT ALSO ON THE PAST.
AS YOU CAN SEE, [illegible] PREVIOUS INPUTS
EACH TARGET DE[illegible]

OUTPUT IN FFORWARD NEURAL NETWORKS

$y = f(x)$

OUTPUT IN RECURRENT NEURAL NETW.

$y_t = f(x_{t_1}, \dots, x_t)$



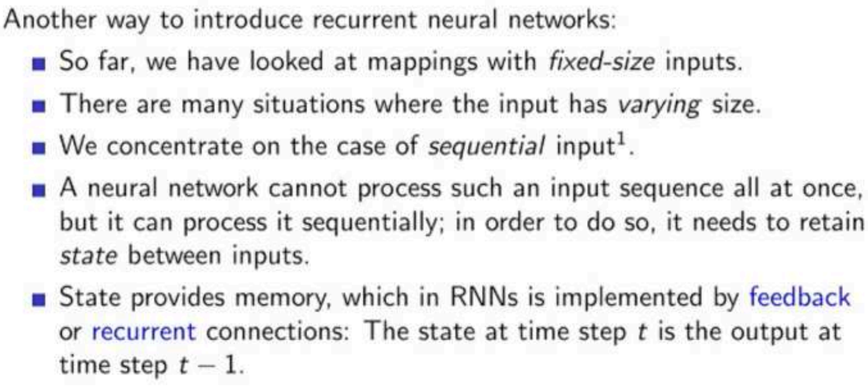
In order to achieve this, the neural network holds a state, which changes when a new input arrives:

$$s_{t+1} = g(s_t, x_t)$$

- State provides memory, which in RNNs is implemented by feedback or recurrent connections: The state at time step t is the output at time step $t-1$.

In summary: **RNN** working with samples that depend on the past. In order to model such a function to which we sequentially [illegible] inputs with are sequential [illegible] depend on the past, the NN need to hold a **state.** While FFNN and CNN, there were trained weights, here, even if the weights don't change, the NON needs some memory to store and process one input after the other. In this definition of the state, **s_t** is the output of the previous step.

Introduction (2)



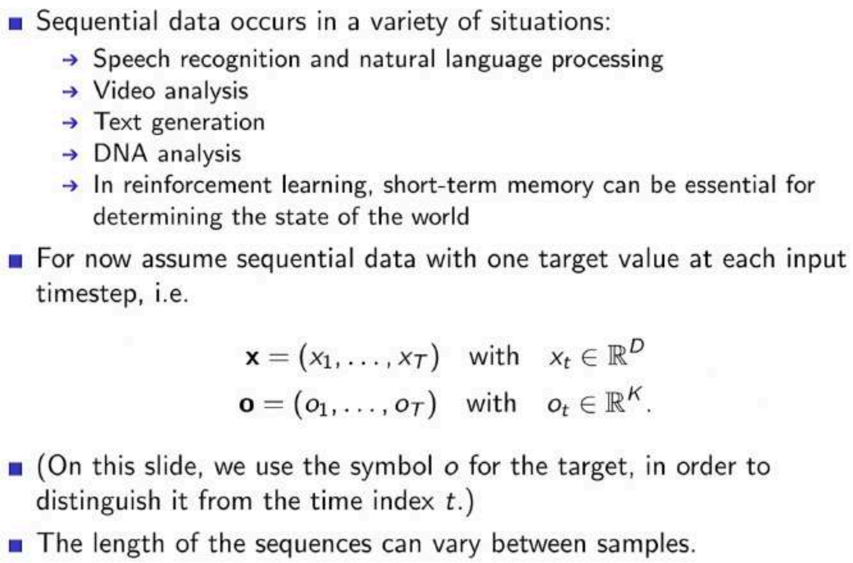
Another way to introduce recurrent neural networks:

- So far, we have looked at mappings with *fixed-size* inputs.
- There are many situations where the input has *varying* size.
- We concentrate on the case of *sequential* input[1].
- A neural network cannot process such an input sequence all at once, but it can process it sequentially; in order to do so, it needs to retain *state* between inputs.
- State provides memory, which in RNNs is implemented by feedback or recurrent connections: The state at time step t is the output at time step $t-1$.

Another way to introduce RNN. In FFNN we look at NN where if the output was o 1000 neutrons, our input needed to have exactly 1000 neurons. This is quite true also for CNN, since they can handle different size inputs, but at the end they return an output which is roughly the same size -> CNN converge an input of a certain size into an output of roughly the similar size. NW. in the last part of CNN we use a fully

connected layer. This fully connected layer requires an input which is similar to the output. This means that the input (i.e. the output of all previous convolutional alters) must be roughly of the same size as the output of this last FC layer. So even in CNN the input, while not of fixed-size, must be someway similar to the output (if the output is a 1000x1000 pixel image, we cannot provide inputs of 1000x1 pixels).
RNN can instead work with different size input and output -> due to the fact that RNN processes the input step by step, RNN can deal with inputs of various sizes. Also, the input might also vary in dimensionality, it can slo be multi-dimensional.

Sequence Modelling



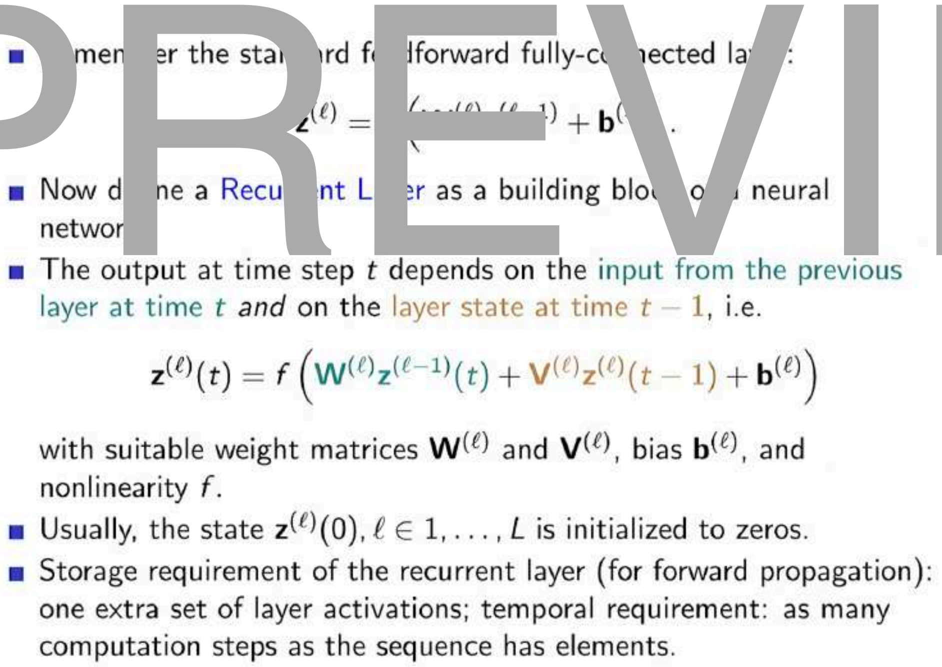
- Sequential data occurs in a variety of situations:
  - → Speech recognition and natural language processing
  - → Video analysis
  - → Text generation
  - → DNA analysis
  - → In reinforcement learning, short-term memory can be essential for determining the state of the world
- For now assume sequential data with one target value at each input timestep, i.e.

$$\mathbf{x} = (x_1, \dots, x_T) \quad \text{with} \quad x_t \in \mathbb{R}^D$$
$$\mathbf{o} = (o_1, \dots, o_T) \quad \text{with} \quad o_t \in \mathbb{R}^K.$$

- (On this slide, we use the symbol o for the target, in order to distinguish it from the time index t.)
- The length of the sequences can vary between samples.

Let's assume we have sequential data with one target value at each input. The means we would have an alignment between input and output.

Recurrent Layer



- Remember the standard feedforward fully-connected layer:

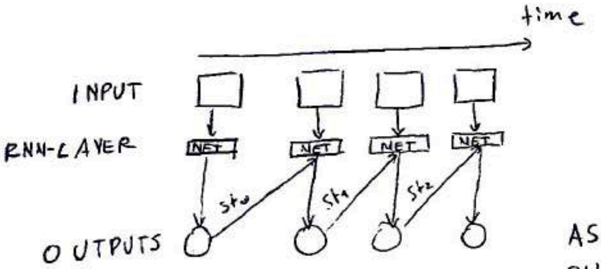
$$\mathbf{z}^{(\ell)} = f\left(\mathbf{W}^{(\ell)}\mathbf{z}^{(\ell-1)} + \mathbf{b}^{(\ell)}\right).$$

- Now define a Recurrent Layer as a building block of a neural network.
- The output at time step t depends on the input from the previous layer at time t *and* on the layer state at time $t-1$, i.e.

$$\mathbf{z}^{(\ell)}(t) = f\left(\mathbf{W}^{(\ell)}\mathbf{z}^{(\ell-1)}(t) + \mathbf{V}^{(\ell)}\mathbf{z}^{(\ell)}(t-1) + \mathbf{b}^{(\ell)}\right)$$

with suitable weight matrices $\mathbf{W}^{(\ell)}$ and $\mathbf{V}^{(\ell)}$, bias $\mathbf{b}^{(\ell)}$, and nonlinearity f.
- Usually, the state $\mathbf{z}^{(\ell)}(0), \ell \in 1, \dots, L$ is initialized to zeros.
- Storage requirement of the recurrent layer (for forward propagation): one extra set of layer activations; temporal requirement: as many computation steps as the sequence has elements.

In order to start this operation we need first to initialise the state at time 0. You just need to remember the previous layer, not all the past (except for task where you would need it).

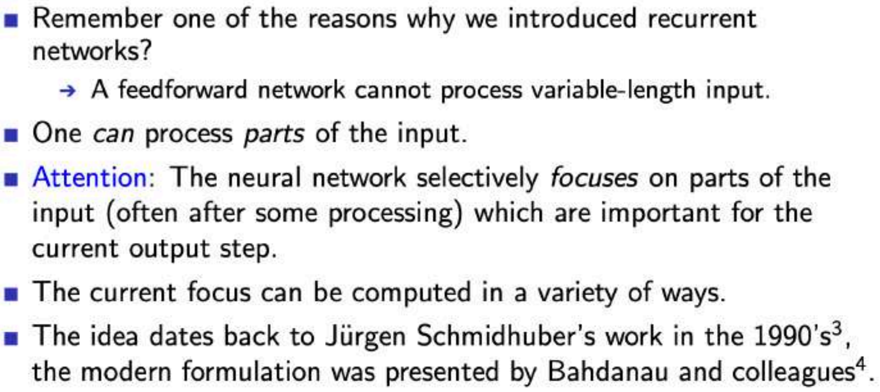


AS YOU CAN SEE, THE PREVIOUS OUTPUT IS PASSED TO THE NEXT LAYER, AND SO ON.

Machine Learning Lesson 16

Attention Mechanism

Introducing Attention



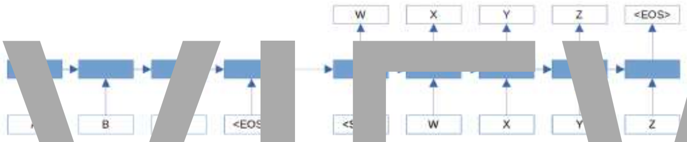
- Remember one of the reasons why we introduced recurrent networks?
  - → A feedforward network cannot process variable-length input.
- One *can* process *parts* of the input.
- Attention: The neural network selectively *focuses* on parts of the input (often after some processing) which are important for the current output step.
- The current focus can be computed in a variety of ways.
- The idea dates back to Jürgen Schmidhuber's work in the 1990's[3], the modern formulation was presented by Bahdanau and colleagues[4].

On of the reasons why we introduced RNN is that the FullyConnectedNN cannot really process inputs of arbitrary length. If we got a sequence that we know has always the same length, we could still use FCNN, but if we have sequences with variable length, the only way to process them is to process elements of the sequence one by one with RNN. Thus, the FFNN could not process variable length input, but it could process parts of the input (process this part and this part and this part and so on).

What attention means? The NN processes a sequence and at each processing step it focuses selectively on a part of the input which is interesting for the current output (this is **Attention**). The idea is that we have some connection from the input to the output which are not as rigid as the encode-decode architecture.

Quick reminder on encoder-decoder architecture

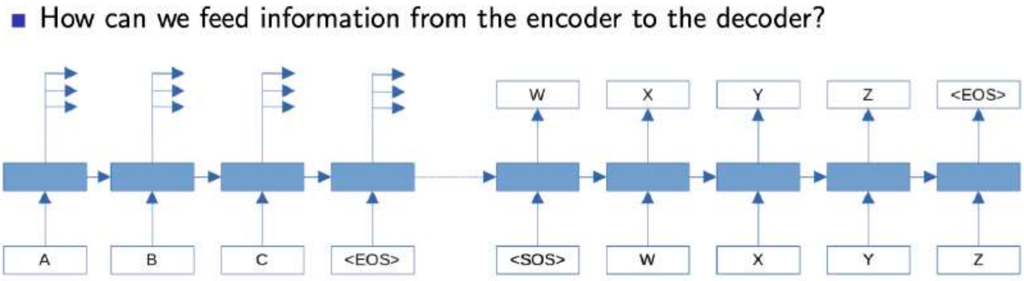
Each of the blue rectangle is a step. This step is needed with a token input (arrow coming towards the rectangle) and can produce an output token (arrow coming out the rectangle)



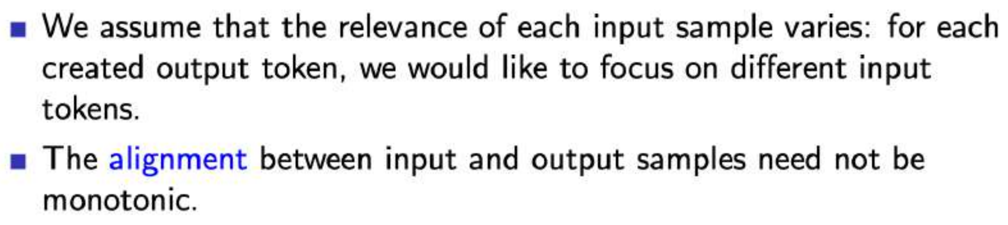
1. We read the inputs until the EOS token
2. We feed a SOS token to tell our system to start processing the output. In the same step the first output is produced (w). The output of this step becomes the input for the next step which taking as input w produces the output x.

Advantages: making the training faster and being able try out different path. To understand this: note that **w** is the token output with the maximal probability. Since the output of a decoder step (suppose **w**) is parsed by a sigmoid (so a value between 0 and 1), we get that w is the target with the highest probability. But there may be other token with slightly less probability (that are not returned as output by the step, since the step returns only the one with the highest probability). **w** is the most likely token. Suppose we have **d** as a token less likely then **w**. We can feed **d** as an input of the next step (instead of feeding **w**). *We are not looking at maximising the likelihood of each single token, but we are looking at maximising the probability of the entire sequence.* ***So feeding the next step with a less likely token may lead to a general higher probability/ likelihood of the sequence.*** By doing these you can produce different paths and see which is the one that, at the end, maximises the probability of the entire sequence. This is what **Beam Search does:** it looks to different sequences (i.e. paths) and chooses the best.

Basic Attention Mechanism

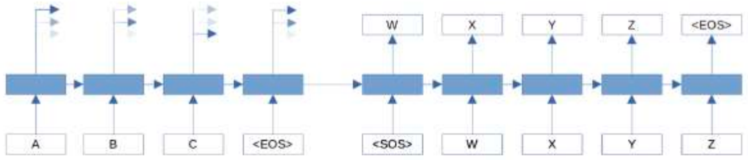


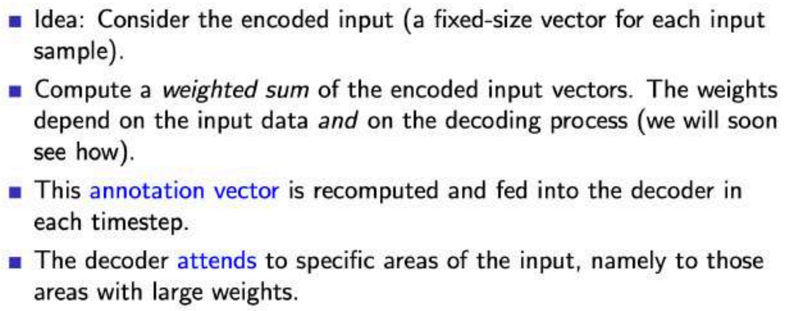
- How can we feed information from the encoder to the decoder?



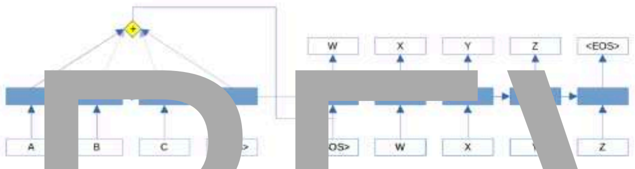
- We assume that the relevance of each input sample varies: for each created output token, we would like to focus on different input tokens.
- The alignment between input and output samples need not be monotonic.

For instance we might have the situation in which the input sample A is relevant just for the beginning of the output(first arrow thick), the B input is relevant just for the second step (second arrow thick) and a bit for the first step, but not for the next.



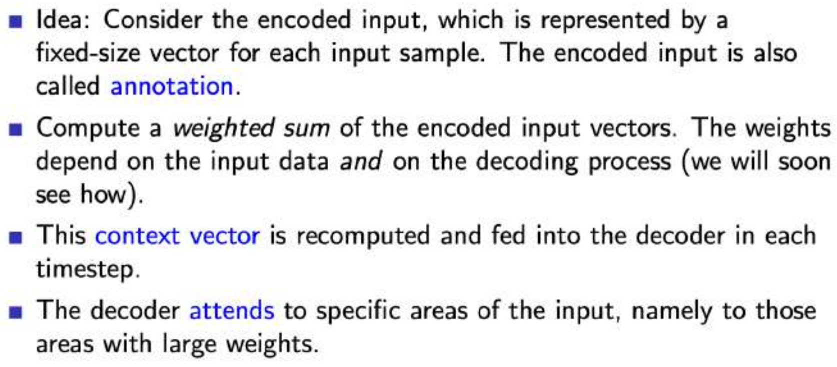


- Idea: Consider the encoded input (a fixed-size vector for each input sample).
- Compute a *weighted sum* of the encoded input vectors. The weights depend on the input data *and* on the decoding process (we will soon see how).
- This annotation vector is recomputed and fed into the decoder in each timestep.
- The decoder attends to specific areas of the input, namely to those areas with large weights.



The Idea of the Attention Mechanism is that for each step we take a weighted sum and feed it into the decoder.
How does this feeding works? Since we have a same, we got a **fixed size** representation of the input sequence. We are simply feeding the decoder with a larger input. This is the attention. The point is, how doe we compute these weights? To recap, each step (blue rectangle) is a LSTM or RNN, which means that produce [illegible]. The arrows that goes out from the input steps are the weights we are talking about. So we do not compute the output directly (since in [illegible] architecture first you process the input and then the outputs), but, for each input step” we can compute a weight that is linked with its input and the future output in the decoder. What the diagram is showing is that for output Y we need to focus on input C. These weights depend on the input data a[illegible] the output. How do we compute [illegible] make the NN compute them. In more precise terms, these weight depends on the input provided in its encoding step, and on the **state of the decoding step** (not on its output).

Bahdanau Attention



- Idea: Consider the encoded input, which is represented by a fixed-size vector for each input sample. The encoded input is also called annotation.
- Compute a *weighted sum* of the encoded input vectors. The weights depend on the input data *and* on the decoding process (we will soon see how).
- This context vector is recomputed and fed into the decoder in each timestep.
- The decoder attends to specific areas of the input, namely to those areas with large weights.

In a trained NN the weights are fixed, here the weights depend on the data.