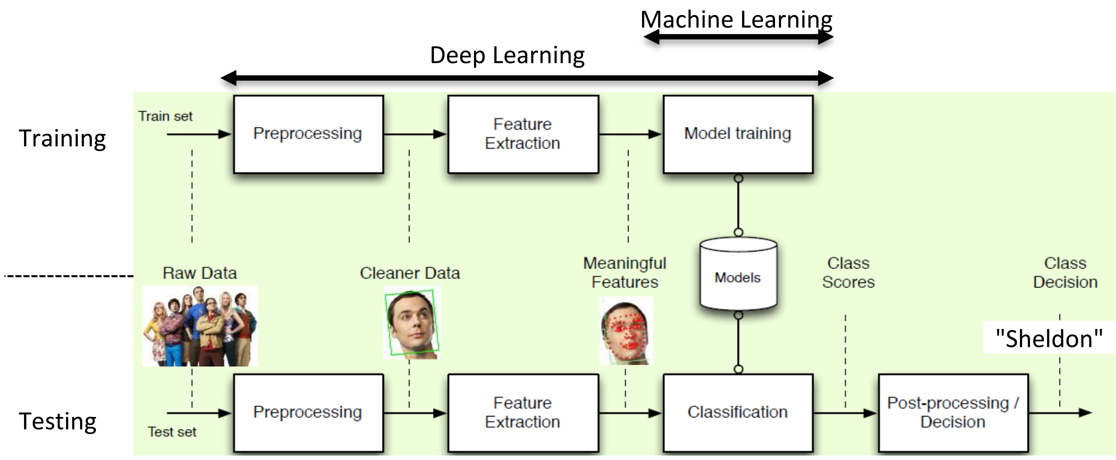
DEEP LEARNING (DL)

1. Introduction

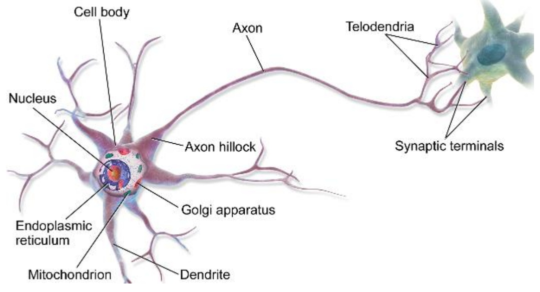
Fields of application	**Computer Vision** (e.g. self-driving cars, object detection, medical diagnosis, image tagging/generation, caption generation, lip reading, lip synch from Audio) **Image synthesis** (generation of scenes in games) **Signal processing** (denoising signals, images) **Natural Language Processing** [NLP] (e.g. translation, speech recognition and synthesis, Q&A-bots)
Machine Learning (ML)	Machine learning consists computer methods that analyse observation data to automatically detect patterns, and then use the uncovered patterns to perform tasks based on new unobserved data.
convergence	**Mathematics** (probability theory, statistics, regression, ...) **Signal processing** (filtering, feature extraction, time series, fft, ...) **Software Engineering** (very large DB, intensive CPU, distributed programming, ...) **Application domain** (finance, medicine, energy, biometrics, ...) **Algorithmic**
tasks	**Classifying**: Predict label of n mutually exclusive classes, e.g. classifying mails as spam or ham **Predicting**: Predict numerical value (response), e.g. predicting house prices in given location and size **Clustering**: Arrange set of entities into groups, e.g. group patients/clients for similar 'treatment' **Robotic Tasks**: e.g. robot conducting certain tasks (walking, cutting the lawn, cleaning, washing dishes)
process	
Paradigms	With **supervised learning**, the goal is to extract some relevant features **x** from raw observation data **o** and to learn a mapping from inputs **x** to outputs **y** given a set of **example** data called the training set. **Labelled** data. Applications: Recognition, Planning, Diagnosis, Robot Control, Prediction
	With **unsupervised** learning, the goal is to discover interesting **structures** from inputs **x** given a set of **data** called the training set. **Unlabelled** data. Applications: Market Segmentation, Astronomical and Solar Data Analysis, Social Network Analysis.
	With **reinforcement learning**, we learn the behaviour in an environment that provides suitable **rewards**. Applications: Learn to play games, solve tasks
Deep Learning (DL)	A sub-branch of machine learning. Neuronal architecture with many layers and neurons.
convergence	**Larger quantities of data** (text, audio, images, videos, ...) -> scalability of learning on very large data sets **New algorithms** (DBN, RBM, CNN, ...) -> ability to learn feature extraction in unsupervised mode and classification in supervised mode **Better computer performance** (GPU, distributed computing, ...) -> train complex mapping functions
Machine Learning vs Deep Learning	Machine Learning typically **requires significant hand-engineering of features**. Deep Learning requires **less feature engineering** than standard machine learning. In DL, the machines find the features **automatically** as part of learning. DL has more flexible and powerful models but are more difficult and need more learn data. DL solves the problems in supervised learning.
supervised learning problems	1. We need **large quantities** of human validated examples! ... and this is **costly** to build. 2. Because of the variabilities, we will need even **more data** and **complex mapping functions**. 3. We spend a lot of time to hand-craft interesting compact features, so called **feature engineering**.
Deep learning answers	1. Let's use all the labelled data and unlabelled data 2. Let's use deep neural networks 3. Let's **learn the feature extraction in unsupervised learning mode**.

1. Perceptron

Biological Neural Systems

Biological Neurons

composed of: cell body, dendrites, axon
composed in a **network** by synaptic terminals
electrical impulses '**signals**' are sent via synapsis
if a neuron receives enough signals, it fires its own signal **'activation'**



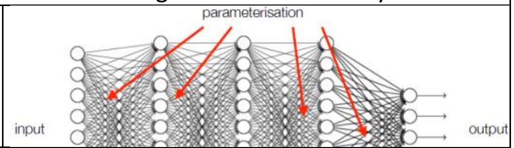
Biological Neural Nets

The connectivity in biological neural systems is huge.
neurons in the brain: $\sim 10^{11}$
connections per neuron: $\sim 10^{4}$
$\rightarrow \sim 10^{15}$ synapses

Neuro-Science and Artificial Neural Networks

Inspired by the biological brain. **Reverse-engineering** the computational principles behind the brain. Deep Learning goes beyond the neuro-scientific perspective and appeals to a more general principle of learning with multiple levels of composition. No claim to model the biological function directly.

Artificial Neural Nets are composed of **Artificial Neurons**
The network is trained to perform the task from **training data** by applying a **learning algorithm** that adjusts the networks parameters.

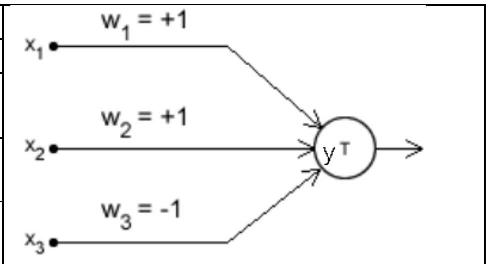


Artificial Neuron

McCulloch-Pitts Neuron (1943)

First artificial neuron as a model for the activation of a neuron.

Number of Neurons	n
Input signal	$x_k = 0{,}1 \;\; 1 \le k \le n$
Weighted	$w_k = \begin{cases} +1 \; (excitatory) \\ -1 \; (inhibitory) \end{cases}$
Sum of all input signals	$S = \sum_{k=1}^{n} w_k x_k$
Output signal	$y = \begin{cases} +1 & (S \ge \Theta) \\ 0 & (S < \Theta) \end{cases}$



Examples

$$AND: \quad y = H(x_1 + x_2 - 2)$$
$$OR: \quad y = H(x_1 + x_2 - 1)$$
$$XOR: \quad y = H(H(x_1 + x_2 - 0.5) + H(1.5 - x_1 - x_2) - 1.5)$$

Heaviside-function
$$H(z) = \begin{cases} 1 \; (z \ge 0) \\ 0 \; (z < 0) \end{cases}$$

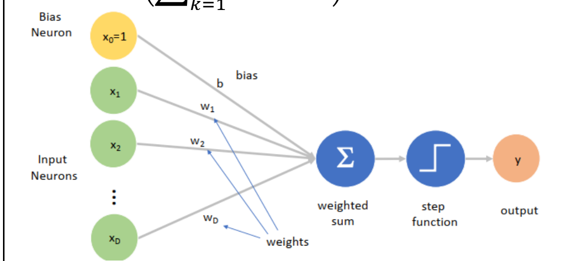
Rosenblatt's Perceptron (1958)

= LTU (Linear Threshold Unit)

for linearly separable binary classification problems
$$1 = yes, 0 = no$$
0. initialize parameters (zero or random)
1. pick sample $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)$
2. compute predicted value: $\hat{y}^{(i)} = H\left(\boldsymbol{w} * \boldsymbol{x}^{(i)} + b\right)$
3. Parameter update rule:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha * \left(\hat{y}^{(i)} - y^{(i)}\right) * \boldsymbol{x}^{(i)}$$
$$b \leftarrow b - \alpha * \underbrace{\left(\hat{y}^{(i)} - y^{(i)}\right)}_{0 \text{ or } 1}$$

$$y = H\left(\sum_{k=1}^{n} w_k x_k + b\right), x_k, w_k, b \in \mathbb{R}$$



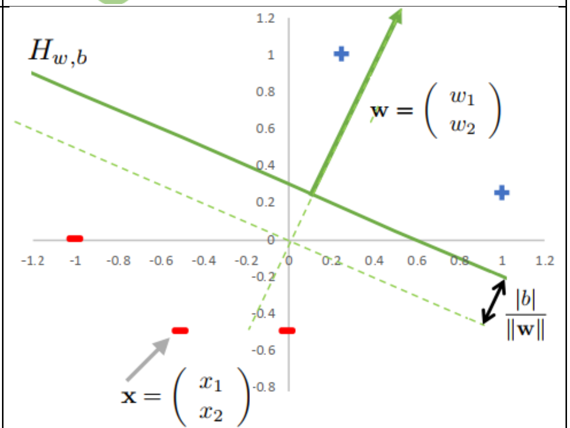
Example with 2D Input Data

Decision Boundary
$$H_{w,b} = w * x = 0$$
$\rightarrow$ dot product is 0 when orthogonal

$$H_{w,b} = w_1 * x_1 + w_2 * x_2 + b = 0$$

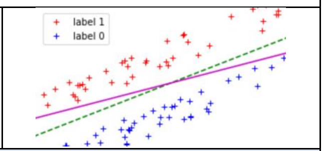
$$x_2 = \frac{-b - w_1 * x_1}{w_2}$$

$$s = -\frac{w_1}{w_2}$$



Perceptron Learning Algorithm

The learning rule searches for a weights vector that defines a hyperplane that separates the points associated with the two classes. This is only possible for linearly separable input sets. The solutions are not unique and not optimal. The optimal solution is called SVM (support vector machine)



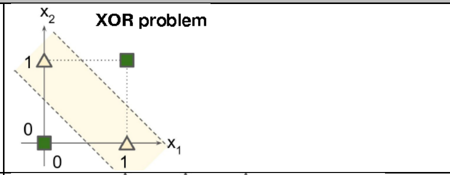
Perceptron Convergence

Perceptron Learning Algorithm converges to a weights vector and bias that separates the two classes – provided that the two classes are **linearly separable**.

Artificial Neural Nets

XOR Problem

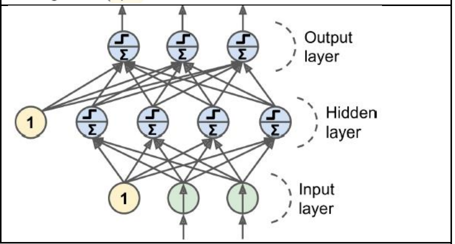
Serious weakness of perceptron's:
Incapable of solving some rather simple Problems
These limitations can be overcome by stacking multiple perceptron's so called MLP.



Multi-Layer Perceptron (MLP)

An MLP is composed of:
Input Layer: Inputs passed through
Hidden Layers: One or more layers of LTUs
Output layer: Final layer of LTU

Input and hidden layers include a bias ($x_0 = 1$) neuron and are **fully connected** to the next layer.



Usage

typically, in areas where it is easy for humans and difficult for computers.

2. Learning and Optimisation

MNIST Dataset

contains a lot of handwritten digits for testing and illustration -> identify which digit they are
2 versions: Original (70'000 28x28pixel images), Lightweight (1'800 8x8 pixel images, faster)
Binary Classification problem (is it a e.g. 5 or not), use model-function $\hat{y} = h(x)$

Data Preparation

Training and Testing

Training set: Used for learning the task.
Test Set: Used for testing how well the learned model performs.
Split in test and train data -> randomly shuffle dataset before splitting
split ratio depends on available data (large set: 99% to train, small set: 70% to train)

Data Normalisation (Feature Scaling)

Bring your values to similar scales (range and importance). Apply to input and output data.

1. Scaling: improves convergence speed and accuracy of the learning algorithm
2. Centring: improves the robustness of the learning algorithm

2 schemas

Z-Normalisation

Shifting and rescaling the data so that a zero mean and a unit-variance is obtained

$$x_k'^{(i)} = \frac{x_k^{(i)} - \mu_k}{\sigma_k}$$

compute on training set:

$\mu_k = \frac{i}{N}\sum_{k=1}^{N} x_k^{(i)}$ → mean deviation

$\sigma_k^2 = \frac{i}{N}\sum_{k=1}^{N}\left(x_k^{(i)} - \mu_k\right)^2$ → standard deviation

Min-Max Rescaling

$$x_k'^{(i)} = \frac{x_k^{(i)} - min_j\left(x_k^{(j)}\right)}{max_j\left(x_k^{(j)}\right) - min_j\left(x_k^{(j)}\right)} \rightarrow [0,1]$$

Min-Max Normalisation

$$x_k'^{(i)} = 2 * \frac{x_k^{(i)} - min_j\left(x_k^{(j)}\right)}{max_j\left(x_k^{(j)}\right) - min_j\left(x_k^{(j)}\right)} - 1 \quad \rightarrow [-1,1]$$

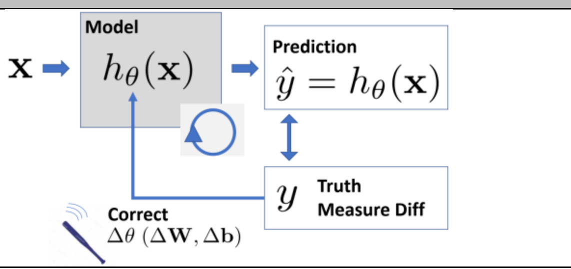
calculate min/max on training set

Notations

m	number of samples in the input dataset
n_x	number of input features, dimension of the input feature vector
$\boldsymbol{x}$	input feature vector of dimension n_x
x_k	k-th component of the input feature vector $(1 \le k \le n_x)$
$\boldsymbol{x}^{(i)}$	input feature vector of the i-th training sample $(1 \le i \le m)$
$x_k^{(i)}$	k-th component of the input feature vector of the i-th training sample
y	scalar output variable, also called target output or label
$\boldsymbol{y}$	output vector (or target output vector) of dimension n_y vector
y_k	k-th component of the output vector (or target output vector) $\left(1 \le k \le n_y\right)$
$\hat{y}$	predicted output, as computed by the mapping function
$\hat{\boldsymbol{y}}$	predicted output vector, as computed by the mapping function
$(\boldsymbol{x}, \boldsymbol{y})$	input sample (pair of input feature vector and corresponding label vector)
$\left(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\right)$	i-th input sample of the input dataset $(1 \le i \le m)$

Model and Cost

Generalised from Rosenblatt's Perceptron



1) Smooth activation function: Sigmoid

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

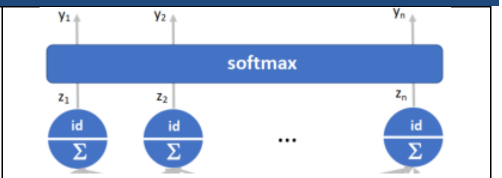
2) Gradient Descent Optimization Algorithm
- widely used in practice
- gives the direction of the steepest ascent
- works 'locally' and finds local wells (gradient=0)
-> not designed to find global minima

Learning Rate α
$\alpha > 0$

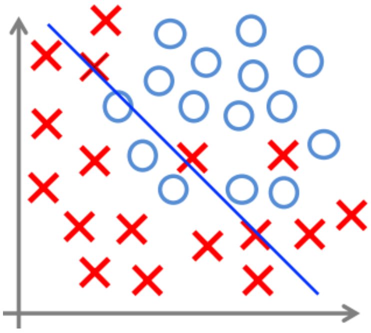
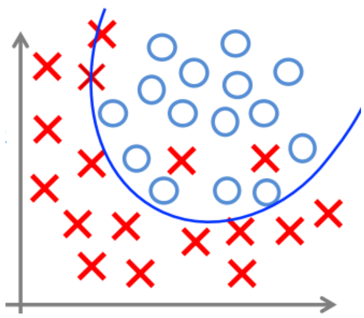
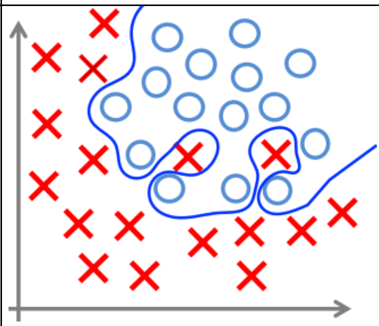
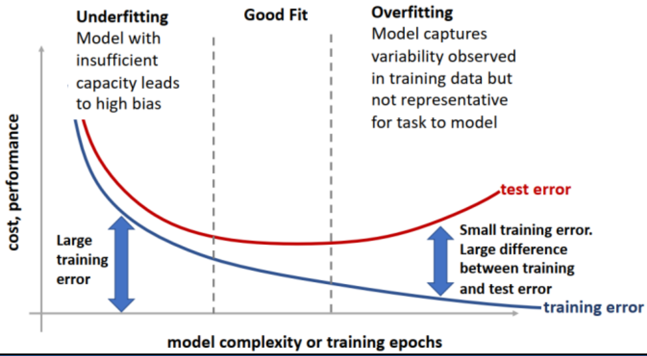
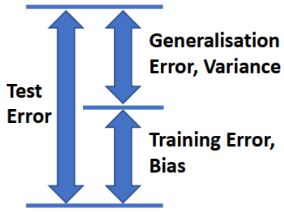
Determines the learning speed. Needs to be tuned to a given problem.
If too large, it is not guaranteed to converge. If too small -> slow convergence of cost and error rate

Epochs	Iterations run until the bottom of the valley reached. If too small -> not optimal values. If too big -> overfitting.	
Probabilistic interpretation	$p(y=1\|\boldsymbol{x},\theta)=h_\theta(\boldsymbol{x})$ $p(y=0\|\boldsymbol{x},\theta)=1-h_\theta(\boldsymbol{x})$	
Cost functions	**Mean Square Error Cost Function** $J_{MSE}(\theta)=\frac{1}{2m}\sum_{i=1}^{m}\left(\hat{y}^{(i)}-y^{(i)}\right)^2$ Square to get positiv result. Training can get stuck.	**Cross-Entropy Cost Function** $J_{CE}(\theta)=-\frac{1}{m}\sum_{i=1}^{m}\log\left(p\left(y^{(i)}\middle\|\boldsymbol{x}^{(i)},\theta\right)\right)$ **Cross Entropy in generalised perceptron** $\nabla J_{CE}(\theta)=\frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)}-y^{(i)}\right)\begin{pmatrix}\boldsymbol{x}^{(i)}\\1\end{pmatrix}$ For classification tasks. Probabilistic consideration.
	Cross-Entropy Loss Function for **binary** classification problem $\mathcal{L}(\hat{y},y)=-(y\log(\hat{y})+(1-y)\log(1-\hat{y}))$ **Cross-Entropy Cost Function** for **binary** classification problem $J_{CE}(\theta)=-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(\hat{y}^{(i)}\right)+\left(1-y^{(i)}\right)\log\left(1-\hat{y}^{(i)}\right)\right)$	
Mathematical Formulation	**Partial Derivative of Function** $J(\theta)=J(\theta_1,\dots,\theta_n)$ $\frac{\delta J}{\delta\theta_k}=\lim_{\epsilon\to 0}\frac{1}{\Delta\theta_k}\left(J(\theta_1,\dots,\theta_k+\Delta\theta_k,\dots,\theta_n)-J(\theta_1,\dots,\theta_k,\dots,\theta_n)\right)$	
	Gradient $\nabla_\theta J=\frac{\delta J}{\delta\theta}=\begin{pmatrix}\frac{\delta J}{\delta\theta_1}\\\dots\\\frac{\delta J}{\delta\theta_n}\end{pmatrix}$	**General Gradient Descent Update-Rules** vector notation: $\theta\leftarrow\theta-\alpha\nabla_\theta J(\theta)$ coordinates: $\theta_k\leftarrow\theta_k-\alpha\frac{\delta J(\theta)}{\delta\theta_k}$ **Update rules for generalised perceptron** $w\leftarrow w-\frac{\alpha}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)}-y^{(i)}\right)\boldsymbol{x}^{(i)}$ $b\leftarrow b-\frac{\alpha}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)}-y^{(i)}\right)$

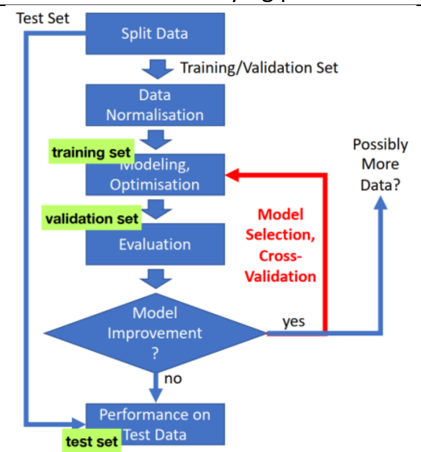
3. Shallow Networks (MLP with a single hidden layer)

SoftMax for Multi-Class	$h_{\theta,l}(\boldsymbol{x})=\frac{\exp(z_l)}{\sum_{j=1}\exp(z_j)}, where\ z_j=\boldsymbol{w}_j*\boldsymbol{x}+b$ It peaks at the largest z_l and smoothly approximates $\max\{z_1,\dots,z_{k-1}\}$ if one element is much larger than all the others. Typically, as final layer. m inputs and m outputs.	
Training	Gradient: $\frac{\delta}{\delta\boldsymbol{w}_j}J_{CE}(\theta)=-\frac{1}{m}\sum_{i=1}^{m}\left(\delta_{j,y^{(i)}}-h_{\theta,j}\left(\boldsymbol{x}^{(i)}\right)\right)\boldsymbol{x}^{(i)}$ $\frac{\delta}{\delta b_j}J_{CE}(\theta)=-\frac{1}{m}\sum_{i=1}^{m}\left(\delta_{j,y^{(i)}}-h_{\theta,j}\left(\boldsymbol{x}^{(i)}\right)\right)$ Update rules: $\boldsymbol{w}_j\leftarrow\boldsymbol{w}_j-\alpha\frac{\delta}{\delta\boldsymbol{w}_j}J_{CE}(\theta)$ $b_j\leftarrow b_j-\alpha\frac{\delta}{\delta b_j}J_{CE}(\theta)$	
Adding Hidden Layers	One additional layer with n neurons, sigmoid activation function. Can improve the performance. But in MNIST one layer already capture the correlation between pixels. How do the formulas look like when applying gradient descent networks with hidden layers?	
Role of activation function	Non linearities in the mapping between input and output of a neural network are crucial for gaining enough power for learning a task with enough accuracy. The choice of activation functions has an impact of robustness and performance.	
Universal Approximation Theorem	A feedforward network with a linear output layer and at least one hidden layer with a non-linear ("squashing") activation function (e.g. sigmoid) can approximate a large class of functions with arbitrary accuracy - provided that the network is given a sufficient number of hidden units and the parameters are suitably chosen.	
e.g.	combine 2 sigmoid to generate a step function -> with this we can approximate a large class of functions	
problem	with a shallow network, any function can be represented, but with problems: - for improving accuracy, more and more neurons are needed (exponentially growing number) - with more dimensions (e.g. image or audio) more sampled data is needed -> **curse of dimensionality** - if we just use the available data and interpolate with step-functions between data points -> we overfit	

Overfitting

Overfitting	**Underfitting** - High Bias: Strong bias in the way the data deviate from the linear model.	**Good Fit** - "Just Right": Model seems to capture just right the underlying structure in the data.	**Overfitting** - "High Variance": Model matches samples perfectly: too well given the number of samples. seems to capture also statistical fluctuations
example $h_\theta(x) =$	 $g(\theta_0+\theta_1x_1+\theta_2x_2)$	 $g\begin{pmatrix}\theta_0+\theta_1x_1+\theta_2x_2+\\ \theta_3x_1^2+\theta_4x_2^2+\theta_5x_1x_2\end{pmatrix}$	 $g\begin{pmatrix}\theta_0+\theta_1x_1+\theta_2x_1^2+\theta_3x_1^2x_2\\ \theta_4x_1^2x_2^2+\theta_5x_1^2x_2^3+\theta_6x_1^3x_2\\ +\cdots\end{pmatrix}$
Occam's Razor	"Among competing hypothesis, the simplest is the best." William of Ockham (1285-1347)		
Definition	Overfitting occurs when the learned hypothesis (trained model) fits the training data set very well - but fails to generalise to new examples.		
can occur when	- the training set is too noisy - its size is too small in comparison with the dimensionality of the input data - the number of parameters of the model is too large, i.e. the model is too "flexible"		
examine overfitting	 	Overfitting occurs when the learned hypothesis (trained model) fits the training data set very well - but fails to generalise to new examples.  	

4. Model Selection Process

Goal	Select from a family of models the model with the best performance.	
Problem	To achieve this goal, we need to evaluate the performance for different models. But to avoid overfitting on the test set we must not use information from the test set to tune params.	
Solution	Split the original training set into **training set** and **validation set**.	
Hyper-Parameter Tuning	Hyper-Parameters specify higher level properties of the mapping function (model) and/or the learning process. These parameters are optimised on the validation set. e.g. learning rate, batch size, number of hidden layers, number of neurons in a layer	
learning curves	With more training data it gets more difficult to perfectly fit a model the model trained with more data captures more details about the underlying problem.	
Cross-Validation	Split data set in different junks (folds, 5-10) with equal size. -> not that important, see ML courses	

Performance Measures

Confusion Matrix

Given a classification system, a confusion matrix evaluates the performance of such system through an $m \times m$ matrix, where m is the number of classes. It shows how many of class "a" were confused as class "b", hence its name "confusion matrix".
Very useful to understand the type of errors the system is doing.

```
from sklearn.metrics import confusion_matrix
confmat = confusion_matrix(y_actual, y_pred)
```

Confusion Matrix Example

		predicted class			
		a	b	c	d
actual class	a	120	21	7	8
	b	8	131	63	19
	c	12	30	80	11
	d	1	11	8	40

$$Overall\ Accuracy = \frac{\Sigma diagonal\ elements}{\#samples}$$

$$\frac{120 + 131 + 80 + 40}{570} = \frac{371}{570} = 65\%$$

$$Error\ rate = 1 - Accuracy$$

$$1 - 0.65 = 35\%$$

Confusion Table

A **Confusion Table** is used to measure the classification performance of a **two-class** system.

		Predicted		
		Positive	Negative	Total
Actual	Positive	True Positive	False Negative	(TP+FN)
	Negative	False Positive	True Negative	(FP+TN)
	Total	(TP+FP)	(FN+TN)	N

Confustion Table Example (digit 5)

		Predicted		
		P	N	Total
Actual	P	809	112	921
	N	128	8951	9079
	Total	937	9063	10'000

$$Class\ accuracy = \frac{TP + TN}{\#sample}$$

correct classification considering a given class against the others.

$$= \frac{809 + 8951}{10'000} = 97.6\%$$

$$Recall = class\ sensitivity = \frac{TP}{TP + FN}$$

correct classification for a given class

$$= \frac{809}{809 + 112} = 87.84\%$$

$$class\ precision = \frac{TP}{TP + FP}$$

correct classification in the predicted outputs for a given class

$$= \frac{809}{809 + 128} = 86.34\%$$

$$F - Score = \frac{2 * precision * recall}{precision + recall}$$

$$= \frac{2 * 0.8634 * 0.8784}{0.8634 + 0.8784} = 87.08\%$$

$$specificity = \frac{TN}{TN + FP}$$

$$\frac{8951}{8951 + 128} = 98.59\%$$

General Rule for Model Selection

Plot curves with these performance measures computed on the training and the validation set, to avoid overfitting, look at the performance on the validation set.

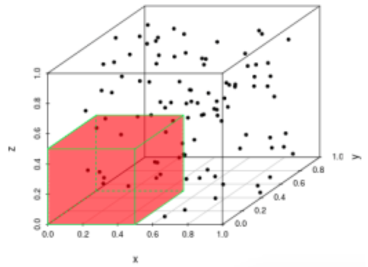
x: error rate

Dimensionality, Curse of Dimensionality

This is best understood by how many parameters we need for each point in the Figure to the right.

- In 1D we only have, say 100 points.
- In 2D we have $100^2 = 10'000$ points
- In 3D we have $100^3 = 1'000'000$ points
- Real world application: an image with 300 dpi of 2400x2400 in RGB (3 channels) = 17'280'000 params. And that's just ONE image from your training data.

In machine learning, the more features our data has, the higher dimensionality problem we'll have. A solution for this is called "dimensionality reduction", and different methods like Principal Component Analysis are used.

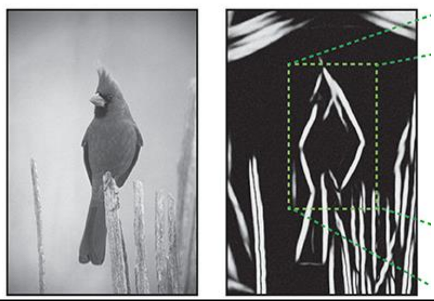


$$m \propto N^d$$

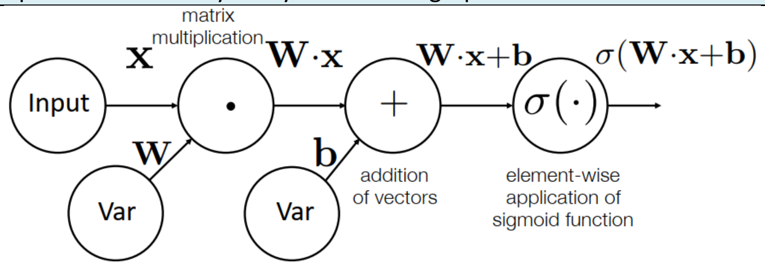
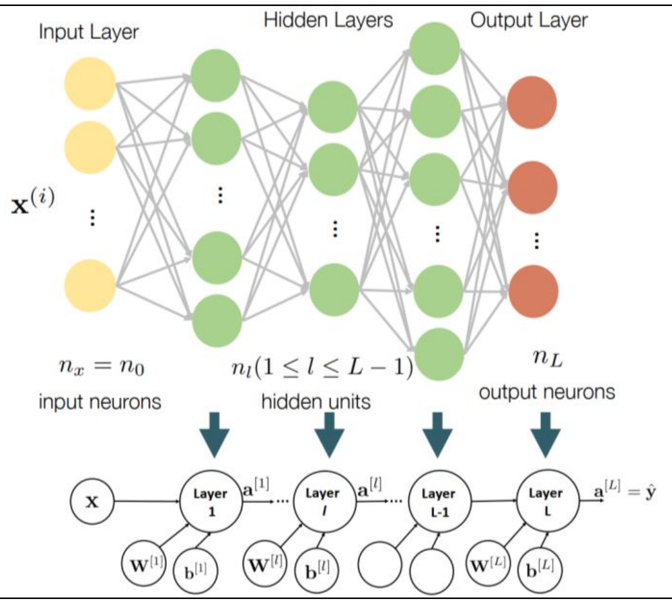
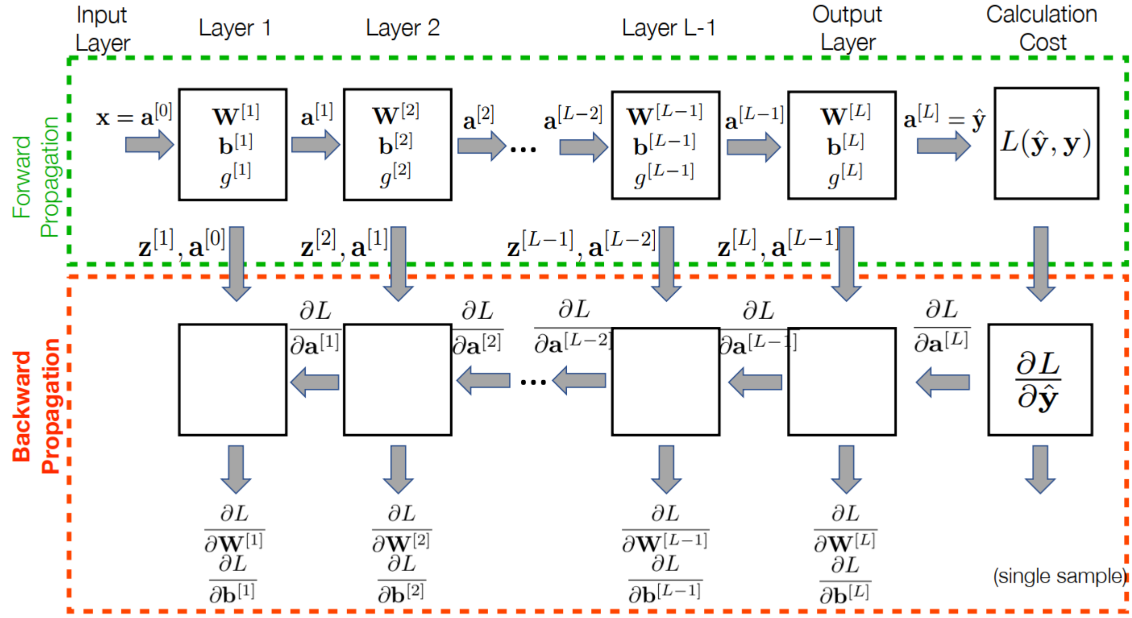
m: Number of samples needed
N: Number of points along each dimension
d: Number of parameters of the model

Feature Variation, Levels of Hierarchy

This brings us to the concept of feature variation and levels of hierarchy. In Feature variation, the idea and solution to the Curse of Dimensionality, is to use a machine learning method that does not really "learn" all the parameters, but instead is robust to variation and finds similar features across the data. images, as the best example, have similar features, and variations from rotations, different pose, moustaches and beards, etc. So, we would build a hierarchy of features as: edges, contours, objects.



5. Back Propagation

Computational Graphs	A computational graph is a directed graph where the nodes correspond to operations or variables. Variables can feed their value into operations, and operations can feed their output into other operations. This way every node in the graph defines a function of the variables.	
e.g.		Input: e.g. MNIST pictures Var w: weights Var b: biases Function: e.g. sigmoid
multi-layer one sample calculus per layer $\boldsymbol{x} = \boldsymbol{a}^{[0]}: n_x \times 1$		weights: $n_l \times n_{l-1}$ (1 row for 1 neuron) $\boldsymbol{W}^{[l]} = \begin{pmatrix} w_{11} & \dots & w_{1n_{l-1}} \\ \dots & & \dots \\ w_{n_l 1} & \dots & w_{n_l n_{l-1}} \end{pmatrix}$ bias: $n_l \times 1$ $\boldsymbol{b}^{[l]} = \begin{pmatrix} b_1^{[l]} \\ \dots \\ b_{n_l} \end{pmatrix}$ activation: $n_{l-1} \times 1$ $\boldsymbol{a}^{[l-1]} = \begin{pmatrix} a_1^{[l-1]} \\ \dots \\ a_{n_l}^{[l-1]} \end{pmatrix}$ activation: $n_l \times 1$ $\boldsymbol{a}^{[l]} = \begin{pmatrix} a_1^{[l]} \\ \dots \\ a_{n_l}^{[l]} \end{pmatrix}$ temporary variable: $n_l \times 1$ $\boldsymbol{z}^{[l]} = \boldsymbol{W}^{[l]}\boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]}$: element wise activation function: $n_l \times 1$ $\boldsymbol{a}^{[l]} = g^{[l]}(\boldsymbol{z}^{[l]})$
multi-layer multiple sample: m calculus per layer $\boldsymbol{X} = \boldsymbol{A}^{[0]}: n_x \times m$	handle multiple samples with one calculation (add 3rd dimension) $\overbrace{n_l \times n_x}^{W} * \overbrace{n_x \times m}^{A} * + \overbrace{n_l \times m}^{b} = \overbrace{n_l \times m}^{Z}$	$\boldsymbol{A}^{[l-1]}: n_{l-1} \times m$ $\boldsymbol{W}^{[l]}: n_l \times n_{l-1}$ $\boldsymbol{Z}^{[l]} = \boldsymbol{W}^{[l]}\boldsymbol{A}^{[l-1]} + \boldsymbol{b}^{[l]}: n_l \times m$ $\boldsymbol{A}^{[l]} = g^{[l]}(\boldsymbol{Z}^{[l]}): n_l \times m$ $\boldsymbol{b}^{[l]}: n_l \times m\ (broadcasting)$
Chain rule	Compute the amount of change of a function by changing one of its variables.	$L = L\left(g\left(h(x)\right)\right) \rightarrow dL = \frac{\delta L}{\delta g}\frac{\delta g}{\delta h}\frac{\delta h}{\delta x}dx$
Backward Propagation based on chain rule	Learning consists in **minimising the cost** with respect to the model parameters. Gradient descent is an iterative scheme to accomplish that. Update rule with learning rate α. We can propagate all gradients with respect to the activations back through all the layers. 	

6. Regularisation and Optimisation

Problem

When learning with backprop, we observe that:
- learning is slower in earlier layers (~length of gradients)
- learning becomes even slower when having more layers

But small gradients here don't imply that we are close to the minimum. Generally, the gradients in deep neural networks are unstable, tending to either vanish (prevalent = dominant) or explode in earlier layers.

Reason

- **Multiplicative Structure** -> consequence of the chain rule
- **Product Term**
- **Coordinating Updates** -> updates of the weights in different layers are highly coupled.

Solution

The problem cannot be completely solved -> but alleviated

- **Parameter initialisation** -> proper initialization of the weights so that the logit z does not grow too large in magnitude.
  - randomly initialise weights -> to break symmetries at start of learning
  - put initial weights at proper weights -> normalize z-values in different layer (Xavier)
- **Batch Normalisation, Gradient Clipping**: Make sure that the weights do not grow too large in magnitude also during training
  - Normalise the input to each layer per mini-batch by adding an operation in the model just before the activation function of each layer: zero-centre and scale the inputs by estimating mean and stdev from the current mini-batch.

Normalisation	
$\left(Z_{norm}^{\{r\},[l]}\right)_{k,i} = \dfrac{Z_{k,i}^{\{r\},[l]} - \mu_k^{\{r\},[l]}}{\sigma_k^{\{r\},[l]} + \epsilon}$	$\mu_k^{\{r\},[l]} = \dfrac{1}{N_B}\sum_{i=1}^{N_B} Z_{k,i}^{\{r\},[l]}$ $\sigma_k^{\{r\},[l]} = \dfrac{1}{N_B}\sum_{i=1}^{N_B}\left(Z_{k,i}^{\{r\},[l]} - \mu_k^{\{r\},[l]}\right)^2$

  - Then let the model learn the optimal scale and mean of the inputs for each layer.

Scaling and Shifting	
$\hat{Z}_{k,i}^{\{r\},[l]} = \gamma_k^{[l]}\left(Z_{norm}^{\{r\},[l]}\right)_{k,i} + \beta_k^{[l]}$	$\gamma_k^{[l]}$ *and* $\beta_k^{[l]}$ have to be learned

  -> can be applied to any input or hidden layer in a network.
  - Or clip the gradient in length during backprop so that they never exceed some threshold. Because neural networks or recurrent networks often have an extremely steep cliff structure.
- **Suitable Activation Functions**: Use activation functions that cannot saturate.
  - use ReLU, LeakyRELU or ELU
  - **ReLU** suffers the dying unit's problem: During training, if a neuron's weight gets updated such that the weighted sum of the neuron's inputs is negative, it is outputting 0. Since the gradient at $z < 0$ is 0, there is no weight update for this neuron and the neuron is likely to stay dead.
  - With the **leaky ReLU** (or its smooth version, the **ELU**) this problem cannot occur.

Regularisation

Problem: Deep neural networks typically have many parameters to fit a huge variety of complex datasets. But bear the risk to **overfitting** the training set.
"Regularisation is any modification to a learning algorithm that is intended to reduce its generalisation error but not its training error."

Weight Decay (Weights Penalties)

Add constraints to the parameters to give preference to simple models - restriction in the length of the parameter vector or in number of parameters (sparsity).

$J = J_{data} + J_{reg}$	J_{data}: Performance term: on the data, how far are we from the ground truth J_{reg}: Regularisation term: we should not let our model become too complex
$J = J_0 + \lambda\Omega(\mathbf{W})$	J_0: original loss function (e.g. RMS loss) $\Omega(\boldsymbol{W})$: penalty term that favours models with smaller weights λ: regularisation parameter

Two forms of penalties are common

	Penalty term	Gradient for the regularised loss function:
L_1	$\Omega(\boldsymbol{W}) = \lVert W \rVert_1 = \sum_{l,k,j} \left\lvert W_{kj}^{[l]} \right\rvert$	$\nabla(J_0(\boldsymbol{W}) + \lambda \lVert \boldsymbol{W} \rVert_1) = \nabla J_0(\boldsymbol{W}) + \lambda sign(\boldsymbol{W})$
L_2	$\Omega(\boldsymbol{W}) = \lVert W \rVert_2^2 = \sum_{l,k,j} \left\lvert W_{kj}^{[l]} \right\rvert^2$	$\nabla\left(J_0(\boldsymbol{W}) + \frac{\lambda}{2}\lVert \boldsymbol{W} \rVert_2^2\right) = \nabla J_0(\boldsymbol{W}) + \lambda \boldsymbol{W}$ update rule for gradient descent $\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla J(\boldsymbol{W})$

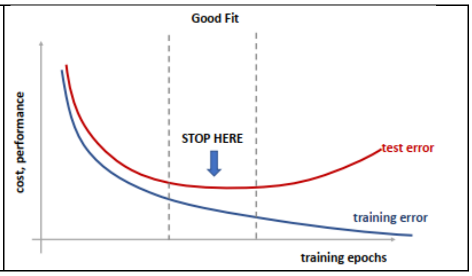
Dropout

Randomly drop neurons during training steps to make the solution less dependent on individual neurons
Popular, successful, imply simpler models, more robust, cheap computation at training, very versatile (=vielfältig), can be combined, need to be bit larger
At every training step, every neuron (input by 20%, hidden by 50%) has a probability p of being temporarily -> masked activations. Hyper-parameter p is called dropout rate.
Correct weights after training, because they have been tuned with dropout rate.

Early Stopping

Stop training at the minimum of the cost function on the validation set to avoid overfitting.
Often, the training error monotonically decreases while the validation error begins to increase after a certain number of epochs. A behaviour that can only be observed when training large models with sufficient representational capacity so that overfitting is possible.

- Run optimisation algorithm to train the model - simultaneously compute the validation set error.
- Store a copy of the model parameters as long as the validation set error improves.
- Iterate until validation set error stops improving (e.g. for k steps)
- Return the parameters where the smallest validation set error is observed.

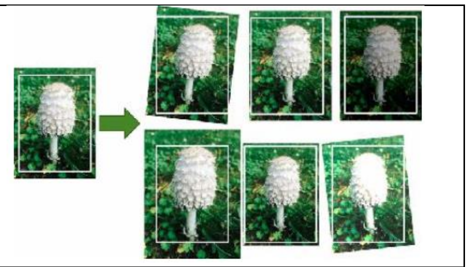


efficient, non-intrusive (aufdringlich), parallelizable, combinable

Data Augmentation

Generate more training data to introduce additional characteristic features (e.g. symmetries) the solution should have.

Reduce model complexity.
Increase the amount of training data (realistic).
Model is forced to be more tolerant to position, orientation, size, contrast, etc.
Typically used for classification tasks.
Be careful with flips and 180° rotation (d-b, 9-6)
May be considered rather as pre-processing step.
But can be applied on the fly. Reduce network bandwidth



Advanced Optimisation Methods

Gradient Descent
+ works for convex function -> but cost function in deep neural networks are typically non-convex.
- can get stuck in a local minimum or saddle point -> hard to find in high dimensional spaces.
- can get very slow -> faster optimisers desirable

Momentum, Nesterov

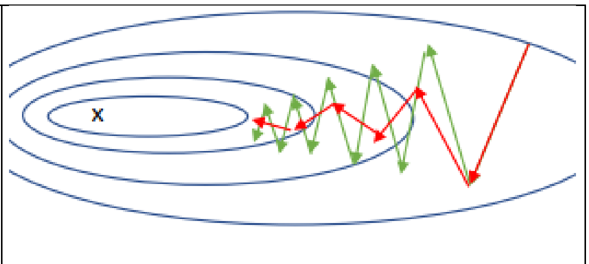
Allows to surpass flat regions or saddle points - like a ball that keeps on rolling down if it has an initial speed when entering flat regions.

Compute an exponentially decay moving average of past gradients and move in the direction of the moving average. 'Momentum' hyper parameter which controls the decay and the friction.

Momentum $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} + \alpha \nabla J(\theta)$
Nesterov $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} + \alpha \nabla J(\theta + \beta_1 \boldsymbol{m})$

$$\theta \leftarrow \theta - m$$

good beta: $\beta_1 = 0.9$



RMS Prop

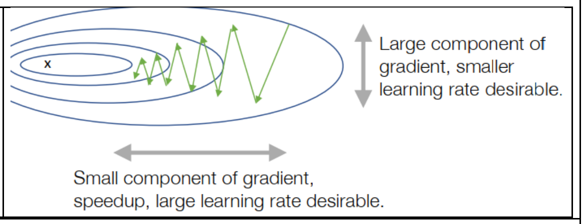
Adaptively adjusts the learning rate to incorporate differences in the steepness along different directions in parameter space.

Increase learning rate in direction of slow progress and decrease in direction of fast progress.

$$s \leftarrow \beta_2 s + (1 - \beta_2) \nabla J(\theta) \odot \nabla J(\theta)$$

$$\theta \leftarrow \theta - \frac{\alpha}{\sqrt{s + \epsilon}} \odot \nabla J(\theta)$$

$\odot$*elementwise operation*



Adam (state of the art)

Combination of Momentum / Nesterov and RMSprop

$$\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} + (1 + \beta_1) \nabla J(\theta)$$

$$s \leftarrow \beta_2 s + (1 - \beta_2) \nabla J(\theta) \odot \nabla J(\theta)$$

$$\widehat{\boldsymbol{m}} = \frac{\boldsymbol{m}}{1 - \beta_1} \quad (init\ with\ 0)$$

$$\hat{\boldsymbol{s}} = \frac{\boldsymbol{s}}{1 - \beta_2} \quad (init\ with\ 0)$$

$$\theta \leftarrow \theta - \alpha \frac{\widehat{\boldsymbol{m}}}{\sqrt{s + \epsilon}} \odot \nabla J(\theta)$$

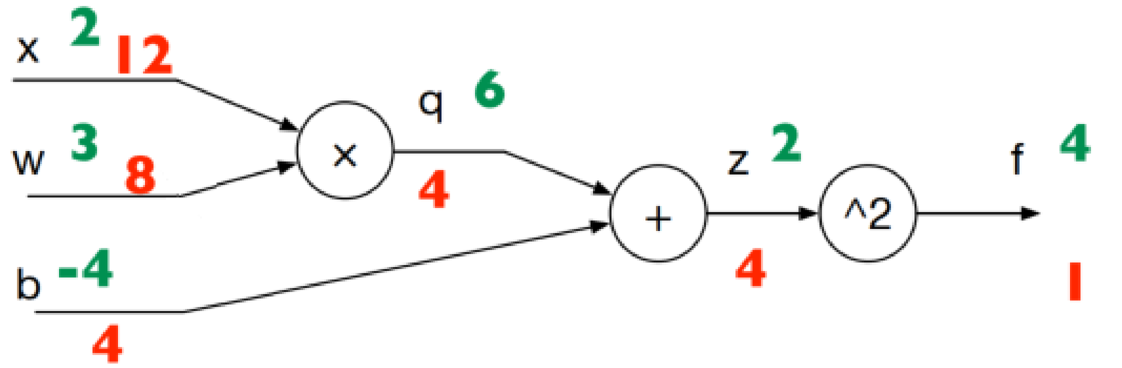
Learning Rate	α	0.001
Momentum	β_1	0.9
RMS Decay	β_2	0.999
Numerical Stabilisation	ϵ	1.0E-08
Learning rate	α	(reduce after epochs)

7. DL-frameworks

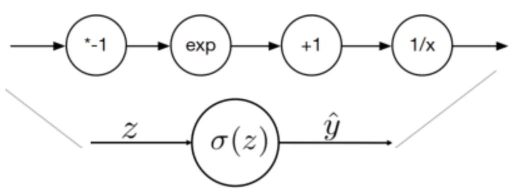
CPU vs GPU **TPU**	Central Processing Unit: Few cores (~10), fast (~4GHz), lots of cache, few parallel processes Graphical Processing Unit: Many cores (~1'000), slow (~1.5GHz), few caches, many parallel processes GPU is suitable for matrix multiplication -> because it's highly parallelizable. Tensor Processing Unit: even more specialised hardware to perform matrix and convolution operations
GPU	CUDA - for NVIDIA only - low level API for programming GPU OpenCL - like CUDA but runs on anything - usually slower on NVIDIA hardware HIP - Write code once in HIP C++ and port on NVIDIA or AMD

Forward
Backward
Node principles

add = distributor
max = router
mul = switcher



→	$x = 2$ $w = 3$ $b = -4$	$q = x * w = 2 * 3$ $q = 6$	$z = q + b = 6 + (-4)$ $z = 2$	$f = z^2 = 2^2$ $f = 4$
←	$\frac{\delta q}{\delta x}\frac{\delta z}{\delta q} = 3 * 4 = 12$ $\frac{\delta q}{\delta w}\frac{\delta z}{\delta q} = 2 * 4 = 8$	$\frac{\delta z}{\delta q}\frac{\delta f}{\delta z} = 1 * 4 = 4$ $\frac{\delta z}{\delta b}\frac{\delta f}{\delta z} = 1 * 4 = 4$	$\frac{\delta f}{\delta z}\frac{\delta f}{\delta f} = 2z * 1 = 4$	$\frac{\delta f}{\delta f} = 1$

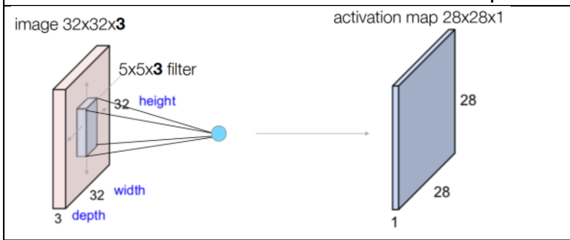
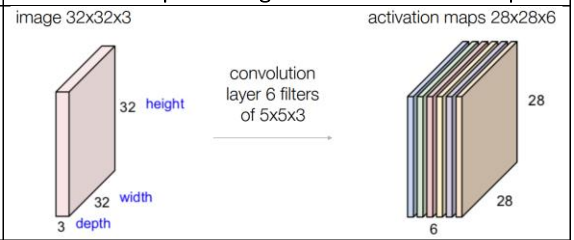
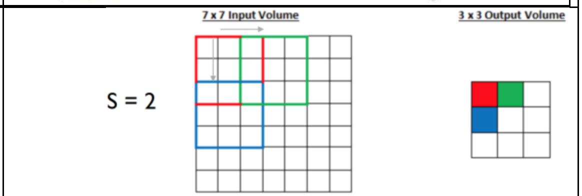
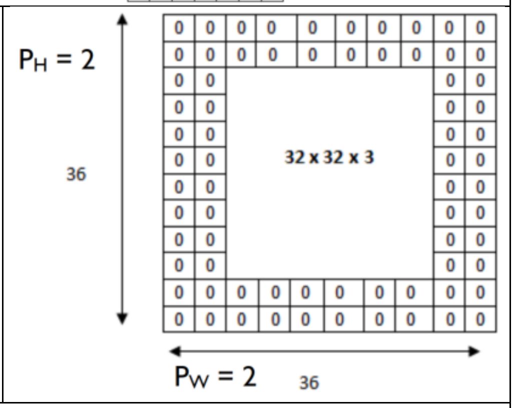
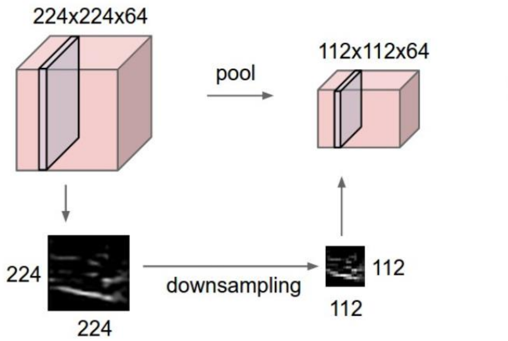
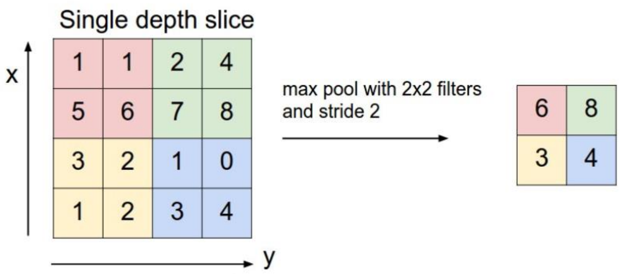
Node composition or factorisation	A sub-graph composed of nodes can be reimplemented in a single node if we can compute an analytic form of the gradients. Any complex learning architecture can be composed from atomic nodes. No need to compute complex global gradients. The loss function can be seen as extra nodes.	$\frac{\delta\sigma}{\delta z}$	

A zoo of frameworks

Academia	Companies
Theano - U Montreal	**TensorFlow** (based on Theano, static)- Google
Torch - IDIAP/NYU/...	**Pytorch** (based on Torch, dynamic)- Facebook
Caffe - UC Berkeley	**Caffe2** - Facebook
MXNet - CMU, MIT, U Wash, HK UST	**MXNet** - Amazon, Intel
	DeepLearning4J - Skymind
	CNTK - Microsoft
	PaddlePaddle - Baidu

Advantages	- Easily build big computational graphs - Easily compute losses and gradients in computation graphs for update rules - Have at hand all the state-of-art strategies for regularisations and optimisations - Switch easily from cpu to gpu when needed.
Problem	The loss is not going down -> assign calls are not executed -> add dummy graph node that depends on the update

8. Keras + CNN

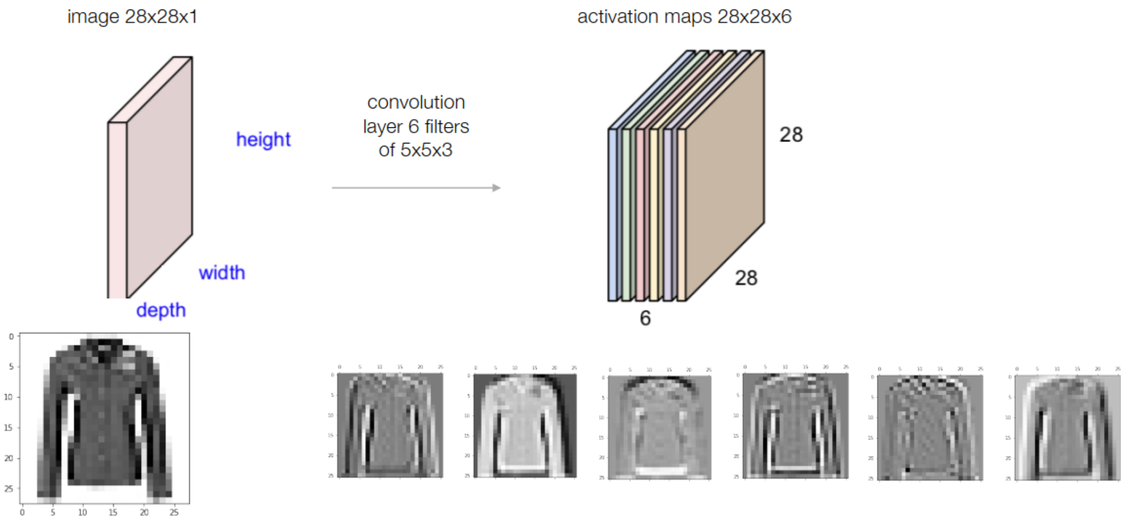
Keras	is a high-level open-source neural networks API written in Python and capable of running on top of TensorFlow, CNTK, Theano. It was developed with a focus on enabling fast experimentation. - minimalistic, straight forward, extensive, python, simple, good documentation, large community https://keras.io
pipeline	Define → Compile → Fit → Evaluate → Use
Models	A model in keras is the way to organise the layers of neurons: sequential or graph The **sequential** model corresponds to a regular stack of layers (1 layer = 1 object that feeds to the next) The **graph** model is used for non sequential architectures (diverge or merge networks)
CNN - Convolutional Neural Networks	General idea: let's define new type of layers and connections that will bring - preservation of spatial (=räumlich) structure - hierarchical feature detection - objects are composed of features that are themselves composed of other features - robustness to object variablilities such as viewpoint, occlusion (=Verdeckung), etc
Convolution layers feature extracting! convolve (=falten)	have the property to preserve the spatial structure and discover the local connectivity. A given filter is translated on **receptive fields** of the input and produces as output an **activation map**. convolution filter results in an activation map.  several filters producing several activation maps 
stride s	The stride specifies a step size when moving the filter across the signal. Larger stride means less overlap and reduction of the output volume. some s might be incompatible. 
padding p	is the size of a zeroed frame added around the input. $P_W = \frac{w-1}{2}$ $P_H = \frac{h-1}{2}$ $o_W = \frac{(W - w + 2P_W)}{s_W} + 1$ $o_H = \frac{(H - h + 2P_H)}{s_h} + 1$ $N_{param} = D\left(\underbrace{whC}_{weight} + \underbrace{1}_{bias}\right)$ P_W: padding in width P_H: padding in height h: height of filter w: width of filter H: height of image W: width of image o_W: output width o_H: output height C = # of channels D = # of filters N_{param}: # of params 
Max pooling layer	Reduce the spatial size of the representation. Applies independently to every depth. Defined by a stride S and a padding P. Most significant activations are kept, reduce the amount of computation and control overfitting.  
Dense layers	regular fully connected layers usually used as the last layers in the architecture to take classification decisions

9. Deep Architectures

CNN for FashionMNIST	60'000 train images, 10'000 test images, 28x28x1 grayscale, 10 classes Alternative to MNIST which is a bit more challenging, https://github.com/zalandoresearch/fashion-mnist Conv2D+Relu, MaxPooling2D, Dropout, Flatten, FC + ReLU, FC + SoftMax -> 91.04%
Going deeper	Means stacking CONV-RELU-POOL(+DROPOUT) layers and playing with the different hyper-parameters. The deeper the network - the bigger the number of filters in the conv layers - the smaller are the activation maps due to max pooling Early layers will extract lower-level features -> late layers will extract higher-level features.
Deep Learning	The whole process (pre-processing + feature extraction) is learned not only to classify/train

Visualisation

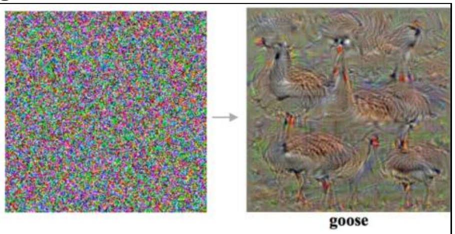
1st Strategy: **Visualize the activation maps** for a given conv layer to see feature extraction.



2nd Strategy: Find **input images that maximise the action of a given neuron**

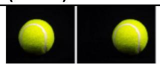
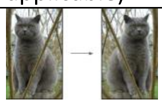
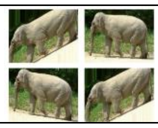
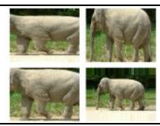
1. Start with a random image **x**
2. Forward: compute the activation of the neuron $a_{i,j}(\boldsymbol{x})$
3. Backward: perform the backprop of the gradient of $a_{i,j}(\boldsymbol{x})$ up to the pixel values: $\frac{\delta a_{i,j}(x)}{\delta x}$
4. This gradient tells us how to change the pixel values to increase the activation of the neuron. We can then apply an update rule in the form of a gradient ascent:

$$x \leftarrow x + a\frac{\delta a_{i,j}(\boldsymbol{x})}{\delta \boldsymbol{x}} + Regularisation\ Term$$

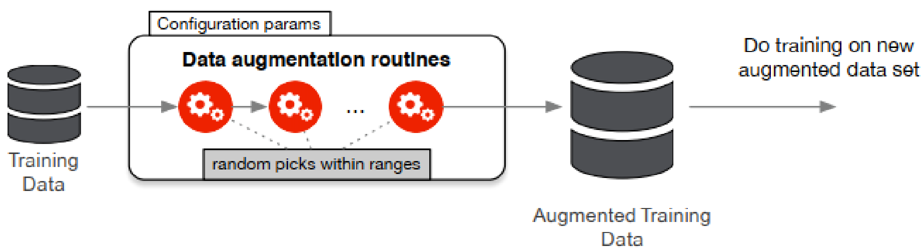


Data Augmentation

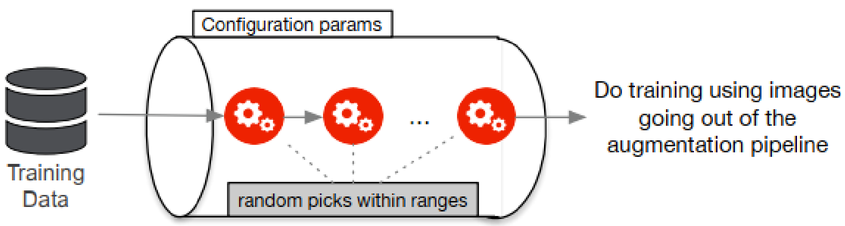
takes the approach of generating artificially more training data from existing training samples.

Rotation (angle)	Translation (shift)	Flip (not to all applicable)	Shear	Zoom	Colour
					???

Pre-Augmentation (offline): Do training on new augmented data set.



Online Augmentation: Do training using images going out of the augmentation pipeline.



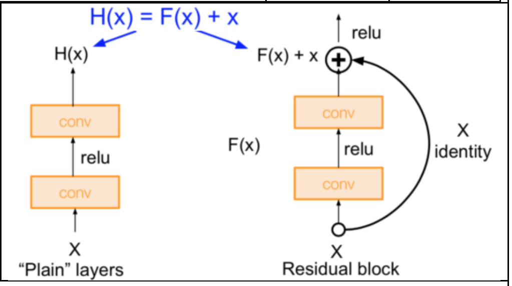
Overall improvement is not so high (because stability of MNIST data), but we avoid overfitting.

Convolutional layer patterns	Common form: `INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC` $0 \leq N \leq 3, \quad 0 \leq M, \quad 0 \leq K \leq 3$ ILSVRC: ImageNet Large Scale Visual Recognition -> Challenge
famous	(see table below)

			accuracy	**layers**
LeNet-5	1998	CONV-POOL-CONV-POOL-FC-FC		shallow
AlexNet	2012	the "boot" of deep architectures	16.4%	8
ZFNet	2013	AlexNet with Hyper parameter tuning	11.7%	8
VGGNet	2014	going deeper with simpler filter	7.3%	19
GoogLeNet	2014	Network in the Network (Inception), no FC, only 5M param	6.7%	22
ResNet	2015	way deeper. more difficult to optimize -> "fall back" layers to go deeper than 50 -> use bottleneck layers	3.57%	152
Shao et al	2016		3%	152
SENet	2017		2.3%	152

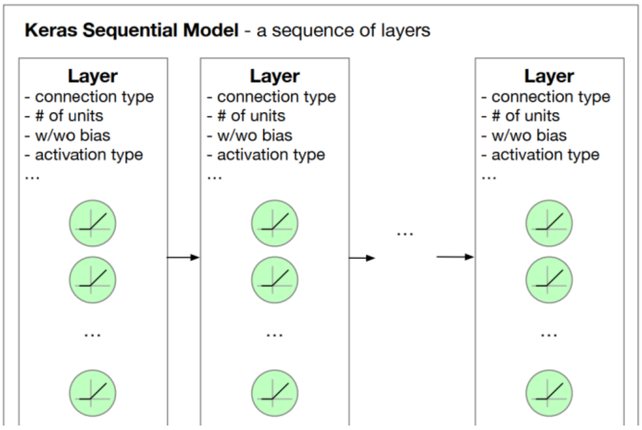
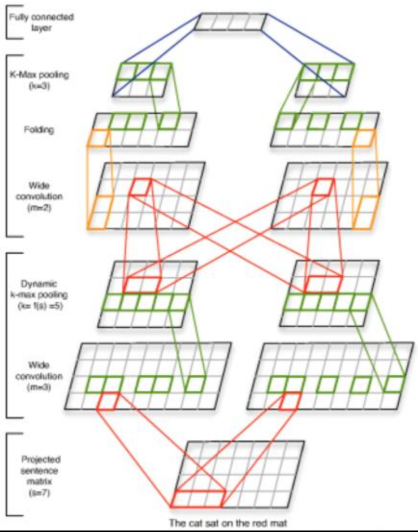
fall back layers

Hypothesis: Deeper models are more difficult to optimize
Ideas: Deeper layers should be able to "fall back" to identity mapping if difficulties to converge.
Easier to model a "delta" from one layer to the other than a full feature.
Solution: Use network layers to fit a residual(=Rest) mapping instead of directly trying to fit a desired underlying mapping.



bottleneck layers	Use bottleneck layers 1x1x64 to reduce the number of operations and parameters

10. CNN3 Keras Functional API, Transfer Learning, Autoencoders

Keras Model types	The **sequential** model corresponds to a regular stack of layers. 	The **functional** API is used for non-sequential architectures (multiple paths/inputs/outputs). 

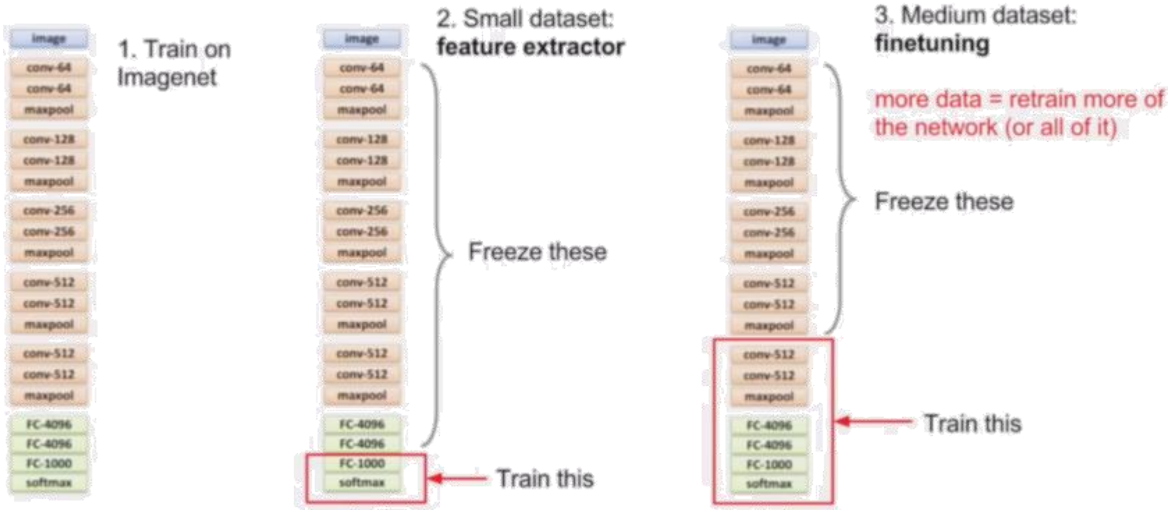
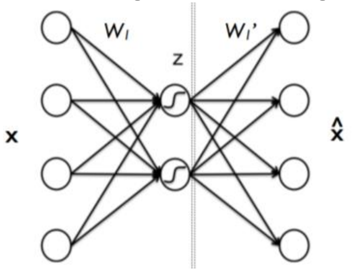
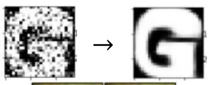
python callable syntax	`add1 = Adder(1) // calls __init__ and __call__`
	`add1(2) // calls __call__`
		`def __init__(self,x=0):`
		`    self.__memory = x`
		`def __call__(self,x):`
		`    self.__memory += x`
		`    return self`

Functional API	`visible = Input(shape=(28,28,1))`
	`conv1 = Conv2D(32, kernel_size=3, activation='relu')(visible)`
	`pool1 = MaxPooling2D(pool_size=(2,2))(conv1)`	

split (multiple path)	`conv1 = Conv2D(32, kernel_size=3, activation='relu')(visible)` `conv2 = Conv2D(32, kernel_size=5, activation='relu')(visible)`	
merge	`merge = concatenate([conv1, conv2])` e.g. multiple images of one object, stereo speech recording, different parameters of acquisition device	

callbacks	Callback are functions to be applied at given stages of the training procedure.
	To get a view on internal states and statistics. Declare them in fit() function.
	`checkpoint = ModelCheckPoint('model-(epoch:03d}.h5', verbose=1, monitor='val_acc', save_best_only=True, mode='auto')`
	`log = model5.fit(...., calbacks=[checkpoint])`	

Best practices	Use **Consistent Variable Names**. Use same variable name for input (visible) and output layers (output), hidden layers (hidden1, hidden2). It will help to connect things together correctly. **Review Layer Summary**. Always print the model summary and review the layer outputs to ensure that the model was connected as you expected. **Review Graph Plots**. Create a plot of the model graph and review it to ensure that everything was put together as you intended. **Name the layers**. You can assign names to layers that are used when reviewing summaries and plots of the model graph. For example: `Dense(1, name='hidden1')`. **Use Callbacks**. You should use callbacks to save the (best) models from epochs to epochs as a safety against interrupt and overfitting	

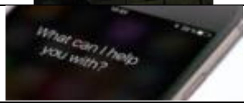
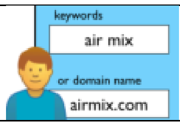
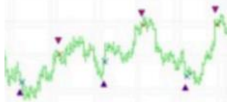
Transfer learning	is about using knowledge learned from tasks for which a lot of labelled data is available in settings where only little labelled data is available. -> Re-use the feature extraction part. Not to different.	
workflow		
Best practice	Freeze reused components and train new layers (classification part). Many pre-trained models for image recognition available in Keras.	
Code	`from keras.applications.mobilenet_v2 import MobileNetV2` `model = MobilNetV2(weights='imagenet', include_top=False, input_shape=(...))`	
bottleneck layers	To reduce number of operations, 1x1 conv-layer are used.	
Bayes Law	$P(C_k\|\boldsymbol{x}) = \frac{p(\boldsymbol{x}\|C_k)P(C_k)}{p(\boldsymbol{x})}$	Elect as winning category the one having the largest a posteriori probability. Doing so we guaranteed to maximise the accuracy. $P(C_k\|x)$: a posteriori probability, probability of class j given observation x $p(x\|C_k)$: likelihood, probability of observing x given class j $P(C_k)$: a priori probability, probability of class j $p(x)$: evidence = probabilty of x, ...unconditional to any class...
Auto-encoders	An autoencoder is a neural network trained to reproduce its input and able to discover "structures" and "efficient coding's" of the input space. The simples form is a feedforward, non-recurrent neural network.	
"diabolo" network	Encoding Decoding  $\boldsymbol{z} = E_{\theta_E}(\boldsymbol{x})$ $\hat{\boldsymbol{x}} = D_{\theta_D}(\boldsymbol{z})$ $\hat{\boldsymbol{x}} = h_\theta(\boldsymbol{x})$	The mapping function $h_\theta(\boldsymbol{x})$ is trained to reconstruct its own inputs, instead of predicting a target value. **Usage**: Data compression / dimensionality reduction Use encoder to obtain features De-noising images  Image in-painting  Initialise deep network for supervised learning **Loss function** $J(\theta) = J(\theta_E, \theta_D) = \frac{1}{2N}\sum_{n=1}^{N}(h_\theta(\boldsymbol{x}_n) - \boldsymbol{x}_n)^2 = \frac{1}{2N}\sum_{n=1}^{N}(\hat{\boldsymbol{x}}_n - \boldsymbol{x}_n)^2$

11.+12. Recurrent Neural Networks (RNN)

RNN

RNNs learn to memorise context information that may be important to draw conclusions on observations made later on.
RNN helps wherever we need context from the previous input.

Applications

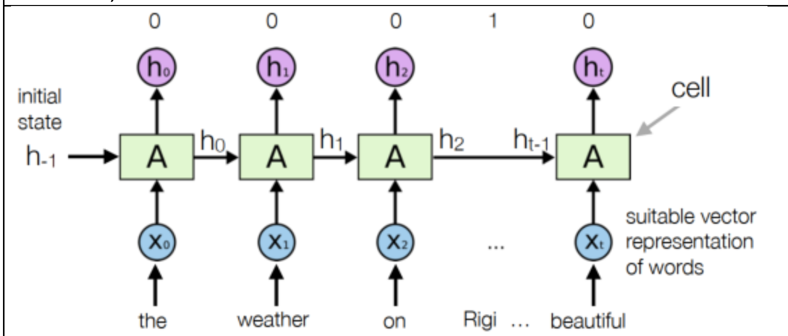
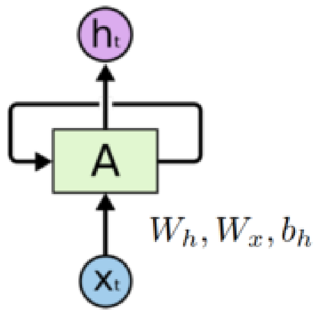
	Input sequence $x = (x_1..x_{T_x})$		Output sequence $y = (y_1..y_{T_y})$
Speech recognition Assistants (Siri, Alexa, ...) Video captioning Transcription to text		→	Today, the weather on Rigi above the fog is beautiful.
Sentiment classification Classification of text Emojification	Today, the weather on Rigi above the fog is beautiful.	→	
Machine translation DeepL, Google Translate	Today, the weather on Rigi above the fog is beautiful.	→	Heute ist das Wetter auf der Rigi oberhalb des Nebels schön.
Captioning, Subtitling images, YouTube videos		→	Sun shining above the clouds.
Chatbots, Q/A Siri, Alexa		→	task, answer
Named entity recognition (NER) flag names in sentences	Today, the weather above the fog on Rigi is beautiful. 0 0 0 0 0 0 0 1 0 0		
Music generation Magenta DeepJazz		→	
Word generation		→	
time sequence modelling, prediction			

Approaches

One-to-many	Many-to-one	many-to-many	many-to-many
image tagging, music generation	sentiment analysis	named entity recognition	language translation, speech recognition, chatbots
X -> YYY	XXX -> Y	XXX -> YYY	XX -> YYYY
$T_x = 1$ $T_y \geq 1$	$T_x \geq 1$ $T_y = 1$	$T_x \geq 1, T_y \geq 1$ $\boldsymbol{T_x = T_y}$	$T_x \geq 1, T_y \geq 1$ $\boldsymbol{T_x \neq T_y}$

Simple RNNs

e.g. $T_x = T_y$
like for NER

un-rolled, un-folded	rolled, folded
	

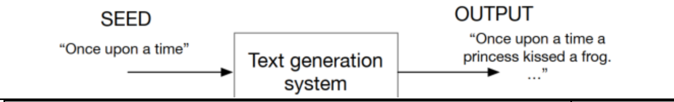
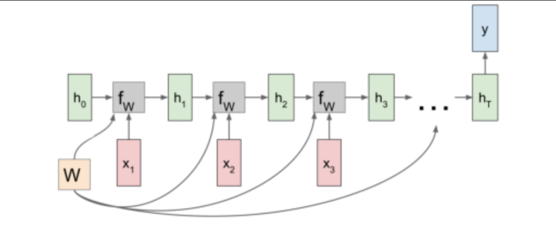
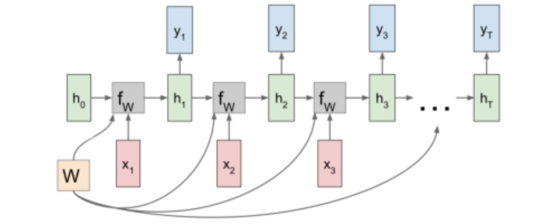
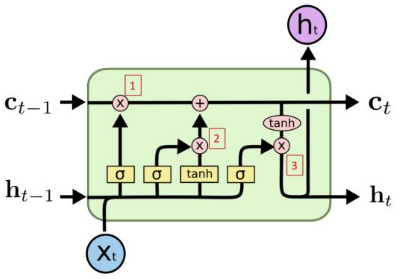
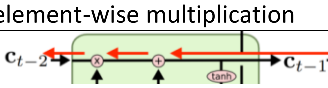
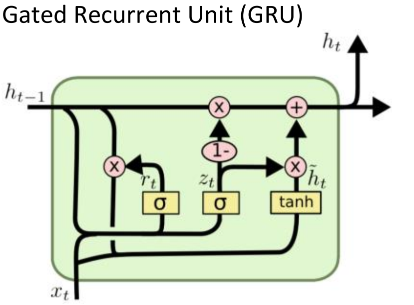
State h is update through a succession of steps. The state **memorises** (part of) the "history". Init with 0.

$h_t = g(\boldsymbol{W_x} * x_t + \boldsymbol{W_h} * h_{t-1} + b_h)$ →W and b are the same for all time step

code

```
from keras.layers import SimpleRNN
model = Sequential()
model.add(SimpleRNN(units=..., input_shape=..., ...)
```

Example Predict gender / home country from first name

Generative RNNs	A generative system is a system able to generate new consistent data from a seed. By consistent, we mean respecting temporal or spatial "structure" that have been learned from the input space.
example	 domain name suggestion Shakespeare sonnets (poetic verse form) C code generation
Approaches	**Many to one**  loss is computed from 1 output **Many to many**  loss is computed from all the outputs The difference is mainly at training time where the RNN cells are emitting an output at each step.
back propagation	run forward and backward through **chunks** of the sequence instead of whole sequence.
Long Term Dependencies Problem	Simple RNN's fail if a wider context is needed and the gap between the words grows too large. Long-range dependencies are hard to learn due to vanishing and exploding gradients problem.
Suitable Initialisation of Weights	**MLPs/ CNNs**: - Properly initialise weights (Xavier/He) by using random numbers (uniform or normal) with mean=0 and suitably scaled stdev. - Batch normalisation **RNNs:** with 'vanilla' recurrent units - IRNN: identity matrix - 67.0 % accuracy - highly sensitive to parameters - np-RNN: normalized-positive definite matrix - 75.2% accuracy - low sensitive to parameters both: - Use non-saturating activation functions (ReLU or LeakyReLU) to alleviate (=mindern) the vanishing gradients problem - Clipping gradients
Long-Short-Term Memory (LSTM) Cells or Gated Recurrent Units (GRU)	- LSTM 78.5% accuracy - low sensitive to parameters LSTM network have the same general structure of cyclically updated cells. It is designed to keep a **long-term memory** that is kept as additional state variable $\boldsymbol{c_t}$ to be updated. LSTM Cell  long-term: $c_t = f_t * c_{t-1} + I - T * \tilde{c}_t$ short-term: $h_t = o_t * \tanh(c_t)$ Long-term memory is updated through **Gates** (marked with ⊗). Gates control how the information flows between short-term state, input and long-term state 1: Forget Gate: $f_t = \sigma(\boldsymbol{W}_{f,x} * x_t + \boldsymbol{W}_{f,h} * h_{t-1} + \boldsymbol{b}_f)$ 2: Input Gate: $i_t = \sigma(\boldsymbol{W}_{i,x} * x_t + \boldsymbol{W}_{i,h} * h_{t-1} + \boldsymbol{b}_i)$ $\tilde{c}_t = \tanh(\boldsymbol{W}_{c,x} * x_t + \boldsymbol{W}_{c,h} * h_{t-1} + \boldsymbol{b}_c)$ 3: Output Gate: $o_t = \sigma(\boldsymbol{W}_{o,x} * x_t + \boldsymbol{W}_{o,h} * h_{t-1} + \boldsymbol{b}_o)$ The gates are implemented by - affine transformation of inputs - activation function - element-wise multiplication backprop: "Super-Highway for Backprop":  Gated Recurrent Unit (GRU)  most popular variation of LSTM. No separate long-term memory forget gate and input gate merged, new relevance gate Relevance Gate: $r_t = \sigma(W_{rh} * h_{t-1} + W_{rx} * x_t + b_r)$ Candidate State: $\tilde{h}_t = \tanh(W_{ch} * r_t * h_{t-1}) + W_{cx} * x_t + b_c)$ Update Gate: $u_t = \sigma(W_{uh} * h_{t-1} + W_{ux} * x_t + b_u)$ Update: $h_t = (1 - u_t) * h_{t-1} + u_t * \tilde{h}_t$ less parameters, since only 3 weight matrices $(n_h \times (n_h + n_x))$
Word Embedding	Word embedding is a projection of a word into vectors of real numbers, i.e. a mapping of a space where each word has its own dimension to a space that is of lower dimensionality. **word2vec** relates to models producing word embedding into space with contextual similarity or words, i.e. words that share a common context are near. Bag of Words eat-apple, driving-car Continuous Bag of Word: Eat an ... every day (apple) -> see google search