ntic master
revolucionamos la comunicación

UNIVERSIDAD COMPLUTENSE DE MADRID

# Deep Learning

Álvaro Romo

# About Me

+8 years of experience in research, consulting and projects in sectors such as telecommunications, marketing and insurance.

## SOME FRAMEWORK PROFICIENCIES

## SOME AREAS OF EXPERTISE

- ★ NLP
- ★ Machine Learning
- ★ Deep Learning
- ★ MLOps
- ★ Computer Vision

# Analytical Index

1. Introduction to Deep Learning
    a. Context
    b. Fundamentals
    c. Regularization techniques
    d. Exercises

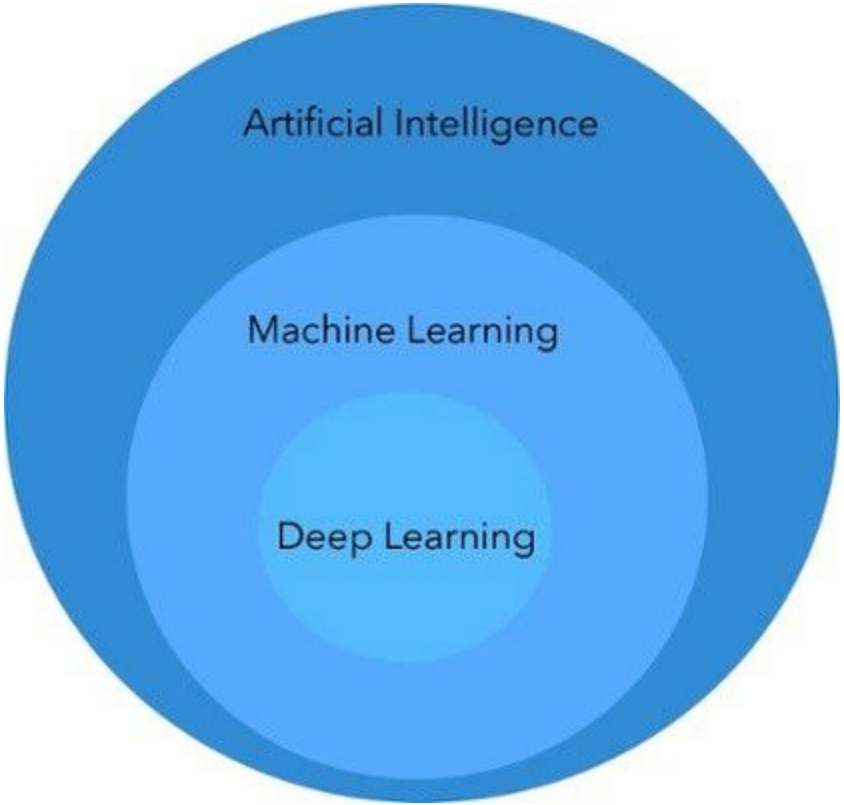
# 1

# Introduction To Deep Learning

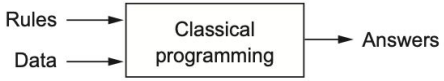
# 1.1

## Context

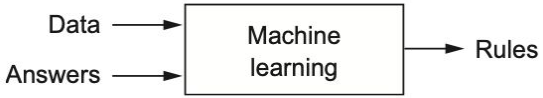
# AI vs ML vs DL




- **Artificial Intelligence:** Automate intellectual tasks normally performed by humans.
  - Symbolic: Set of explicit rules




- **Machine Learning:** Algorithms that improve automatically through experience.
  - Trained rather than explicitly programmed



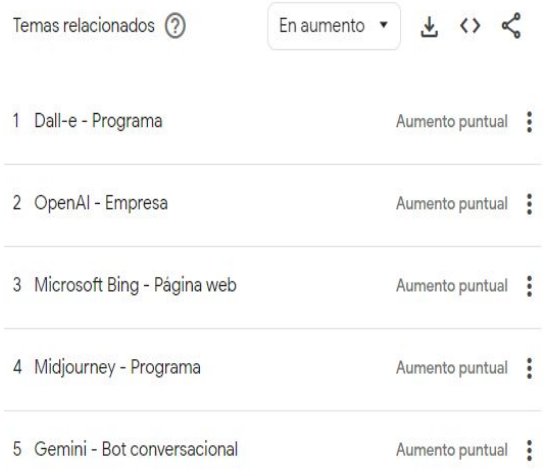

- **Deep Learning:** Artificial Neural Networks

# Why Deep Learning?

| | |
|---|---|
| NLP | Sentiment analysis, speech recognition, chatbots, machine translation, question answering, summarization, text generation… |
| Computer Vision | Object detection, face recognition, image segmentation, image captioning, optical character recognition (OCR), emotion recognition, gesture recognition… |
| Medicine | Disease identification/diagnosis, medical image analysis, genomic tasks, drug discovery, personalized medicine, telemedicine… |
| Playing Games | Superhuman Chess, Go, Atari games… |
| Robotics | Self-driving cars, autonomous drones… |
| Smart Cities | Traffic management, urban planning… |
| Finance | Fraud detection, credit risk assessment, algorithmic trading… |

# Increasing popularity of AI

- In recent years there has been an increase of more than 600% in the number of AI-related publications in arXiv. It is almost 5% of all global publications.
- In sectors such as technology and telecommunications, 50% of employees use generative AI on a regular basis by 2023.

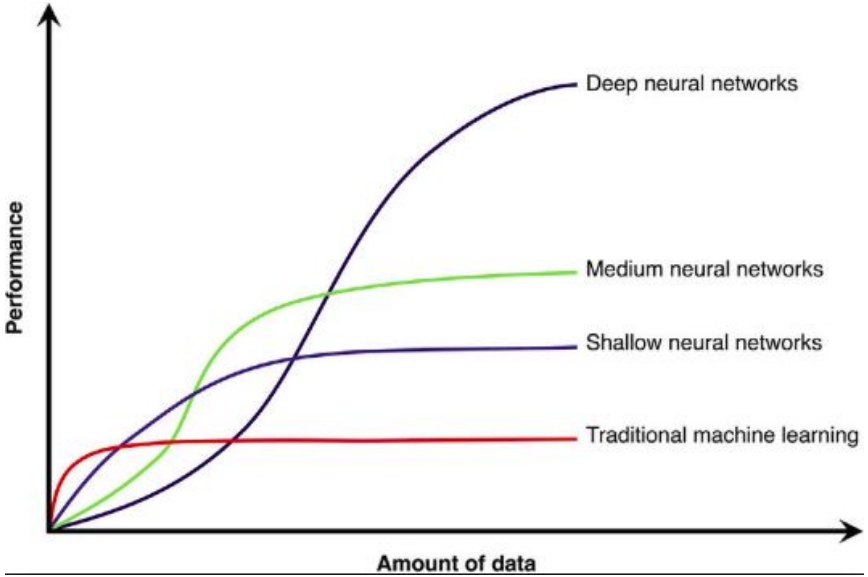




AI generative | Mckinsey report

Paper publications Standford

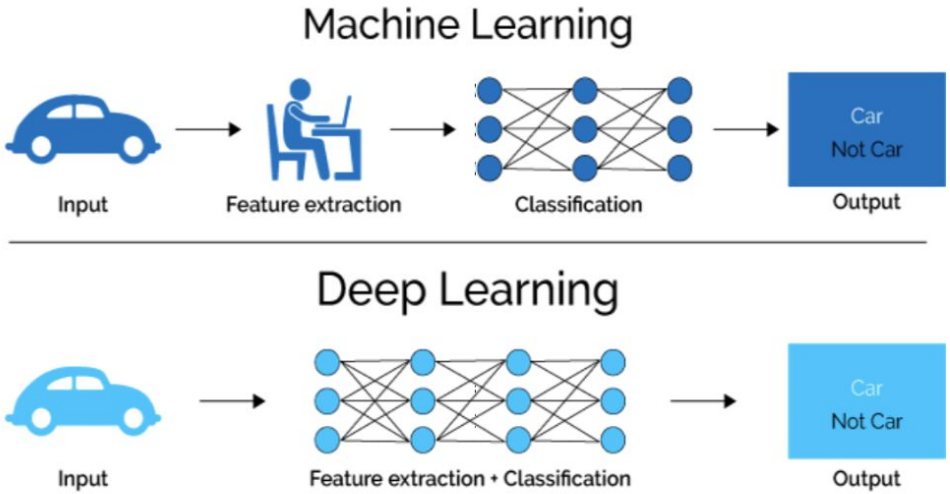
Google trends query

# Why Deep Learning?

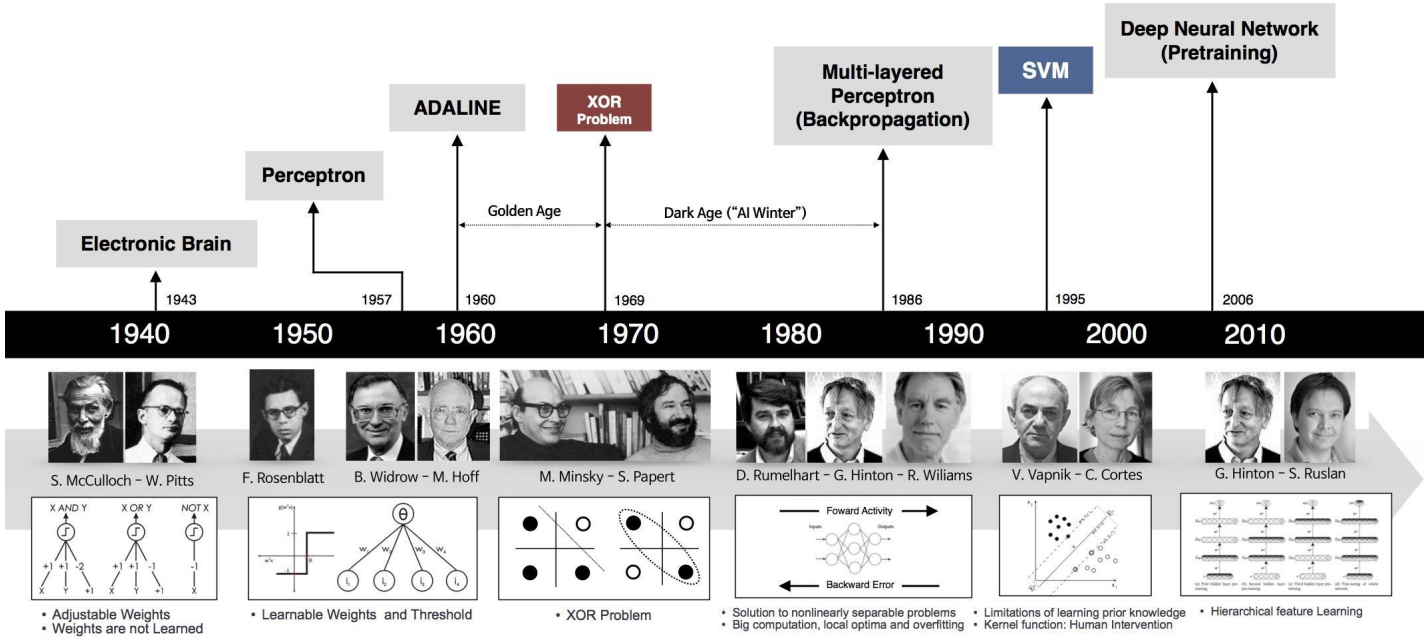



- Data:
  - Structured: tables
  - Unstructured: Audio, image, text
- Performance increases a lot with depth and with the amount of data. Specially with unstructured data.
- With small datasets, traditional machine learning usually performs better.
  - Transfer learning have solved this problem

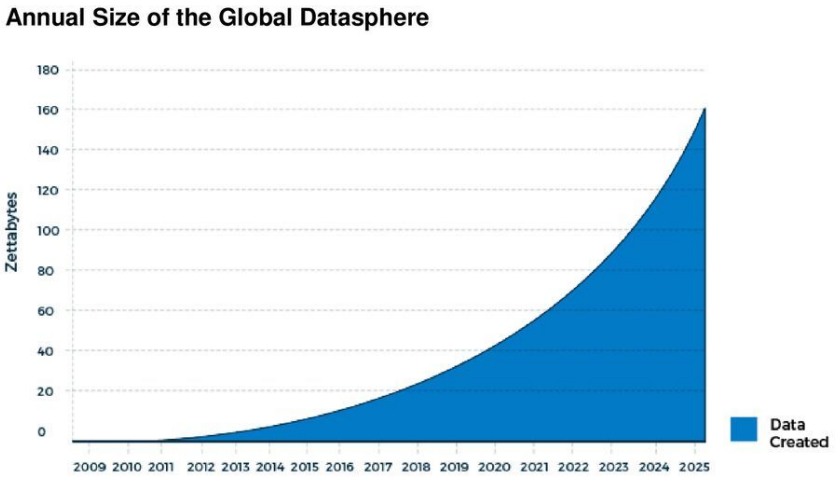
# Why Deep Learning?



# Why Now?


Deep Neural Network (Pretraining)
SVM
Multi-layered Perceptron (Backpropagation)
XOR Problem
ADALINE
Perceptron
Electronic Brain
Golden Age
Dark Age ("AI Winter")
1943
1957
1960
1969
1986
1995
2006
1940
1950
1960
1970
1980
1990
2000
2010
S. McCulloch - W. Pitts
F. Rosenblatt
B. Widrow - M. Hoff
M. Minsky - S. Papert
D. Rumelhart - G. Hinton - R. Wiliams
V. Vapnik - C. Cortes
G. Hinton - S. Ruslan
X AND Y
X OR Y
NOT X
Foward Activity
Backward Error
• Adjustable Weights
• Weights are not Learned
• Learnable Weights and Threshold
• XOR Problem
• Solution to nonlinearly separable problems
• Big computation, local optima and overfitting
• Limitations of learning prior knowledge
• Kernel function: Human Intervention
• Hierarchical feature Learning

# Why Now? Data



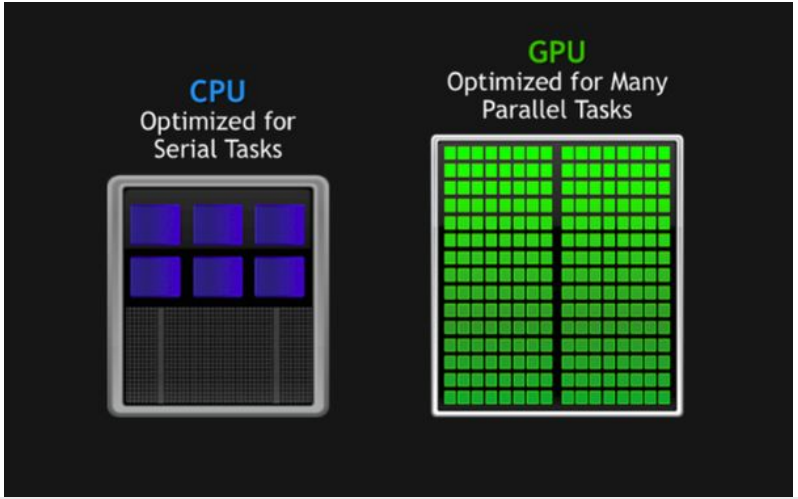

- Raw material that powers AI.
- The new electricity: Just as electricity revolutionized industries, deep learning is transforming businesses, research, and technology.
- ZettaByte Era:
  - Connected devices per person : 0.08 in 2000, 3.47 in 2015, 6.58 in 2020.
  - Before computers, humanity accumulated **12 exabytes** (10^18) of data (500 000 years DVD video). In 2002, **5 exabytes** of data.
- Web mining(content, use, structure), IoT, 5G, smartPhones, Wikipedia …
- 4th revolution

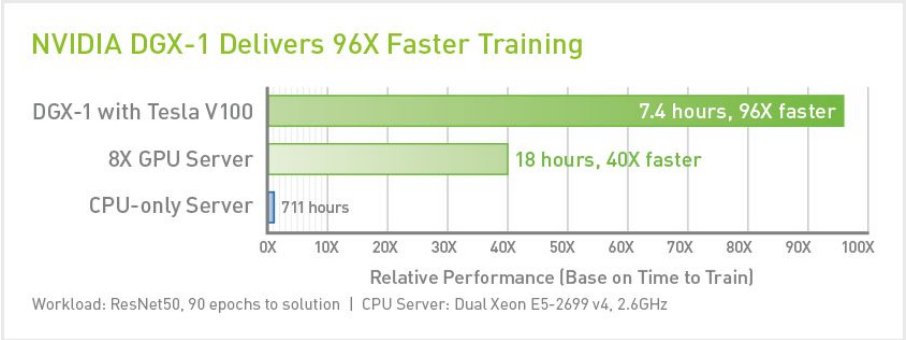
# Why Now? Hardware

- Between 1990-2010 CPUS 5000x faster
- Video games industry: GPU (massive parallel chips)
- Google TPU (Tensor Processing Unit)

- Hardware Accessibility: Cloud computing platforms (AWS, GCP, Azure) provide access to state-of-the-art hardware.

- NVIDIA GeForce RTX 4090:
  - 1800 $
  - 16384 cores
  - 80 TFLOPS, 450 W
- 2000: IBM ASCI White
  - Best supercomputer
  - 106 tons, 6MW (2 wind turbine)
  - 12 TFLOPS



# Why Now? Software, DL Frameworks:

We cannot build a neural net from the scratch every time we need one, right? ;)

The most important frameworks need to be/have:

- Optimized for performance.
- Easy to code
- Need to have a good community
- Need to automatically compute gradients

R.I.P.

theano

Caffe

Contenders

mxnet

DEEPLEARNING4J

Rockstars

PYTORCH

TensorFlow

Keras

# Pytorch

- Open-source, Python-based machine and deep learning framework.
- Developed by Facebook's AI Research lab (FAIR) with the first release in 2016.
- Easy to use API.
- **Dynamic Computation Graphs**: the graph is built along and can be manipulated at run-time.
- The framework is more “Pythonic”
- Extended in academia
- Rich ecosystem: Offers numerous tools and libraries, such as torchvision for computer vision, torchaudio for audio processing, and torchtext for NLP.





# Tensorflow basics


- Developed by the Google Brain Team and released in 2015 as an open-source library.
- Multi-platform support: Runs on CPUs, GPUs, and TPUs
- Since Tensorflow 2.0 (2019): dynamic graph definitions similar to PyTorch.
- Device compatibility: Supports Android and other devices through TensorFlow Lite, facilitating deployment on lightweight platforms such as mobile or IoT devices.
- Comprehensive ecosystem: Includes a wide range of tools and libraries, such as TensorFlow.js for web applications, TensorFlow Extended (TFX) for end-to-end ML pipelines, and TensorFlow Hub for pre-trained models.

# Keras


- Keras, a high-level neural network API, is officially integrated into TensorFlow 2.0 as the default API.
- Simple architecture. It is more readable and concise
- Keras was developed and maintained by François Chollet, a Google engineer:
  - Modularity
  - Minimalism
  - Extensibility:
  - Python: Everything is native Python.

# Hugging Face

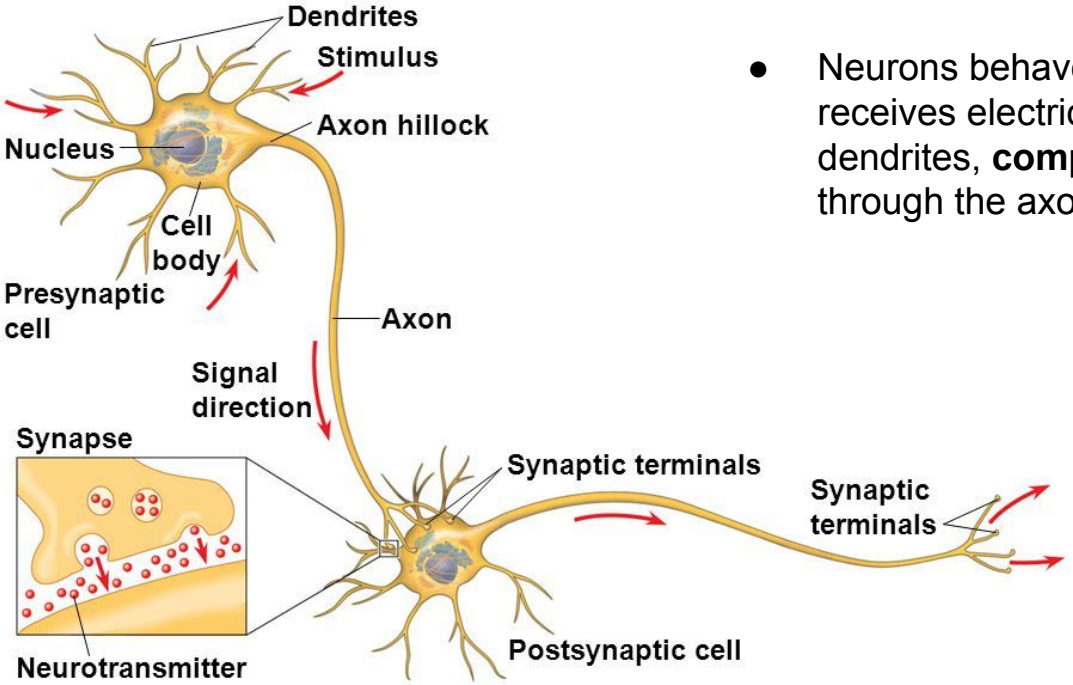
- There are a number of frameworks that have been developed to avoid having to develop neural networks from scratch.
- Optimized and developed to be used with hardware accelerators such as GPU's.




# 1.2

# Deep Learning Fundamentals

# Bio-inspired Model

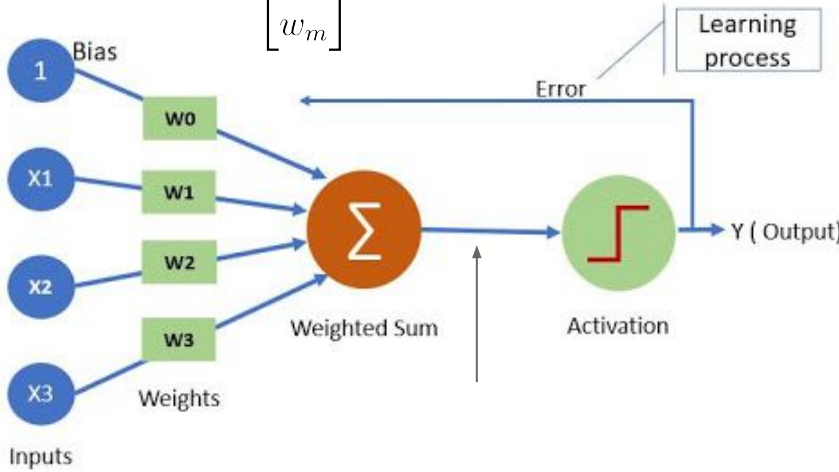



- Neurons behave as computing units. Each neuron receives electric input signals (called *spikes*) through the dendrites, **computes** an output with them and sends it through the axon to other neurons connected to it

- Axon-dendrite transmission is done in a mechanism called synapse.
- This process never changes, yet the brain is always learning

# Perceptron: Forward Propagation

$$\mathbf{w} = \begin{bmatrix} w_0 \\ \vdots \\ w_m \end{bmatrix}$$




$$y = g(w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3) = g(\mathbf{w}^T\mathbf{x})$$

$$\mathbf{x} = \begin{bmatrix} x_0 \\ \vdots \\ x_m \end{bmatrix}$$

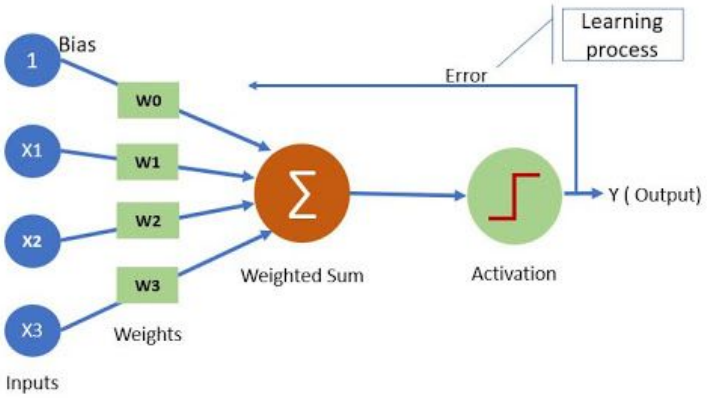
**Single Layer Perceptron**

Output — Linear combination of inputs

$$\hat{y} = g\left(\sum_{i=1}^{m} x_i \; w_i\right)$$

Non-linear activation function

# Perceptron: Logistic Regression




Single Layer Perceptron

$$\hat{y} = \sigma(w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3) = \sigma(\mathbf{w}^T \mathbf{x})$$

$$\hat{y} \leftarrow P(y = 1 | \mathbf{x})$$

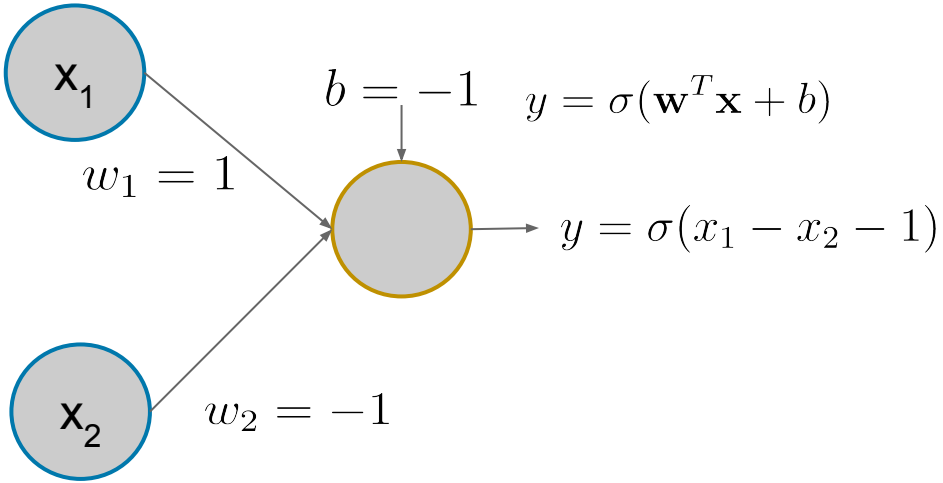
- Activation function:
  - sigmoid
- Loss function:
  - binary cross-entropy
- Gradients:

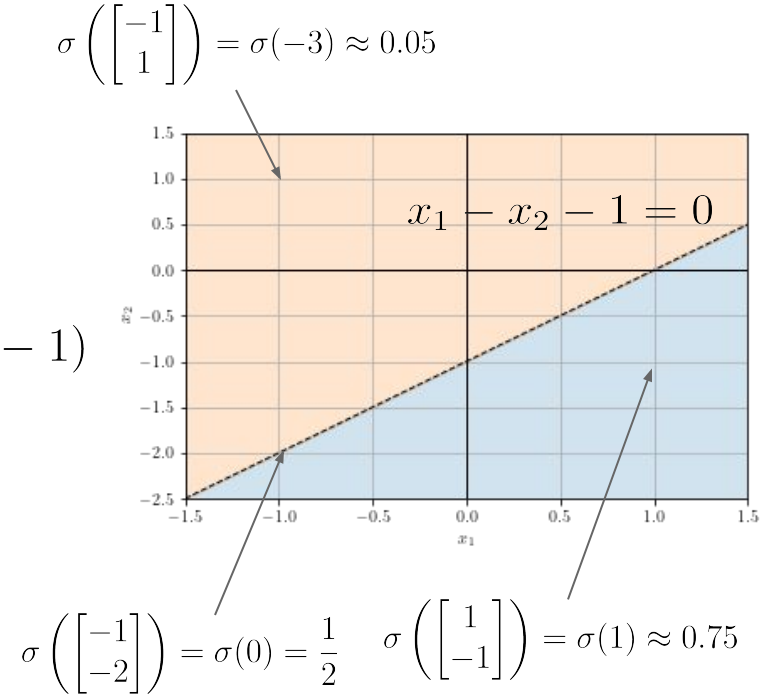
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$L_{\mathbf{w}}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^{m} \left[ - y_i \log \left( \sigma(\mathbf{w}^T \mathbf{x}_i) \right) - (1 - y_i) \log \left( 1 - \sigma(\mathbf{w}^T \mathbf{x}_i) \right) \right]$$

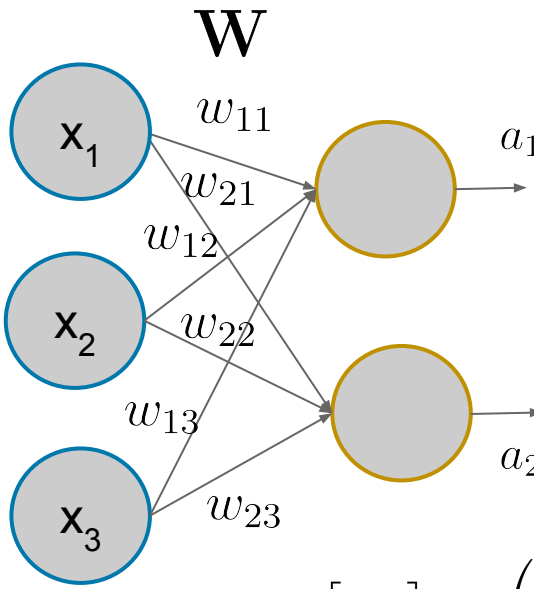
$$\nabla_{w_j} L_{\mathbf{w}}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^{m} \left( \sigma(\mathbf{w}^T \mathbf{x}_i) - y_i \right) x_j$$

# Perceptron: Example



# Perceptron: Multi Output



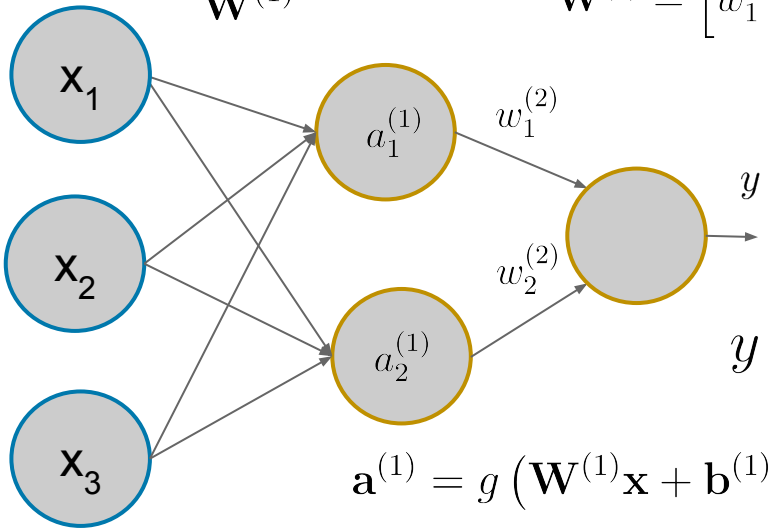

$$a_1 = g(w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1)$$

$$\begin{cases} a_1 = g(w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1) \\ a_2 = g(w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2) \end{cases}$$

$$a_2 = g(w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2)$$

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = g\left( \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & \omega_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \right) = g\Big(\mathbf{W}\mathbf{x} + \mathbf{b}\Big)$$

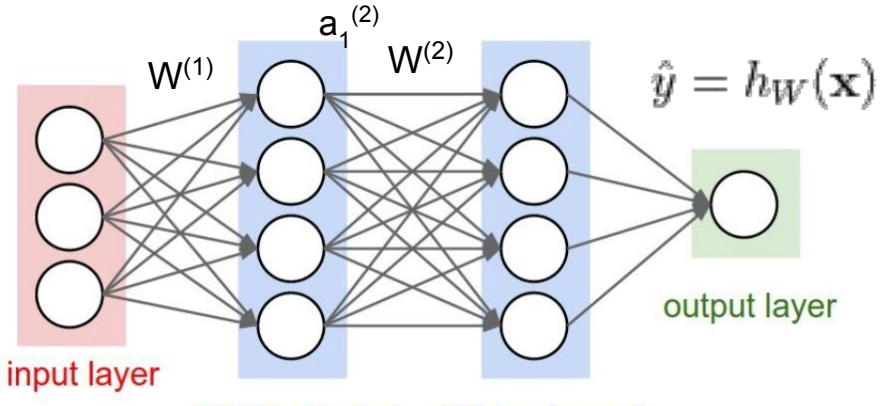
# Single Layer Neural Network




$$y = g\left(\mathbf{W}^{(2)}\mathbf{a}^{(1)} + b^{(2)}\right) = g\left(w_1^{(2)}a_1^{(1)} + w_2^{(2)}a_2^{(1)} + b^{(2)}\right)$$

$$y = g\left(\mathbf{W}^{(2)} g\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right)$$

# Multi-Layer Perceptron MLP




$$\mathbf{a}^{(i)} = g(W^{(i)}\mathbf{a}^{(\mathbf{i-1})} + \mathbf{b}^{(\mathbf{i})})$$

$$\mathbf{a}^{(i)} = g(W^{(i)}g(W^{(i-1)}\mathbf{a}^{(\mathbf{i-2})} + \mathbf{b}^{(\mathbf{i-1})}) + \mathbf{b}^{(\mathbf{i})})$$

$$\hat{y} = h_W(\mathbf{x}) = g(g(g(g(\cdots))))$$

Dense, Feed forward ...

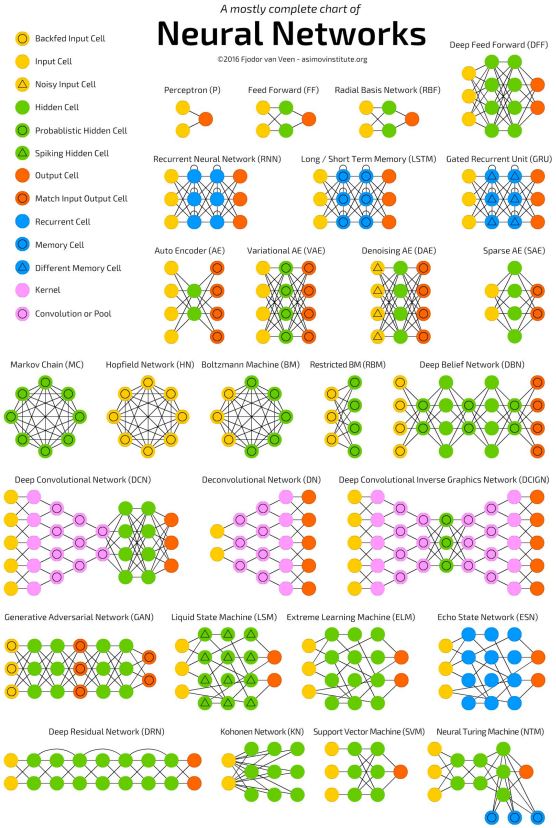
$a_i^{(j)}$ = activation of the i-th neuron of layer j

$W^{(j)}$ = matrix of parameters multiplied by the inputs (activations) from layer j to compute the activations of layer j+1

We see the whole network as a function $h_{\mathbf{w}} : \mathbb{R}^p \rightarrow \mathbb{R}^K$

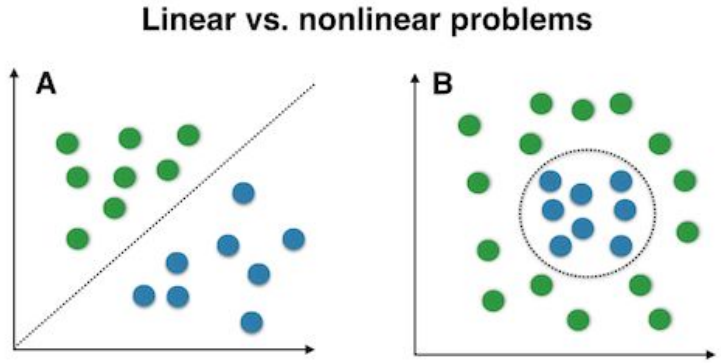
(*p* is the number of features)
(*K* is the number of classes)

# NN Architectures



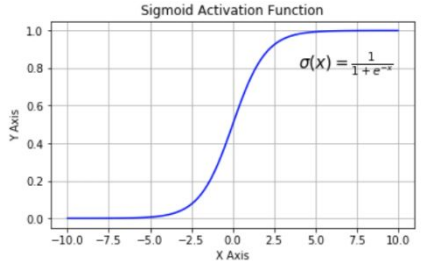

jupyter

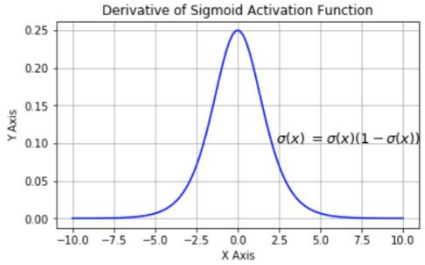
# Activation Functions I

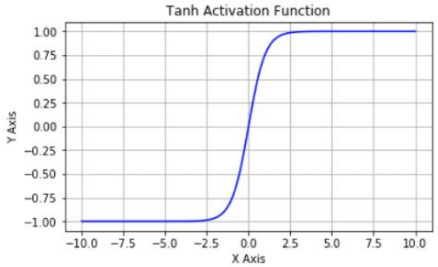


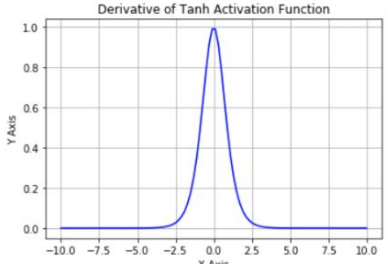

Introduce non-linearities into the network and helps to approximate nonlinear complex functions

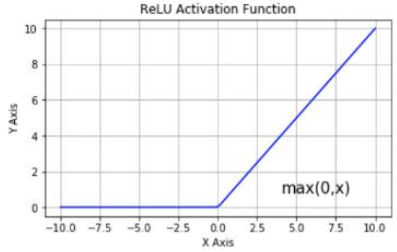
# Activation Functions II

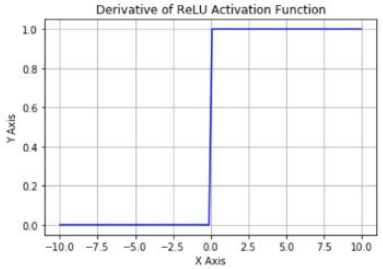



- Sigmoid: (0,1)
  - **Output:** binary classification.
  - Vanishing gradients
  - Non-zero centered

```
tf.keras.activations.sigmoid(x)
```

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

- Tanh: (-1,+1)
  - Vanishing gradients

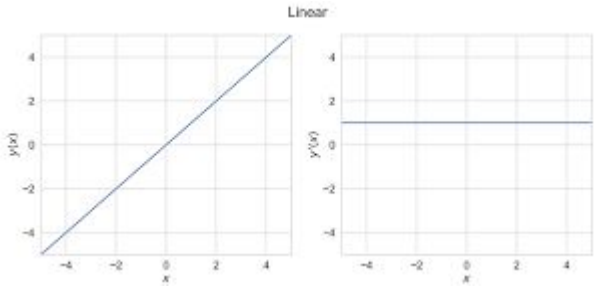
```
tf.keras.activations.tanh(x)
```

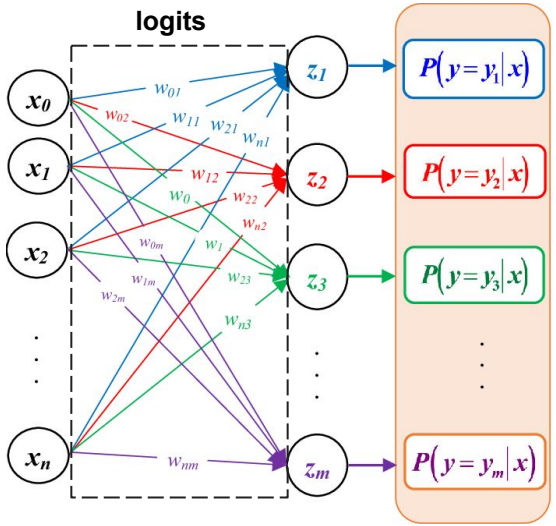
$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- ReLu: [0, +∞) $\mathrm{ReLu}(x) = \max(0, x)$
  - **Output:** Positive regression.
  - Non-zero centered

```
model.add(layers.Dense(64, activation='relu'))
```

# Activation Functions III




$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

- Linear: (-∞, +∞)
  - **Output:** Regression.
  - Only for output layer.
  - Any activation.

- Softmax:
  - Softmax converts a real vector to a vector of categorical probabilities.
  - Output: Multi-class classification.
  - Multiple neurons.

```
tf.keras.activations.softmax(x, axis=-1)
```

# Loss/Cost Function

We need to find the best parameters for model the given data.

**Loss function:** quantifies the error in prediction, how the model deviates from the correct results.

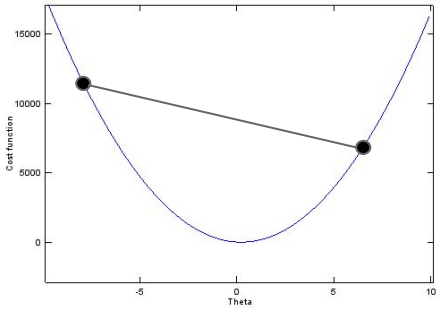
**Objective function, cost function:** Measure the total loss over the entire dataset

$$\mathcal{L}\left(f\left(\boldsymbol{x}^{(i)};\boldsymbol{W}\right),\boldsymbol{y}^{(i)}\right)$$

Prediction

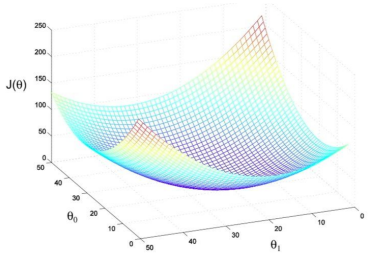
Correct

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\left(f\left(\boldsymbol{x}^{(i)};\boldsymbol{W}\right),\boldsymbol{y}^{(i)}\right)$$




Cost function of one parameter

Ideally the **cost function is convex**: for every pair of points, the curve is always below the line (or hyper-plane) between them




Cost function of two parameters

# Cost Functions: Regression

- Predict a numerical value given some input.
- **Mean squared error (MSE):** Average of the **square** of the difference between the predicted and actual target variables.
  - The most classic measure.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

```
model.compile(optimizer='sgd', loss='mse')
```

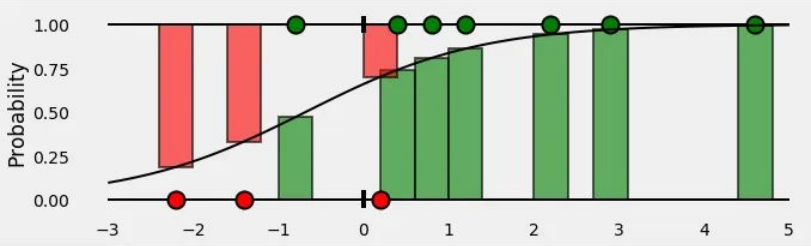
- **Mean absolute error (MAE)**: Average of the **absolute value** of the difference between the predicted and actual target variables.
  - Harder differentiability and convergence.
  - More robust to outliers than MSE.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$$

```
model.compile(optimizer='sgd', loss='mae')
```

# Cost Functions: Classification

- Specify which of k categories some input belongs to.




- **Binary Cross-Entropy (log-loss):** Similar to the logistic loss function

$$\mathcal{L}(\hat{y}, y) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)$$

```
model.compile(optimizer='sgd', loss='binary_crossentropy')
```

- **Categorical Cross-Entropy:**
  - Minimizing the cross-entropy is equivalent to minimizing the **divergence** between the distribution of the real targets and the predictions.

$$\mathcal{L}(\hat{y}, y) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \cdot \log(\hat{y_{ik}})$$

```
model.compile(optimizer='sgd', loss='sparse_categorical_crossentropy')
```

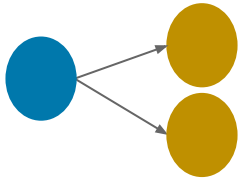
# Optimization: dimension problem

Cost optimization of networks is not convex we can reach local minima.
The dimension of the hyperparameter search space is huge, we cannot apply methods that ensure global minima.

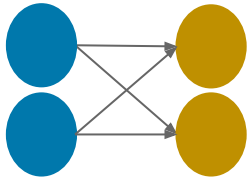
**Neuroanl Network**

**1 parameter**

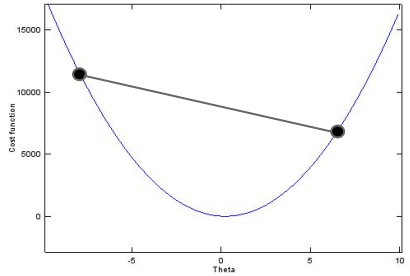
**2 parameter**

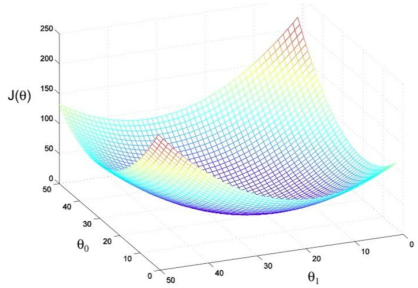


**4 parameter**



**Search space**

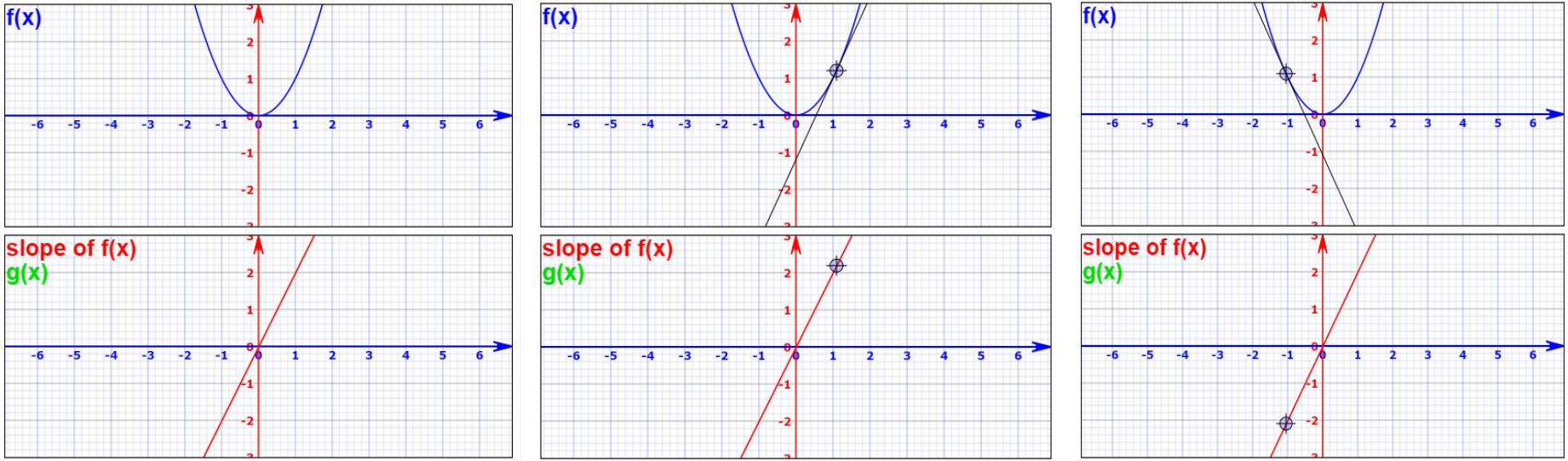


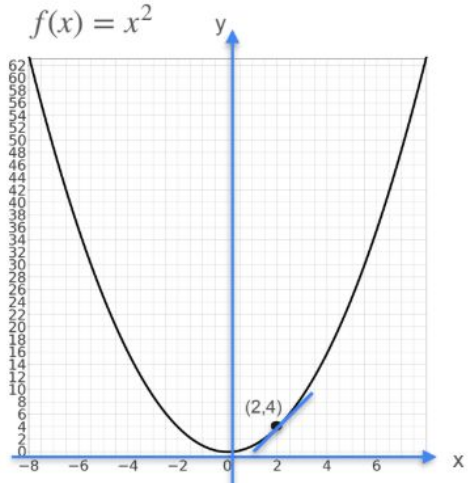

**Cannot be visualize**

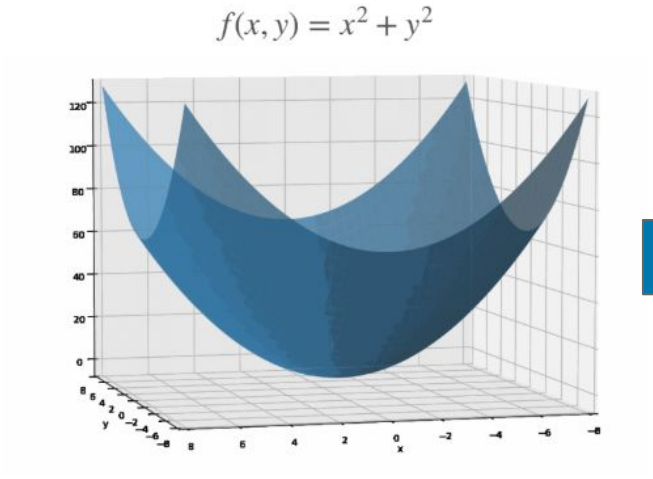
# Optimization: derivative

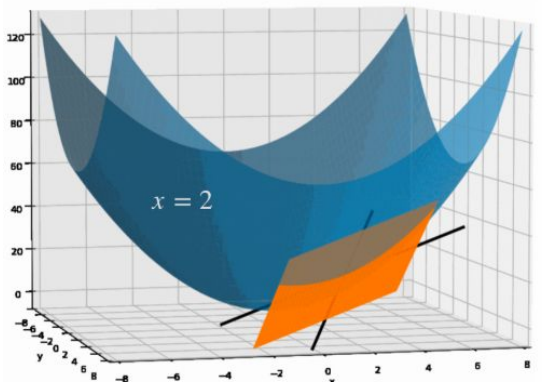
To optimize our cost function we will use the derivative of the activation functions of our network.
The derivative allows us to obtain the slope at a given point.



# Optimization: 2D derivative

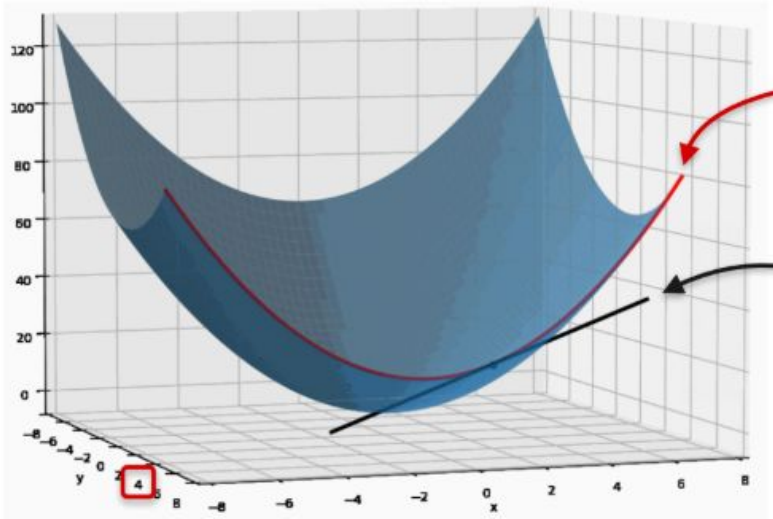


# Optimization: partial derivative

$$f(x, y) = x^2 + y^2$$




Treat y as a constant

Function of one variable x

Slope?

Partial derivative

Constant

$$f(x, y) = x^2 + \boxed{y^2}$$

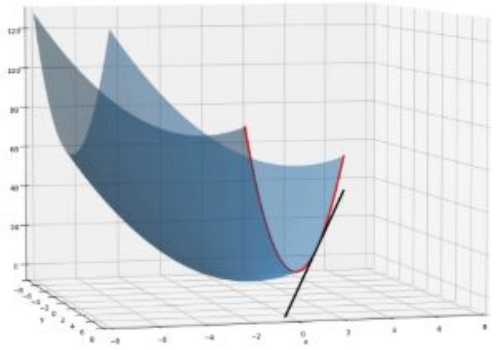
$$\frac{\partial f}{\partial x} = 2x + \boxed{0}$$

Derivative = 0

# Optimization: gradient

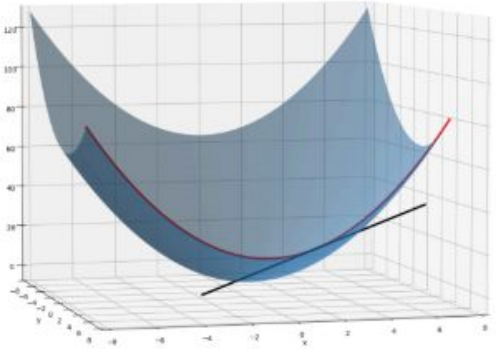
$$f(x, y) = x^2 + y^2$$

Treat y as a constant



$$\frac{\partial f}{\partial x} = 2x$$

Treat x as a constant



$$\frac{\partial f}{\partial y} = 2y$$

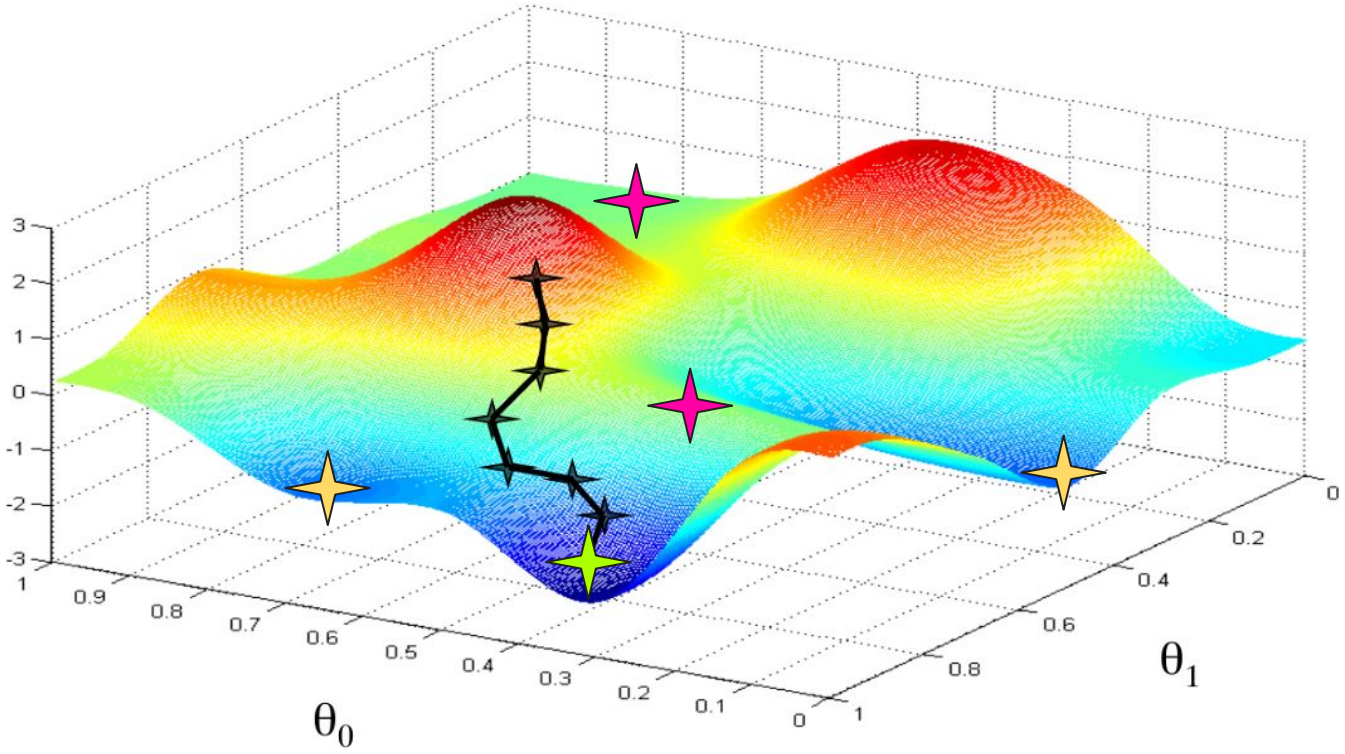
Gradient

$$\begin{bmatrix} 2x \\ 2y \end{bmatrix}$$

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

# Optimization

Follow gradient direction $\nabla_W J$

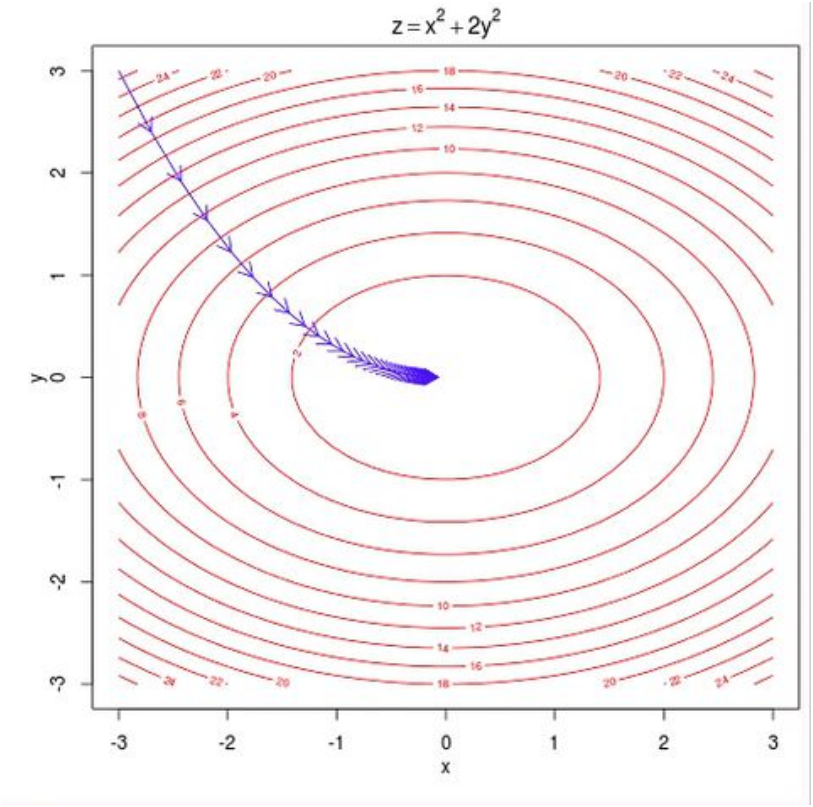



Local minimum

Saddle point

Global minimum

# Optimization: Gradient Descend

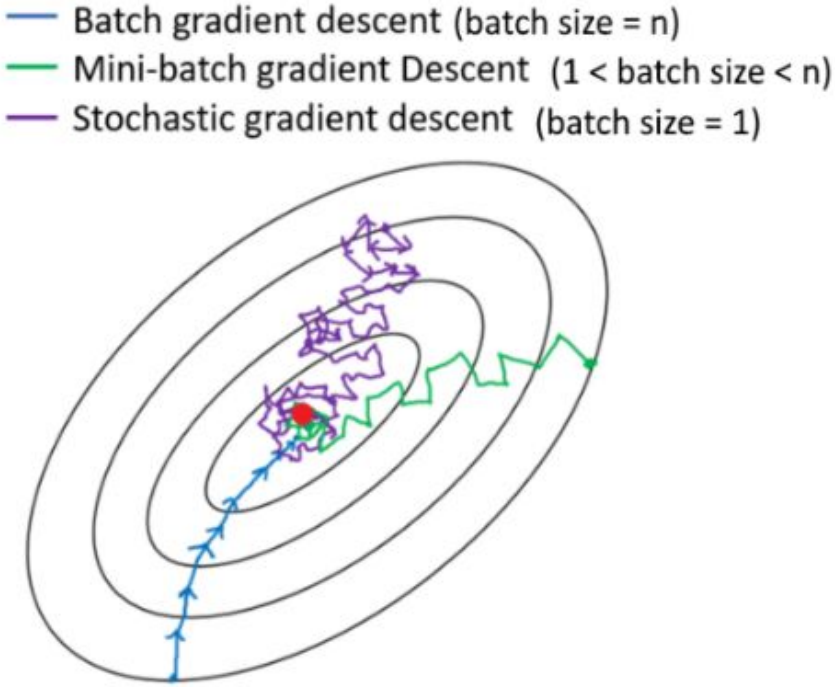



- In every step goes through the entire dataset

$$\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\left(f\left(x^{(i)};\boldsymbol{W}\right),\boldsymbol{y}^{(i)}\right)$$

- We need big datasets…
- Good for convex functions.

# Optimization: Mini-batch Gradient Descent




- **Stochastic gradient descent:**
  - Estimate gradient with only one sample choose randomly

$$\nabla_W J(\mathbf{W}) \approx \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right)\right.$$

- **Mini-batch gradient descent:**
  - estimate gradient with some samples (m)

$$\nabla_W J(\mathbf{W}) \approx \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right)\right.$$

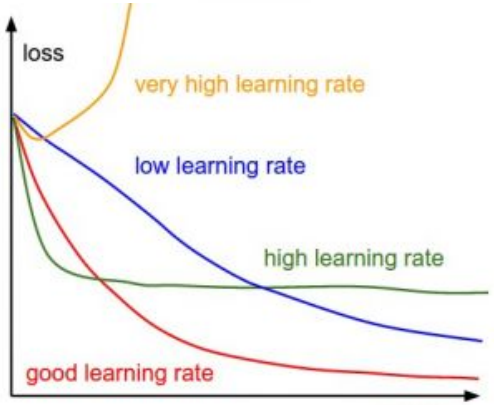
- **Batch size:** Length of the minibatch
- **Iteration:** Every time we update the weights
- **Epoch:** One pass over the whole training set.

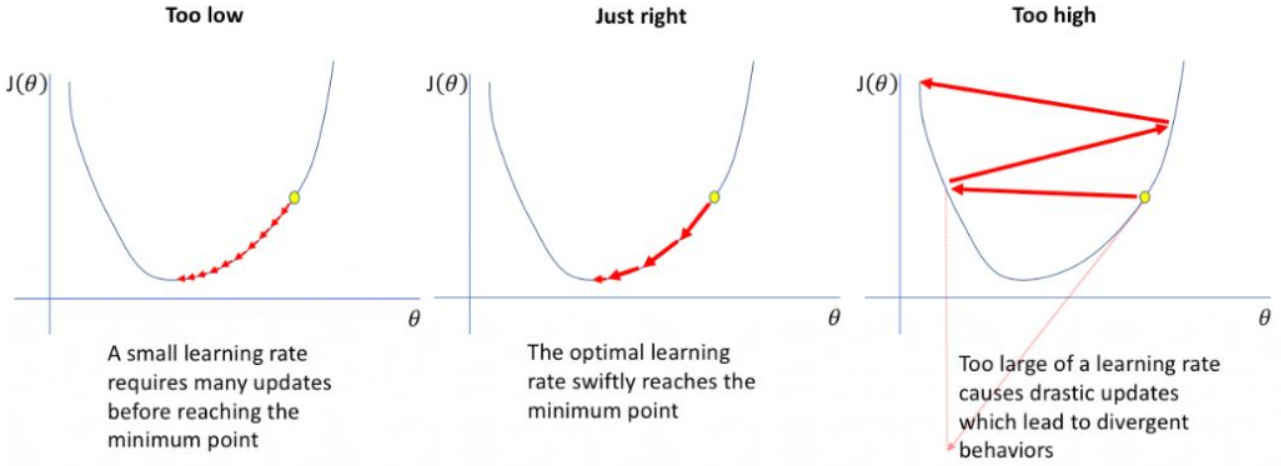
# Optimization: Learning rate

- Initialize weights randomly
- In each step, update weights until convergence:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \cdot \nabla_W J(\mathbf{W})$$

**Learning rate**



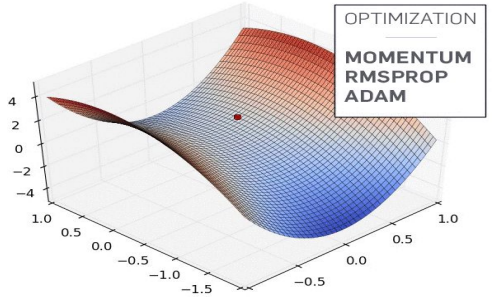
# Optimization: Advances

**Root Mean Square Propagation (RMSProp)**

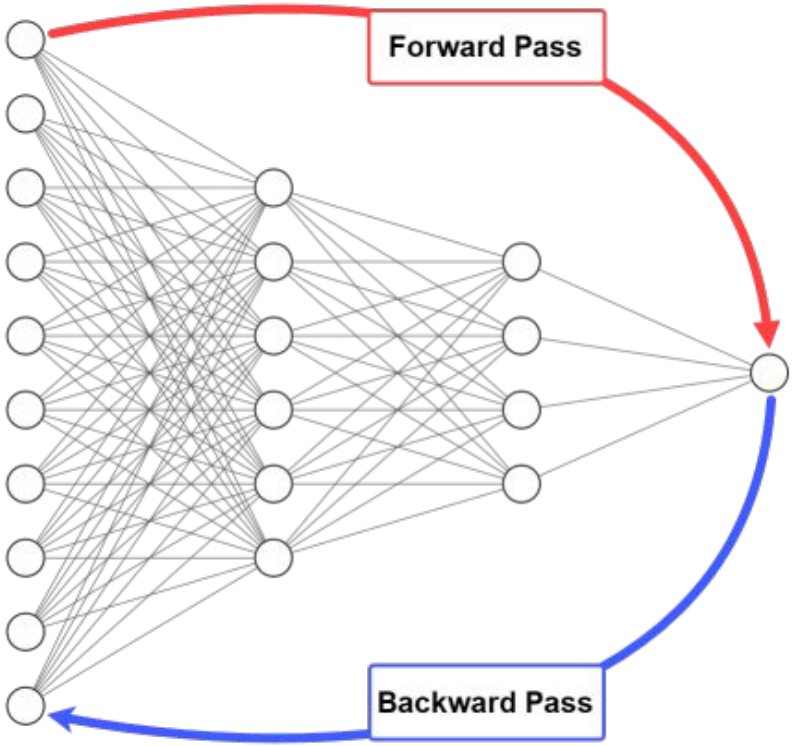
- The direction of the gradient need not be always the one that indicates the path to the global optimum.
- Take this factor into account by scaling the gradient in the dimension with the steepest slope.

**Adaptive Moment Optimization (Adam)**

- Designed to treat sparse gradients.
- Improves convergence through momentum which ensures that the model does not get stuck at saddle point.



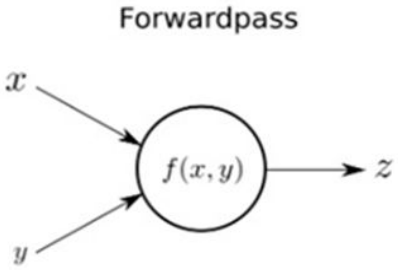
# Backpropagation

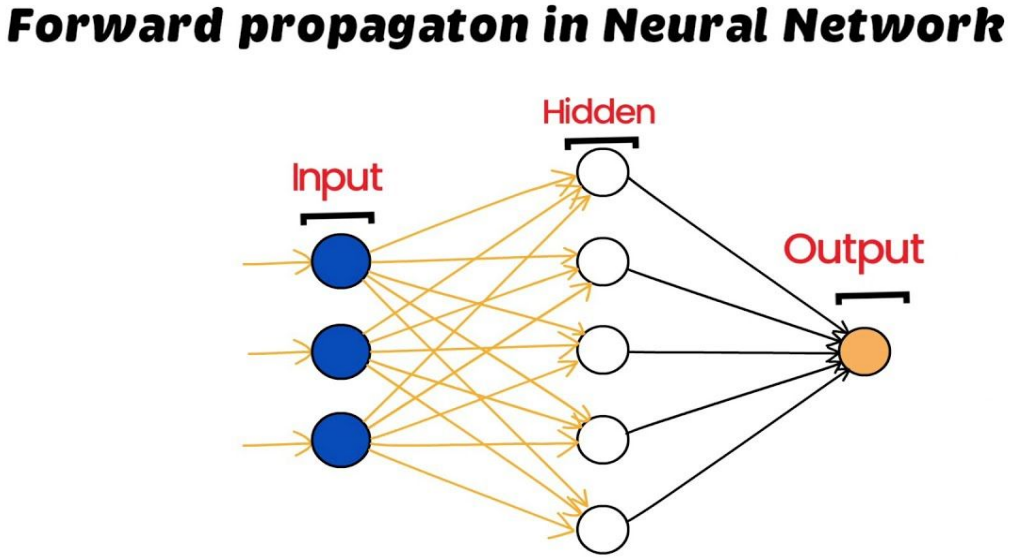



- **Intuition**: calculate the contribution of each neuron to the final error and modify their weights accordingly.
- Algorithm to calculate the partial derivatives of the cost function with respect to each parameter.
- The process is composed of two parts: the forward process and the backward process.

# Backpropagation: forward

Solve the outputs of each hidden layer and the output of the current layer based on the current weights.

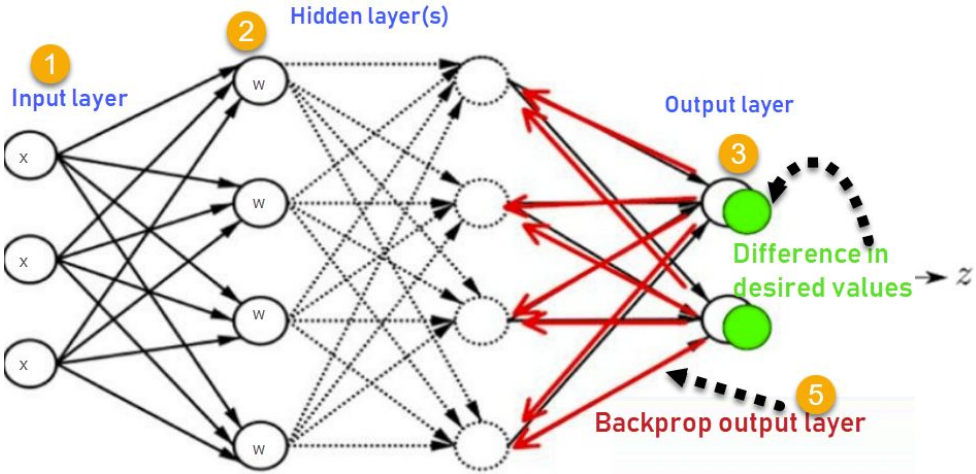


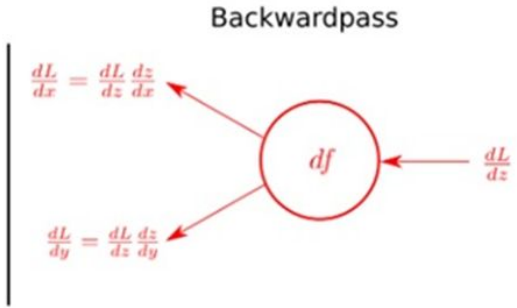



# Backpropagation: backward

- The difference of the values predicted by the network with respect to the real values is calculated.

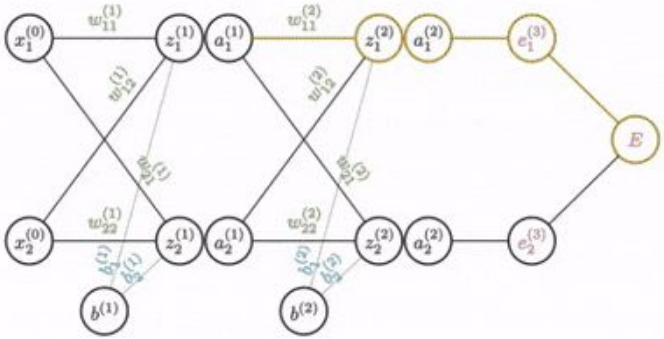
- The error is propagated from the last layer to the initial layers by calculating the gradients used to update the weights in this iteration.



# Backpropagation: chain rule

$$\frac{\partial E}{\partial w_{11}^{(2)}} = \frac{\partial e_1^{(3)}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{11}^{(2)}}$$




$E$ total error

$e_n^{(N)}$ unit error

$a_n^{(N)}$ activation output

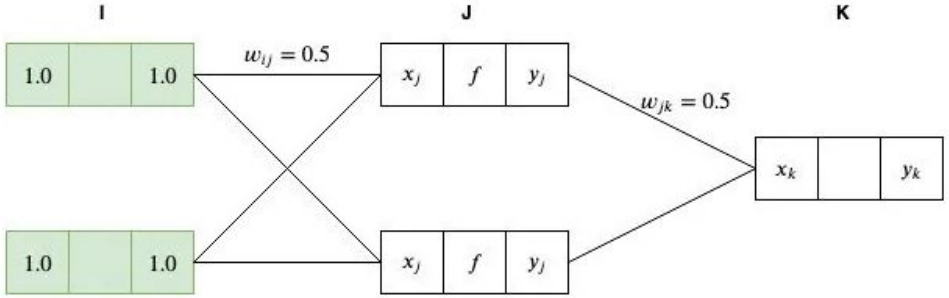
$z_n^{(N)}$ weights * input

$w_n^{(N)}$ weights

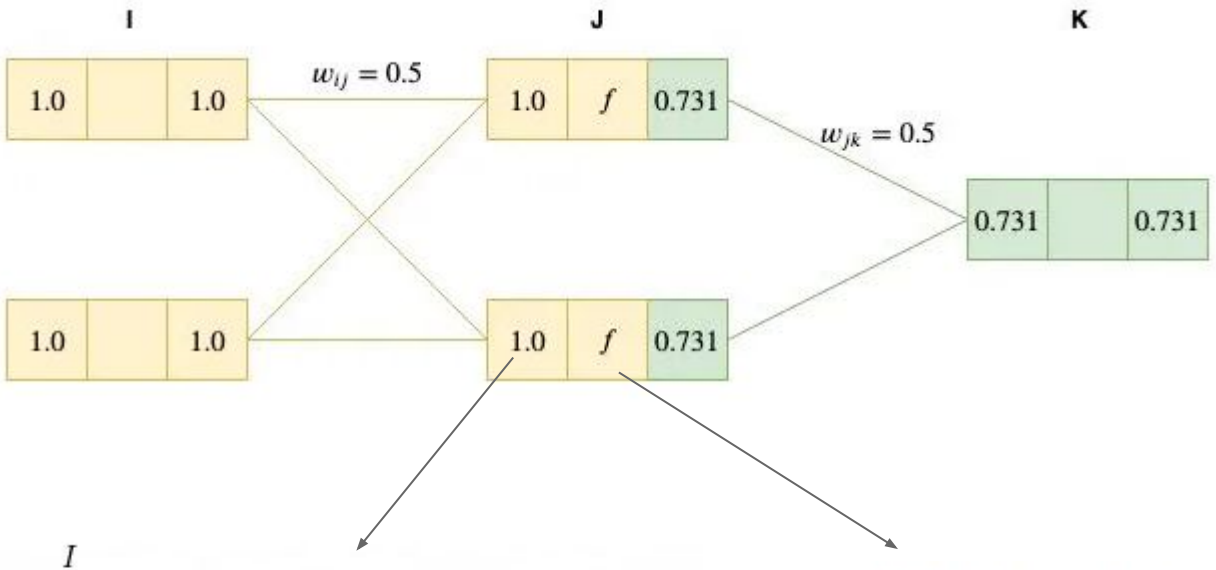
https://towardsdatascience.com/understanding-backpropagation-abcc509ca9d0

# Backpropagation example

- Regression problem
- Input example =  {x_1=1, x_2=1}
- All weights initialized as 0.5
- Activation fn in hidden layer = sigmoid
- Activation fn in final layer = linear
- Loss function =  y - y_pred



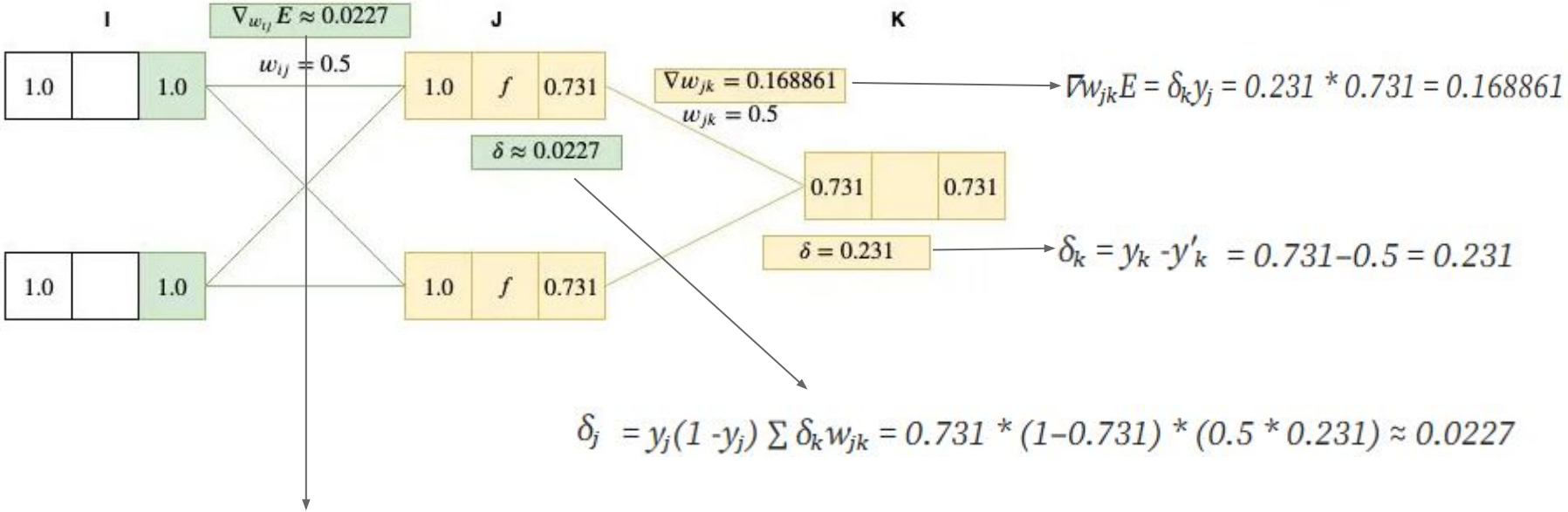
# Backpropagation example: forward




$$x_j = \sum_{i=1}^{I} y_i w_{ij} = 1.0 * 0.5 + 1.0 * 0.5 = 1.0$$

$f(x) = 1/(1 + e^{-x})$

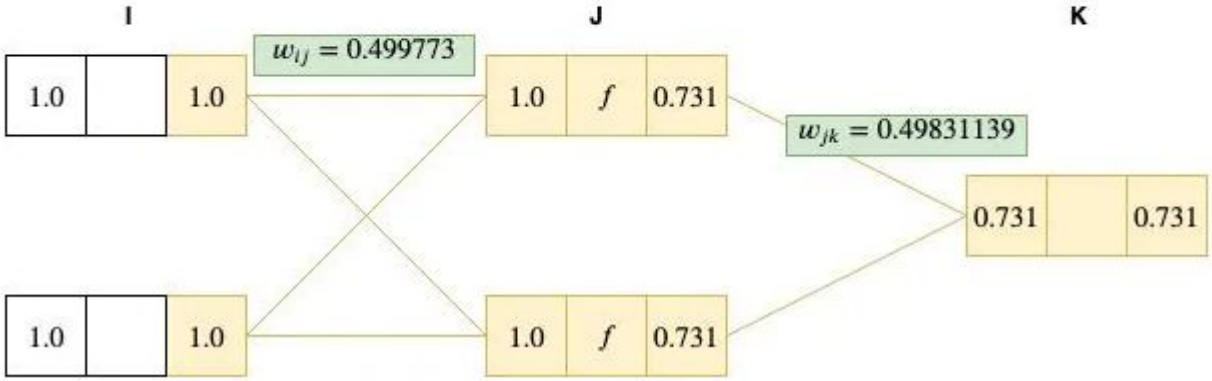
# Backpropagation example: backward



# Backpropagation example: update weights

- $w_{ij} = w_{ij} - \epsilon \nabla w_{ij} E = 0.5 - 0.01 * 0.0227 = 0.499773$
- $w_{jk} = w_{jk} - \epsilon \nabla w_{jk} E = 0.5 - 0.01 * 0.168861 = 0.49831139$

Iteration completed

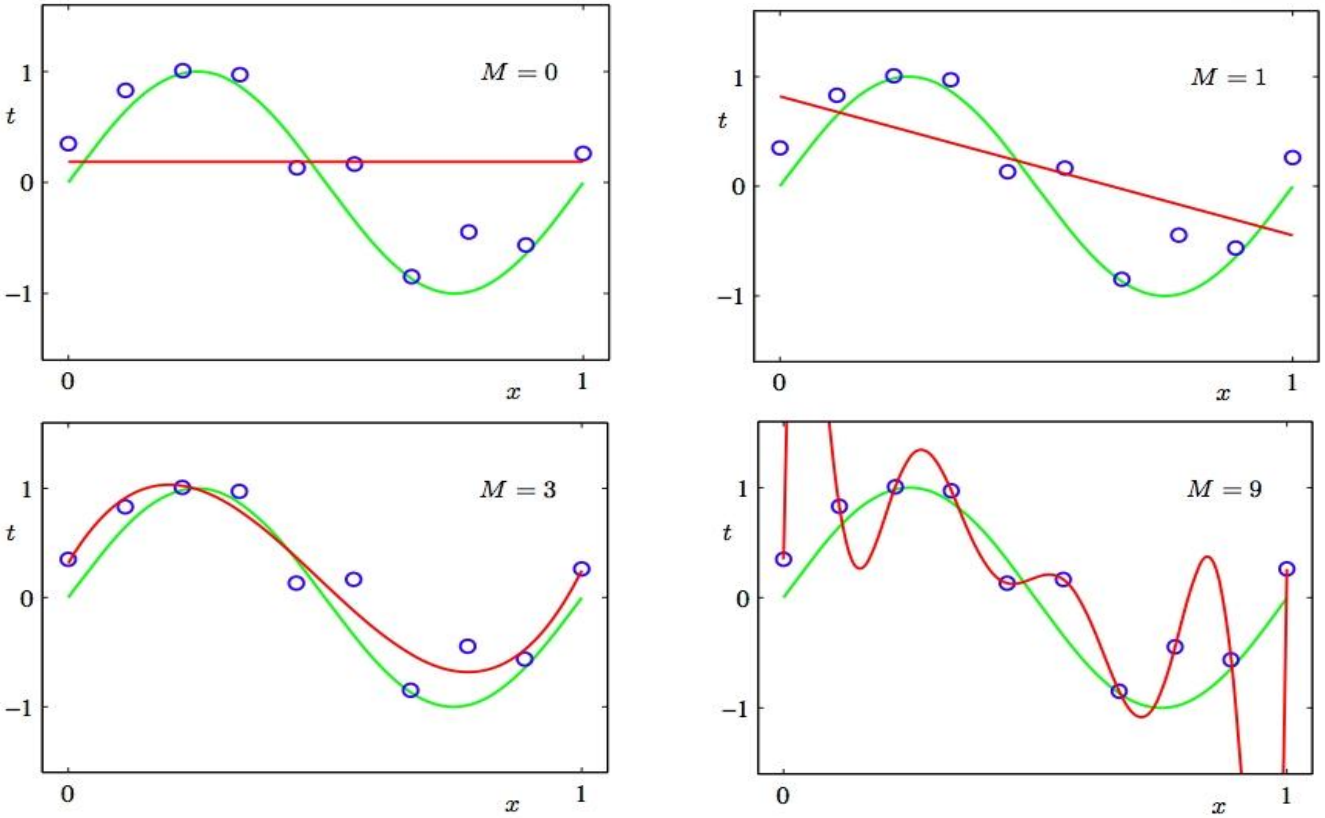



jupyter

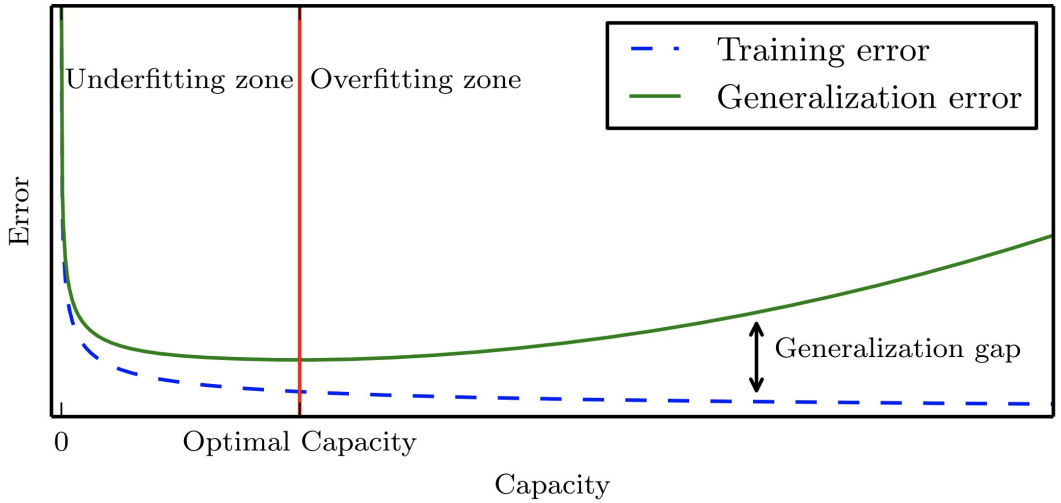
# 1.3

## Regularization techniques

# Regularization: Overfitting



# Regularization: Overfitting



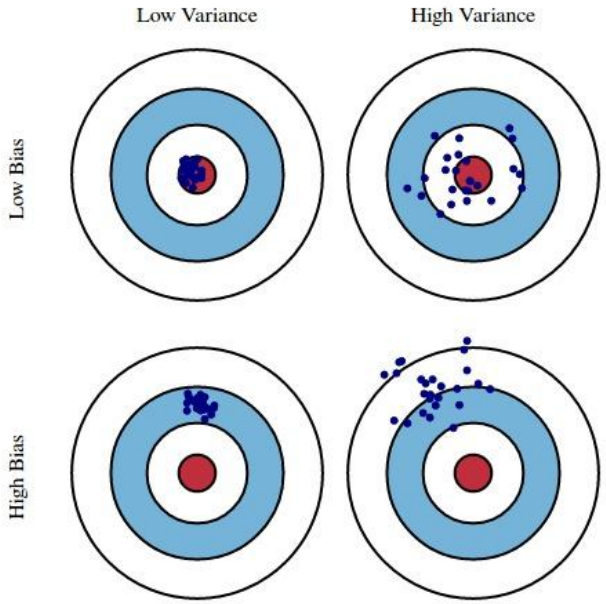

**Generalization**: The ability to perform well on previously unobserved inputs.

During training we reduce **training error**.

**Generalization error ( test error):** Expected value of the error in a new unobserved input. We want reduce test error, not only training (as in an optimization task)

**Capacity**: ability to fit a wide variety of functions.

# Regularization: Overfitting




$$\text{Bias}_D\left[\hat{f}(x;D)\right] = \mathrm{E}_D\left[\hat{f}(x;D)\right] - f(x)$$

$$\text{Var}_D\left[\hat{f}(x;D)\right] = \mathrm{E}_D[\left(\mathrm{E}_D[\hat{f}(x;D)] - \hat{f}(x;D)\right)$$

**Generalization** : The ability to perform well on previously unobserved inputs.

Trade off :

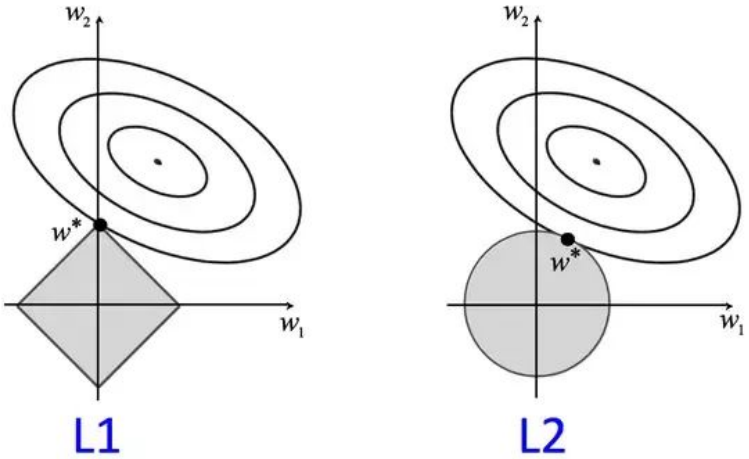
- Overfitting :
  - Low bias, High variance.
  - Complex, flexible model.
  - Memorizes training data, captures noise.
  - Gap between training and test error is big.
- Underfitting :
  - High bias, Low Variance
  - Simple, rigid model.
  - Fails to capture underlying patterns in the data.
  - High training and test error, poor performance.
- Solutions:
  - **Overfitting**: Simplify the model, use more training data, apply regularization, reduce features.
  - **Underfitting**: Increase model complexity, use more features, feature engineering, fine-tune hyperparameters.

# Regularization: Fundamentals

- **Regularization:** Any modification we make to a learning algorithm for reducing its generalization error but not its training error.

- Lots of types:
  - Adding restrictions on the parameter values.
  - Adding extra terms in the objective function.

# Regularization: L2 y L1

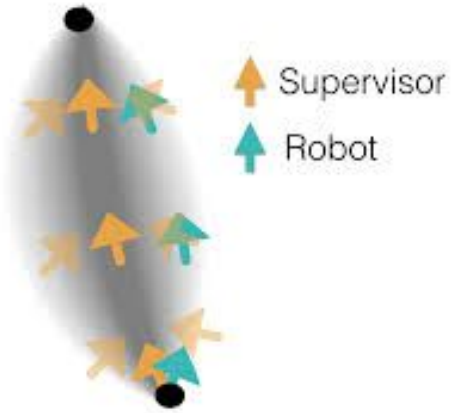



- Add a penalty to the cost function:

$$\tilde{J}(W) = J(W) + \lambda \cdot F(W)$$

- L2: $||W||_2^2 = \sum_i w_i^2$
  - Keeps weights near zero.
  - Simplest one, differentiable.
- L1: $||W||_1 = \sum_i |w_i|$
  - Sparse results, feature selection.
  - Not differentiable, slower.

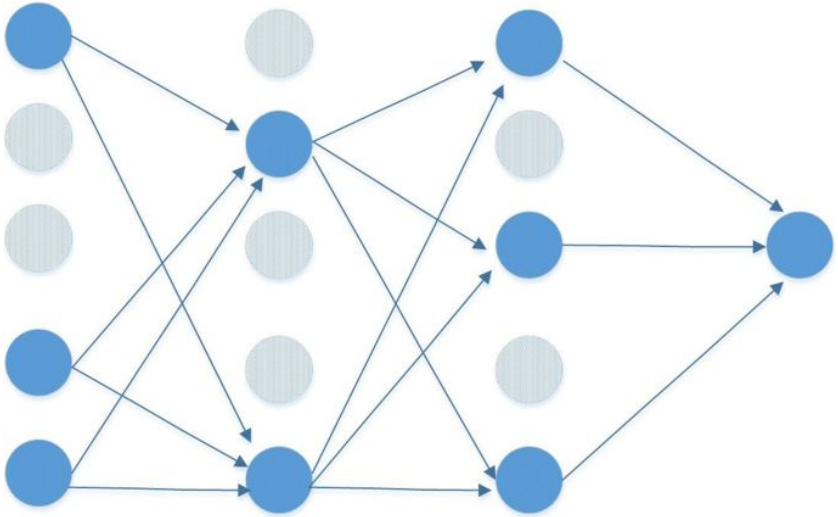
# Regularization: Noise injection




Noise Injection

- Add Random Noise During Training.
- Injecting noise in the input to a neural network can also be seen as a form of data augmentation.
- With a bayesian point its equivalent to other regularization:
  - Gaussian noise: L2
  - Laplacian noise: L1

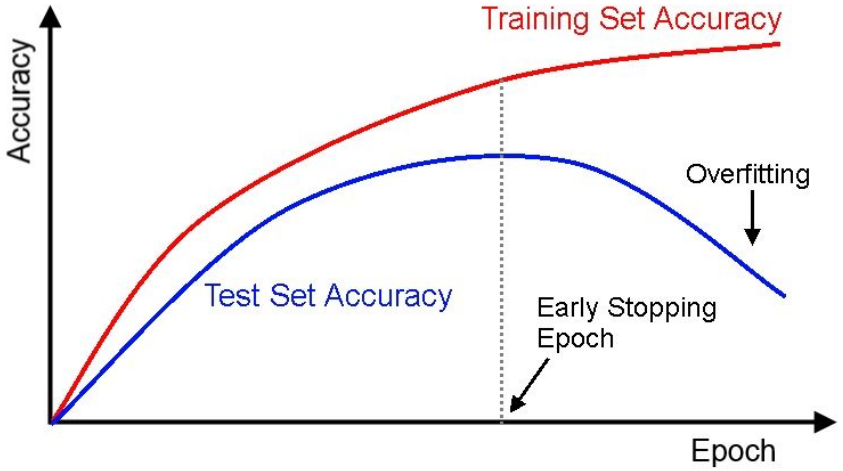
# Regularization: Dropout



- During training, randomly set some activations to 0 with probability p.
- Not in prediction.

```
from tensorflow.keras.layers import
Dropout
model = Sequential()
model.add(Dense(60, activation='relu',
input_shape=input_shape))
model.add(Dropout(0.2))
model.add(Dense(30, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

# Regularization: Early Stopping

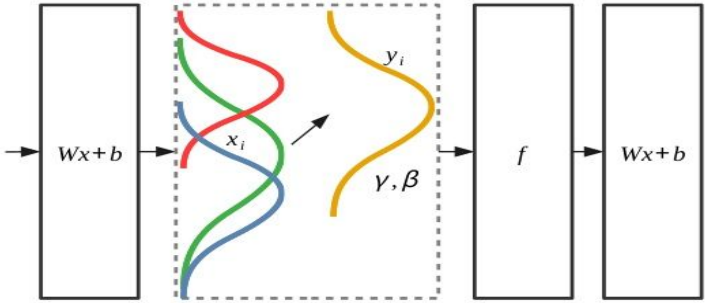



- Early stop before overfitting occurs
- Usually controlled by indicating the maximum number of epochs that the validation error can remain unimproved.
- The weights of the best model are preserved to avoid re-training again.

# Regularization: normalizations

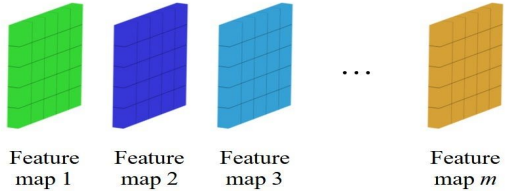
**Batch normalization**

Ensure the output statistics of a layer are fixed.




- Normalizing input activations across mini-batches to improve training stability.
- Improves convergence and speeds up training.
- Commonly used in CNN

LAYER NORMALIZATION

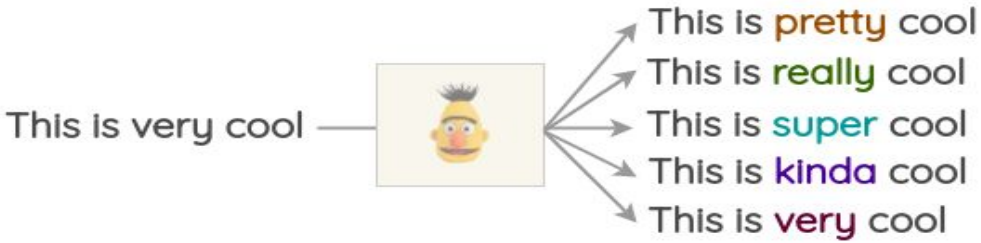



Normalize layer output for one training sample

- Normalizing input activations of each neuron by calculating the mean and standard deviation over the features of a single input.
- Better for small batch sizes or sequential tasks
- Commonly used in RNN or transformers

**Both methods have two learnable parameters:**
**γ:** A scaling factor. After normalizing the activations, the output is scaled by this parameter.
**β:** A shifting factor. After scaling, the output is shifted by this parameter.

# Data Augmentation



# 1.4

## Exercices

# Prevent Overfitting, Regression, Imbalanced


UNIVERSIDAD
COMPLUTENSE
DE MADRID