



# Deep Learning

Alberto Ezpondaburu



## About Me

60% Math    35% Computers    5% Philosophy



### SOME FRAMEWORK PROFICIENCIES

---

- 
- A vertical list of five areas of expertise, each preceded by a black star:
- NLP
  - Machine Learning
  - Deep Learning
  - Signal Processing
  - Computer Vision

### SOME AREAS OF EXPERTISE

---

## Analytical Index

1. Introduction to Deep Learning
2. Convolutional Neural Networks (CNN)
3. Recurrent Neural Networks (RNN)
4. Natural Language Processing with Deep Learning
5. Deep Generative Modelling

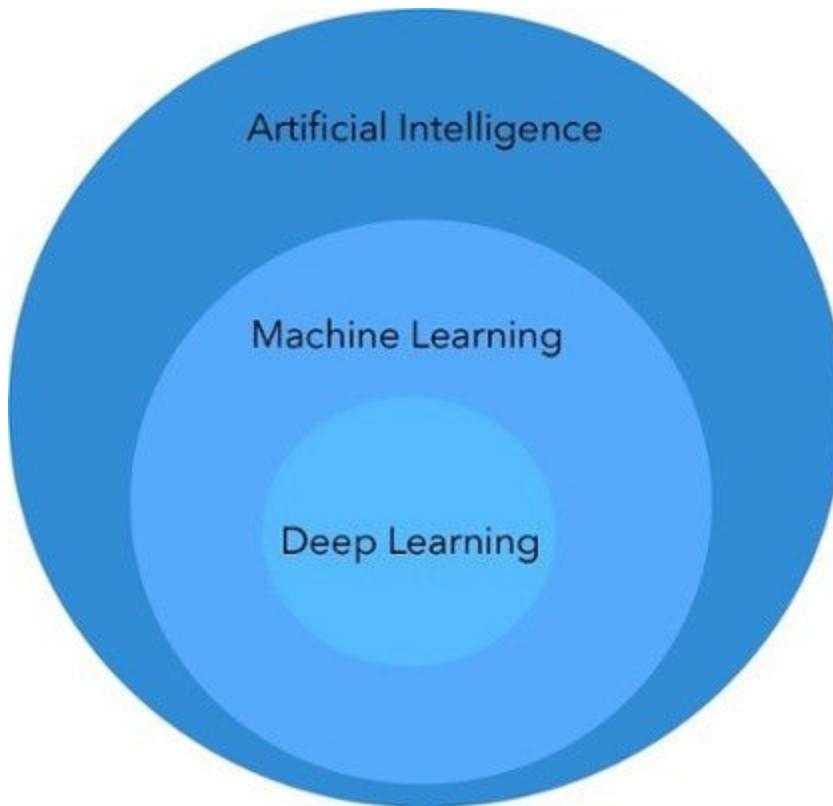
# Analytical Index

1. Introduction to Deep Learning
  - a. DL vs ML
  - b. DL Frameworks
  - c. DL Fundamentals
  - d. Exercises
2. Convolutional Neural Networks (CNN)
3. Recurrent Neural Networks (RNN)
4. Natural Language Processing with Deep Learning
5. Deep Generative Modelling

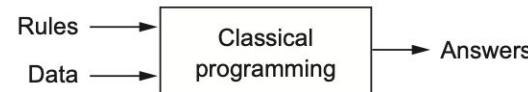
# 1.1

## Introduction To Deep Learning

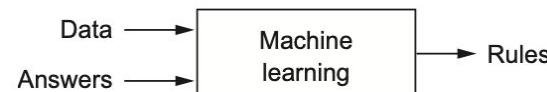
# AI vs ML vs DL



- **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

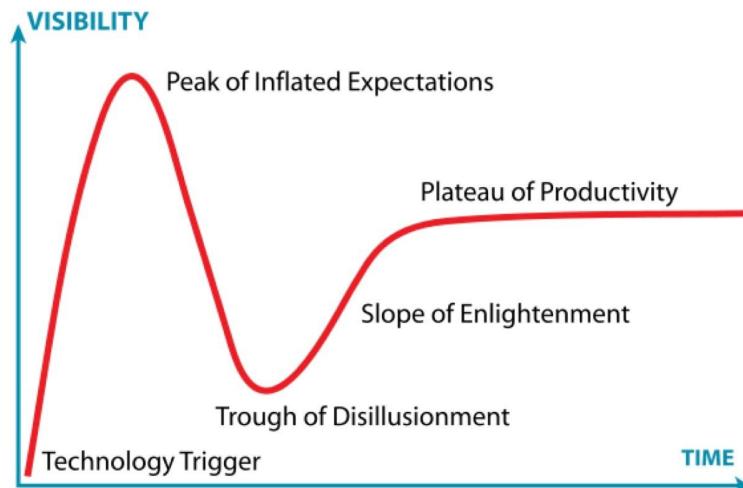


# Why Deep Learning?

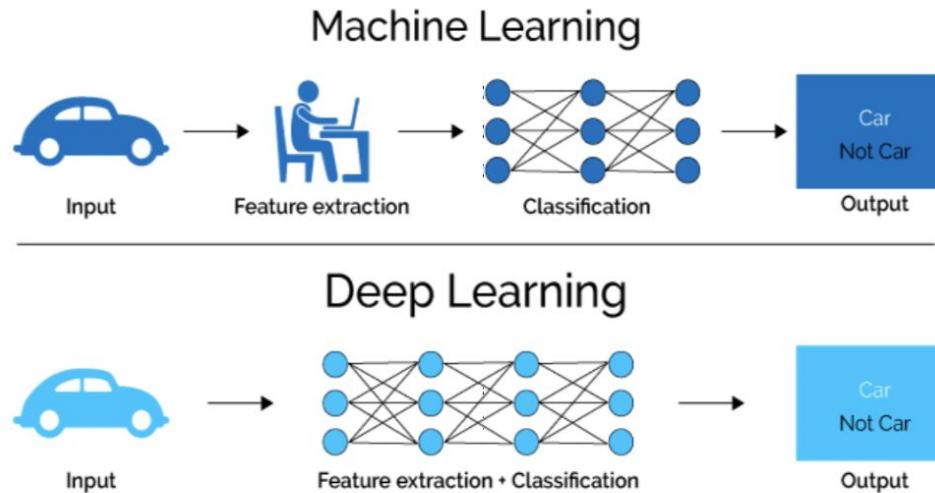
NLP	Sentiment analysis, speech recognition, chatbots, machine translation, question answering, text summarization, handwriting generation...
Computer Vision	Object detection, face recognition, image segmentation, image captioning, optical character recognition (OCR), emotion recognition, gesture recognition...
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...

# Why Deep Learning?

## Gartner Hype Cycle



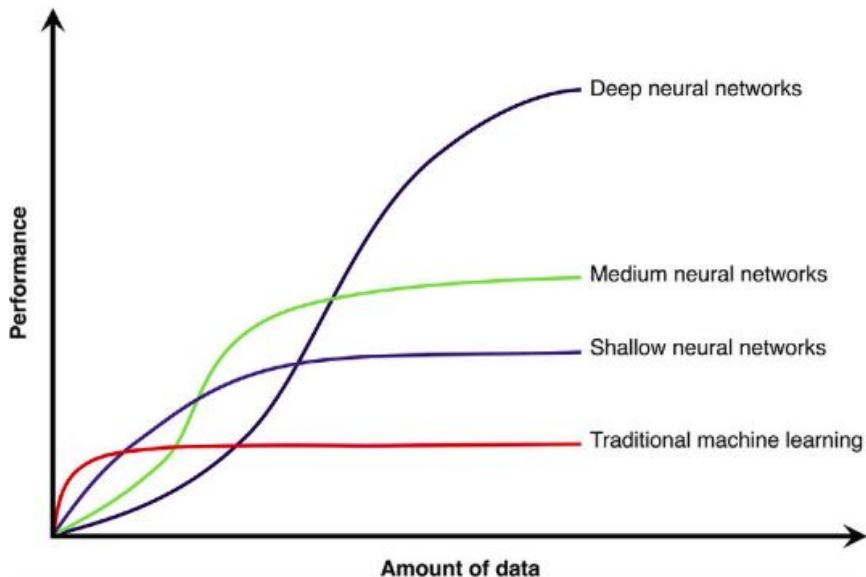
# Why Deep Learning?



# Why Deep Learning?

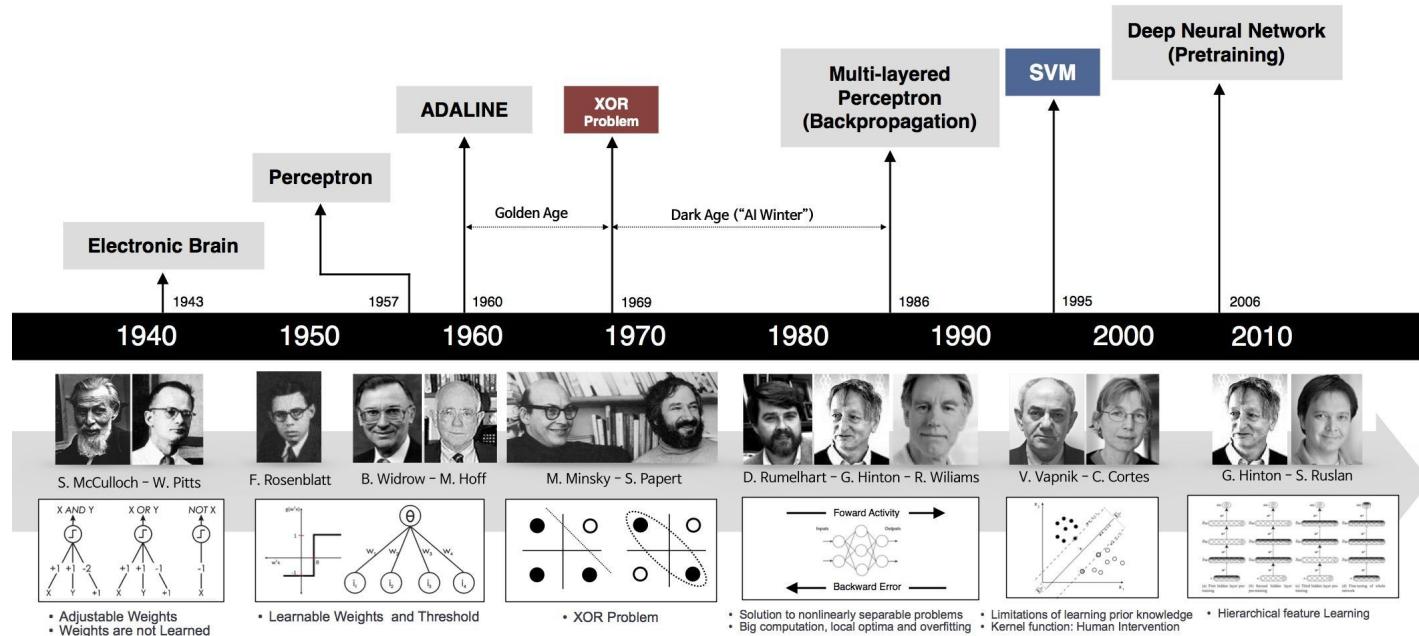


# Why Deep Learning?



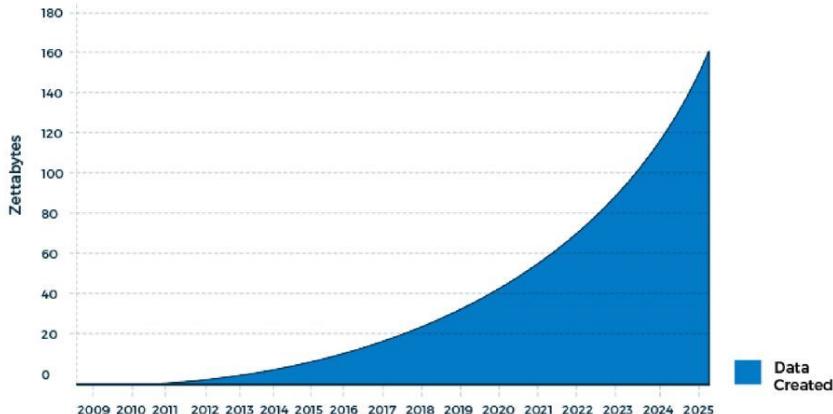
- Data:
  - Structured: tables
  - Unstructured: Audio, image, text
- Performance increases a lot with depth and with the amount of data. Specially with unstructured data.
- With small datasets, traditional machine learning usually performs better.

# Why Now?



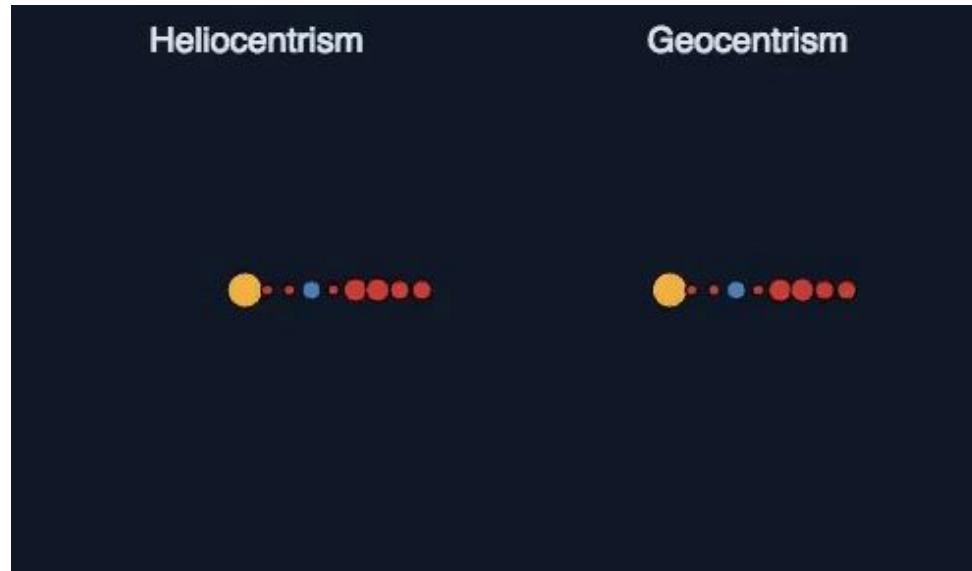
# Why Now? Data

Annual Size of the Global Datasphere



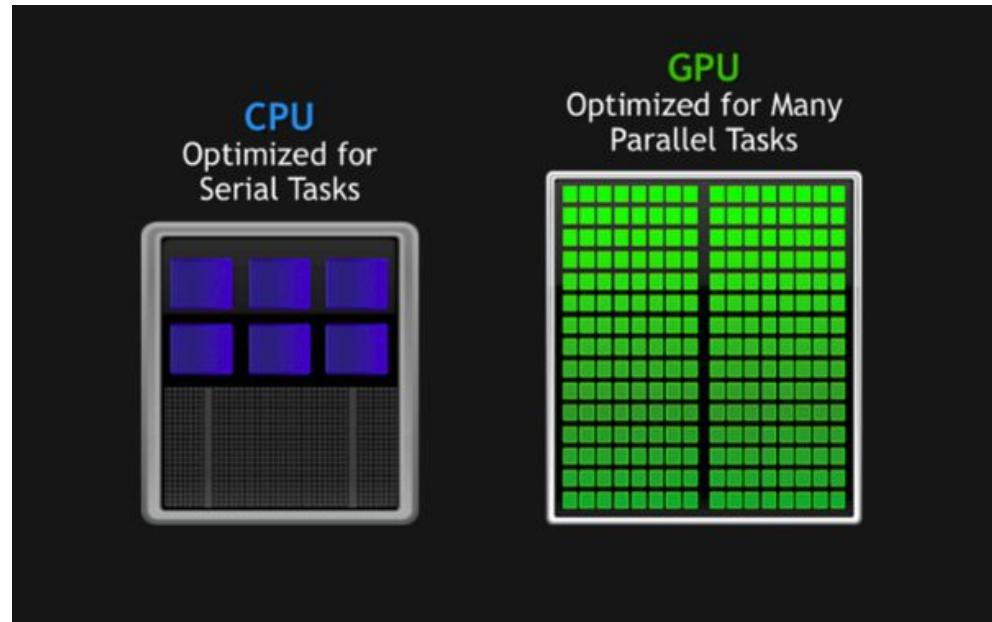
- Raw material that powers AI.
- The new electricity: Just as electricity revolutionized industries, deep learning is transforming businesses, research, and technology.
- ZettaByte Era:
  - Connected devices per person : 0.08 in 2000, 3.47 in 2015, 6.58 in 2020.
  - Before computers, humanity accumulated **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? Data



# 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



theano

R.I.P.

Caffe

Contenders

DEEPMLEARNING4J

Rockstars



K 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.

## Example: Digit Recognition



# Example: Digit Recognition - Code Generation (GPT-4)

A Create a notebook with comments for my deep learning master students to see their first tensorflow model of MNIST digit classification with a dense network

Below is a Jupyter Notebook outline with comments for an introductory TensorFlow model for MNIST digit classification using a dense network. This outline will help guide your students through the process of implementing their first TensorFlow model for this task.

```
python ⌂ Copy code
# Import required libraries
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Load the MNIST dataset
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize the input data
x_train, x_test = x_train / 255.0, x_test / 255.0

# Define the model architecture
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)), # Flatten the 28x28 input
    tf.keras.layers.Dense(128, activation='relu'), # Dense layer with 128 neurons
    tf.keras.layers.Dropout(0.2), # Dropout layer for regularization to prevent overfitting
    tf.keras.layers.Dense(10, activation='softmax') # Output layer with 10 neurons
])

# Compile the model
model.compile(optimizer='adam', # Use the Adam optimizer
              loss='sparse_categorical_crossentropy', # Sparse categorical crossentropy loss
              metrics=['accuracy']) # Track accuracy metric during training

# Train the model on the dataset
history = model.fit(x_train, y_train, epochs=10, validation_split=0.1)

# Evaluate the model on the test dataset
test_loss, test_accuracy = model.evaluate(x_test, y_test)

# Print the test loss and accuracy
print(f"Test loss: {test_loss}, Test accuracy: {test_accuracy}")

# Visualize the training history
plt.plot(history.history['accuracy'], label='Training accuracy')
plt.plot(history.history['val_accuracy'], label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
```

```
# Define the model architecture
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)), # Flatten the 28x28 input
    tf.keras.layers.Dense(128, activation='relu'), # Dense layer with 128 neurons
    tf.keras.layers.Dropout(0.2), # Dropout layer for regularization to prevent overfitting
    tf.keras.layers.Dense(10, activation='softmax') # Output layer with 10 neurons
])

# Compile the model
model.compile(optimizer='adam', # Use the Adam optimizer
              loss='sparse_categorical_crossentropy', # Sparse categorical crossentropy loss
              metrics=['accuracy']) # Track accuracy metric during training

# Train the model on the dataset
history = model.fit(x_train, y_train, epochs=10, validation_split=0.1)

# Evaluate the model on the test dataset
test_loss, test_accuracy = model.evaluate(x_test, y_test)

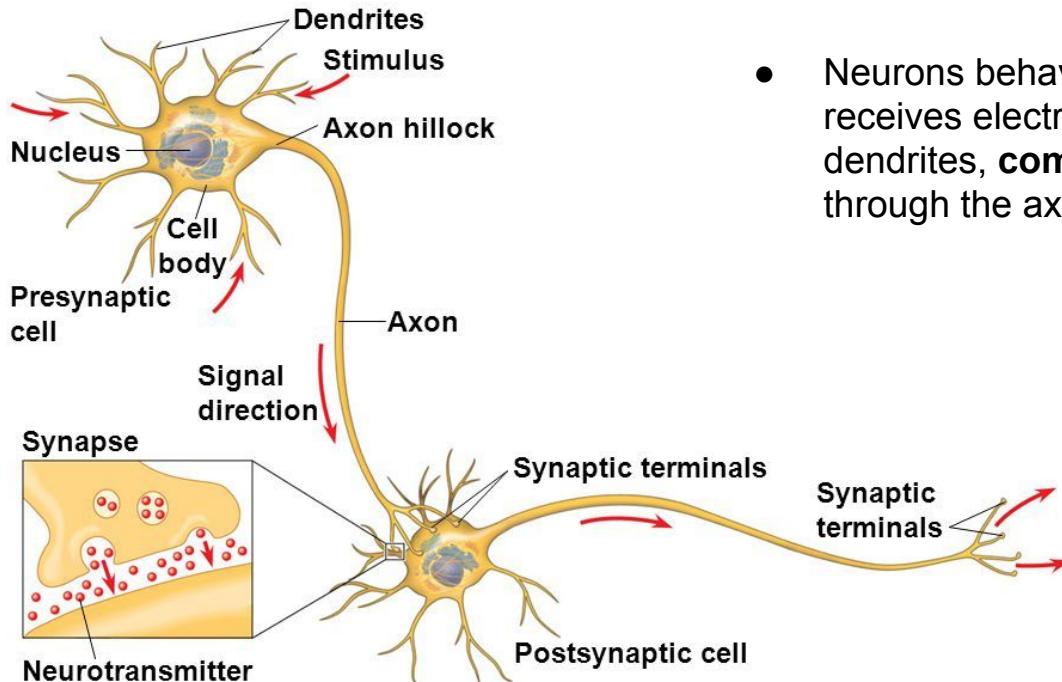
# Print the test loss and accuracy
print(f"Test loss: {test_loss}, Test accuracy: {test_accuracy}")

# Visualize the training history
plt.plot(history.history['accuracy'], label='Training accuracy')
plt.plot(history.history['val_accuracy'], label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
```

# 1.2

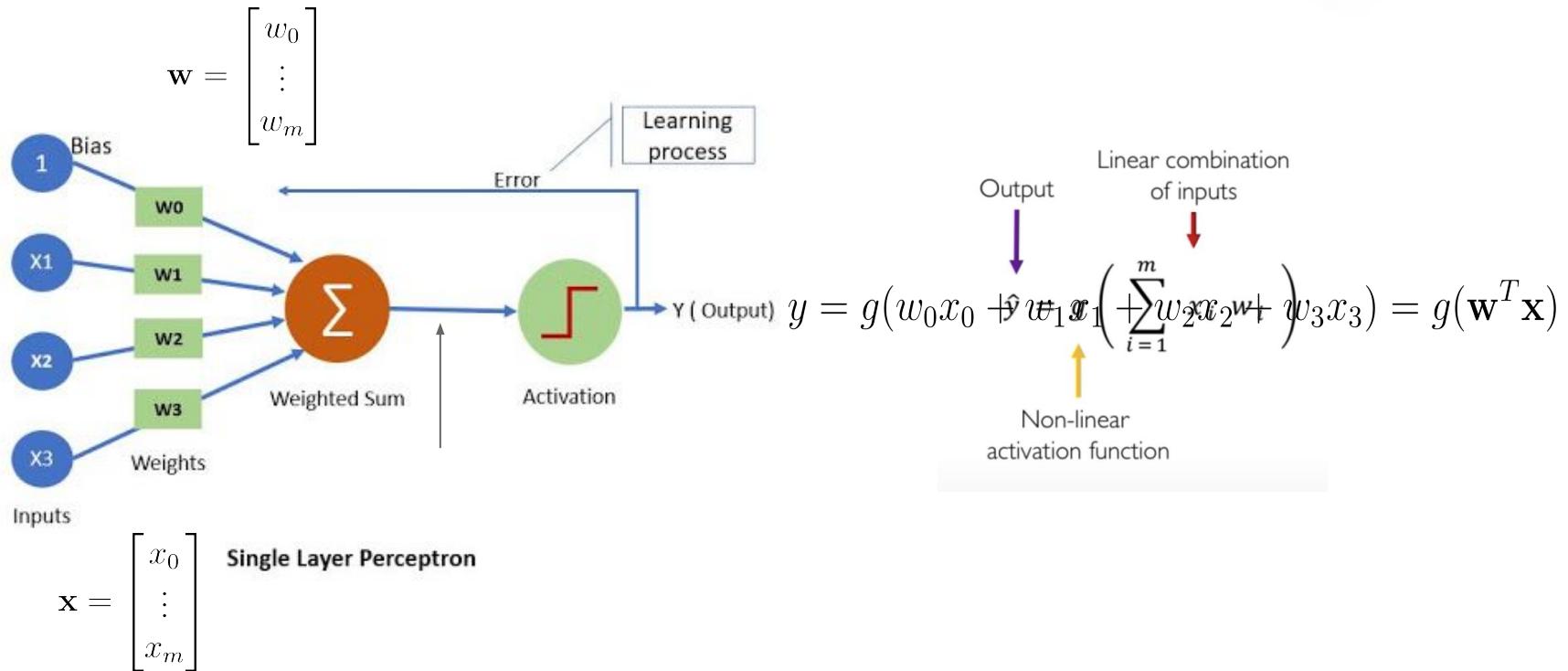
## Deep Learning Fundamentals

# Bio-inspired Model

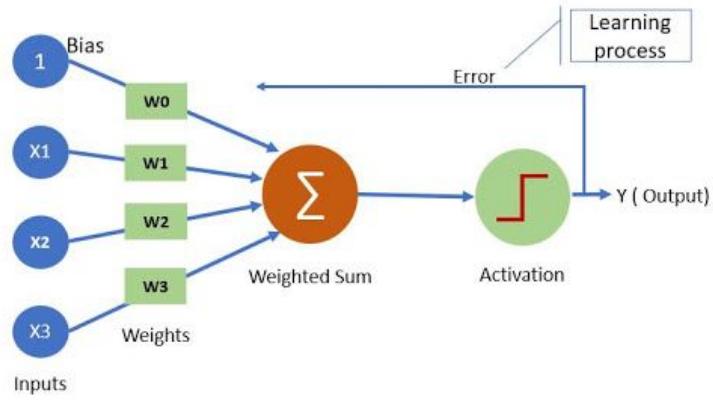


- Neurons behave as computing units. Each neuron receives electric input signals (called *spikes*) through the dendrites, **computes** an output with them and sends it through the axon to other neurons connected to it
- Axon-dendrite transmission is done in a mechanism called synapse.
- This process never changes, yet the brain is always learning

# Perceptron: Forward Propagation



# Perceptron: Logistic Regression



$$\hat{y} = \sigma(w_0 + w_1x_1 + w_2x_2 + w_3x_3) = \sigma(\mathbf{w}^T \mathbf{x})$$

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

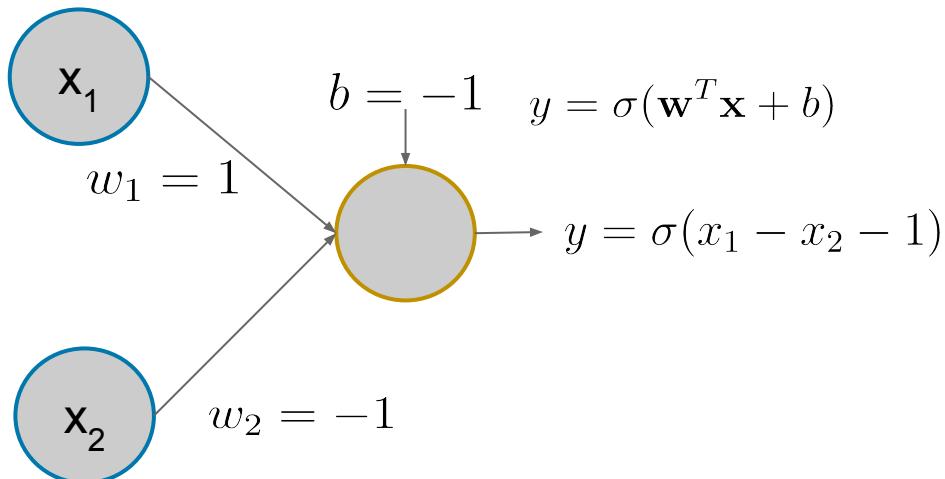
- Activation function:
  - sigmoid
- Loss function:
  - binary cross-entropy
- Gradients:

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

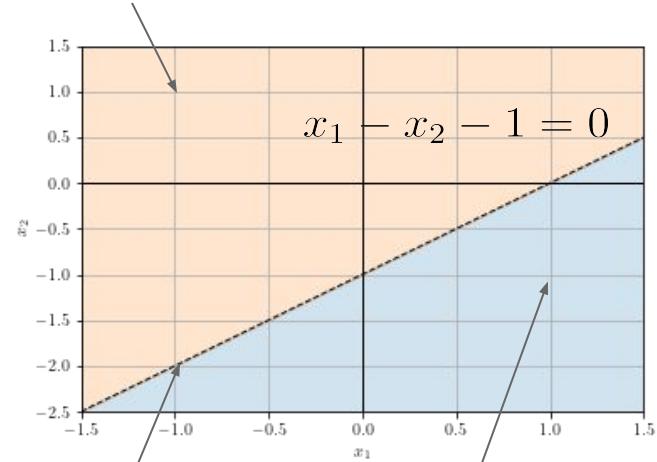
$$L_{\mathbf{w}}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^m [ -y_i \log (\sigma(\mathbf{w}^T \mathbf{x}_i)) - (1 - y_i) \log (1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) ]$$

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

## Perceptron: Example

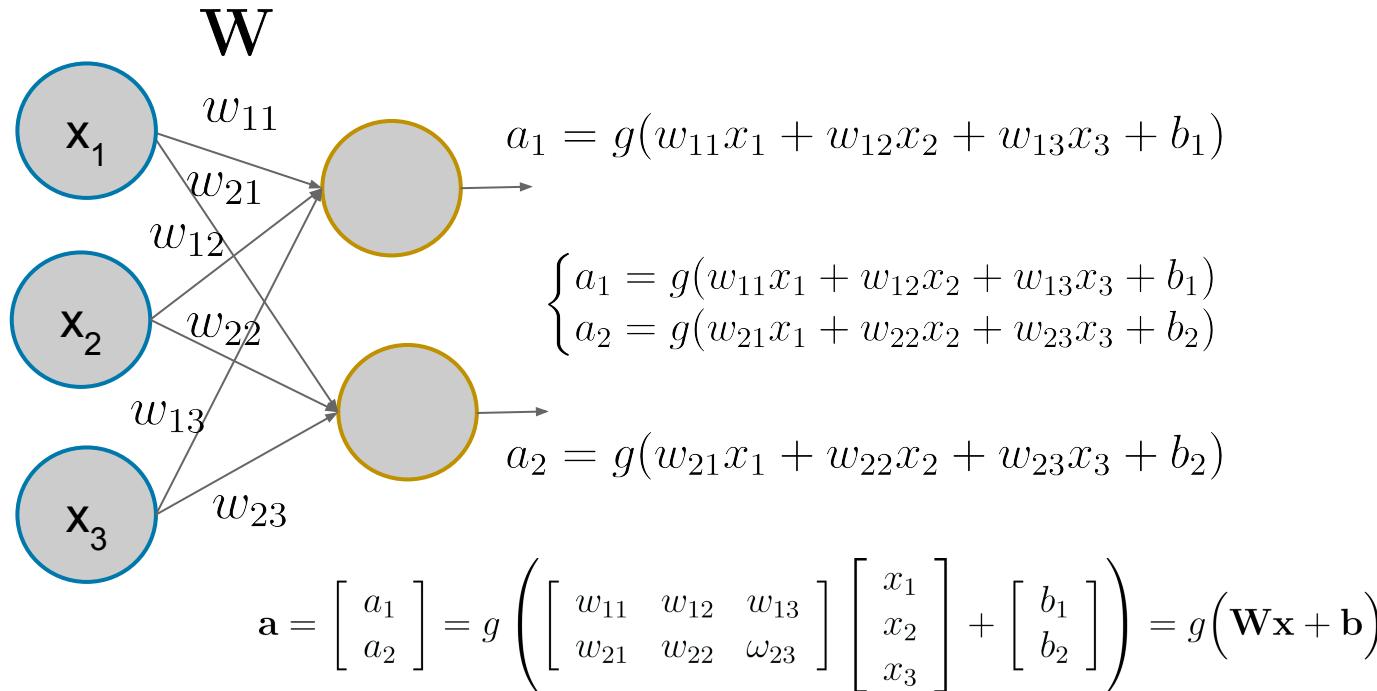


$$\sigma \left( \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right) = \sigma(-3) \approx 0.05$$

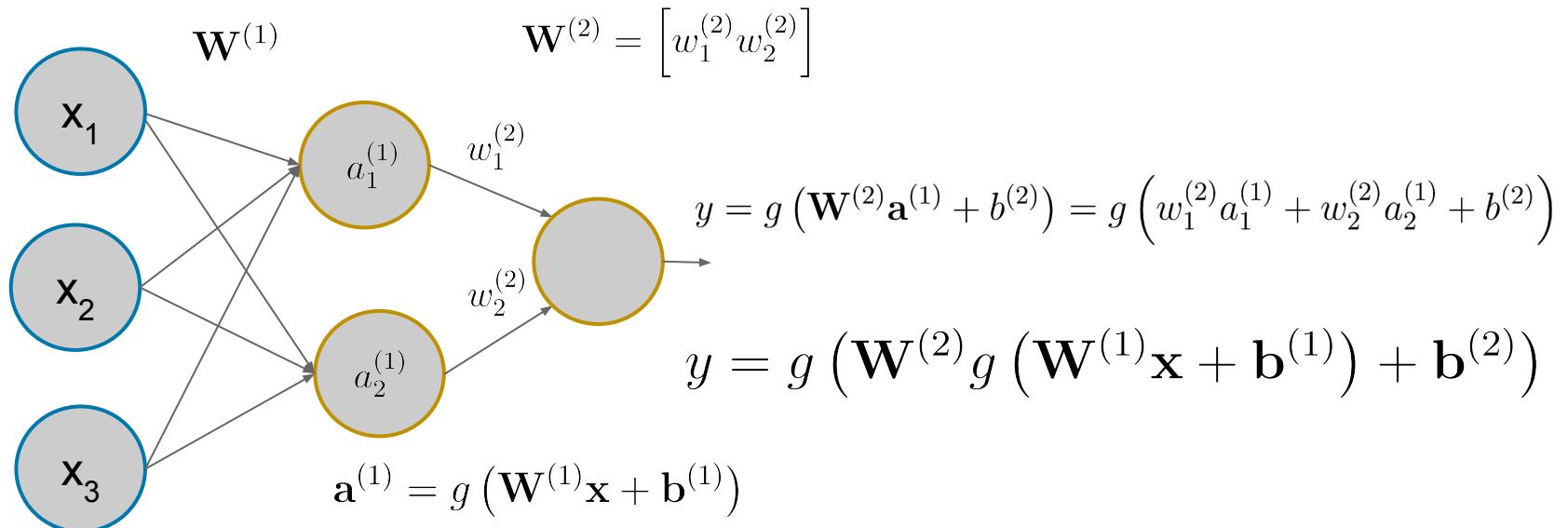


$$\sigma \left( \begin{bmatrix} -1 \\ -2 \end{bmatrix} \right) = \sigma(0) = \frac{1}{2} \quad \sigma \left( \begin{bmatrix} 1 \\ -1 \end{bmatrix} \right) = \sigma(1) \approx 0.75$$

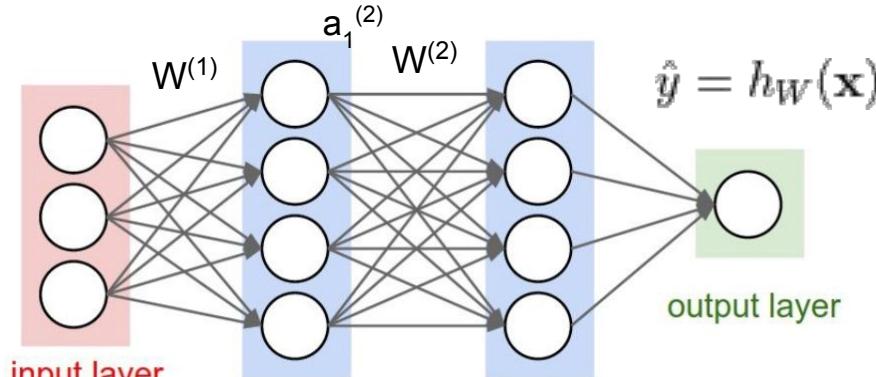
## Perceptron: Multi Output



# Single Layer Neural Network



# Multi-Layer Perceptron MLP



input layer

hidden layer 1    hidden layer 2

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

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

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

Dense, Feed forward ...

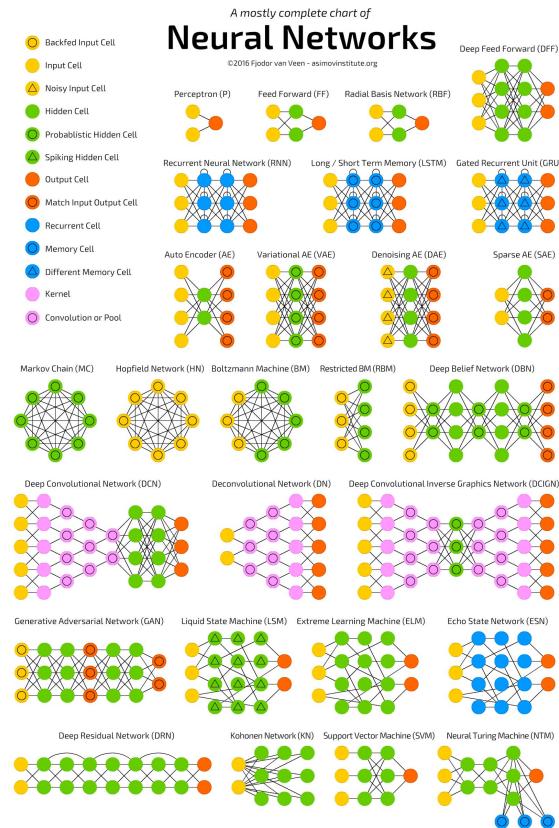
$a_i^{(j)}$  = activation of the i-th neuron of layer j

$W^{(j)}$  = 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)

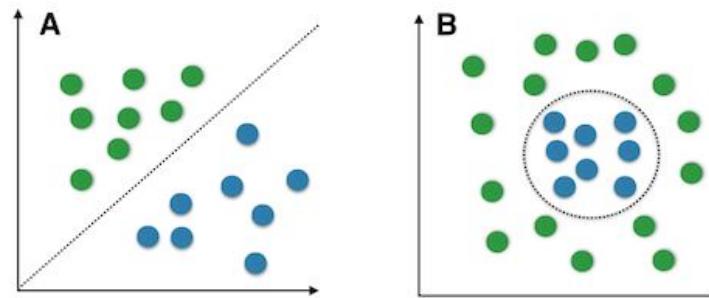
# NN Architectures





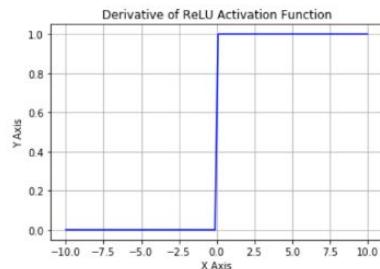
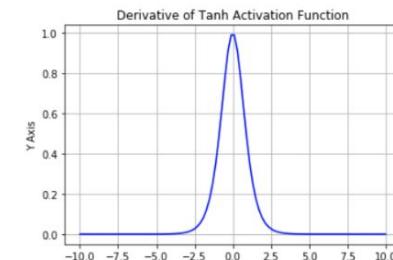
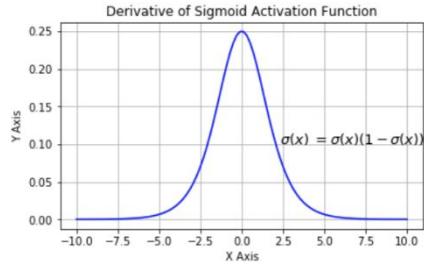
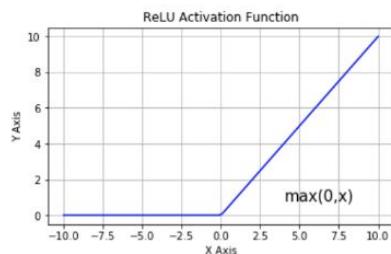
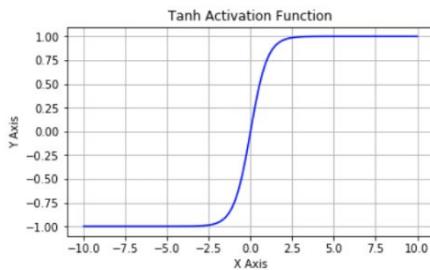
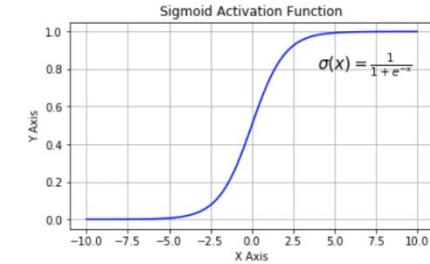
# Activation Functions I

Linear vs. nonlinear problems



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)`

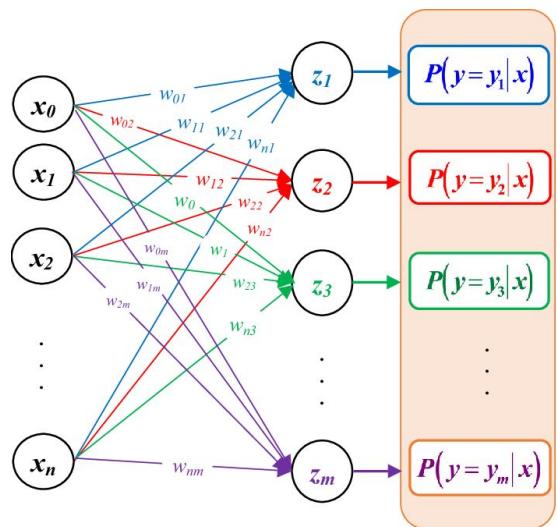
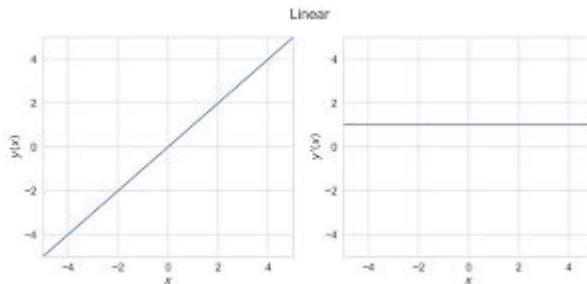
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- ReLU:  $[0, +\infty)$   $\text{ReLU}(x) = \max(0, x)$ 
  - **Output:** Positive regression.
  - Non-zero centered

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



# Activation Functions III



- Linear:  $(-\infty, +\infty)$ 
  - 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.

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

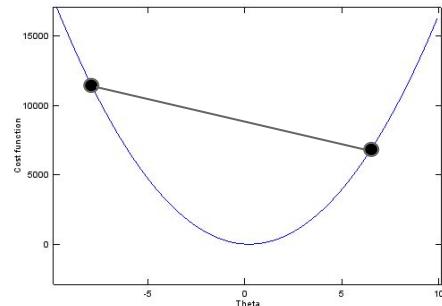
```
tf.keras.activations.softmax(x, axis=-1)
```

# 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



Cost function of one parameter

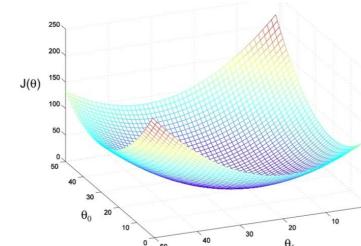
$$\mathcal{L}(f(\mathbf{x}^{(i)}; \mathbf{W}), \mathbf{y}^{(i)})$$

Prediction

Correct

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

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='categorical_crossentropy')
```

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



## Optimization: Training The Network

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^n \mathcal{L} (f \left( x^{(i)}; \mathbf{W} \right), \mathbf{y}^{(i)})$$

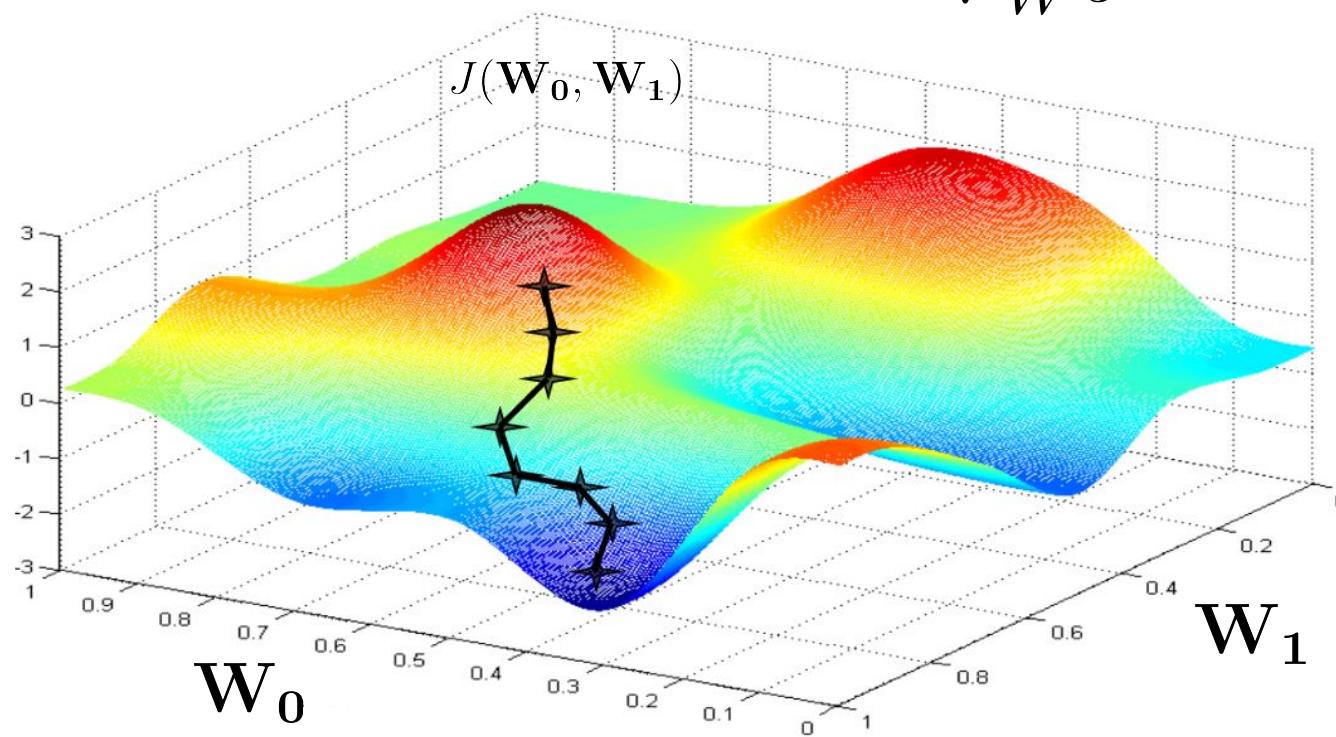
$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W})$$



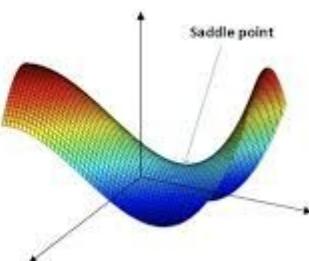
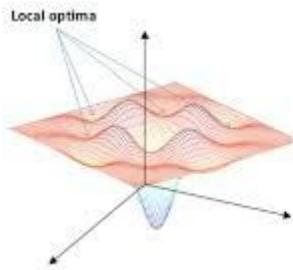
# Optimization

Follow gradient direction

$$\nabla_W J$$



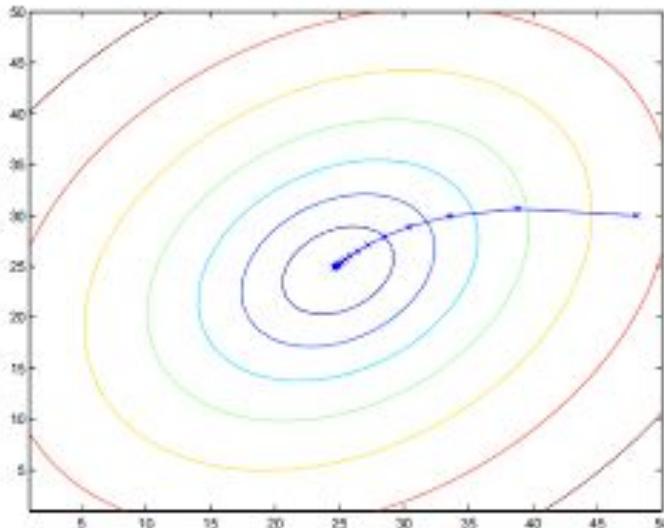
# Optimization: Challenges



- The difficulty of training neural networks is mainly attributed to their optimization part.
- Plateaus, saddle points and local minima grows exponentially with the dimension.
- Classical convex optimization algorithms don't perform well.



# Optimization: Gradient Descend



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

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

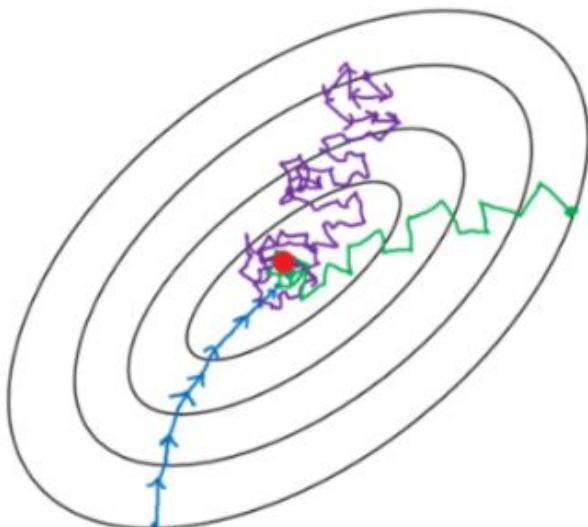
- In every step goes through the entire dataset

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

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

# Optimization: Mini-batch Gradient Descent

- Batch gradient descent (batch size = n)
- Mini-batch gradient Descent ( $1 < \text{batch size} < n$ )
- Stochastic gradient descent (batch size = 1)



- **Stochastic gradient descent:**
  - Estimate gradient with only one sample choose randomly
$$\nabla_{\mathbf{W}} J(\mathbf{W}) \approx \mathcal{L}(f(x^{(i)}; \mathbf{W}))$$
- **Mini-batch gradient descent:**
  - estimate gradient with some samples (m)
$$\nabla_{\mathbf{W}} J(\mathbf{W}) \approx \frac{1}{m} \sum_{i=1}^m \mathcal{L}(f(x^{(i)}; \mathbf{W}))$$
- **Batch size:** Length of the minibatch
- **Iteration:** Every time we update the weights
- **Epoch:** One pass over the whole training set.



# Optimization: Advances

$$\omega_t = \omega_{t-1} + \Delta\omega_t$$

$$\Delta\omega_t = -\frac{\mu}{\sqrt{v_t + \epsilon}} \cdot g_t$$

$$v_t = \rho \cdot v_{t-1} + (1 - \rho) \cdot g_t^2$$

$v_t$  exponential average of gradients

$\mu$  initial learning rate

$g_t$  gradient at time t at  $\omega$

Root Mean Square  
Propagation (RMSProp)

$$\omega_t = \omega_{t-1} + \Delta\omega_t$$

$$\Delta\omega_t = -\frac{\mu \cdot v_t}{\sqrt{s_t + \epsilon}} \cdot g_t$$

$$v_t = \beta_1 \cdot v_{t-1} + (1 - \beta_1) \cdot g_t$$

$$s_t = \beta_2 \cdot s_{t-1} + (1 - \beta_2) \cdot g_t^2$$

$v_t$  exponential average of gradients

$s_t$  exponential average of squares of gradients

$\mu$  initial learning rate

$\beta_1, \beta_2$  hyperparameters

$g_t$  gradient at time t at  $\omega$

Adaptive Moment  
Optimization (Adam)



# Optimization: Keras

- RMSprop
- SDG
- Adam

```
tf.keras.optimizers.RMSprop(learning_rate=0.1)
```

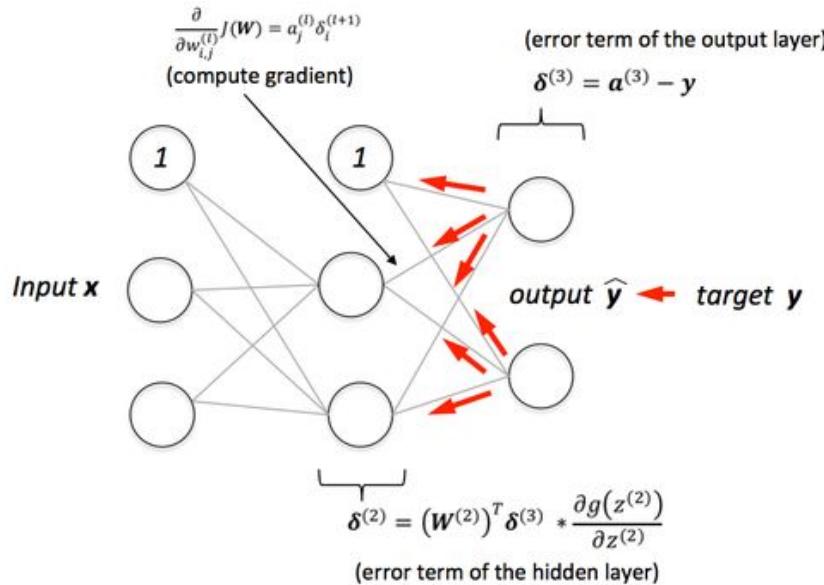
```
tf.keras.optimizers.SGD(learning_rate=0.1)
```

```
tf.keras.optimizers.Adam(learning_rate=0.1)
```

```
from tensorflow import keras
model = keras.Sequential(
    [
        keras.layers.Dense(
            256, activation="relu",
            input_shape=input_shape
        ),
        keras.layers.Dense(1, activation="sigmoid"),
    ]
)
model.compile(
    optimizer=keras.optimizers.Adam(1e-2),
    loss="binary_crossentropy"
)
```



# Backpropagation

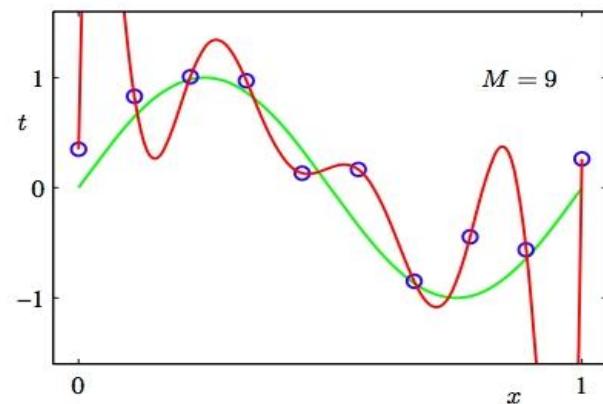
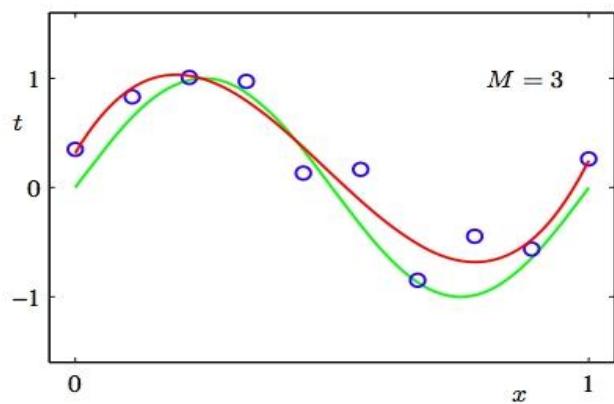
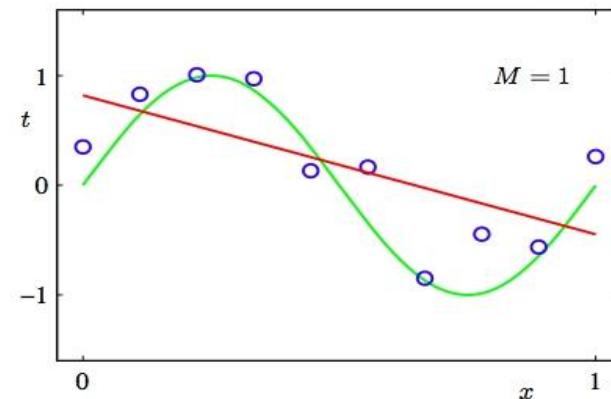
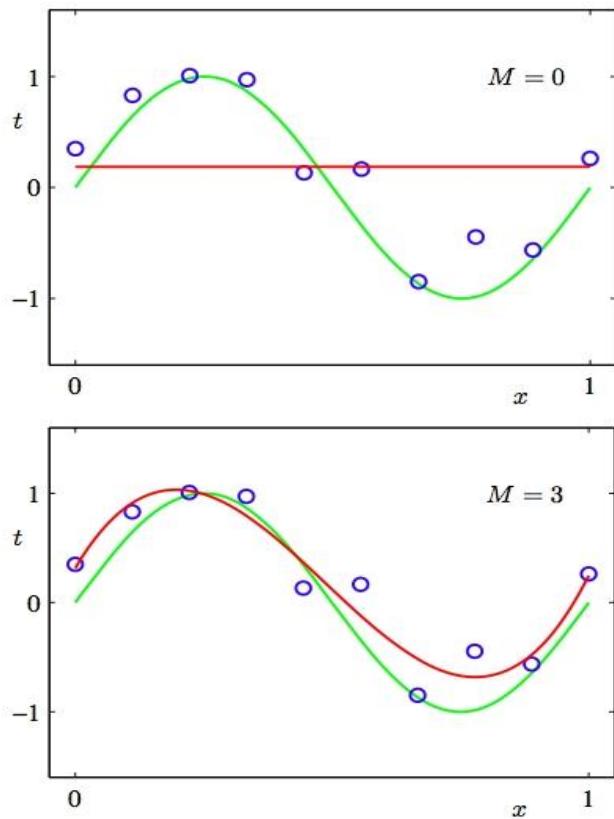


Algorithm to compute the partial derivatives of the cost function with respect to each parameter.

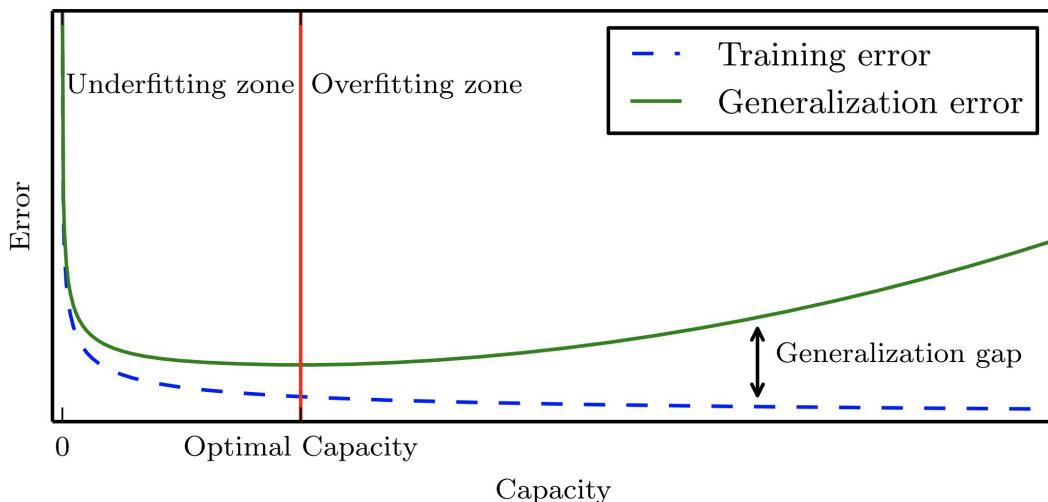
- **Intuition:** compute the contribution of each neuron to the final error, and change its weights accordingly
  - Compute the contribution (*deltas*) of each neuron to the error of each example separately, and then accumulate over all examples to obtain the total contribution of each neuron to the total error.

Example: we first apply forward propagation to compute every  $a_j^{(l)}$  and the output of the network  $h_B(x)$

# Regularization: Overfitting



# Regularization: Overfitting



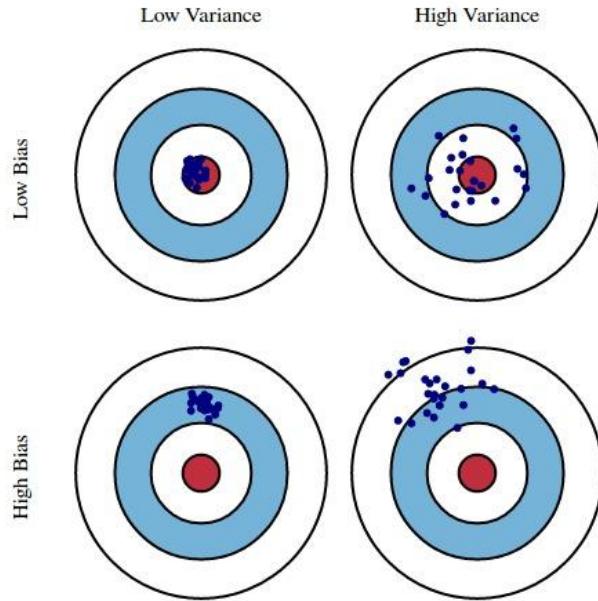
**Generalization:** The ability to perform well on previously unobserved inputs.

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



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

$$\text{Var}_D [\hat{f}(x; D)] = \mathbb{E}_D [(\mathbb{E}_D [\hat{f}(x; D)] - \hat{f}(x; D))^2]$$

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

Trade off :

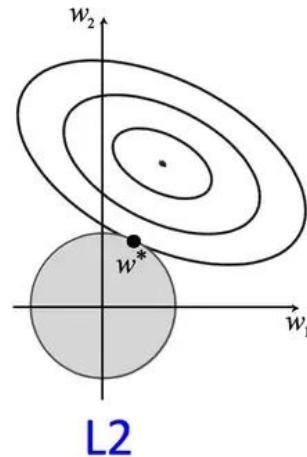
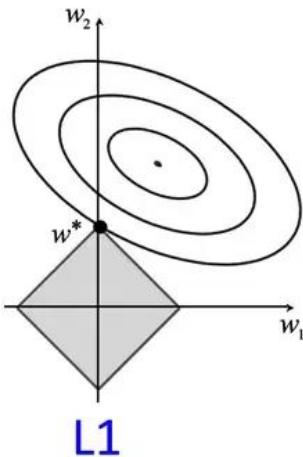
- **Overfitting :**
  - Low bias, High variance.
  - Complex, flexible model.
  - Memorizes training data, captures noise.
  - Gap between training and test error is big.
- **Underfitting :**
  - High bias, Low Variance
  - Simple, rigid model.
  - Fails to capture underlying patterns in the data.
  - 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.
- Central problem in machine learning.
- Lots of types:
  - Adding restrictions on the parameter values.
  - Adding extra terms in the objective function.
- In deep learning, trading increased bias for reduced variance.
- Ensemble methods.



# Regularization: L2 y L1



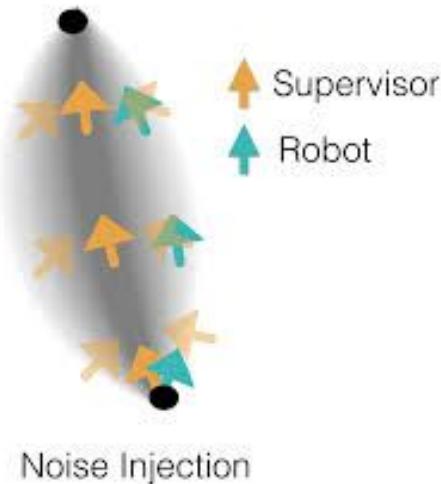
- Add a penalty to the cost function:

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

- 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.

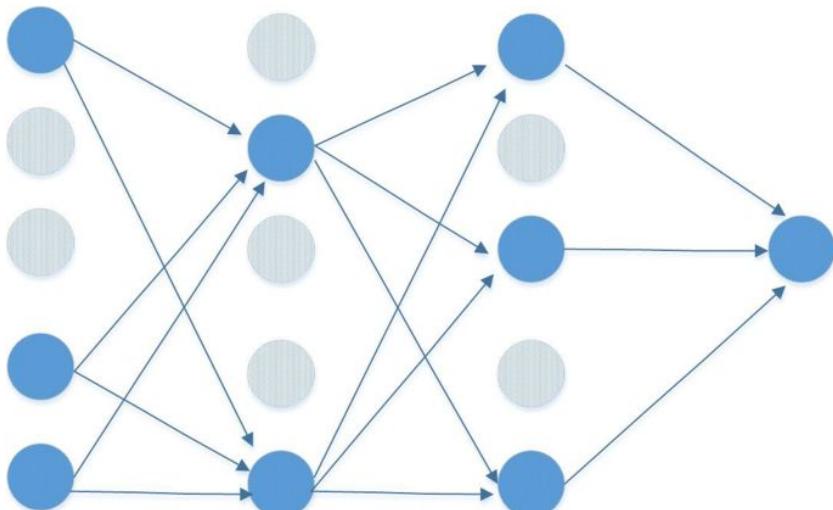


# Regularization: Noise injection



- 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

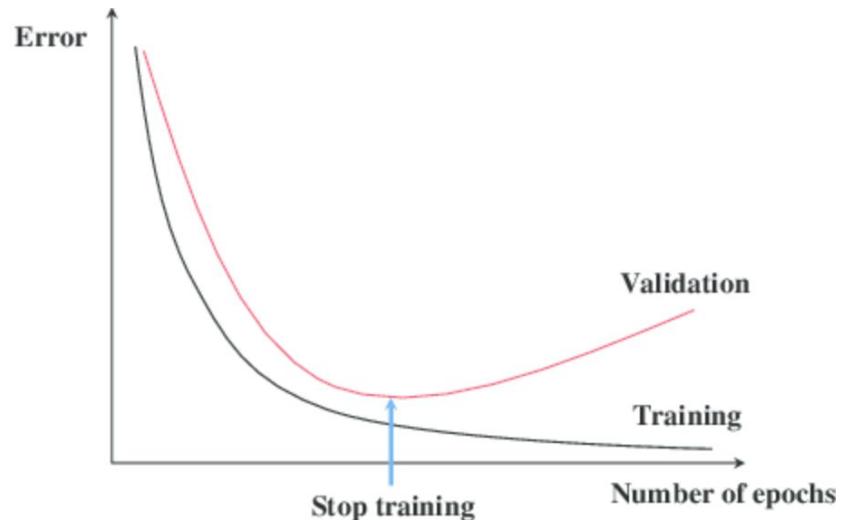


- During training, randomly set some activations to 0 with probability  $p$ .
- Not in prediction.

```
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

Stop training before overfit

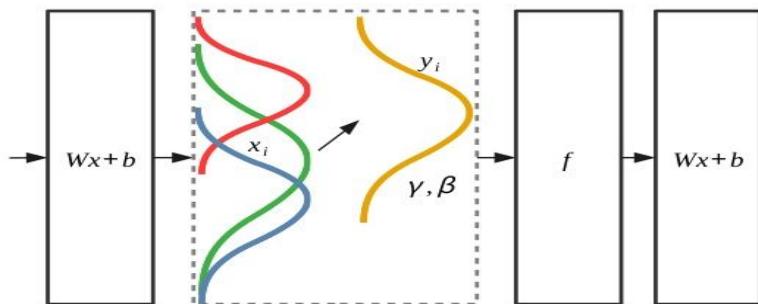


# Regularization: batch normalization

- Normalizing input activations across mini-batches to improve training stability.
- Accelerate the training.

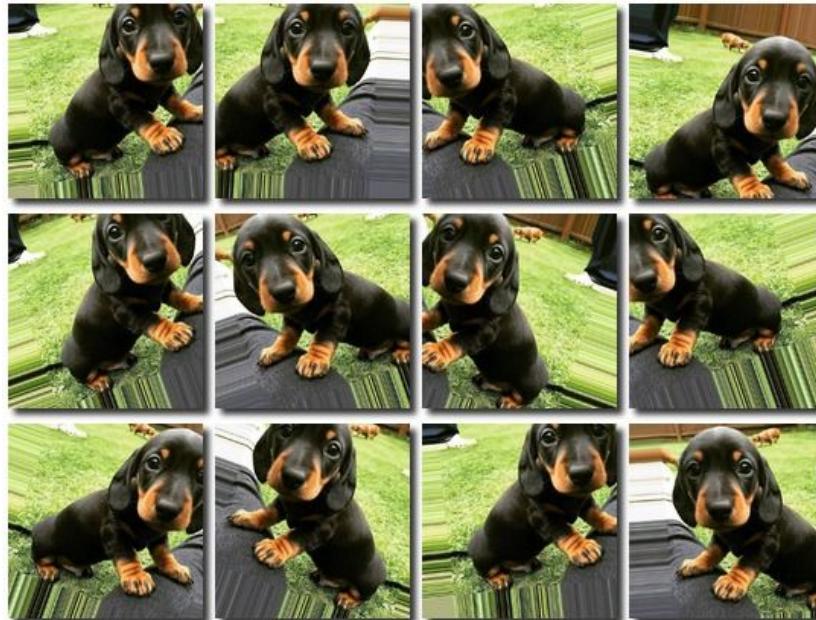
## Batch normalization

Ensure the output statistics of a layer are fixed.



```
from keras.layers.normalization import  
BatchNormalization  
model = Sequential()  
model.add(Dense(64, input_shape=))  
model.add(BatchNormalization())  
model.add(Activation('tanh'))  
model.add(Dropout(0.5))
```

# Data Augmentation



# 1.3

## Exercices





UNIVERSIDAD  
COMPLUTENSE  
DE MADRID

