Neural Networks

Bryan Cardenas
Robert Jan Schlimbach
Caspar van leeuwen

Background Prerequisites

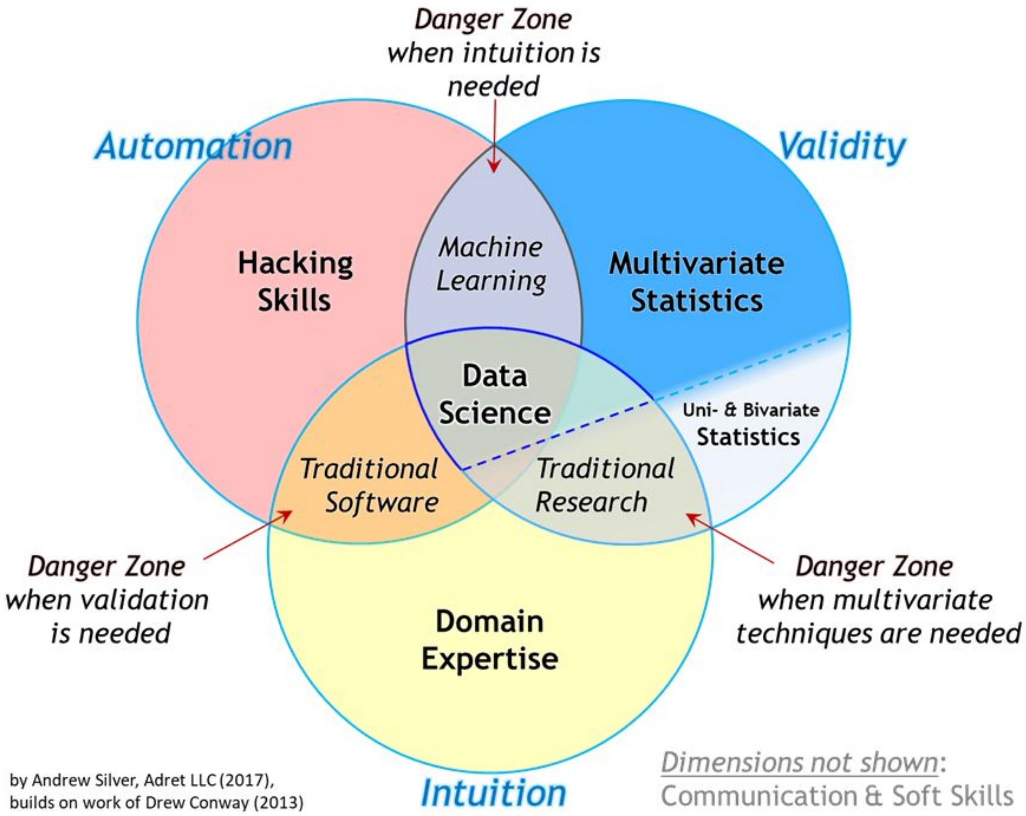
Programming

R / Python

Statistics, Calculus

Machine Learning / Deep Learning

Parallel Computing



Topics List

1. General Introduction to ML and DL, basic principles
2. Algorithms and Models
3. Convolutional Neural Networks
4. Generative Models
5. Recurrent Neural Networks
6. (brief) Reinforcement Learning

Course Plan until lunch

- 9:00-9:45 Introduction ML & DL, basic principles
- 9:45-10:00 Introduction to PyTorch
- 10:05-10:50 Hands-on 1: fully connected network
- 10:50-11:00 Recap hands-on
- 11:00-11:15 Coffee break
- 11:15-11:45 CNN theory
- 11:45-12:30 Hands-on 2: CNN
- 12:30-13:30 Lunch break

Course Plan After Lunch

- 12:30-13:30 Lunch break
- 13:30-13:45 Recap hands-on
- 13:45-14:30 Hands-on 3: CNN, Fine-tuning
- 14:30-14:45 Recap hands-on
- 14:45-15:00 Coffee break
- 15:00-15:45 VAE theory
- 15:45-16:30 Hands-on 4: VAE, ‘demo’ notebook
- 16:30-17:00 Questions & wrap-up

What is Machine Learning?

It is **NOT:**

Mimicking human intelligence

Robotics

Deep Learning

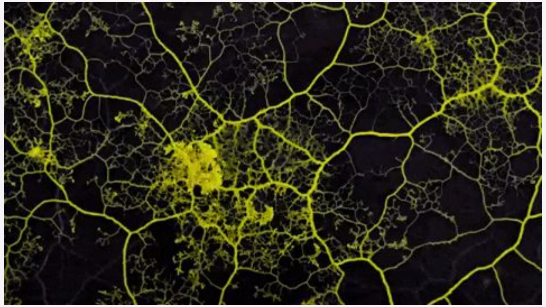
ML *is the study of computer algorithms that can improve automatically through experience and by the use of data.*[1] *It is seen as a part of artificial intelligence.*

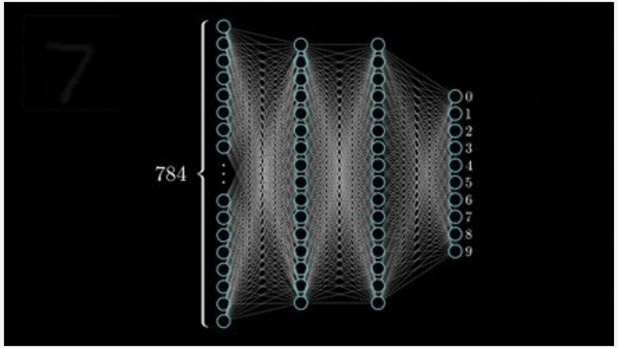
- *wikipedia*

AI vs ML vs DL

Artificial Intelligence

Having computers to exert
Intelligent behaviour





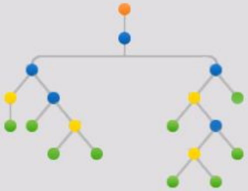
AI vs ML vs DL

Artificial Intelligence

Having computers to exert
Intelligent behaviour

Machine Learning

Perform tasks without
Explicitly programmed
from data



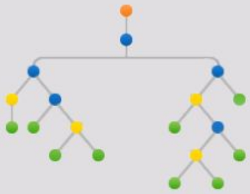
AI vs ML vs DL

Artificial Intelligence

Having computers to exert
Intelligent behaviour

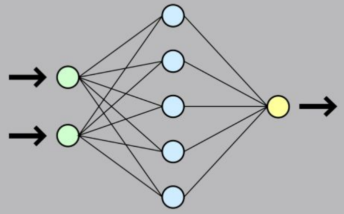
Machine Learning

Perform tasks without
Explicitly programmed
from data



Deep Learning

Use (deep) neural networks



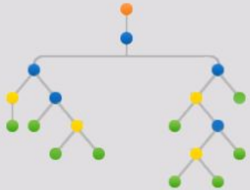
AI vs ML vs DL

Artificial Intelligence

Having computers to exert
Intelligent behaviour

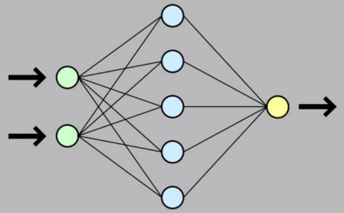
Machine Learning

Perform tasks without
Explicitly programmed
from data



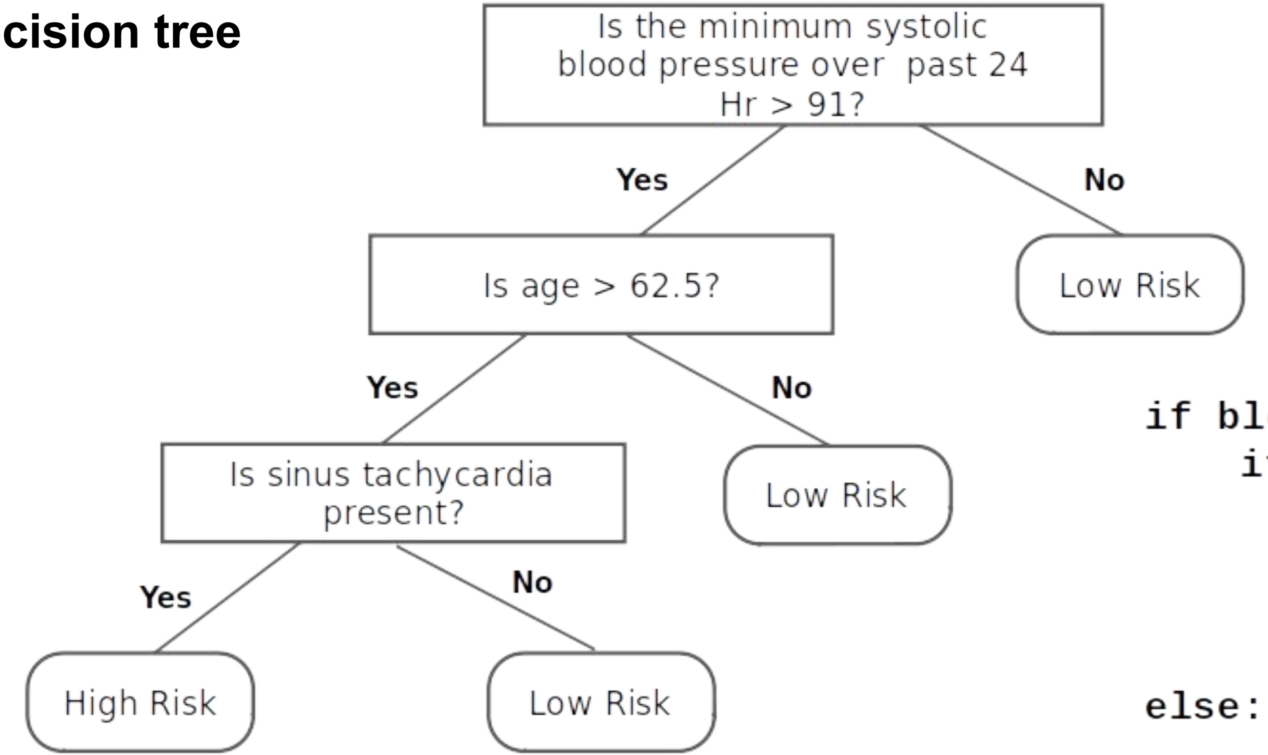
Deep Learning

Use (deep) neural networks



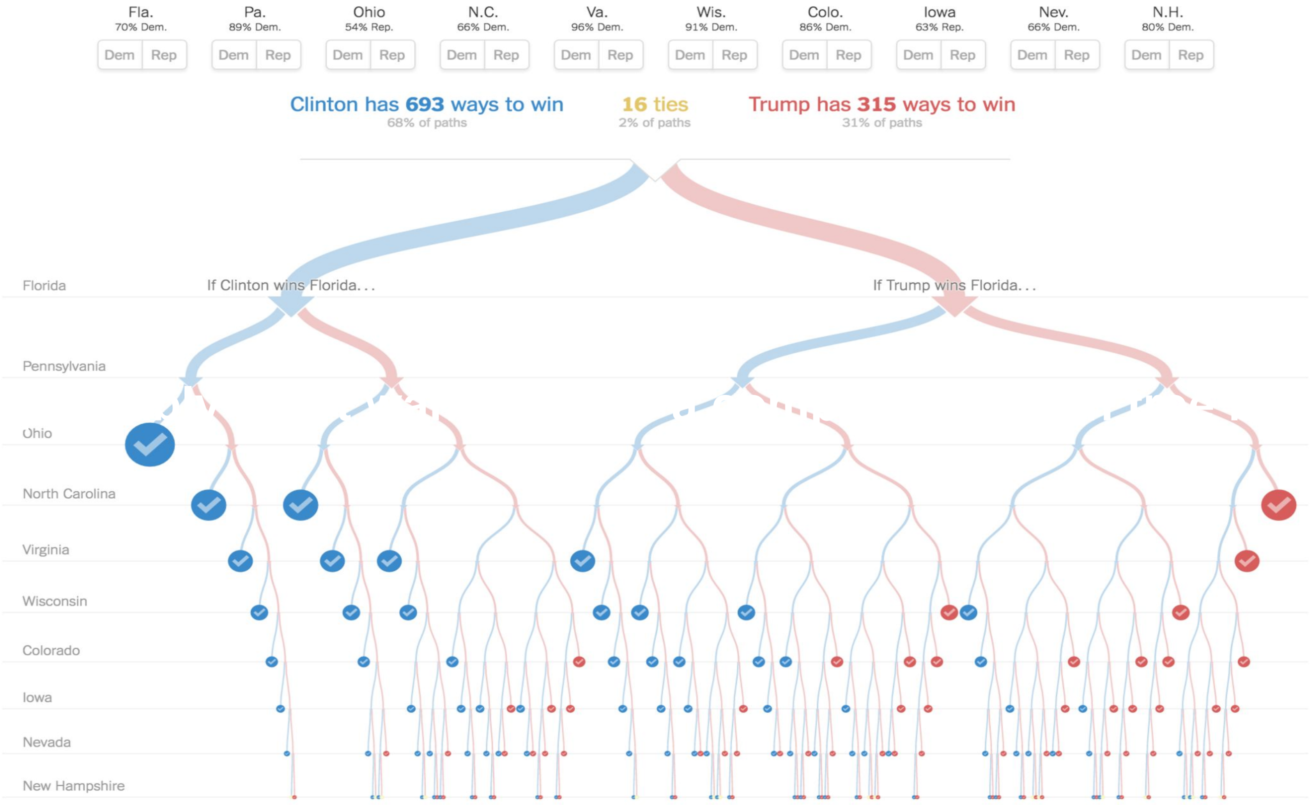
Why Machine Learning?

Think of a simple **decision tree**

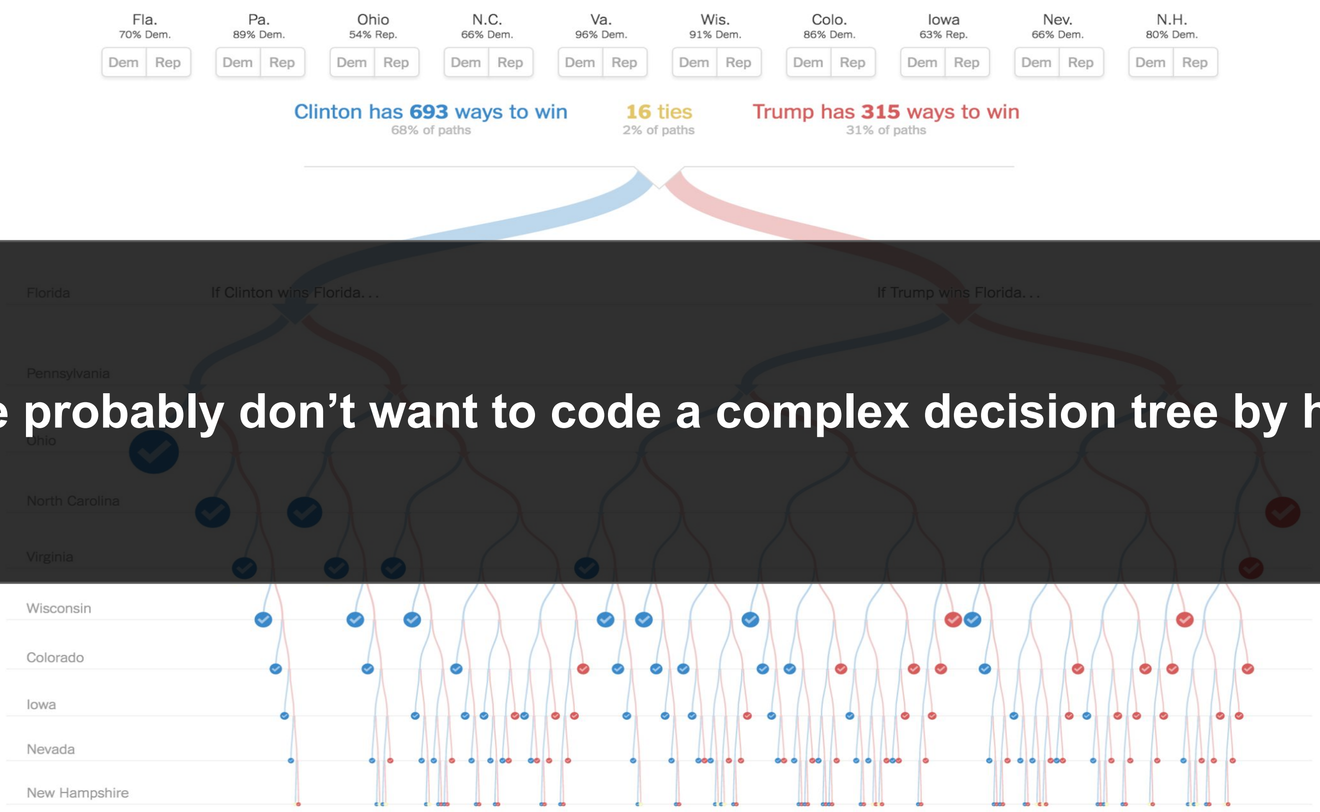



```
if blood_pressure > 91:
    if age > 62.5:
        if sinus_tach:
            …
        else:
            …
else:
    …
```

Why Machine Learning?



Why Machine Learning?



We probably don't want to code a complex decision tree by hand

Why Machine Learning?

What is a **dog**?

Why Machine Learning?

What is a **dog**?

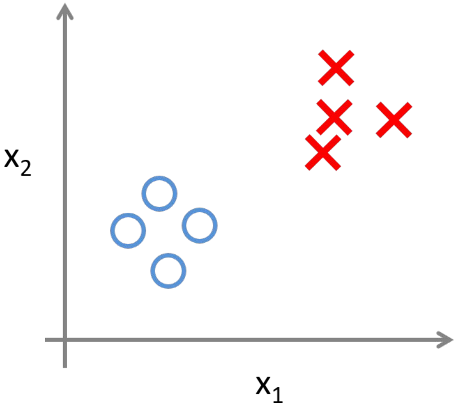
Driven by Data

Categories of Machine Learning

Supervised

Learn from labels

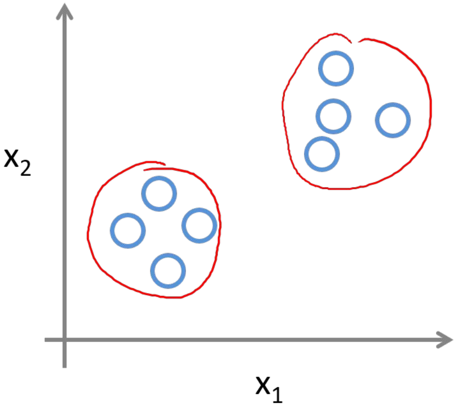
Regression, Classification



Unsupervised

Detect patterns in the data

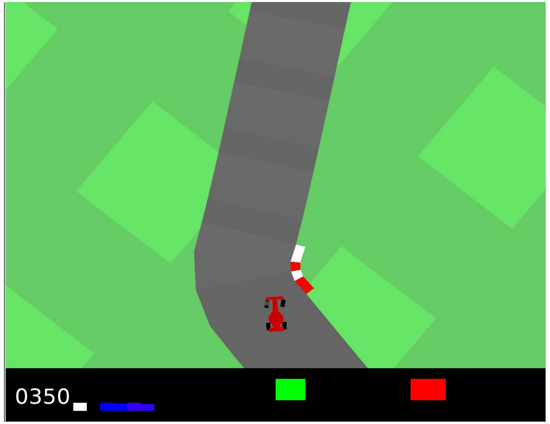
Clustering, Dimensionality reduction



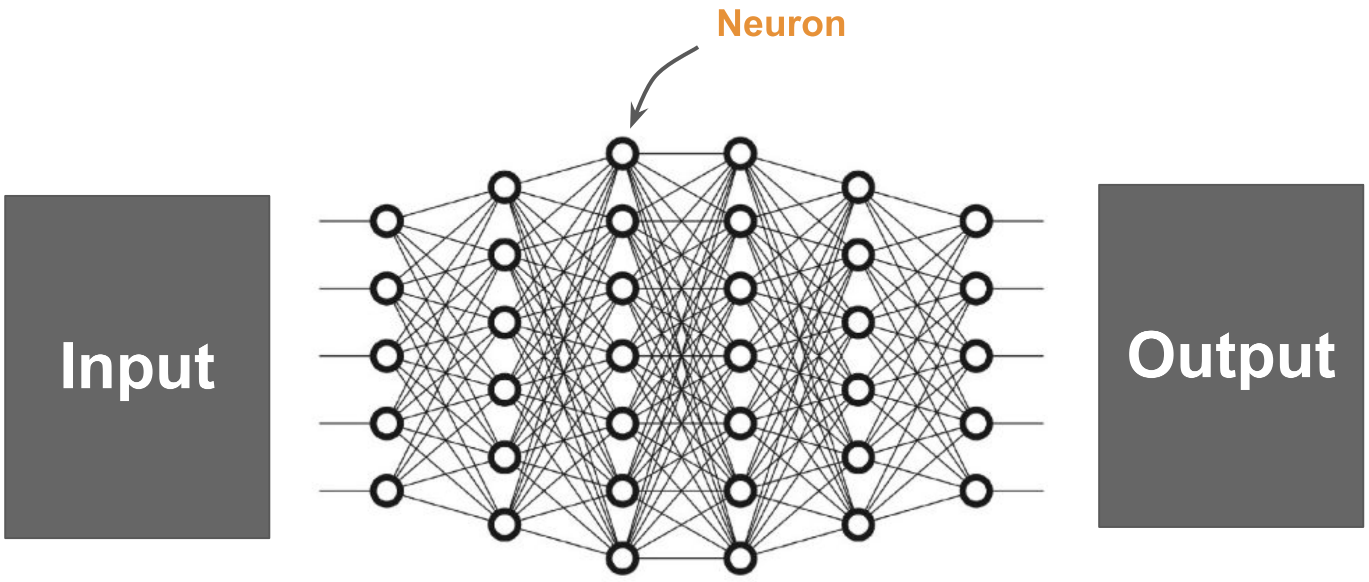
Reinforcement

Learn from mistakes

Control, Gaming

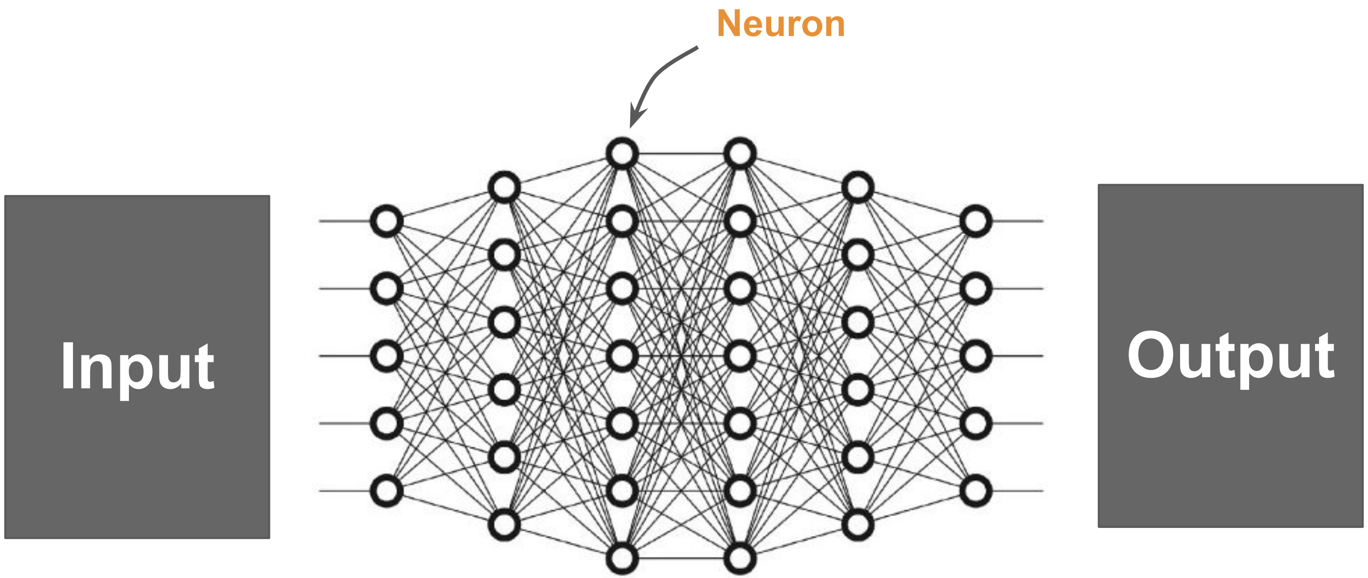


Neural Networks



Use a (deep) neural network to **approximate** an unknown **function**

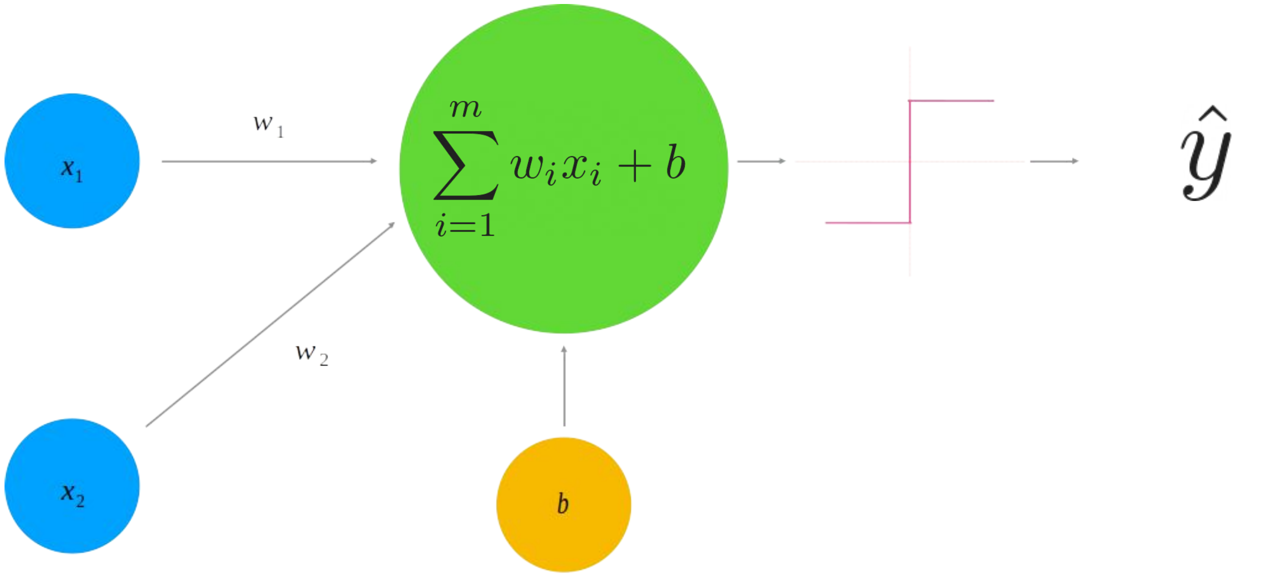
Neural Networks



$$\hat{y} = f_{NN}(x_1, x_2, \ldots, x_n)$$

Use a (deep) neural network to **approximate** an unknown **function**

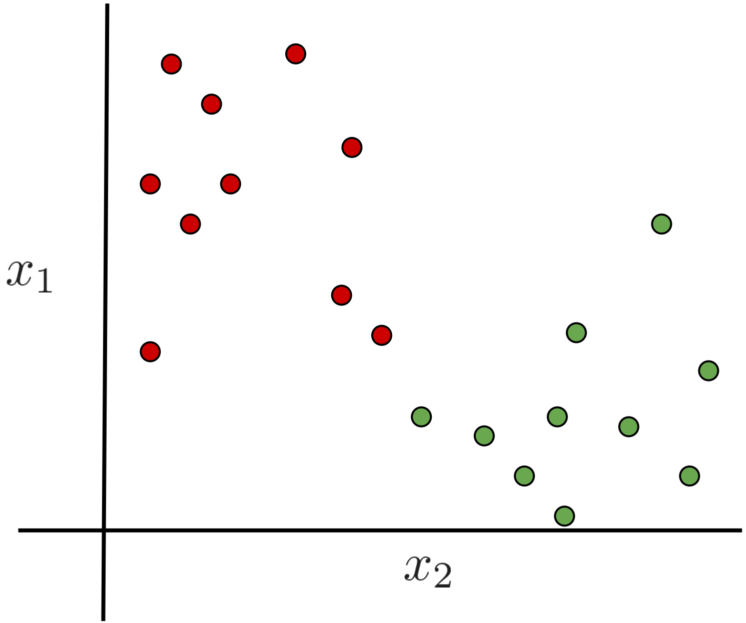
Anatomy of Neural Networks



Inputs weights Summation + bias Activation Function Output

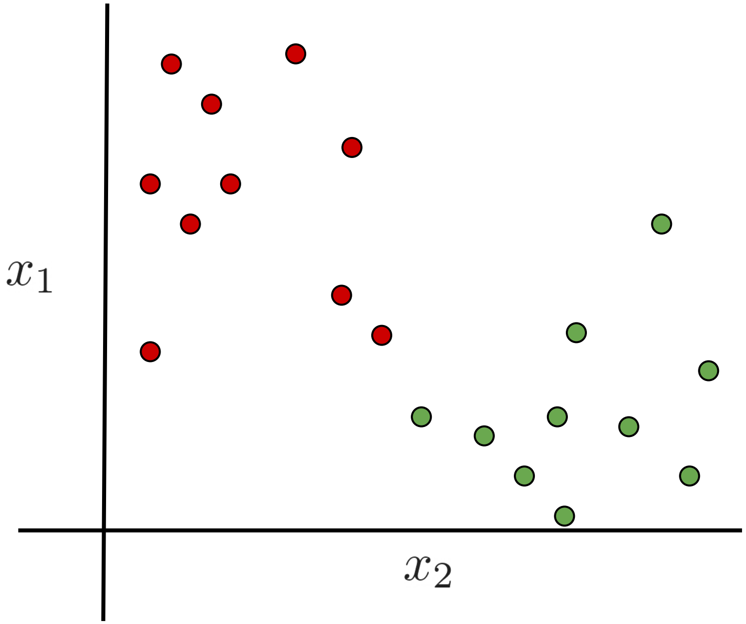
Perceptron & Activation

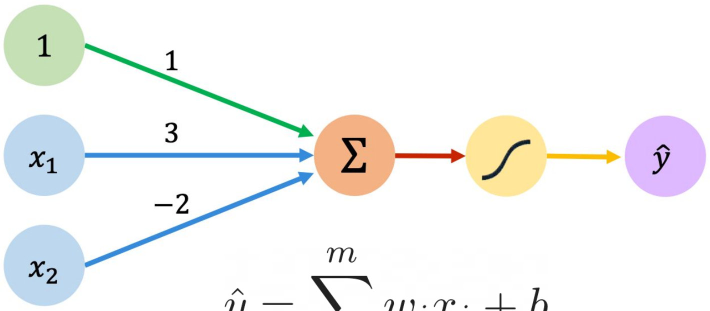
Binary Classification Task



Perceptron & Activation

Binary Classification Task

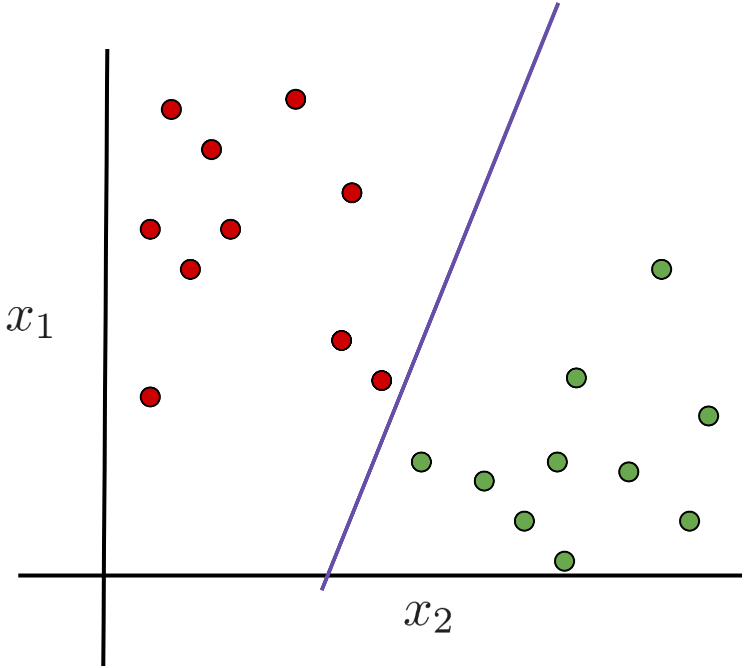


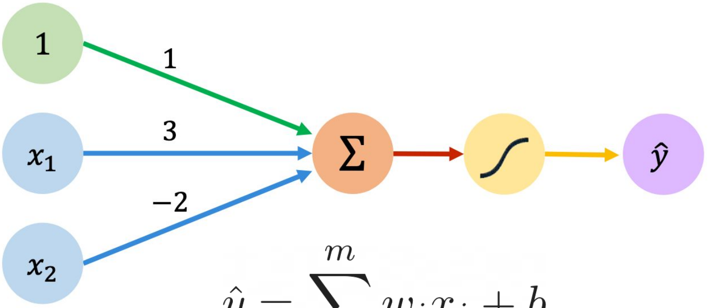


$$\hat{y} = \sum_{i=1}^{m} w_i x_i + b$$

Perceptron & Activation

Binary Classification Task

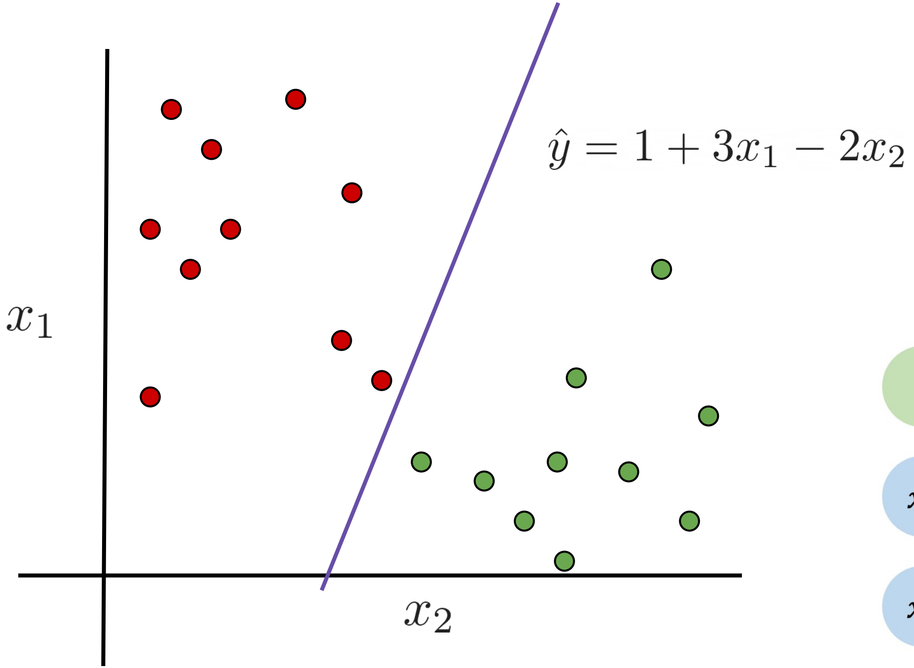


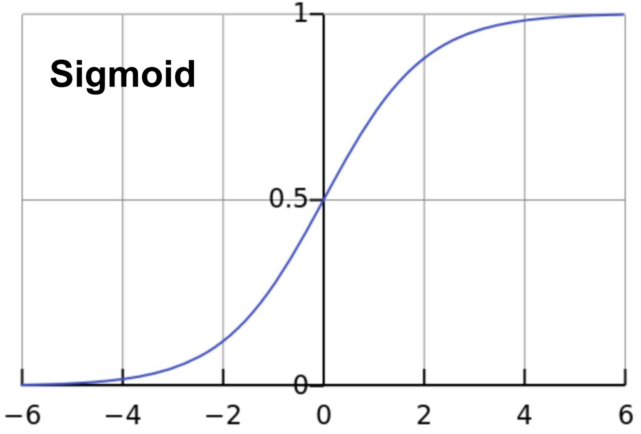


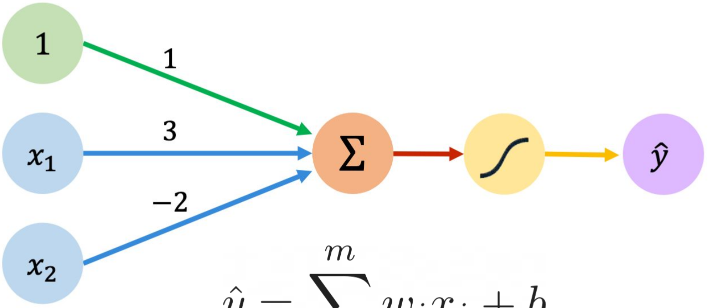
$$\hat{y} = \sum_{i=1}^{m} w_i x_i + b$$

Perceptron & Activation

Binary Classification Task



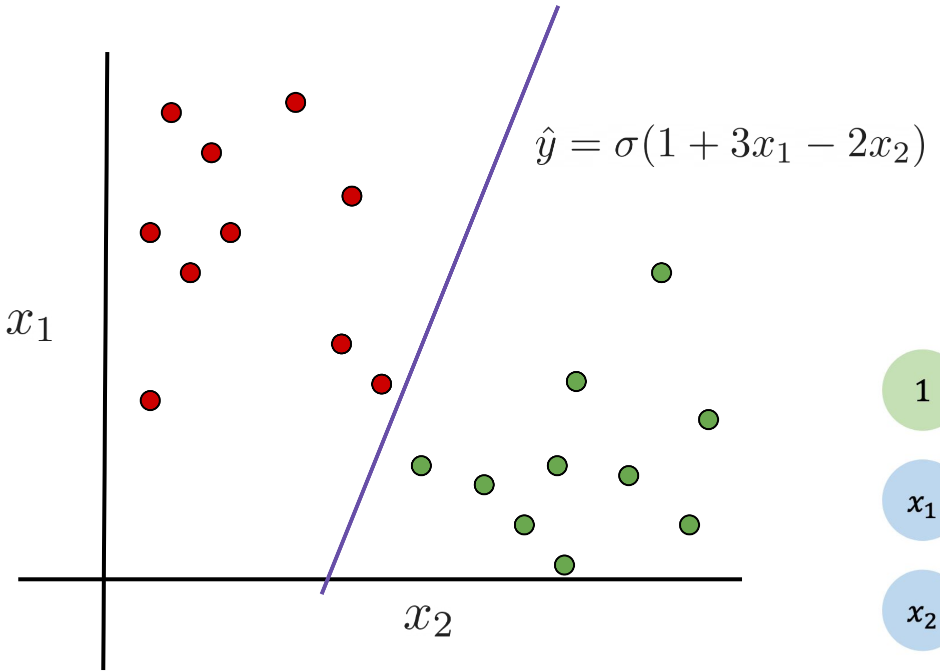


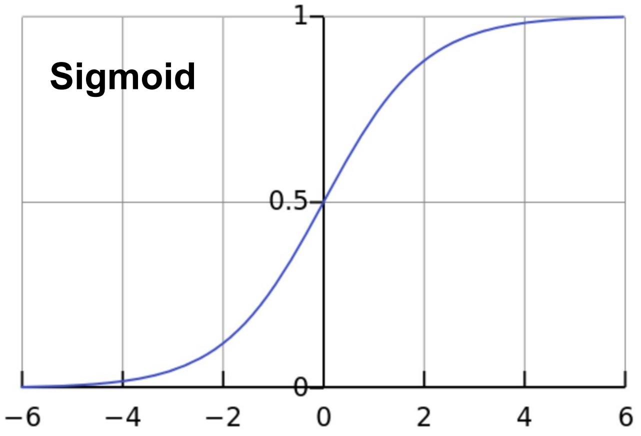


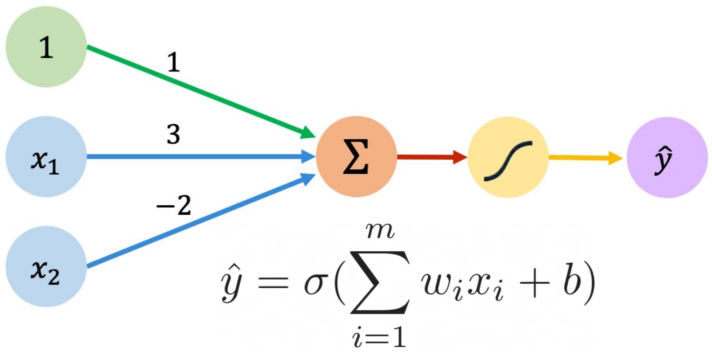
$$\hat{y} = \sum_{i=1}^{m} w_i x_i + b$$

Perceptron & Activation

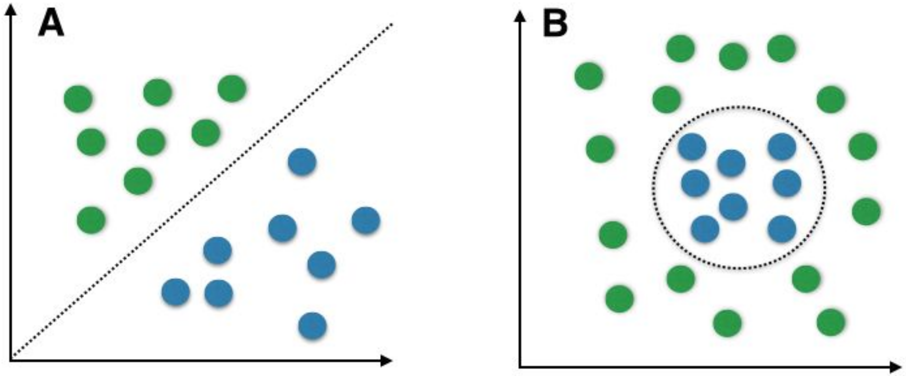
Binary Classification Task

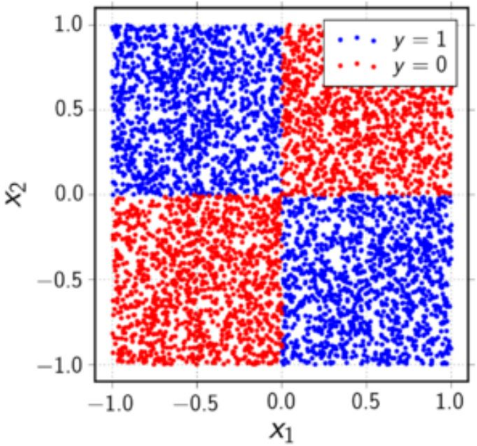






Limitation of Linear Single-Layer Classifiers



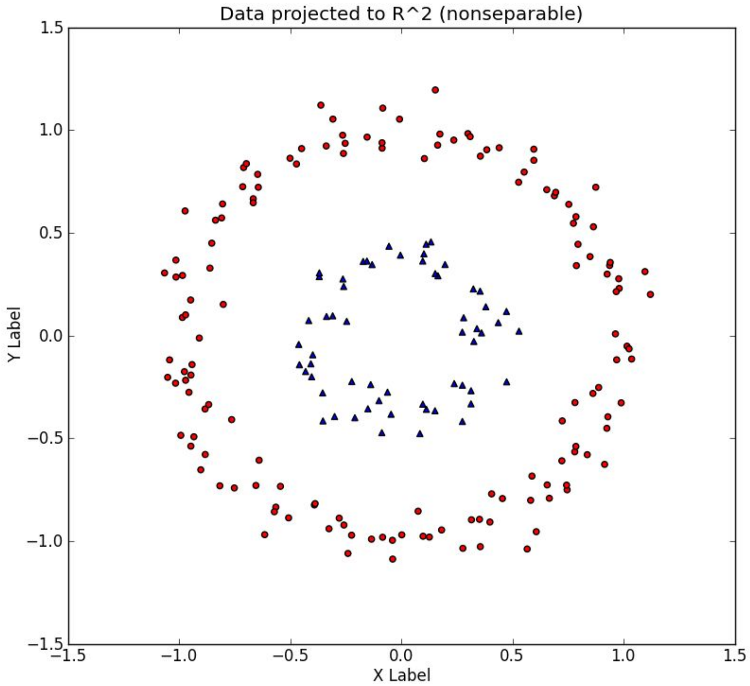


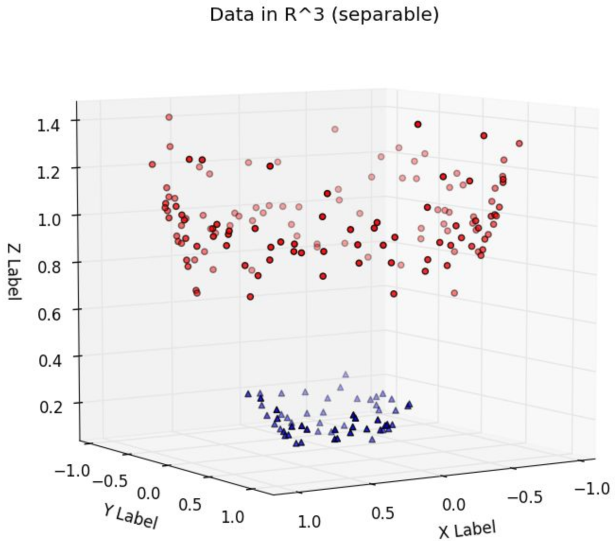
XOR Problem

Possible solutions:

- Add more layers (deep learning)
- Map into another (higher dimensional) space

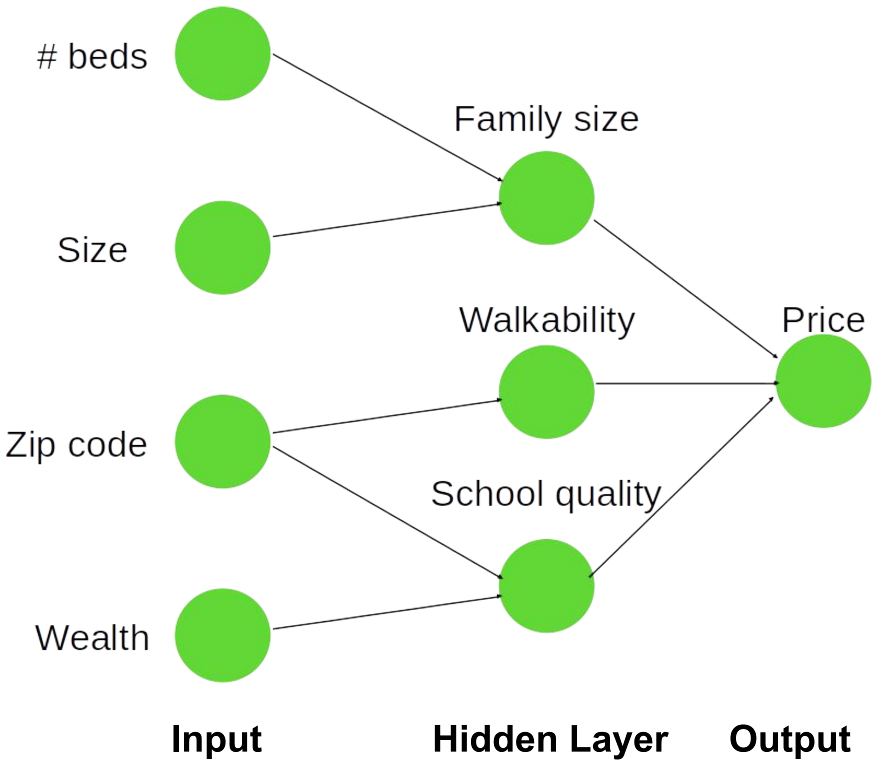
Kernel Trick





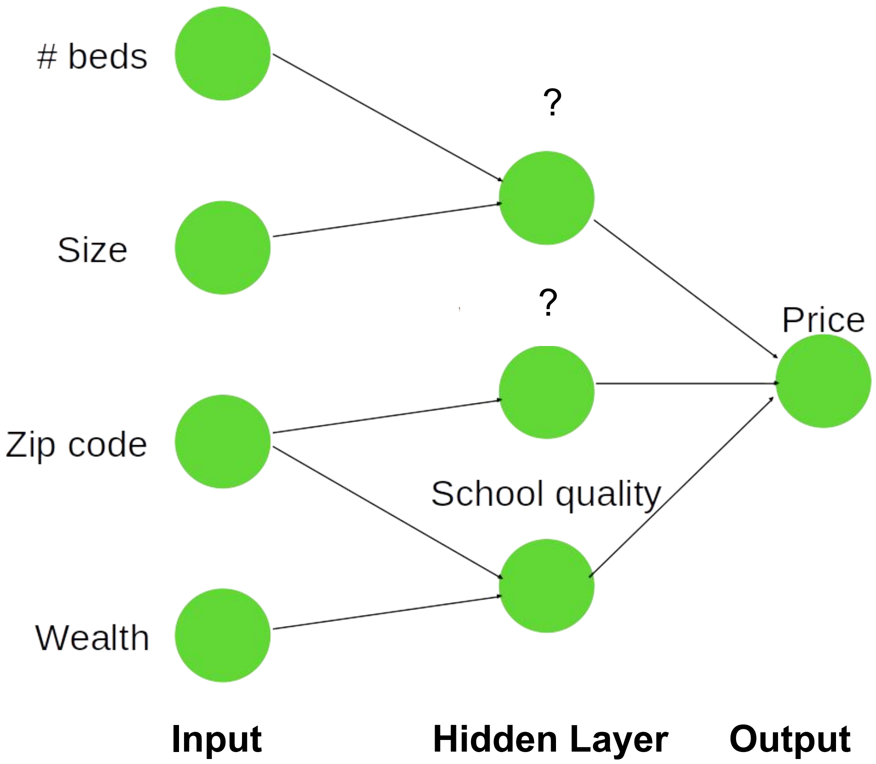
Adding Layers

House price prediction



Adding Layers

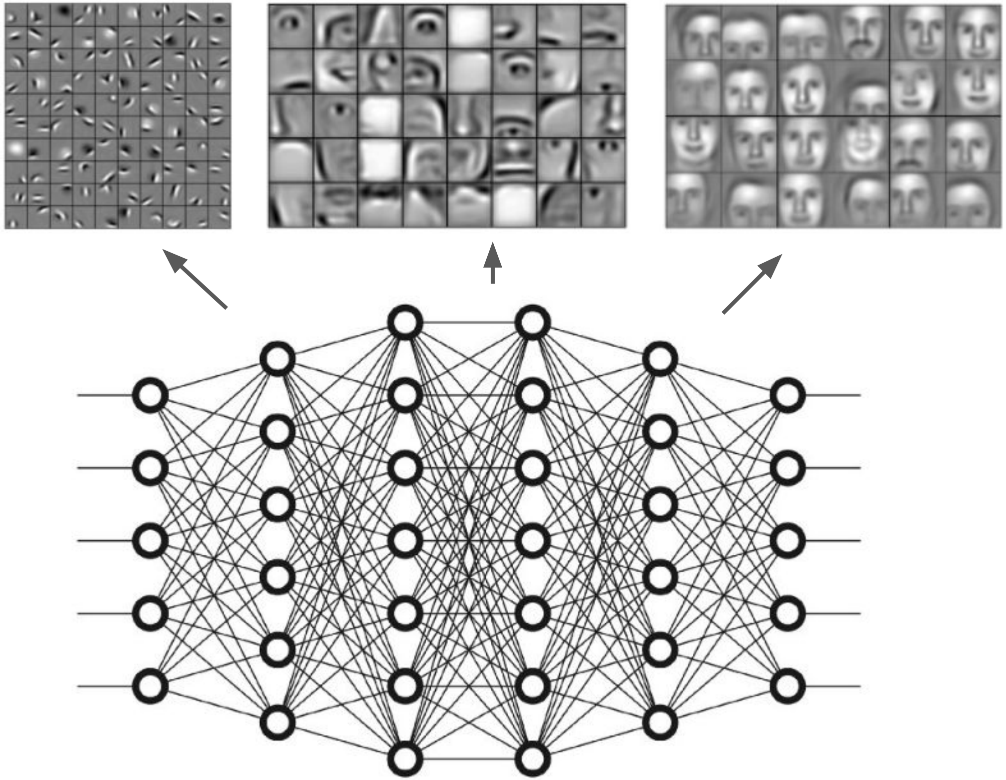
House price prediction



During the **optimization** process, The machine learns to **encode** a **representation** that maps the input to the output

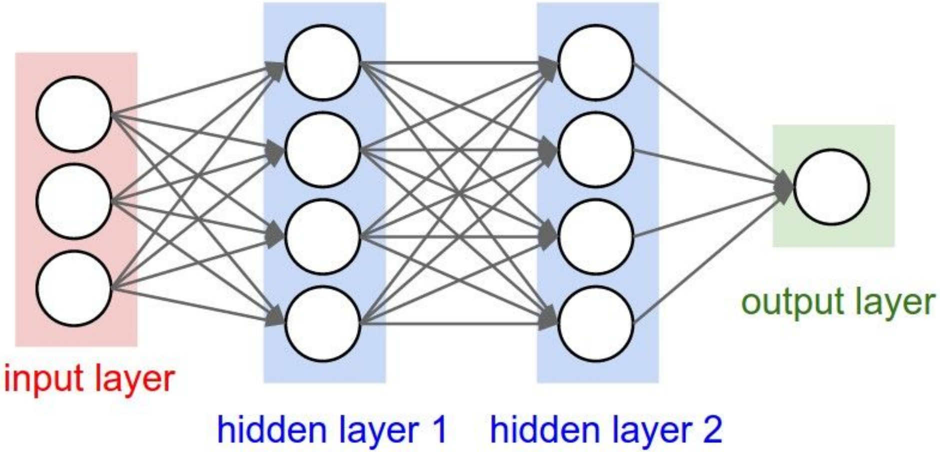
Adding Layers

A deep neural network **encodes** the **representation** in an increasingly abstract way



Adding Layers

- Neural networks are made from neurons and edges
- A collection of neurons in a layer
- The output of previous layer is used as an input to the next layer
- The input layer is data input and the output is a prediction
- Anything in between is **hidden**
- Layers are represented as vectors
- Edges are usually represented as matrices - **The weights**
- We train the weights



Adding Layers

Universal Approximation Theorem

“Given a neural network with a **single hidden layer** of **sufficient size**, the network can Approximate any continuous function”

In other words:

- There exists a true function relating the inputs to the outputs
- A neural network can approximate this function to arbitrary precision given sufficient layer size
- The required layer size can be extremely large and grow rapidly with the dimensionality of the problem

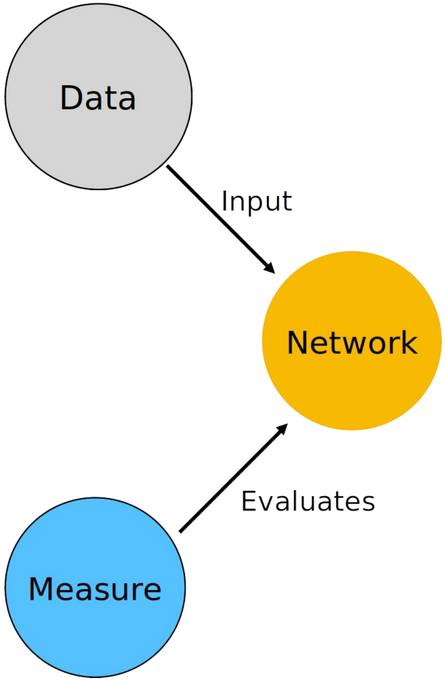
Use **multiple hidden layers** $\longrightarrow$ Encoding becomes increasingly more abstract

Adding Layers

Estimate $$\hat{y} = f_{NN}(x_1, x_2, \ldots, x_n)$$

Loss

Ground Truth

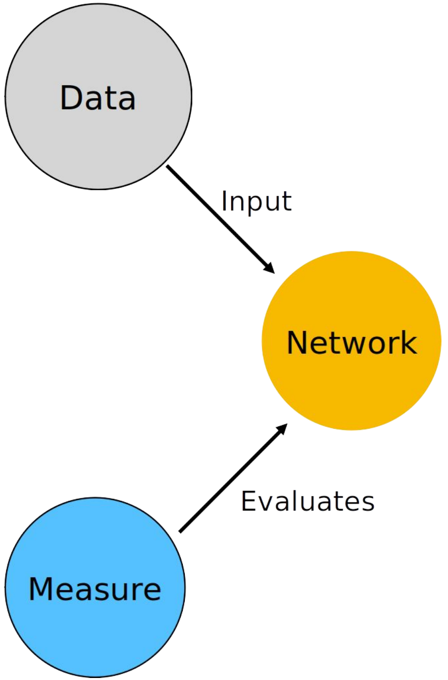


Adding Layers

Estimate $$\hat{y} = f_{NN}(x_1, x_2, \ldots, x_n)$$

Loss $$L(y, \hat{y}) = L(W, b) = \qquad (y_i - \hat{y}_i)^2$$

Ground Truth

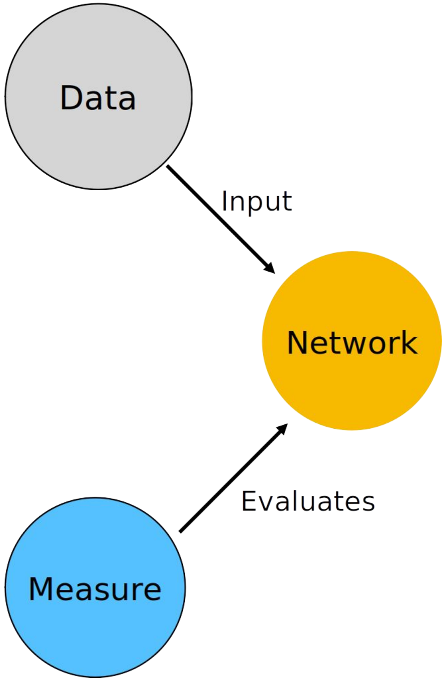


Adding Layers

Estimate $$\hat{y} = f_{NN}(x_1, x_2, \ldots, x_n)$$

Loss $$L(y, \hat{y}) = L(W, b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Ground Truth

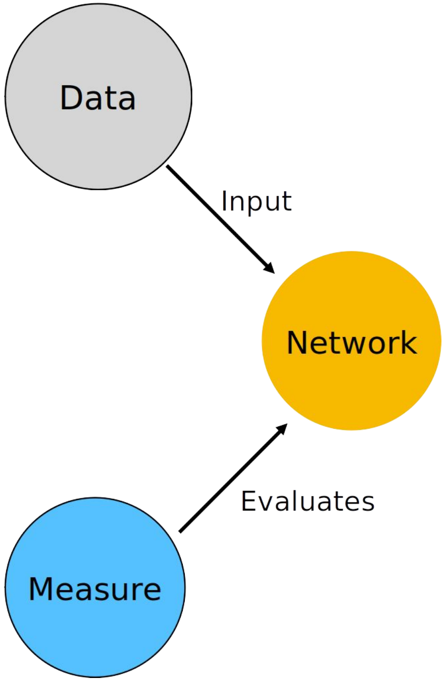


Adding Layers

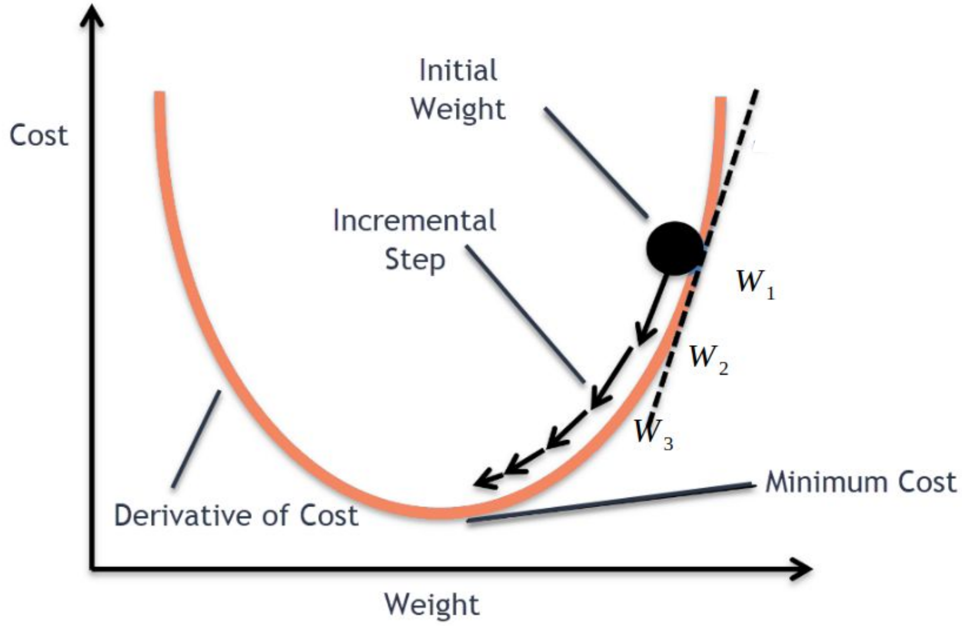
Estimate $$\hat{y} = f_{NN}(x_1, x_2, \ldots, x_n)$$

Loss $$L(y, \hat{y}) = L(W, b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

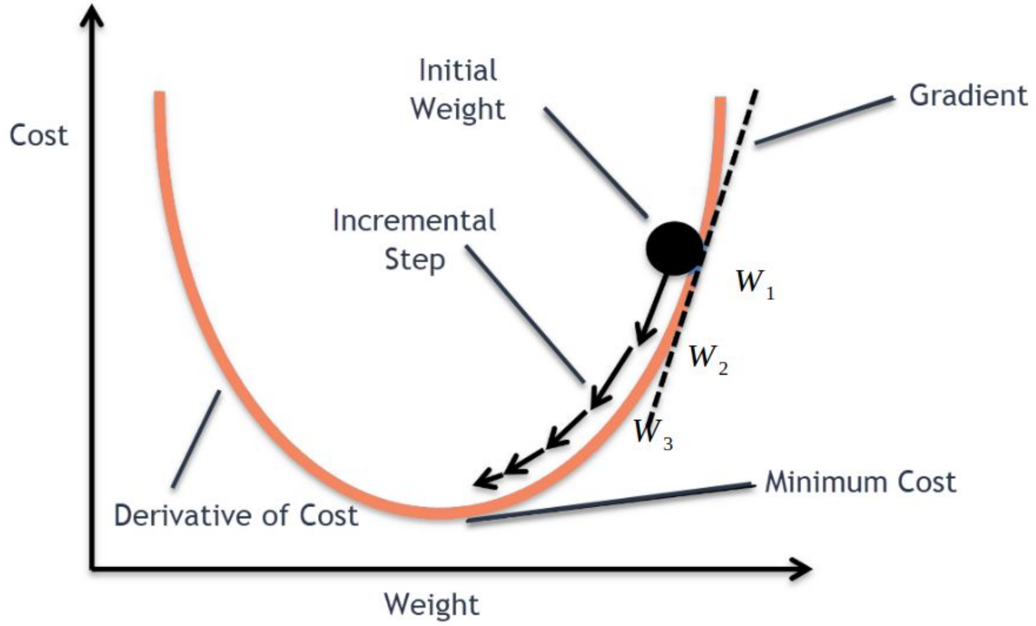
Ground Truth $$\mathbf{x} = (x_1, \ldots, x_m), y$$



Training a Neural Network

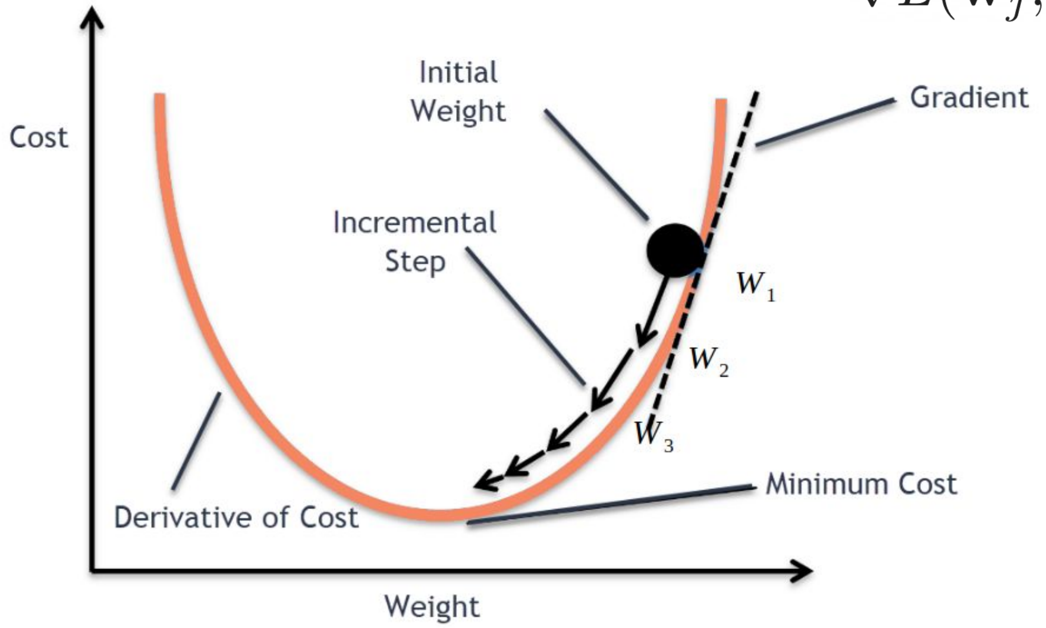


Training a Neural Network



$$L(y, \hat{y}) = L(W, b) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

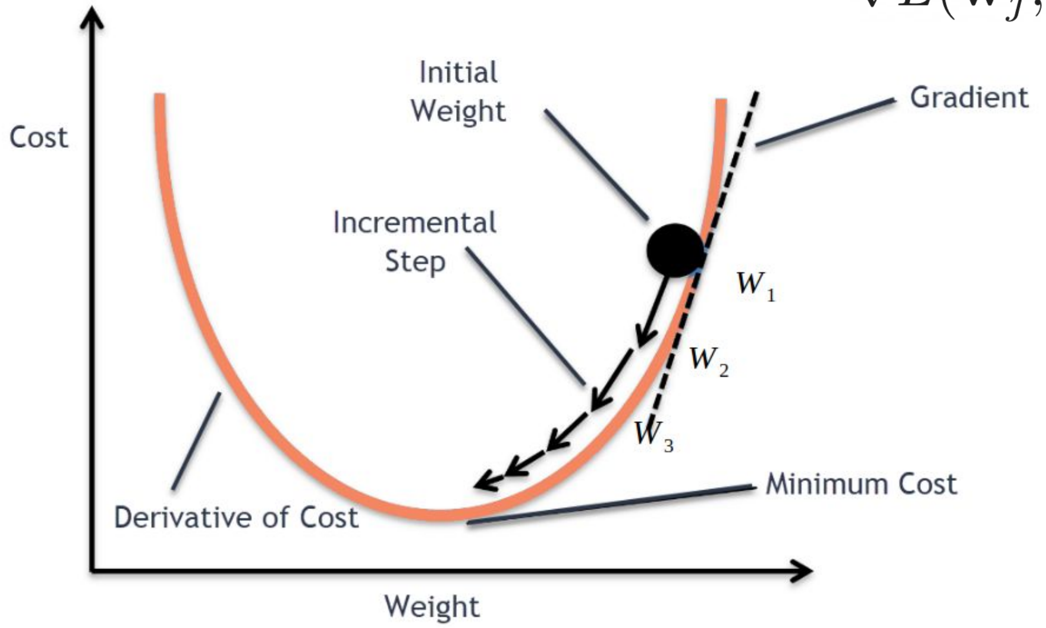
Training a Neural Network



$$\nabla L(\mathbf{w}_j, b)$$

$$L(y, \hat{y}) = L(W, b) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Training a Neural Network



$$\nabla L(\mathbf{w}_j, b)$$

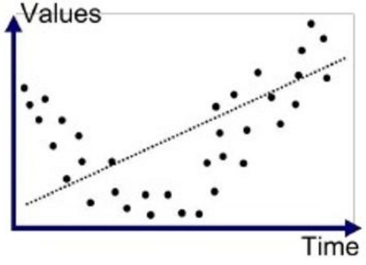
$$L(y, \hat{y}) = L(W, b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Learning Rate

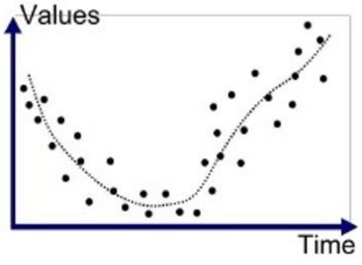
$$\mathbf{w}_{j+1} = \mathbf{w}_j - \alpha \nabla L(\mathbf{w}_j, b)$$

Overfitting

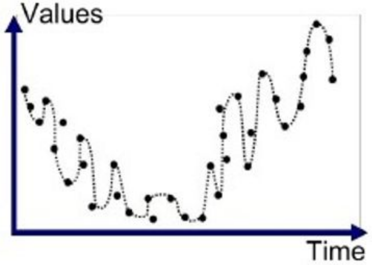
Do we want the lowest loss? **Not Really**



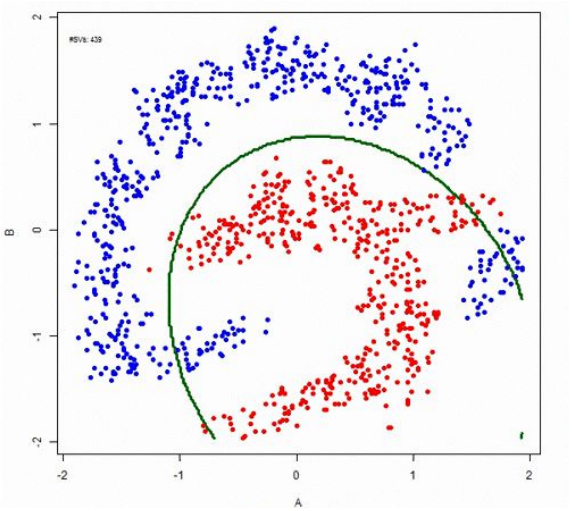
Underfitted



Good Fit/Robust



Overfitted



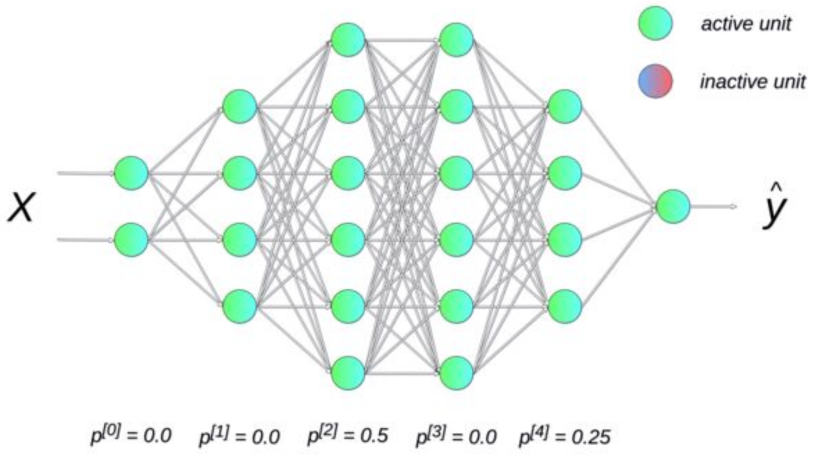
Overfitting

We have to combat overfitting

A few ways to do so is by:

- Simply stopping training earlier

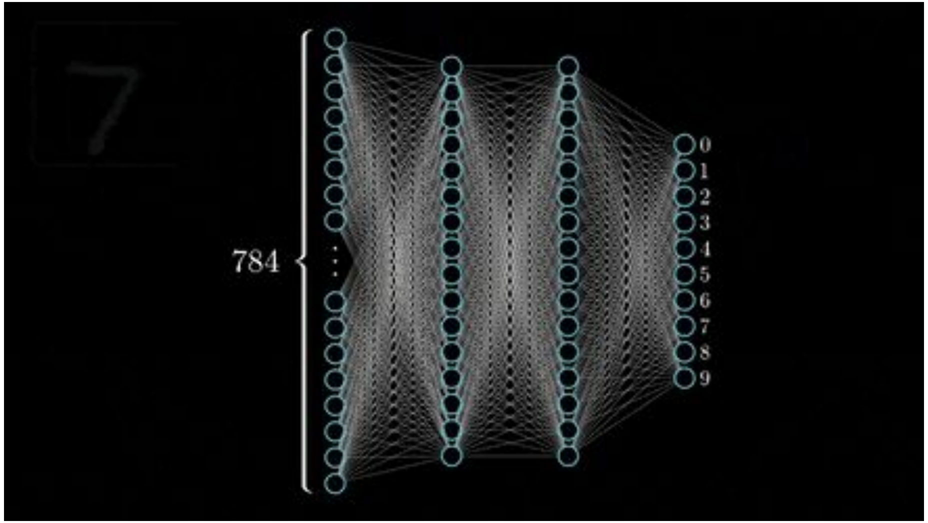
- Dropout: deactivate a neuron and its connections for the forward propagation with a certain probability



- Decay the value of your weights over time

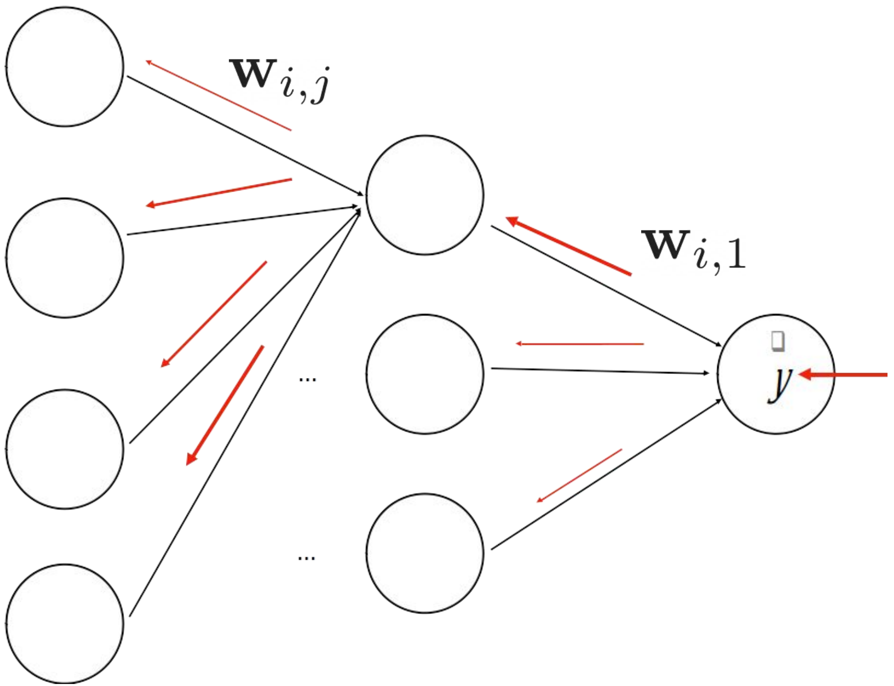
Training: Backwards Propagation

Forwards Propagation



Training: Backwards Propagation

Backwards Propagation

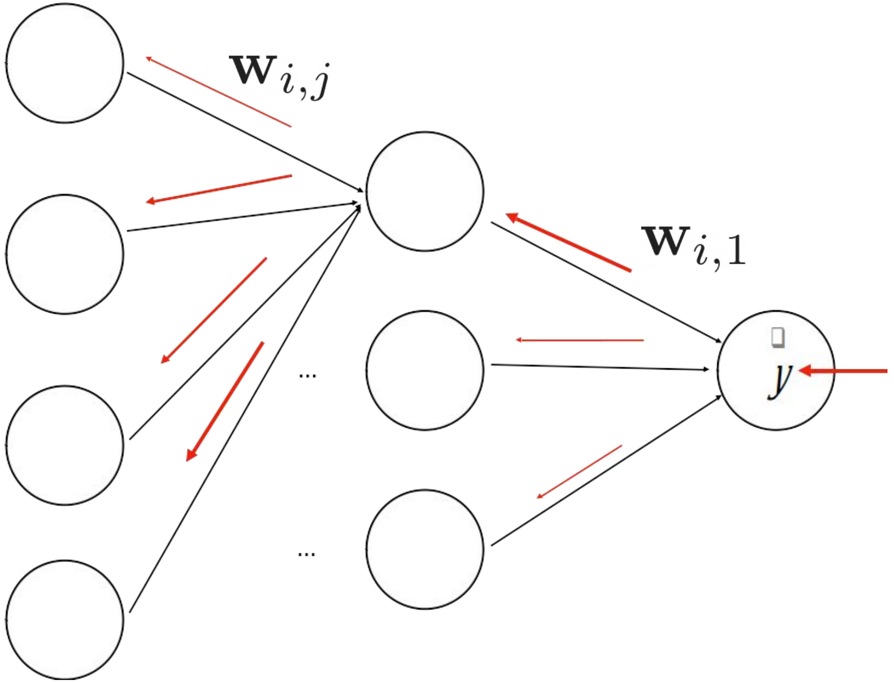


A (deep) neural network is a deeply **nested functions**:

- We need to compute the gradient for each layer
- Apply the **chain rule**
- This is **backpropagation**

Training: Backwards Propagation

Backwards Propagation



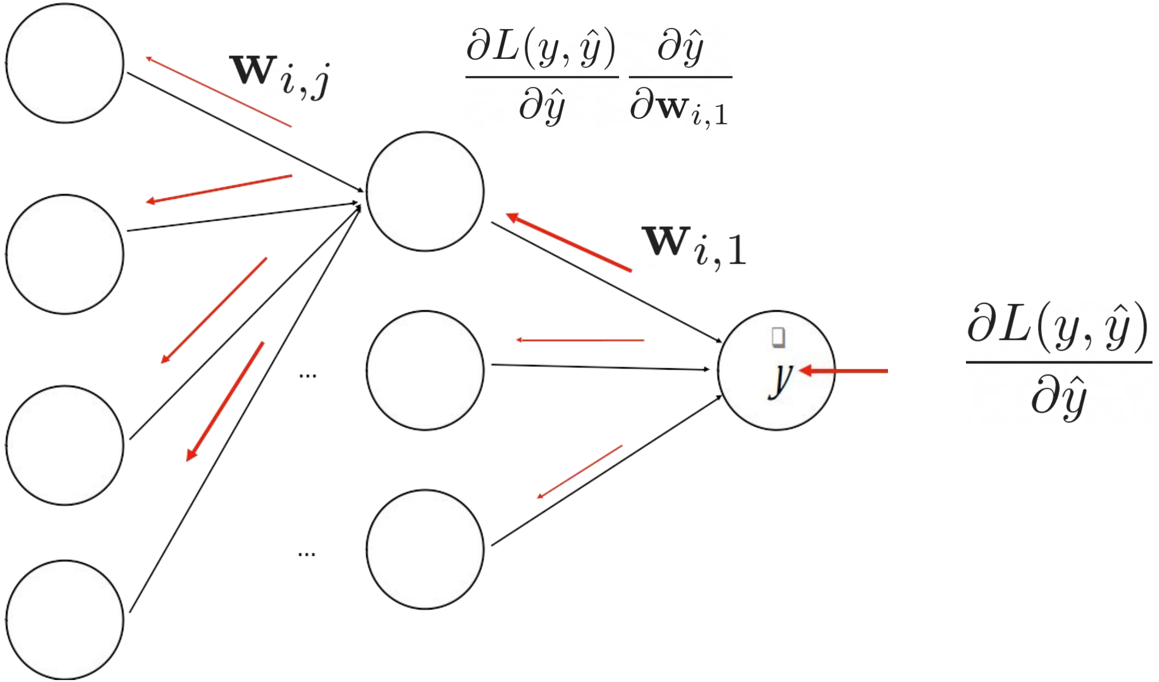
$$\frac{\partial L(y, \hat{y})}{\partial \hat{y}}$$

A (deep) neural network is a deeply **nested functions**:

- We need to compute the gradient for each layer
- Apply the **chain rule**
- This is **backpropagation**

Training: Backwards Propagation

Backwards Propagation



A (deep) neural network is a deeply **nested functions**:

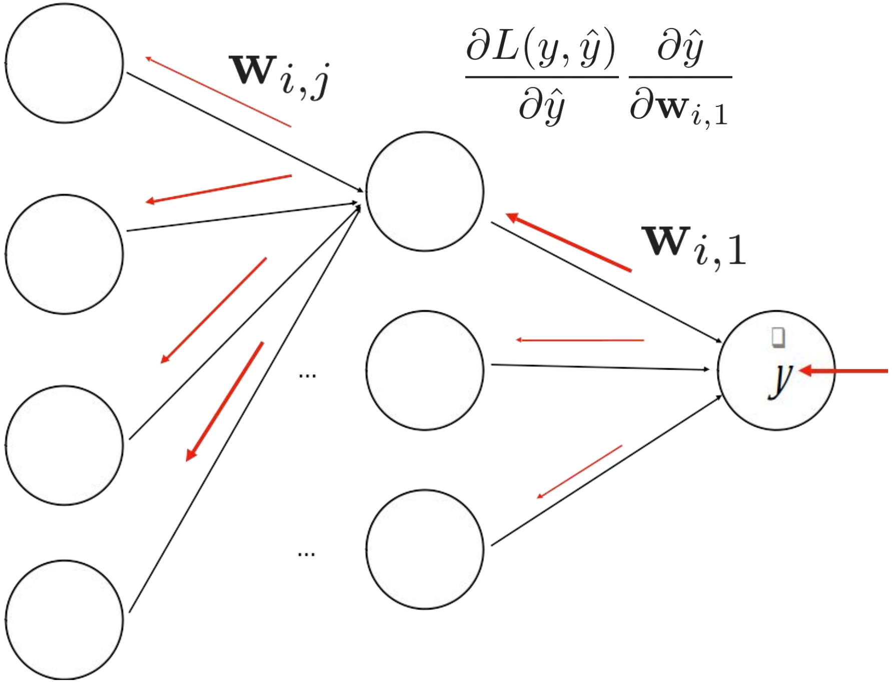
- We need to compute the gradient for each layer
- Apply the **chain rule**
- This is **backpropagation**

Training: Backwards Propagation

Backwards Propagation

$$\frac{\partial L(y,\hat{y})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{a}_i} \frac{\partial \mathbf{a}_i}{\partial \mathbf{w}_{i,j}}$$

$$\frac{\partial L(y,\hat{y})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}_{i,1}}$$



$$\frac{\partial L(y,\hat{y})}{\partial \hat{y}}$$

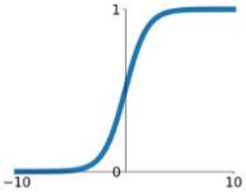
A (deep) neural network is a deeply **nested functions**:

- We need to compute the gradient for each layer
- Apply the **chain rule**
- This is **backpropagation**

Activation Functions

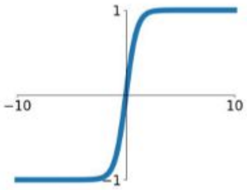
Sigmoid

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



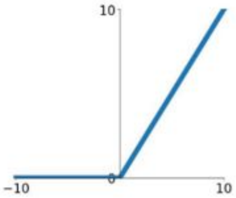
tanh

$\tanh(x)$



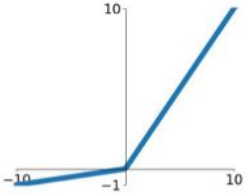
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

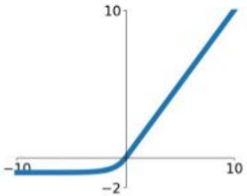


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Many more! We can design our own!

Commonly used loss functions

Regression

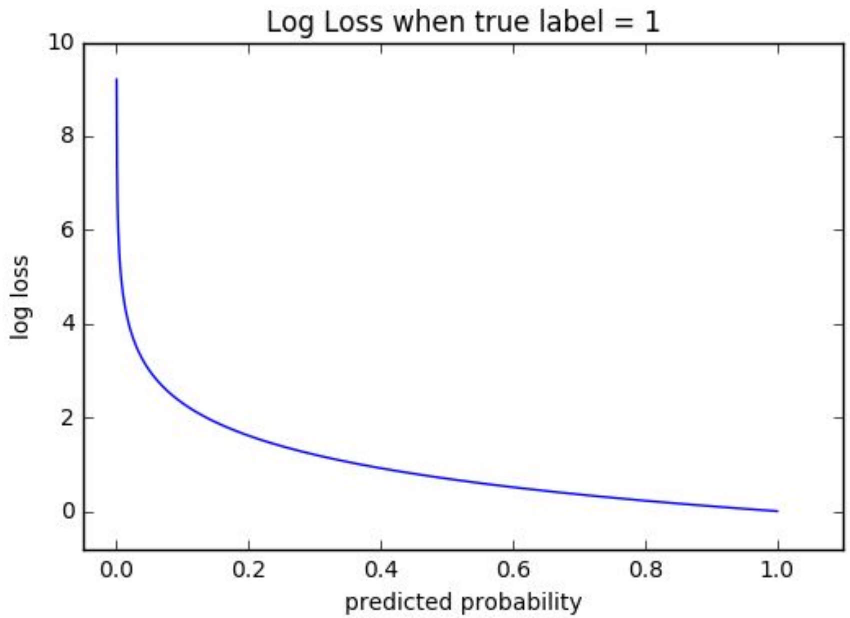
- Mean Squared Error (MSE)
- Mean Squared Log Error
- Mean Absolute Error

Binary Classification

- Binary cross-entropy
- Hinge Loss

Multi-Class Classification

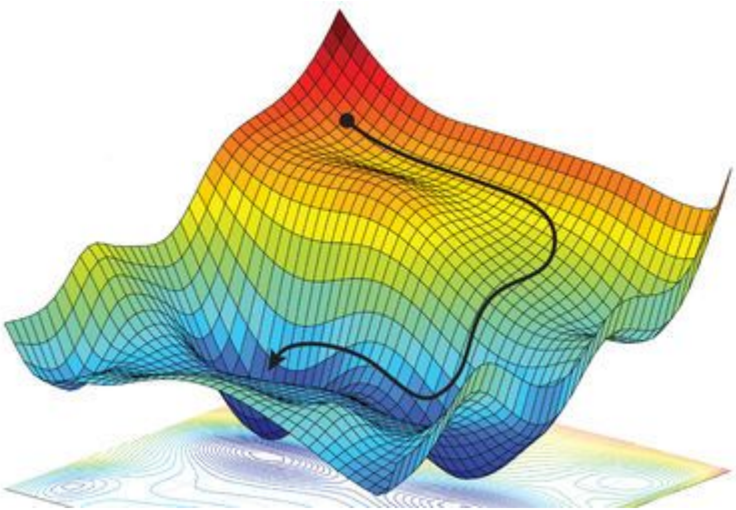
- Multi-class cross-entropy
- Kullback-Leibler Divergence



Cross-entropy loss outputs a log probability

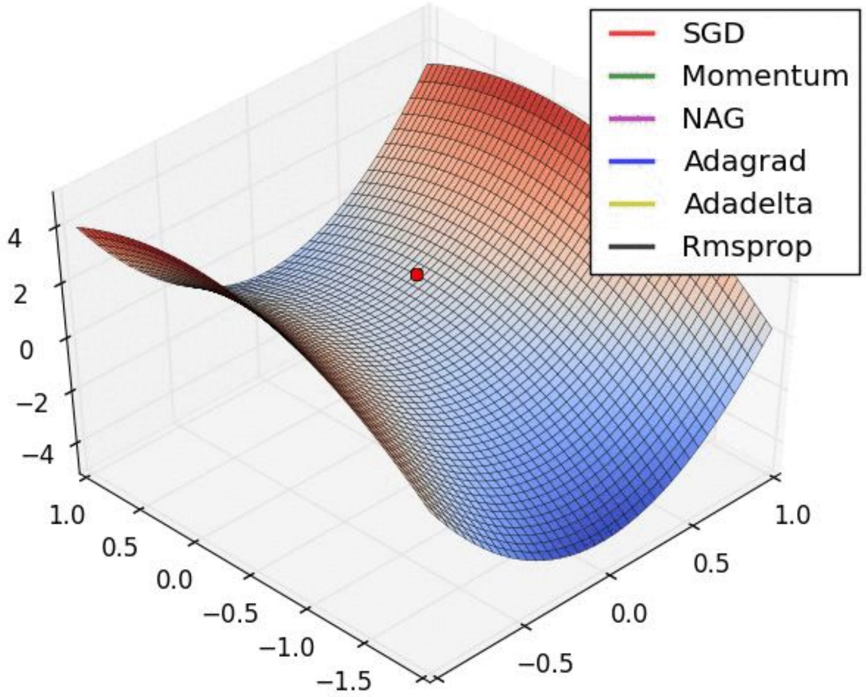
Optimizers

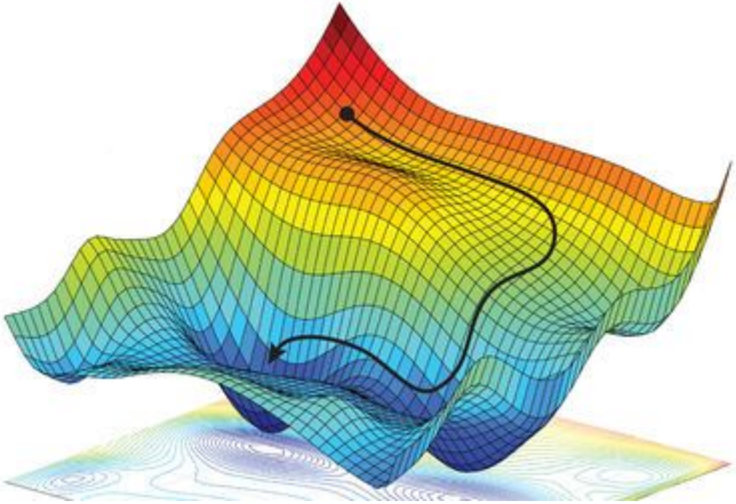
In what way should we change the weights?



Optimizers

In what way should we change the weights?



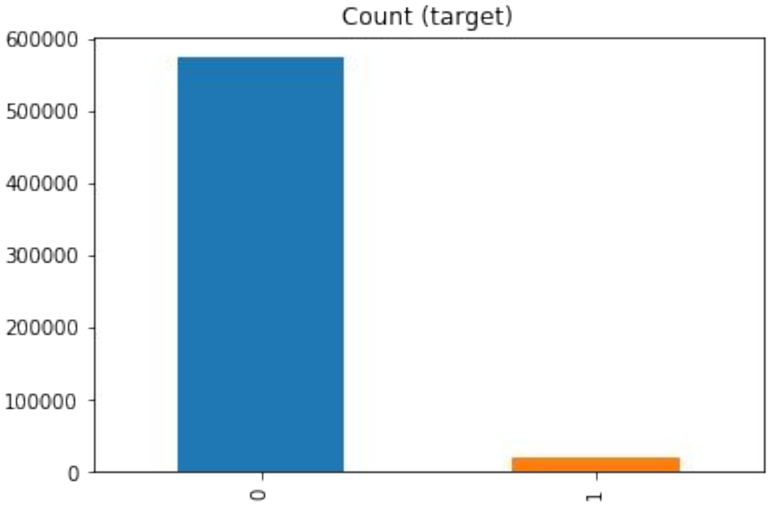


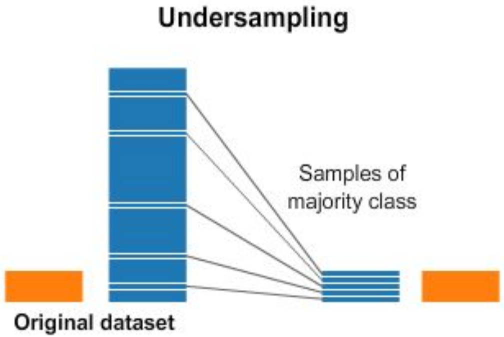
General Workflow of ML

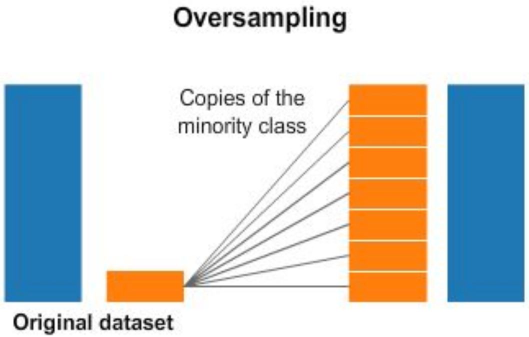
You need to know your data and your models well.

Artificial Intelligence still heavily relies on human intelligence

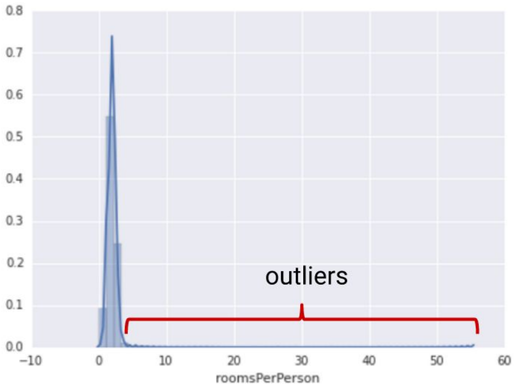
Imbalanced Training set

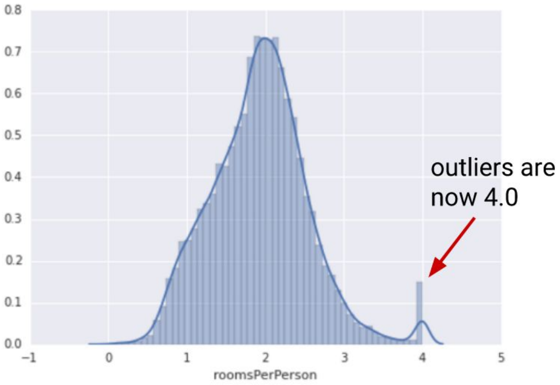


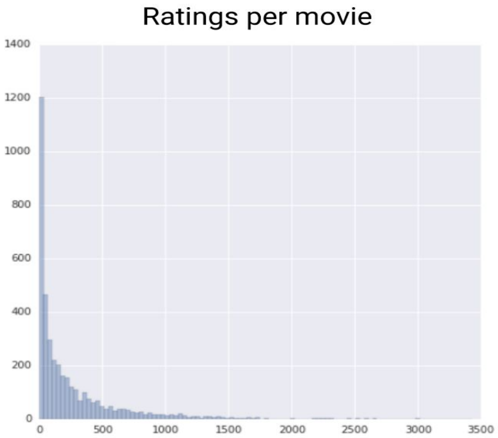


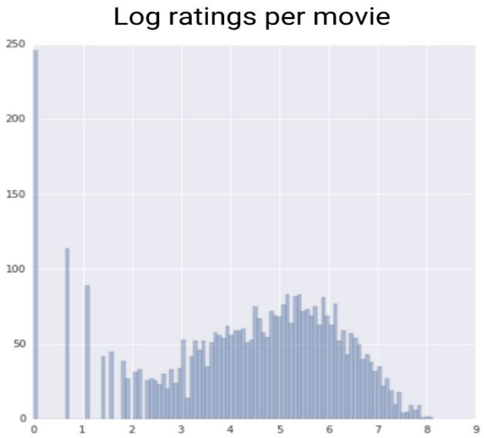


Data Normalization









A process to transform the input **data** in a **well-behaved** form

Dataset Splitting

Training set

Model training

Best model parameters
Weights

70%

Validation set

Model selection

Best hyperparameters
Learning rate
#neurons
#layers

20%

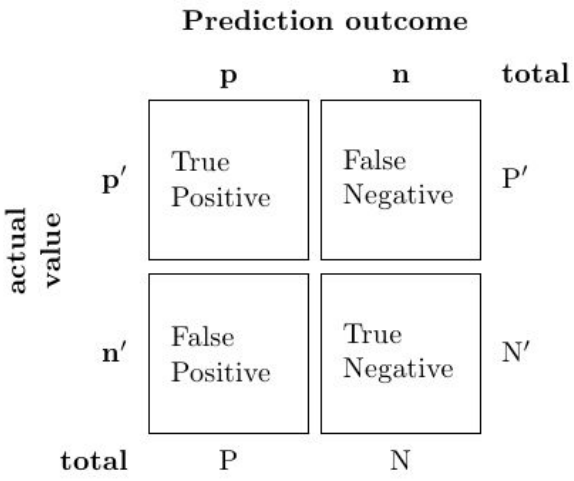
Test set

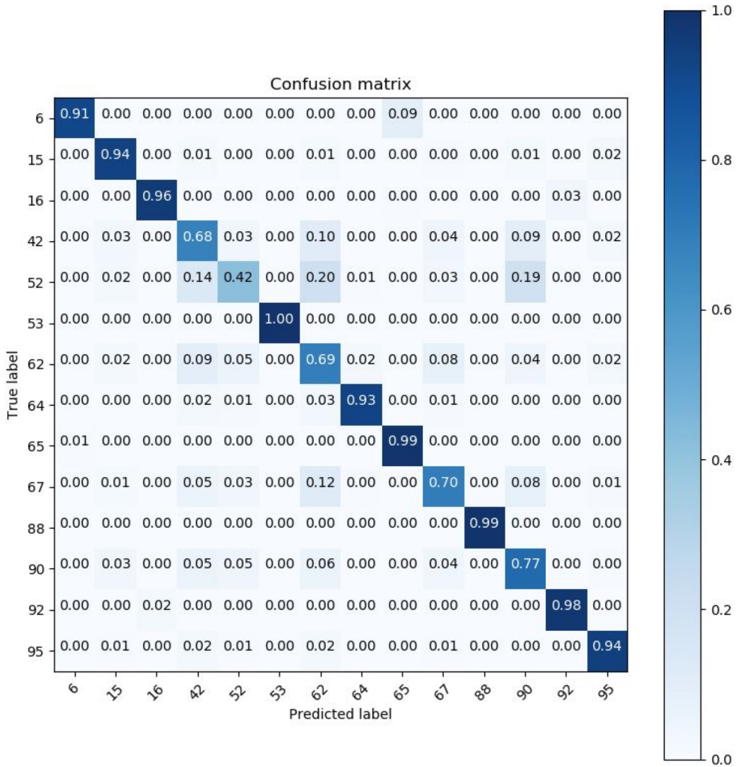
Model testing

Final model
Accuracy
Sensitivity/
specificity

10%

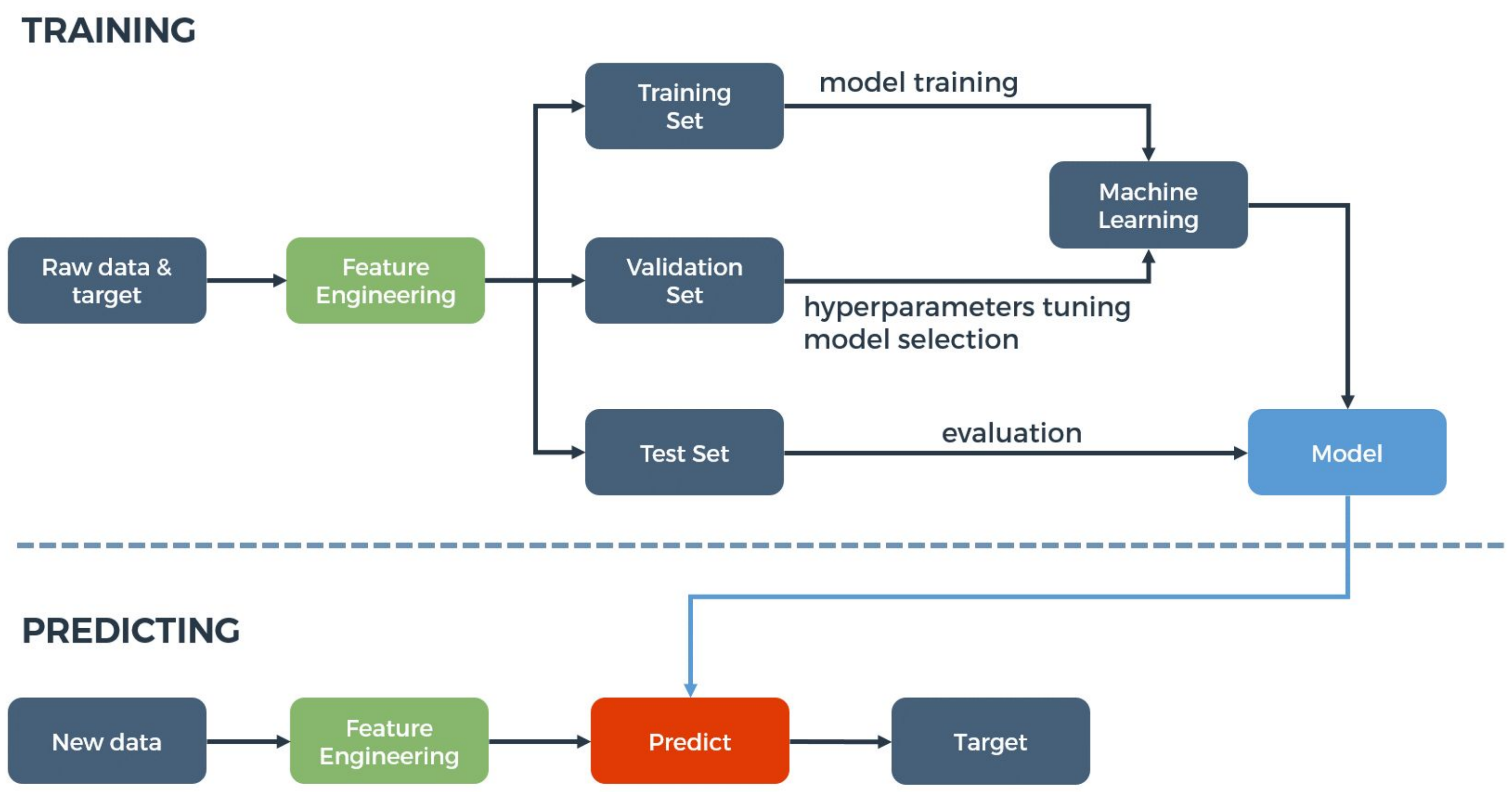
Network Evaluation





Accuracy = (TP + TN) / (TP + FN + FP + TN)
Precision (p) = TP / (TP + FP)
Recall (r) = TP / (TP + FN)

Workflow


TRAINING
Raw data & target
Feature Engineering
Training Set
Validation Set
Test Set
model training
Machine Learning
hyperparameters tuning
model selection
evaluation
Model
PREDICTING
New data
Feature Engineering
Predict
Target

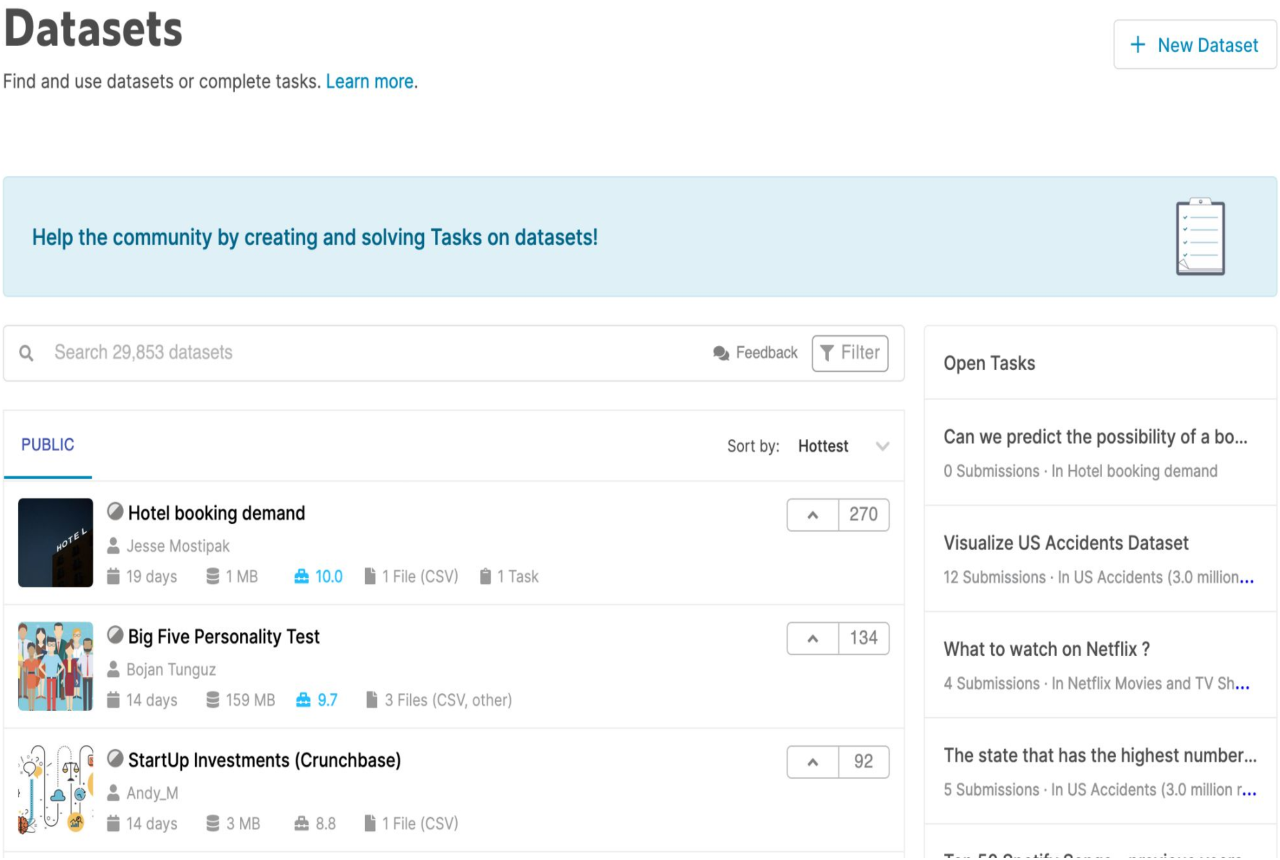
DL Frameworks

In DL you need to

- Define neurons and layers
- Define loss function
- Calculate losses
- Calculate gradient
- Propagate backward
- Update weights

- Existing frameworks exist:
    - TensorFlow (Keras)
    - Torch
    - Jax
    - MXNet

Open Datasets



Processed, balanced, well-behaved and labelled datasets

tensorflow.org/datasets

kaggle.com/datasets

topepo.github.io/caret/data-sets.html

github.com/awesomedata/awesome-public-datasets

Take Home Messages

Machine Learning

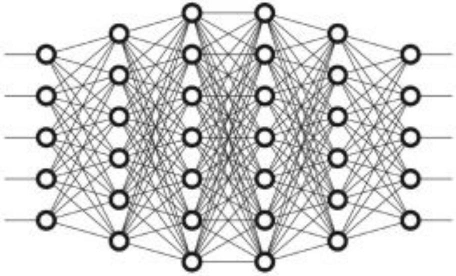
New paradigm of programming, driven by data
An optimization process

Deep Learning

A subfield of ML
Relies on deep neural networks
Learns to encode the input data using many layers of concept hierarchies

Take Home Messages

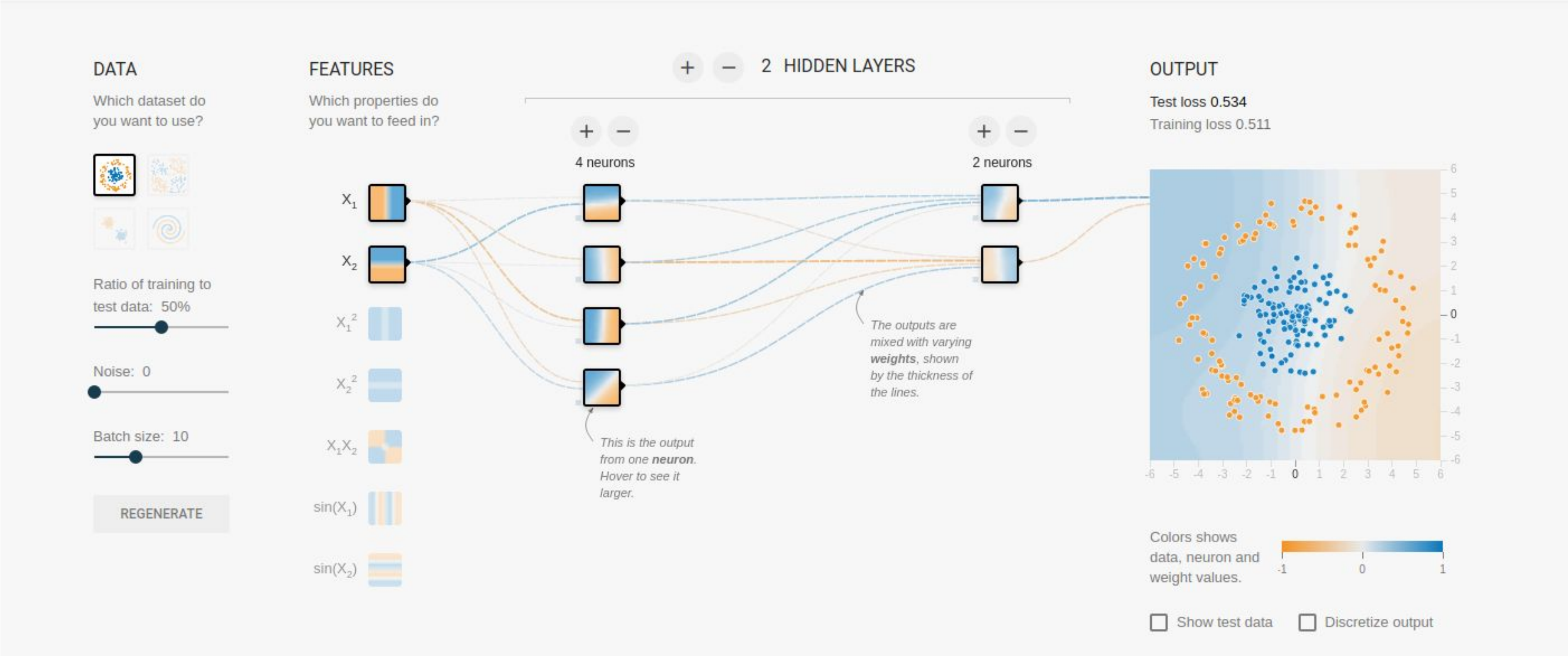
In a neuron:
... the main job is to calculate a weighted average
... the decision is made through the activation function



In a neural network:
... losses are calculated using the loss function
... losses are calculated by comparing the labels and the prediction
... predictions are made through forward propagation
... weights are updated through the backward propagation process
... optimizers are used to decide the weights updating strategies

In a deep learning workflow:
... the heavy lifting is mostly done by DL frameworks
... open datasets are crucial for benchmarking and bootstrapping DNNs

Live Demo



https://playground.tensorflow.org/

PyTorch

Three Levels of Abstraction

1. **Tensor:** imperative ndarray, possible to run on GPU/TPU

2. (node) **Variable:** Node in the built computational graph; data, gradient storage

3. **(**NN) **Module:** A neural network layer, store the state and the weights of the neural network

PyTorch

Three Levels of Abstraction

1. **Tensor:** imperative ndarray, possible to run on GPU

2. (node) **Variable:** Node in the built computational graph; data, gradient storage

3. **(**NN) **Module:** A neural network layer, store the state and the weights of the neural network

Pytorch will helps us with:

Defining a dataset
Automatic Gradient Computation

Defining Neural Networks

Optimization

Scheduling

Distributing

PyTorch

https://pytorch.org/docs/stable/

PyTorch | Get Started | Ecosystem | Mobile | Blog | Tutorials | Docs | Resources | GitHub

1.11.0 ▼

Search Docs

Notes [+]

Language Bindings [+]

Python API [-]

torch
torch.nn
torch.nn.functional
torch.Tensor
Tensor Attributes
Tensor Views
torch.autograd
torch.cuda
torch.cuda.amp
torch.backends
torch.distributed
torch.distributed.algorithms.join
torch.distributed.elastic
torch.distributed.fsdp
torch.distributed.optim
torch.distributions
torch.fft
torch.futures
torch.fx
torch.hub

Docs > torch.nn

Shortcuts

TORCH.NN

These are the basic building blocks for graphs:

torch.nn

- Containers
- Convolution Layers
- Pooling layers
- Padding Layers
- Non-linear Activations (weighted sum, nonlinearity)
- Non-linear Activations (other)
- Normalization Layers
- Recurrent Layers
- Transformer Layers
- Linear Layers
- Dropout Layers
- Sparse Layers
- Distance Functions
- Loss Functions
- Vision Layers
- Shuffle Layers
- DataParallel Layers (multi-GPU, distributed)
- Utilities
- Quantized Functions
- Lazy Modules Initialization

PyTorch

General Structure for training Neural Networks

data loader

model

optimizer

loss function

PyTorch

General Structure for training Neural Networks

data loader

model

optimizer

loss function

For every datapoint, y in data_loader

PyTorch

General Structure for training Neural Networks

data loader

model

optimizer

loss function

For every datapoint, y in data_loader

        optimizer.zero_grad()

PyTorch

General Structure for training Neural Networks

data loader

model

optimizer

loss function

For every datapoint, y in data_loader

    optimizer.zero_grad()

    prediction = model(datapoint)

PyTorch

General Structure for training Neural Networks

data loader

model

optimizer

loss function

For every datapoint, y in data_loader

    optimizer.zero_grad()

    prediction = model(datapoint)

    loss = loss_function(prediction, y)

PyTorch

General Structure for training Neural Networks

data loader

model

optimizer

loss function

For every datapoint, y in data_loader

    optimizer.zero_grad()

    prediction = model(datapoint)

    loss = loss_function(prediction, y)

    loss.backward()

PyTorch

General Structure for training Neural Networks

data loader

model

optimizer

loss function

For every datapoint, y in data_loader

  optimizer.zero_grad()

  prediction = model(datapoint)

  loss = loss_function(prediction, y)

  loss.backward()

  optimizer.step()

PyTorch

General Structure for training Neural Networks

data loader

model

optimizer

loss function

For every datapoint, y in **data_loader**

  optimizer.zero_grad()

  prediction = **model**(datapoint)

  loss = **loss_function**(prediction, y)

  loss.backward()

  optimizer.step()

PyTorch

General Structure for training Neural Networks

data loader

model

optimizer

loss function

For every datapoint, y in **data_loader**

optimizer.zero_grad()

prediction = **model**(datapoint)

loss = **loss_function**(prediction, y)

loss.backward()

optimizer.step()

```
for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)

    optimizer.zero_grad()
    output = model(data)
    loss = F.nll_loss(output, target)
    loss.backward()
    optimizer.step()
```

PyTorch

General Structure for training Neural Networks

data loader

model

optimizer

loss function

For every datapoint, y in **data_loader**

    optimizer.zero_grad()

    prediction = **model**(datapoint)

    loss = **loss_function**(prediction, y)

    loss.backward()

    optimizer.step()

```
for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)

    optimizer.zero_grad()
    output = model(data)
    loss = F.nll_loss(output, target)
    loss.backward()
    optimizer.step()
```

$$\mathbf{w}_{j+1} = \mathbf{w}_j - \alpha \nabla L(\mathbf{w}_j, b)$$

PyTorch

Data:

d_1 = [0.9, -0.2], y = 0

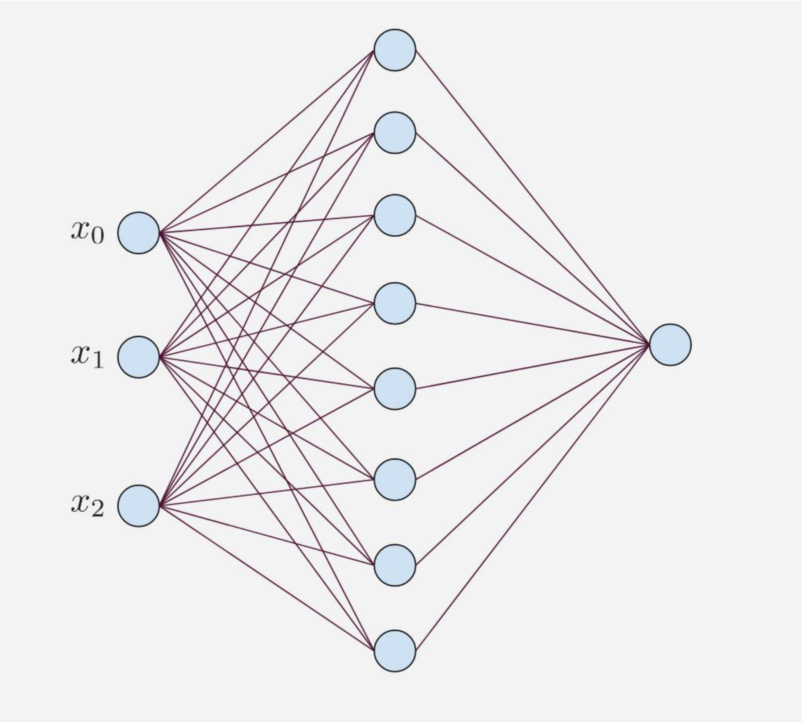
d_2 = [0.75, 0.6],y = 1

PyTorch

Data:

d_1 = [0.9, -0.2], y = 0

d_2 = [0.75, 0.6],y = 1



PyTorch

Data:

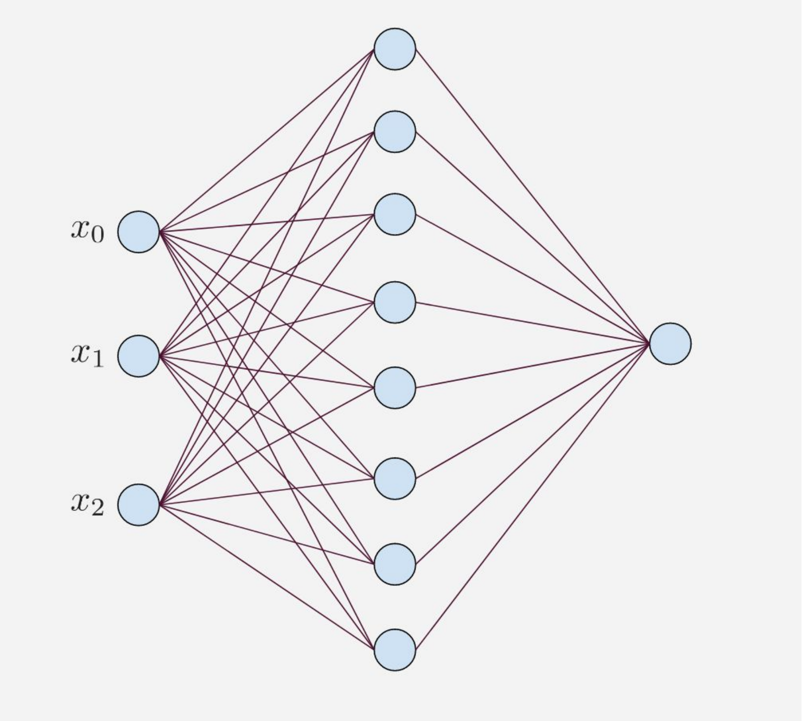
d_1 = [0.9, -0.2], y = 0

d_2 = [0.75, 0.6],y = 1

Learning rate = 0.01

Optimizer = Stochastic Gradient Descent

Loss = Binary Cross Entropy



PyTorch

Data:

d_1 = [0.9, -0.2], y = 0

d_2 = [0.75, 0.6],y = 1

Learning rate = 0.01

Optimizer = Stochastic Gradient Descent

Loss = Binary Cross Entropy

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\\text{size}}} y_i \cdot \log \hat{y}_i$$