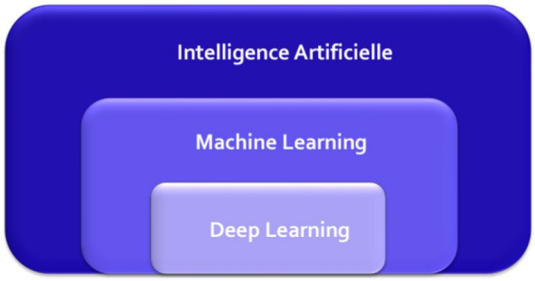
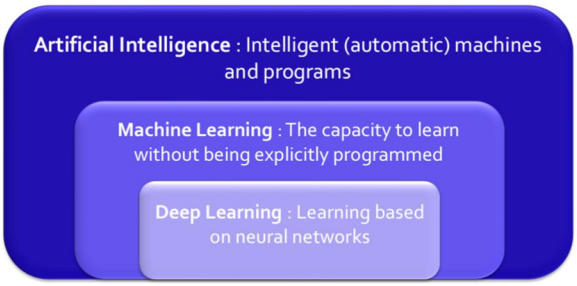
ANN (intermediate course between ML & DL

The spheres of AI >>





Deep Learning Libraries

TensorFlow:

- Created by: Google.
- For/by researchers
- Free
- **Keras** (interface or API for TensorFlow (the front end or middleware) to use TensorFlow Services

Pytorch (more user friendly than TensorFlow**)**:

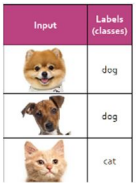
- Created by: META.
- For/by researchers
- Free

DL Types:

- **Supervised** learning: train model on existing **labeled data** (categories)
- **Unsupervised** learning or self-supervised learning: train model on **unlabeled data** (clusters) like cats images similar to dogs
- **Semi-Supervised**: train model on small labeled data and **large unlabeled data, predict missing labels.**
- **Reinforcement** learning (environment): train a model **based on a reward system**. (The system makes a decision with respect to its present state and receives a positive or a negative reward in response to its decision), the type which using chatgpt is called HLRL (Human in the loop reinforcement learning)

ML:

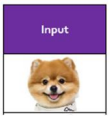
- A branch of artificial intelligence
- Trains an algorithm to recognize patterns and features(characteristics) in a given dataset.
- Predicts and makes decisions for new data based on what it had learned.



How does Machine learning work?

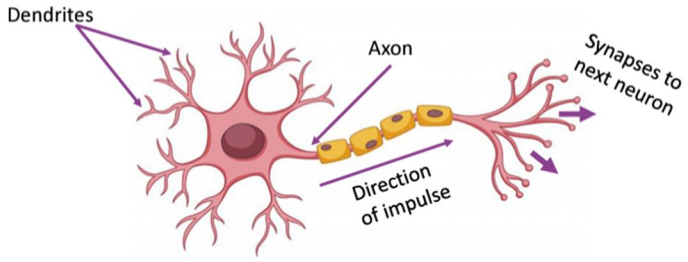
- Database preparation (cleaning),
  - labeled: labeled by humans, (classes to learn are defined by humans)

- non-labeled: model extracts the features and defines the categories by itself. The model extracts the characteristics then defines the classes itself, with no human interaction.

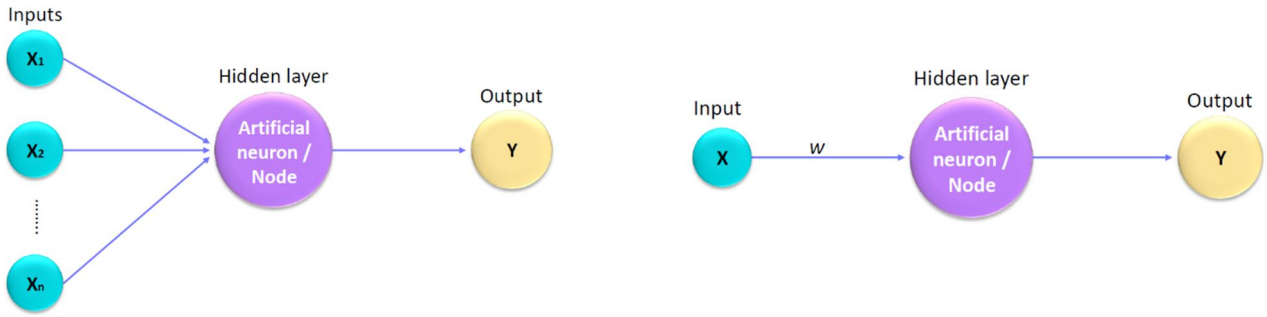


- Select model learning algorithms.
- Algorithm training to obtain the model.
- Using the trained model.

ANN: used for Prediction, classification, find pattern, extract features from data.



The artificial neuron: the building block of neural networks



Y = Input x weight >> $Y = X.w$

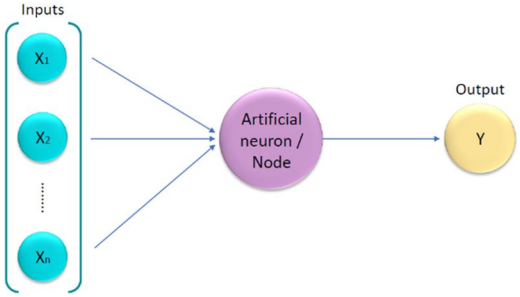
If multiple inputs, then the output:

$$Y = X_1w_1 + X_2w_2 + \cdots + X_nw_n >> Y = \sum_{i=1}^{n} X_i * W_i$$

ANN Layers: Inputs or Features (independent variables for a single observation.)

Can be

- Categorical
- Numerical



Example :

If our *observation* is a *car* :

X1 : brand

X2 : manufacturing year

X3 : color

X4 : automatic/mechanical

⋮

The inputs:

Data preparation :

To avoid false influence of the different input values and intervals on the trained model

1

Standardization :

1. Subtract the mean
2. Divide by the standard deviation

$Xi - mean \ / \ standard\_dev$

$==> Mean = 0 \ \& \ Variance = 1$

2

Normalization :

1. Subtract minimum value
2. Divide by the data range

$X_i - X_{min} \ / \ (X_{max} - X_{min})$

$==> 0 \leq X_{i\_new} \leq 1$

1

Standardization :

Z-Score normalization

Mean = 0

Variance = 1

More robust to outliers (preserves them

Can be applied when the indepent variables follow normal (Gaussian) distributions with different means and standard deviations

2

Normalization :

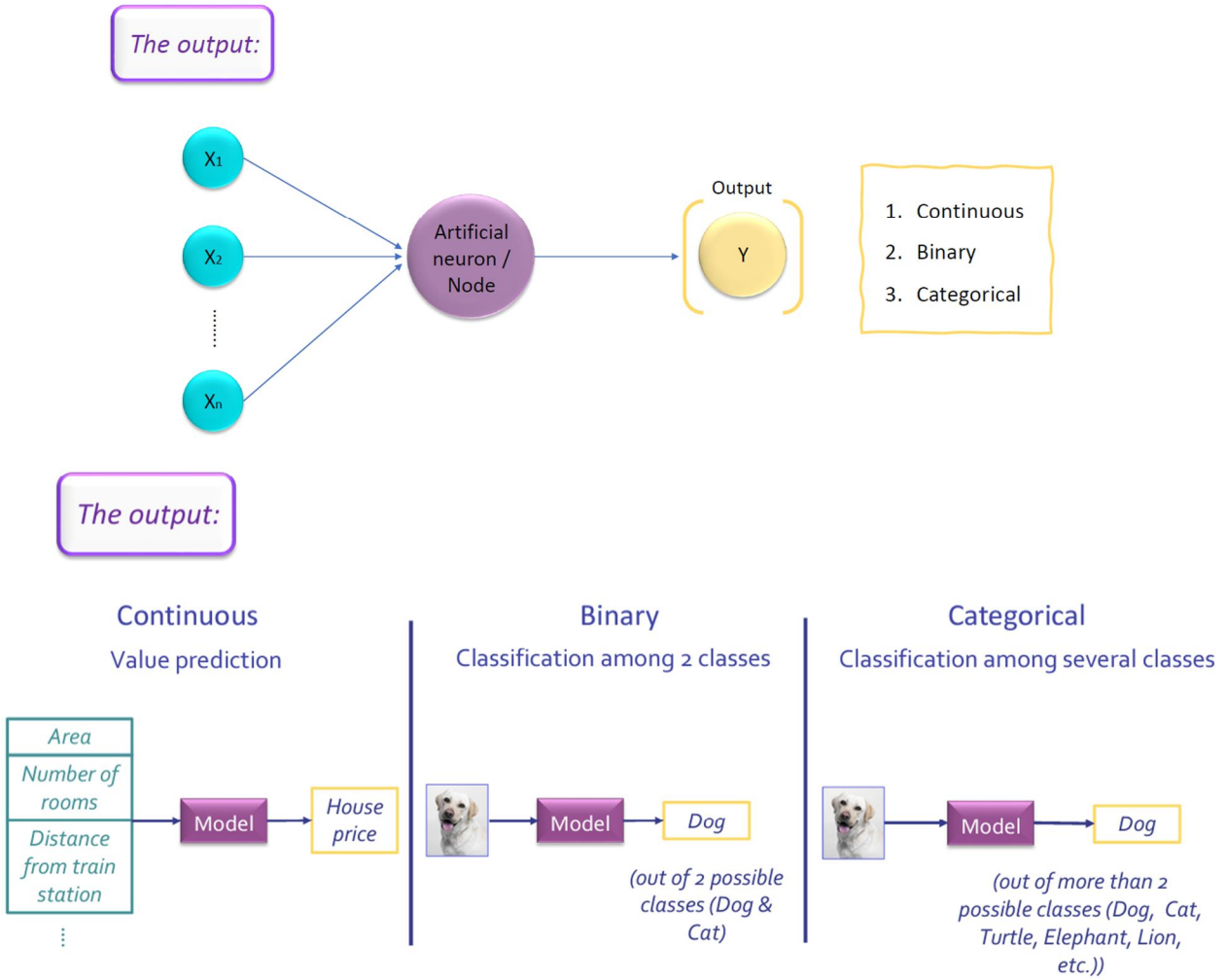
Min-Max Scaling

1. Subtract minimum value
2. Divide by the data range

$X_i - X_{min} \ / \ (X_{max} - X_{min})$

$==> 0 \leq X_{i\_new} \leq 1$

Can be applied when the indepent variables vary in different scales and intervals



Inputs & Outputs

Categorical Data Encoding: Most Machine Learning models do not accept text labels for inputs and outputs.

- We have to encode those categorical labels to numerical values.

1. Ordinal Encoding: Every label is represented by an ordered number

Example: If we are classifying countries:

The system will probably establish false relations between these countries.

Example : It can consider India superior to Japan while there is no hierarchical relation between the countries !

Country (Label)	Ordinal encoding
India	1
Japan	2
Madagascar	3
USA	4

2. One-Hot Encoding: Creates new binary variables based on the number of labels → N labels are represented by N bits, while only one bit = 1

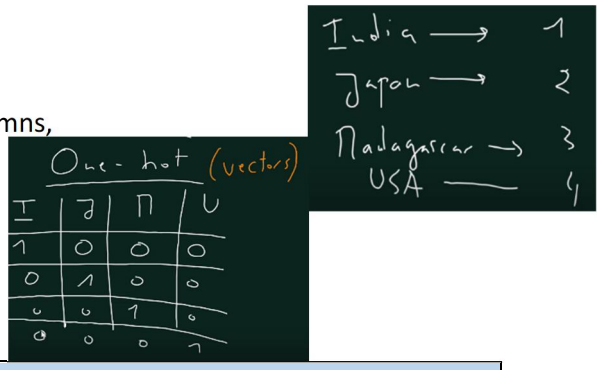
Example : If we are classifying countries :

Country (Label)	One-Hot encoding
India	1 0 0
Japan	0 1 0
Madagascar	0 0 1

If we have a large number of categories, the variable can grow to demand a considerable amount of memory storage.

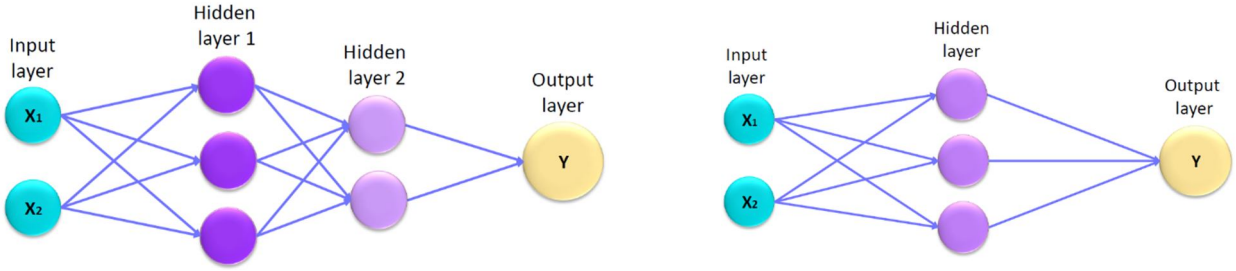
Feature Encoding

- Ordinal (sequence of numbers)
- One-hot (vectors) >> distribute the categories in columns, and give 1 exist or 0 if not,
  (if too many columns it could cause memory issue)



So, which encoding should we use?

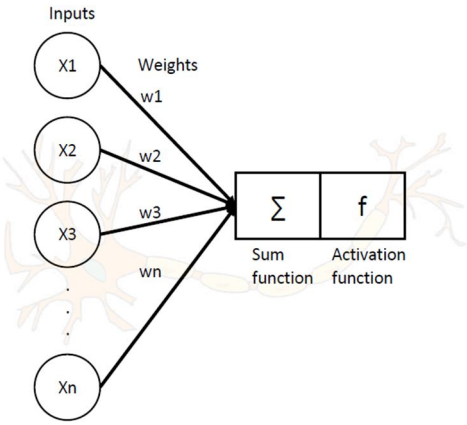
Ordinal Encoding	One Hot Encoding
- When the labels are ordered primary school, secondary school, university - The number of categories is very large	- When the labels are not ordered (countries, animals) - The number of categories is not too large to cause memory problems

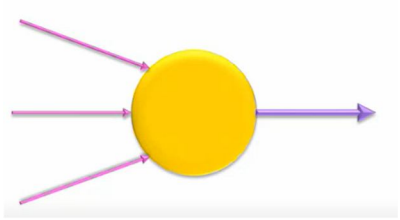


- The hidden layer is situated between inputs and outputs of a neural network
- The hidden nodes have no direct connection with the output world
- A collection of hidden nodes forms a « hidden layer »
- The output of one neuron is the input of another

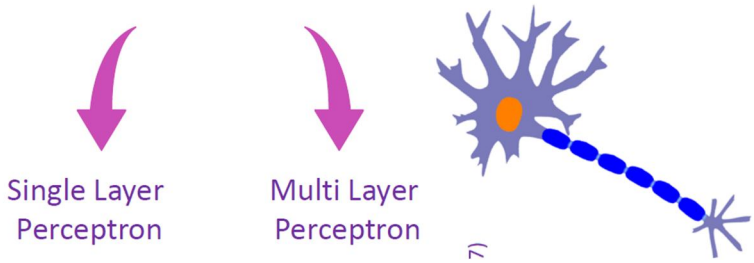
- There usually are multiple hidden layers in a neural network
- ANNs can have zero or multiple hidden layers
- Traditional neural networks typically have fully connected hidden layers (**dense**)
- Fully connected layer(**dense**) : every neuron is connected to all the neurons in the previous and the next layers

The perceptron





- **The Perceptron model is proposed by Frank Rosenblatt (1957)**

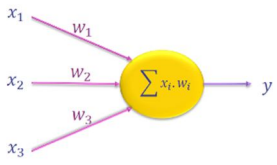


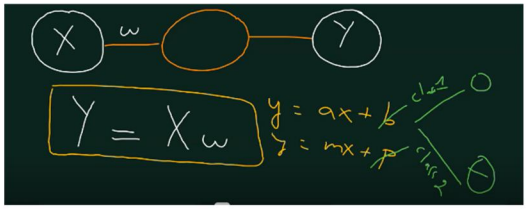
- **Modeled after the essential unit of the human brain – the neuron.**

Single Layer Perceptron

- SLP is the simplest type of ANN.
- **contains no hidden layers**.
- can only classify linearly separable classes (a straight line to separate classes)
- can only perform binary classification (2 classes)

Single layer perceptron is only good for linear data.
For binary classification 0 or 1 (rare) most of the data is missy and not linear



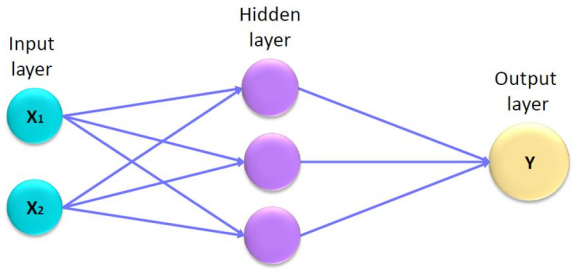


$$y = \begin{cases} 1 \; if \sum x_i.w_i > \theta \; (predetermined) \\ 0 \quad otherwise \end{cases}$$

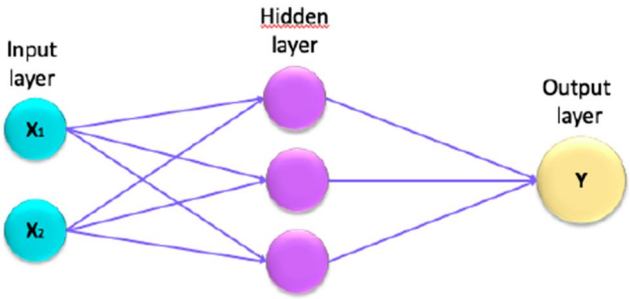
- Each input has a corresponding weight to designate how important it is.
- The output y provides a decision (one of two classes)

Multi-Layer Perceptron

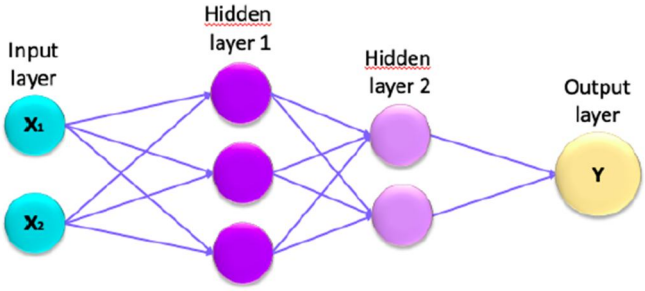
MLP is a class of feedforward ANNs.



- Has one or more hidden layers
- Perceptron's team up with each other to solve complex problems even non-linearly separated ones.
- Each layer can have multiple perceptron's, and there can be multiple layers.



An ANN with one hidden layer is called a shallow or non-deep neural network.



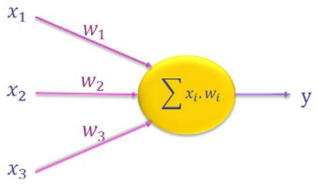
An ANN *with more than one hidden layer is called deep neural network*.

Difference between MLP and Neural Network

MLP	Neural Networks
- Uses a step function for decision - The decision is binary	- Evolved from MLP - Other activation functions can be used - Outputs are real values, probability-based or classes

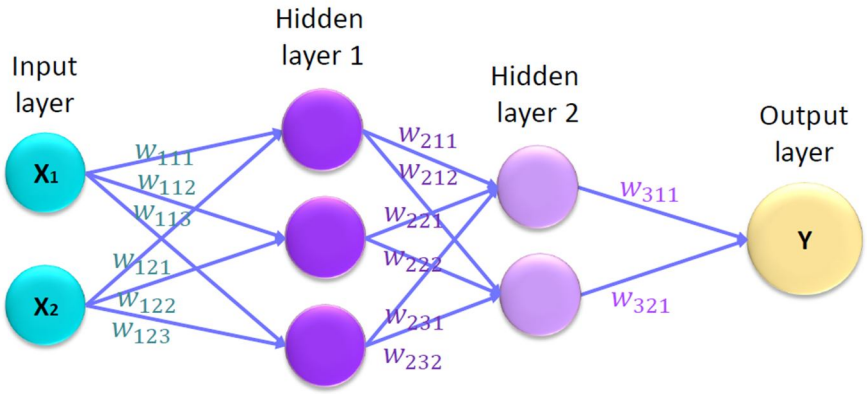
Weights and Biases

Weights determine the relative importance of every feature in the classification decision.



$$y = \sum x_i . w_i$$

Weights

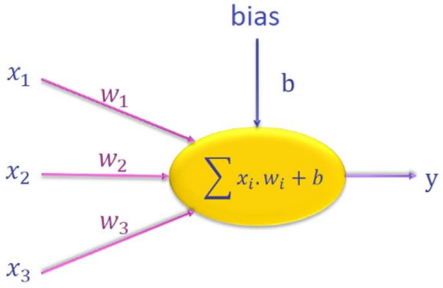


Biases

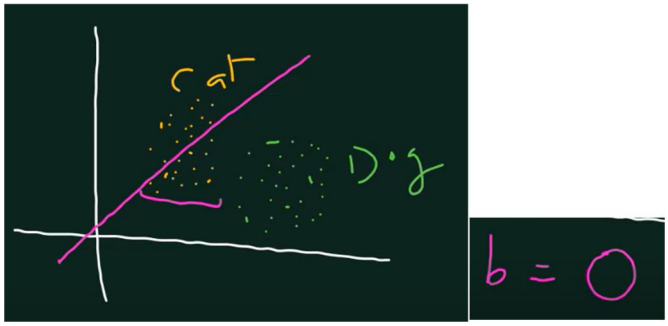
A bias b:

- is a constant that helps the model fit the given data best.
- behaves as the intercept of a linear equation.

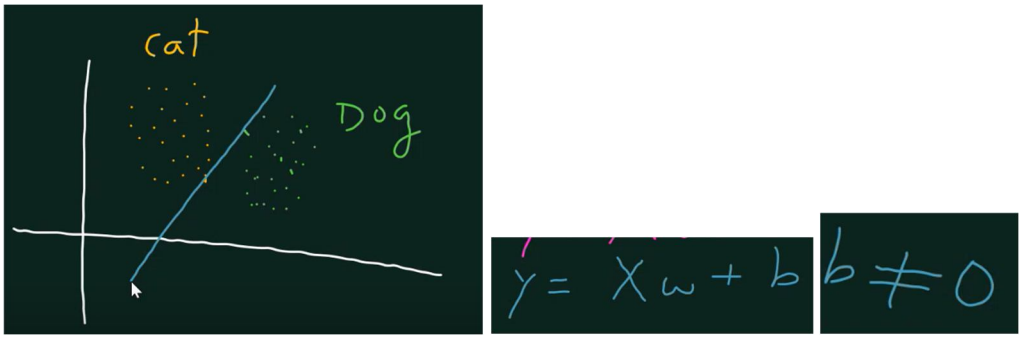
$$y = \sum x_i . w_i + b$$



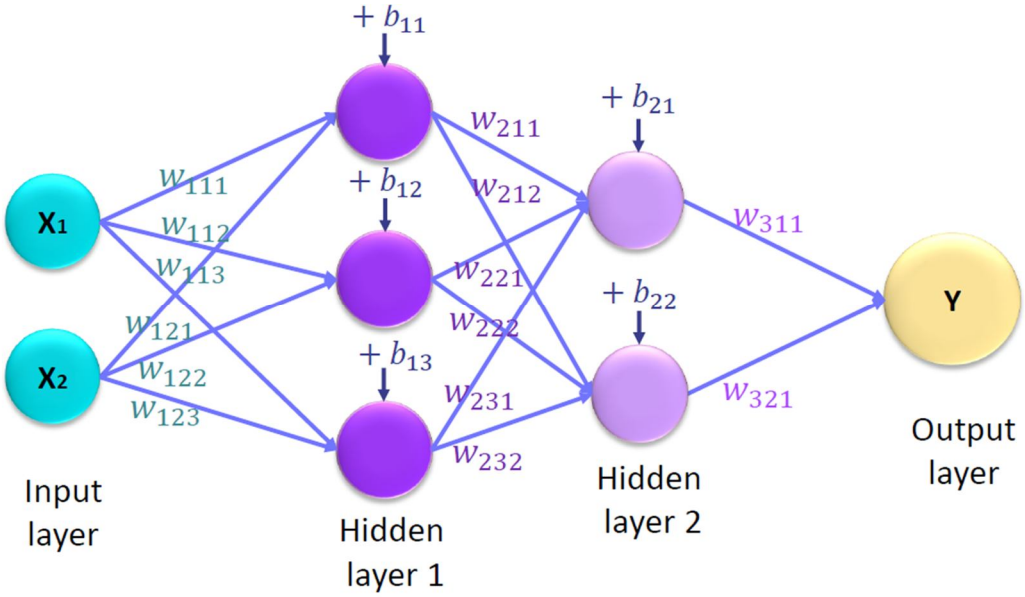
At this case still some data of cates with dog data



So we use biases to fit the given data best.



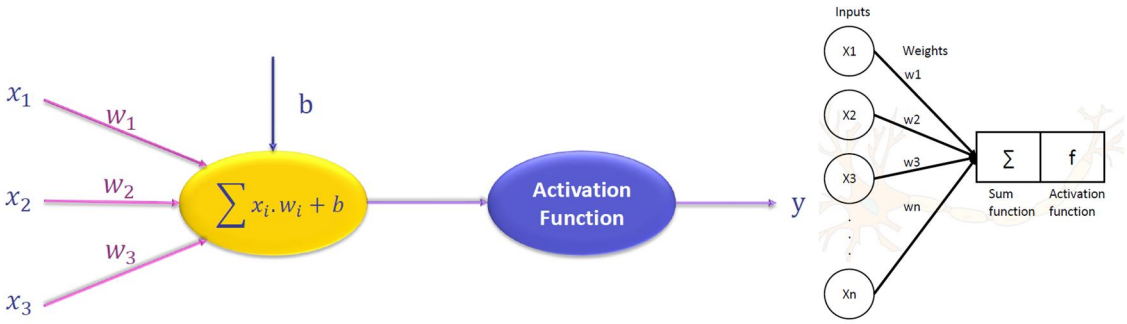
Parameters of the neural network is Weight (w) and biases (b)



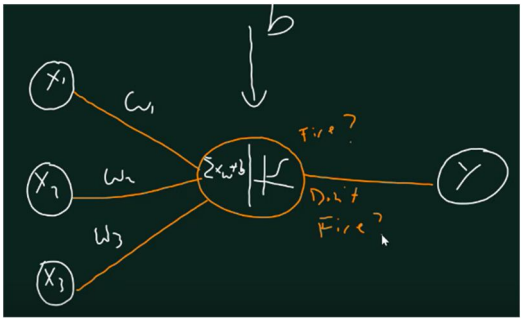
Activation Functions

What are Activation Functions?

Mathematical functions that determine the output of a neural network
Connected to each neuron and determine if it has to be activated or not, based on its value.



The activation function checks the value that it receives and determines the output of the neuron accordingly.



Why are activation functions useful?

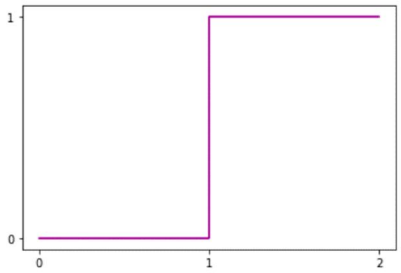
- Add non-linearities to neural networks.
- the neural network can learn to powerfully solve complex problems (ex. image recognition)

Without an activation function, the output of a neural network would be a linear relationship between the inputs and their weights.

- No input management would take place.
- No non-linearities will be introduced to solve complex problems in analogy with the human brain.

- Help normalize the output of every neuron to a range between 0 & 1 or -1 and 1

Common activation functions:

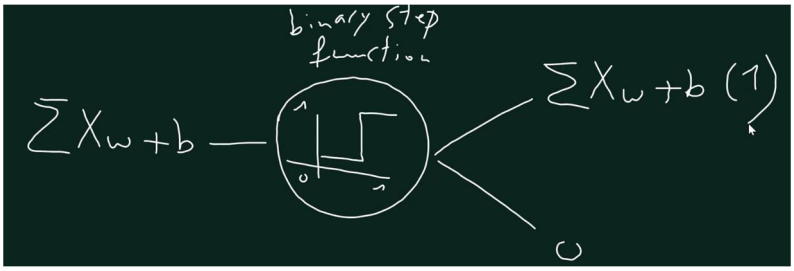


Binary Step Function

Threshold-based

Does not introduce non-linearity

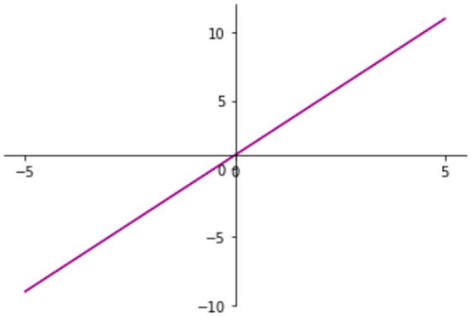
If input ≥ threshold,
Neuron is activated, and the input value is passed as is
Otherwise, the neuron is deactivated, and its output is 0



Linear Activation Function

The output is proportional to the input.

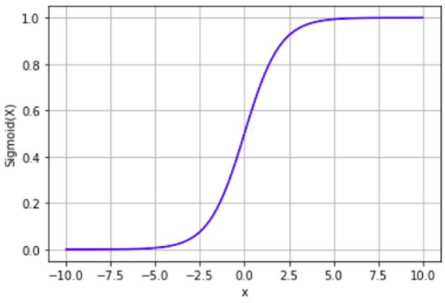
Does not introduce non-linearity



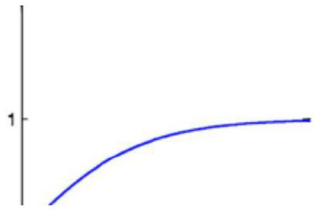
Sigmoid/ Logistic Function

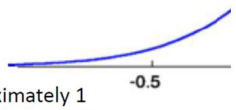
- Output values are between 0 and 1
- They can be interpreted as probability.
- Non-linear

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



$$y = \frac{1}{1 + e^{-x}}$$

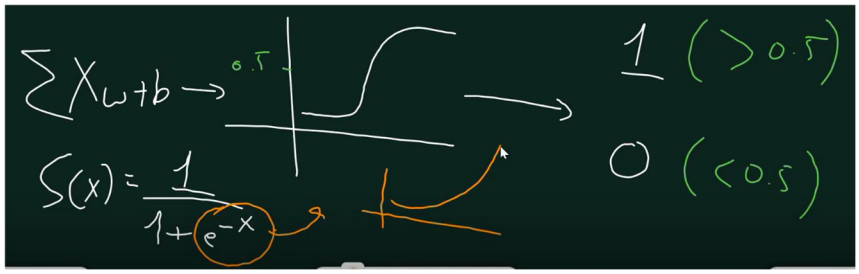




If X is high, the value is approximately 1
If X is small, the value is approximately 0

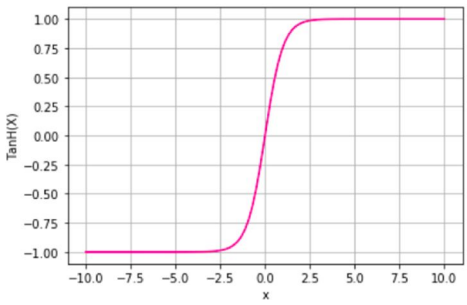
- Computationally expensive
- Vanishing gradients (to be seen later)

SIGMOID FUNCTION



TanH/ HyperbolicTangent Function

- Zero-centered
- Output values are between-1and 1
- Non-linear



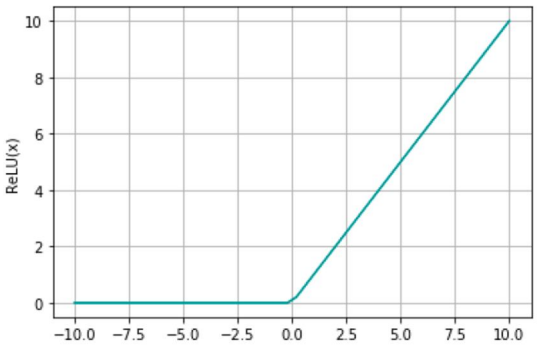
$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- Computationally expensive
- Vanishing gradients (to be seen later)

Vanishing gradients

Rectified Linear Unit -ReLU

$$ReLU(x) = \max(0, x)$$



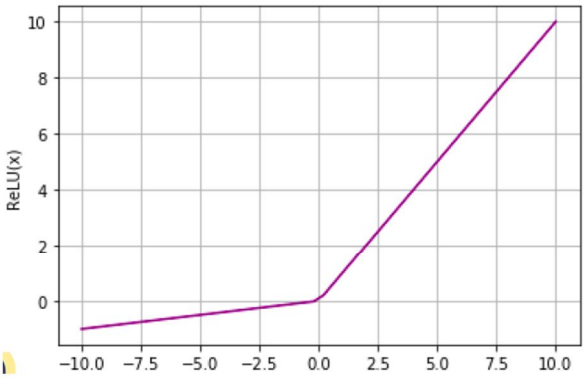
Returns 0 if input < 0 (this solve the issue of vanishing gradient) otherwise it keeps the values as it is.

- Returns the same value as the input if input > 0.
- Computationally efficient: allows the network to converge fast.
- Non-linear

The dying ReLU problem:

When the input is ≤ 0, the network cannot learn.

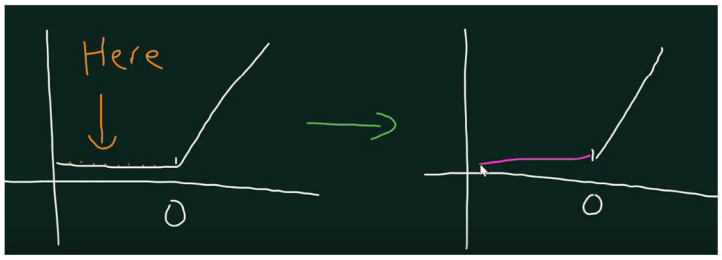
LeakyReLU



Prevents the dying ReLU problem has a small positive slope in the negative side so learning can be done

- Computationally efficient
- Non-linear

$$Leaky\_ReLU(x) = \max(0.1 * x, x)$$



Inconsistent predictions for negative input values

Parametric ReLU (PReLU)

Is the best one between the ReLU

Its leakage coefficient is a learned parameter.

Its slope is learnable.

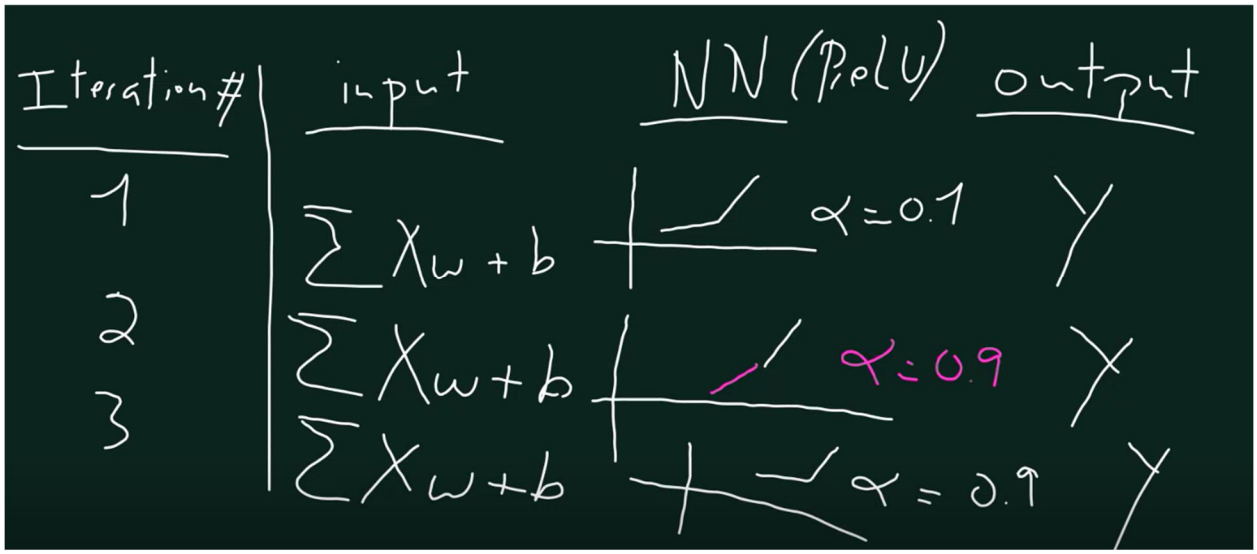
- Prevents the dying ReLU problem.
- Computationally efficient
- Non-linear

$$PReLU(x) = \begin{cases} x & if\ x > 0 \\ ax & otherwise \end{cases}$$

leakage coefficient

It will do 1st iteration, in the PReLU the coefficient alpha equal to 0.1, after first iteration it starts get inconsistent prediction. (-62, -60) when divided by 10 both will be same result (0.6) it might lead to classify 2 different categories to be same.

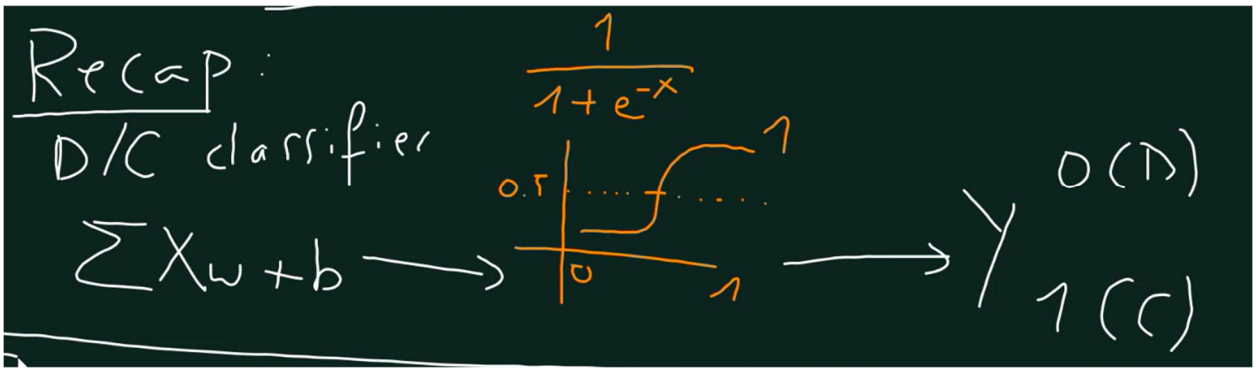
Then it will discover that issue with activation function, then it will change for example to be 0.9 (dynamic)



The issue is that it is adjust the activation function based on the data it solve to fit, so it will not be accurate if used the same with different data.

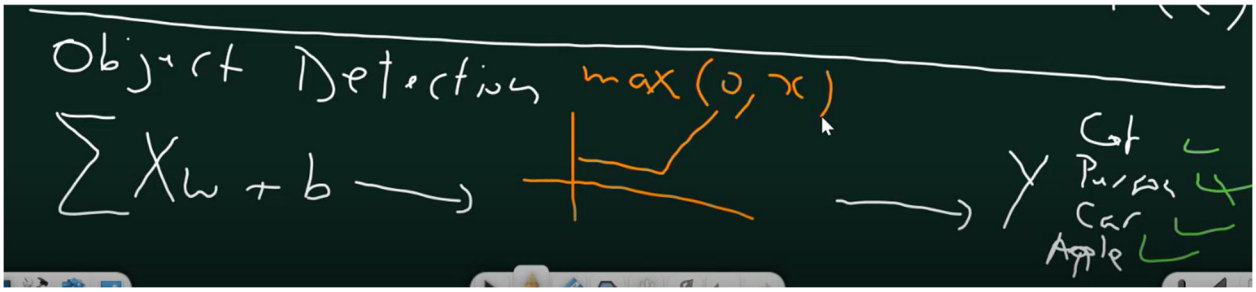
$$f(x) = \begin{cases} x & if\ x > 0 \\ ax & otherwise \end{cases}$$

Dog / or cat classifier (0 or 1), we decide to use sigmoid function



Another example:

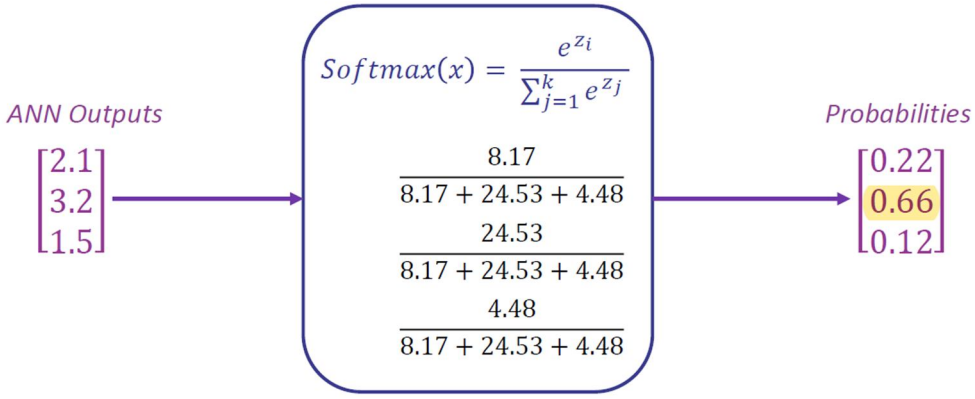
Object detection and the output categorize the image of the object, this will not work with sigmoid function using ReLu in this way.



Softmax *(like sigmoid but for multi classification)*

Able to handle multiple classes (normalizes the output of each class between 0 & 1→ classification probabilities)
The output probabilities sum to 1.
Non-linear
Typically used for the output layer

$$Softmax(x) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$



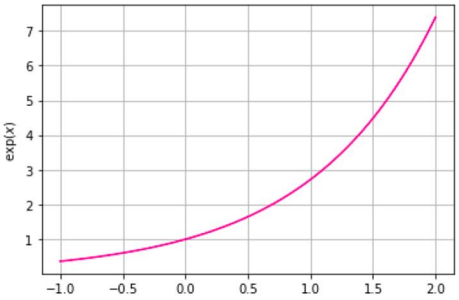
The exponential function makes high values even higher.

It makes sure that the probability of the most probable class stands out

$$e^0 = 1$$

$$e^2 = 7.4$$

$$e^4 = 54.6$$



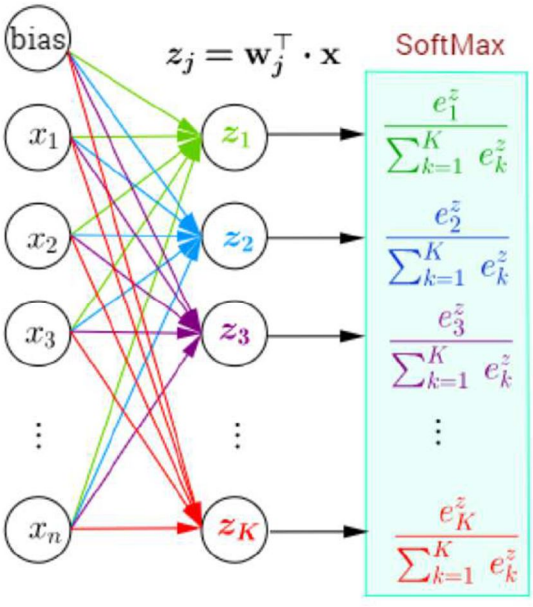
Normalisation without Softmax

ANN Outputs $\begin{bmatrix}2.1\\3.2\\1.5\end{bmatrix}$ → $\frac{2.1}{2.1+3.2+1.5}$, $\frac{3.2}{2.1+3.2+1.5}$, $\frac{1.5}{2.1+3.2+1.5}$ → *Probabilities* $\begin{bmatrix}0.31\\0.47\\0.22\end{bmatrix}$

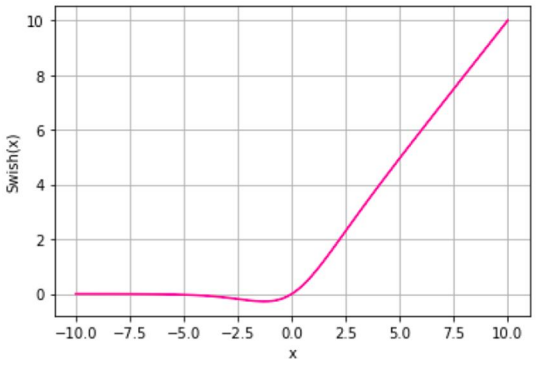
Normalisation with Softmax

ANN Outputs $\begin{bmatrix}2.1\\3.2\\1.5\end{bmatrix}$ → $Softmax(x) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$ → $\frac{8.17}{8.17+24.53+4.48}$, $\frac{24.53}{8.17+24.53+4.48}$, $\frac{4.48}{8.17+24.53+4.48}$ → *Probabilities* $\begin{bmatrix}0.22\\0.66\\0.12\end{bmatrix}$

$$Y = \frac{e(x)}{\sum e(x)}$$



Swish Activation Function



Discovered by researchers at Google Brain
it looks very much like ReLU and Leaky ReLU.
Same advantage of ReLU
but with better performance (more efficient)
Non-linear

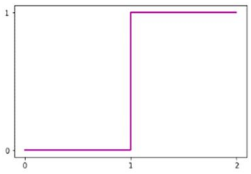
$$Swish(x) = \frac{x}{1 + e^{-x}}$$

Most used (ReLU, Sigmoid, SoftMax)

RECAP

Common activation functions

Binary Step Function



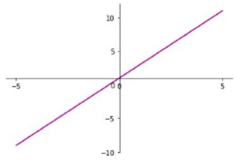
Threshold-based

If input ≥ threshold, neuron is activated,
and the input value is passed as is. Otherwise, the neuron is deactivated, and its output is 0

Does not introduce non-linearity

Linear Activation Function

The output is proportional to the input.



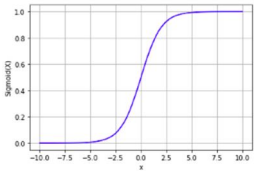
Does not introduce non-linearity

Sigmoid/ Logistic Function

Output values are between 0 and 1, Non-linear.

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

If X is high, the value is approximately 1
If X is small, the value is approximately 0

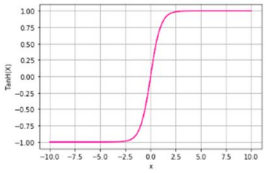


- Computationally expensive
- Vanishing gradients (to be seen later)

TanH/ HyperbolicTangent Function

Zero-centered, Output values are between-1and 1 , Non-linear

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

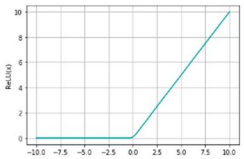


- Computationally expensive
- Vanishing gradients (to be seen later)

Rectified Linear Unit -ReLU

Returns 0 if input < 0, otherwise it keeps the values as it is , Non-linear

$$ReLU(x) = \max(0, x)$$

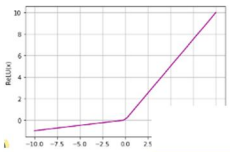


The dying ReLU problem:
When the input is ≤ 0, the network cannot learn

LeakyReLU

Prevents the dying ReLU problem.
has a small positive slope in the negative side
so learning can be done, Non-linear

$$Leaky\_ReLU(x) = \max(0.1 * x, x)$$



Inconsistent predictions for negative input values

Parametric ReLU (PReLU)

Is the best one between the ReLU
Its leakage coefficient is a learned parameter (changed dynamically).
Its slope is learnable, Non-linear
Prevents the dying ReLU problem.

$$PReLU(x) = \begin{cases} x & if\ x > 0 \\ ax & otherwise \end{cases}$$

leakage coefficient

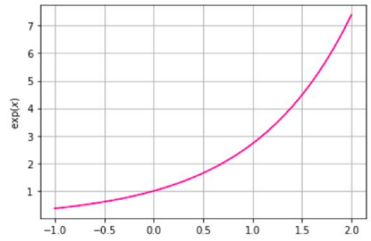
May perform differently in different problems

Computationally efficient

Softmax *(like sigmoid but for multi classification)*

Able to handle multiple classes (normalizes the output of each class between 0 & 1 classification probabilities)

$$Softmax(x) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

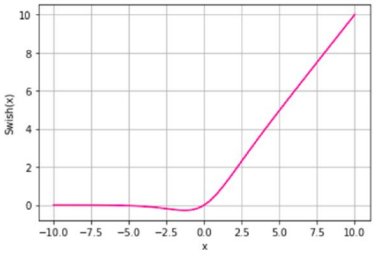


The exponential function makes high values even higher

Swish Activation Function

Google Brain, Non-linear

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



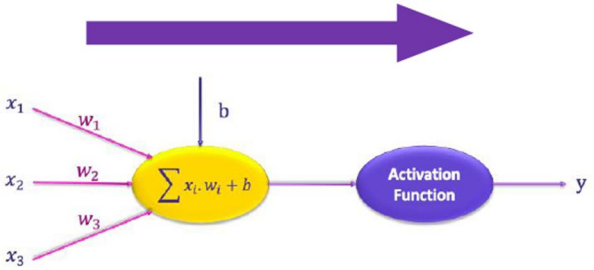
Outperforms ReLU for deep networks.

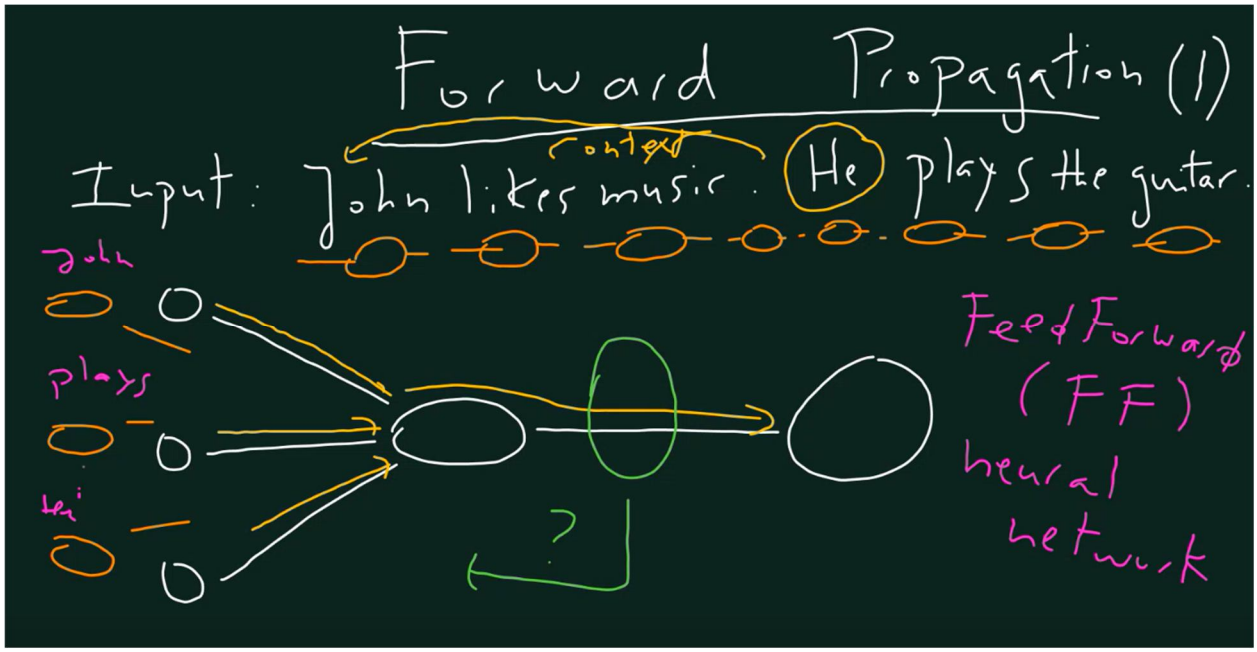
Forward Propagation

Propagation means the direction of learning.

What are Feedforward Neural Networks?

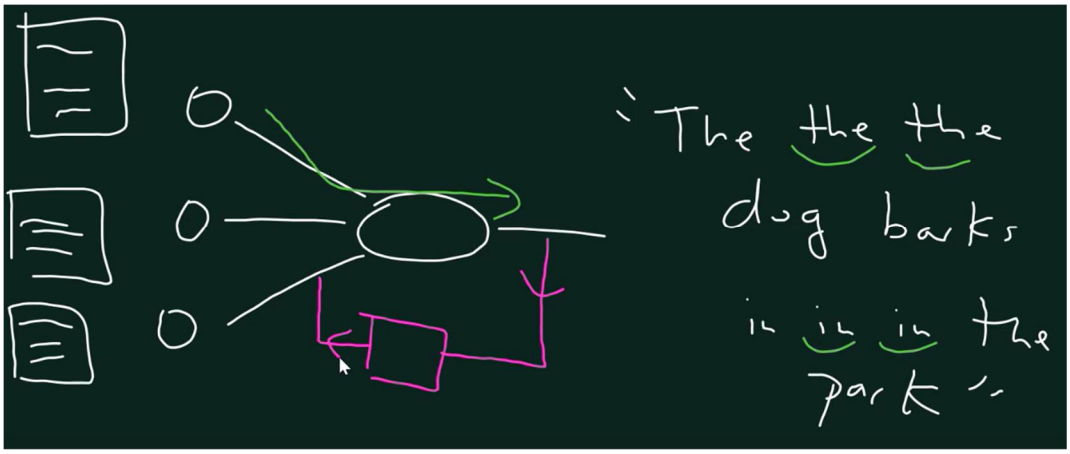
- ANNs where the input data flows in the forward direction through the network.
- The data does not flow backwards during output generation (no memory).
- No cycles or loops exist in this network.
- Feedforward Neural Networks support forward propagation.





It will not know what He refers to.

You are giving a text, novel and ask the NN to generate a text, or give a sentence and ask to complete it.



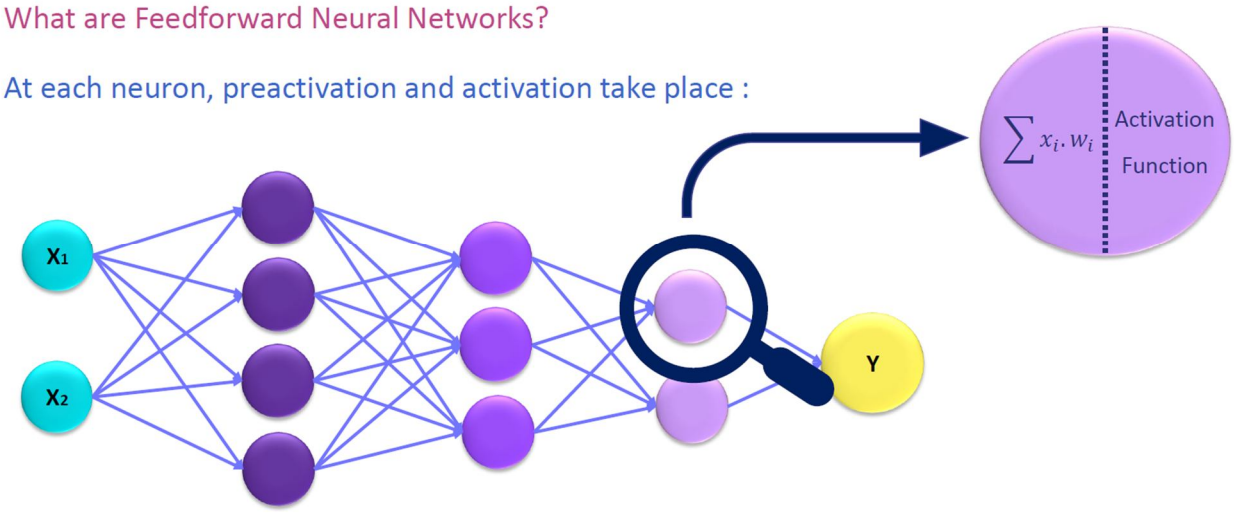
Because these stop words are commonly shown in the given text which cause to give it more weights from the activation function, they will have higher probability.

So, researchers fix that they said to add Memory or the NN to remember the previous data so that they can learn the context of the sentences. They added a memory gate before the output (LSTM: Long Short-Term Memory) Neron to go back and look at the previous input. (RNN: Recurrent Neural Networks)

That is commonly used with translation, chatbot.

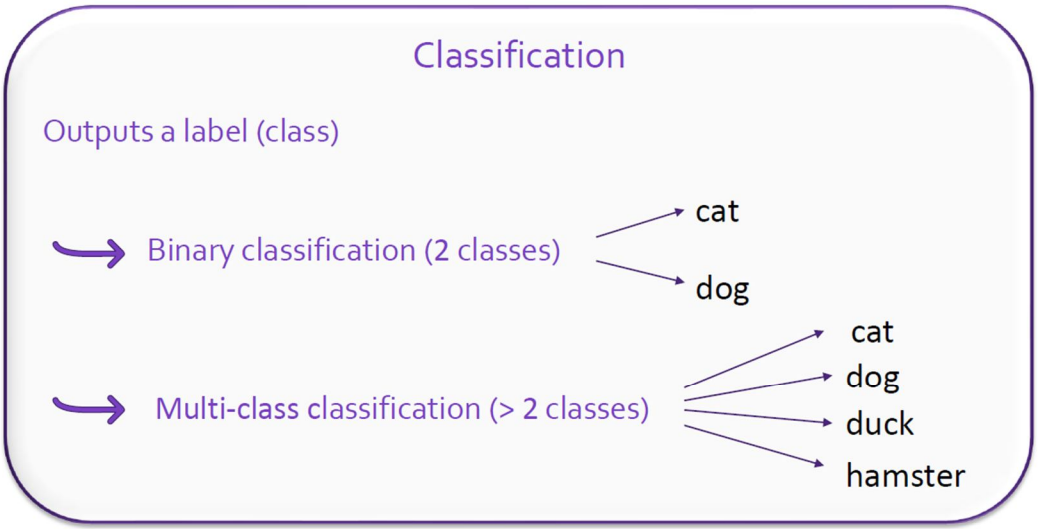
At each neuron, two steps take place:

1. Preactivation: The weighted inputs are summed up.
2. Activation: The weighted sum of inputs is passed to the activation function.



Classification and Regression

What is the difference between classification and regression?



Regression

Predicts a quantity based on the learned data, such as :

- House price
- Weight
- Pandemic contamination rate

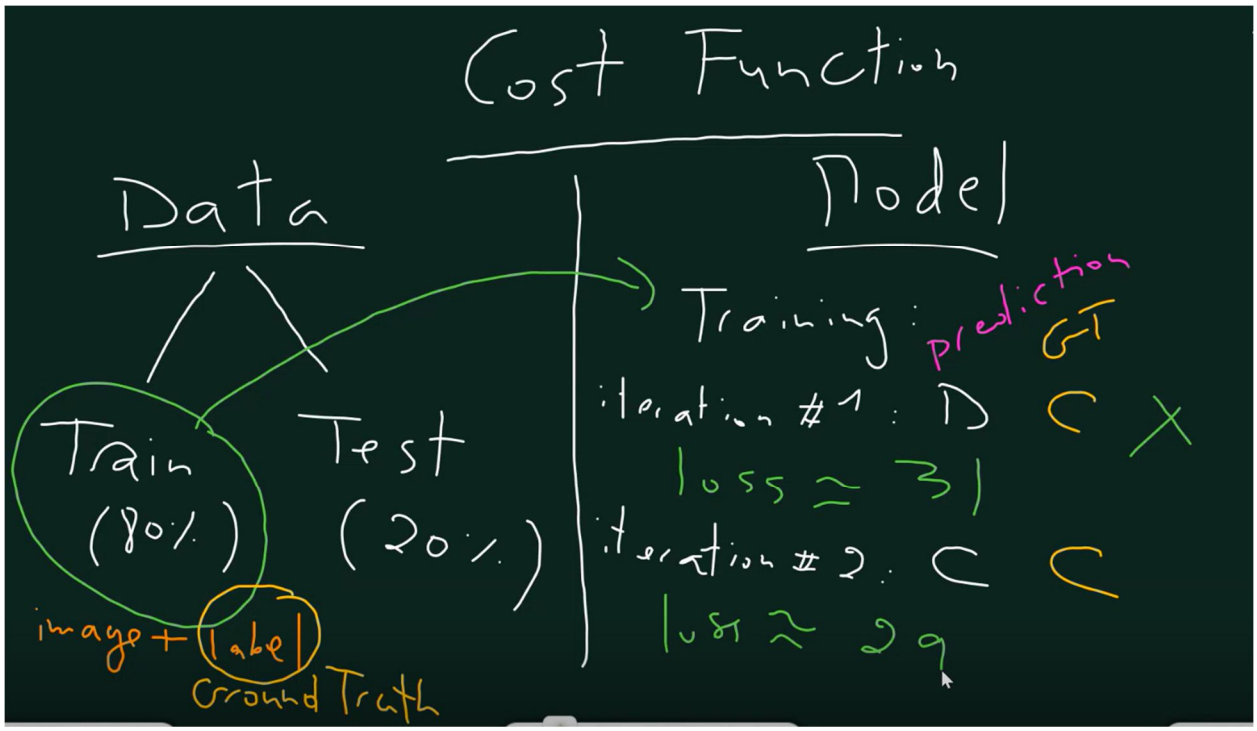
Cost Function

What is the cost/ loss function?

The cost function measures the error between the true value and the predicted value by the trained model.

- It evaluates how well the trained model performs.
- If performance is bad, the cost function yields a high number.
- The better the performance, the lower the cost function result
- During training, the cost function result must decrease. Otherwise, this means that our model is not learning!
- Our aim is therefore to MINIMIZE the Cost function.

(Supervised learning always).



The model train 80% of the data then calculates the lose by prediction – actual value then updates the weight to minimize the lose value and get more accurate prediction.

There is no optimal value for the lose function it should be close to zero

Regression Loss:

(Cost/loss function) called Mean Squared Error:

Performs direct comparisons between the true value and the model's output value.

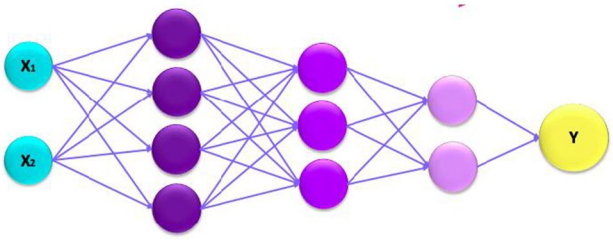
n = Total number of data points

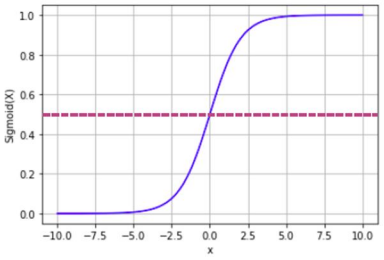
$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_{true_i} - y_{predicted_i})^2$$

Classification Losses

Binary Classification Loss

Binary classification has 1 output node giving a probability (using the Sigmoid Activation Function)





Probability = 0.5

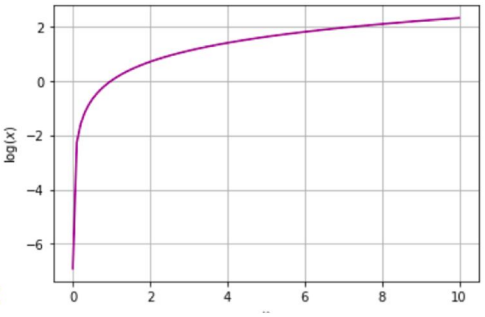
If probability ≥ 0.5 → label 1 ('*dog*')

If probability < 0.5 → label 0 ('*cat*')

Let's review the Natural Logarithm (also known as log to the base e or ln)

As x approaches 0, Ln(x) approaches -∞

Ln(1) = 0
Ln(0) = -∞ >>> -Ln(0)= +∞

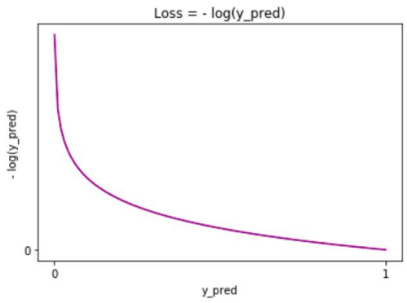


Decreases very fast, ln(1)=0 and at some pint

Binary Cross Entropy (BCE): loss function used for binary classification Loss.

Note: Binary Cross Entropy Loss is also known as Log Loss (because its based on logarithmic function)

If we minus the maximum it will be minimum so
By changing the loss to be minus which will flip the chart



when we need to predict the class with label 1 ($y_{true} = 1$)
We use: $the\ loss\ for\ class\ 1$

$$Loss = -log(y_{pred})$$

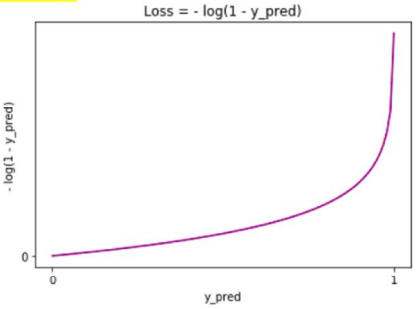
Example:

y_{true}	y_{pred}
1	0.92
1	0.09

For $y_{pred} = 0.92$ ➔ Loss = - log(0.92) = 0.083 ✓

For $y_{pred} = 0.09$ ➔ Loss = - log(0.09) = 2.41

Used when we need to predict the class with label 0, ($y_{true} = 0$)
We use: $the\ loss\ for\ class\ 0$



$$Loss = -log(1 - y_{pred})$$

Example:

y_{true}	y_{pred}
0	0.03
0	0.94

For $y_{pred} = 0.03$ ➔ Loss = - log(1 – 0.03) = 0.03 ✓

For $y_{pred} = 0.94$ ➔ Loss = - log(1 – 0.94) = 2.81

The loss function used for binary classification is the **Binary Cross Entropy (BCE)**
The complete mathematical representation of the BCE loss function:

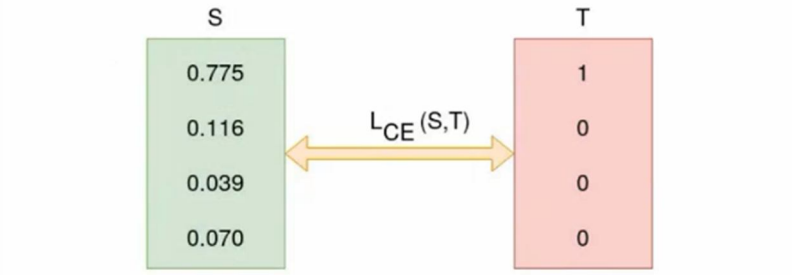
$BCE\ Loss =$ $(the\ loss\ for\ class\ 1)as\ weigh\ X\ (y\ true) +$
$(the\ loss\ for\ class\ 0)as\ weight\ X\ (1 - y\ true)$

$$BCE\ Loss = y_{true}(-log(y_{pred})) + (1 - y_{true})(-log(1 - y_{pred}))$$

If y true =1 and y predict is 1 >> =1(-log (1)) +0(-log (0) = 0
If y true =1 and y predict is 0 >> =1(-log (0)) +0(-log (1) = big number
If y true =0 and y predict is 0 >> =0+1(-log (1)) = 0
If y true =0 and y predict is 1 >> =0+1(-log (0)) = big number

Multiclass Classification Loss

Categorical Cross Entropy (CCE): Loss function used for multiclass classification.
Multiclass classification predicts 1 possible class out of several.



Logits(S) and one-hot encoded truth label(T) with Categorical Cross-Entropy loss function used to measure the 'distance' between the predicted probabilities and the truth labels. (Source: Author)

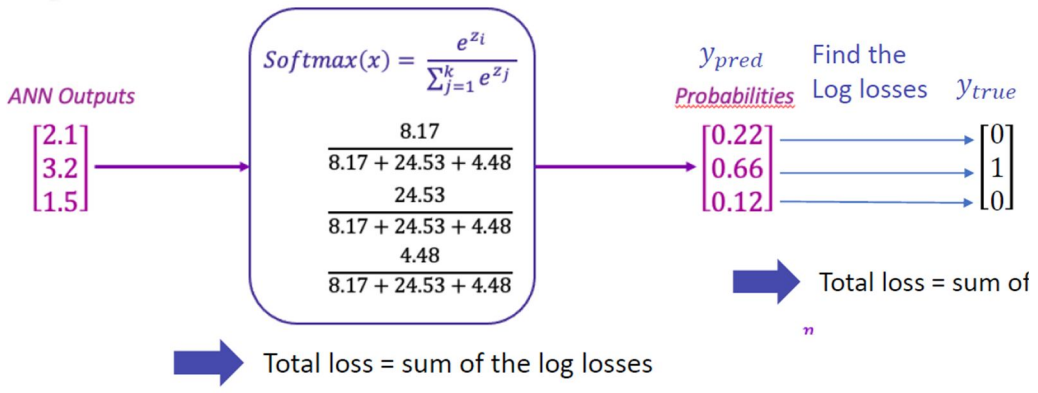
The categorical cross-entropy is computed as follows

$$
\begin{aligned}
L_{CE} &= -\sum_{i=1} T_i \log(S_i) \\
&= -\left[1\log_2(0.775) + 0\log_2(0.126) + 0\log_2(0.039) + 0\log_2(0.070)\right] \\
&= -\log_2(0.775) \\
&= 0.3677
\end{aligned}
$$

Adding a sigmoid activation function to every output would give a probability to every class, but the sum of those probabilities would not add up to 1!

We use SoftMax activation function.

Never use sigmoid function in a multiclass classification but use SoftMax instead.
Categorical Cross Entropy (CCE): Loss function used for multiclass classification.
Multiclass classification predicts 1 possible class out of several.

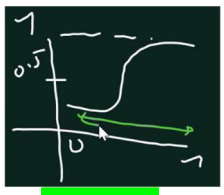


$$Loss = \sum_{i}^{n} y_{true_i}(-\log(y_{pred_i})) + (1 - y_{true_i})(-\log(1 - y_{pred_i}))$$

Multi label classification can predict several classes at the same time.

Example: recognize the fruits in a fruit basket

We **cannot use SoftMax** because it forces to have one best class.
We add **sigmoid** to every output node to predict the unique probability of each class
To calculate the loss, we do the same as with multi class classification: calculate the log loss for every node then sum the losses up

Binary Class	Multi Class	Multi Label
Cat 0 Dog 1  We use sigmoid function. $S(x) = \frac{1}{1+e^{-x}}$ It gives one probability. It will give a number between 0 and 1 which is probability like 0.75 closed to 1 then the output will be Dog	Cat 0.3 Dog 0.6 Bird 0.1 We used SoftMax. $S(x) = \frac{e^{z_i}}{\sum_j^k e^{z_j}}$ It will give a list of probabilities	We have to identify multiple classes like fruit basket, and we want to identify all the classes exist inside it Apple 0.70 (exist) Orange 0.55 (exist) Banana 0.4 (not exist) Grape 0.1 (not exist) $S(x) = \frac{1}{1+e^{-x}}$ Will use the Sigmoid not SoftMax function. We add **sigmoid for each neuron, 4 times one for each class** and output 4 different probabilities not related to each other. Then as per the threshold will consider the probability like more than 0.5, and total probabilities greater than 1 which is normal.

Introduction to TensorFlow 2

What is TensorFlow?

TensorFlow is an **end-to-end open-source platform** developed by Google for machine learning. https://www.tensorflow.org/

What is Keras?

Keras is a high-level deep learning API (Application Programming Interface) running on top of TensorFlow.

It is an extension to TensorFlow that makes machine learning more user-friendly.
https://keras.io/

What is Google Colab?

Google is a web IDE (Integrated Development Environment) for Python:

- Enables Machine Learning operations and storage on the cloud.
- Is a Jupyter notebook environment that requires no setup.
- Enables access to Google drive.
- Allows using code, text, and images.
- Enables connection to GPU

https://colab.research.google.com/notebooks/mlcc/intro_to_neural_nets.ipynb

Other libraries:

NumPy: Scientific computing package
https://numpy.org/

Scikit-Learn : Machine Learning and data analysis built on NumPy, SciPy, and Matplotlib
https://scikit-learn.org/stable/

Matplotlib: Static, animated, and interactive visualizations in Python
https://matplotlib.org/stable/index.html

Datasets

MNIST: Handwritten digits
http://yann.lecun.com/exdb/mnist/

COCO: Common Objects in Context

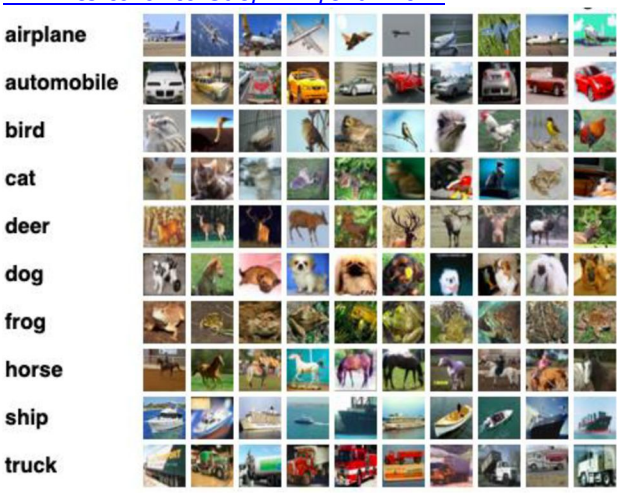
https://cocodataset.org/#overview

ImageNet: Image dataset of different classes

www.image-net.org

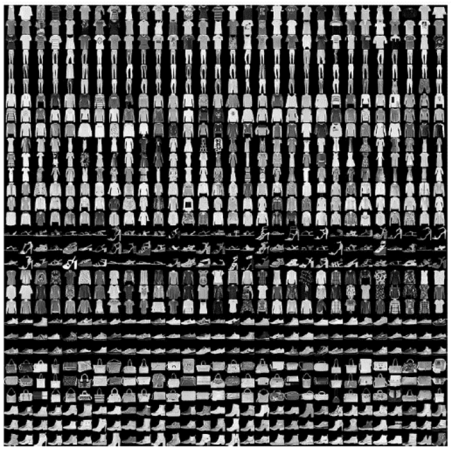
CIFAR-10 and CIFAR-100:
Image datasets of different classes

www.cs.toronto.edu/~kriz/cifar.html



FASHION-MNIST:
Image datasets of Zalando's article images

https://github.com/zalandoresearch/fashion-mnist



Wine Dataset: Wine quality classification based on features like color, alcohol level, etc.
https://scikit-learn.org/0.21/modules/generated/sklearn.datasets.load_wine.html

Dataset scan be found on:

Kaggle
https://www.kaggle.com/datasets

Google Dataset Search
https://datasetsearch.research.google.com/

Keras
https://keras.io/api/datasets/

Best Public Datasets for Machine Learning and Data Science
https://towardsai.net/p/machine-learning/best-datasets-for-machine-learning-and-data-science-d80e9f030279

First Experience with Keras:

- The core data structures of Keras are models and layers.
- The Sequential model is the simplest model type --> a linear stack of layers.

```
from tensorflow.keras.models import Sequential

model = Sequential()
```

- To stack the layers, we use the add() method

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()

model.add(Dense(units=64, activation='relu'))
model.add(Dense(units=10, activation='softmax'))
```

'Dense' refers to a fully-connected layer

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
model=Sequential()
# adding one layer of 64 neuron
model.add(Dense(units=64, activation='relu'))
# adding one layer of 10 neuron
model.add(Dense(units=10, activation='softmax'))
```

Once the model configuration is done, we configure the training/learning process. This is done using the *compile()* method

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

```
model.compile(loss='categorical_crossentropy', optimizer='sgd',
metrics=['accuracy'])
```

- The compile() method allows us to configure the learning further
- Here, we configure the optimizer by specifying the learning rate and the momentum

```
import keras
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.SGD(learning_rate=0.01, momentum=0.9))
```

```
import keras
model.compile(loss=keras.losses.categorical_crossentropy,
optimizer=keras.optimizers.SGD(learning_rate=0.01, momentum=0.9))
```

- Train the model using the fit() method :

```
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

```
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Notes:

- x_train = training data
- y_train = training labels
- Evaluate model performance using the evaluate() method :

```
loss, accuracy = model.evaluate(x_test, y_test)
```

```
loss, accuracy = model.evaluate(x_test, y_test)
```

Notes:

- x_test = test/evaluation data
- y_test = test/evaluation labels

- Test on new data using the predict()method :

```
classes = model.predict(x_test)
```

```
classes = model.predict(x_test)
```

Data Loading:

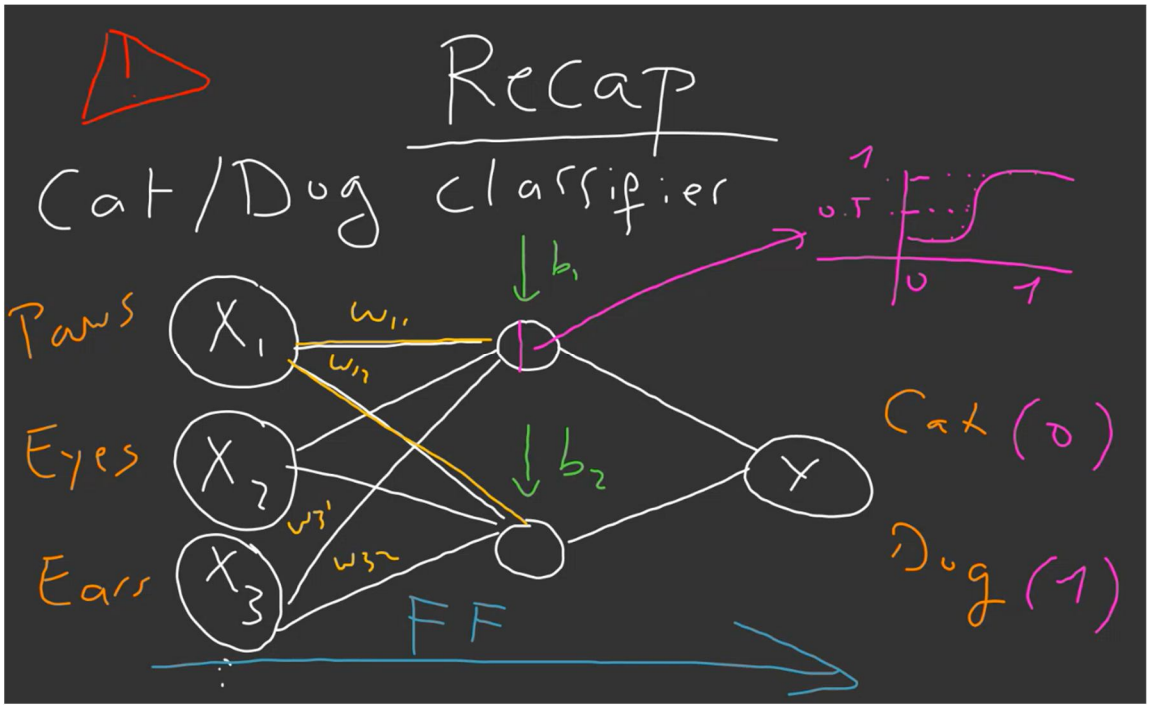
1. Dataset importation:

```
import pandas as pd
from sklearn.datasets import load_wine
wine_data = load_wine()

data_frame = pd.DataFrame(wine_data.data[:, :], columns=wine_data.feature_names[:])
data_frame['label'] = wine_data.target
data_frame
```

2. Dataset loading from .csv file :

```
import pandas as pd

dataset = pd.read_csv('Churn_Modelling.csv')
#Kaggle database (source: https://www.kaggle.com/aakash50897/churn-modellingcsv?select=Churn_Modelling.csv)
dataset
```

https://scikit-learn.org/stable/

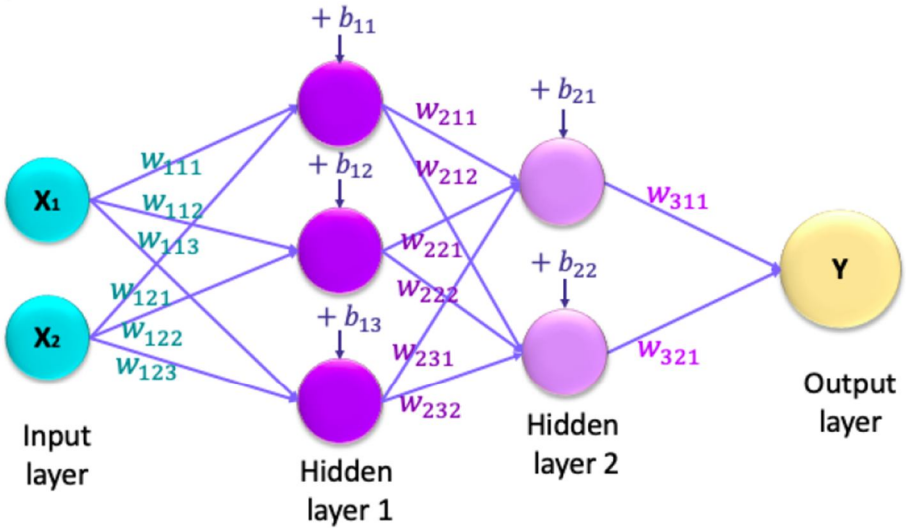

Recap
Cat/Dog classifier
Paws
Eyes
Ears
X_1
X_2
X_3
w_{11}
w_{12}
w_{31}
w_{32}
b_1
b_2
Y
FF
Cat (0)
Dog (1)

Optimizers

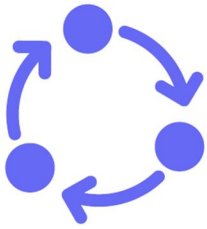
What are optimizers?

Optimizers (optimization algorithms), minimize the loss function by finding the most accurate model parameters possible.

Model parameters correspond to weights and biases.



Training a model is an iterative process.



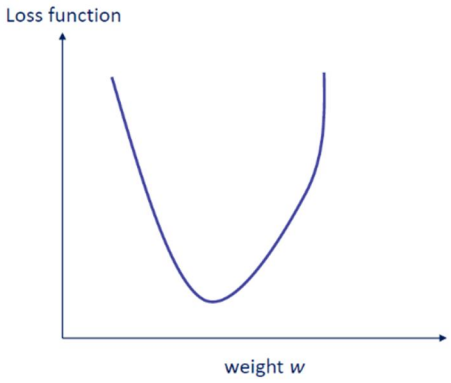
It is essential for the optimizers to be:

- Fast,
- Accurate

Optimizers example:

- Gradient Descent.
- Stochastic Gradient Descent.
- Mini Batch Gradient Descent.
- Gradient Descent with Momentum.
- Adagrad.
- RMS Prop.
- Adam.

1- Gradient Descent



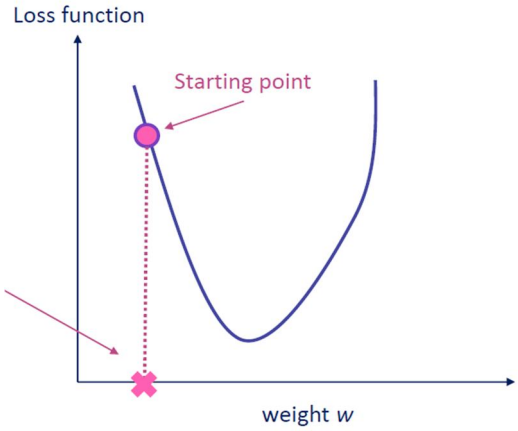
One way of measuring the change is using the derivative and gradient to be able to optimize a function. (A more efficient method is the Gradient Descent)

Calculating the loss for every possible value of w is an exhaustive and an inefficient way to find the minimum loss.

Some cases the function will have many minima, the minimum will not be clear when we are doing it manually, will never finish, might thought the first one is the minimum. Here is Gradient descent coming.

- An optimization algorithm that finds the minimum value of a function by taking steps starting from an initial value until it finds the best value.

- It takes big steps towards the minimum if far from the optimum value and smaller steps as it gets closer.

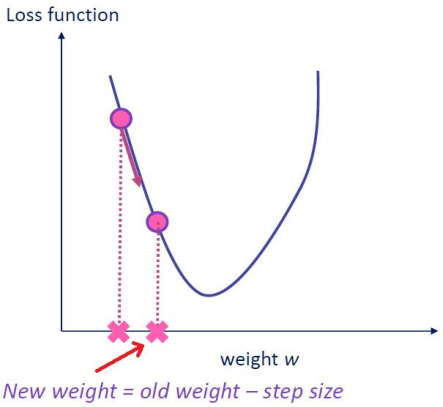


Step 1: We set a starting value for weight w (this value can be randomly chosen)

Step 2: The Gradient Descent algorithm calculates the gradient of the loss function (derivative) at the starting point.

The gradient tells which direction allows the cost function to decrease.

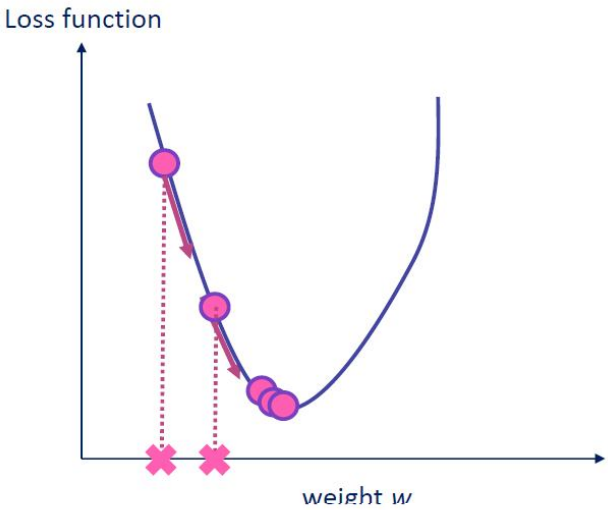
Step 3: The algorithm takes a step in the negative gradient direction (note: the gradient always points towards the direction of stepest increase >> we use the negative gradient)



Step size = gradient magnitude * learning rate
New weight = old weight - step size

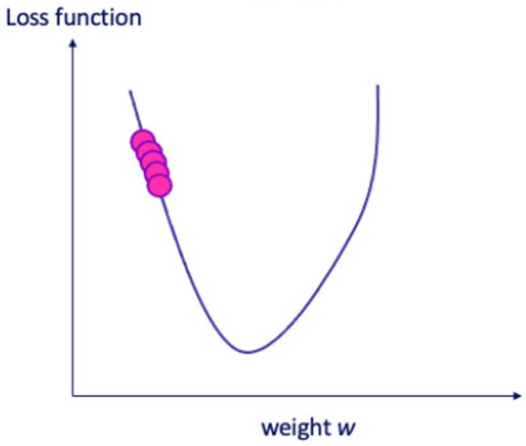
The Gradient Descent algorithm repeats these steps until:

- Step size gets very close to 0 (ex. min step size = 0.001 or smaller
- Maximum number of steps is reached (ex. 1000 or more)

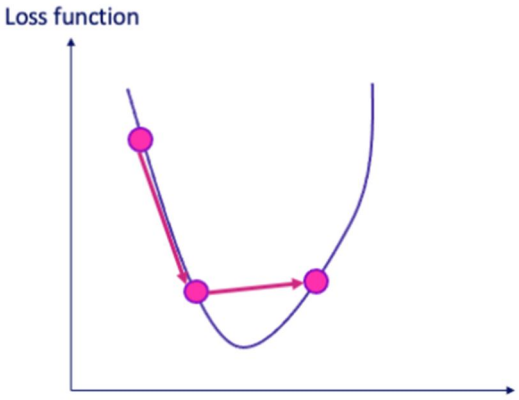


Learning rate:
A scalar that determines the step size

If learning rate is very small -> learning can take too long

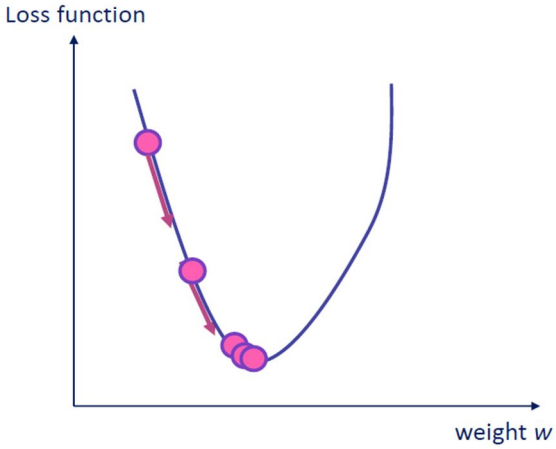


If learning is very large -> the point will perpetually bounce across the curve minimum



Ideal learning rate:

- Larger when the point is far from the minimum &
- Smaller as it gets closer.



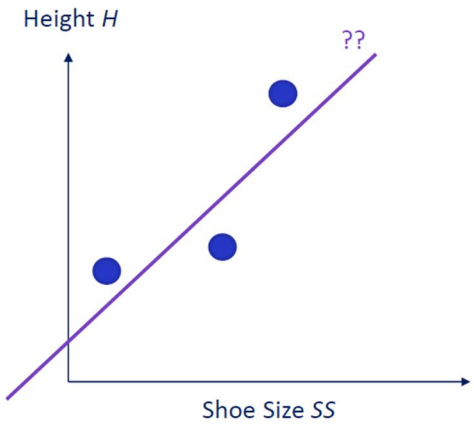
Gradient Descent is very sensitive to learning rate.

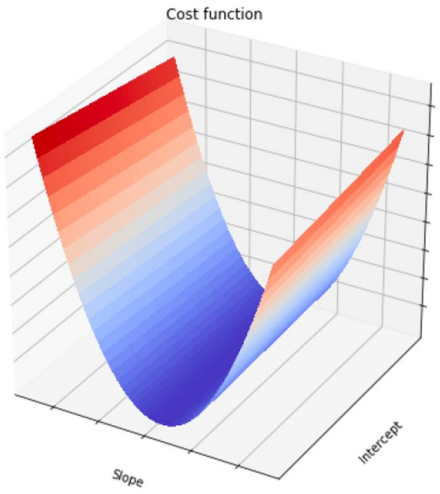
In practice, the learning rate can be determined automatically during training (starts large and diminishes gradually).
This strategy is called schedule.

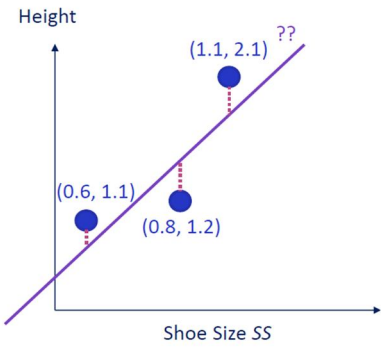
Example: How does Gradient Descent fit a line to data?

Use Gradient Descent to estimate the optimal values for the slope and the intercept:

$$H_{pred} = slope\ SS + intercept$$







$$H_{pred} = slope\ .SS + intercept$$

Y = (slope.ss) as X.w + (intercept) as b

$$Lose\ function = \ Sum\ of\ Squared\ Residuals\ (SSR)$$

$$= (H_{true1} - \ H_{pred1})^2 + (H_{true2} - \ H_{pred2})^2 + (H_{true3} - \ H_{pred3})^2$$

$$= (1.1. - (slope * 0.6 + intercept))^2 + (1.2 - \ (slope * 0.8 + intercept))^2 + (2.1 - \ (slope * 1.1 + intercept))^2$$

To calculate the gradient with respect to each of the parameters
We need to find the partial derivative of the loss function with respect to each parameters the slop and intercept

$$\frac{\partial(Loss\ function)}{\partial(intercept)}\ ,\ \frac{\partial(Loss\ function)}{\partial(slop)}$$

We use the Chain Rule

Loss function = Sum of Squared residuals (SSR)

$$= \overbrace{(1.1 - (slope * 0.6 + intercept))}^{U}{}^2$$
$$+ (1.2 - (slope * 0.8 + intercept))^2$$
$$+ (2.1 - (slope * 1.1 + intercept))^2$$

$$\frac{\partial(Loss\ function)}{\partial(intercept)} = \frac{\partial(Loss\ function)}{\partial(U)}\ x\ \frac{\partial(U)}{\partial(intercept)}$$

$$= 2U_1\ x\ (-1) + 2U_2\ x\ (-1) + 2U_3\ x\ (-1)$$

$$= -2\ (1.1 - (0.6 * \text{slope} + \text{intercept}))$$
$$-2\ (1.2 - (0.8 * \text{slope} + \text{intercept}))$$
$$-2\ (2.1 - (1.1 * \text{slope} + \text{intercept}))$$

$$\frac{\partial(Loss\ function)}{\partial(slope)} = \frac{\partial(Loss\ function)}{\partial(U)}\ x\ \frac{\partial(U)}{\partial(slope)}$$

$$= 2U_1\ x\ (-0.6) + 2U_2\ x\ (-0.8) + 2U_3\ x\ (-1.1)$$

$$= -1.2\ (1.1 - (0.6 * \text{slope} + \text{intercept}))$$
$$-1.6\ (1.2 - (0.8 * \text{slope} + \text{intercept}))$$
$$-2.2\ (2.1 - (1.1 * \text{slope} + \text{intercept}))$$

Note: The partial derivatives of a multi variate function are stored in a vector called the **Gradient** (∇)

$$\nabla\ Loss\ Function = \begin{bmatrix} \frac{\partial(Loss\ function)}{\partial(slope)} \\ \frac{\partial(Loss\ function)}{\partial(intercept)} \end{bmatrix}$$

We can now use this Gradient to descend to the minimal point in the cost function:

Step 1 : We choose a random intercept (= 0) and a random slope (=1)
Step 2 : We plug the values in the partial derivative formulas and get 2 values
Step 3 :

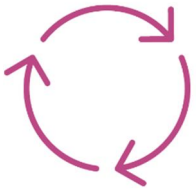
$$\text{stepsize}_{slope} = value_1 * learning\ rate$$
$$\text{stepsize}_{intercept} = value_2 * learning\ rate$$

Step 4 :

$$\text{slope}_{new} = \text{slope}_{old} - \text{stepsize}_{slope}$$
$$\text{intercept}_{new} = \text{intercept}_{old} - \text{stepsize}_{intercept}$$

We can now use this Gradient to descend to the minimal point in the cost function (hence the name Gradient Descent) :

Repeat steps 1 to 4 until:



✓The step sizes are very small.
✓the maximum number of steps is reached.

Note: In the case where we have more parameters to estimate (weights and biases of our ANN), we only use more derivatives, while the whole procedure remains the same.

Note: Gradient Descent can be used with any loss function.

Conclusion :

Area	Number of rooms	Distance from train station	city	Price (label)
32	3	5	Amiens	60 000
25	1	8	Lille	70 000
80	4	10	Versailles	600 000
55	3	10	Nice	450 000

The data is entirely plugged into the Neural Network and the parameters are adjusted.

2- Stochastic Gradient Descent SGD Optimization:

Gradient Descent considers the whole dataset for each parameter estimation step

If we have thousands of parameters and millions of data points, repeating all the necessary steps to find the optimal parameters takes a huge amount of time.

- **uses 1 sample per step calculation** of new adjusted parameters.
- reduces the time taken for derivative calculation.

Area	Number of rooms	Distance from train station	city	Price (label)
32	3	5	Amiens	60 000
25	1	8	Lille	70 000
80	4	10	Versailles	600 000
55	3	10	Nice	450 000

The data is plugged in the Neural Network one sample at a time and the parameters are adjusted after every sample

The data is plugged into the Neural Network one sample at a time and the parameters are adjusted after every sample.

Notice the low memory requirement as compared to Gradient Descent:
- Estimates concerning previous data do not have to be conserved in memory.
- We can only store the very last estimates, and use them when new data is added

Note:
- Sensitivity to learning rate also applies to SGD
- The same schedule' strategy is used: Starting with large numbers and reducing them gradually

SDG keep only one line in the memory.

3- Mini-Batch Gradient Descent

- uses a small subset of data (mini batch) for each step

- gives more stable results than using one sample per step Stochastic Gradient Descent
- faster than using the whole dataset (Gradient Descent)

mini-batch

Area	Number of rooms	Distance from train station	city	Price (label)
32	3	5	Amiens	60 000
25	1	8	Lille	70 000
80	4	10	Versailles	600 000
55	3	10	Nice	450 000
Etc.	Etc.	Etc.	Etc.	Etc.

4- Gradient Descent with Momentum

Gradient Descent might make a lot of steps and keep on oscillating very slowly towards the minimum of the lost function

- uses the exponentially weighted average of the gradients to update the weights.
- Smoothes out the steps of the Gradient Descent because it takes into consideration.
- the average of the past parameters
- is faster than the regular Gradient Descent

1. On each iteration, we compute the partial derivatives while plugging in the initial slope and intercept values.

$$\underbrace{\frac{d(\text{Loss function})}{d(slope)}}_{value_{slope}} \;\&\; \underbrace{\frac{d(\text{Loss function})}{d(intercept)}}_{value_{intercept}}$$

2. Compute :

$$weighted\_average_{slope} = \beta\, weighted\_average_{slope} + (1-\beta) value_{slope}$$

$$weighted\_average_{intercept} = \beta\, weighted\_average_{intercept} + (1-\beta) value_{intercept}$$

3. Update :

$$slope_{new} = slope_{old} - (learning\_rate * weighted\_average_{slope})$$

$$intercept_{new} = intercept_{old} - (learning\_rate * weighted\_average_{intercept})$$

β is a parameter that controls the weighted average.
(common value = 0.9)
Initial weighted_averages = 0

5- AdaGrad (Adaptive Gradients):

AdaGrad is a technique to change the learning rate over time.

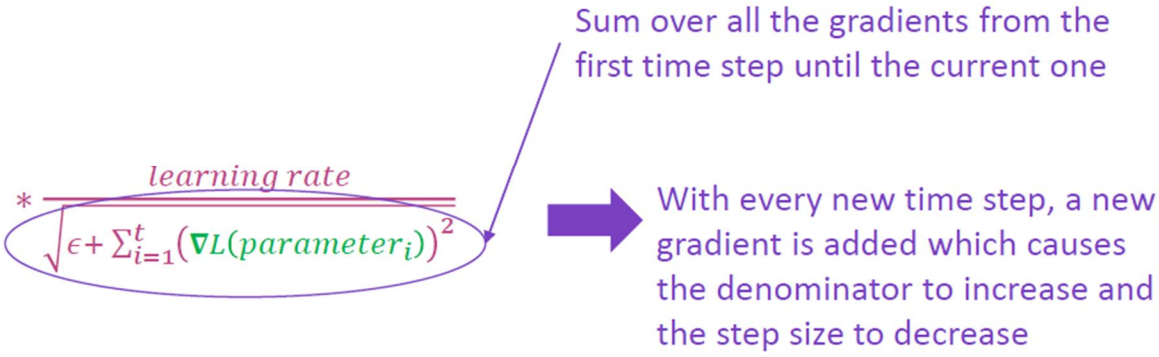
$$parameter_{new} = parameter_{old} - stepsize$$

Where

$$stepsize_{parameter_i} = \nabla L(parameter_i) * \frac{learning\ rate}{\sqrt{\epsilon + \sum_{i=1}^{t} (\nabla L(parameter_i))^2}}$$

and

$$\nabla L(parameter_i) = \frac{d(Loss\ function)}{d(parameter_i)}$$



ε A small value to avoid division by 0

β is a parameter that controls the weighted average (common value = 0.9)

Initial $weighted\_averages = 0$

RMS Prop (Root Mean Squared Propagation)

RMS Prop is very similar to AdaGrad

However, instead of using the sum of gradients, it uses the exponentially weighted average of the squared gradients.

Instead of being concerned about all the gradients, we are more concerned about the most recent gradients.

AdaGrad : Learning rate decreases monotonously.
RMS Prop: Learning rate can adapt to increase or decrease with every step

$$parameter_{new} = parameter_{old} - stepsize$$

Where

$$stepsize_{parameter_i} = \nabla L(parameter_i) * \frac{learning\ rate}{\sqrt{\epsilon + weighted\_average(\nabla L^2(parameter_i))}}$$

And

$$\nabla L(parameter_i) = \frac{d(Loss\ function)}{d(parameter_i)}$$

Default values :
Learning rate = 0.001
β = 0.9

Adam Optimizer (Adaptive Moment Estimation)

Adam is another optimizer with adaptive learning rates for each parameter.
Adam:

- uses the exponentially weighted average of the past squared gradients (like RMS Prop)
- uses an exponentially weighted average of past gradients (like GD with momentum)

$$parameter_{new} = parameter_{old} - stepsize$$

Where:

$$stepsize_{parameter_i} = weighted\_average(\nabla L(parameter_i)) * \frac{learning\ rate}{\sqrt{\epsilon + weighted\_average(\nabla L^2(parameter_i))}}$$

And

$$\nabla L(parameter_i) = \frac{d(Loss\ function)}{d(parameter_i)}$$

Default values :
$\beta_1 = 0.9$
$\beta_2 = 0.999$
$\epsilon = 10^{-8}$

However, as the weighted averages are initialized to 0, they are biased towards 0 during the first iterations.
bias-corrected moments are calculated and used.

$$\widehat{m_i} = \frac{m_i}{1-\beta_1}$$

$$\widehat{v_i} = \frac{v_i}{1-\beta_2}$$

$$m_i = \beta_1 m_{i-1} + (1-\beta_1)g_i \qquad w_{i+1} = w_i - \frac{\alpha \widehat{m_i}}{\epsilon + \sqrt{\widehat{v_i}}}$$

$$stepsize_{w_i} = \widehat{m_i} * \frac{learning\ rate}{\epsilon + \sqrt{\widehat{v_i}}} \qquad v_i = \beta_2 v_{i-1} + (1-\beta_2)g_i^2$$

Bias Correction in Exponentially Weighted Average

- Bias correction makes the computation of the exponentially weighted averages more accurate
- Initialization values are taken = 0

➔ $V_t = \beta V_{t-1} + (1-\beta)\theta_t$

if $V_0 = 0$ & $\beta = 0.9 \Rightarrow V_1 = 0.1\ \theta_1$ This is not a good estimate of the first moving average !

- The effect of these very low values remains for several iterations
- In order to correct this bias :

$$\widehat{V_t} = \frac{V_t}{1-\beta^t}$$ where t = current iteration

➔ $\widehat{V_1} = \frac{0.1\ \theta_1}{1-\beta}$

As t increases, β^t approaches 0

➔ The correction has less influence

How does Exponentially Weighted Average give more weight to the most recent observations?

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

$$V_{100} = 0.9V_{99} + 0.1\theta_{100}$$

$$V_{99} = 0.9V_{98} + 0.1\theta_{99}$$

$$V_{98} = 0.9V_{97} + 0.1\theta_{98}$$

➔ $V_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9(0.1\theta_{98} + 0.9V_{97}))$

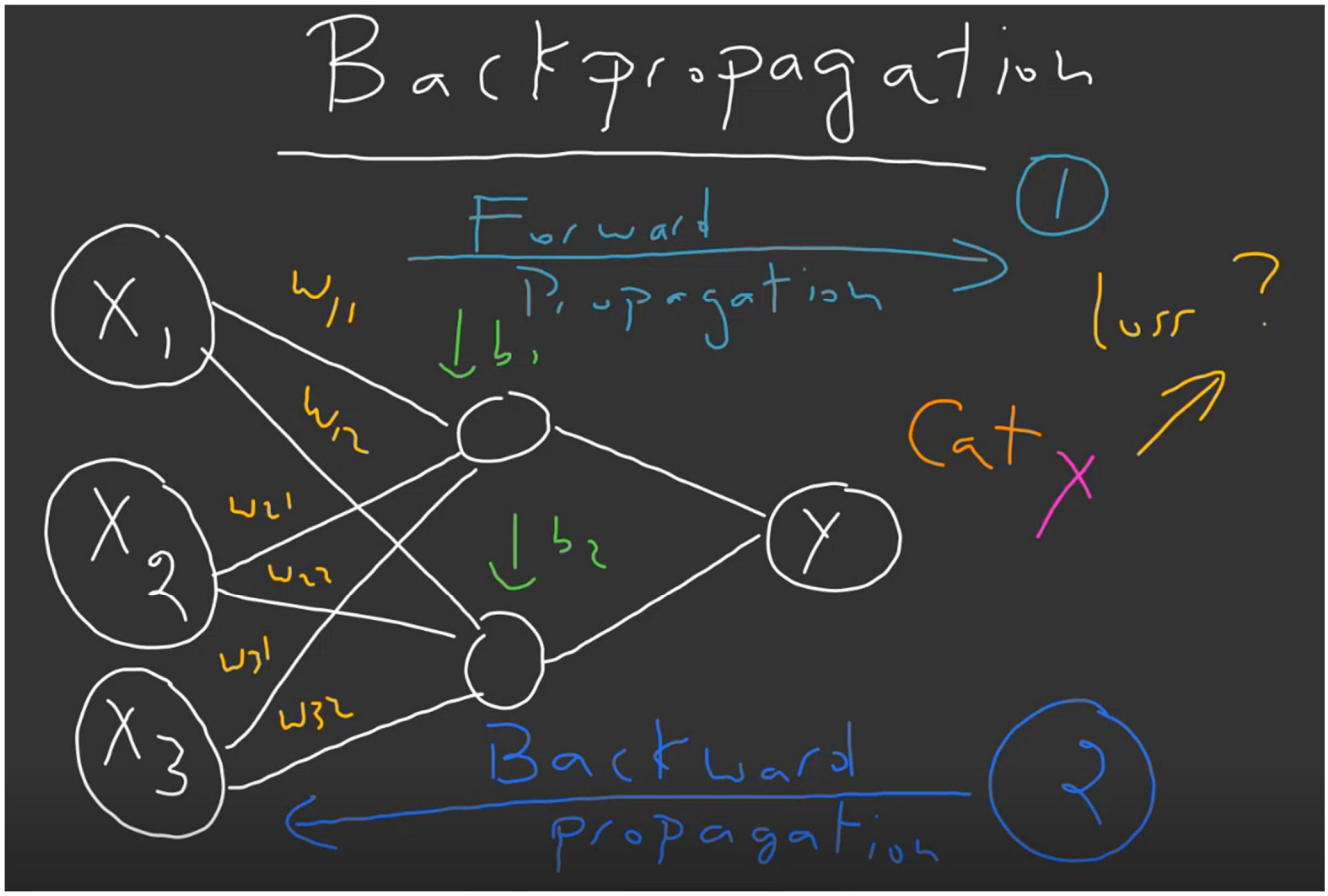
➔ $V_{100} = 0.1\theta_{100} + 0.09\theta_{99} + 0.081\theta_{98} + 0.729V_{97}$

The current observation will always have the highest coefficient.
The coefficient diminishes gradually further backwards.

Backpropagation of error

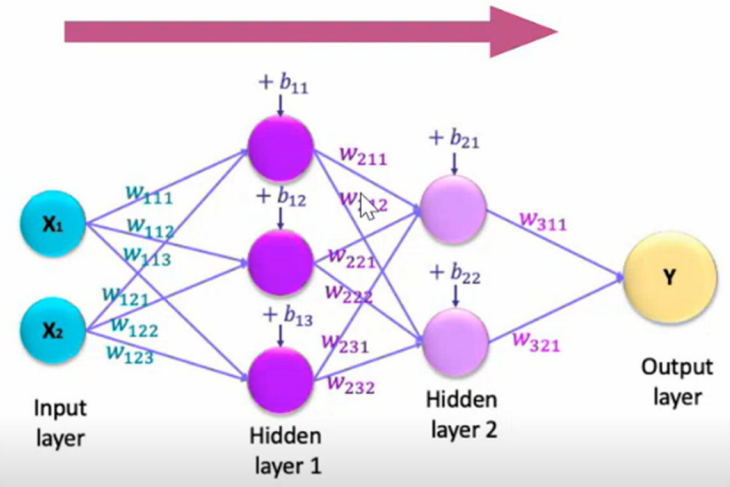
- An algorithm to calculate the gradient of a loss function relative to the model parameters.
- Those gradients are then used by the optimizer to update the model weights.
- Gradients are calculated backwards through the network starting at the output layer, one layer at a time Together.

Together, backpropagation and Stochastic Gradient Descent (or variants) can be used to train a neural network.



The Learning Mechanism

How does the network learn?



Step 1: Forward propagation:
At each neuron
- Pre-activation
- Activation

Step 2: Error calculation by the loss function:

The cost function measures the error between the trues values and the predicted values.

Step 3: Backpropagation and Optimization:

- Backpropagation calculates the gradients of the loss function with respect to the model weights.
- The optimizer uses the gradients to find new values for the weights that can reduce the loss.

Step 4: Steps 1 to 3 are repeated until

- The assigned number of iterations is reached.
- The defined loss value is reached.

NN Fine-Tuning

- Data
  - Increase data size.
  - Data Augmentation.
  - Improve feature Engineering.
- Model
  - Architecture (layers, neurons).
  - Hyperparameters (batch size, learning rate, epochs, ….).

NN Learning Risks

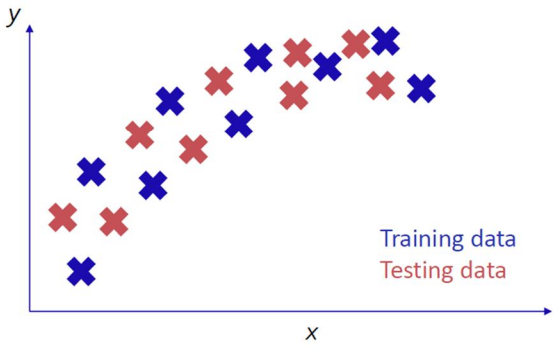
- Underfitting
  - Is too simple and inflexible to learn.
  - NN does not learn enough, Incapable in specification.
  - Cannot generalize to treat new data.
  - A model that has poor performance. (This bad performance is detected through the evaluation metrics).

How to reduce/avoid Underfitting?

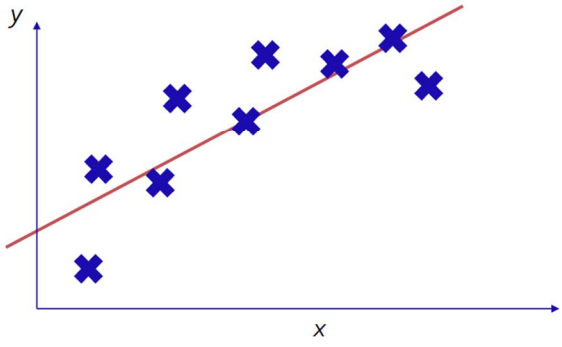
- Increase model complexity.
- Increase data features (independent variables)
- Improve data (remove noise, remove redundancy)
- Add more data.
- Increase the number of epochs.
- Increase training duration.

- Overfitting
  - NN learns too well.
  - Performance badly on new data.
  - Incapable of generalization.

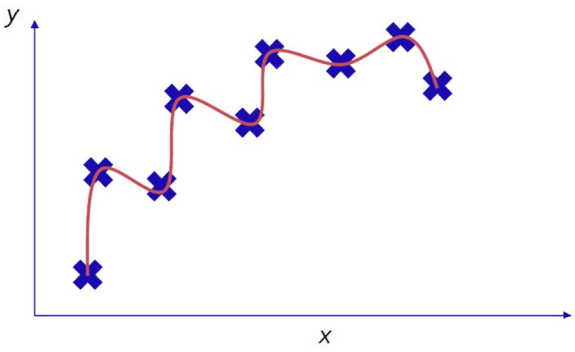
Bias-Variance Tradeoff :



Trained Model 1



Trained Model 2

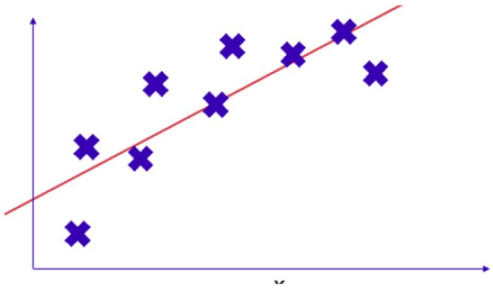


We compare both trained models using RMSE (Root-Mean-Square Error) :

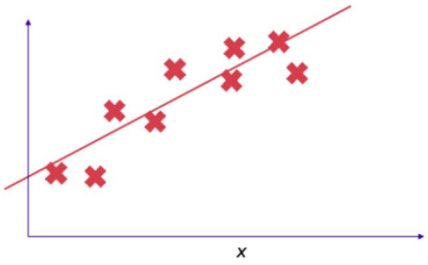
$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$$

We compare both trained models using the test data:
RMSE 1 < RMSE 2 -> Model 1 has a better performance with new data!

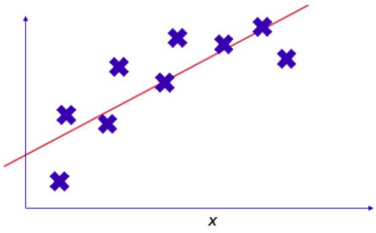
The incapability of a model to reproduce data exactly is called bias.



The different fits of a trained model to different datasets is called the variance.

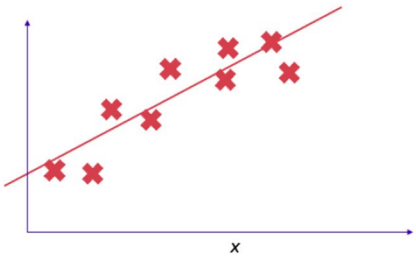


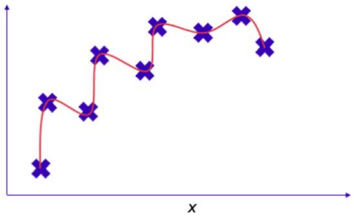
Predictable model: Relatively high bias, relatively low variance
Unpredictable mode: very low bias, very high variance.



Relatively high bias
Relatively low variance

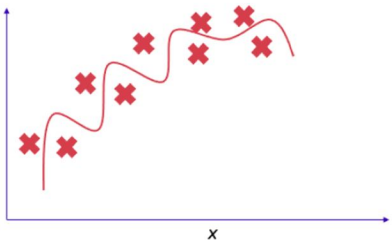
➔ Predictible model





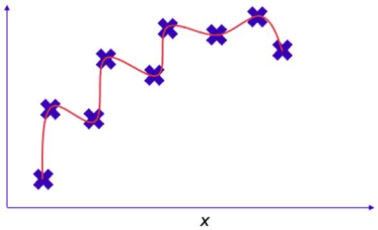
Very low bias
Very high variance

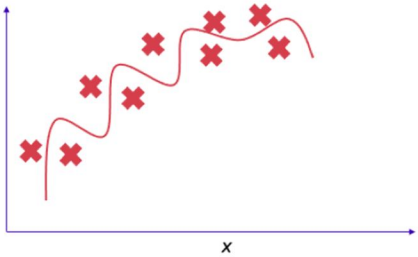
➔ Unpredictible model



Overfitting :

The ideal model has both the bias and the variance low





Since this model has a perfect fit to the training data and a bad performance with new data
-> The model is overfit

How to reduce/avoid Overfitting?

- Cross validation to tune hyperparameters.
- Use more data for training.
- Feature selection (removing irrelevant features)
- Early stopping
- Regularization

Learning, Validation & Testing

Dataset

- Learning
- Validation
- Testing

Due to overfitting risk, the learning loss is not a good indication of a well trained model

We need validation and training datasets

Note : These three datasets are separate and contain no repeated data

Learning (Training) dataset

The data used to train the model (adjust the weights)

- During each epoch, our model will be trained over the entire learning dataset.
- Labeled (loss & accuracy in each epoch can be provided)

Validation dataset

- The data used to validate our model during training.
- After weights are adjusted, the model is validated using this dataset.
- Does not contribute in the weight adjustment (learning)
- Helps us choose the best hyperparameters for our model.
- Helps us detect overfitting.
- Labeled (loss & accuracy in each epoch can be

Test dataset

The data used to test the model after it has been trained and validated
Should not be labeled (the same condition as in using our deployed model in real life)

The ultimate goal of machine learning and deep learning is to build models that are able to generalize and perform well on new unseen data.

Hyperparameters:

- Variables set before the model's training.
- Help in model selection.
- Include parameters that:
    - specify the network structure (number of hidden layers)
    - determine how the network is trained (number of epochs, batch size, learning rate)

Batch size :
The size of the training set

Hidden layers :
Layers between the input and the output layers in the neural network

Learning rate :
A parameter that controls the rate at which the neural network learns (ranges between 0.0 and 1.0)

Epoch :
A pass over the entire training database

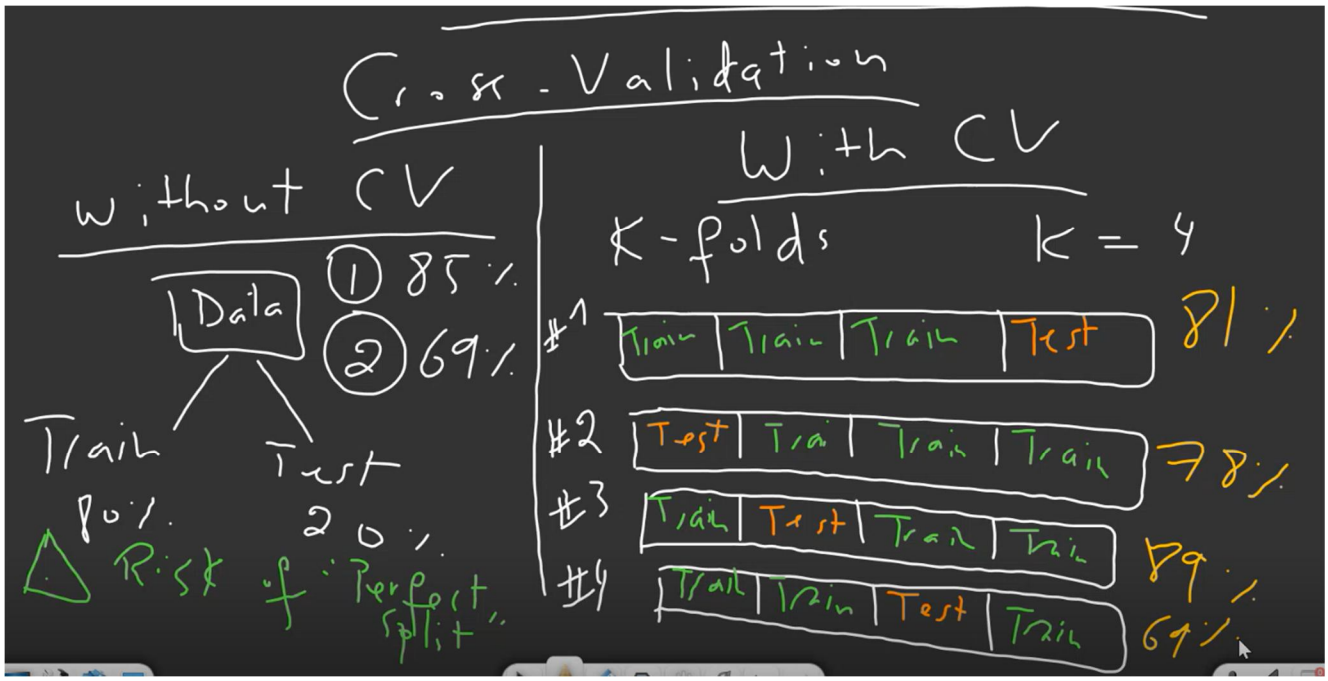
Cross validation trains the model on subsets of the available data and evaluates it on the complementary subset that was not used for training.

K fold cross validation
- The data is split into k subsets/folds.
- Training is done on k 1 subsets
- Evaluation is done on the left out subset.
- This process is repeated k times with a different subset left for evaluation each time.

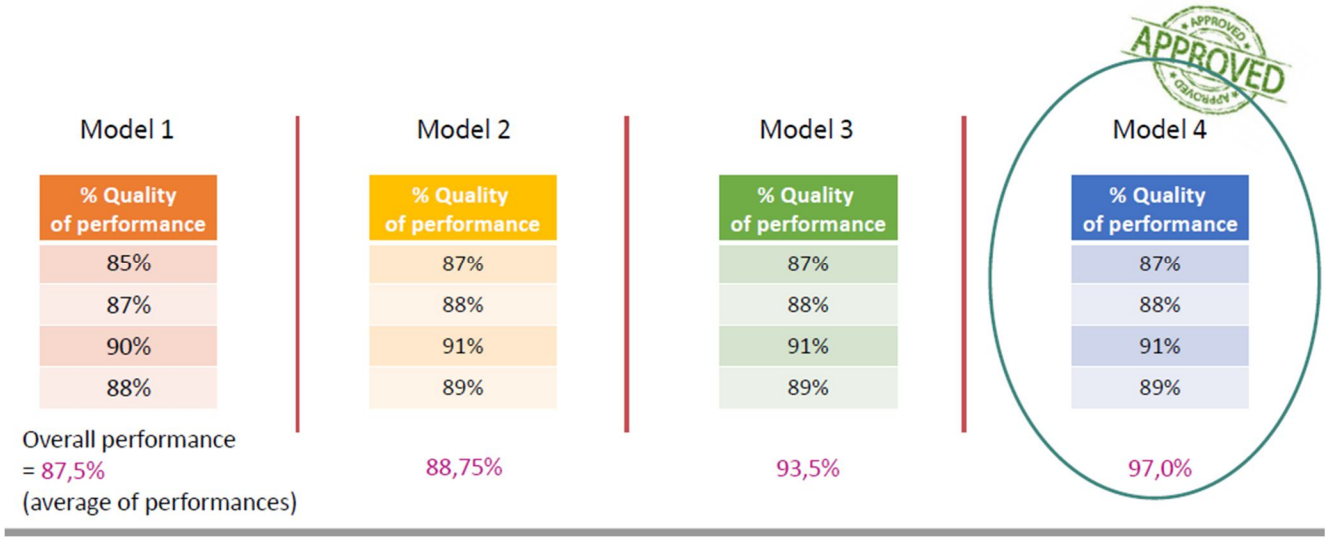
Cross-Validation :

Helps detect overfitting.

- Is used to optimize hyperparameters of a model
- Is used for model selection



Example : Model selection with k fold cross validation (with k = 4)



Regularization

- is a technique that introduces slight modifications to the learning algorithm for it to generalize better
- also improves the model's performance on new unseen data
- helps reduce overfitting by reducing the model's complexity
- introduces a small bias to our model that, in turn, reduces the variance significantly.

1. L1 Regularisation (Lasso Regression) :

$$L1Regularisation = Lossfunction + \lambda \sum_{i=1}^{n} |w_i|$$

Regularisation term that adds bias to the model to avoid perfect fitting

λ determines how strongly the regularization influences the network's training.

- If λ = 0 --> no regularization at all
- If λ = 1,000,000,000 --> very high regularization (too much bias, model hardly learns anything)

Reasonable choices of the regularization strength λ (0.001, 0.01, 0.1, 1.0, etc.)

The best λ can be chosen using cross validation

2. L2 Regularization (Ridge Regression):

$$L2Regularisation = Lossfunction + \lambda \sum_{i=1}^{n} w_i^2$$

Regularisation term that adds bias to the model to avoid perfect fitting

The best λ can be chosen using cross validation

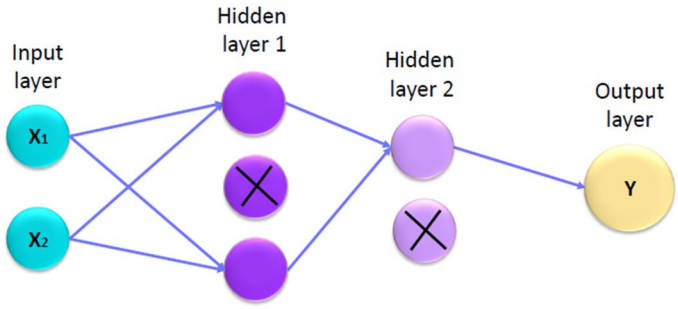
L1 versus L2 Regularization

L2 Regularization shrinks less important parameters close to 0 but not equal to 0	L1 Regularization shrinks less important parameters to 0
L2 Regularization penalizes larger weights more severely: ** Example: A weight of 10	
in L2 Regularization: get a penalty =100	**in L1 Regularization:** get a penalty = 10

3. Dropout Regularization

Dropout regularization
- is a computationally cheap type of regularization.
- Produces very good results.
- Is the most frequently used regularization technique in deep learning.

Dropout regularization: At every iteration, the algorithm randomly selects some nodes and removes all connections (inputs & outputs) to them.

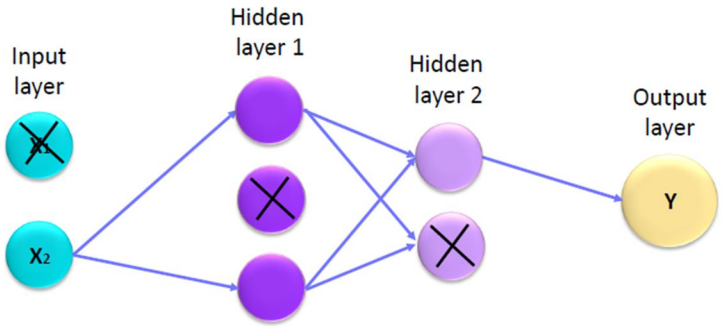


Each iteration has a different set of dropped out nodes.

The probability of choosing the number of nodes to drop out is a hyper parameter that we can assign to the dropout function.

Dropout can be applied to both the hidden and the input layers.

Dropout is usually preferred for large neural networks.



3. Early Stopping
As soon as the performance on the validation dataset starts to decrease (loss starts to increase), the training is stopped immediately, hence the early stopping

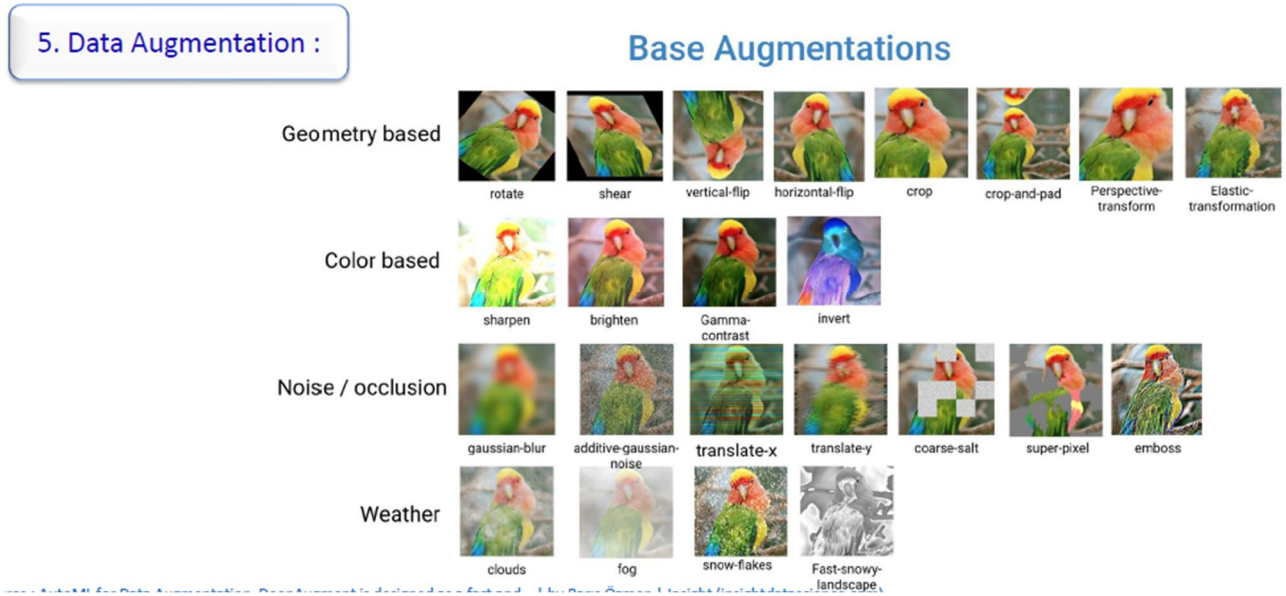
Example :

The training is stopped at the dotted line

Data augmentation corresponds to increasing the size of the training dataset.
** Example: We can introduce variations in images: rotation, flipping, rescaling, shifting

4. Data Augmentation:

5. Data Augmentation :



https://blog.insightdatascience.com/automl-for-data-augmentation-e87cf692c366

```
parameters = {'batch_size': [20, 30], 'nb_epoch': [50, 100], 'optimizer':
['adam', 'sgd']}

grid_search = GridSearchCV(estimator=my_model, param_grid=parameters, scoring
= 'accuracy', cv = 10)



grid_search = grid_search.fit(X, y)

best_parameters = grid_search.best_params_
best_accuracy = grid_search.best_score_

print(best_parameters)
print(best_accuracy)
```