Management of Multimedia Information in Internet

Module 5. Natural Language Processing (NLP)

Unit 4. Neural NLP

Boni García

http://bonigarcia.github.io/
boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

2020/2021

uc3m | Universidad Carlos III de Madrid

Table of contents

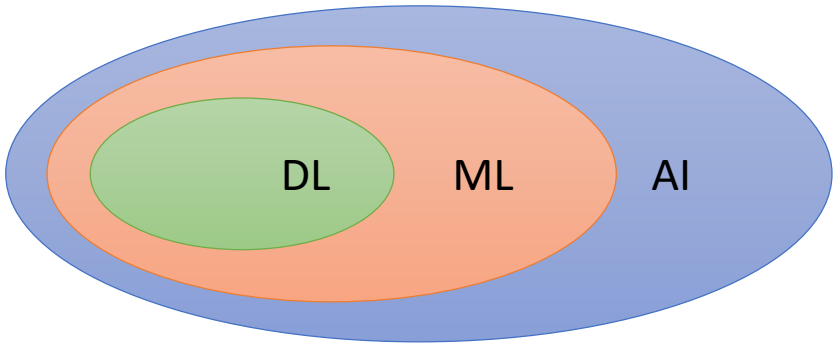
1. Introduction

- As of the 2010s, **deep neural networks** became widespread in Natural Language Processing (NLP)
  - This approach is nowadays known as **neural NLP**
- In neural NLP, we use **Deep Learning (DL)** and/or **Artificial Neural Networks (ANN)** to implement NLP applications (e.g. text classifiers, chatbots, etc.)
  - DL is a kind of ANN in which more than one hidden layer (typically a lot)
- In this course, we will use **Keras** to build NLP applications using ANNs with Python (in Jupyter Notebooks)
  - Keras is a high-level framework written in Python which provides a friendly API optimized for common use cases
  - Keras has been built on the top of **TensorFlow**

Table of contents

2. Deep Learning

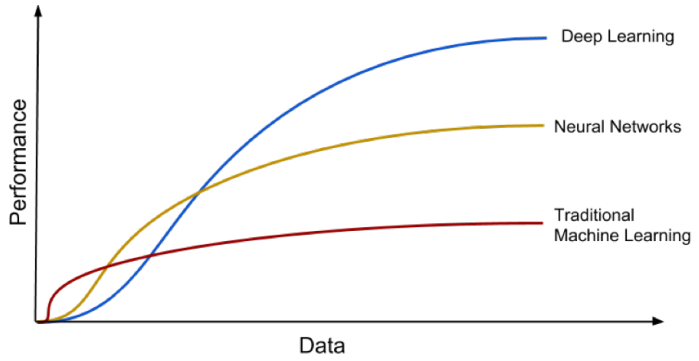
- As we have seen previously, **Machine Learning (ML)** is a branch of Artificial Intelligence (AI) which enables computers to learn and make predictions from raw data
  - In contrast to the traditional method of hardcoded rules or algorithms
- **Deep Learning (DL)** is a branch of ML based on **Artificial Neural Networks (ANN)**
  - ANNs are computing systems designed to simulate the way the human brain analyzes and processes information



2. Deep Learning - Benefits

- There are two main benefits of DL compared to ML

1. Scalability
   - When ANNs are trained with more and more data, their performance continues to increase
   - This is generally different to other ML techniques that reach a earlier limit in performance



Source:
https://www.sumologic.com/blog/machine-learning-deep-learning/

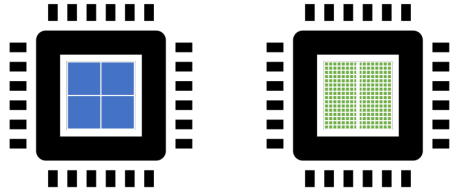
2. Feature learning
   - ML algorithms can have a bottleneck when it comes to creating features (this process is sometimes called feature engineering)
   - DL can automate this feature engineering, since this process becomes part of the training process (first layer of the ANN)

2. Deep Learning - Why now?

- The key concepts of DL were defined in the decade of 1980's
- Nevertheless, there has been an unprecedented rise in DL-based applications and architectures in the first two decades of the twenty-first century
- This widespread adoption has been driven by:
  1. **Data**. The availability of huge volumes of data on the Internet
  2. **Hardware**. Evolution of high-end processors in the form of Graphical Processing Units (GPUs) and Tensor Processing Units (TPUs) has supplemented the rise of DL-based applications by making it possible to perform heavy calculations
  3. **Tools**. Increasing availability of open-source libraries, such as **TensorFlow** or **Keras**, which make easy to implementation DL-based applications

2. Deep Learning - Why now?

- **GPUs** were originally designed to manipulate computer graphics:
  - A Central Processing Unit (CPU) is designed with fewer processor cores that have higher clock speeds than the ones found on GPUs
  - GPUs, on the other hand, have much greater number of cores and render images more quickly than a CPU because of its parallel processing architecture, which allows it to perform multiple calculations across streams of data simultaneously



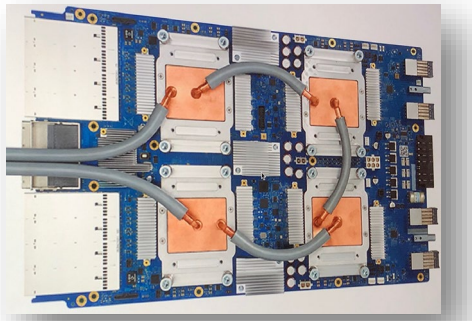
CPU	GPU
Central Processing Unit	Graphics Processing Unit
4-8 Cores	100s or 1000s of Cores
Low Latency	High Throughput
Good for Serial Processing	Good for Parallel Processing
Quickly Process Tasks That Require Interactivity	Breaks Jobs Into Separate Tasks To Process Simultaneously
Traditional Programming Are Written For CPU Sequential Execution	Requires Additional Software To Convert CPU Functions to GPU Functions for Parallel Execution

Source: https://towardsdatascience.com/parallel-computing-upgrade-your-data-science-with-a-gpu-bba1cc007c24

- Since GPUs perform parallel operations on multiple sets of data, starting int early 2010's, GPUs were started to be used to accelerate calculations not only related to graphics (e.g. data science)

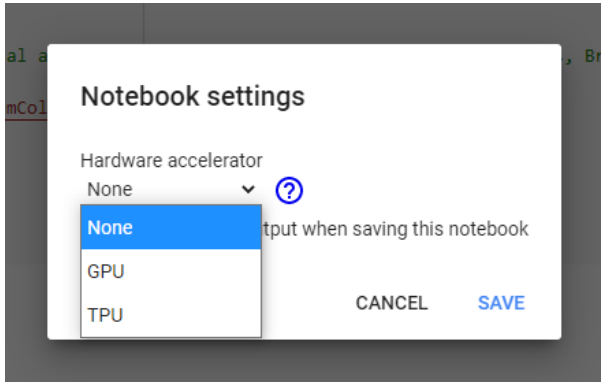
2. Deep Learning - Why now?

- In May 2016, Google announced its Tensor Processing Unit (**TPU**)
  - TPU is an AI accelerator application-specific integrated circuit (ASIC) developed by Google specifically for NN and ML using TensorFlow



Source: https://en.wikipedia.org/wiki/Tensor_Processing_Unit

- **Google Colaboratory** allows to use GPU and TPU (experimentally) architecture as runtime for Jupyter Notebooks for free (depending on the availability)



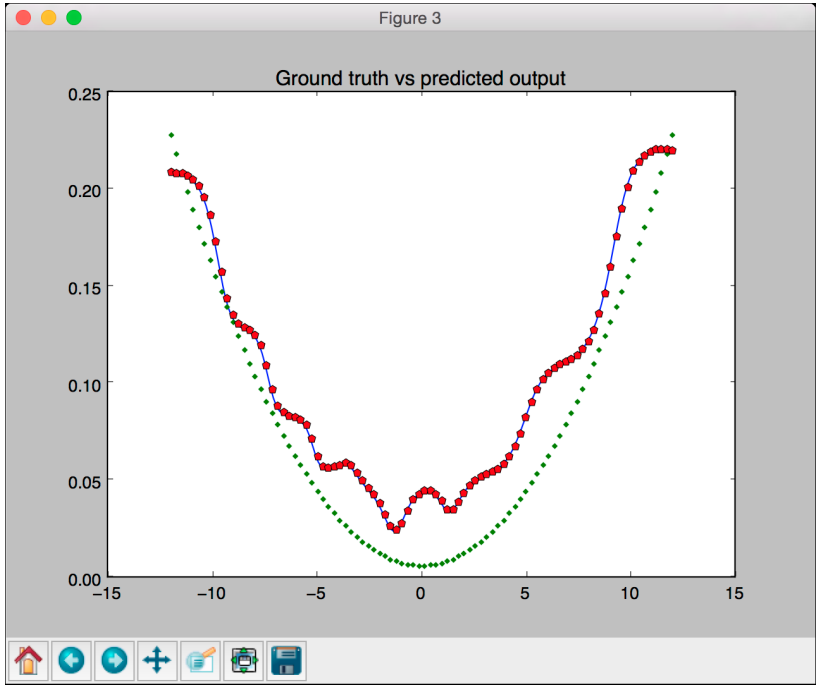
2. Deep Learning - Applications

- Some examples of DL-based applications are:
  - Self-driving cars: Through sensors and onboard analytics with DL, cars are learning to recognize obstacles, facilitate situational awareness and strive to react appropriately
  - Image recognition and labeling: Deep learning algorithms enable machines not only used to recognize pictures, but also to find meaningful descriptions thereof
  - Text classification (also known as text categorization): Including sentiment analysis, news categorization, question answering, or natural language inference
  - Language translation services: Neural machine translation is replacing the use of statistical machine translation producing more accurate translations
  - AlphaGo: which is a computer program that plays the board game Go
- These applications can produce results comparable to (and in some cases, surpassing) human expert performance

Table of contents

3. Neural Networks

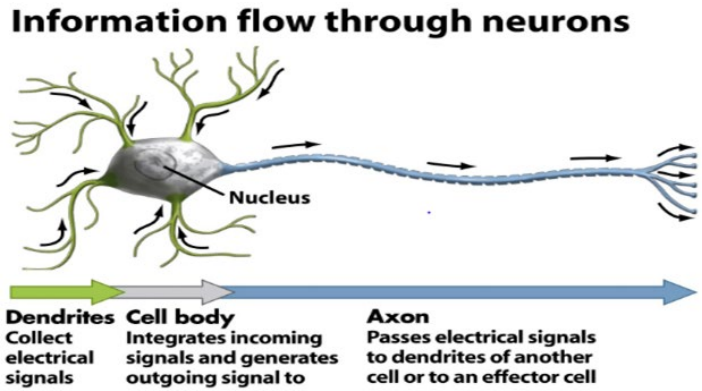
- Artificial Neural Networks (ANNs), or simply **Neural Networks (NNs)** are computing systems that attempts to identify the hidden trends within data by using a process that mimics the functioning of neural networks within human brain (biological **neurons**)
  - A NN is a mathematical construct that can approximate almost any function, and generate predictions for complex problems
  - NNs have the ability to evolve as per new information available so the network produces the best possible result without the need to redesign the output criteria



Source: Joshi, P. (2016) ***Python Machine Learning Cookbook***. Packt Publishing.

3. Neural Networks - Biological neurons

- A biological **neuron** is a nerve cell specialized to transmit information throughout the body
- The main components which make possible this transmission are:
  - Dendrites (receivers). Each dendrite has a weight associated to the incoming signal. This weight dictate the importance of the signal coming in. These values get changed dynamically
  - Cell body: Incoming signals are summed up in the cell body (called soma). As this value reaches a particular threshold, the summed-up signal is propagated through the axon
  - Axons (transmitters). Connected to other cells (e.g. dendrites from other neurons)



Source: https://www.fromthegenesis.com/artificial-neural-network-part-2/

3. Neural Networks - Biological neurons

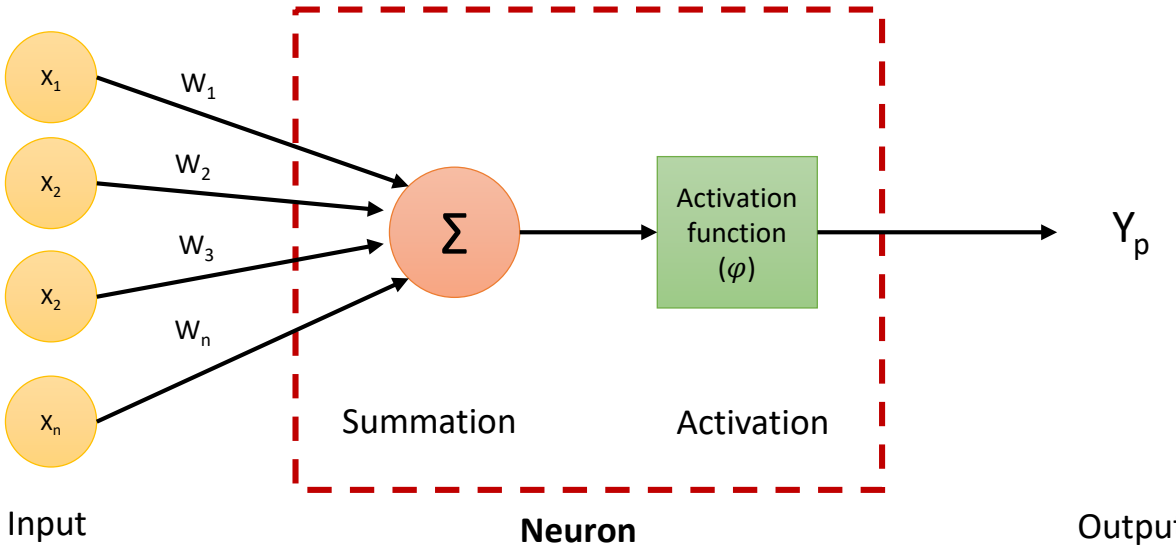
- The human **brain** is essentially a bunch of neurons connected to each other in a huge interconnected network (it is estimated that there are 100 billion neurons in the human brain)
- Neural connections can be **strong** and **weak**
  - A strong connection allows more charge to flow between them and a weak one allows lesser
  - A neuron pathway which frequently transmits charge will eventually become a **strong pathway**
  - Neural pathways become **stronger** upon frequent usage, and our brain essentially tries to use pathways which have proven to give us better results over time
- We humans live our lives and decide whether our actions are good or bad, we are training our brain to make sure we do not repeat our previous mistakes

3. Neural Networks - Perceptron

- The basic unit a NN is called **neuron** (or **perceptron**). A neuron accepts some input and generates some output. A neuron is composed by:
  - Input: incoming (weighted) data from other networks/neurons → Similar to dendrites
  - Summation: aggregates the input signal received
  - Activation: takes the aggregated information and fires a signal only if the aggregated input crosses a certain threshold
  - Output (similar to axon): connected to other neurons/networks or final output layer (for predictions)

Similar to soma

Similar to axon



$$Y_p = \varphi\left(\sum_{i=1}^{n} \boldsymbol{W_i} \cdot \boldsymbol{X_i} + \boldsymbol{b}\right)$$

The neuron add as bias value (b) for having a margin of adjustability (not dependent on the input)

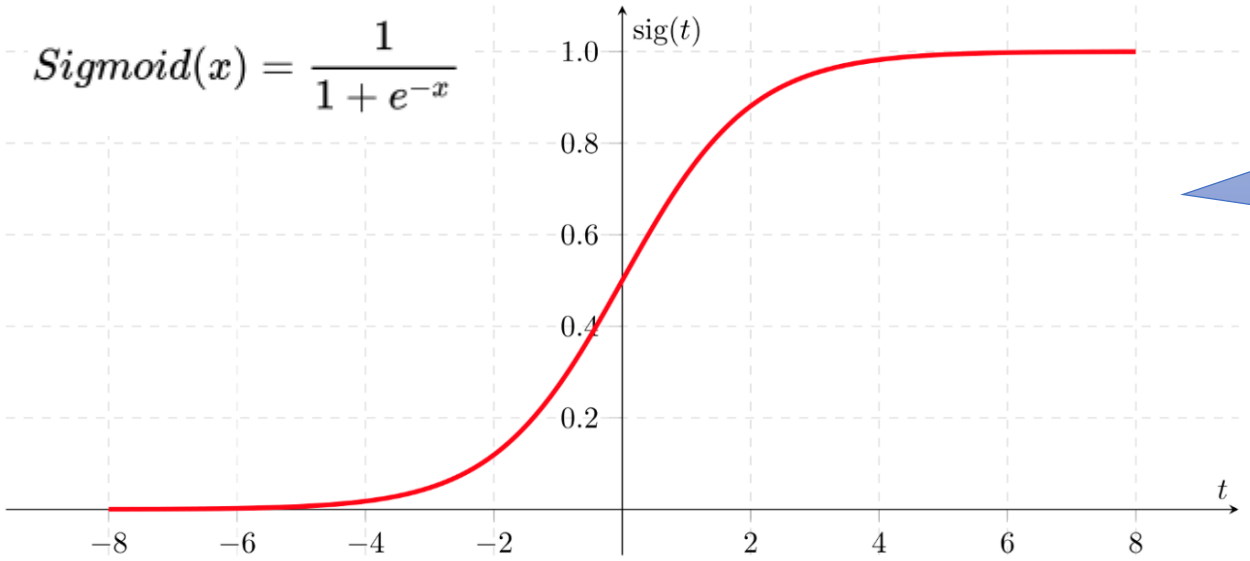
3. Neural Networks - Activation function

- The **activation function** (a.k.a. transfer or step function) control the threshold that decides to fire the output
  - It is used to determine the output layer like a yes or no
  - It maps the resulting values in between 0 to 1 or -1 to 1
- Activation functions introduce **nonlinearity** in the network
  - Without nonlinearity, the network would be performing linear mappings between the input
  - Linearity in not desirable since in real world data often there is non linear relationships between the input and output variables
  - Hence all we have to do is keep some non linear function as the activation function for each neuron is capable of fitting on non linear data

3. Neural Networks - Activation function

- Some examples of activations functions are:
  - Sigmoid
  - Hyperbolic tangent (Tanh)
  - Rectified linear unit (ReLU)
- The **sigmoid** activation function constrains the output in a range between 0 and 1 (used for tasks involving binary outputs)

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

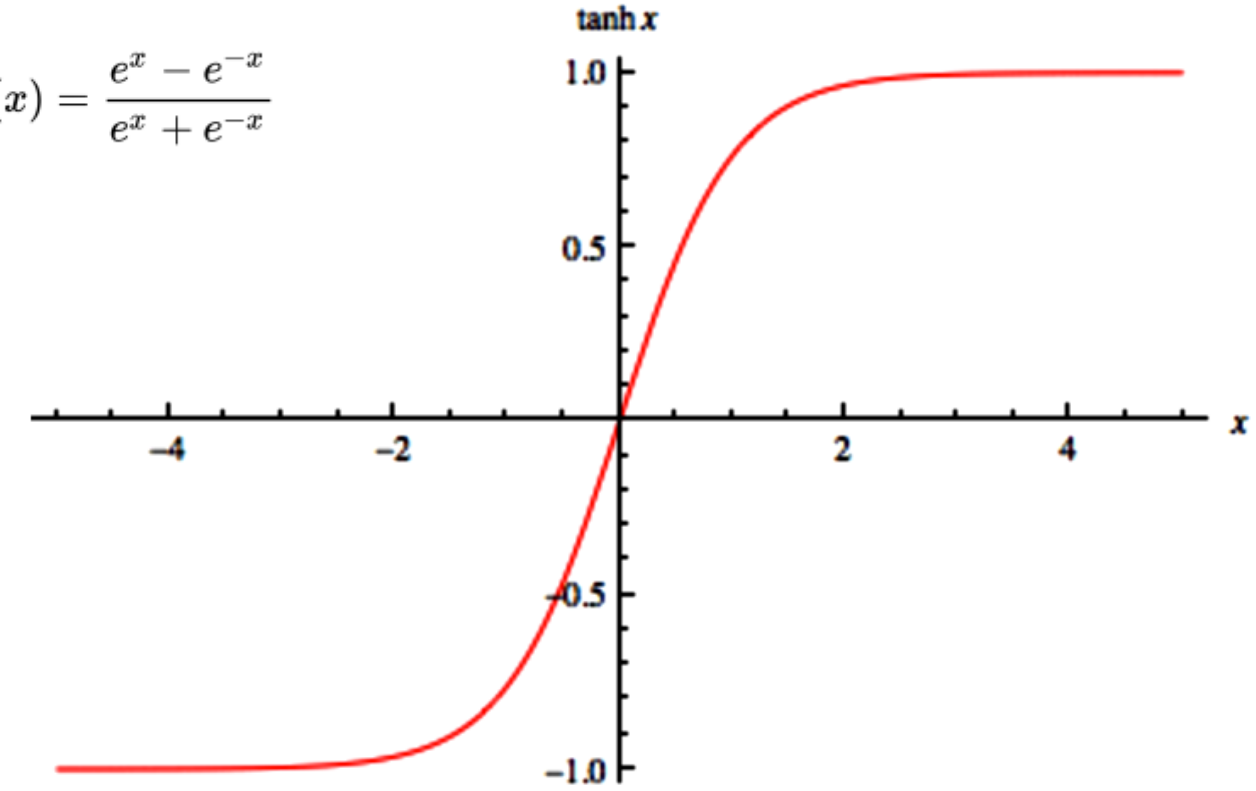


Each neuron bias value (b) allows to shift the activation function to the left or right

3. Neural Networks - Activation function

- The **hyperbolic tangent** (Tanh) converts the input values within the range of -1 to 1. It is a zero-centered function (unlike sigmoid)

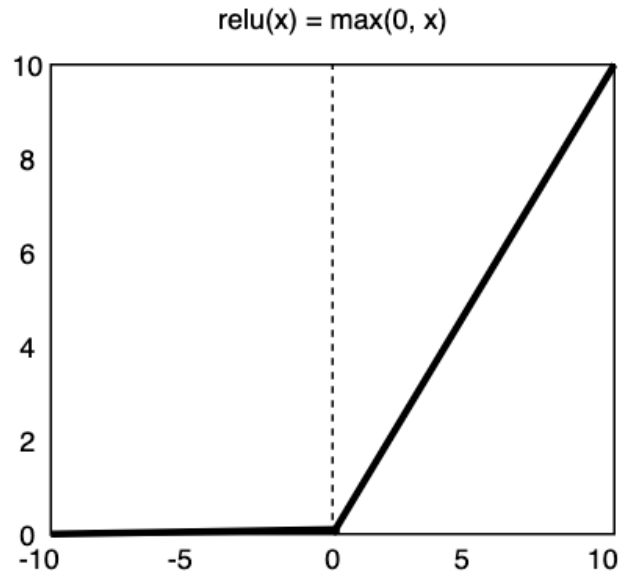
$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



3. Neural Networks - Activation function

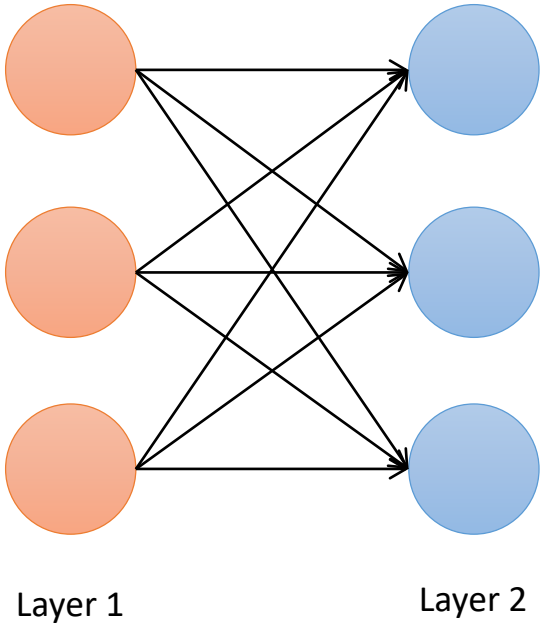
- The rectified linear unit (**ReLU**) is one the most commonly used activation function today
- ReLU is calculated with simple formula, which does not require complex computations

$$ReLu(x) = max(0, x)$$



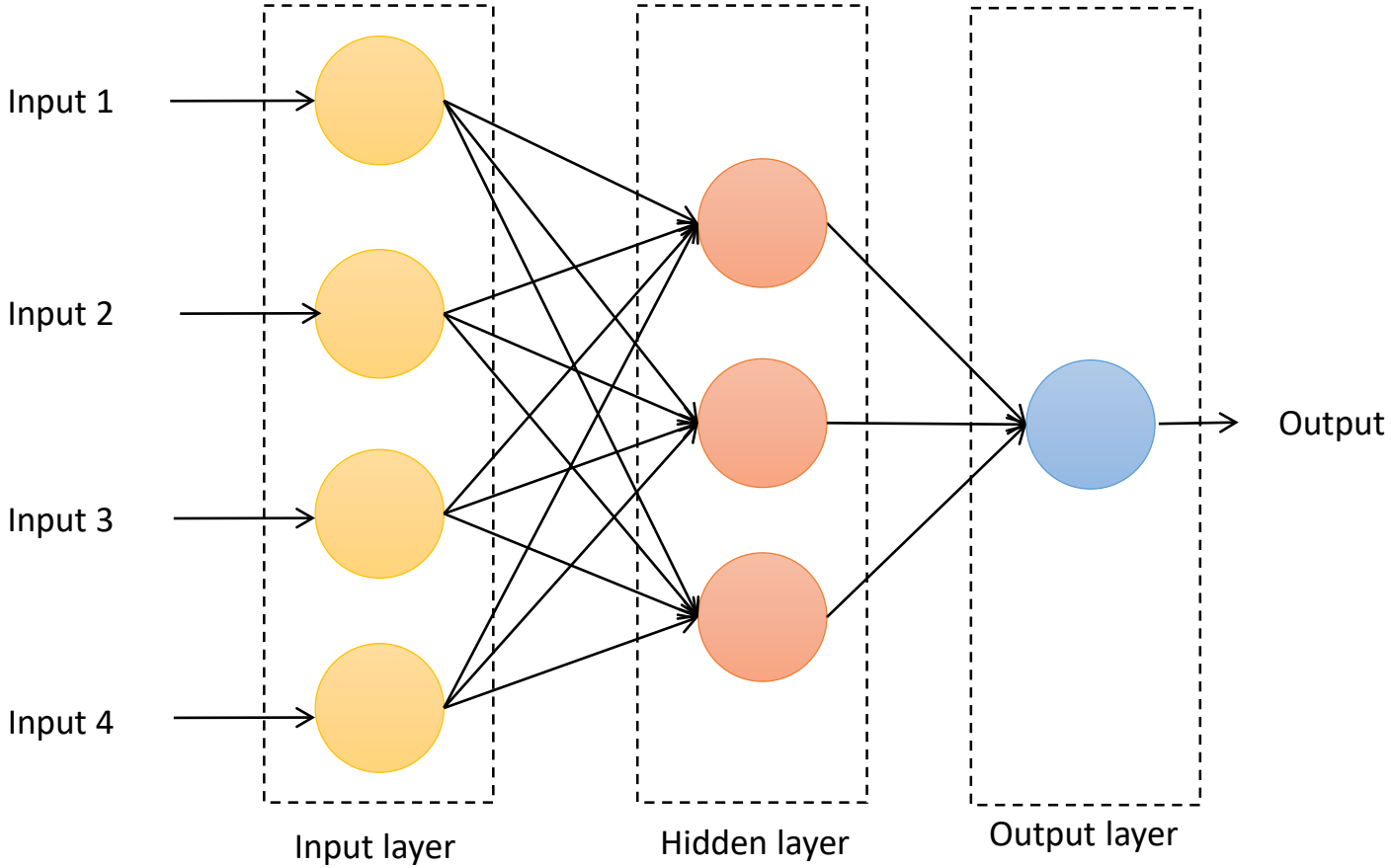
3. Neural Networks - Architecture

- A NN consists of many **nodes** (neurons) in many **layers**
- Each layer can have any number of nodes and a neural network can have any number of layers



3. Neural Networks - Architecture

- An NN consists of three types of layers: **input**, **hidden**, and **output**

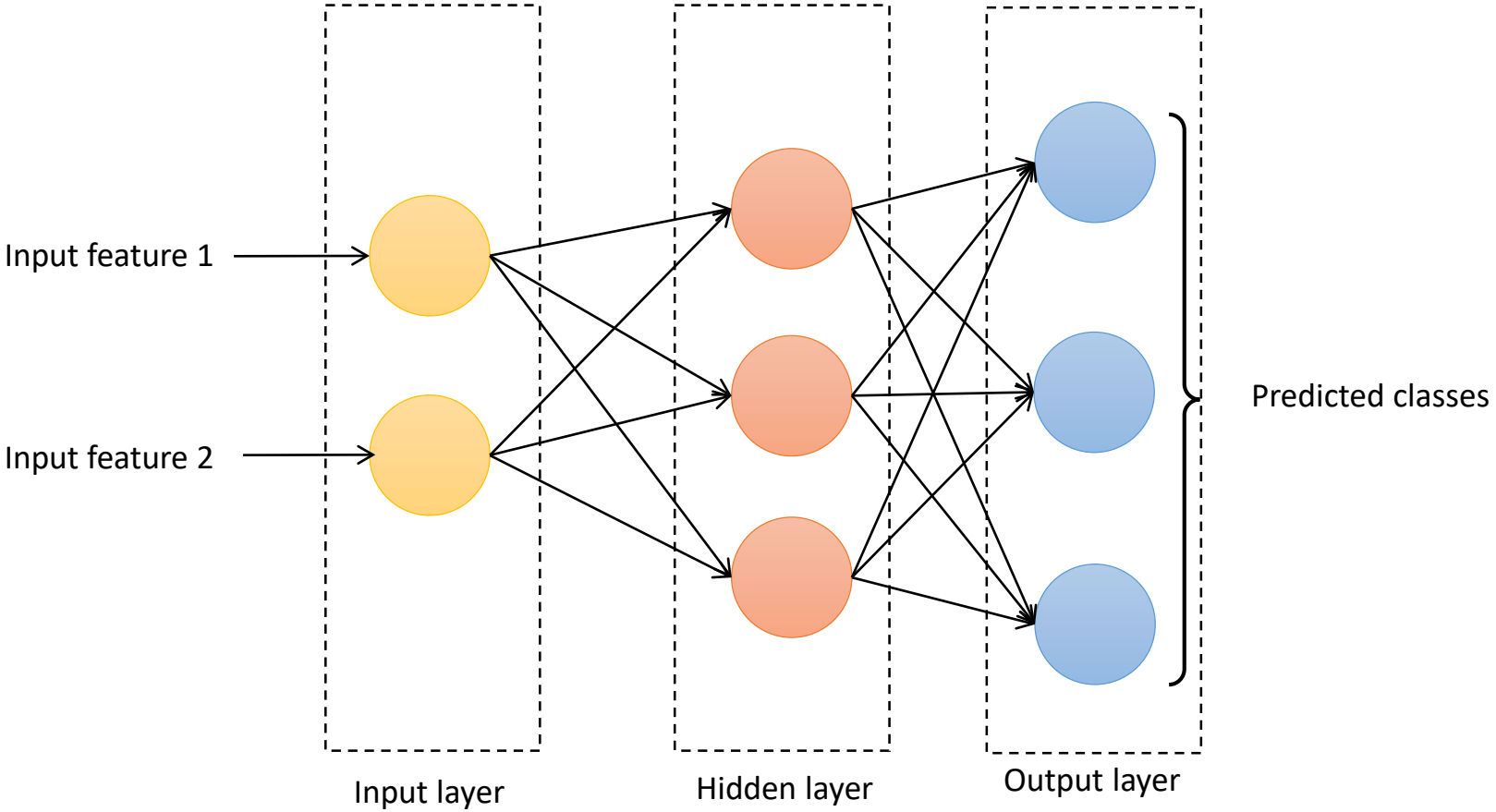


3. Neural Networks - Architecture

1. Input layer:
   - The number of nodes (or neurons) in this layer is equal to the number of features that to be fed to the network
2. Hidden layer:
   - Intermediate layers in a NN
   - An NN can have one or more hidden layers. For simple datasets, NN usually has only one hidden layer. A NN is referred as deep when there is more than one hidden layer
   - The relationships and patterns in data are derived in these layers
   - The number of hidden layers and nodes in each hidden layer are hyperparameters and need to be tuned
3. Output layer:
   - Final layer in NN that provides the output for a particular input
   - The number of nodes in the output layer depends on the type of problem being solved. For example, the output layer has only one node for a binary classification problem or equals to the number of classes in a multiclass classification problems

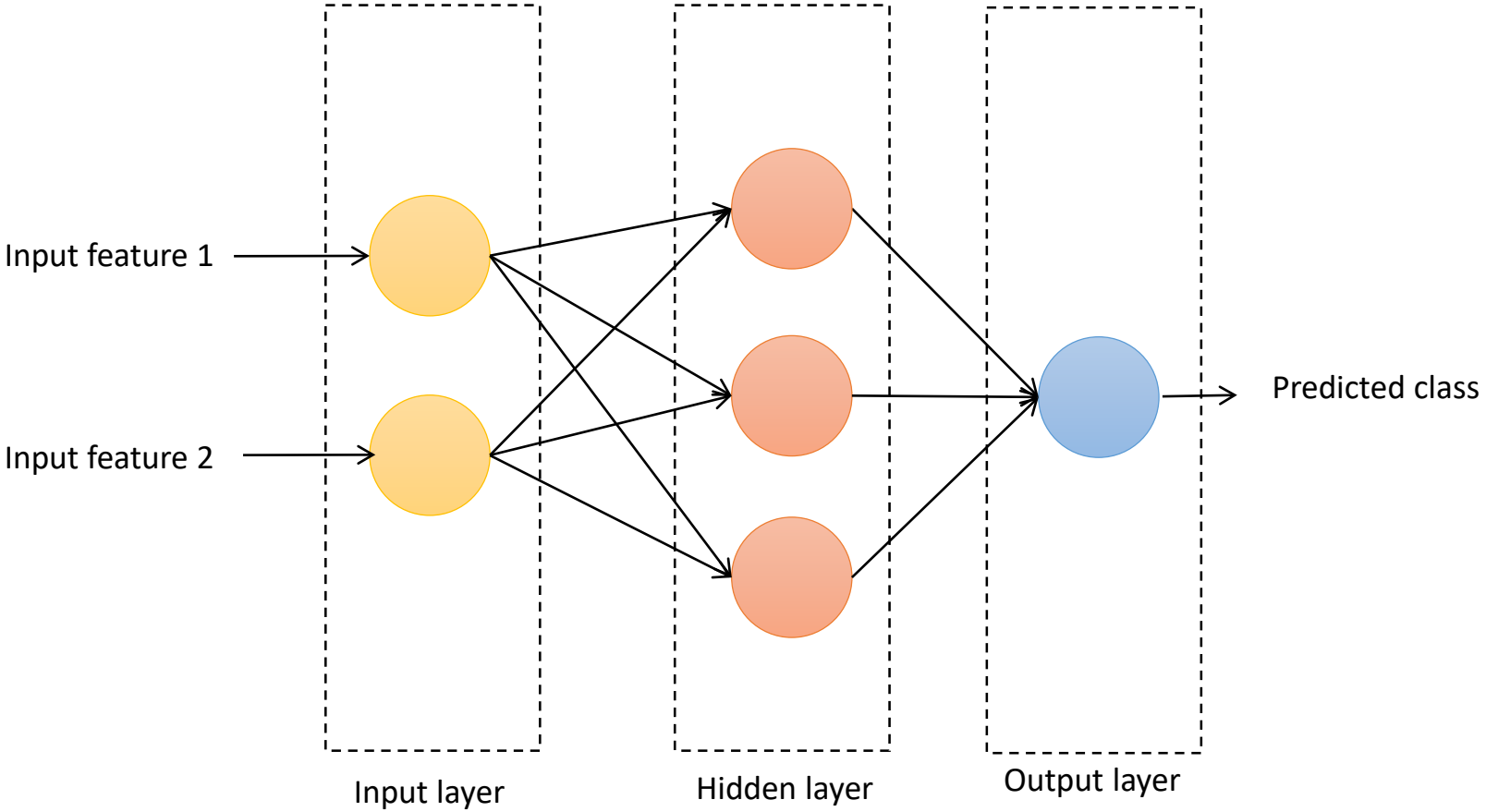
3. Neural Networks - Architecture

- Example (NN architecture for solving a multilabel classification problem):



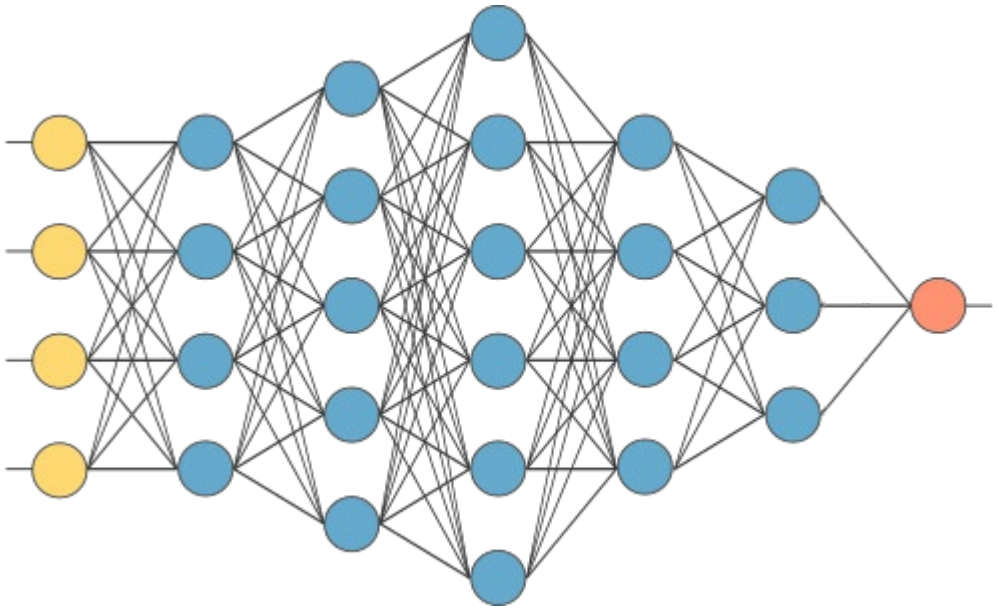
3. Neural Networks - Architecture

- Example (NN architecture for solving a binary classification problem):



3. Neural Networks - Architecture

- When a NN has more than one hidden layer, it is known as a **Deep Neural Network** (DNN)



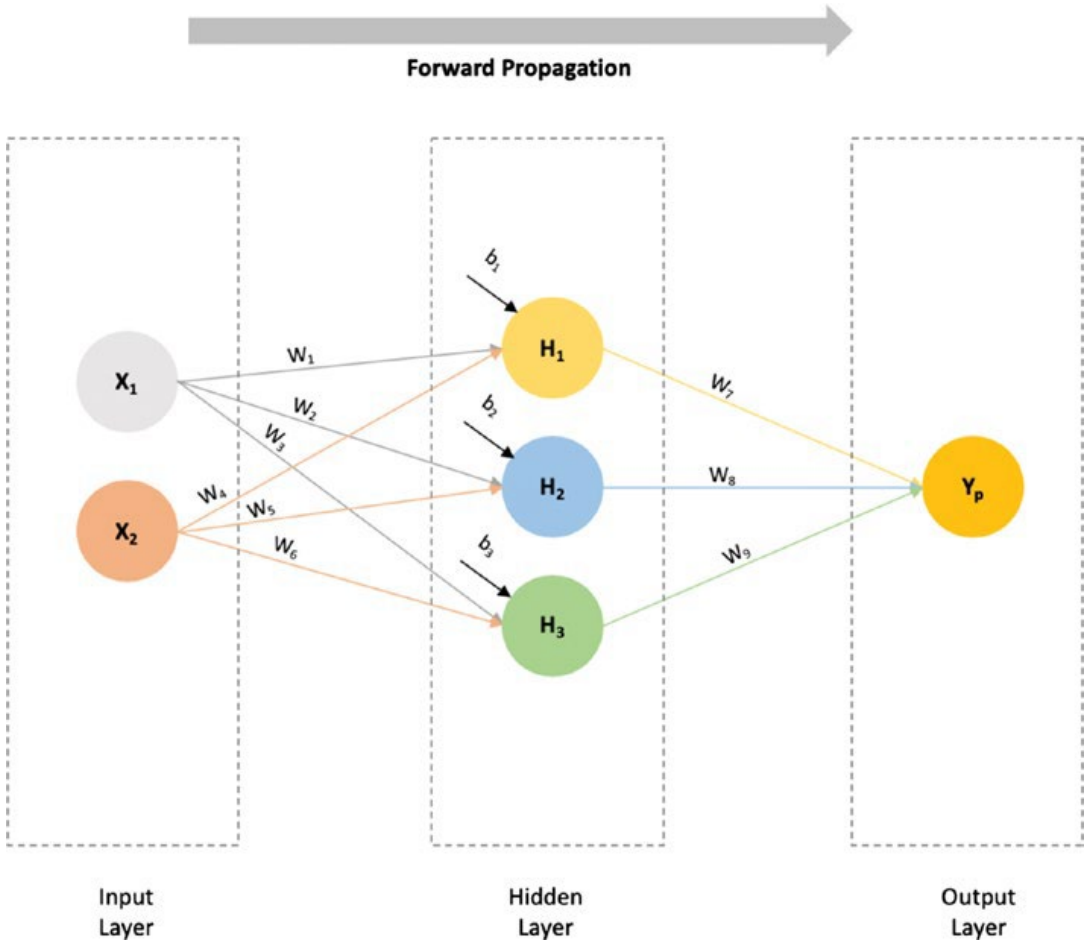
Input layer Hidden layers Output layer

3. Neural Networks - Training

- Now that we have seen how a NN is represented, we can go on to see how exactly it works
- Since there are many layers having many neurons, there exists a complex set of weights to get an output from some input variables
- Each weight in this network can be changed and hence there are countless configurations a neural network can have
- A trained NN has some weights configuration which accurately predicts correct outputs from some input data and that is what we hope to achieve
- **Backward propagation** (or simply backpropagation) is the name of the algorithm a NN uses to train itself

3. Neural Networks - Training

- In a fully connected NN, when the inputs pass through the neurons (hidden layer to output layer) and the final value is calculated at the output layer, we say that the inputs have **forward propagated**

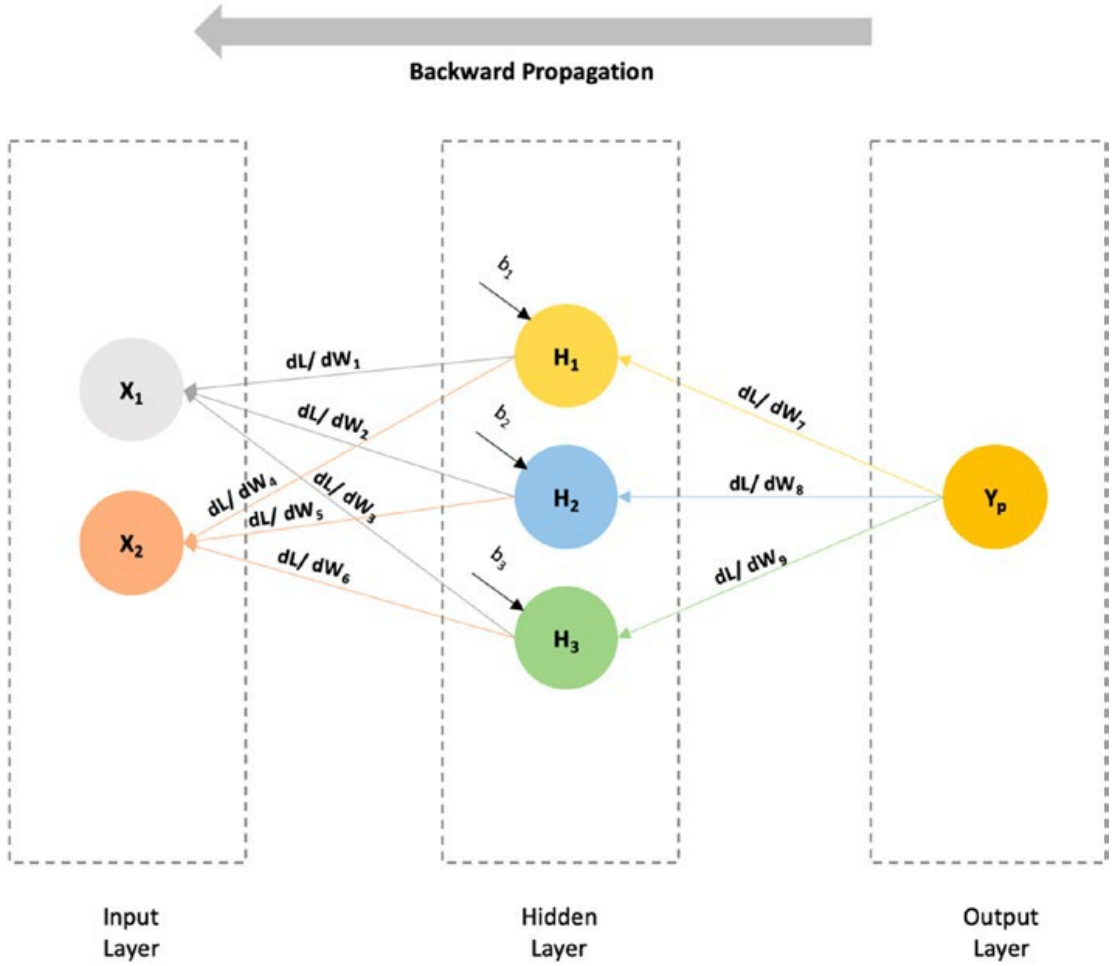


3. Neural Networks - Training

- Supposing we know the actual value of the output (Y), we can calculate the loss value (L) as difference between the actual value and the predicted value (Y_p):

$$L = (Y - Y_p)^2$$

- To minimize L, we try to optimize the weights accordingly, by taking a partial derivate of L to the previous weights (**backward propagation**)



3. Neural Networks - Training

- Summary of the typical learning process of a NN:

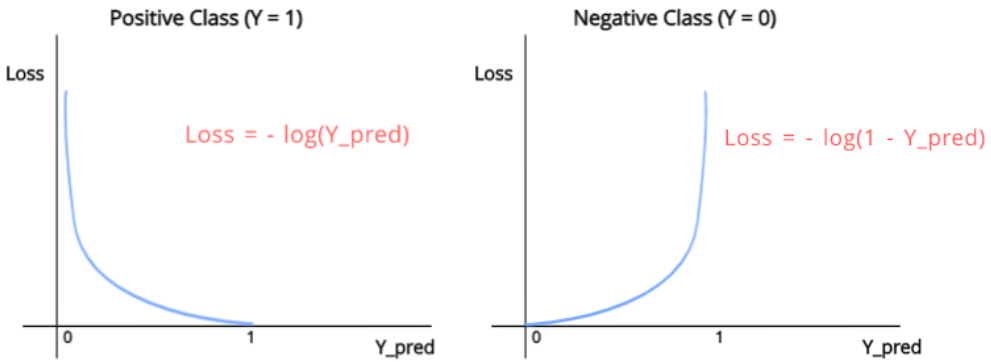
1. Randomly initialize the weights to small values close to zero (but not equal to zero)
2. Forward Propagation: In which information is passed through the NN starting from input layer until the output layer
3. Compare the predicted value and actual value to find out the loss value (also known as difference or error)
4. Back Propagation: In the step error are propagated back to the input layer to adjust the weights
5. Repeat step 2 to 4 until we get the minimum value of the loss function with optimized value of weights for each of the input variable
   - This entire cycle is called one **epoch.** Usually NNs can take several epochs to train and it is up to us to decide how many epochs it will train for

3. Neural Networks - Loss function

- The loss function calculate how well or bad our model is performing by comparing what the model is predicting with the actual value it is supposed to output.
  - If Y_{pred} is very far off from Y, the loss value (L) will be very high
  - However if both values are almost similar, L will be very low
  - Hence we need to keep a loss function which can penalize a model effectively while it is training on a dataset
- In **classification**, a NN is trying to predict a discrete value (i.e. a class for a given input)

3. Neural Networks - Loss function

- In **binary classification**, there will be only one node in the output layer (we will be predicting between two classes)
- In order to get the output in a probability format, we can use the **sigmoid function** to obtain a real value between 0 and 1 as output
  - If the output is above 0.5 (50% Probability), we will consider it to be falling under the positive class and if it is below 0.5 we will consider it to be falling under the negative class
- The loss function we use for binary classification is called **binary cross entropy (BCE)**
  - This function effectively penalizes the neural network for binary classification task



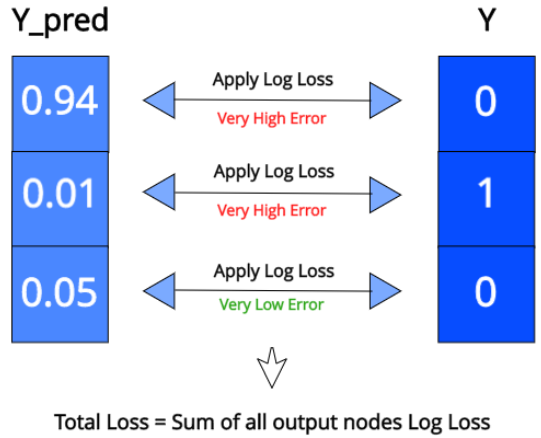
Source: https://deeplearningdemystified.com/article/fdl-3

3. Neural Networks - Loss function

- **Multiclass classification** is appropriate when we need our model to predict **one** possible class output every time
- The activation function we use in this case is **softmax**
  - The goal of softmax is to make sure one value is very high (close to 1) and all other values are very low (close to 0)
  - This function ensures that all the output nodes have values between 0–1 and the **sum of all** output node values **equals to 1 always**

$$Softmax(y_i) = \frac{e^{y_i}}{\sum_{i=0}^{n} e^{y_i}}$$

- The loss function we use for multiclass classification is called **sparse categorial cross entropy**



Source:
https://deeplearningdemystified.com/article/fdl-3

3. Neural Networks - Optimizers

- The loss function tells us how poorly the model is performing
- We need to use this loss to **train** our NN such that it performs better
- For that, we need to try to **minimize** that loss by changing the **weights** of the NN. This process is called **optimization**
- The **optimizers** are the components which implements the **backpropagation algorithm**

3. Neural Networks - Optimizers

- There are several optimizers we can chose when creating a NN, e.g.:
  - Gradient Descent, also called Stochastic Gradient Descent (SGD)
  - Momentum
  - Nesterov Accelerated Gradients (NAG)
  - Adagrad (adaptive gradients)
  - RMSProp
  - Adam
- More info on:

https://deeplearningdemystified.com/article/fdl-4

3. Neural Networks - Generalization

- **Generalization** is the ability of a NN to predict unseen data correctly
- One of the problems that occur during NN training is called **overfitting**
  - The error on the training set is driven to a very small value, but when new data is presented to the NN the error is large
  - The NN memorized the training examples, but it is not able to generalize to new situations
- There are certain aspects of NN which we can control in order to prevent overfitting, such as:
  - Dropout neurons (ignoring some nodes during training)
  - Number of parameters (weights and bias, if any, in the NN) by tuning the number of layers and/or nodes per layer
  - Early stopping (reduce number of epochs before the loss increases)

3. Neural Networks - Review

- A NN is composed of **layers** that are chained together, maps the input data to predictions
- The **loss function** then compares these predictions to the targets, producing a **loss value**: a measure of how well the network's predictions match what was expected
- The **optimizer** uses this loss value to update the network's weights
- This optimization process is repeated a number of **epochs**

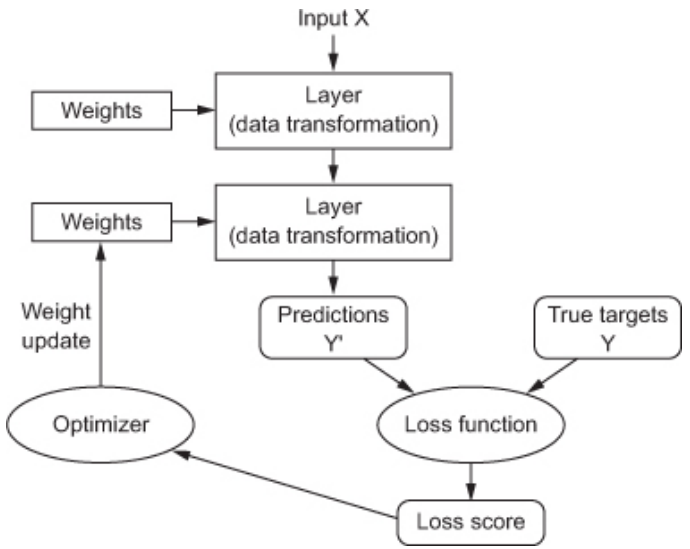


Table of contents

4. Keras

- **Keras** is a high-level framework that can be used to build NNs
  - It is written in Python and provides numerous APIs and modules for defining, building, and training NNs with ease
  - Keras provides a wrapper around frameworks such as TensorFlow (by default), CNTK, or Theano and hides low-level details
- **TensorFlow** is an open-source library developed by Google for ML model building and deployment
  - Developed in C++, it provides different high-level and low-level APIs in different languages, such as Python, JavaScript, C++, Java, Go, or Swift
- In this course we use Keras since it provides a **user-friendly** API optimized for common use cases which provides clear and actionable feedback for user errors

4. Keras - Sequential mode

- A NN can be envisioned as a graph in which layers are stacked
  - Keras provides an API to build these stacks of layers
- The simplest is the **sequential model**, which is a **linear stack of layers**
- **Dense layer** is the regular deeply connected NN layer
  - All neurons in each layer are connected to all neurons in the next layer
  - It is most common and frequently used layer
  - Dense layer does the below operation on the input and return the output

output = activation(dot(input, kernel) + bias)

- input represent the input data
- kernel represent the weight data
- dot represent numpy dot product of all input and its corresponding weights
- bias represent a biased value used in machine learning to optimize the model
- activation represent the activation function.

https://www.tensorflow.org/guide/keras/sequential_model

4. Keras - Sequential mode

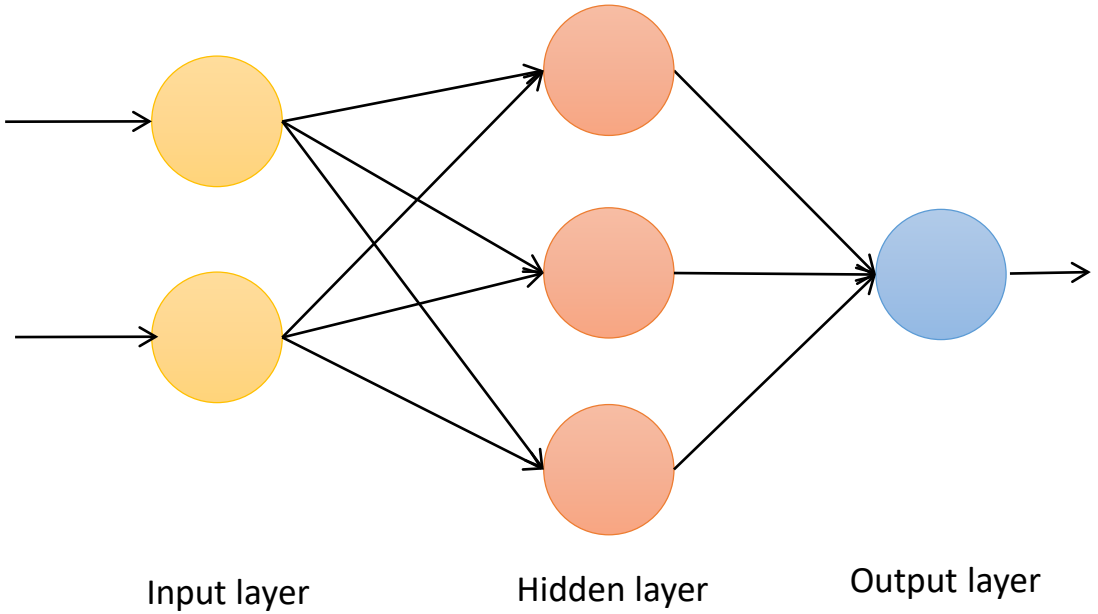
- Example of model definition using sequential mode in Keras:

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

model = Sequential()
model.add(Dense(3, input_dim=2))
model.add(Dense(1))
```

The first stack defines the **input** layer an the first **hidden** layer

The last stack defines the **output** layer



4. Keras - Sequential mode

- Example of model definition using sequential mode in Keras:

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

model = Sequential()
model.add(Dense(3, input_dim=2))
model.add(Dense(1))

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 3)                 9
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 4
=================================================================
Total params: 13
Trainable params: 13
Non-trainable params: 0
_________________________________________________________________
```

The method `summary()` provides a summary of the Keras model. The **parameters** are **weights** that are learnt during training. In the sequential mode of Keras, the number of parameters of each layer is calculated as:

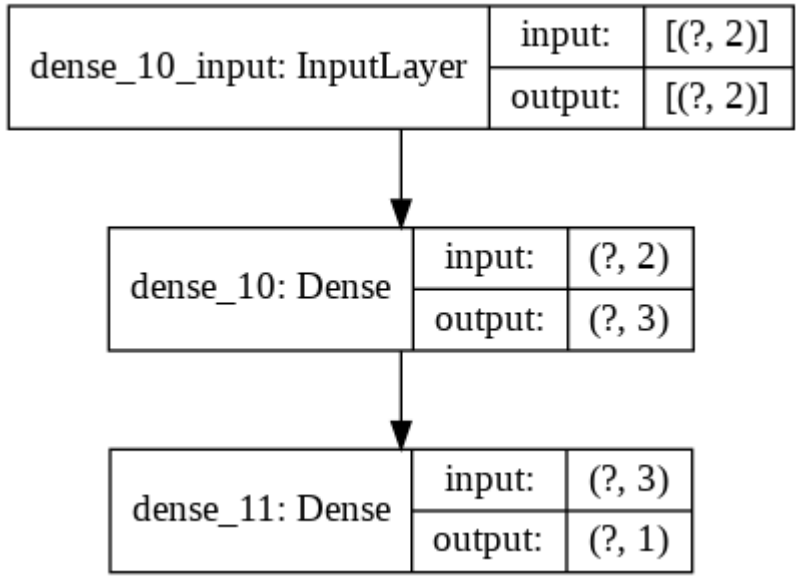
output_size * (input_size + 1)

4. Keras - Sequential mode

- Example of model definition using sequential mode in Keras:

```
from keras.utils.vis_utils import plot_model

plot_model(model, show_shapes=True, show_layer_names=True)
```



We can use the function `plot_model()` to visualize our NN

4. Keras - Sequential mode

- Once the model is defined we need to **compile** it providing the following parameters: loss function, optimizer, and metrics:

```
model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
```

- The last step is to **train** our model using the training and validation sets:

```
history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=5)
```

```
Epoch 1/5
594/594 [==============================] - 3s 4ms/step - loss: 2.3002e-0  acy: 1.0000 -
val_loss: 0.0929 - val_accuracy: 0.9753
Epoch 2/5
594/594 [==============================] - 2s 4ms/step - los
val_loss: 0.0945 - val_accuracy: 0.9753
Epoch 3/5
594/594 [==============================] - 2s 4ms/step - los
val_loss: 0.0959 - val_accuracy: 0.9753
Epoch 4/5
594/594 [==============================] - 2s 4ms/step - los
val_loss: 0.0974 - val_accuracy: 0.9753
Epoch 5/5
594/594 [==============================] - 2s 4ms/step - loss: 6.1434e-06 - accuracy: 1.0000 -
val_loss: 0.0987 - val_accuracy: 0.9753
```

One **epoch** is when the entire dataset is passed forward and backward through the NN. Since one epoch can be too big to feed to the NN at once, we divide it in several smaller batches. The **batch size** is the number of training data present in a single batch.

4. Keras - Text classification

- Hints to create a NN for text classification with Keras:

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

model = Sequential()
model.add(Dense(30, input_dim=vocabulary_size, activation="relu"))
model.add(Dense(number_of_classes, activation="softmax"))
```

In this course, we are going to handle two text classification problems: **binary** and **multiclass**. First, we need to define the size of our NN in Keras. The table below provides some hints about it

Classifier type	Number of nodes in the input layer	Number of hidden layers	Number of nodes per hidden layer	Number of nodes in the output layer
Binary classifier	Number of features (vocabulary size)	Typically 1 or 2	?	1
Multiclass classifier			?	Number of classes

Using too few neurons in the hidden layers will result in **underfitting**, but using too much neuron in the hidden layer(s) may result in **overfitting**

4. Keras - Text classification

- Hints to create a NN for text classification with Keras:

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

model = Sequential()
model.add(Dense(30, input_dim=vocabulary_size, activation="relu"))
model.add(Dense(number_of_classes, activation="softmax"))
```

For each layer, we need to define the **activation function**. In text classification, the most convenient activation functions are depicted in the table below:

Classifier type	Layer	Activation function
Binary classifier	Hidden	`relu`
	Output	`sigmoid`
Multiclass classifier	Hidden	`relu`
	Output	`softmax`

4. Keras - Text classification

- Hints to create a NN for text classification with Keras:

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

model = Sequential()
model.add(Dense(30, input_dim=vocabulary_size, activation="relu"))
model.add(Dense(number_of_classes, activation="softmax"))

model.compile(loss="sparse_categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
```

In text classification, the most convenient parameters (loss function, optimizer, and metric) to **compile** the model are depicted in the table below:

Classifier type	Loss function	Optimizer	Metrics
Binary classifier	`binary_crossentropy`	`adam`	`accuracy`
Multiclass classifier	`sparse_categorical_crossentropy`	`adam`	`accuracy`

4. Keras - Example: multiclass classifier


```
# Dataset

from google.colab import drive
from sklearn.datasets import load_files

drive.mount("/content/drive")

# Raw data (BCC article datasets) obtained from the Insight Project
# http://mlg.ucd.ie/datasets/bbc.html
loaded_data = load_files("/content/drive/My Drive/data/bbc")

raw_dataset, y, y_names = loaded_data.data, loaded_data.target, loaded_data.target_names

print("Number of (raw) documents:", len(raw_dataset))
print("Labels (automatically generated from subfolder names):", y_names)
print("First label values:", y[:5])
```

```
Mounted at /content/drive
Number of (raw) documents: 2225
Labels (automatically generated from subfolder names): ['business', 'entertainment', 'politics', 'sport', 'tech']
First label values: [0 4 2 3 2]
```

In this example, the expected labels are already in numeric format. If this is not the case (it depends on the dataset) we should encode the labels:

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
y_enc = le.fit_transform(y)
```

4. Keras - Example: multiclass classifier


```
# Text preprocessing

import nltk
from nltk.tokenize import regexp_tokenize
from nltk.stem.snowball import SnowballStemmer
from nltk.corpus import stopwords
nltk.download("stopwords")

dataset = []
stemmer = SnowballStemmer("english")
stopwords_en = stopwords.words("english")

for i in range(0, len(raw_dataset)):
  tokens = regexp_tokenize(str(raw_dataset[i]), r"\w+")
  stems = [stemmer.stem(token) for token in tokens]
  words_no_stopwords = [word for word in stems if word not in stopwords_en]
  document = ' '.join(words_no_stopwords)
  dataset.append(document)
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

```
# Feature extraction (converting text to vectors)

from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(dataset).toarray()
```

4. Keras - Example: multiclass classifier

```
# Exploratory analysis

vocabulary_size = X.shape[1]
number_of_classes = len(y_names)

print("Number of (preprocessed) documents:", len(dataset))
print("Vocabulary size:", vocabulary_size)
print("Number of classes:", number_of_classes)
print("Vectorized dataset (number of documents, vocabulary size):", X.shape)
```

```
Number of (preprocessed) documents: 2225
Vocabulary size: 22576
Number of classes: 5
Vectorized dataset (number of documents, vocabulary size): (2225, 22576)
```

```
# Split training and testing sets

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

print("Training set shape:", X_train.shape)
print("Test set shape:", X_test.shape)
```

```
Training set shape: (1780, 22576)
Test set shape: (445, 22576)
```

4. Keras - Example: multiclass classifier

```
# Create model

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

model = Sequential()
model.add(Dense(30, input_dim=vocabulary_size, activation="relu"))
model.add(Dense(number_of_classes, activation="softmax"))

model.compile(loss="sparse_categorical_crossentropy", optimizer="adam", metrics=["accuracy"])

model.summary()
```

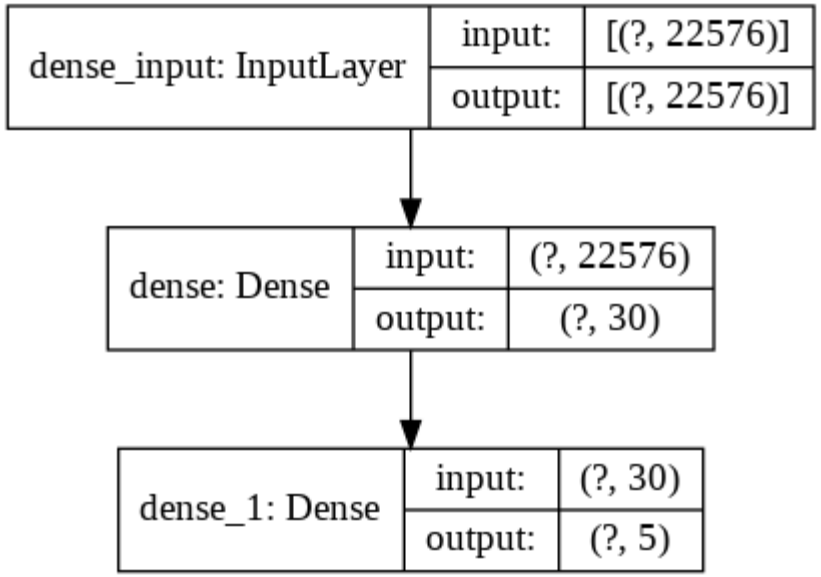
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 30)                677310
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 155
=================================================================
Total params: 677,465
Trainable params: 677,465
Non-trainable params: 0
_________________________________________________________________
```

4. Keras - Example: multiclass classifier

```
# Plot model

from keras.utils.vis_utils import plot_model

plot_model(model, show_shapes=True, show_layer_names=True)
```



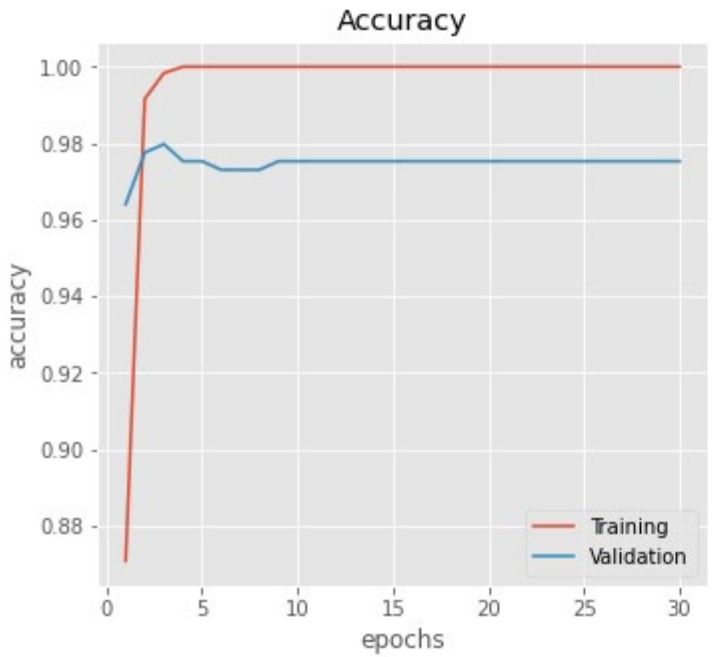
4. Keras - Example: multiclass classifier

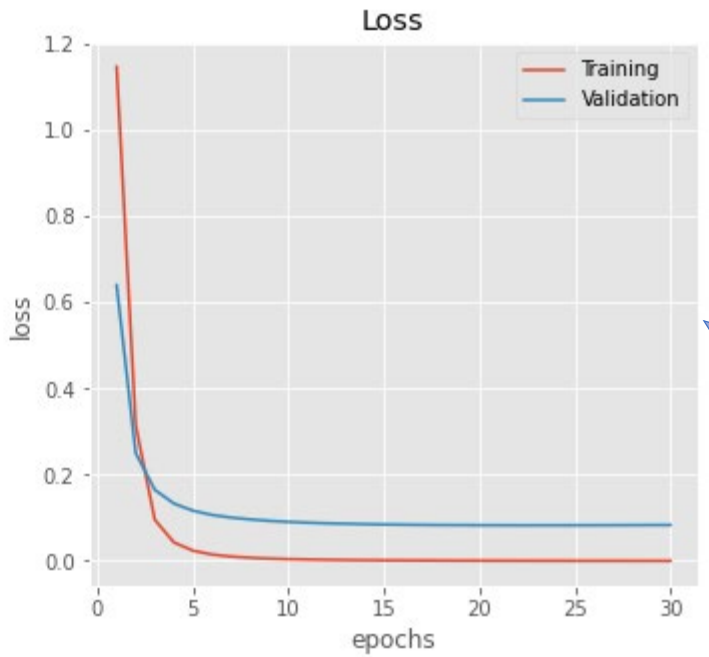
```
# Evaluate model

loss, accuracy = model.evaluate(X_train, y_train, verbose=False)
print("Training Accuracy: {:.4f}".format(accuracy))

loss, accuracy = model.evaluate(X_test, y_test, verbose=False)
print("Testing Accuracy:  {:.4f}".format(accuracy))
```

```
Training Accuracy: 1.0000
Testing Accuracy:  0.9753
```





Accuracy is a metric that describes just what percentage of your test data are classified correctly.
Loss can be seen as the distance between the true values of the problem and the values predicted by the model.

4. Keras - Example: binary classifier

```
# Dataset

from google.colab import drive
import pandas as pd
import numpy as np

drive.mount("/content/drive")

y_names = ["negative", "positive"]

# Sentiment Labelled Sentences Data Set from the UCI (University of California Irvine) Machine Learn
ing Repository
# https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences
dataset = pd.read_csv("/content/drive/My Drive/data/sentiment labelled sentences/yelp_labelled.txt",
 sep="\t", names=y_names)

y = np.array(dataset.get(y_names[1]).tolist())
raw_dataset = dataset.get(y_names[0]).tolist()

print("Number of (raw) documents:", len(raw_dataset))
print("Labels:", y_names)
print("First label values:", y[:5])
```

```
Mounted at /content/drive
Number of (raw) documents: 1000
Labels: ['negative', 'positive']
First label values: [1 0 0 1 1]
```

4. Keras - Example: binary classifier

```
# Create model

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

model = Sequential()
model.add(Dense(10, input_dim=vocabulary_size, activation="relu"))
model.add(Dense(8, activation="relu"))
model.add(Dense(1, activation="sigmoid"))

model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])

model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_12 (Dense)             (None, 10)                15760
_________________________________________________________________
dense_13 (Dense)             (None, 8)                 88
_________________________________________________________________
dense_14 (Dense)             (None, 1)                 9
=================================================================
Total params: 15,857
Trainable params: 15,857
Non-trainable params: 0
_________________________________________________________________
```

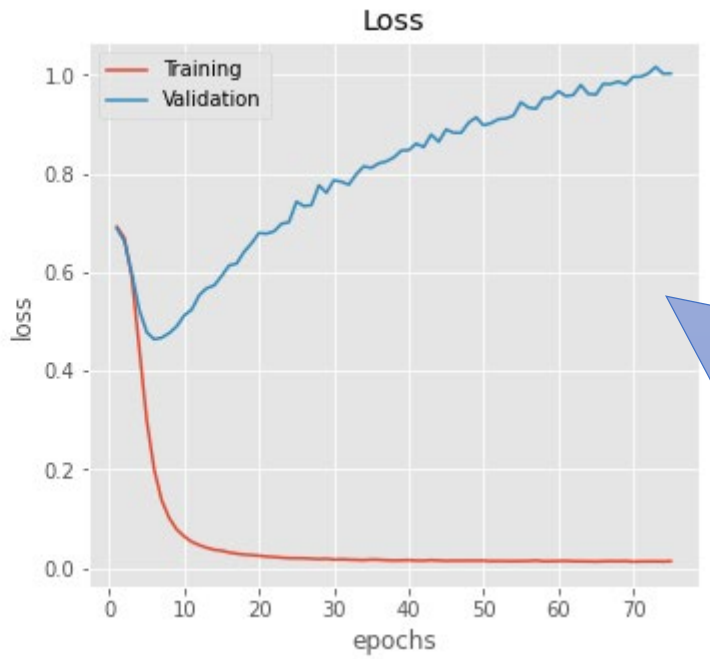
4. Keras - Example: binary classifier


```
# Evaluate model

loss, accuracy = model.evaluate(X_train, y_train, verbose=False)
print("Training Accuracy: {:.4f}".format(accuracy))

loss, accuracy = model.evaluate(X_test, y_test, verbose=False)
print("Testing Accuracy:  {:.4f}".format(accuracy))
```

```
Training Accuracy: 0.9937
Testing Accuracy:  0.7550
```



When the loss is increasing in time (epochs) during training, we have a problem of **overfitting**. To solve this problem, we can try different alternatives:

- Dropout neurons (ignoring some nodes during training)
- Change number of parameters (number of layers and/or nodes per layer)
- Early stopping (reduce number of epochs before the loss increases)

4. Keras - Fine-tuning

- We can try fine-tuning the various of the model hyperparameters, including the following:
  - Number of hidden layers
  - Number of neurons in each layer
  - Different activation functions
  - Different optimizers
  - Batch size
  - The number of epochs

Table of contents

5. Takeaways

- Neural Networks (NNs) are computing systems that attempts to make predictions using a network that mimics the human brain
- Like in ML, a NN should be trained and validated using a proper dataset (a collection of documents in the case of NLP)
- In this course, we have used Keras to create NNs in Python in two types of text classifiers: binary and multiclass
- The parameters we need to tune in a NN with Keras are: number of hidden layers, number of nodes on the output layer, activation and loss functions, optimizer, and metrics
- If overfitting happens, we need to change number of parameters of the NN (number of layers and/or nodes per layer), add dropout neurons, or include an early stopping callback