

# COS30018 – Intelligent Systems

## Week 6 tutorial

This week we will cover the following items:

- Deep learning libraries/frameworks introduction (TensorFlow, PyTorch)
- Neural Network fundamentals
- Your first Neural Network implementation to recognise handwritten digits

### 1. Deep learning libraries/frameworks introduction (TensorFlow, Pytorch)

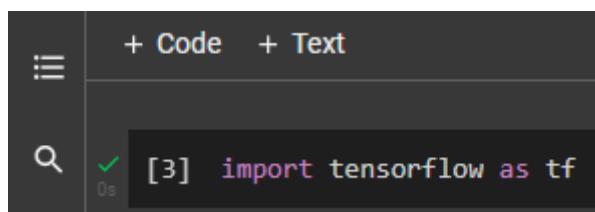
A deep learning framework is an interface that allows developers to build and deploy deep learning models faster and easier. A tool like this allows enterprises to scale their machine learning efforts securely while maintaining a healthy ML/DL lifecycle. Deep learning frameworks have become standard practice in recent years. They provide democratization in the development of ML/DL algorithms while also speeding up the process.

#### TensorFlow:

Created by the Google Brain team and initially released to the public in 2015, TensorFlow is an open source library for numerical computation and large-scale machine learning. TensorFlow bundles together a slew of machine learning and deep learning models and algorithms (aka neural networks) and makes them useful by way of common programmatic metaphors. TensorFlow also has a broad library of **pre-trained models** that can be used in your own projects. You can also use code from the **TensorFlow Model Garden** as examples of best practices for training your own models.

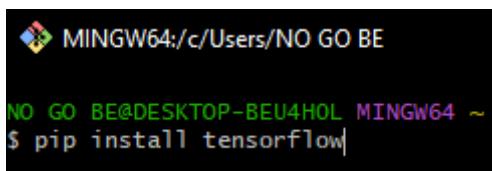
#### Using TensorFlow:

If you are using Google Colab, you do not need to install anything because TensorFlow is already installed on Colab for you. You just need to import to use it:



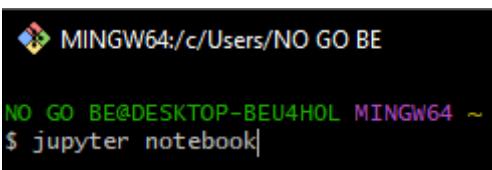
A screenshot of a Jupyter Notebook cell. The cell has a dark background with light-colored text. At the top, there are two buttons: '+ Code' and '+ Text'. Below the buttons, there is a search bar with a magnifying glass icon and the text '0s'. The main area of the cell contains the Python code: '[3] import tensorflow as tf'. The code is highlighted in blue, and there is a green checkmark icon to its left.

If you run Jupyter Notebook on your local machine, you need to install TensorFlow by running “**pip install tensorflow**” in your command line:



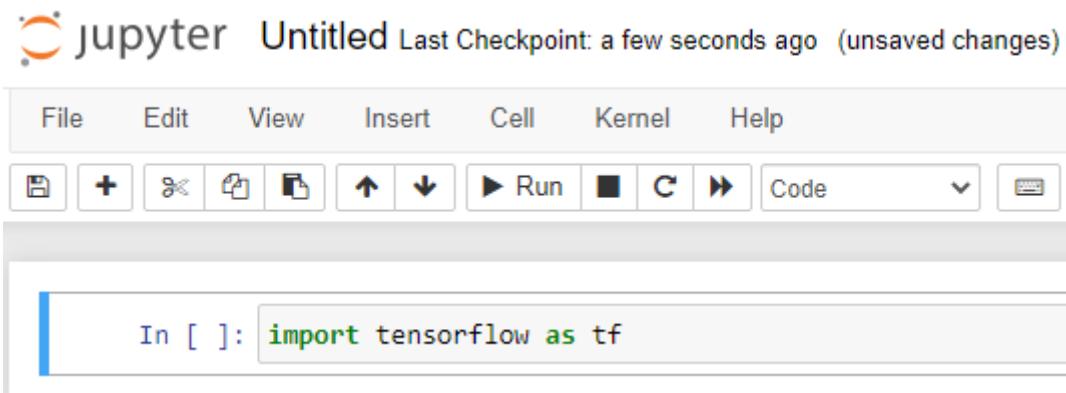
A screenshot of a terminal window with a black background and white text. The window title is 'MINGW64:/c/Users/NO GO BE'. The prompt shows 'NO GO BE@DESKTOP-BEU4HOL MINGW64 ~'. The user has typed the command '\$ pip install tensorflow' and is waiting for the command to complete.

Then, you can open your Jupyter Notebook by typing the following in the command line:



A screenshot of a terminal window with a black background and white text. The window title is 'MINGW64:/c/Users/NO GO BE'. The prompt shows 'NO GO BE@DESKTOP-BEU4HOL MINGW64 ~'. The user has typed the command '\$ jupyter notebook' and is waiting for the command to complete.

Create a new notebook and now you can start using TensorFlow by importing it like this:



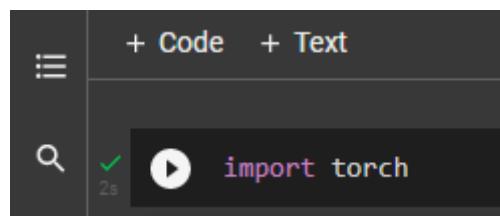
The screenshot shows a Jupyter Notebook interface. At the top, there's a menu bar with File, Edit, View, Insert, Cell, Kernel, and Help. Below the menu is a toolbar with various icons for file operations like Open, Save, and Run. The main area is a code cell labeled "In [ ]:" containing the Python code "import tensorflow as tf".

### PyTorch:

Created by the Facebook AI Research (FAIR) team in 2017, PyTorch has become a highly popular and efficient framework to create Deep Learning (DL) model. This open-source machine learning library is based on Torch and designed to provide greater flexibility and increased speed for deep neural network implementation. Currently, PyTorch is one of the most favoured libraries for AI (Artificial Intelligence) researchers and practitioners worldwide in the industry and academia, besides TensorFlow.

### Using PyTorch:

Same as TensorFlow, if you are using Google Colab, then you do not need to install anything because Colab already has it installed when you start the session. You just need to import to use it:



If you run Jupyter Notebooks on your local machine, you need to install PyTorch. PyTorch has a very nice installation guide that provides you the exact command to install the library depends on your OS. The installation guide is here <https://pytorch.org/>:

## INSTALL PYTORCH

Select your preferences and run the install command. Stable represents the most currently tested and supported version of PyTorch. This should be suitable for many users. Preview is available if you want the latest, not fully tested and supported, 1.12 builds that are generated nightly. Please ensure that you have **met the prerequisites below (e.g., numpy)**, depending on your package manager. Anaconda is our recommended package manager since it installs all dependencies. You can also [install previous versions of PyTorch](#). Note that LibTorch is only available for C++.

Additional support or warranty for some PyTorch Stable and LTS binaries are available through the [PyTorch Enterprise Support Program](#).

PyTorch Build	Stable (1.12.0)		Preview (Nightly)		LTS (1.8.2)			
Your OS	Linux		Mac		Windows			
Package	Conda		Pip	LibTorch		Source		
Language	Python		C++ / Java					
Compute Platform	CUDA 10.2	CUDA 11.3	CUDA 11.6	ROCM 5.1.1	CPU			
Run this Command:	<pre>pip3 install torch torchvision torchaudio --extra-index-url https://download.pytorch.org/whl/cu116</pre>							
<a href="#">Previous versions of PyTorch &gt;</a>								

Note: Installing PyTorch with CUDA compute platform allows you to utilise GPU when training models, which is way faster than training it on CPU.

Next step is to run the generated command on your command prompt. Remember if you make any changes in this command, it will not install PyTorch and give an error message.

After that, you can run pip list command to check PyTorch is installed successfully or not:

```
C:\Python35\Scripts>pip list
numpy (1.16.4)
Pillow (6.0.0)
pip (8.1.1)
scipy (1.3.0)
setuptools (20.10.1)
six (1.12.0)
torch (1.1.0)
torchvision (0.3.0)
You are using pip version 8.1.1, however version 19.1.1 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
```

Now you can import PyTorch in your notebook and use it:

# jupyter Untitled Last Checkpoint: a few seconds ago (unsaved changes)

```
File Edit View Insert Cell Kernel Help  
Run Code
```

In [ ]: `import torch`

## 2. Neural Network fundamentals

### Components of Neural Network:

- Layers: Input, Hidden, and Output layers
- Neurons
- Activation Function
- Weights and Bias

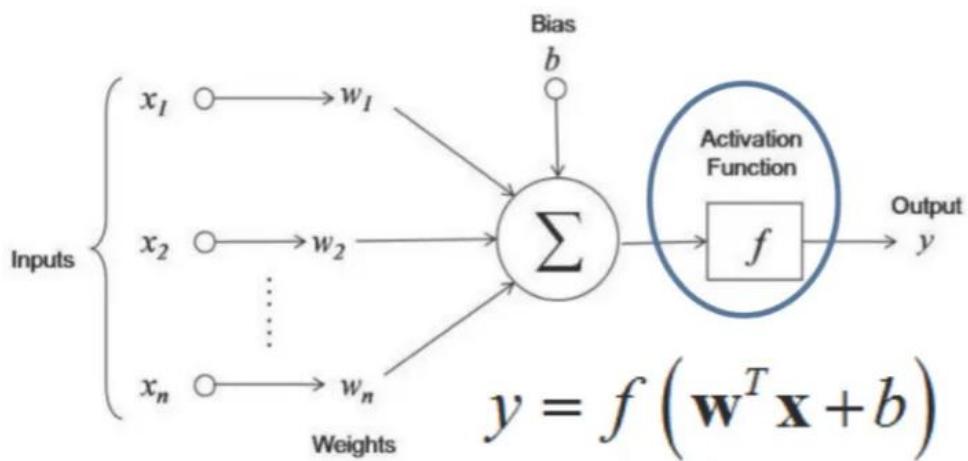
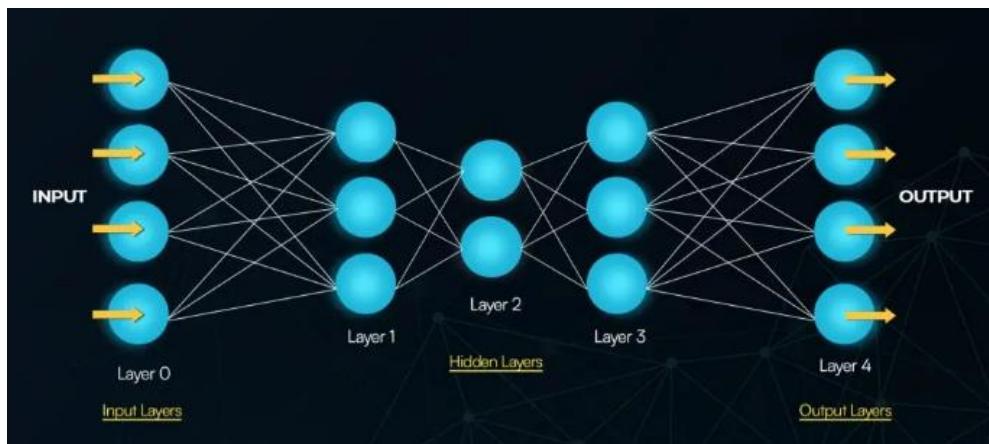


Figure 1. Components of Neural Network

Layers:

- **Input layer:** receive the input information such as text, images, audio, ...
- **Hidden layer:** is an intermediate layer allowing neural networks to learn. This said to be "hidden" because there is no direct contact with the outside world. The outputs of each hidden layer are the inputs of the units of the following layer.
  - Single Layer Perceptron: The neural net with one single hidden layer
  - Multilayer Perceptron: The neural net with more than one hidden layer
- **Output layer:** produces the result (the prediction).

### Neuron:

Neurons are the primary and basic processing unit in the neural network. It receives information or data, performs simple calculations, and then passes it further. The neurons are in the hidden and the output layers. The input layer **DOESN'T** have any neurons; the circles in the input layer represent the input features.

The number of nodes in the output layer depends on the kind of the business problem:

- **Regression:** there will be one node in the output layer that predicts the continuous value.
- **Classification:** the nodes in the output layer are equal to the number of the classes or the categories. For binary classification, we can either have one or two nodes in the output layer.

The number of neurons in the hidden layers is subject to the user.

Structure of a neuron:

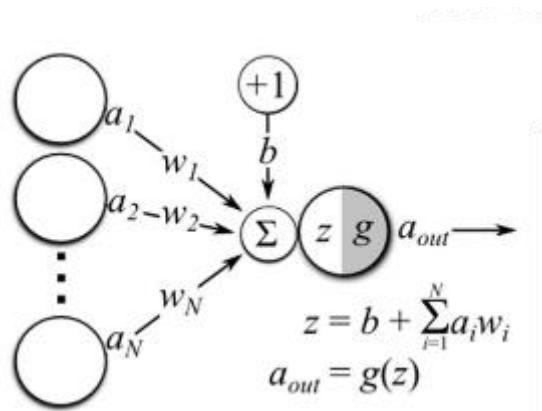


Figure 2. Structure of a neuron

### Weights and Bias

Any given neuron can have many to many relationships with multiple inputs and output connections. Weights and bias are applied to each of the connections during the nodes' transmission between the layers.

These weights represent the relative importance of the neural net and meaning indicates how much precedence the input  $X$  or the subsequently derived neurons will have on the output. This will naturally lead to the neurons with the higher weight will have more influence in the next layer, and ultimately, the neurons with not significant weights will get dropped out.

Bias is an additional input to each layer. The bias is not dependent nor impacted by the preceding layer. In simple words, bias is the intercept term and is constant. It implies that even if there are no inputs or independent variables, the model will be activated with a default value of the bias.

The weights and biases are the learnable parameters of the model. At first, the weights + bias are randomly generated, then they will be optimised during training to reduce the error.

### Activation Function

In figure 2, we have:

$$z = \sum_{i=1}^N (\text{weight} * \text{input}) + \text{bias}$$

The above equation is a linear combination of the weights and the bias. This will be an overly simplified neural network. It will not perform any complex computations or detect any patterns. To prepare the model for such complexity, we need transformation, where activation functions step in.

Activation Functions are an integral part of Neural Networks. The purpose of the activation function is to bring non-linearity to the net by applying the transformation.

$$a_{out} = g(z)$$

where  $g$  is an activation function.

There are different activation functions available depending upon their functionality and the layer to be applied upon. To know more about these functions and their applicability can refer to these blogs [here](#) and [here](#).

### Cost/loss function

A **loss function** is a function that **compares** the target and predicted output values; measures how well the neural network models the training data. When training, we aim to minimize this loss between the predicted and target outputs.

The average loss is the average of loss values on all output nodes.

$$J(w^T, b) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)})$$

The **hyperparameters** are adjusted to minimize the average loss — we find the weights ( $w^T$ ) and biases ( $b$ ) that minimize the value of  $J$  (average loss).

A few mostly used loss functions:

- **Mean Squared Error (MSE):** MSE finds the average of the squared differences between the target and the predicted outputs

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

- **Mean Absolute Error (MAE):** MAE finds the average of the absolute differences between the target and the predicted outputs

$$MAE = \frac{1}{n} \sum_{i=1}^n |y^{(i)} - \hat{y}^{(i)}|$$

- **Binary Cross-Entropy/Log Loss:** This is the loss function used in binary classification models - where the model takes in an input and has to classify it into one of two pre-set categories.

$$CE\ Loss = \frac{1}{n} \sum_{i=1}^N - (y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i))$$

- **Categorical Cross-Entropy Loss:** In cases where the number of classes is greater than two, we utilize categorical cross-entropy — this follows a very similar process to binary cross-entropy

$$CE\ Loss = -\frac{1}{n} \sum_{i=1}^N \sum_{j=1}^M y_{ij} \cdot \log(p_{ij})$$

## Backpropagation

This is the most popular and conventional method to train a Neural Network.

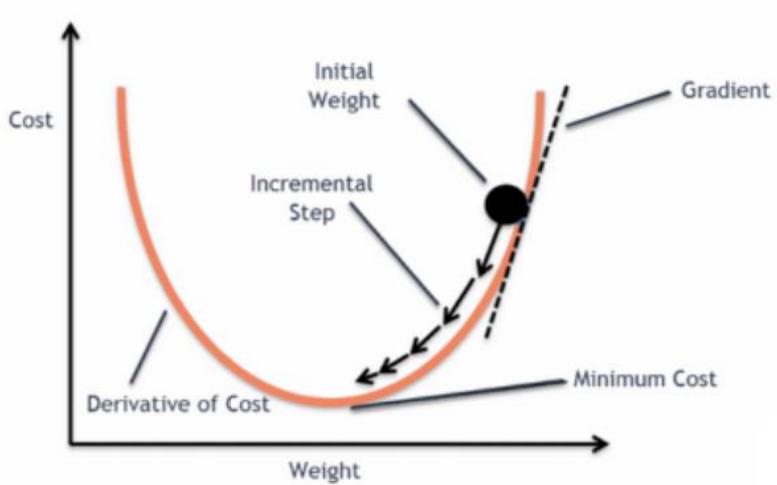
After doing the feedforward propagation and calculate the loss. We will propagate the loss backward to adjust the weights. And the weights are adjusted using an algorithm so called Gradient Descent. This is how Gradient Descent work:

$$W := W - \alpha \frac{\partial J}{\partial W}$$

Where:

- $\alpha$ : learning rate
- $W$ : weight
- $J$ : cost function
- $\frac{\partial J}{\partial W}$ : partial derivative of  $J$  with respect to  $W$ .

The intuition of the above equation:

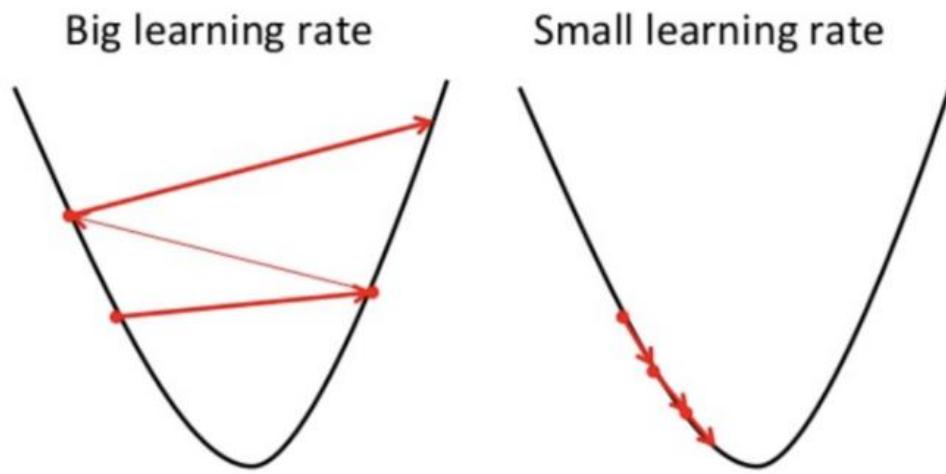


**Partial Derivative:** We use partial derivatives to get the slope of a function at a given point.

Importance of the learning rate:

How big the steps the gradient descent takes into the direction of the local minimum are determined by the learning rate, which figures out how fast or slow we will move towards the optimal weights.

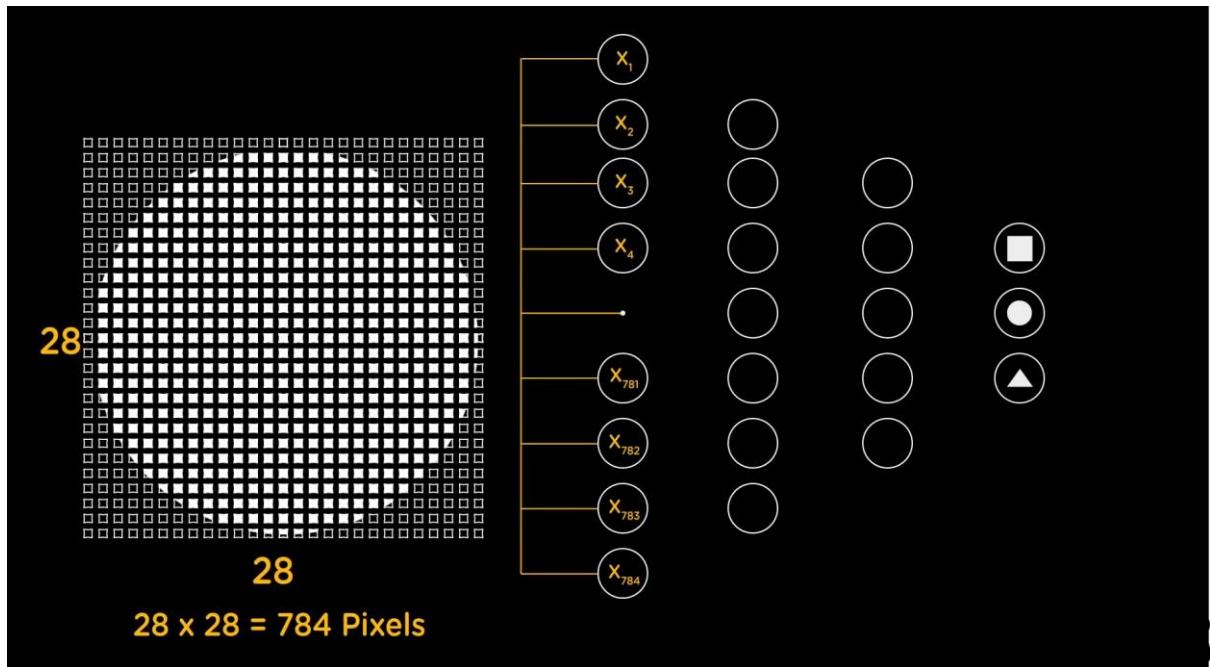
For Gradient Descent (GD) to converge, we must set the learning rate to an appropriate value, which is neither too low nor too high. Because if the steps it takes are too big, it may not reach the local minimum because it bounces back and forth between the convex function of gradient descent. If we set the learning rate to a very small value, gradient descent will eventually reach the local minimum but that may take a while.



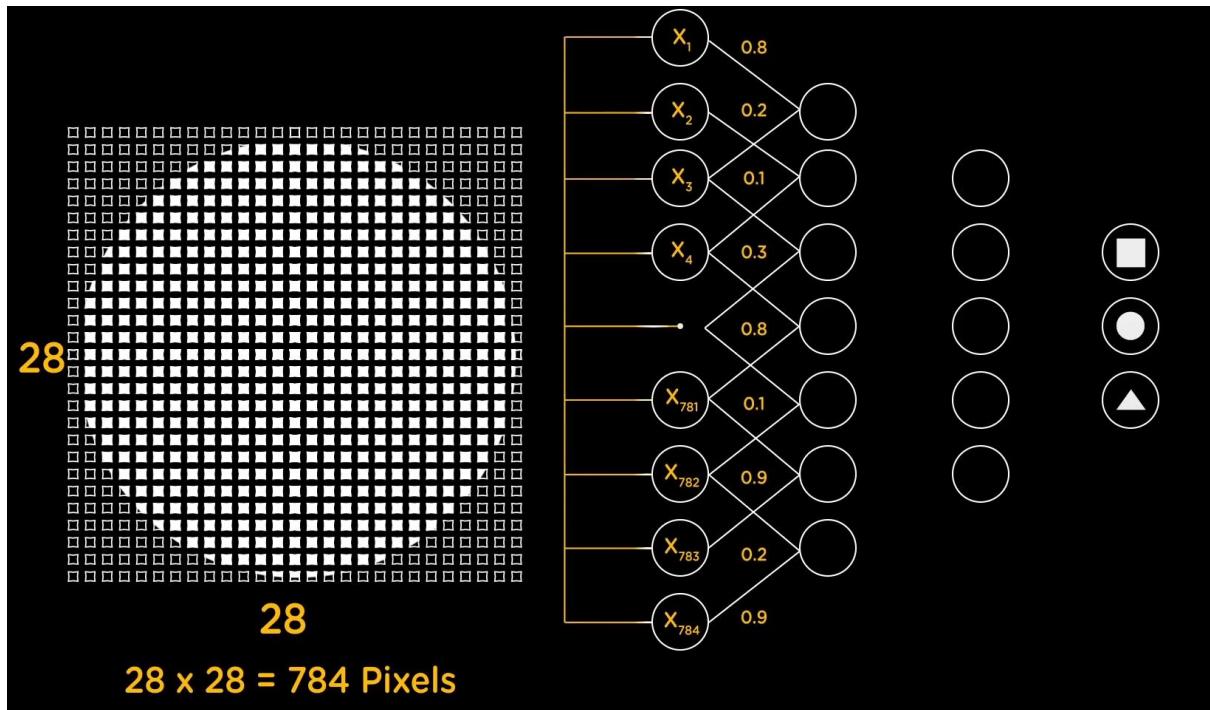
**Let's look at an example of how a neural network is trained:**

Let's say we're training a neural network to recognise an image of square, circle, and triangle. And the current image is a circle.

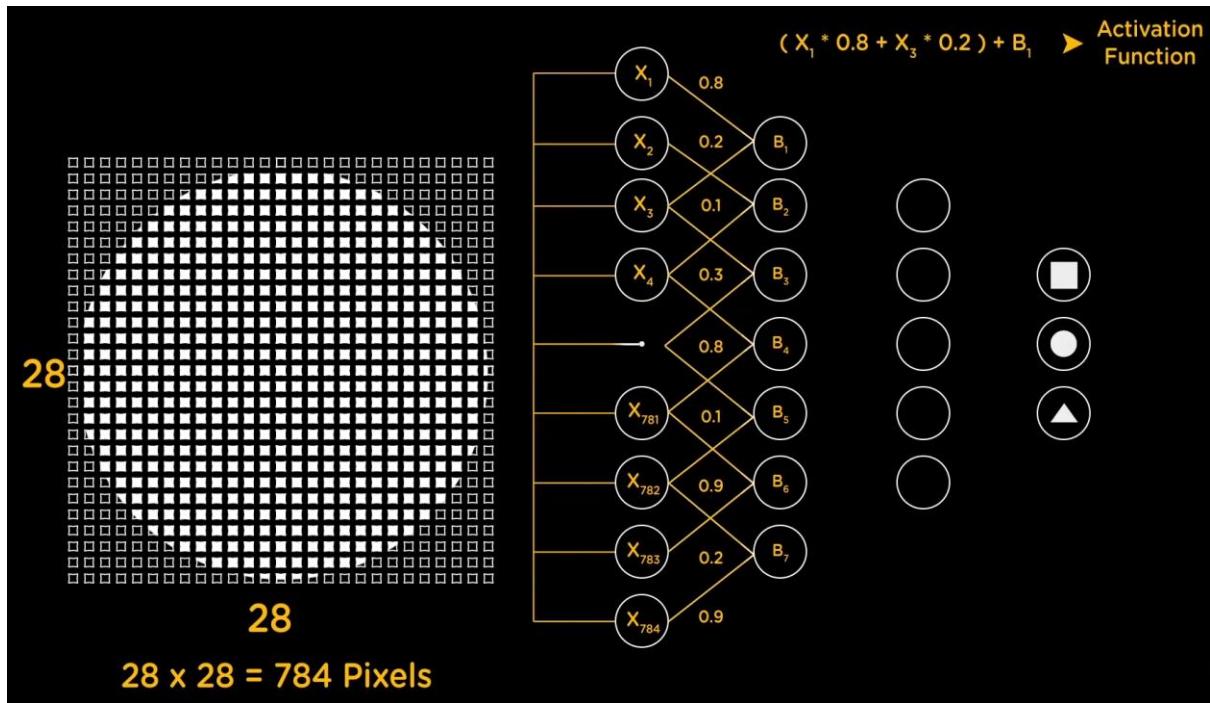
Because the image is 28x28 pixels, therefore we have  $28 \times 28 = 784$  values in the input layer:



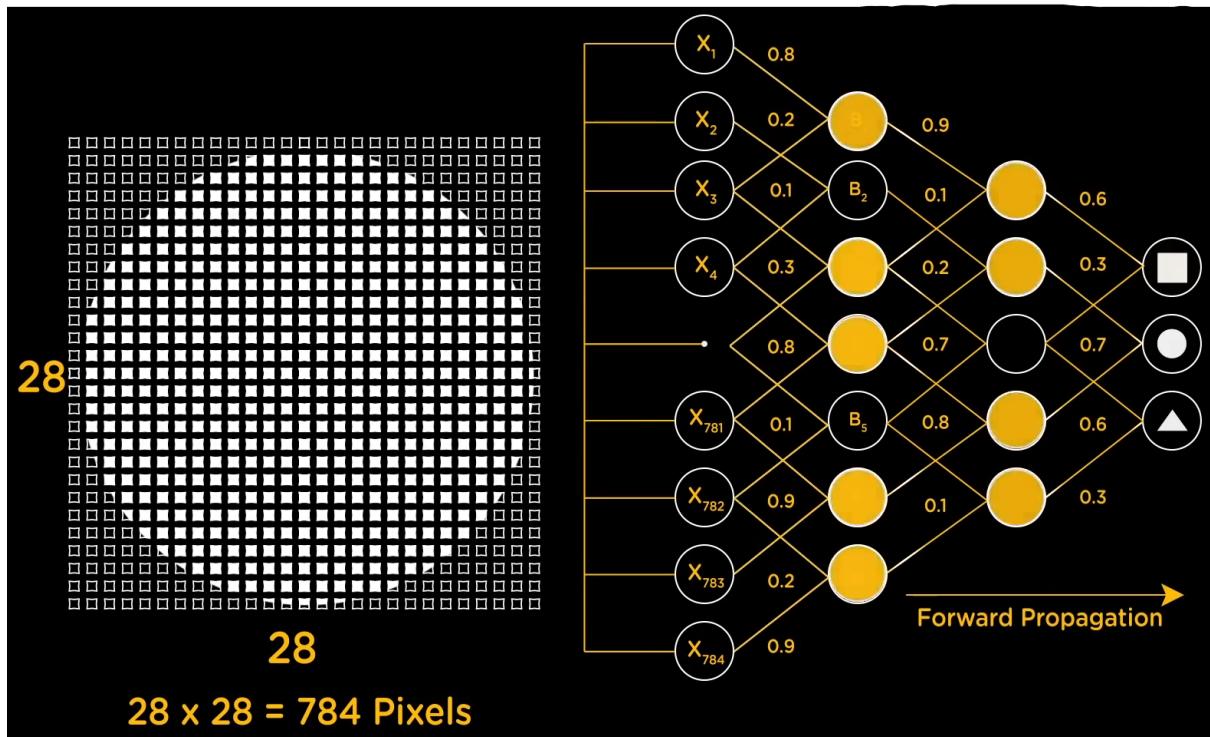
Then the weights of the first layer is randomly initialised:



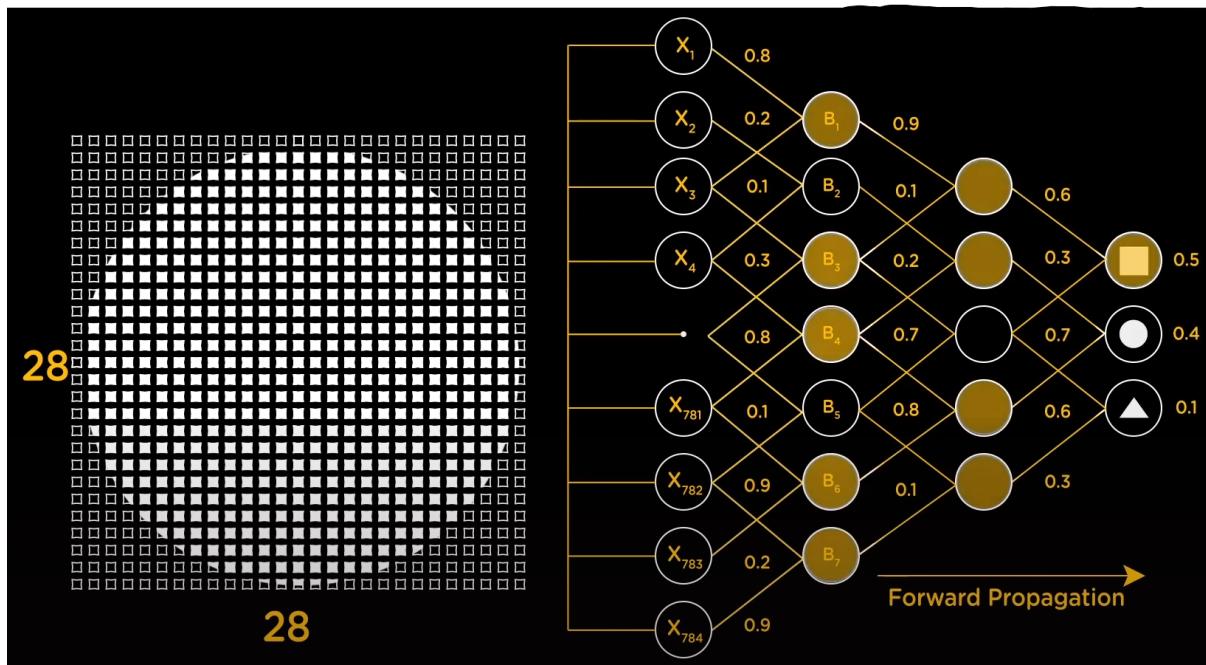
The value of each node in the subsequent layer is calculated by multiplying the output value of the previous nodes that are connected to the current node with the corresponding weights. Then plus the bias and feed that result over an activation function:



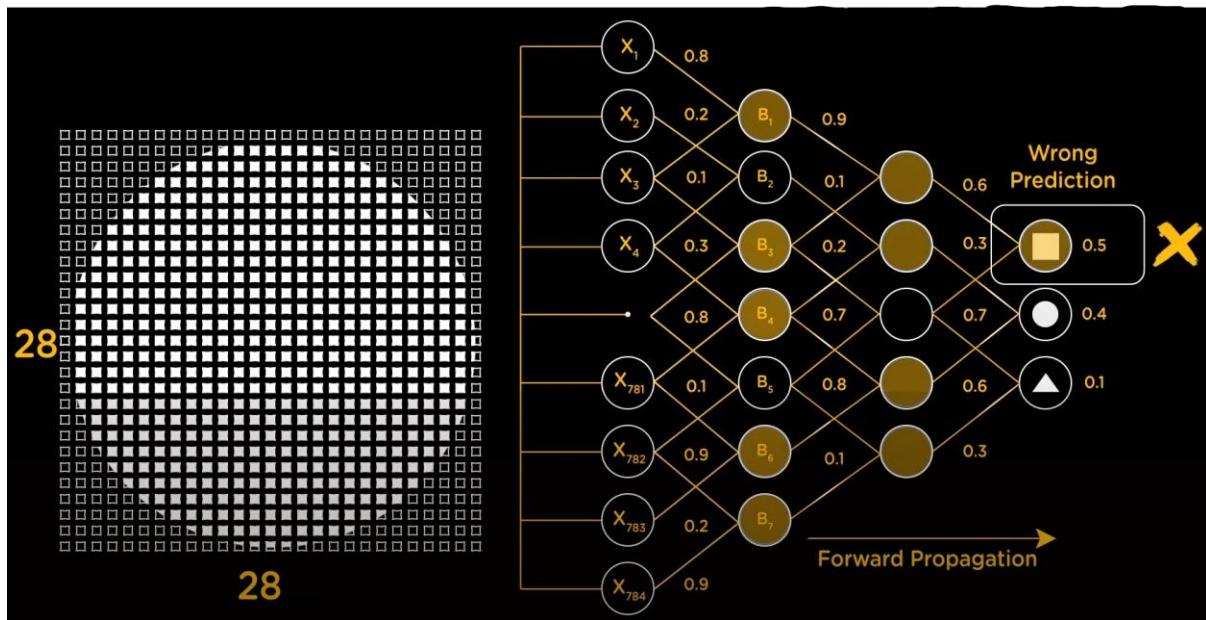
The result of the activation function determines if the particular neuron will get activated or not. Activated neurons transmit data to neurons in the next layer:



In the output layer, the neuron with the highest value determines the output. The values are probabilities of the image being a particular shape. For example here, the value associated with square has the highest probability = 50%. Hence, that's the output our neural network predicted.

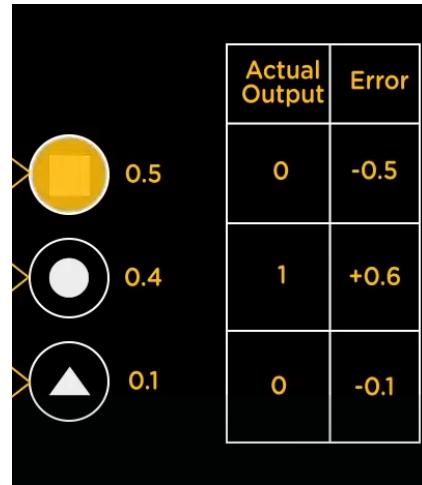


We can see that our NN made a wrong prediction:



But remember that our weights and bias were randomly generated, we haven't trained it. That's why the result is not accurate.

Now we have to perform backpropagation to update the weights and bias in the direction of reducing the average loss. First, we calculate the loss at each output node. The magnitude indicates how wrong we are, and the sign suggests if our predicted values are higher/lower than the expected value.



Based on this error, the weights and biases are adjusted using Gradient Descent.

This forward and backward propagation is repeated multiple times until we get the set of weights and biases such that the network can predict the shapes correctly in most of the time.

#### **How long does it take a NN to be trained:**

Depends on the data and the network structure, it can take minutes, to hours, and even to months (such as BERT, GPT-3, ...)

### **3. Implement NN to recognise handwritten digits**

Let's implement a simple NN using TensorFlow to recognise handwritten digits. The code and explanation of each step is given in the file "neural\_net.ipynb"

For more intuition about Neural Network with handwritten digits recogniser, I find this video is a great resource:

[https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQBOWTQDNU6R1\\_67000Dx\\_ZCJB-3pi](https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQBOWTQDNU6R1_67000Dx_ZCJB-3pi)