UNIT V:

Neural Networks and Deep Learning: Introduction to Artificial Neural Networks with Keras, Implementing MLPs with Keras, Installing TensorFlow 2, Loading and Preprocessing Data with TensorFlow.

---

1. INTRODUCTION TO ARTIFICIAL NEURAL NETWORKS WITH KERAS:

Artificial Neural Network:

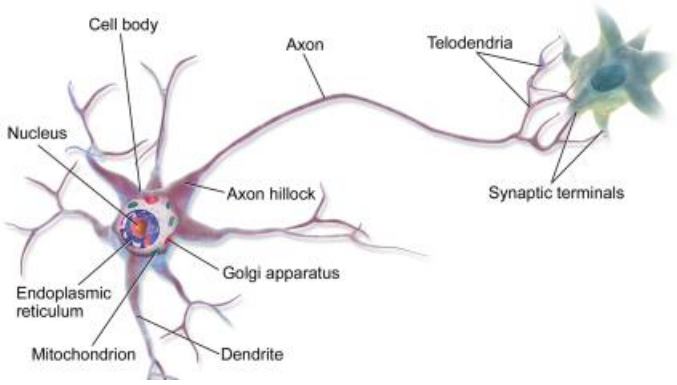
- A Neural Network is a system designed to operate like a human brain. Human information processing takes place through the interaction of many billions of neurons connected to each other sending signals to other neurons.
- Similarly, a Neural Network is a network of artificial neurons, as found in human brains, for solving artificial intelligence problems such as image identification. They may be a physical device or mathematical constructs.
- In other words, Artificial Neural Network is a parallel computational system consisting of many simple processing elements connected to perform a particular task.

From Biological to Artificial Neurons:

- ANNs were first introduced back in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts.
- The early successes of ANNs until the 1960s led to the widespread belief that we would soon be conversing with truly intelligent machines.
- In the early 1980s there was a renewal of interest in connectionism, as new architectures were invented and better training techniques were developed.
- But progress was slow, and by the 1990s other powerful Machine Learning techniques were invented, such as Support Vector Machines.
- These techniques seemed to offer better results and stron- ger theoretical foundations than ANNs.
- Finally, we are now witnessing yet another wave of interest in ANNs.
- There are a few good reasons to believe that this wave is different and that it will have a much more profound impact on our lives:
    1. There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
    2. The training algorithms have been improved.
    3. Some theoretical limitations of ANNs have turned out to be benign in practice.

Biological Neurons:

- It is an unusual-looking cell mostly found in animal cerebral cortexes (e.g., your brain), composed of a cell body containing the nucleus and most of the cell's complex components, and many branching extensions called dendrites, plus one very long extension called the axon.
- Near its extremity the axon splits off into many branches called telodendria, and at the tip of these branches are minuscule structures called synaptic terminals which are connected to the dendrites of other neurons.
- Biological neurons receive short electrical impulses called signals from other neurons via these synapses.
- When a neuron receives a sufficient number of signals from other neurons within a few milliseconds, it fires its own signals.



Logical Computations with Neurons:

- Warren McCulloch and Walter Pitts proposed a very simple model of the biological neuron, which later became known as an artificial neuron: it has one or more binary (on/off) inputs and one binary output.
- The artificial neuron simply activates its out- put when more than a certain number of its inputs are active.
- For example, let's build a few ANNs that perform various logical computations, assuming that a neuron is activated when at least two of its inputs are active.

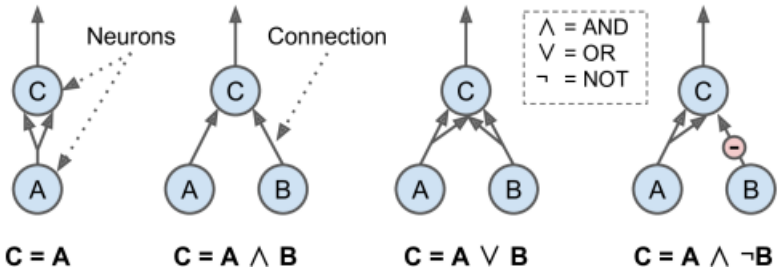


Figure: ANNs performing simple logical computations

- The first network on the left is simply the identity function: if neuron A is activated, then neuron C gets activated as well (since it receives two input signals from neuron A), but if neuron A is off, then neuron C is off as well.
- The second network performs a logical AND: neuron C is activated only when both neurons A and B are activated (a single input signal is not enough to activate neuron C).
- The third network performs a logical OR: neuron C gets activated if either neuron A or neuron B is activated (or both).
- Finally, if we suppose that an input connection can inhibit the neuron's activity (which is the case with biological neurons), then the fourth network computes a slightly more complex logical proposition: neuron C is activated only if neuron A is active and if neuron B is off. If neuron A is active all the time, then you get a logical NOT: neuron C is active when neuron B is off, and vice versa.

The Perceptron:

- The Perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt.
- It is based on a slightly different artificial neuron called a threshold logic unit (TLU), or sometimes a linear threshold unit (LTU): the inputs and output are now numbers (instead of binary on/off values) and each input connection is associated with a weight.
- The TLU computes a weighted sum of its inputs ($z = w1\ x1 + w2\ x2 + \cdots + wn\ xn = x^T w$), then applies a step function to that sum and outputs the result: $h_w(x) = \text{step}(z)$, where $z = x^T w$.

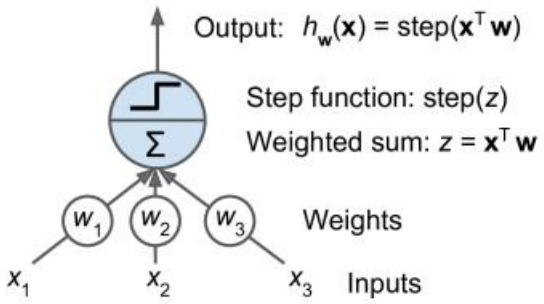


Figure: Threshold logic unit

- The most common step function used in Perceptrons is the Heaviside step function. Common step functions used in Perceptrons are

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

- A Perceptron is simply composed of a single layer of TLUs, with each TLU connected to all the inputs.
- When all the neurons in a layer are connected to every neuron in the previous layer (i.e., its input neurons), it is called a fully connected layer or a dense layer.
- To represent the fact that each input is sent to every TLU, it is common to draw special passthrough neurons called input neurons: they just output whatever input they are fed. All the input neurons form the input layer.
- an extra bias feature is generally added (x0 = 1): it is typically represented using a special type of neuron called a bias neuron, which just outputs 1 all the time.

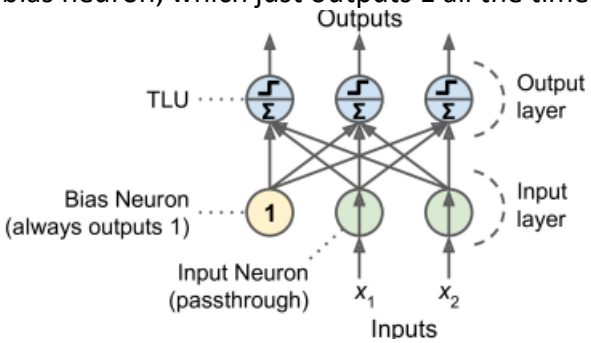
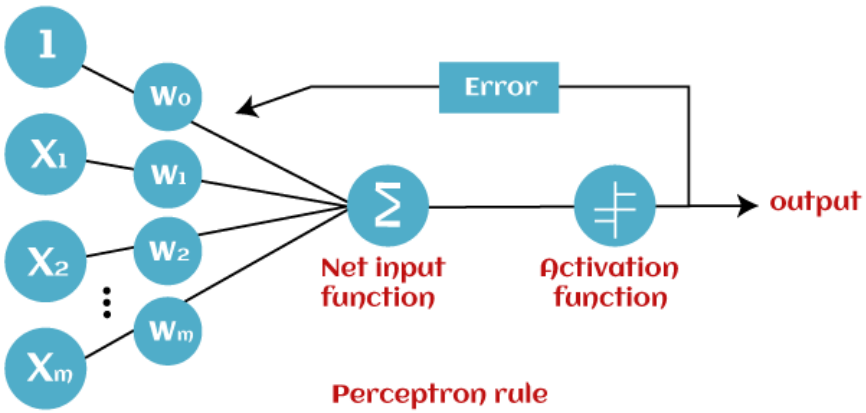


Figure: Perceptron diagram

Perceptron Training Rule:

- The perceptron model begins with the multiplication of all input values and their weights, then adds these values together to create the weighted sum.
- Then this weighted sum is applied to the activation function 'f' to obtain the desired output. This activation function is also known as the **step function** and is represented by **'f'**.



- This step function or Activation function plays a vital role in ensuring that output is mapped between required values (0,1) or (-1,1). It is important to note that the weight of input is indicative of the strength of a node. Similarly, an input's bias value gives the ability to shift the activation function curve up or down.
- Perceptron model works in two important steps as follows:

  Step-1: In the first step, multiply all input values with corresponding weight values and then add them to determine the weighted sum. Mathematically, we can calculate the weighted sum as follows:

  $$\sum wi*xi = x1*w1 + x2*w2 + \ldots wn*xn$$

  Step-2: In the second step, an activation function is applied with the above-mentioned weighted sum, which gives us output either in binary form or a continuous value as follows:
  - If the Output o=t (target value), then we no need to change the weights which are chosen randomly.
  - If the output o!= t, then we need to change the weight values as follows:
    - $W_i = w_i + \Delta w_i$
    - $\Delta w_i = n(t-o)\, x_i$

  Where n is the learning rate(constant value generally 0.1,0.2….)
  - The above process is repeated until o=t

Multi-layer Perceptron and Back Propagation:

- An MLP is composed of one input layer , one or more layers of TLUs or “Hidden Layers” and one output layer.
- The layers close to the input layers are called as “Lower Layer” and the one that are close to the output layers are called “Upper Layer”.
- Every layer except the output layer includes a bias neuron.
- The multi-layer perceptron model is also known as the Backpropagation algorithm, which executes in two stages as follows:

  Forward Stage: Activation functions start from the input layer in the forward stage and terminate on the output layer.

  Backward Stage: In the backward stage, weight and bias values are modified as per the model's requirement. In this stage, the error between actual output and demanded originated backward on the output layer and ended on the input layer.

Advantages of Multi-Layer Perceptron:

- A multi-layered perceptron model can be used to solve complex non-linear problems.
- It works well with both small and large input data.
- It helps us to obtain quick predictions after the training.
- It helps to obtain the same accuracy ratio with large as well as small data.

Disadvantages of Multi-Layer Perceptron:

- In Multi-layer perceptron, computations are difficult and time-consuming.

- In multi-layer Perceptron, it is difficult to predict how much the dependent variable affects each independent variable.
- The model functioning depends on the quality of the training.

2. IMPLEMENTING MLPS WITH KERAS:

In Keras we can build the model in several ways:

1. Sequential API : As the name suggest, this API is to build the model in sequential style, i.e. output of one layer will be fed to the later layer.

2. Function API : This API allows us to completely control the flow of data (doesn't need to be sequential).

3. Subclassing the tf.keras.models class : This will gives a complete control of the overall model. We can have loops, conditional branching, and other dynamic behaviors in the model.

1. Sequential API:

- Multi-Layer Perceptron (MLP) is a type of artificial neural network that consists of several layers of nodes, where each node is a neuron that performs a nonlinear operation on its input.
- Keras is a popular high-level neural networks API that makes it easy to build, train, and deploy deep learning models.
- First, we need to install Keras and its dependencies. You can do this by running the following command in your terminal:
  pip install keras
- Once you have installed Keras, you can start building your MLP. Here's an example of a simple MLP with one hidden layer:

```
from keras.models import Sequential
from keras.layers import Dense
# define the model architecture
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=100))
model.add(Dense(10, activation='softmax'))
# compile the model
model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
# train the model
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

- In the above code, we first import the necessary Keras modules, which include the Sequential model and the Dense layer.
- Then, we define the architecture of our MLP by adding layers to our model.
- The first layer is a dense layer with 64 neurons, ReLU activation, and an input dimension of 100. The second layer is a dense layer with 10 neurons and a softmax activation.
- Next, we compile the model by specifying the loss function, the optimizer, and the evaluation metric.
- In this example, we use categorical cross-entropy as the loss function, stochastic gradient descent (SGD) as the optimizer, and accuracy as the evaluation metric.
- Finally, we train the model by calling the fit() method on our model object, passing in our training data (x_train and y_train), the number of epochs to train for, and the batch size to use during training.

2. Function API:

- In addition to the Sequential API, Keras also provides a Functional API for building neural networks. The Functional API offers more flexibility and allows you to build more complex models with shared layers, multiple inputs, and multiple outputs.

- Let's start by importing the necessary modules:

```
from keras.layers import Input, Dense
from keras.models import Model
```

- Next, we define the architecture of our MLP using the Functional API. Here's an example of a MLP with one hidden layer:

```
a = Input(shape=(100,))
x = Dense(64, activation='relu')(a)
b = Dense(10, activation='softmax')(x)
model = Model(inputs=a, outputs=b)
```

- In the above code, we first define the input shape of our MLP by creating an Input object with the shape (100,).
- Then, we define the first hidden layer by creating a Dense object with 64 neurons and a ReLU activation function.
- We connect this layer to the input layer by calling it with the input object a.
- This returns a new object, which we assign to the variable x.
- Next, we define the output layer by creating another Dense object with 10 neurons and a softmax activation function.
- We connect this layer to the previous layer (x) by calling it with x. This returns a new object, which we assign to the variable b.
- Finally, we create a Model object by specifying the input and output layers. We pass the input object to the inputs parameter, and the output object to the outputs parameter.
- We can compile and train the model in the same way as we did with the Sequential API:

```
model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

3. Subclassing API:

- Keras also provides a Subclassing API for building neural networks. The Subclassing API allows for even greater flexibility and customization than the Sequential and Functional APIs. In this tutorial, we will learn how to implement MLPs with the Keras Subclassing API.
- Let's start by importing the necessary modules:

```
from keras.layers import Layer, Dense
from keras.models import Model
```

- Next, we define a custom class for our MLP by subclassing the Layer class:

```
class MLP(Layer):
    def __init__(self):
        super(MLP, self).__init__()
        self.dense1 = Dense(64, activation='relu')
        self.dense2 = Dense(10, activation='softmax')
    def call(self, inputs):
        x = self.dense1(inputs)
        x = self.dense2(x)
        return x
```

- In the above code, we define a custom class called MLP, which subclasses the Layer class. In the constructor (init), we create two Dense layers, one with 64 neurons and a ReLU activation function, and another with 10 neurons and a softmax activation function. We store these layers as member variables of the class using the self keyword.
- In the call method, we define the forward pass of our MLP. We first pass the input tensor (inputs) through the first Dense layer (dense1), and then pass the output through the second Dense layer (dense2). We return the output tensor (x).

- We can create an instance of our MLP class and use it to build a Keras model:

```
a = Input(shape=(100,))
mlp = MLP()
b = mlp(a)
model = Model(inputs=a, outputs=b)
```

- In the above code, we first define the input shape of our MLP by creating an Input object 'a' with the shape (100,).
- Then, we create an instance of our MLP class called mlp, and pass the input object 'a' to it.
- This returns the output tensor of our MLP, which we assign to the variable 'b'.
- Finally, we create a Model object by specifying the input and output layers. We pass the input object to the inputs parameter, and the output tensor to the outputs parameter.
- We can compile and train the model in the same way as we did with the other APIs:

```
model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

3. INSTALLING TENSORFLOW 2:

- Step-by-step guide to install TensorFlow 2:

1. First, you need to install Python on your computer. You can download the latest version of Python from the official website: https://www.python.org/downloads/

2. Once you have installed Python, you can use pip (the package installer for Python) to install TensorFlow. Open a terminal or command prompt and type the following command:

```
pip install tensorflow
```

This will install the latest version of TensorFlow available on the PyPI package index.

3. Alternatively, you can install a specific version of TensorFlow by specifying the version number in the pip command. For example, to install TensorFlow version 2.7.0, you would use the following command:

```
pip install tensorflow==2.7.0
```

4. LOADING AND PREPROCESSING DATA WITH TENSORFLOW:

- Loading and preprocessing data is a crucial step in machine learning and deep learning pipelines. TensorFlow provides several tools and APIs for loading and preprocessing data efficiently. Here are some common methods:

1. Loading Data: TensorFlow provides a tf.data API that makes it easy to load and process data. This API provides several classes for reading data from different sources, such as files, numpy arrays, or tensors. For example, to read data from a CSV file, you can use tf.data.experimental.CsvDataset.

2. Data Augmentation: Data augmentation is a technique used to increase the diversity of the training set by applying various transformations to the data. TensorFlow provides several APIs for data augmentation, such as tf.image, which provides methods for image processing, such as flipping, rotating, and resizing.

3. Normalization: Normalizing the data is an essential preprocessing step that helps to ensure that the input features are in the same scale. You can use the tf.keras.layers.Normalization API to normalize the data.

4. Batching: Batching is a technique used to split the data into small batches and process them in parallel. TensorFlow provides several APIs for batching, such as tf.data.Dataset.batch.

5. Shuffling: Shuffling the data helps to prevent the model from overfitting to the order of the examples in the training set. You can use the tf.data.Dataset.shuffle API to shuffle the data.

6. Caching: Caching the data in memory can help to speed up the training process. You can use the tf.data.Dataset.cache API to cache the data.

- Here is an example of how to load and preprocess data using TensorFlow:

```
import tensorflow as tf
# Load the data
dataset = tf.data.experimental.CsvDataset('data.csv', [tf.float32, tf.float32, tf.int32], header=True)
# Data augmentation
dataset = dataset.map(lambda x, y, z: (tf.image.random_flip_left_right(x), y, z))
# Normalization
normalization_layer = tf.keras.layers.Normalization()
# Batching
dataset = dataset.batch(32)
# Shuffling
dataset = dataset.shuffle(buffer_size=1000)
# Caching
dataset = dataset.cache()
# Train the model
model.fit(dataset, epochs=10)
```

1. The Data API:

- The Data API in TensorFlow is a set of tools and APIs for building efficient and scalable input pipelines for machine learning models. It provides a simple and flexible way to read and preprocess data from various sources, such as CSV files, image files, and databases, and feed it to a machine learning model for training or inference.
- Here are some of the key features of the Data API:

  1. Efficient data loading: The Data API provides a number of tools for efficient data loading, including support for reading data from various file formats, shuffling, batching, and prefetching. These tools allow you to minimize the time spent on input processing and maximize the time spent on training your model.

  2. Flexible data preprocessing: The Data API supports a wide range of data preprocessing operations, such as data augmentation, normalization, and feature engineering. These operations can be applied in a highly parallel and efficient manner, allowing you to easily preprocess your data on the fly as it is being read from disk.

  3. Customizable pipeline construction: The Data API provides a flexible and extensible pipeline construction framework, allowing you to easily build custom input pipelines that suit your specific needs. You can easily add new preprocessing operations, data sources, or data formats to your pipeline, and the API will take care of the rest.

  4. Multi-GPU and distributed training: The Data API provides built-in support for multi-GPU and distributed training, allowing you to scale your input pipeline to multiple GPUs or even multiple machines. This can significantly reduce the time required to train large models on large datasets.

2. TFRecord Format:

- TFRecord is a binary file format used in TensorFlow for efficient storage and loading of large datasets.
- It is a flexible and extensible format that is designed to be fast, scalable, and platform-independent.
- The TFRecord format consists of a sequence of binary records, each containing a serialized protocol buffer message.
- The TFRecord format is commonly used for large-scale machine learning tasks, where datasets can be very large and difficult to store and load efficiently.
- The format is especially useful when dealing with datasets that contain a large number of

small files, such as image datasets.

- The main steps for creating and reading TFRecord files in TensorFlow:

➢ **Creating a TFRecord file:** To create a TFRecord file, you need to define a schema for your data and then serialize the data into binary records. Here is an example of how to create a TFRecord file:

```
import tensorflow as tf
# Define the schema for the data
feature_description = { 'feature1': tf.io.FixedLenFeature([], tf.float32),
                        'feature2': tf.io.FixedLenFeature([], tf.string),
                      }
# Serialize the data into binary records
with tf.io.TFRecordWriter('data.tfrecord') as writer:
    for data in dataset:
        example = tf.train.Example(features=tf.train.Features(feature=
                  { 'feature1': tf.train.Feature(float_list=tf.train.FloatList(value=[data['feature1']])),
                    'feature2': tf.train.Feature(bytes_list=
                                tf.train.BytesList(value=[data['feature2'].encode('utf-8')])), }))
        writer.write(example.SerializeToString())
```

➢ **Reading a TFRecord file:** To read a TFRecord file, you need to define a schema for the data and then parse the binary records into tensors. Here is an example of how to read a TFRecord file

```
import tensorflow as tf
# Define the schema for the data
feature_description = { 'feature1': tf.io.FixedLenFeature([], tf.float32),
                        'feature2': tf.io.FixedLenFeature([], tf.string), }

# Parse the binary records into tensors
def parse_example(example_proto):
    return tf.io.parse_single_example(example_proto, feature_description)
dataset = tf.data.TFRecordDataset('data.tfrecord')
dataset = dataset.map(parse_example)
```

3. The Features of API:

APIs in TensorFlow, an open-source machine learning framework developed by Google, have some unique features that make it easier for developers to build and deploy machine learning models. Some of the key features of TensorFlow's API include:

1. High-level abstractions: TensorFlow's API provides high-level abstractions for building machine learning models, allowing developers to focus on the overall structure of the model rather than the low-level details of the implementation.

2. Compatibility with multiple programming languages: TensorFlow's API supports multiple programming languages, including Python, C++, and Java, making it easier for developers to integrate machine learning models into their existing software systems.

3. Flexibility: TensorFlow's API is designed to be flexible and modular, allowing developers to build custom models and integrate them with other tools and frameworks.

4. Distributed computing: TensorFlow's API supports distributed computing, allowing developers to train and deploy machine learning models across multiple machines and devices.

5. Pre-built models: TensorFlow's API includes a library of pre-built models for common use cases, such as image classification and natural language processing, making it easier for developers to get started with machine learning.

6. Visualization tools: TensorFlow's API includes tools for visualizing and monitoring the performance of machine learning models, making it easier for developers to identify and fix issues.