Department of Information Engineering Technology

The Superior University, Lahore

Artificial Intelligence Lab

Experiment No.8

Practice the Basics of Deep Learning Using Keras and Tensorflow.

Prepared for

____

By:

Name: ____________

ID: _______________

Section: ___________

Semester: __________

Total Marks: _________________

Obtained Marks: ______________

Signature: __________________

Date: _____________

Experiment No. 8

Practice the Basics of Deep Learning Using Keras and Tensorflow. (Rubrics)

Name: ____ **Roll No. :** _____________________

A. PSYCHOMOTOR

Sr. No.	Criteria	Allocated Marks	Unacceptable 0%	Poor 25%	Fair 50%	Good 75%	Excellent 100%	Total Obtained
1	Follow Procedures	1	0	0.25	0.5	0.75	1	
2	Software and Simulations	2	0	0.5	1	1.5	2	
3	Accuracy in Output Results	3	0	0.75	1.5	2.25	3	
Sub Total		**6**	**Sub Total marks Obtained in Psychomotor(P)**					

B. AFFECTIVE

Sr. No.	Criteria	Allocated Marks	Unacceptable 0%	Poor 25%	Fair 50%	Good 75%	Excellent 100%	Total Obtained
1	Respond to Questions	1	0	0.25	0.5	0.75	1	
2	Lab Report	1	0	0.25	0.5	0.75	1	
3	Assigned Task	2	0	0.5	1	1.5	2	
Sub Total		**4**	**Sub Total marks Obtained in Affective (A)**					

Instructor Name:___________________________ **Total Marks (P+A):** _______________

Instructor Signature:_________________________ **Obtained Marks (P+A):** ___________

Introduction to Deep Learning

In deep learning, the "deep" part comes from the use of multiple layers in a neural network. This depth enables the model to learn intricate patterns and representations from the data, making it particularly powerful for tasks like image and speech recognition, natural language processing, and more.

Neural networks are inspired by the way our brains process information. In a neural network, each node or neuron is connected to others, and these connections have weights that are adjusted during training. The network learns by adjusting these weights to minimize the difference between its predictions and the actual outcomes.

There are different types of layers in a neural network, such as the input layer, hidden layers, and output layer. Each layer plays a specific role in processing information. Activation functions, like the sigmoid or ReLU (Rectified Linear Unit), introduce non-linearities to the model, allowing it to capture complex relationships in the data.

Training a neural network involves presenting it with a set of labeled examples, letting it make predictions, and then adjusting the weights based on the error. This process is often done through an optimization algorithm like gradient descent.

The field of deep learning has seen remarkable advancements in recent years, leading to breakthroughs in various domains. From self-driving cars to medical diagnoses, the applications of deep learning are vast and continually expanding.

Why using deep learning?

- **Complex Pattern Recognition:** Deep learning excels at automatically learning intricate patterns and representations from data. This is particularly useful in tasks like image and speech recognition, where traditional methods may struggle to capture the complexity of the data.
- **Feature Learning:** Deep learning models can automatically learn relevant features from the raw input data, eliminating the need for manual feature engineering. This is a significant advantage, especially when dealing with large and high-dimensional datasets.
- **Scalability:** Deep learning models can scale with data. As you provide more data for training, deep learning models can often improve their performance, capturing more nuances and generalizing better to new, unseen examples.
- **Versatility:** Deep learning can be applied to a wide range of domains, from computer vision and natural language processing to speech recognition and game playing. This versatility makes it a go-to choose for tackling various complex tasks.
- **End-to-End Learning**: Deep learning enables end-to-end learning, where a model learns to perform a task without the need for explicit intermediate steps. This can simplify the overall system and improve performance in certain applications.
- **Adaptability:** Deep learning models can adapt to changing input data, making them suitable for dynamic and evolving environments. This adaptability is valuable in scenarios where the relationships within the data may change over time.
- **State-of-the-Art Performance:** In many tasks, deep learning models have achieved state-of-the-art performance, surpassing traditional machine learning approaches. This has contributed to the widespread adoption of deep learning in various industries.

Introduction to Perceptron

A perceptron is a single-layer neural network that takes multiple binary inputs, applies weights to them, sums up the weighted inputs, and then passes the result through an activation function to produce a binary output.

Let's say you have inputs $x_1, x_2, \ldots, x_n$, corresponding weights $w_1, w_2, \ldots, w_n$ and a bias term b. The weighted sum (z) is calculated as

$$\boldsymbol{z = w_1 * x_1 + w_2 * x_2 + \cdots + w_n * x_n + b}$$

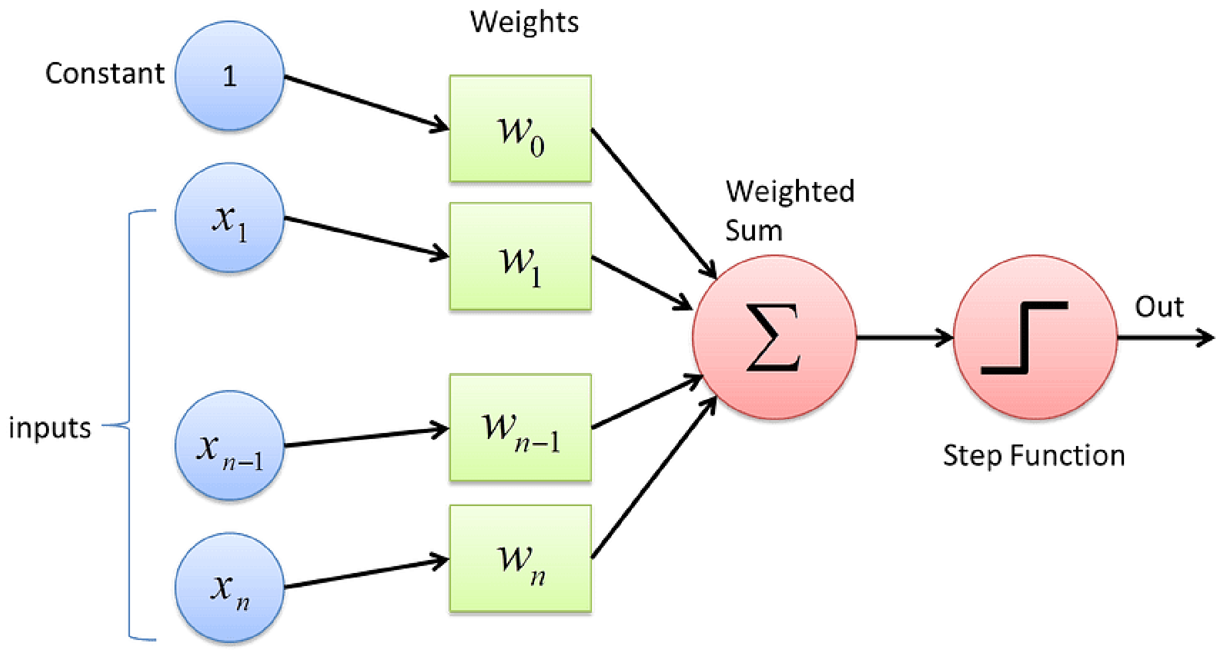
and the output (y) is the result of passing (z) through an activation function (*y=f(z))*

Activation Function: The activation function introduces non-linearity to the model. Common choices include the step function (resulting in a binary output), the sigmoid function, or the more commonly used Rectified Linear Unit (ReLU). The activation function determines whether the perceptron "fires" or not based on the weighted sum of inputs.

During the training process, the weights and bias are adjusted based on the error in the model's predictions. This adjustment is typically done using a learning algorithm, such as the perceptron learning algorithm or gradient descent.

Limitations: A single perceptron has limitations in handling complex patterns and non-linear relationships in data. However, by combining multiple perceptrons into layers and stacking these layers, we can create more sophisticated models capable of learning complex representations. This leads to the development of multi-layer perceptrons, which are the foundation of deep learning.

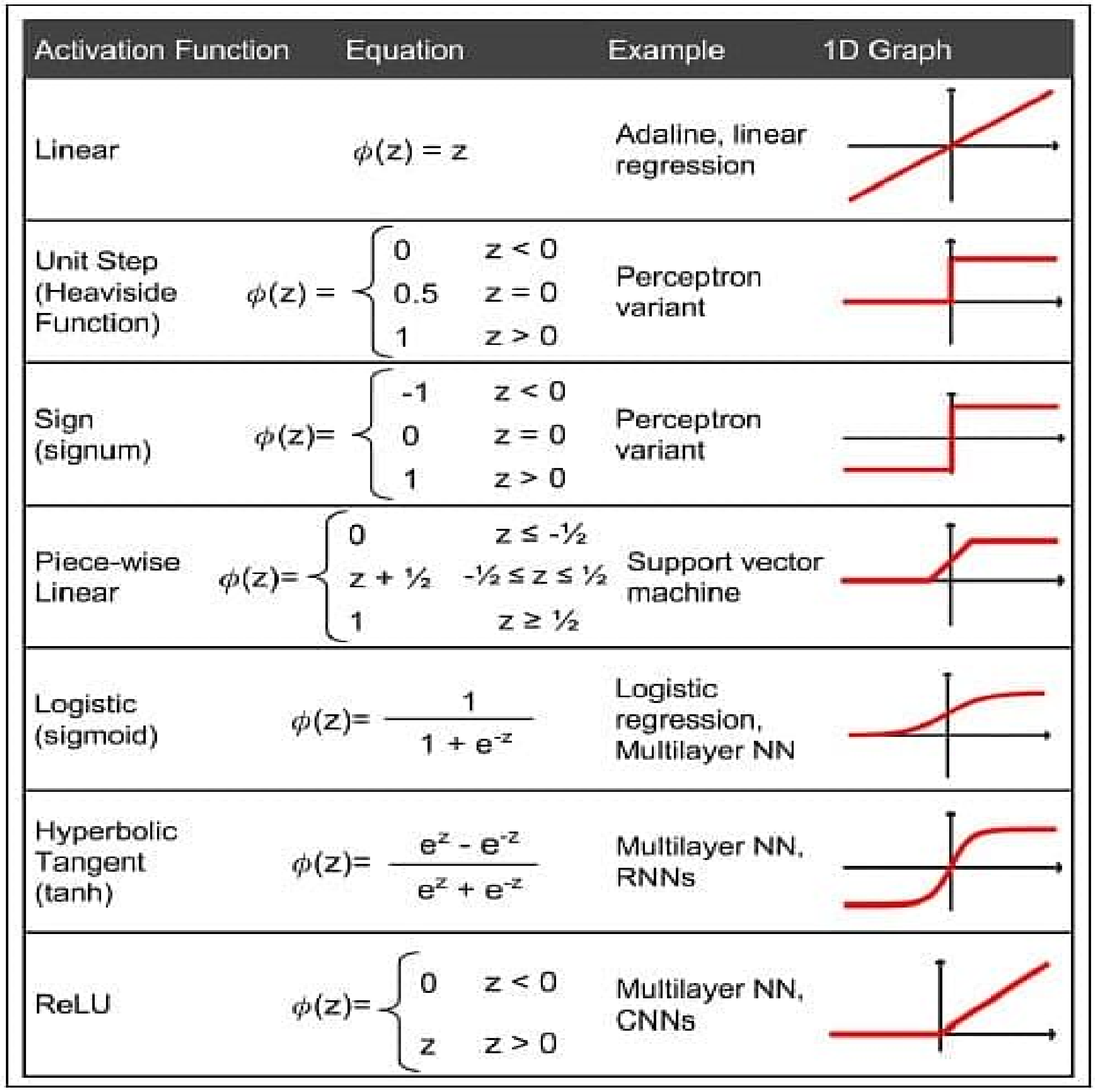
In deep learning, perceptrons are often used as the basic units in artificial neural networks, forming the building blocks of hidden layers. These networks can then learn and represent complex features in the input data through the training process.



Introduction to Activation Function

Activation functions play a crucial role in artificial neural networks by introducing non-linearity to the model. This non-linearity allows neural networks to learn complex patterns and relationships in data.

Here are a few commonly used activation functions:



Activation Function	Equation	Example	1D Graph
Linear	$\phi(z) = z$	Adaline, linear regression	
Unit Step (Heaviside Function)	$\phi(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Sign (signum)	$\phi(z)= \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Piece-wise Linear	$\phi(z)= \begin{cases} 0 & z \le -\frac{1}{2} \\ z + \frac{1}{2} & -\frac{1}{2} \le z \le \frac{1}{2} \\ 1 & z \ge \frac{1}{2} \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z)= \frac{1}{1 + e^{-z}}$	Logistic regression, Multilayer NN	
Hyperbolic Tangent (tanh)	$\phi(z)= \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multilayer NN, RNNs	
ReLU	$\phi(z)= \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases}$	Multilayer NN, CNNs	

Neural Network with Python

Here's how Neural Network works:

Import the necessary libraries. TensorFlow is used for building and training neural networks, and Keras is a high-level neural networks API running on top of TensorFlow. Matplotlib is used for plotting, and NumPy is for numerical operations.

```
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
import numpy as np
```

Loading the MNIST dataset, a collection of 28x28 pixel grayscale images of handwritten digits (0 through 9). It splits the dataset into training and testing sets.

```
(X_train, y_train) , (X_test, y_test) = keras.datasets.mnist.load_data()
```

Printing the number of samples in the training set.

```
len(X_train)
```

Displaying the first training image using Matplotlib.

```
plt.matshow(X_train[0])
```

Conversion from 2D to 1D

Reshaping the training images from 2D (28x28) to 1D arrays (28*28=784 elements).

```
X_train.reshape(len(X_train),28*28)
```

Normalizing the pixel values of the images by scaling them between 0 and 1. This is a common preprocessing step in machine learning.

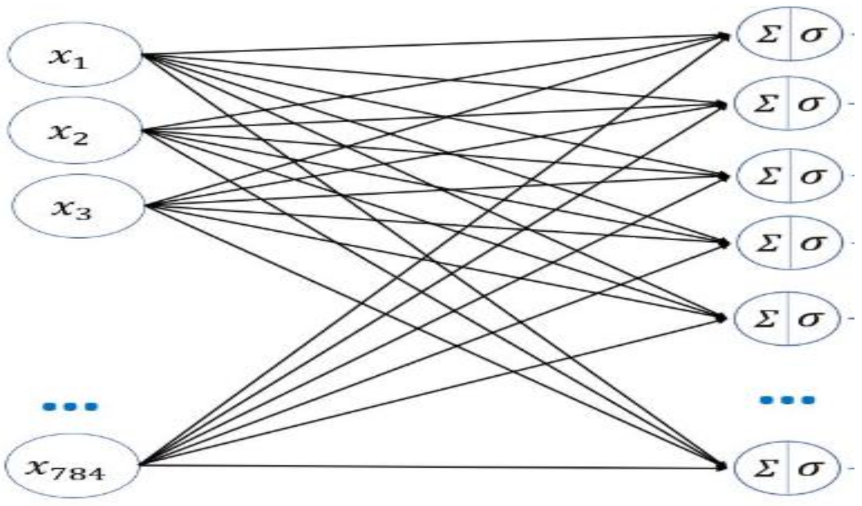
```
X_train = X_train / 255
X_test = X_test / 255
```

Reshaping the entire training and testing sets to 1D arrays after normalization.

```
X_train_flattened = X_train.reshape(len(X_train), 28*28)
X_test_flattened = X_test.reshape(len(X_test), 28*28)
```

➢ Neural Network with no hidden layers

A neural network with no hidden layers is essentially a single-layer neural network, often referred to as a perceptron or a single-layer perceptron.



Let's break down the key components and concepts:

- **Input Layer:** The only layer in the network is the input layer. Each node in this layer represents a feature or input variable. In the case of image data, like the MNIST dataset we were working with, each input node might correspond to a pixel in the image.
- **Weights and Bias:** Each connection between an input node and the output node (the single node in the output layer) has a weight. These weights determine the strength of the connection and are adjusted during the training process to optimize the model's performance. There is also a bias term that is added to the weighted sum before passing it through the activation function.
- **Activation Function:** The output of the network is obtained by applying an activation function to the weighted sum of inputs. In the code, the activation function used is the sigmoid function (activation='sigmoid'). The sigmoid function squashes the output between 0 and 1, making it suitable for binary classification problems.
- **Output Layer:** The output layer consists of a single node in this case. The output is the result of applying the activation function to the weighted sum of inputs.
- **Loss Function and Optimization:** During training, the model tries to minimize a loss function, which measures the difference between the predicted output and the actual target. The optimizer (in this case, 'adam') adjusts the weights and bias to reduce this loss, making the model more accurate.

Example:

Creating a simple neural network model using Keras. It has one layer with 10 neurons, each using the sigmoid activation function. The input shape is set to (784,), which corresponds to the flattened shape of the input images.

```
model = keras.Sequential([
    keras.layers.Dense(10, input_shape=(784,), activation='sigmoid')
])
```

Compiling the model, specifying the optimizer ('adam'), the loss function ('sparse_categorical_crossentropy' for multi-class classification), and the metric to monitor during training ('accuracy').

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

Training the model using the training data (X_train_flattened and y_train) for 5 epochs. The training involves adjusting the model's weights based on the specified optimizer and loss function.

```
model.fit(X_train_flattened, y_train, epochs=5)
```

Evaluating the model's performance on the testing data and prints the loss and accuracy.

```
model.evaluate(X_test_flattened, y_test)
```

Predicting the class probabilities for the first testing image.

```
y_predicted = model.predict(X_test_flattened)
y_predicted[0]
```

Displaying the first testing image using Matplotlib.

```
plt.matshow(X_test[0])
```

Finding the index of the maximum probability in the predicted probabilities, effectively giving the predicted label for the first testing image.

```
np.argmax(y_predicted[0])
```

Generating predicted labels for the entire testing set by finding the index of the maximum probability for each image.

```
y_predicted_labels = [np.argmax(i) for i in y_predicted]
y_predicted_labels[:5]
```

Creating a confusion matrix to evaluate the performance of the model on the testing set.

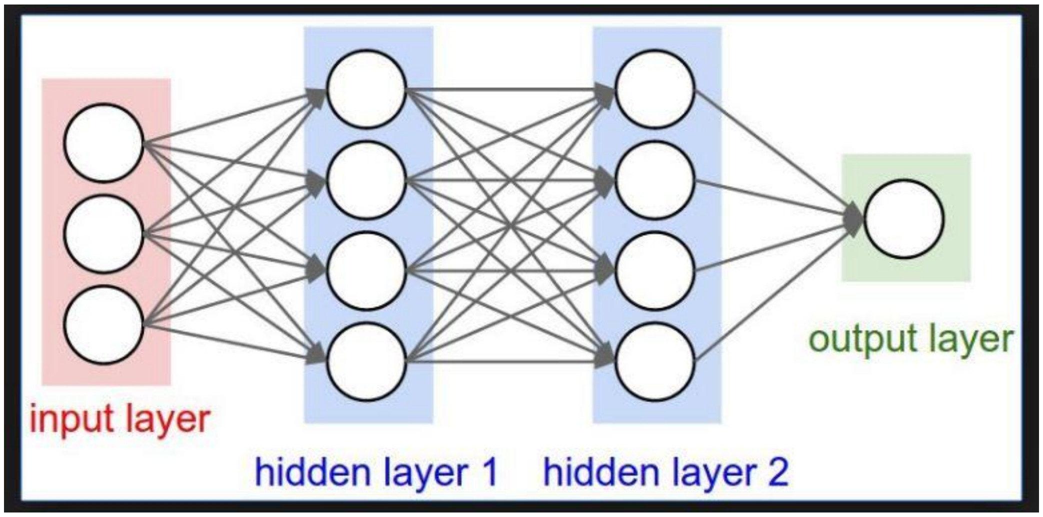
```
cm =
tf.math.confusion_matrix(labels=y_test,predictions=y_predicted_labels)
cm
```

Visualizing the confusion matrix using Seaborn, providing insights into the model's performance for each class. The x-axis represents predicted labels, and the y-axis represents true labels.

```
import seaborn as sn
plt.figure(figsize = (10,7))
sn.heatmap(cm, annot=True, fmt='d')
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

➢ Neural Network with hidden layers

A hidden layer in a neural network is a layer between the input layer and the output layer where neurons or nodes perform computations. It's called "hidden" because the data that flows through it during training is not directly observable from the outside. In other words, the input to a hidden layer is a function of the input data and the weights, but it is not part of the input or output of the overall system.



Here's a breakdown of key points about hidden layers:

- **Neurons/Nodes:** Each node in a hidden layer performs a weighted sum of its inputs, applies an activation function, and produces an output that serves as the input to the next layer.

- **Activation Function:** An activation function introduces non-linearity to the model, allowing the neural network to learn complex patterns. Common activation functions for hidden layers include ReLU (Rectified Linear Unit), sigmoid, and hyperbolic tangent (tanh).
- **Number of Hidden Layers and Neurons:** The number of hidden layers and the number of neurons in each layer are hyperparameters that you can experiment with to optimize the model's performance. Deep neural networks have multiple hidden layers, and the choice of architecture depends on the complexity of the task.
- **Learning Representations:** The hidden layers are responsible for learning hierarchical representations of the input data. As you move deeper into the network, the features become more abstract and complex.

Example:

Defining a neural network model using Keras Sequential API. It has two layers:

- **Input Layer (Dense): T**he first Dense layer has 100 neurons with the **ReLU activation function**. The input_shape is set to (784,), indicating that each input sample has 784 features (flattened size of an image in this case).
- **Output Layer (Dense):** The second Dense layer has 10 neurons, each representing one digit (0 through 9). The activation function used here is the **sigmoid function**.

```
model = keras.Sequential([
    keras.layers.Dense(100, input_shape=(784,), activation='relu'),
    keras.layers.Dense(10, activation='sigmoid')
])
```

Compiling the model, specifying 'adam' as the optimizer, 'sparse_categorical_crossentropy' as the loss function (suitable for multi-class classification), and 'accuracy' as the metric to monitor during training.

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

Training the model using the training data. The training will run for 5 epochs.

```
model.fit(X_train_flattened, y_train, epochs=5)
```

Evaluating the model's performance on the testing data and prints the loss and accuracy.

```
model.evaluate(X_test_flattened,y_test)
```

Generating predicted labels for the testing set and creates a confusion matrix to evaluate the model's performance.

```
y_predicted = model.predict(X_test_flattened)
```

```
y_predicted_labels = [np.argmax(i) for i in y_predicted]
cm =
tf.math.confusion_matrix(labels=y_test,predictions=y_predicted_labels)
```

Visualizing the confusion matrix using Seaborn, providing insights into the model's performance for each class. The x-axis represents predicted labels, and the y-axis represents true labels.

```
plt.figure(figsize = (10,7))
sn.heatmap(cm, annot=True, fmt='d')
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

➢ Neural Network with Flatten layers

The Flatten layer is a type of layer in neural networks, often used as the input layer in image processing tasks. Its purpose is to convert multi-dimensional input data into a one-dimensional array. This is particularly useful when transitioning from convolutional layers, which work with 3D data (height, width, channels), to fully connected layers, which require 1D input.

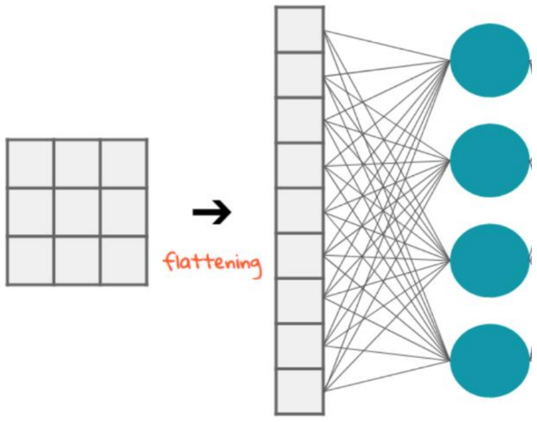
In the context of image data:

Before Flatten Layer (Convolutional Layers):

- Convolutional layers process input images with multiple channels (e.g., RGB channels).
- The data is in the form of a 3D tensor: (height, width, channels).

After Flatten Layer (Fully Connected Layers):

- Fully connected layers expect 1D input.
- The Flatten layer is used to reshape the output of the convolutional layers into a 1D array.



Example:

Defining a neural network model using Keras Sequential API. It has three layers:

- **Input Layer (Flatten):** The Flatten layer transforms the input data from a 2D array (28x28 pixels) to a 1D array (784 pixels). It's used to convert the image matrix into a vector so that it can be fed into the subsequent dense layers.
- **Hidden Layer (Dense):** The first Dense layer has 100 neurons with **the ReLU activation function.** This layer is responsible for learning and capturing patterns in the flattened input data.
- **Output Layer (Dense):** The second Dense layer has 10 neurons, each representing one digit (0 through 9). The activation function used here is the **sigmoid function**.

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(10, activation='sigmoid')
])
```

Compiling the model, specifying 'adam' as the optimizer, 'sparse_categorical_crossentropy' as the loss function (suitable for multi-class classification), and 'accuracy' as the metric to monitor during training.

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

Training the model using the training data (X_train and y_train) for 10 epochs. During training, the model adjusts its weights and biases to minimize the specified loss function.

```
model.fit(X_train, y_train, epochs=10)
```

Evaluating the model's performance on the testing dataset and prints the loss and accuracy. It helps us understand how well our model generalizes to new, unseen data.

```
model.evaluate(X_test,y_test)
```

Lab Task:

Build a neural network to recognize handwritten digits (0 to 9) from the MNIST dataset using a deep learning framework like TensorFlow

Load the MNIST dataset.

Build a simple neural network with:

- Experiment with different architectures (add more layers, neurons) so that your accuracy reach 99%.
- Try different activation functions for the hidden layer.
- Compile the model with appropriate loss function, optimizer, and metrics.
- Train the model on the training set for a reasonable number of epochs.
- Evaluate the model on the testing set and analyze its accuracy.
- Visualize the model's predictions on random test samples.

Home Task:

Build a neural network to classify images as either cats or dogs.

Download the Cats vs. Dogs dataset, which typically contains labeled images of cats and dogs.

Build a NN with:

- Neural Network layers with activation functions like ReLU.
- Flatten layer to transition from convolutional layers to fully connected layers.
- Dense layers with appropriate activation functions.
- Compile the model with binary crossentropy loss (since it's a binary classification task) and an optimizer of your choice.
- Train the model on the training set for a reasonable number of epochs.
- Evaluate the model on the testing set and analyze its performance.