ASSIGNMENT No: 04

Title:-

Design and implement CNN for image classification.

a) Select a suitable image classification dataset (medical engineering, agricultural, etc.).

b) Optimized with different hyper-parameters including learning rate, filter size, no. of layers, optimizers, dropouts, etc.

Mapping with Syllabus –

Unit 3

Objective –

Design and implement a Convolutional Neural Network (CNN) for image classification on a selected image classification dataset, such as medical engineering, agricultural to optimize with different hyper-parameters including learning rate, filter size, no. of layers, optimizers, dropouts, etc.

Outcome –

Implement the technique of Convolution neural network (CNN)

Software Requirements –

- Python (3.x recommended)
- Jupyter Notebook or any Python IDE or Google Colab

Hardware Requirements –

A machine with sufficient RAM and processing power for model training (8GB RAM recommended)

Prerequisites –

- Basic understanding of Python programming
- Familiarity with the concepts of Neural Networks

Dataset –

Inbuilt dataset - MNIST
https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz

Libraries or Modules Used –

- Keras (for building and training neural network models)
- NumPy (for numerical operations)
- Matplotlib (for plotting images)
- TensorFlow (deep learning framework)
- Adam (optimizer for training)

Theory –

Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNN) are a powerful type of deep learning model specifically designed for processing and analyzing visual data, such as images and videos. They have revolutionized the field of Computer Vision, enabling remarkable advancements in tasks like Image Recognition, Object Detection, and Image Segmentation.

To grasp the essence of Convolutional Neural Networks (CNNs), it is essential to have a solid understanding of the basics of Deep Learning and acquaint yourself with the terminology and principles of neural networks. If you're new to this, don't fret! I have previously covered these fundamentals in my blog posts, serving as primers to help you lay a strong foundation.

Basic Architecture

The architecture of Convolutional Neural Networks is meticulously designed to extract meaningful features from complex visual data. This is achieved through the use of specialized layers within the network architecture. It comprises of three fundamental layer types:

1. Convolutional Layers
2. Pooling Layers
3. Fully-Connected Layers

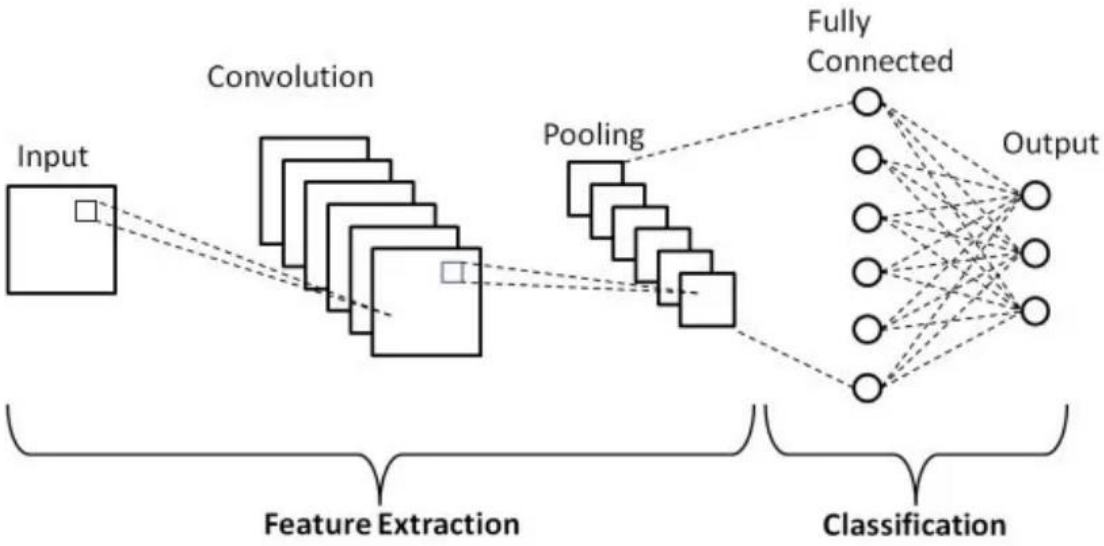


Image Source: Research Gate

Now, let's delve into each of these layers in detail to gain a deeper understanding of their role and significance in Convolutional Neural Networks (CNN).

The Convolution Layer:

The convolutional layer serves as the fundamental building block within a Convolutional Neural Network (CNN), playing a central role in performing the majority of computations. It relies on several key components, including input data, filters, and feature maps.

Convolution preserves the spatial relationship between pixels by learning image features using small squares of input data. It is a tensor operation (dot product) where two tensors serve as input, and a resulting tensor is generated as the output. This layer employs a tile-like filtering approach on an input tensor using a small window known as a kernel. The kernel specifies the specific characteristics that the convolution operation seeks to filter, generating a significant response when it detects the desired features. To explore further details about various kernels and their functionalities, refer here.

The convolutional layer computes a dot product between the filter value and the image pixel values, and the matrix formed by sliding the filter over the image is called the Convolved Feature, Activation Map, or Feature Map.

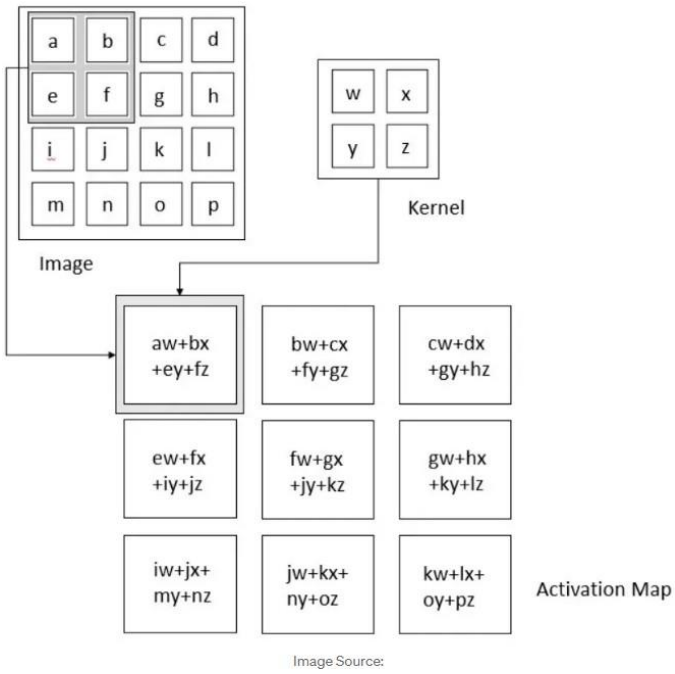


Image Source:

Each element from one tensor (image pixel) is multiplied by the corresponding element (the element in the same position) of the second tensor(kernel value), and then all the values are summed to get the result.

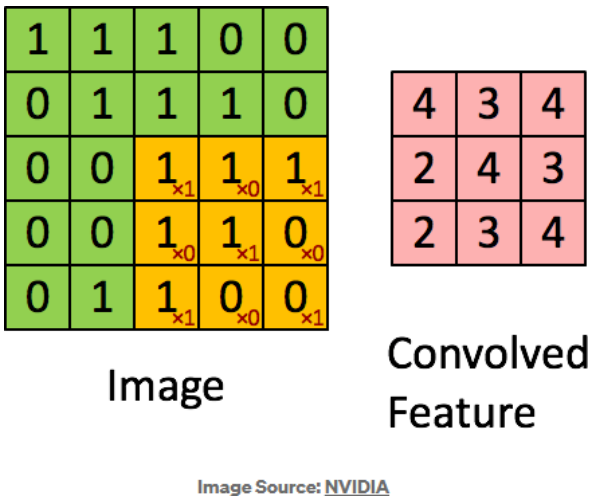


Image Source: NVIDIA

The output size of the convoluted layer is determined by several factors, including the input size, kernel size, stride, and padding. The formula to calculate the output size is as follows:

$$H_{out} = 1 + \frac{H_{in} + (2.pad) - K_{height}}{S}, \quad W_{out} = 1 + \frac{W_{in} + (2.pad) - K_{width}}{S}$$

Image By Author: Output size of the convolution image

Let's take an example to better understand this concept. Imagine we have an input image with dimensions of 6x6 pixels. For the convolutional operation, we use a kernel with dimensions of 3x3 pixels, a stride of 1, and no padding (padding of 0).

To calculate the output size of the convoluted image, we can apply the following formula: output_size = 1 + (input_size * kernel_size + (2 * padding)) / stride.

Plugging in the values, we get: output_size = 1 + (6*3 + (2 * 0)) / 1 = 1 + (3 / 1) = 1 + 3 = 4.

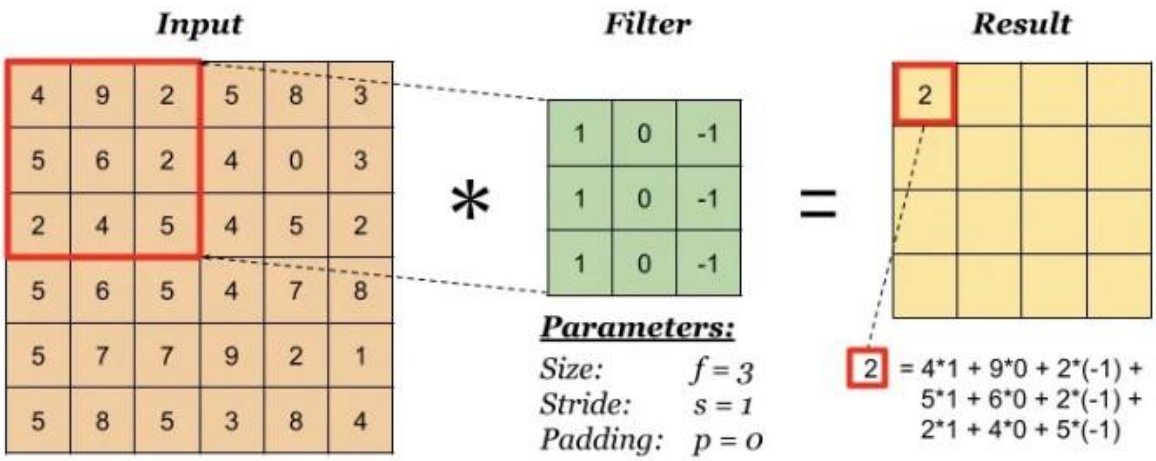


Image By Author: Convolution on 2D Image / Single Channel

Hence, the resulting convoluted image will have dimensions of 4x4 pixels.

When the input has more than one channel (e.g. an RGB image), the filter should have a matching number of channels. To calculate one output cell, perform convolution on each matching channel, then add the result together.

After each convolution operation, a CNN applies a Rectified Linear Unit (ReLU) transformation to the feature map, introducing nonlinearity to the model.

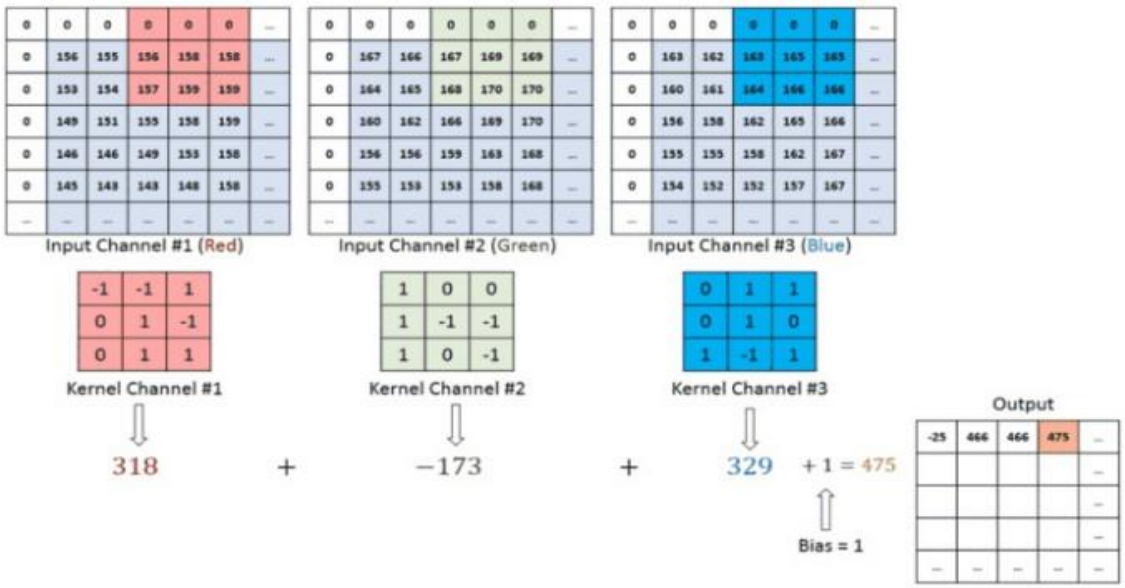


Image By Author: Convolution on RGB Image

The Pooling Layer:

Pooling layers, also referred to as downsampling, serve to reduce the dimensionality of the input, thereby decreasing the number of parameters. Similar to convolutional layers, pooling operations involve traversing a filter across the input. However, unlike convolutional layers, the pooling filter does not possess weights. Instead, the filter applies an aggregation function to
the values within its receptive field, generating the output array. Two primary types of pooling are commonly employed:

Max Pooling: It selects the pixel with the maximum value to send to the output array.
Average pooling: It calculates the average value within the receptive field to send to the output array.

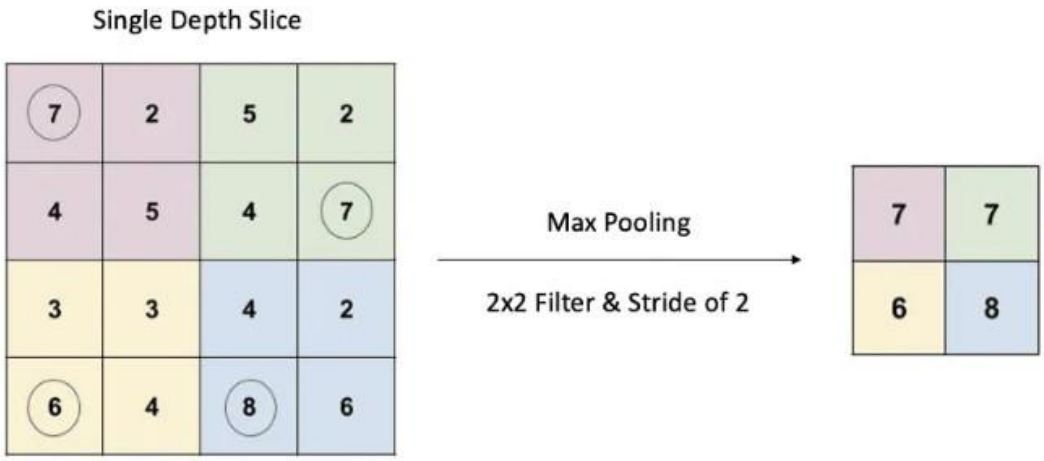


Image By Author: Max Pooling

Pooling offers a significant advantage in that it does not require learning any parameters. However, this attribute also presents a potential drawback as pooling may discard crucial information. While pooling serves to reduce dimensionality and extract key features, there is a possibility that important details can be lost during this process.

Fully-Connected Layer:

The Fully Connected Layer i.e dense layer aims to provide global connectivity between all neurons in the layer. Unlike convolutional and pooling layers, which operate on local spatial regions, the fully connected layer connects every neuron to every neuron in the previous and subsequent layers.

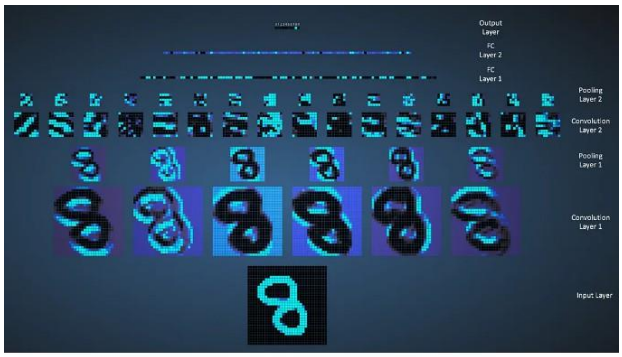


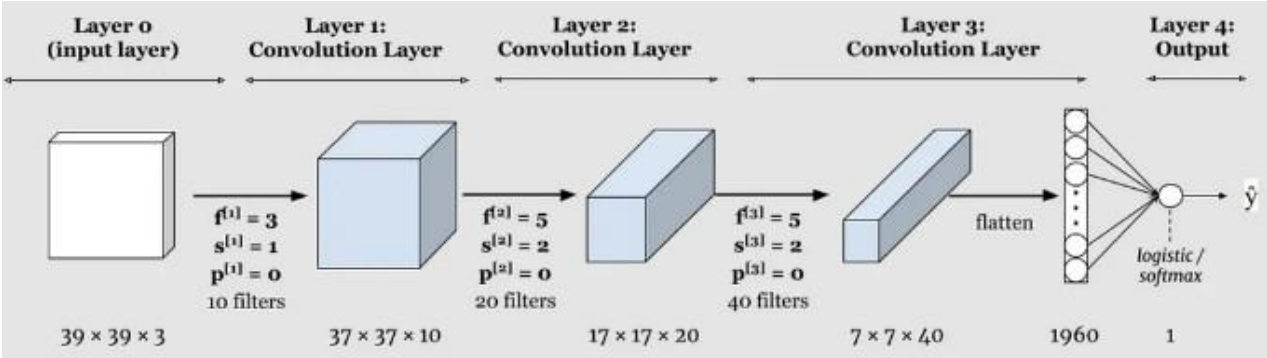
Image Source: Here

The fully connected layer typically appears at the end of the ConvNet architecture, taking the flattened feature maps from the preceding convolutional and pooling layers as input. Its purpose is to combine and transform these high-level features into the final output, such as

class probabilities or regression values, depending on the specific task. While convolutional and pooling layers tend to use ReLu functions, FC layers usually leverage a softmax activation function to classify inputs appropriately, producing a probability from 0 to 1.

This layer converts a three-dimensional layer in the network into a one-dimensional vector to fit the input of a fully-connected layer for classification. For example, a 5x5x2 tensor would be converted into a vector of size 50. This part is in principle the same as a regular Neural Network.

Now that we have explored the concepts of convolution, pooling, and fully connected layers individually, let's combine them to understand the basic architecture of a Convolutional Neural Network (CNN). In a typical CNN, the input data passes through a series of convolutional layers, which extract features using filters. The output of each convolutional layer is then downsampled using pooling layers to reduce dimensionality and capture the most salient information. Finally, the resulting feature maps are flattened and fed into one or more fully connected layers, which perform the classification or regression tasks.



A Sample Convnet: Resource

This combination of convolution, pooling, and fully connected layers forms the core structure of a CNN and enables it to learn and recognize complex patterns in images or other data.

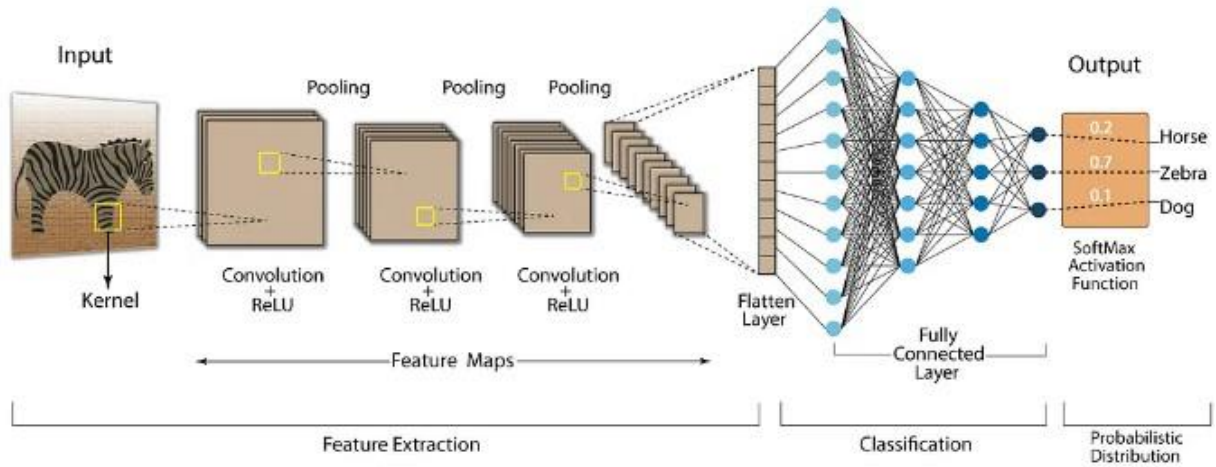


Image Source: Analytics Vidhya

Algorithm -

1. **Load and Prepare Dataset:**

Load the chosen image classification dataset (e.g., medical, agricultural) either from a local directory or using TensorFlow Datasets.
Split the dataset into training and validation sets.
Preprocess the images (resize, normalization) and preprocess labels (if necessary).

2. **Define CNN Architecture:**

Design the CNN architecture using TensorFlow's Keras API.
Construct the model with convolutional layers, pooling layers, fully connected layers, and appropriate activation functions.
Specify input shape, number of classes, and layer configurations.

3. **Compile the Model:**

Compile the model by specifying the optimizer, loss function, and evaluation metrics. Choose an appropriate optimizer (e.g., Adam) and a suitable loss function (e.g., sparse categorical crossentropy for multi-class classification).

4. **Hyperparameter Tuning and Training:**

Set hyperparameters like learning rate, batch size, and number of epochs. Train the CNN model on the training set using the fit function.
Utilize techniques like early stopping to prevent overfitting and monitor validation loss/accuracy.

5. **Evaluate and Test:**

Evaluate the trained model's performance on the validation set to assess its accuracy and generalization.

Once satisfied, use the model to predict and evaluate its accuracy on the test set to assess its real-world performance.

Application –

1. Medical Imaging:

Disease Detection: Identifying tumors, lesions, or anomalies in MRI, X-ray, or CT scans for conditions like cancer, fractures, or internal organ abnormalities.
Diagnosis Assistance: Analyzing retinal scans for diabetic retinopathy or identifying skin conditions through dermatology images.

2. Agriculture:

Crop Disease Identification: Classifying plant images to detect diseases, nutrient deficiencies, or pest infestations in crops using systems like Plant Village.
Weed Detection: Identifying and distinguishing weeds from crops to enable targeted herbicide application.

3. Autonomous Vehicles:

Object Recognition: Recognizing pedestrians, traffic signs, and other vehicles for safe navigation and decision-making in self-driving cars.
Lane Detection: Identifying lane markings for autonomous vehicle guidance.

4. Retail and E-commerce:

Product Classification: Categorizing products for inventory management and cataloging in e- commerce platforms.
Visual Search: Enabling visual search capabilities to find similar products using images captured by users.

5. Security and Surveillance:

Facial Recognition: Verifying identities or identifying individuals in security systems or surveillance footage.
Anomaly Detection: Spotting suspicious activities or objects in surveillance images or videos.

Inference –

The process involves training the model, preprocessing image, prediction, evaluation and iteration.
The use of different architectures such as Simple Neural Network, CNN allows for comparison and analysis of different images for image classification.
Optimization with different hyper-parameters including learning rate, filter size, no. of layers, optimizers, dropouts, etc. are understood in this practical.

Code -

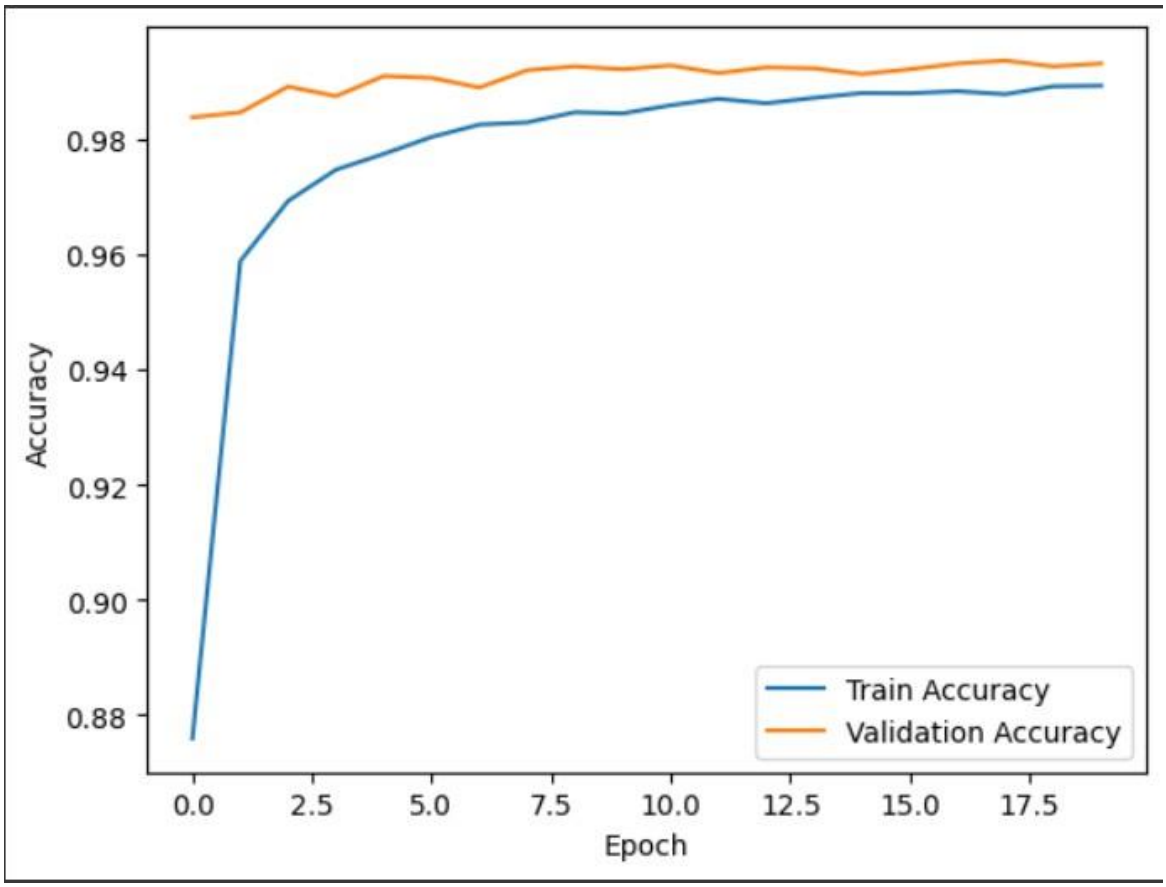
```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models from
tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator from
sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Load and preprocess the MNIST dataset
(train_images, train_labels), (test_images, test_labels) = datasets.mnist.load_data() train_images, test_images
= train_images / 255.0, test_images / 255.0

# Add channel dimension to the images
train_images = train_images.reshape((60000, 28, 28, 1))
test_images = test_images.reshape((10000, 28, 28, 1))

# Split the dataset into training and validation sets
train_images, val_images, train_labels, val_labels = train_test_split( train_images,
train_labels, test_size=0.1, random_state=42
)

# Data augmentation for training images
datagen = ImageDataGenerator(rotation_range=10, zoom_range=0.1, width_shift_range=0.1,
height_shift_range=0.1)
datagen.fit(train_images)

# Create a CNN model with hyperparameter tuning and regularization model =
models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.Flatten()) model.add(layers.Dropout(0.5))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.001), loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

# Train the model with data augmentation
history = model.fit(datagen.flow(train_images, train_labels, batch_size=64),
```

```
epochs=20, validation_data=(val_images, val_labels)) # Evaluate the

model on the test set
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"Test Accuracy: {test_acc}")

# Plot training history
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend() plt.show()
```

Output –

```
11490434/11490434 [==============================] - 0s 0us/step
Epoch 1/20
844/844 [==============================] - 87s 100ms/step - loss: 0.3852 - accuracy:
0.8760 - val_loss: 0.0543 - val_accuracy: 0.9838 Epoch
2/20
844/844 [==============================] - 83s 98ms/step - loss: 0.1320 - accuracy:
0.9589 - val_loss: 0.0491 - val_accuracy: 0.9847 Epoch
3/20
844/844 [==============================] - 72s 85ms/step - loss: 0.0998 - accuracy:
0.9693 - val_loss: 0.0361 - val_accuracy: 0.9892 Epoch
4/20
844/844 [==============================] - 70s 83ms/step - loss: 0.0812 - accuracy:
0.9747 - val_loss: 0.0391 - val_accuracy: 0.9875 Epoch
5/20
844/844 [==============================] - 71s 84ms/step - loss: 0.0727 - accuracy:
0.9775 - val_loss: 0.0308 - val_accuracy: 0.9910 Epoch
6/20
844/844 [==============================] - 70s 83ms/step - loss: 0.0639 - accuracy:
0.9804 - val_loss: 0.0317 - val_accuracy: 0.9907 Epoch
7/20
844/844 [==============================] - 70s 83ms/step - loss: 0.0563 - accuracy:
0.9826 - val_loss: 0.0370 - val_accuracy: 0.9890 Epoch
8/20
844/844 [==============================] - 70s 83ms/step - loss: 0.0557 - accuracy:
0.9829 - val_loss: 0.0283 - val_accuracy: 0.9920 Epoch
9/20
844/844 [==============================] - 71s 84ms/step - loss: 0.0507 - accuracy:
0.9847 - val_loss: 0.0269 - val_accuracy: 0.9927 Epoch
10/20
```

844/844 [==========] - 74s 87ms/step - loss: 0.0497 - accuracy: 0.9845 - val_loss: 0.0300 - val_accuracy: 0.9922 Epoch 11/20
844/844 [==========] - 71s 84ms/step - loss: 0.0465 - accuracy: 0.9859 - val_loss: 0.0259 - val_accuracy: 0.9928 Epoch 12/20
844/844 [==========] - 72s 85ms/step - loss: 0.0432 - accuracy: 0.9870 - val_loss: 0.0336 - val_accuracy: 0.9915 Epoch 13/20
844/844 [==========] - 71s 84ms/step - loss: 0.0427 - accuracy: 0.9863 - val_loss: 0.0300 - val_accuracy: 0.9925 Epoch 14/20
844/844 [==========] - 70s 83ms/step - loss: 0.0429 - accuracy: 0.9872 - val_loss: 0.0254 - val_accuracy: 0.9923 Epoch 15/20
844/844 [==========] - 70s 83ms/step - loss: 0.0380 - accuracy: 0.9880 - val_loss: 0.0321 - val_accuracy: 0.9913 Epoch 16/20
844/844 [==========] - 68s 81ms/step - loss: 0.0385 - accuracy: 0.9880 - val_loss: 0.0283 - val_accuracy: 0.9922 Epoch 17/20
844/844 [==========] - 72s 85ms/step - loss: 0.0357 - accuracy: 0.9884 - val_loss: 0.0243 - val_accuracy: 0.9932 Epoch 18/20
844/844 [==========] - 70s 83ms/step - loss: 0.0381 - accuracy: 0.9878 - val_loss: 0.0264 - val_accuracy: 0.9937 Epoch 19/20
844/844 [==========] - 70s 82ms/step - loss: 0.0352 - accuracy: 0.9892 - val_loss: 0.0259 - val_accuracy: 0.9927 Epoch 20/20
844/844 [==========] - 71s 84ms/step - loss: 0.0351 - accuracy: 0.9893 - val_loss: 0.0233 - val_accuracy: 0.9932
313/313 [==========] - 4s 12ms/step - loss: 0.0162 - accuracy: 0.9951
Test Accuracy: 0.9951000213623047



References:

https://medium.com/codex/understanding-convolutional-neural-networks-a-beginners- journey-into-the-architecture-aab30dface10

https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz

Conclusion: Thus Designed and implemented a CNN for Image Classification.