Experiment - 3

Aim:Design a CNN architecture to implement the image classification task over an image dataset. Perform the Hyper-parameter tuning and record the results.

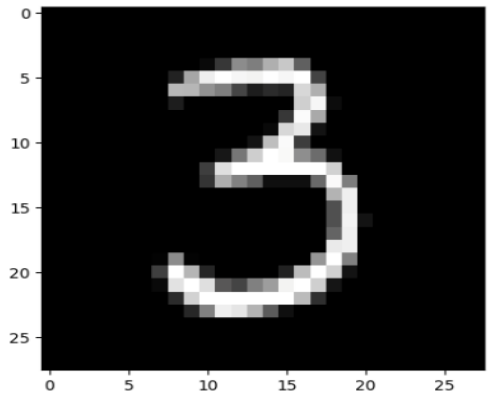
Database

- The data that will be incorporated is the **MNIST database** which contains 60,000 images for training and 10,000 test images.
- The dataset consists of small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9
- The MNIST dataset is conveniently bundled within Keras, and we can easily analyze some of its features in Python.

```
from tensorflow import keras
from keras.datasets import mnist      # MNIST dataset is included in Keras
(X_train, y_train), (X_test, y_test) = mnist.load_data()

print("X_train shape", X_train.shape)
print("y_train shape", y_train.shape)
print("X_test shape", X_test.shape)
print("y_test shape", y_test.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
X_train shape (60000, 28, 28)
y_train shape (60000,)
X_test shape (10000, 28, 28)
y_test shape (10000,)
```

```
# Visualize any random image
import matplotlib.pyplot as plt
i=50;
plt.imshow(X_train[i], cmap='gray');
```



```
Formatting the Input
# Single-channel input data (grey-scale)
# First apply convolutions then flatten

X_train = X_train.reshape(60000, 28, 28, 1) # single-channel input
X_test = X_test.reshape(10000, 28, 28, 1)

X_train = X_train.astype('float32')         # change integers to 32-bit
floating point numbers
X_test = X_test.astype('float32')

X_train /= 255                              # min-max normalization
X_test /= 255

print("Training matrix shape", X_train.shape)
print("Testing matrix shape", X_test.shape)
```

```
Training matrix shape (60000, 28, 28, 1)
Testing matrix shape (10000, 28, 28, 1)
```

Convolutional Neural Network

- CNNs are a class of deep neural networks designed for processing and analyzing visual data, especially images.
- They consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers.

1. **Convolution Operation:**
   - Convolution is a key operation in CNNs that involves applying filters (kernels) to input data, typically images.
   - Kernels traverse through the input image, extracting local features and producing feature maps.
   - Each kernel specializes in detecting specific patterns or features, such as edges, textures, or more complex structures.

2. **Feature Maps:**
   - Feature maps are the output of the convolutional operation. They represent the presence of learned features in the input data.
   - Multiple convolutional layers can be stacked to learn hierarchical representations of visual features.

3. **Learning Different Characteristics:**
   - Each kernel in a CNN is responsible for learning different characteristics or features of the input data.
   - The first layers may capture basic features like edges, while deeper layers can learn more abstract and complex features.

4. **Max Pooling:**
   - Max pooling is a down-sampling operation often used in CNNs to reduce the spatial dimensions of the feature maps.
   - It helps in reducing the number of learnable parameters, thus decreasing computational cost and memory requirements.
   - Max pooling retains the most significant information by selecting the maximum value from a group of values in the input.

5. **Benefits of CNNs:**
   - CNNs are well-suited for image-related tasks due to their ability to automatically learn spatial hierarchies of features.
   - They have shown exceptional performance in tasks such as image classification, object detection, and image segmentation.

6. **Transfer Learning:**
   - CNNs, particularly pre-trained models, can be used for transfer learning, where a model trained on a large dataset for one task is fine-tuned for a different but related task.
   - This helps leverage knowledge gained from large datasets and accelerates training on smaller datasets.

7. **Applications:**
   - CNNs find applications in various domains, including computer vision, medical imaging, autonomous vehicles, and more.

Building a Convolutional Neural Network

```
from keras import backend as K
from keras import __version__

print('Using Keras version:', __version__, 'backend:', K.backend())
```

```
Using Keras version: 2.15.0 backend: tensorflow
```

```
# import cnn layers
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense
import tensorflow as tf
```

2D convolution layer creates a convolution kernel that is convolved with the layer input over a single spatial (or temporal) dimension to produce a tensor of outputs. If use_bias is True, a bias vector is created and added to the outputs. Finally, if activation is not None, it is applied to the outputs as well.

`Conv2D` indicates a 2D convolutional layer. Convolutional layers are commonly used in image processing tasks in neural networks.

1. *Activation Function:* An activation function is a mathematical operation applied to each node (or neuron) in a neural network, introducing non-linearity to the network. This non-linearity is essential for the neural network to learn complex patterns and relationships in the data. *In other words, activation functions determine the output of a neuron given its input.*

   - ReLU (Rectified Linear Unit): The activation function introduces non-linearity to the model. ReLU is a commonly used activation function that outputs the input directly if it is positive, and zero otherwise. Mathematically, it is defined as f(x)=max(0,x). ReLU is often preferred because it helps the model learn faster and can mitigate the vanishing gradient problem.
   - Softmax: Softmax is an activation function used in the output layer of a neural network for multi-class classification problems. It converts a vector of raw scores (logits) into probabilities. The output values of the softmax function represent the probabilities of each class, and they sum up to 1.

2. *Padding:* Padding is the process of adding extra layers of pixels around the input data. It is typically done to ensure that the convolution operation can be applied to the edges of the input without losing information. There are two common types of padding:

   - Valid Padding (No Padding): No extra pixels are added. The convolution operation is only applied to positions where the entire filter fits within the input.
   - Same Padding: Padding is added to the input so that the output has the same spatial dimensions as the input. This is achieved by adding zeros around the input.

3. *Strides:* Stride refers to the step size the convolutional filter takes when sliding over the input data. A stride of 1 means the filter moves one pixel at a time, while a stride of 2 means the filter moves two pixels at a time. Larger strides reduce the spatial dimensions of the output.

4. *Pooling (pool_size):* Pooling is a downsampling operation commonly used after convolutional layers. It helps reduce the spatial dimensions of the input volume, reducing the number of parameters and computation in the network. The pool size determines the size of the pooling window. Common types of pooling include:

   - Max Pooling: Takes the maximum value from a group of values in the input.
   - Average Pooling: Takes the average value from a group of values in the input.

The input shape is specified directly within the convolutional layer using the `input_shape` parameter. This is a common practice, particularly for the first layer of a neural network. Instead of using `model.build()`, which is an alternative for setting up input shape, the input_shape is directly included in the layer definition. This parameter informs the model about the shape of the input data it will receive. This approach is concise and often used when the model architecture is straightforward.

```
model = Sequential()  # Linear stacking of layers

# Convolution Layer 1: 8 filters, kernel size 3x3, relu activation, valid
padding, stride 1
model.add(Conv2D(8, (3, 3), activation='relu', padding='valid', strides=(1,
1), input_shape=(28, 28, 1)))
```

```
# MaxPooling: pool size 2, stride 2
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

# Convolution Layer 2: 16 filters, kernel size 3x3, relu activation, valid
padding, stride 1
model.add(Conv2D(16, (3, 3), activation='relu', padding='valid', strides=(1,
1)))

# MaxPooling: pool size 2, stride 2
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

# Flatten final feature matrix into a 1d array
model.add(Flatten())

# Fully Connected Layer: 64 units and relu activation
model.add(Dense(64, activation='relu'))

# Dropout layer, 0.2 rate
model.add(Dropout(0.2))

# Final output dense Layer
model.add(Dense(10, activation='softmax'))

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

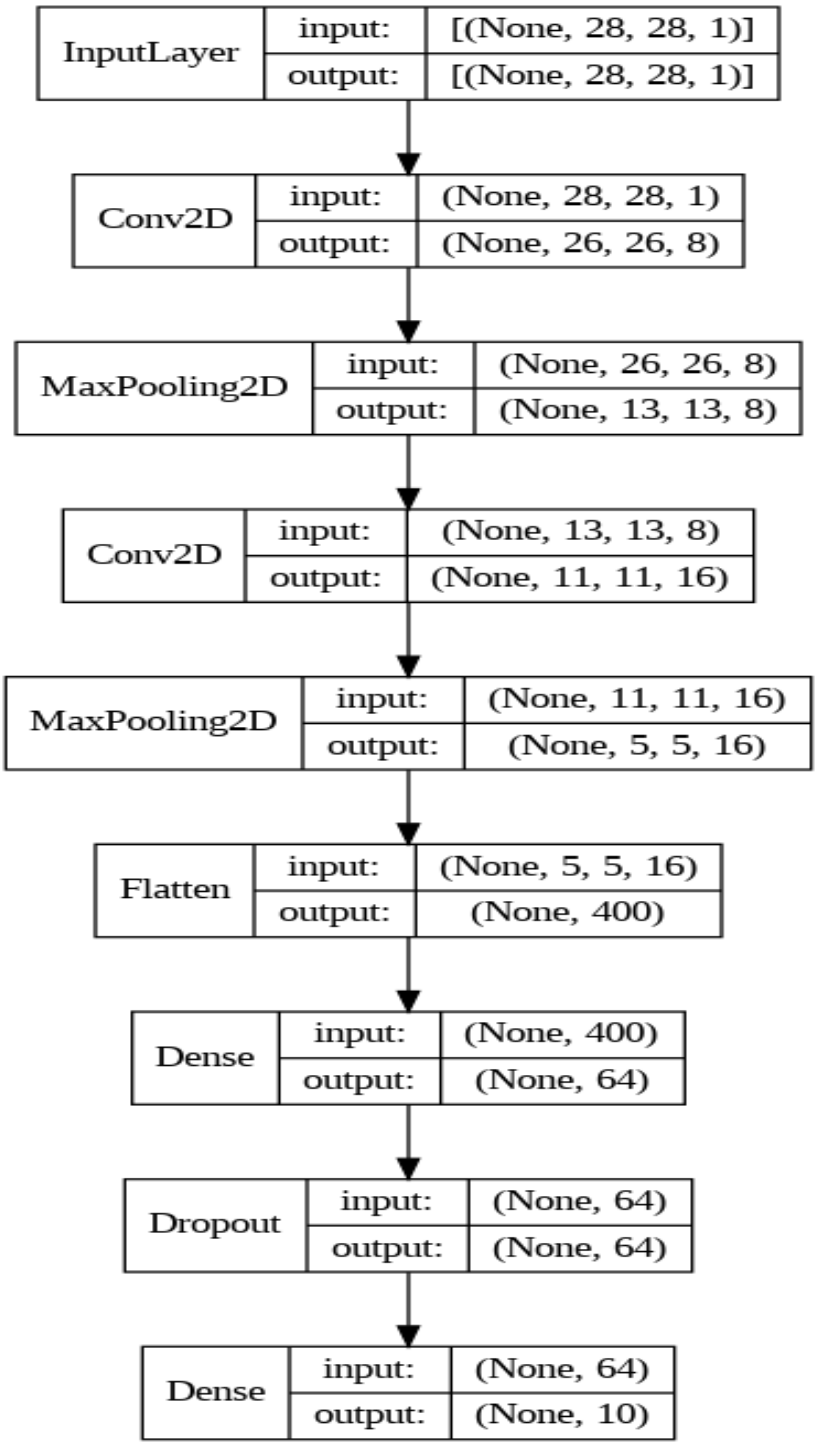
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 8)         80

 max_pooling2d (MaxPooling2  (None, 13, 13, 8)         0
 D)

 conv2d_1 (Conv2D)           (None, 11, 11, 16)        1168

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 16)          0
 g2D)

 flatten (Flatten)           (None, 400)               0

 dense (Dense)               (None, 64)                25664

 dropout (Dropout)           (None, 64)                0

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 27562 (107.66 KB)
Trainable params: 27562 (107.66 KB)
Non-trainable params: 0 (0.00 Byte)
```

```
# Conv1: 3x3 kernels, one for each the single channel, 8 such filters and 8
biases
print('Conv1: ',3*3*1*8 + 8)
# Conv2: 3x3 kernels, one for each of the 8 channels, 16 such filters and 16
biases
print('Conv2: ',3*3*8*16 + 16)
# input to dense layer
print('Flatten:', 5*5*16)
# 400 inputs, 1 bias connected to each of 64 units in dense layer
print('Dense1: ',400*64+64)
# 64 inputs, 1 bias connected to each of 10 units in output layer
print('Dense2: ',64*10+10)
```

```
Conv1:  80
Conv2:  1168
Flatten: 400
Dense1:  25664
Dense2:  650
```

```
# Visualize the model
import tensorflow.keras
keras.utils.plot_model(model, show_shapes=True, show_layer_names=False)
```



Train the model

1. Validation Data:
   - It's common practice to split a dataset into training and validation sets to evaluate the model's performance. Here, you have a total of 60,000 data points.
   - The validation data is determined as 20% of the total dataset, which is $0.2 \times 60{,}000 = 12{,}000$.
   - This means you set aside 12,000 data points specifically for validation, and the remaining 48,000 data points are used for training.

2. Batch Size:
   - Batch size refers to the number of data points used in each iteration of training. In this case, the batch size is set to 128.
   - During each training iteration, the model processes and updates its parameters based on 128 data points.

3. Number of Batches During Training:
   - The number of batches during training is calculated by dividing the total number of training data points by the batch size.
   - For this scenario: $(60{,}000 - 12{,}000) / 128 = 48{,}000/128 \approx 375$
   - This means there are 375 batches of 128 data points each that the model will iterate through during the training process.

```
# Train the model
batch_size=128
epochs=10
hist = model.fit(X_train,
y_train,epochs=epochs,batch_size=batch_size,verbose=1,validation_split=0.2)
```

```
Epoch 1/10
375/375 [==============================] - 15s 37ms/step - loss: 0.5012 - accuracy: 0.8482 - val_loss: 0.1305 - val_accuracy: 0.9619
Epoch 2/10
375/375 [==============================] - 14s 36ms/step - loss: 0.1381 - accuracy: 0.9586 - val_loss: 0.0851 - val_accuracy: 0.9752
Epoch 3/10
375/375 [==============================] - 14s 37ms/step - loss: 0.1012 - accuracy: 0.9695 - val_loss: 0.0766 - val_accuracy: 0.9768
Epoch 4/10
375/375 [==============================] - 14s 37ms/step - loss: 0.0840 - accuracy: 0.9743 - val_loss: 0.0597 - val_accuracy: 0.9818
Epoch 5/10
375/375 [==============================] - 14s 37ms/step - loss: 0.0729 - accuracy: 0.9778 - val_loss: 0.0595 - val_accuracy: 0.9822
Epoch 6/10
375/375 [==============================] - 14s 37ms/step - loss: 0.0619 - accuracy: 0.9810 - val_loss: 0.0538 - val_accuracy: 0.9843
Epoch 7/10
375/375 [==============================] - 15s 39ms/step - loss: 0.0585 - accuracy: 0.9823 - val_loss: 0.0491 - val_accuracy: 0.9859
Epoch 8/10
375/375 [==============================] - 17s 44ms/step - loss: 0.0535 - accuracy: 0.9832 - val_loss: 0.0490 - val_accuracy: 0.9851
Epoch 9/10
375/375 [==============================] - 14s 36ms/step - loss: 0.0484 - accuracy: 0.9846 - val_loss: 0.0474 - val_accuracy: 0.9862
Epoch 10/10
375/375 [==============================] - 14s 37ms/step - loss: 0.0444 - accuracy: 0.9859 - val_loss: 0.0455 - val_accuracy: 0.9849
```

Evaluate Model

```
score = model.evaluate(X_test, y_test, verbose = 0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.035951033234596 25
Test accuracy: 0.9876999855041504
```

```
# make one prediction
print('Actual class:',y_test[0])
print('Class Probabilities:')
model.predict(X_test[0].reshape(1,28,28,1))
```

```
Actual class: 7
Class Probabilities:
1/1 [==============================] - 0s 101ms/step
array([[1.3207576e-09, 9.3301921e-08, 5.9214752e-08, 8.1510763e-07,
        1.9626360e-11, 1.3402344e-09, 6.9678783e-16, 9.9999666e-01,
        7.9560741e-10, 2.3162620e-06]], dtype=float32)
```

```
import numpy as np
yhat_test = np.argmax(model.predict(X_test),axis=-1)
print(yhat_test[0:10])
print(y_test[0:10])
```

```
313/313 [==============================] - 1s 5ms/step
[7 2 1 0 4 1 4 9 5 9]
[7 2 1 0 4 1 4 9 5 9]
```

```
from sklearn.metrics import accuracy_score
print('Accuracy:')
print(float(accuracy_score(y_test, yhat_test))*100,'%')
```

```
Accuracy:
98.77 %
```

```
from sklearn.metrics import confusion_matrix
print('Confusion Matrix:')
print(confusion_matrix(y_test, yhat_test))
```

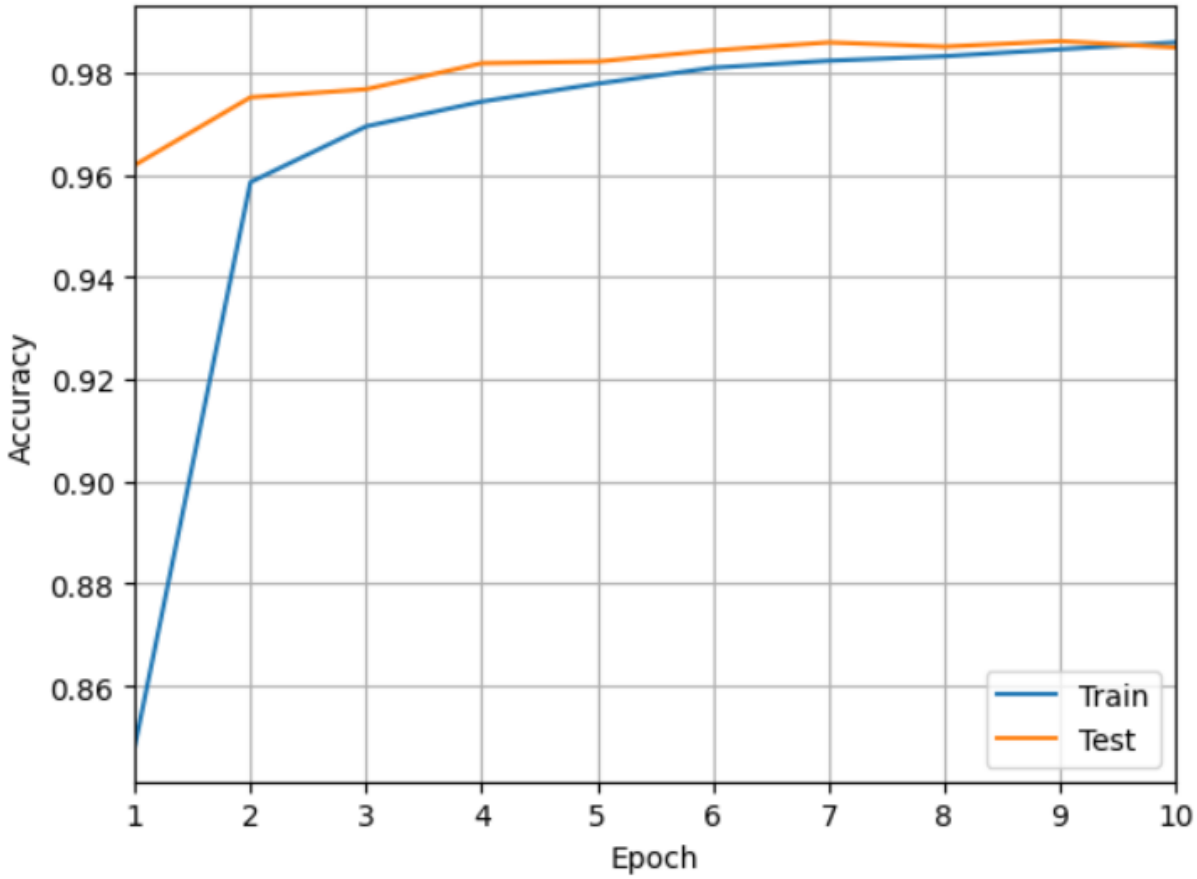
```
Confusion Matrix:
[[ 974    0    0    0    0    2    2    1    1    0]
 [   0 1131    1    1    0    0    1    1    0    0]
 [   1    1 1025    0    1    0    1    2    1    0]
 [   0    0    2  998    0    5    0    1    3    1]
 [   1    0    1    0  971    0    2    0    2    5]
 [   1    0    0    3    0  886    1    1    0    0]
 [   5    1    0    0    1    8  943    0    0    0]
 [   1    3   10    3    0    0    0 1007    1    3]
 [   5    0    4    0    1    3    0    4  953    4]
 [   2    3    1    1    2    6    0    4    1  989]]
```

Plot Learning Curves

```
hist.history.keys()
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
# Plot Accuracy vs epochs (DIY)
epochRange = range(1,epochs+1);
plt.plot(epochRange,hist.history['accuracy'])
plt.plot(epochRange,hist.history['val_accuracy'])
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.grid()
plt.xlim((1,epochs))
plt.legend(['Train','Test'])
plt.show()
```



```
# Plot Loss vs epochs (DIY)
epochRange = range(1,epochs+1);
plt.plot(epochRange,hist.history['loss'])
plt.plot(epochRange,hist.history['val_loss'])
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid()
plt.xlim((1,epochs))
plt.legend(['Train','Test'])
plt.show()
```