Experiment - 1

Aim: Build an Artificial Neural Network to implement Binary Classification task using the Back-propagation algorithm and test the same using appropriate data sets.

Description

The data used here is : '**Pima Indians Diabetes Dataset**'. It is downloaded from : https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.csv

It is a binary (2-class) classification problem. There are 768 observations with 8 input variables and 1 output variable.

The variable names are as follows:

1. Number of times pregnant.
2. Plasma glucose concentration 2 hours in an oral glucose tolerance test.
3. Diastolic blood pressure (mm Hg).
4. Triceps skinfold thickness (mm).
5. 2-Hour serum insulin (mu U/ml).
6. Body mass index (weight in kg/(height in m)^2).
7. Diabetes pedigree function.
8. Age (years).
9. Class variable (0 or 1).

1. Data Import and Processing

- **Importing Necessary Libraries**

```
import numpy as np              # For numerical operations and array handling
import pandas as pd             # For data manipulation and analysis
import matplotlib.pyplot as plt # For data visualization
import sklearn                  # For machine learning algorithms and tools
```

This code uses the pandas library to read the Pima Indians Diabetes dataset from the specified URL. The header=None argument indicates that the data file does not contain column headers.

After loading the data, two key operations are performed:

`.info()` method is used to display essential information about the DataFrame, such as the data types of columns, the number of non-null values, and memory usage.

`.head()` method is employed to show the initial rows of the dataset, providing a glimpse of the data structure and the first few records.

```
# load data
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.csv'
data_pd = pd.read_csv(url, header = None)
print(data_pd.info())
print(data_pd.head())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   0       768 non-null    int64  
 1   1       768 non-null    int64  
 2   2       768 non-null    int64  
 3   3       768 non-null    int64  
 4   4       768 non-null    int64  
 5   5       768 non-null    float64
 6   6       768 non-null    float64
 7   7       768 non-null    int64  
 8   8       768 non-null    int64  
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
None
   0    1   2   3    4     5      6   7  8
0  6  148  72  35    0  33.6  0.627  50  1
1  1   85  66  29    0  26.6  0.351  31  0
2  8  183  64   0    0  23.3  0.672  32  1
3  1   89  66  23   94  28.1  0.167  21  0
4  0  137  40  35  168  43.1  2.288  33  1
```

`StandardScaler()` is a preprocessing technique in machine learning that scales and centers numerical features to have zero mean and unit variance.

```
#Scaling Numerical columns
from sklearn.preprocessing import StandardScaler
std = StandardScaler()
scaled = std.fit_transform(data_pd.iloc[:,0:8])
scaled = pd.DataFrame(scaled)
scaled.head()
```

	0	1	2	3	4	5	6	7
0	0.639947	0.848324	0.149641	0.907270	-0.692891	0.204013	0.468492	1.425995
1	-0.844885	-1.123396	-0.160546	0.530902	-0.692891	-0.684422	-0.365061	-0.190672
2	1.233880	1.943724	-0.263941	-1.288212	-0.692891	-1.103255	0.604397	-0.105584
3	-0.844885	-0.998208	-0.160546	0.154533	0.123302	-0.494043	-0.920763	-1.041549
4	-1.141852	0.504055	-1.504687	0.907270	0.765836	1.409746	5.484909	-0.020496

Transforming and Extracting Features: Converts the scaled data to a NumPy array using `to_numpy()` and stores it in the variable `X_data`. The shape of the resulting array is printed to provide insights into the dimensions of the feature matrix.

Extracting Labels: Extracts the labels (target variable) from the original dataset using `iloc[:, 8]` and stores them in the variable `Y_data`. The shape of the resulting array is printed to provide insights into the dimensions of the label array.

```
# Transforming and Extracting Features
X_data = scaled.to_numpy()
print('X_data:', np.shape(X_data))

# Extracting Labels
Y_data = data_pd.iloc[:,8]
print('Y_data:', np.shape(Y_data))
```

```
X_data: (768, 8)
Y_data: (768,)
```

```
# Split data into X_train, X_test, y_train, y_test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_data, Y_data,
test_size=0.25, random_state= 0)

# Check the dimension of the sets
print('X_train:',np.shape(X_train))
print('y_train:',np.shape(y_train))
print('X_test:',np.shape(X_test))
print('y_test:',np.shape(y_test))
```

```
X_train: (576, 8)
y_train: (576,)
X_test: (192, 8)
y_test: (192,)
```

2. Design the Model

```
import keras
from keras.models import Sequential    # importing Sequential model
from keras.layers import Dense         # importing Dense layers
```

`Sequential()` A linear stack of layers executed sequentially in deep learning, easily created in Keras for building neural networks.

`Dense()` A fundamental fully connected layer in a neural network, connecting each neuron to all neurons in the previous and subsequent layers, characterized by parameters like units and activation function.

```
# declaring model
basic_model = Sequential()

# First layers: 8 neurons/perceptrons that takes the input and uses 'sigmoid'
activation function.
basic_model.add(Dense(8, input_dim = 8, activation = 'sigmoid'))

# Second layers: 4 neurons/perceptrons, 'sigmoid' activation function.
basic_model.add(Dense(4, activation = 'sigmoid'))

# Final layer: 1 neuron/perceptron to do binary classification
basic_model.add(Dense(1, activation = 'sigmoid'))
```

Compiling a Neural Network Model

This compilation step is essential before training a model, as it sets up the necessary configuration for the learning process.

```
basic_model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics
= ['accuracy'])
```

`(loss='binary_crossentropy')` specifies the loss function to measure the model's performance during training. In this case, it's binary crossentropy, commonly used for binary classification problems.

`(optimizer='adam')` defines the optimization algorithm to update the weights of the neural network during training. Adam is a popular and efficient optimization algorithm.

`(metrics=['accuracy'])` specifies the evaluation metric(s) to be monitored during training. Here, accuracy is chosen to measure how well the model predicts the target variable.

3. Train the Model

```
# training the model
epochs=120
history = basic_model.fit(X_train, y_train, validation_data=(X_test, y_test),
epochs=epochs)
```

Epoch 1/120
18/18 [==============================] - 0s 4ms/step - loss: 0.6666 - accuracy: 0.6424 - val_loss: 0.6279 - val_accuracy: 0.6771
Epoch 2/120
18/18 [==============================] - 0s 2ms/step - loss: 0.6582 - accuracy: 0.6424 - val_loss: 0.6236 - val_accuracy: 0.6771
Epoch 3/120
18/18 [==============================] - 0s 2ms/step - loss: 0.6518 - accuracy: 0.6424 - val_loss: 0.6208 - val_accuracy: 0.6771

.

.

.

Epoch 120/120
18/18 [==============================] - 0s 1ms/step - loss: 0.4801 - accuracy: 0.7778 - val_loss: 0.4451 - val_accuracy: 0.7865

`epochs` represents one complete pass through the entire training dataset. The parameter epochs specifies the number of times the learning algorithm will work through the entire training dataset. In this case, it's set to 120, indicating that the model will go through the training data 120 times.

`model.fit()` is used to train the neural network model. It takes the training data (X_train and y_train) as input, along with optional parameters like validation data (validation_data), the number of epochs (epochs), and more. The training process involves adjusting the model's weights based on the provided data, aiming to minimize the defined loss function.

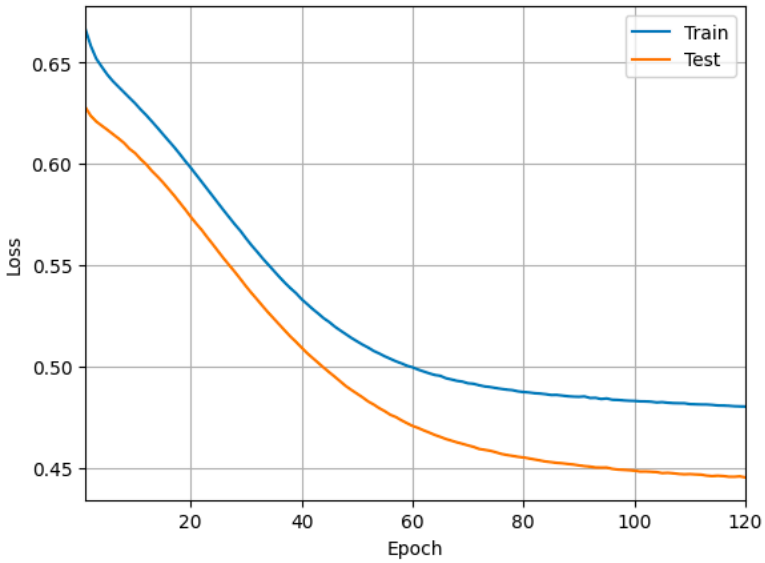
- The `validation_data` parameter is used to evaluate the model's performance on separate data during training. It helps monitor generalization and detect overfitting by comparing results on both training and validation sets, ensuring the model's reliability.

4. Evaluate the Model

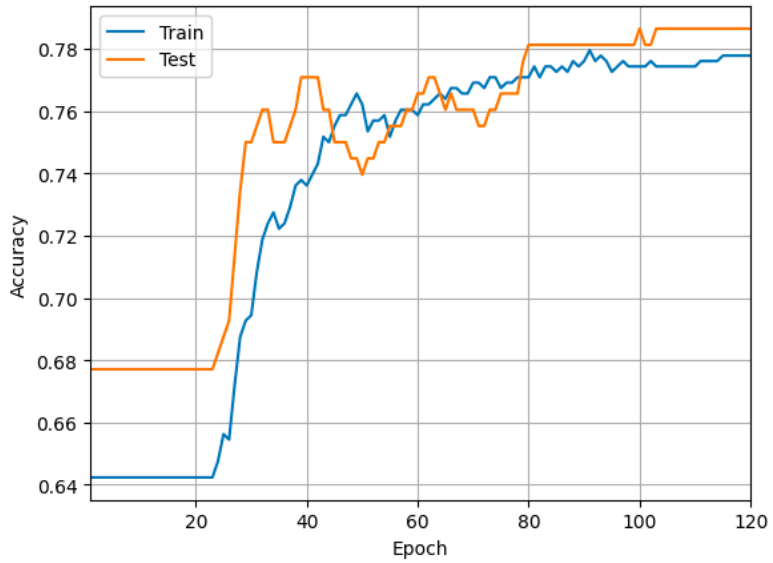
Model evaluation is the process of assessing the performance and effectiveness of a machine learning model on a given dataset. It involves measuring how well the model generalizes to new, unseen data. The goal is to understand how accurately the model predicts the target variable and whether it exhibits desirable characteristics for a specific task.

1. **Accuracy:** A metric representing the proportion of correctly predicted instances to the total, offering a simple measure of overall model correctness but may be limited for imbalanced datasets.
2. **Loss:** The discrepancy between predicted and actual values, minimized during training using optimization algorithms; diverse loss functions are used for various tasks like regression and classification.
3. **Epochs:** One complete pass through the entire training dataset during model training, determining the number of times the learning algorithm iterates over the data to improve the model's parameters and convergence.

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



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




```
# Test, Loss, and accuracy
loss_and_metrics = basic_model.evaluate(X_test, y_test)
print('Loss = ',loss_and_metrics[0])
print('Accuracy = ',loss_and_metrics[1])
```

```
6/6 [==============================] - 0s 542us/step - loss: 0.4451 - accuracy: 0.7865
Loss =  0.44512930512428284
Accuracy =  0.7864583134651184
```

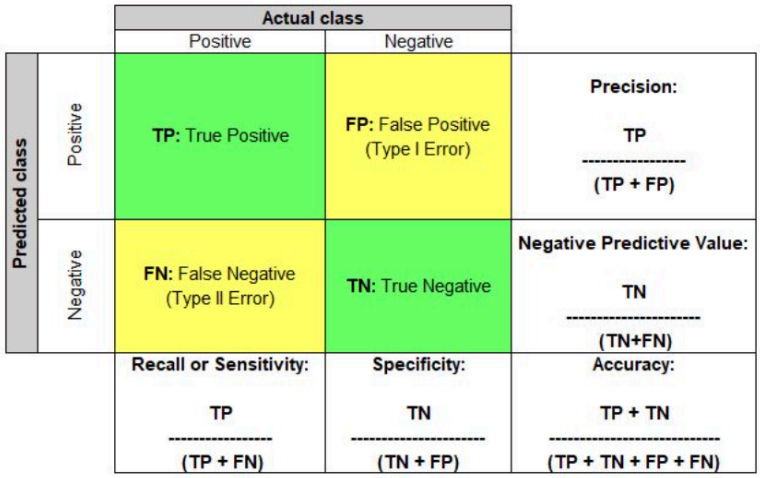
In conclusion, the visualizations of loss vs. epochs and accuracy vs. epochs showcase the model's effective learning, evidenced by decreasing losses and increasing accuracies.

The convergence of these metrics indicates a well-balanced optimization, ensuring the model not only understands the training data but also generalizes effectively to new examples.

It's crucial to recognize that while both loss and accuracy aim for improvement, low loss doesn't necessarily guarantee high accuracy, and vice versa.

Therefore, a holistic evaluation of multiple metrics is essential for a comprehensive assessment of the model's performance.

5. Classification Model Performance measures




```
y_pred = basic_model.predict(X_test)
print(y_pred[:5])
print(y_test[:5])
```

```
6/6 [==============================] - 0s 486us/step
[[0.7795038 ]
 [0.11919793]
 [0.08359037]
 [0.7181756 ]
 [0.12440205]]
661    1
122    0
113    0
14     1
529    0
Name: 8, dtype: int64
```

```
y_pred =[1 if y_pred[aa]>=0.5 else 0 for aa in range(len(y_pred)) ]
print(y_pred[:5])
```

```
[1, 0, 0, 1, 0]
```

- The model predicts probabilities for each example, and a threshold (commonly 0.5) is typically applied to convert these probabilities into binary predictions (0 or 1).
- A common threshold of 0.5 is applied to convert these probabilities into binary predictions (0 or 1).

- The model predicts class 1 for examples 1 and 4, while predicting class 0 for examples 2, 3 and 5. Comparing these predictions with the actual labels reveals instances where the model correctly or incorrectly identifies the class.
- This assessment helps evaluate the model's performance in terms of accuracy and highlights areas where it may require further refinement.

Assessing Performance on Test Data

Precision is the ratio of true positive predictions to the total predicted positives.

Recall (Sensitivity) is the ratio of true positive predictions to the total actual positives.

F1-Score is the harmonic mean of precision and recall, providing a balanced measure of a classifier's performance.

Support: The number of actual occurrences of each class in the specified dataset.

Accuracy: Overall accuracy of the model in correctly predicting both classes is 0.79 (79%)

Macro Avg: The unweighted average of precision, recall, and F1-score across classes.

Weighted Avg: The weighted average of precision, recall, and F1-score, considering the number of instances for each class.

```
print(sklearn.metrics.classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.83      0.86      0.85       130
           1       0.68      0.63      0.66        62

    accuracy                           0.79       192
   macro avg       0.76      0.75      0.75       192
weighted avg       0.78      0.79      0.78       192
```

- The model performs well in classifying class 0, as indicated by high precision, recall, and F1-score. Class 1 has slightly lower performance, with lower precision, recall, and F1-score.
- Overall accuracy is 79%, suggesting a reasonably accurate classification.
- Macro and weighted averages provide a summary of the model's overall performance across classes, considering both precision and recall.

Experiment - 2

Aim: Build an Artificial Neural Network to implement Multi-Class Classification task using the Back-propagation algorithm and test the same using appropriate data sets

Database

- The data that will be incorporated is the **MNIST database** (Modified National Institute of Standards and Technology 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.

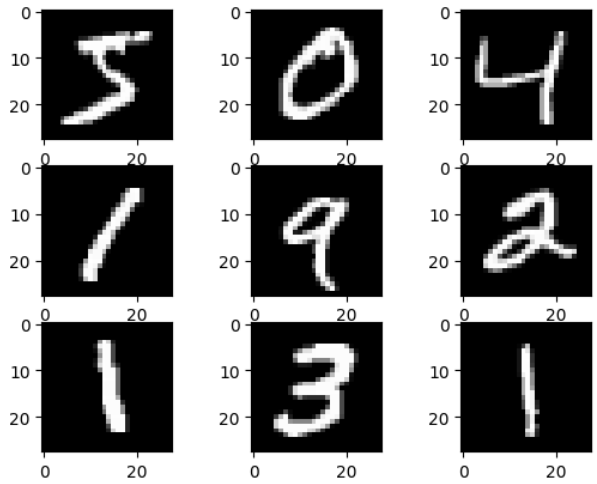
```
pip install matplotlib
```

Here we used TensorFlow and Keras to load the MNIST dataset for handwritten digit recognition. It unpacks the dataset into training and test sets (X_train, y_train, X_test, y_test) and prints their respective shapes:

```
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)
```

```
X_train shape (60000, 28, 28)
y_train shape (60000,)
X_test shape (10000, 28, 28)
y_test shape (10000,)
```

```
# Plot first few images
import matplotlib.pyplot as plt
for i in range(9):
        # define subplot
        plt.subplot(3,3,i+1) # 3 rows, 3 col, pos
        # plot raw pixel data
        plt.imshow(X_train[i], cmap='gray')
# show the figure
plt.show()
```



```
X_train[i].shape
```

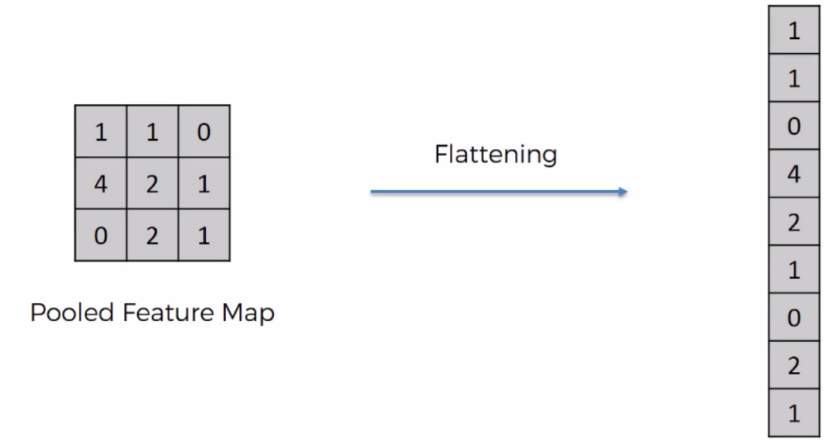
```
(28, 28)
```

```
# Each pixel is an 8-bit integer from 0-255 (0 is full black, 255 is full white)
# single-channel pixel or monochrome image
X_train[i][10:20,10:20]
```

```
array([[  0,   0,  20, 254, 254, 108,   0,   0,   0,   0],
       [  0,   0,  16, 239, 254, 143,   0,   0,   0,   0],
       [  0,   0,   0, 178, 254, 143,   0,   0,   0,   0],
       [  0,   0,   0, 178, 254, 143,   0,   0,   0,   0],
       [  0,   0,   0, 178, 254, 162,   0,   0,   0,   0],
       [  0,   0,   0, 178, 254, 240,   0,   0,   0,   0],
       [  0,   0,   0, 113, 254, 240,   0,   0,   0,   0],
       [  0,   0,   0,  83, 254, 245,  31,   0,   0,   0],
       [  0,   0,   0,  79, 254, 246,  38,   0,   0,   0],
       [  0,   0,   0,   0, 214, 254, 150,   0,   0,   0]], dtype=uint8)
```

Formatting the input data

- Reshape (or flatten) the 28x28 image into a 784-length vector.



- Input values [0-255] are Normalized in the range [0-1]

A Min-Max Scaling is typically done via the following equation:

$$X_{norm} = \frac{X_i - X_{min}}{X_{max} - X_{min}}$$

X_i is the i^{th} sample of dataset.

```
# reshape 28 x 28 matrices into 784-length vectors
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)

# normalize each value for each pixel for the entire vector for each input
# change integers to 32-bit floating point numbers
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# normalize by dividing by largest pixel value
X_train /= 255
X_test /= 255

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

```
Training matrix shape (60000, 784)
Testing matrix shape (10000, 784)
```

DNN for Multi-Class classification using Keras library

Build the model

To construct the model for deep learning, import necessary functions like Sequential and Dense from the Keras library. Use Sequential for model architecture and Dense for layer creation.

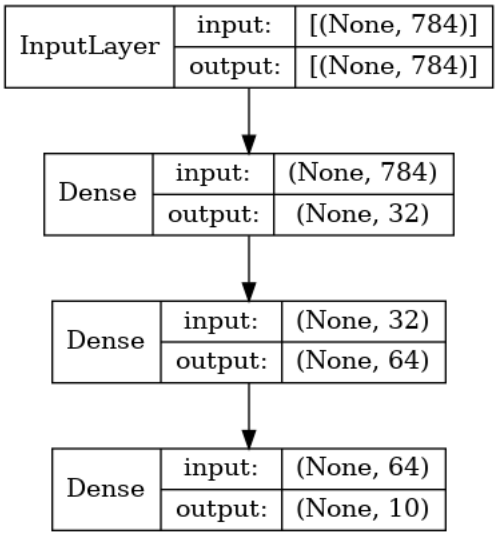
- **Sequential**: A linear stack of layers for building neural networks. It allows you to add one layer at a time, ensuring a straightforward and organized architecture.
- **Dense**: This represents a fully connected layer in the neural network. Each node in a dense layer is connected to every node in the previous layer. It is a fundamental building block for constructing neural networks.

Store the model function in a variable for easier access. Create a series of dense layers, stacking them with 'relu' as the activation function. Further stack layers with 'softmax' activation.

Compile the entire model using categorical-crossentropy as the loss function, adam as the optimizer, and accuracy as the evaluation metric for the MNIST dataset.

```
# Sequential keras model with Dense layes (DIY)

from keras.models import Sequential  # Model type to be used
from keras.layers import Dense # Types of layers to be used in our model

from keras.models import Sequential
from keras.layers import Dense
model = Sequential()

# Hidden layer with 32 units and relu activation
model.add(Dense(32, input_dim = 28 * 28, activation= 'relu'))
# Input layer with 64 units and relu activation
model.add(Dense(64, activation = 'relu'))
# Output layer with 10 units and softmax activation
model.add(Dense(10, activation = 'softmax'))

# Compile model
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=[
        'accuracy',
        'AUC',
    ]
)
```

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



```
# # Visualize the model
# from keras.utils.vis_utils import plot_model
# plot_model(model, show_shapes=True, show_layer_names=False)
# Display model summary
model.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_21 (Dense)            (None, 32)                25120

 dense_22 (Dense)            (None, 64)                2112

 dense_23 (Dense)            (None, 10)                650

=================================================================
Total params: 27882 (108.91 KB)
Trainable params: 27882 (108.91 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
#understand model summary
784*64 + 64
```

```
50240
```

The expression **`784 * 64 + 64`** represents the total number of parameters in a dense neural network layer, where 784 is the number of input nodes, 64 is the number of neurons, and the additional 64 accounts for bias terms associated with each neuron.

```
64*32 + 32
```

```
2080
```

```
32*10 + 10
```

```
330
```

Convert labels to "one-hot" vectors using the to_categorical function

0 -> [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
1 -> [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
2 -> [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
etc.

```
from tensorflow.keras.utils import to_categorical
y_train1 = to_categorical(y_train)
y_test1 = to_categorical(y_test)
print(y_test[6])
print(y_test1[6,:])
```

```
4
[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
```

Train the model

Model training is the process of teaching a neural network to make predictions by adjusting its internal parameters based on a labeled dataset. It involves presenting the model with input data, comparing its predictions to the actual labels, and updating the parameters to minimize the prediction error.

Parameters:

- **Batch Size:** The number of training examples utilized in one iteration. It impacts memory usage and training speed. Common values are 32, 64, or 128.

  If unspecified, by default `batch_size = 32`
  60,000/64 = 938 minibatches

- **Epochs:** An epoch is a complete pass through the entire training dataset. During each epoch, the model goes through all the training samples. Training for more epochs allows the model to learn from the data for an extended period.

- **Validation Data:** A separate set of data used to evaluate the model's performance during training. It helps prevent overfitting, where the model performs well on training data but fails on new, unseen data.

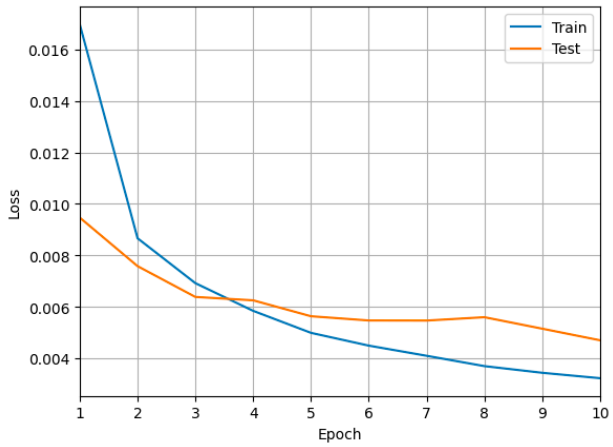
```
# Train the model
epochs=10
batch = 64
history = model.fit(X_train, y_train1,epochs=epochs,
batch_size=batch,verbose=1, validation_data=(X_test, y_test1))
```

```
Epoch 1/10
938/938 [==============================] - 2s 1ms/step - loss: 0.0170 - accuracy:
0.8887 - auc: 0.9889 - val_loss: 0.0095 - val_accuracy: 0.9383 - val_auc: 0.9942
Epoch 2/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0087 - accuracy:
0.9445 - auc: 0.9949 - val_loss: 0.0076 - val_accuracy: 0.9510 - val_auc: 0.9956
Epoch 3/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0069 - accuracy:
0.9556 - auc: 0.9962 - val_loss: 0.0064 - val_accuracy: 0.9598 - val_auc: 0.9963
.
.
.
Epoch 10/10
938/938 [==============================] - 1s 1ms/step - loss: 0.0032 - accuracy:
0.9803 - auc: 0.9978 - val_loss: 0.0047 - val_accuracy: 0.9714 - val_auc: 0.9969
```

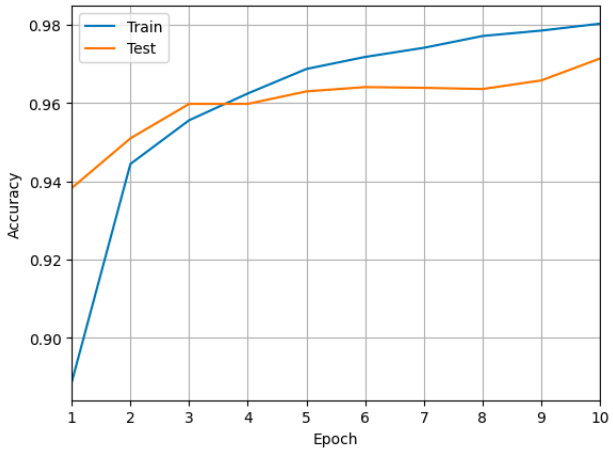
Evaluate Model

Plot Learning graphs

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




```
plt.plot(epochRange,history.history['accuracy'])
plt.plot(epochRange,history.history['val_accuracy'])
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.grid()
plt.xlim((1,epochs))
plt.legend(['Train','Test'])
plt.show()
```



Performance metrics

```
import numpy as np
yhat_test_mdl_prob = model.predict(X_test);
yhat_test_mdl = np.argmax(yhat_test_mdl_prob,axis=-1)
print(yhat_test_mdl_prob[0])
print(yhat_test_mdl[0:10])
print(y_test[0:10])
313/313 [==============================] - 0s 505us/step
[3.65186601e-08 1.17585762e-06 4.24575010e-05 9.66793741e-04
 1.00649966e-09 1.65969539e-07 3.17241296e-13 9.98968720e-01
 1.04621195e-05 1.01733503e-05]
[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_mdl))*100,'%')
```

```
Accuracy:
97.14 %
```

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

```
Confusion Matrix:
[[ 967    0    2    2    0    2    3    2    2    0]
 [   0 1120    4    3    0    1    0    3    3    1]
 [   3    1 1004    6    4    2    2    6    4    0]
 [   0    0   12  976    0    7    1    5    9    0]
 [   1    1    9    0  951    1    4    2    1   12]
 [   5    2    0    6    2  861    8    1    5    2]
 [   9    2    1    2    4    6  929    1    4    0]
 [   0    4   16    1    0    0    0  996    6    5]
 [   2    1    5    7    3    5    2    4  943    2]
 [   5    4    0    9    9    3    0    3    9  967]]
```

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.

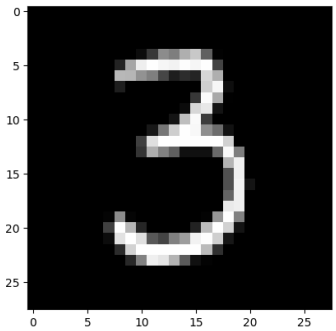
Database

- 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)
```

```
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');
```



`.reshape(60000, 28, 28, 1)` reshapes the training dataset X_train from a 3D array (60000, 28, 28) to a 4D array (60000, 28, 28, 1). This is necessary because convolutional layers in a CNN expect input data to have a certain shape, typically in the form (batch_size, height, width, channels). In this case, the images are grayscale, so they only have one channel.

`.astype('float32')` to convert the data type of the training dataset to float32. Neural networks often perform better with floating-point numbers, and this conversion ensures consistency in data type.

`/= 255` performs min-max normalization on the training dataset X_train. It divides each pixel value by 255, the maximum pixel value for grayscale images. This operation scales the pixel values to the range [0, 1], making the neural network easier to train.

```
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: 2.13.1 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'])
```

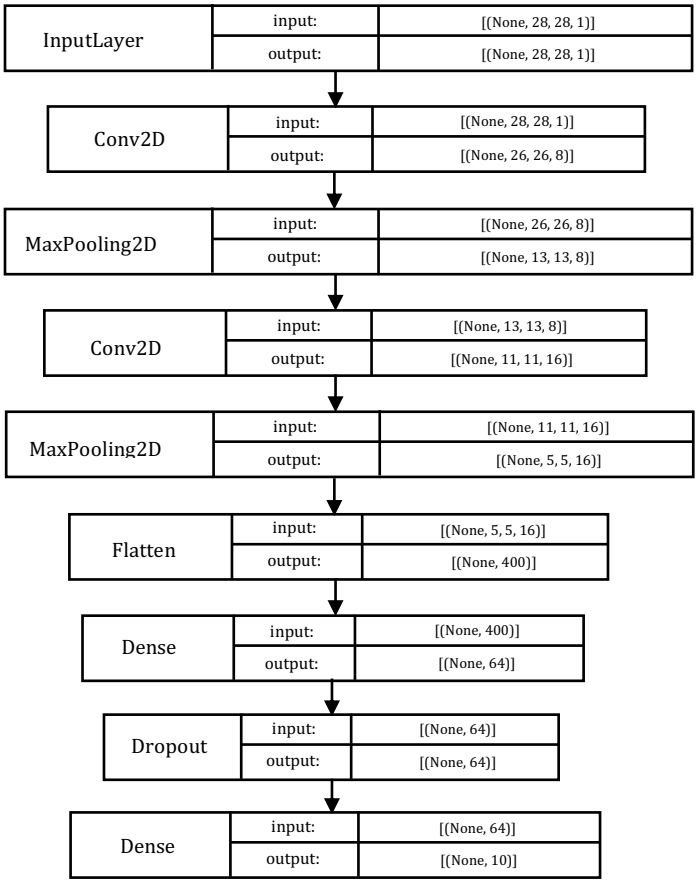
```
model.summary()
```

```
Model: "sequential_22"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_24 (Conv2D)          (None, 26, 26, 8)         80

 max_pooling2d_24 (MaxPooli  (None, 13, 13, 8)         0
 ng2D)

 conv2d_25 (Conv2D)          (None, 11, 11, 16)        1168

 max_pooling2d_25 (MaxPooli  (None, 5, 5, 16)          0
 ng2D)

 flatten_12 (Flatten)        (None, 400)               0

 dense_20 (Dense)            (None, 64)                25664

 dropout_12 (Dropout)        (None, 64)                0

 dense_21 (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 ≈ 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 [==============================] - 4s 9ms/step - loss: 0.5659 -
accuracy: 0.8289 - val_loss: 0.1552 - val_accuracy: 0.9530
.

.

.
Epoch 10/10
375/375 [==============================] - 3s 8ms/step - loss: 0.0566 -
accuracy: 0.9815 - val_loss: 0.0502 - val_accuracy: 0.9858
```

Evaluate Model

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

```
Test loss: 0.042321018874645 23
Test accuracy: 0.9865000247955322
```

```
# 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 35ms/step

array([[3.1840685e-08, 8.0150443e-08, 2.7107694e-07, 3.4357882e-07,
        5.7956888e-09, 7.4466217e-10, 5.0047981e-13, 9.9999881e-01,
        2.2633260e-09, 4.1972484e-07]], 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 [==============================] - 0s 1ms/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:
98.65 %
```

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

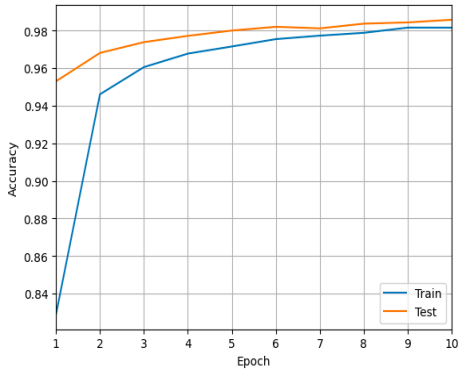
```
Confusion Matrix:
[[ 975    1    0    0    0    0    3    1    0    0]
 [   0 1129    1    0    0    1    3    1    0    0]
 [   4    3 1017    1    0    0    0    6    0    1]
 [   0    0    3  995    0    6    0    3    3    0]
 [   1    0    0    0  970    0    2    0    0    9]
 [   2    0    0    3    0  881    4    1    1    0]
 [   4    2    0    0    1    3  946    0    2    0]
 [   1    2    6    3    0    0    0 1012    2    2]
 [   8    0    1    3    0    1    1    3  957    0]
 [   4    2    0    4    4    2    0    5    5  983]]
```

Plot Learning Curves

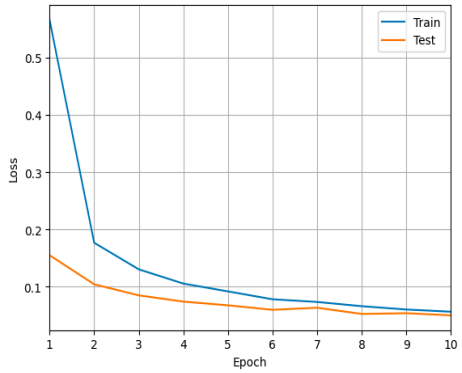
```
hist.history.keys()
```

```
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()
```



Model 2

Fine-tuning the model's hyperparameters to assess its performance with updated settings: adjusting epochs, kernel size, filter size, pool size, and dropout rate for enhanced optimization.

```
model = Sequential()

# Convolution Layer 2: 16 filters, kernel size 4x4, relu activation, same
padding, stride 1
model.add(Conv2D(16, (4, 4), activation='relu', padding=same, strides=(2, 2),
input_shape=(28,28,1) ))

# MaxPooling: pool size 2, stride 1
model.add(MaxPooling2D(pool_size=(2, 2), strides=(1, 1)))

# Convolution Layer 2: 32 filters, kernel size 2x2, relu activation, valid
padding, stride 1
model.add(Conv2D(32, (2, 2), 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, 0.3 rate
model.add(Dropout(0.3))

# Final output dense Layer
model.add(Dense(15, activation='softmax'))

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

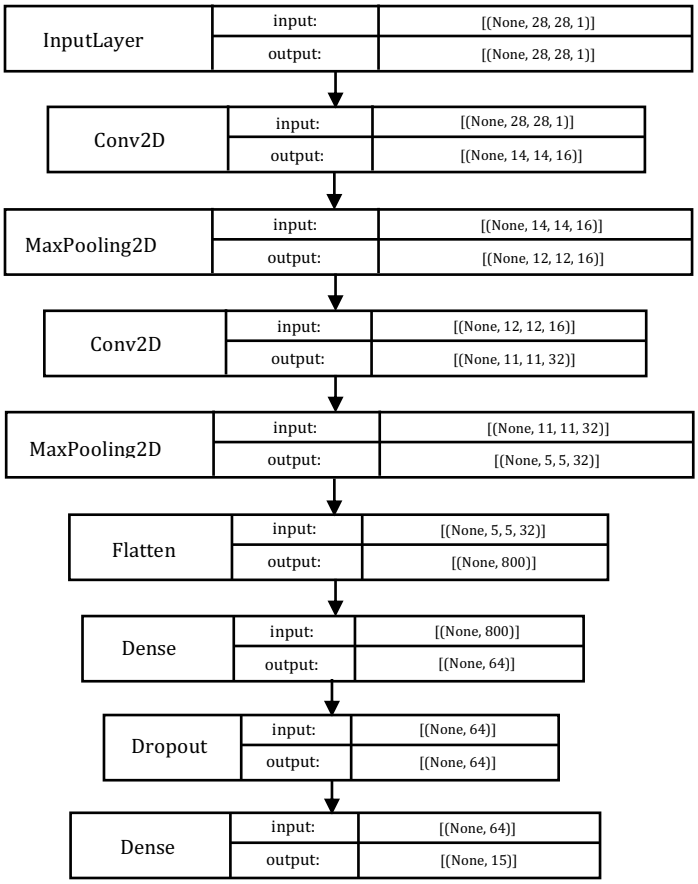
```
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 14, 14, 16)        272

 max_pooling2d_6 (MaxPoolin  (None, 12, 12, 16)        0
 g2D)

 conv2d_7 (Conv2D)           (None, 11, 11, 32)        2080

 max_pooling2d_7 (MaxPoolin  (None, 5, 5, 32)          0
 g2D)

 flatten_3 (Flatten)         (None, 800)               0

 dense_6 (Dense)             (None, 64)                51264

 dropout_3 (Dropout)         (None, 64)                0

 dense_7 (Dense)             (None, 15)                975

=================================================================
Total params: 54591 (213.25 KB)
Trainable params: 54591 (213.25 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:  272
Conv2:  2080
Flatten: 800
Dense1:  51264
Dense2:  975
```

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




```
# 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
94/94 [==============================] - 3s 25ms/step - loss: 1.2600 -
accuracy: 0.5849 - val_loss: 0.2901 - val_accuracy: 0.9259
.

.

.
Epoch 50/50
94/94 [==============================] - 2s 26ms/step - loss: 0.0879 -
accuracy: 0.9741 - val_loss: 0.0501 - val_accuracy: 0.9843
```

Evaluate Model

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

```
Test loss: 0.03131077438592911
Test accuracy: 0.9902999997138977
```

```
# 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 42ms/step

array([[1.1213746e-15, 1.5772349e-11, 1.1608250e-09, 5.2119318e-09,
        1.6685747e-16, 3.8218425e-11, 2.3289047e-22, 9.9999988e-01,
        1.7811259e-11, 7.5471739e-08, 3.6340089e-15, 1.8331014e-17,
        2.4177230e-18, 2.4037379e-16, 3.7645257e-16]], 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 [==============================] - 0s 1ms/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,'%')
```

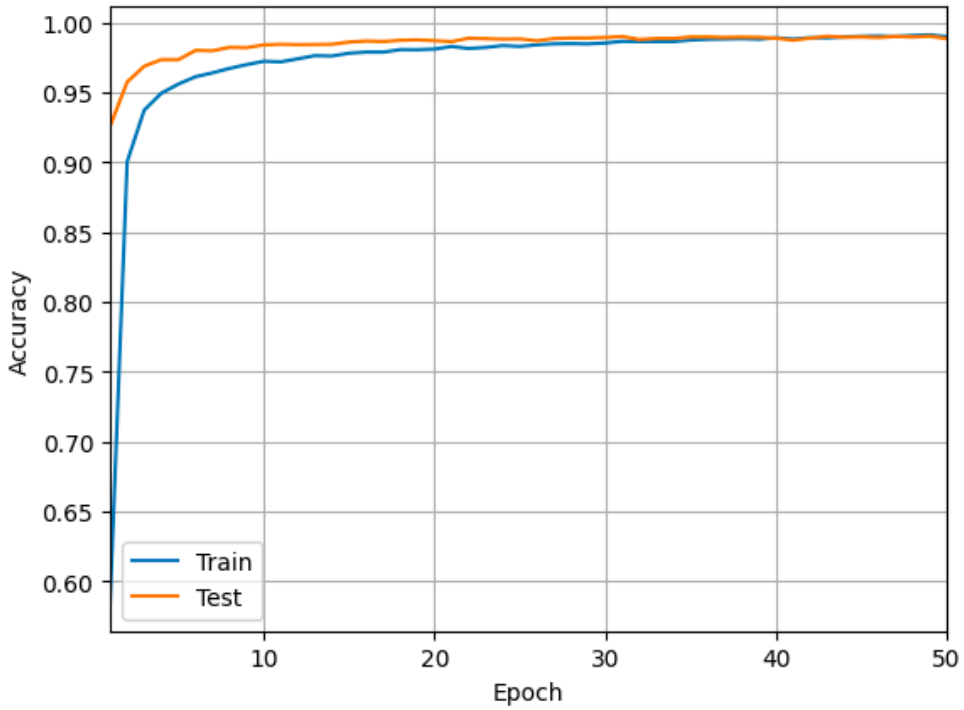
```
Accuracy:
99.03 %
```

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

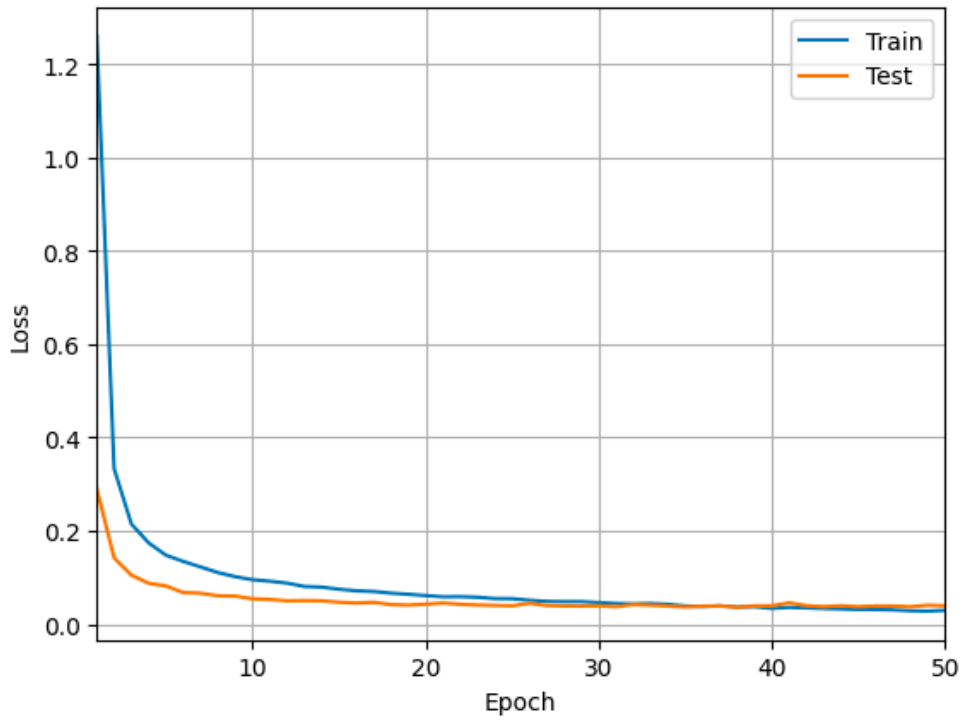
```
Confusion Matrix:
[[ 977    0    0    0    0    0    2    1    0    0]
 [   0 1133    1    0    0    0    1    0    0    0]
 [   1    2 1018    1    1    0    0    7    2    0]
 [   0    0    0 1001    0    3    0    2    4    0]
 [   0    0    0    0  977    0    1    2    0    2]
 [   2    0    0    5    0  882    2    1    0    0]
 [   2    2    1    0    1    3  949    0    0    0]
 [   0    2    2    0    0    0    0 1022    1    1]
 [   3    1    0    1    0    1    1    1  960    6]
 [   1    0    0    0    8    5    0    5    6  984]]
```

Plot Learning Curves

```
# 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()
```



Comparing Models Performance

```
# Model Evaluation
import pandas as pd
model_performance = pd.DataFrame(columns=['Accuracy'])
```

```
model_performance.loc['Model 1'] = [m1Acc[0]]
model_performance.loc['Model 2'] = [m2Acc[0]]
```

```
# Evaluation and Display
model_performance.fillna(.90, inplace=True)
formatted_performance = model_performance.applymap(lambda x:
'{:.2%}'.format(x))
model_performance.fillna(.90,inplace=True)
model_performance.style.background_gradient(cmap='coolwarm').format({
                                                    'Accuracy': '{:}%'
                                                     })
```

	Accuracy
Model 1	98.49%
Model 2	99.03%

Experiment - 4

Aim: Implement an image classification task using pre-trained models like VGGNet, InceptionNet and ResNet and compare the results.

Description

Transfer Learning is a machine learning technique where a model trained on one task is reused or adapted as a starting point for a new task. It's particularly useful when you have a small dataset for your target task because it allows leveraging knowledge gained from a larger dataset or a different but related task. Basically in Transfer Learning, you can take the pre-trained VGG16 model, remove its output layer, and replace it with a new set of fully connected layers tailored to your classification task, and fine-tune the model with your data.

Models used for this experiment are:

1. VGG16: VGG16 is a convolutional neural network architecture with 16 layers developed by the Visual Geometry Group at the University of Oxford. It's known for its simplicity and uniform architecture.
2. InceptionNet (Inception V3): InceptionNet is a deep convolutional neural network architecture developed by Google. It's known for its inception modules, which allow the network to efficiently use computational resources.
3. ResNet: ResNet, short for Residual Network, is a deep neural network architecture that introduced skip connections to address the vanishing gradient problem in very deep networks. ResNet comes in various depths such as ResNet-50, ResNet-101, etc.

Program

```
import keras
import numpy as np
from keras import Input
from keras import models
from keras import layers
from keras import optimizers
from keras.models import Model
from keras import applications
# from keras import backend
import matplotlib.pyplot as plt
from keras.optimizers import SGD, Adam
from keras.layers import  Dense, Flatten, Conv2D, MaxPooling2D
# from keras.preprocessing import image
from keras.models import Sequential, Model
from keras.preprocessing.image import ImageDataGenerator
from keras.layers import Dropout, Flatten, Dense, GlobalAveragePooling2D
# from keras.callbacks import ModelCheckpoint, LearningRateScheduler,
TensorBoard, EarlyStopping
```

Loading the Training and Testing Data and Defining the Basic Parameters

About data: In this data-set, 39 different classes of plant leaf and background images are available. The data-set containing 61,486 images. We used six different augmentation techniques for increasing the data-set size. The techniques are image flipping, Gamma correction, noise injection, PCA color augmentation, rotation, and Scaling.

- We are resizing the input image to 128 * 128
- In the dataset : Training Set : 70% Validation Set : 20% Test Set : 10%

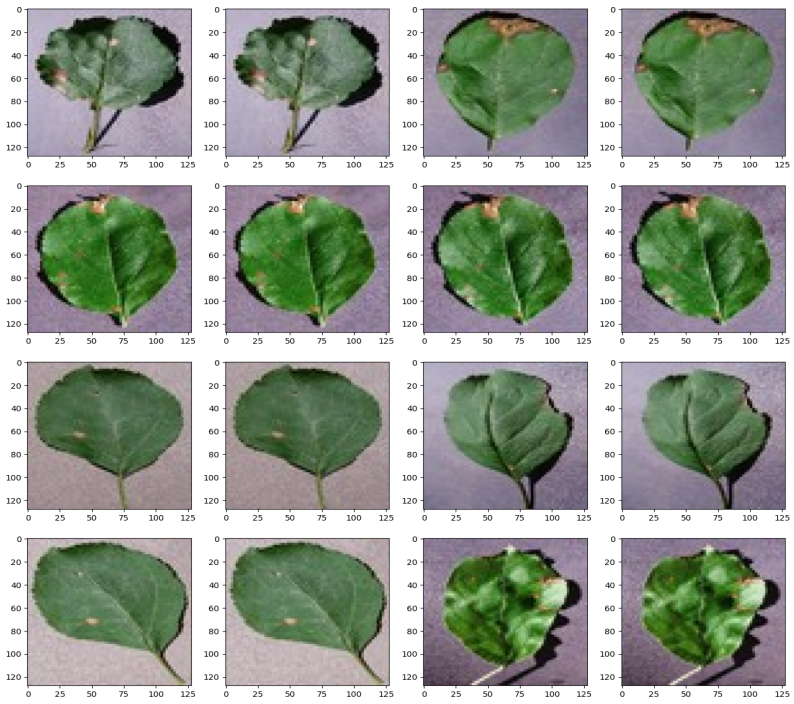
```
!unzip "Google\ Drive/Documents/plant_village.zip" -d "plant_village"
```

```
# Normalize training and validation data in the range of 0 to 1
train_datagen = ImageDataGenerator(rescale=1/255) # vertical_flip=True,
                                                  # horizontal_flip=True,
                                                  # height_shift_range=0.1,
                                                  # width_shift_range=0.1
validation_datagen = ImageDataGenerator(rescale=1/255)
test_datagen = ImageDataGenerator(rescale=1/255)

# Read the training sample and set the batch size
train_generator = train_datagen.flow_from_directory(
        "/Users/ab01/Library/CloudStorage/GoogleDrive-
amanbarthwal0110@gmail.com/My
Drive/Programming/Python/AiML/GenAi/plant_village/train",
        target_size=(128, 128),
        batch_size=16,
        class_mode='categorical')

# Read Validation data from directory and define target size with batch size
validation_generator = validation_datagen.flow_from_directory(
        "/Users/ab01/Library/CloudStorage/GoogleDrive-
amanbarthwal0110@gmail.com/My
Drive/Programming/Python/AiML/GenAi/plant_village/val",
        target_size=(128, 128),
        batch_size=16,
        class_mode='categorical',
        shuffle=False)

test_generator = test_datagen.flow_from_directory(
        "/Users/ab01/Library/CloudStorage/GoogleDrive-
amanbarthwal0110@gmail.com/My
Drive/Programming/Python/AiML/GenAi/plant_village/test",
        target_size=(128, 128),
        batch_size=1,
        class_mode='categorical',
        shuffle=False)
```

```
Found 3033 images belonging to 4 classes.
Found 0635 images belonging to 4 classes.
Found 0566 images belonging to 4 classes.
```

Visualization of Few Images

```
plt.figure(figsize=(16, 16))
for i in range(1, 17):
  plt.subplot(4, 4, i)
  img, label = test_generator.next()
  # print(img.shape)
  # print(label)
  plt.imshow(img[0])
plt.show()
```



What is ImageNet?

ImageNet is formally a project aimed at (manually) labeling and categorizing images into almost 22,000 separate object categories for the purpose of computer vision research.

However, when we hear the term “ImageNet” in the context of deep learning and Convolutional Neural Networks, we are likely referring to the ImageNet Large Scale Visual Recognition Challenge, or ILSVRC for short.

The goal of this image classification challenge is to train a model that can correctly classify an input image into 1,000 separate object categories.

Models are trained on ~1.2 million training images with another 50,000 images for validation and 100,000 images for testing.

Exploring Keras Applications for Transfer Learning

Keras provides a set of pre-trained models through its keras.applications module, which are trained on large-scale image datasets like ImageNet. These pre-trained models can be used for transfer learning by either using them as feature extractors or fine-tuning them for specific tasks. Some commonly used pre-trained models available in Keras include:

1. VGG16, VGG19: A convolutional neural network architecture with 16 layers, known for its simplicity. VGG19 is similar to VGG16 but with 19 layers.
2. ResNet50, ResNet101, ResNet152: Residual neural network architectures with 50, 101, and 152 layers respectively, which introduced skip connections to address the vanishing gradient problem.
3. InceptionV3: A deep convolutional neural network architecture developed by Google, known for its inception modules.
4. Xception: A deep convolutional neural network architecture also developed by Google, known for its depthwise separable convolutions.
5. MobileNet: A lightweight deep convolutional neural network architecture designed for mobile and embedded vision applications.
6. MobileNetV2: An improved version of MobileNet with inverted residuals and linear bottlenecks.
7. DenseNet121, DenseNet169, DenseNet201: Dense convolutional neural network architectures where each layer is connected to every other layer in a feed-forward fashion.

VGG16

```
from tensorflow.keras.applications.vgg16 import VGG16
```

```
## Loading VGG16 model
base_model = VGG16(weights="imagenet", include_top=False, input_shape= (128, 128, 3)) # Include_top = False means excluding the model fully connected layers
base_model.trainable = False ## Not trainable weights

base_model.summary()
```

```
Model: "vgg16"

=================================================================
Total params: 14714688 (56.13 MB)
Trainable params: 0 (0.00 Byte)
Non-trainable params: 14714688 (56.13 MB)
_________________________________________________________________
```

Adding top layers according to number of classes in our data

```
flatten_layer = layers.GlobalAveragePooling2D()
prediction_layer = layers.Dense(4, activation='softmax')

model = models.Sequential([
    base_model,
    flatten_layer,
    prediction_layer
])

model.summary()
```

```
Model: "sequential_4"

=================================================================
Total params: 14716740 (56.14 MB)
Trainable params: 2052 (8.02 KB)
Non-trainable params: 14714688 (56.13 MB)
_________________________________________________________________
```

Training

```
# We are going to use accuracy metrics and cross entropy loss as performance
parameters
model.compile(optimizer = Adam(learning_rate = 0.001),
loss='categorical_crossentropy', metrics=['acc'])
# Train the model
history = model.fit(train_generator,
      steps_per_epoch=train_generator.samples/train_generator.batch_size,
      epochs=10,
      validation_data=validation_generator,

validation_steps=validation_generator.samples/validation_generator.batch_size
,
      verbose=1)
```

```
Epoch 1/10

190/189 [==============================] - ETA: 0s - loss: 1.0615 -
acc: 0.5724

.

.

.
Epoch 10/10
189/189 [==============================] - 170s 898ms/step - loss:
0.2497 - acc: 0.9311 - val_loss: 0.2365 - val_acc: 0.9370
```

Saving the model

```
model.save("VGG16_plant_deseas.h5")
print("Saved model to disk”)
```

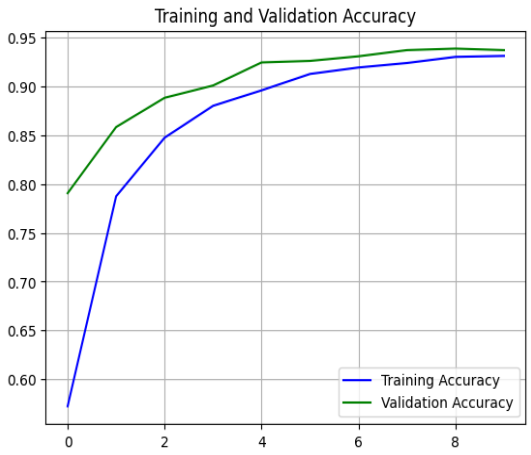
Loading the model

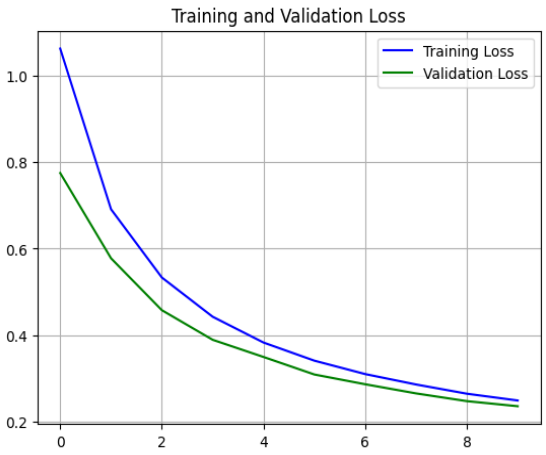
```
model = models.load_model('VGG16_plant_deseas.h5')
print("Model is loaded”)
```

Visualization of training over epoch

```
train_acc = history.history['acc']
val_acc = history.history['val_acc']
train_loss = history.history['loss']
val_loss = history.history['val_loss']
```

```
epochs = range(len(train_acc))
plt.plot(epochs, train_acc, 'b', label='Training Accuracy')
plt.plot(epochs, val_acc, 'g', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.grid()
plt.legend()
plt.figure()
plt.show()
plt.plot(epochs, train_loss, 'b', label='Training Loss')
plt.plot(epochs, val_loss, 'g', label='Validation Loss')
plt.title('Training and Validation Loss')
plt.grid()
plt.legend()
plt.show
```





Performance measure

```
# Get the filenames from the generator
fnames = test_generator.filenames

# Get the ground truth from generator
ground_truth = test_generator.classes

# Get the label to class mapping from the generator
label2index = test_generator.class_indices

# Getting the mapping from class index to class label
idx2label = dict((v,k) for k,v in label2index.items())

# Get the predictions from the model using the generator
predictions = model.predict_generator(test_generator,
steps=test_generator.samples/test_generator.batch_size,verbose=1)
predicted_classes = np.argmax(predictions,axis=1)

errors = np.where(predicted_classes != ground_truth)[0]
print("No of errors = {}/{}".format(len(errors),test_generator.samples))
```

```
1/566 [..............................] - ETA: 1:27

566/566 [==============================] - 31s 54ms/step
No of errors = 57/566
```

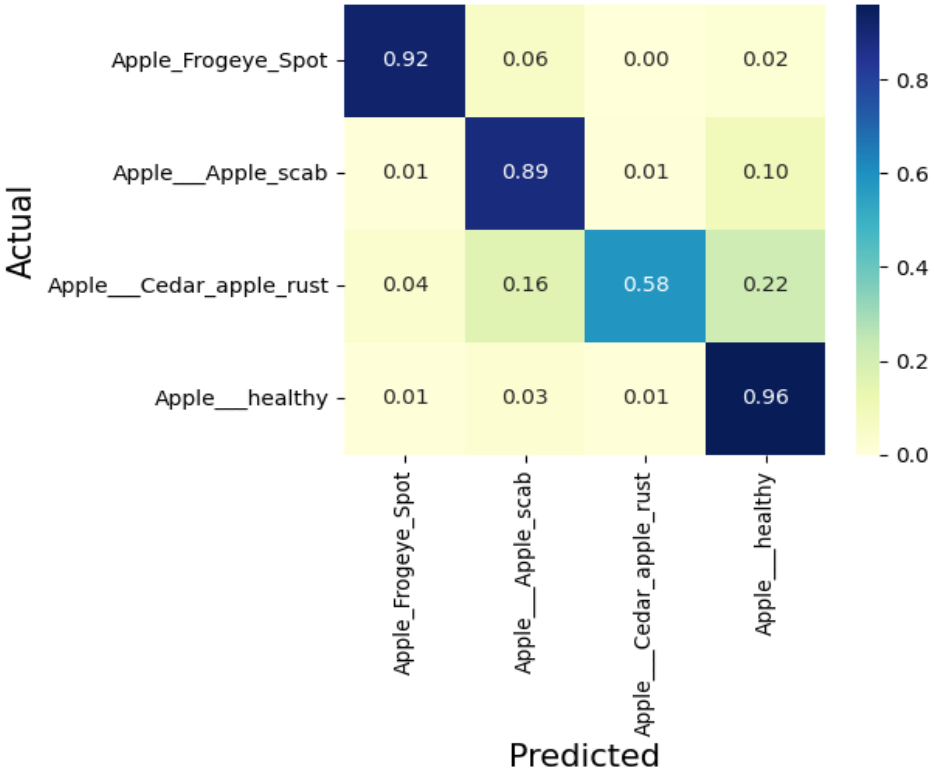
```
accuracyVGG = ((test_generator.samples-len(errors))/test_generator.samples) * 100
accuracyVGG
```

```
89.92932862190813
```

```
!pip install seaborn
```

```
Requirement already satisfied: seaborn in matplotlib!=3.6.1,>=3.1-
>seaborn) (2.8.2)
```

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import numpy as np
from matplotlib import pyplot as plt
cm = confusion_matrix(y_true=ground_truth, y_pred=predicted_classes)
cm = np.array(cm)
# Normalise
cmn = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
fig, ax = plt.subplots(figsize=(5,4))
sns.heatmap(cmn, annot=True, fmt='.2f', xticklabels=label2index, yticklabels=label2index,
cmap="YlGnBu")
plt.ylabel('Actual', fontsize=15)
plt.xlabel('Predicted', fontsize=15)
plt.show(block=False)
```




```
from sklearn.metrics import classification_report
print(classification_report(ground_truth, predicted_classes,
target_names=label2index))
```

```
                      precision    recall  f1-score   support

      Apple_Frogeye_Spot       0.95      0.92      0.94       103
      Apple___Apple_scab       0.84      0.89      0.87       134
Apple___Cedar_apple_rust       0.91      0.58      0.71        55
        Apple___healthy       0.91      0.96      0.93       274

               accuracy                           0.90       566
              macro avg       0.90      0.84      0.86       566
           weighted avg       0.90      0.90      0.90       566
```

InceptionNet

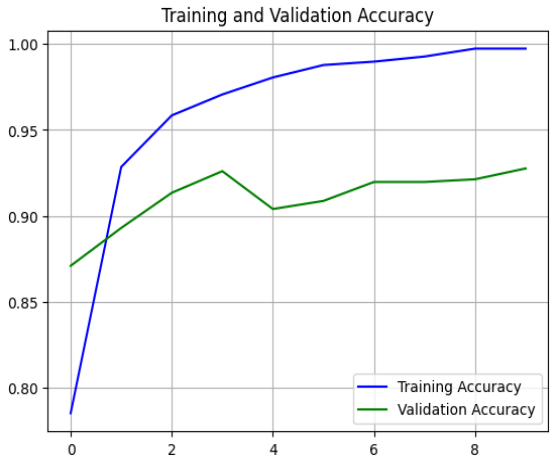
```
from keras import applications

## Loading InceptionV3 model
base_model = applications.InceptionV3(weights="imagenet", include_top=False,
input_shape= (128, 128, 3))
base_model.trainable = False ## Not trainable weights

base_model.summary()
```

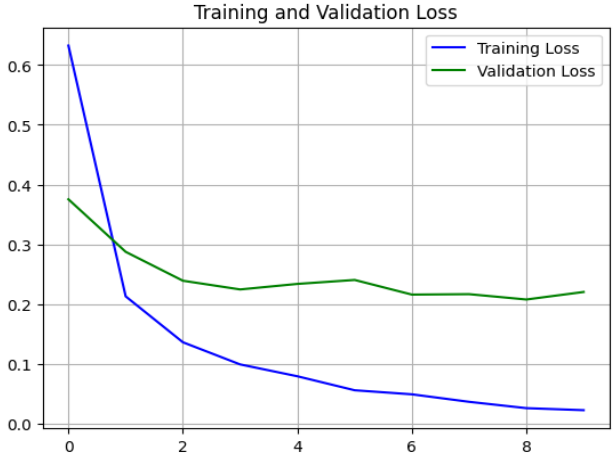
```
=================================================================
=============================
Total params: 21802784 (83.17 MB)
Trainable params: 0 (0.00 Byte)
Non-trainable params: 21802784 (83.17 MB)
_________________________________________________________________
_______________________________
```

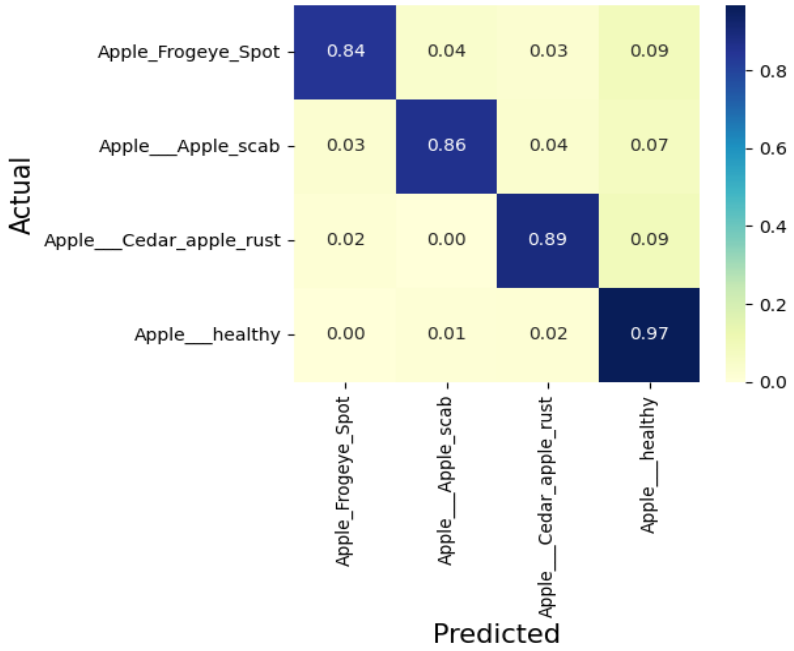
```
# include GlobalAveragePooling2D
flatten_layer = layers.GlobalAveragePooling2D()

# include final Dense layer
prediction_layer = layers.Dense(4, activation='softmax')

model = models.Sequential([
    base_model,
    flatten_layer,
    prediction_layer
])

model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 inception_v3 (Functional)   (None, 2, 2, 2048)        21802784

 global_average_pooling2d_5  (None, 2048)              0
  (GlobalAveragePooling2D)

 dense_5 (Dense)             (None, 4)                 8196

=================================================================
Total params: 21810980 (83.20 MB)
Trainable params: 8196 (32.02 KB)
Non-trainable params: 21802784 (83.17 MB)
_________________________________________________________________
```

```
# Save model
model.save("InceptioNet_plant_deseas.h5")
print("Saved model to disk")
```







	precision	recall	f1-score	support
Apple_Frogeye_Spot	0.95	0.84	0.89	103
Apple___Apple_scab	0.93	0.86	0.89	134
Apple___Cedar_apple_rust	0.79	0.89	0.84	55
Apple___healthy	0.92	0.97	0.94	274
accuracy			0.91	566
macro avg	0.90	0.89	0.89	566
weighted avg	0.91	0.91	0.91	566

ResNet

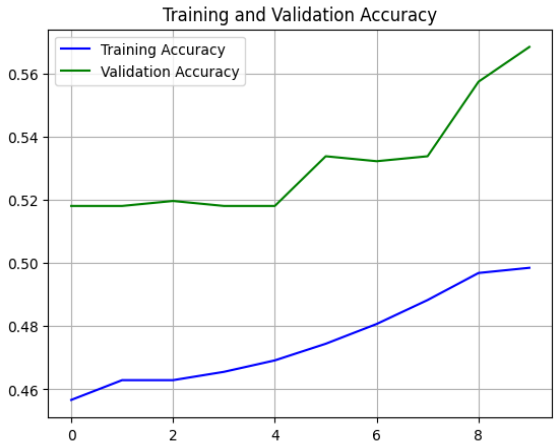
```
from keras import applications

## Loading VGG16 model
base_model = applications.ResNet50(weights="imagenet", include_top=False,
input_shape= (128, 128, 3))
base_model.trainable = False ## Not trainable weights

base_model.summary()
```

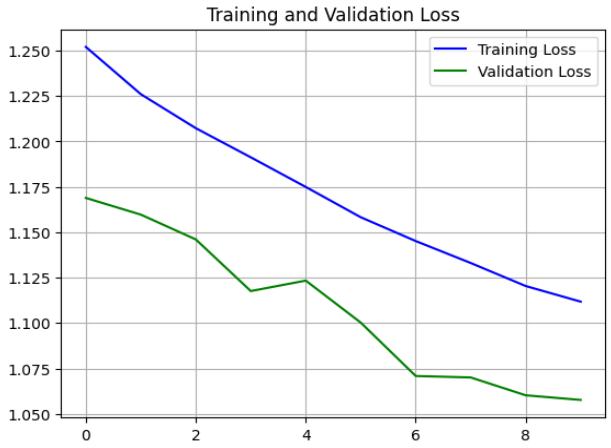
```
Model: "resnet50"

==================================================================================
==============================
Total params: 23587712 (89.98 MB)
Trainable params: 0 (0.00 Byte)
Non-trainable params: 23587712 (89.98 MB)

_________________________________________________________________________________
_______________________________
```

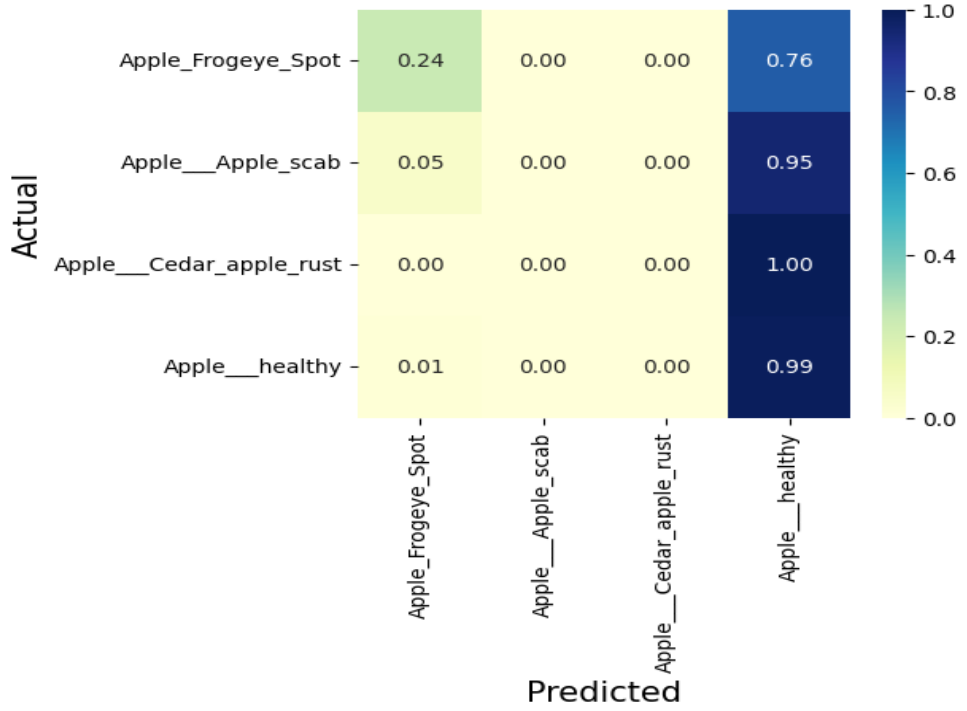
```
pip install matplotlib
# include GlobalAveragePooling2D
flatten_layer = layers.GlobalAveragePooling2D()

# include final Dense layer
prediction_layer = layers.Dense(4, activation='softmax')

model = models.Sequential([
    base_model,
    flatten_layer,
    prediction_layer
])
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50 (Functional)       (None, 4, 4, 2048)        23587712

 global_average_pooling2d_6  (None, 2048)              0
  (GlobalAveragePooling2D)

 dense_6 (Dense)             (None, 4)                 8196

=================================================================
Total params: 23595908 (90.01 MB)
Trainable params: 8196 (32.02 KB)
Non-trainable params: 23587712 (89.98 MB)

_________________________________________________________________
```

```
# Save the model
model.save("ResNet_plant_deseas.h5")
print("Saved model to disk")
```







	precision	recall	f1-score	support
Apple_Frogeye_Spot	0.71	0.24	0.36	103
Apple___Apple_scab	0.00	0.00	0.00	134
Apple___Cedar_apple_rust	0.00	0.00	0.00	55
Apple___healthy	0.51	0.99	0.67	274
accuracy			0.52	566
macro avg	0.31	0.31	0.26	566
weighted avg	0.38	0.52	0.39	566

Conclusion

The performance of VGG16, InceptionNet, and ResNet for image classification using transfer learning varies based on the specific dataset and task at hand. In this scenario, after training each model for 30 epochs, InceptionNet achieved the highest accuracy score of 91.17%, followed by VGG16 with 89.93%, and ResNet with 52.30%.

```
# Model Evaluation
import pandas as pd
model_performance = pd.DataFrame(columns=['Accuracy'])
```

```
model_performance.loc['VGG'] = accuracyVGG
model_performance.loc['Inception Net'] = accuracyInceptioNet
model_performance.loc['Residual Net'] = accuracyResNet
```

```
# Evaluation and Display
model_performance.fillna(.90, inplace=True)
formatted_performance = model_performance.applymap(lambda x: '{:.2%}'.format(x))
model_performance.fillna(.90,inplace=True)
model_performance.style.background_gradient(cmap='coolwarm').format({
                                                'Accuracy': '{:.2f}%'
                                                })
```

	Accuracy
VGG	89.93%
Inception Net	91.17%
Residual Net	52.30%

Experiment 5

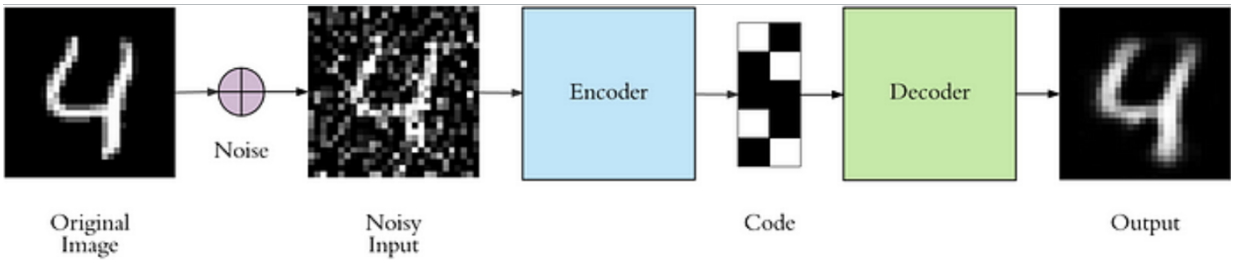
Aim: Implement Autoencoder architecture for denoising images.

Description

An autoencoder is a special type of neural network that is trained to copy its input to its output. For example, given an image of a handwritten digit, an autoencoder first encodes the image into a lower dimensional latent representation, then decodes the latent representation back to an image.

- The encoder- This network downsamples the data into lower dimensions.
- The decoder- This network reconstructs the original data from the lower dimension representation.

The one thing which we must remember about autoencoders is that they are only able to compress the data that is similar to what they have been trained on. They are also lossy in nature which means that the output will be degraded with respect to the original input.

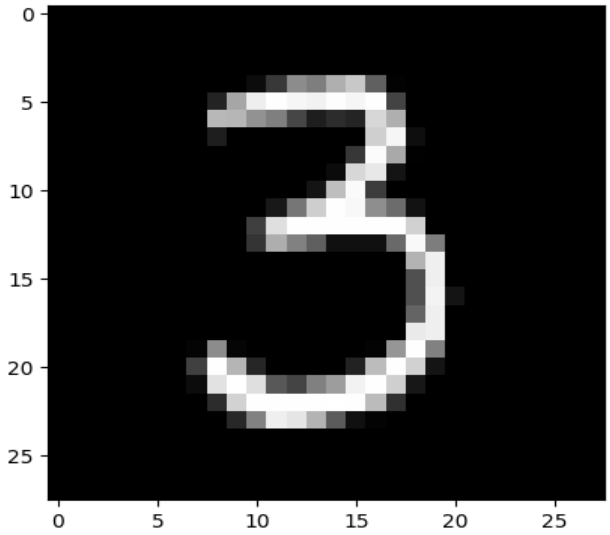


Program

```
# Import Dataset
import tensorflow
from tensorflow.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()
```

```
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

```
# reshape 28 x 28 matrices into 784-length vectors
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)

# normalize each value for each pixel for the entire vector for each input
# change integers to 32-bit floating point numbers
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# normalize by dividing by largest pixel value
X_train /= 255
X_test /= 255

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

```
Training matrix shape (60000, 784)
Testing matrix shape (10000, 784)
```

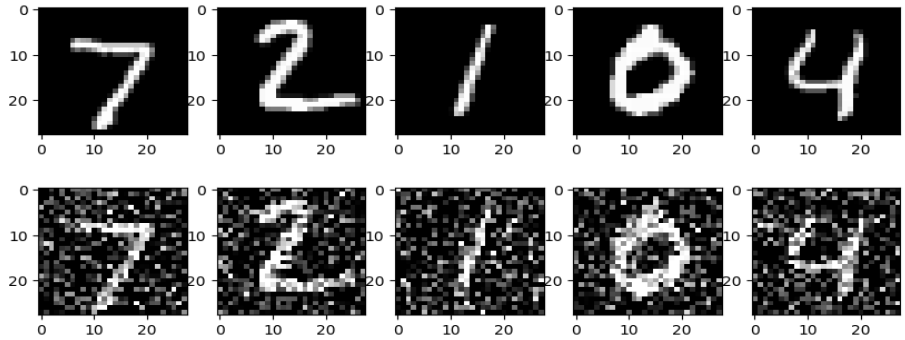
```
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Dense, Input
```

Create noisy data

While solving the problem statement, we have to remember our goal which is to make a model that is capable of performing noise removal on images. To be able to do this, we will use existing images and add them to random noise. Here we will feed the original images as input and we get the noisy images as output and our model (i.e, autoencoder) will learn the relationship between a clean image and a noisy image and learn how to clean a noisy image. So let's create a noisy version of our MNIST dataset and give it as input to the decoder network.

We start with defining a noise factor which is a hyperparameter. The noise factor is multiplied with a random matrix that has a mean of 0.0 and a standard deviation of 1.0. This matrix will draw samples from a normal (Gaussian) distribution. While adding the noise, we have to remember that the shape of the random normal array will be similar to the shape of the data you will be adding the noise.

```
# Add noise to input
import numpy as np

noise_factor = 0.4
X_train_noisy = X_train + noise_factor * np.random.normal(size=X_train.shape)
X_test_noisy  = X_test  + noise_factor * np.random.normal(size=X_test.shape)

X_train_noisy= np.clip(X_train_noisy, 0.0, 1.0)
X_test_noisy = np.clip(X_test_noisy , 0.0, 1.0)

n = 5
plt.figure(figsize=(10, 4.5))
for i in range(n):
    # plot original image
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(X_test[i].reshape(28, 28))
    plt.gray()
    if i == n/2:
        ax.set_title('Original Images')

    # plot noisy image
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(X_test_noisy[i].reshape(28, 28))
    plt.gray()
    if i == n/2:
        ax.set_title('Noisy Input')
```



To ensure that our final images array item values are within the range of 0 to 1, we may use np.clip method. The clip is a Numpy function that clips the values outside of the Min-Max range and replaces them with the designated min or max value.

Design and train Fully connected DAE

It will have an input layer of 784 neurons since we have an image size of 784 due to 28 by 28 pixels present (i.e. we have the input dimension and an output layer of 784 neurons).

```
X_train[i].shape
input_size = 784
hidden_size = 128
code_size = 32

# Complete the code
dae = Sequential()

# Input layer
dae.add(Dense(784, activation='relu', input_dim=input_size))
# input_layer = Input(shape=(input_size,784))

# Hidden layer 1 in Encoder with 128 units, relu activation
dae.add(Dense(128, activation='relu'))

# Hidden layer 2 (Code ) in Encoder with 32 units, relu activation
dae.add(Dense(32, activation='relu'))

# Hidden layer 1 in Decoder with 128 units, relu activation
dae.add(Dense(128, activation='relu'))

# Hidden layer 2 in Encoder with 784 units, sigmoid activation
dae.add(Dense(784, activation='sigmoid'))

# Compile the model, adam optimizer MeanSquaredError loss function
dae.compile(optimizer='adam', loss='mean_squared_error')
# autoencoder = Model(input_layer, decoder_layer)
```

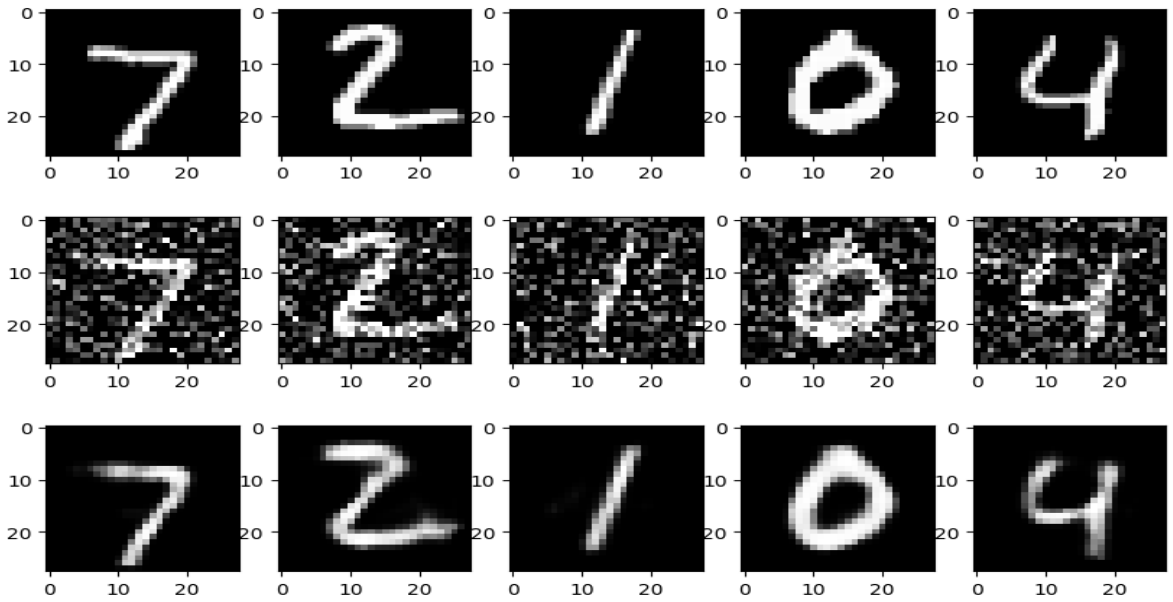
```
# Display model Summary
dae.summary
```

```
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_54 (Dense)            (None, 784)               615440

 dense_55 (Dense)            (None, 128)               100480

 dense_56 (Dense)            (None, 32)                4128

 dense_57 (Dense)            (None, 128)               4224

 dense_58 (Dense)            (None, 784)               101136

=================================================================
Total params: 825408 (3.15 MB)
Trainable params: 825408 (3.15 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
# Reconstruct Images from Noisy X_test images
dae.fit(X_train_noisy, X_train, validation_data=(X_test_noisy, X_test),
epochs=5)
X_test_noisy_recons = dae.predict(X_test_noisy)
```

```
Epoch 1/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.015
0 - val_loss: 0.0171
Epoch 2/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.014
7 - val_loss: 0.0167
Epoch 3/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.014
4 - val_loss: 0.0167
Epoch 4/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.014
2 - val_loss: 0.0165
Epoch 5/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.013
9 - val_loss: 0.0166
313/313 [==============================] - 0s 895us/step
```

```
n = 5

plt.figure(figsize=(10, 7))

for i in range(n):
    # plot original image
    ax = plt.subplot(3, n, i + 1)
    plt.imshow(X_test[i].reshape(28, 28))
    plt.gray()
    if i == n/2:
        ax.set_title('Original Images')

    # plot noisy image
    ax = plt.subplot(3, n, i + 1 + n)
    plt.imshow(X_test_noisy[i].reshape(28, 28))
    plt.gray()
    if i == n/2:
        ax.set_title('Noisy Input')

    # plot noisy image
    ax = plt.subplot(3, n, i + 1 + 2*n)
    plt.imshow(X_test_noisy_recons[i].reshape(28, 28))
    plt.gray()
    if i == n/2:
        ax.set_title('Autoencoder Output')
```



In the above output, the images in the:

- The first row is for original test images,
- The second row is for noisy images, and
- The third row is for cleaned (reconstructed) images.

Conclusion

The autoencoder effectively enhanced image quality by denoising, removing unwanted artifacts and distortions while retaining essential features. Through its learning process, it identified and reconstructed clean versions of noisy images, resulting in clearer, crisper outputs compared to their original noisy counterparts.

The reconstructed images may appear blurry due to factors like insufficient training epochs. To address this, consider increasing the number of epochs and reevaluate the image quality. Despite some imperfections, the autoencoder effectively cleans highly noisy images from the test dataset. Considering the significant improvement from the original noisy images, integrating this autoencoder into a photo enhancement app could enhance photo clarity and crispness.

Experiment - 6

Aim: Implement GAN architecture on MNIST dataset to recognize the handwritten digits

Generative Adversarial Networks (GANs)

Generative Adversarial Networks is basically an approach to generative modeling that generates a new set of data based on training data that look like training data. GANs have two main blocks(two neural networks) which compete and are able to capture, copy, and analyze the variations in a dataset. The two models are usually called Generator and Discriminator.

- Generative – To learn a generative model, which describes how data is generated in terms of a probabilistic model. In simple words, it explains how data is generated visually.
- Adversarial – The training of the model is done in an adversarial setting.
- Networks – use deep neural networks for training purposes.

Importing Necessary Libraries

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Input, Dense, Reshape, Flatten, Conv2D,
Conv2DTranspose, LeakyReLU
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import mnist

# Load MNIST data
(X_train,_), (_, _) = mnist.load_data()

X_train.shape
```

```
(60000, 28, 28)
```

```
# Normalize data
X_train = (X_train.astype(np.float32) - 127.5) / 127.5
X_train = np.expand_dims(X_train, axis=3)
print(X_train.shape)
```

```
(60000, 28, 28, 1)
```

Building Discriminator

It is a supervised approach means It is a simple classifier that predicts data is fake or real. It is trained on real data and provides feedback to a generator.

```
def build_discriminator():
    model = Sequential()

    # Conv2D layer with 64 filters, kernel size 4x4, stride 2, padding same (no
activation)
    model.add(Conv2D(64, (4,4), strides=2, padding="same", input_shape=(28,28,1)))

    # LeakyReLU activation with negative_slope 0.2
    model.add(LeakyReLU(0.2))

    # Conv2D layer with 128 filters, kernel size 4x4, stride 2, padding same (no
activation)
    model.add(Conv2D(128, (4,4), strides=2, padding="same"))

    # LeakyReLU activation with negative_slope 0.2
    model.add(LeakyReLU(0.2))

    # Flatten layer
    model.add(Flatten())

    # Final Dense layer with sigmoid activation
    model.add(Dense(1, activation="sigmoid"))

    return model

# Compile discriminator
discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002,
beta_1=0.5), metrics=['accuracy'])
discriminator.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_8 (Conv2D)           (None, 14, 14, 64)        1088

 leaky_re_lu_16 (LeakyReLU)  (None, 14, 14, 64)        0

 conv2d_9 (Conv2D)           (None, 7, 7, 128)         131200

 leaky_re_lu_17 (LeakyReLU)  (None, 7, 7, 128)         0

 flatten_4 (Flatten)         (None, 6272)              0

 dense_8 (Dense)             (None, 1)                 6273

=================================================================
Total params: 138561 (541.25 KB)
Trainable params: 138561 (541.25 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- The discriminator architecture comprises convolutional layers followed by LeakyReLU activation to learn features from input images, aiding in distinguishing between real and fake images.
- The convolutional layers are designed to downsample the input images while extracting relevant features.
- The model then flattens the output tensor to prepare for classification and includes a final dense layer with sigmoid activation for binary classification, determining the authenticity of the input image.
- The discriminator is compiled with binary cross-entropy loss and optimized using the Adam optimizer with specific learning rate and momentum parameters.

Building Generator

It is an unsupervised learning approach. It will generate data that is fake data based on original(real) data. It is also a neural network that has hidden layers, activation, loss function. Its aim is to generate the fake image based on feedback and make the discriminator fool that it cannot predict a fake image. And when the discriminator is made a fool by the generator, the training stops and we can say that a generalized GAN model is created.

```
#The generator uses tf.keras.layers.Conv2DTranspose (upsampling) layers
#to produce an image from a seed (random noise).
def build_generator():
    model = Sequential()
    # Dense layer with 7 * 7 * 128 units, input_dim=100, no activation
    model.add(Dense(7*7*128, input_dim=100))

    # LeakyReLU activation with negative_slope 0.2
    model.add(LeakyReLU(0.2))

    # Reshape output to (7, 7, 128)
    model.add(Reshape((7, 7, 128)))

    # Conv2DTranspose with 64 filters, kernel size 4x4, stride 2, same padding
    model.add(Conv2DTranspose(64, (4,4), strides=2, padding="same"))

    # LeakyReLU activation with negative_slope 0.2
    model.add(LeakyReLU(0.2))

    # Conv2DTranspose with 1 filter, kernel size 4x4, stride 2, same padding, tanh
activation
    model.add(Conv2DTranspose(1, (4,4), strides=2, padding="same",
activation="tanh"))

    return model

# Combine generator and discriminator into a single model
generator = build_generator()

generator.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_9 (Dense)             (None, 6272)              633472

 leaky_re_lu_18 (LeakyReLU)  (None, 6272)              0

 reshape_4 (Reshape)         (None, 7, 7, 128)         0

 conv2d_transpose_8 (Conv2D  (None, 14, 14, 64)        131136
 Transpose)

 leaky_re_lu_19 (LeakyReLU)  (None, 14, 14, 64)        0

 conv2d_transpose_9 (Conv2D  (None, 28, 28, 1)         1025
 Transpose)

=================================================================
Total params: 765633 (2.92 MB)
Trainable params: 765633 (2.92 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- The input dimension, set to 100, determines the size of the input noise vector.
- The generator architecture consists of a densely connected layer followed by LeakyReLU activation, reshaping it to a 3D tensor.
- Subsequently, two Conv2DTranspose layers are added for upsampling, introducing spatial dimensions to the output.
- LeakyReLU activation is applied again for non-linearity, and finally, a Conv2DTranspose layer with a single filter and tanh activation function is utilized to generate the output image.
- The model is designed to produce images with pixel values in the range [-1, 1], hence the choice of tanh activation for the final layer.

Why do we use Leaky RELU?

Leaky relu helps the Gradient flow easily through the neural network architecture.

The ReLU activation function only takes the maximum value between input and zero. If we use ReLU then it is a chance that the network can get stuck in a state known as Dying State. If this happens then it produces nothing other than zero for all outputs. Our aim is to get the value of gradient from Discriminator to make the generator work, and If the network gets stuck then learning will not happen. Leaky ReLU uses a parameter known as alpha to control negative values and never zero passes. If the input is positive then it will exhibit a positive value, and if receive negative then multiply it with alpha and allow some negative value to pass through the network.

Represent Noise Sample

Now we will create input to represent noise samples from latent space. And we pass this noise to a generator to generate an Image. After this, we pass the generator Image to Discriminator and predict that it is Fake or real. In the initial phase, we do not want the discriminator to be trained and the image is Fake.

```
# Input Latent Variable
z = Input(shape=(100,))
# Output of generator
img = generator(z)
# Freeze Discriminator
discriminator.trainable = False
# Give output of Generator to the Discriminator
validity = discriminator(img)

# Build DCGAN

gan = Model(z, validity)
gan.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002,
beta_1=0.5))

gan.summary()
```

```
Model: "model_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 100)]             0

 sequential_9 (Sequential)   (None, 28, 28, 1)         765633

 sequential_8 (Sequential)   (None, 1)                 138561

=================================================================
Total params: 904194 (3.45 MB)
Trainable params: 765633 (2.92 MB)
Non-trainable params: 138561 (541.25 KB)
_________________________________________________________________
```

Train Discriminator and then Generator to generate Images

In this code snippet, we initiate the training loop for a Deep Convolutional Generative Adversarial Network (DCGAN) using Keras.

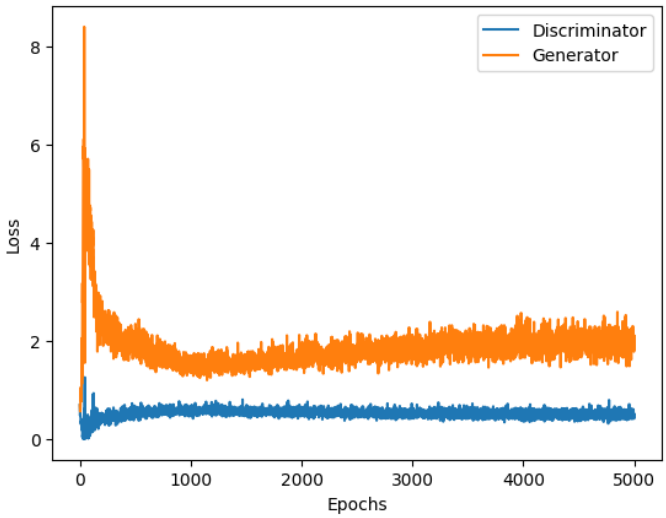
- Over the specified number of epochs (5000 in this case), we iterate through the training data in batches.
- For each epoch, we randomly select a batch of real images from the training dataset and generate an equivalent batch of fake images using random noise as input to the generator.
- Subsequently, we train the discriminator by optimizing its ability to distinguish between real and fake images.
- The discriminator is trained on both batches separately, and the loss is computed and averaged.
- Simultaneously, we train the generator to produce images that the discriminator classifies as real.
- The generator loss is calculated based on the output of the combined GAN model when fed with randomly generated noise.
- The discriminator and generator losses for each epoch are stored for further analysis.
- Additionally, progress is periodically printed to monitor the training process, updating every 1000 epochs, displaying the epoch number along with the discriminator and generator losses.

```
# Train DCGAN
epochs = 5000
batch_size = 64
d_loss_all,g_loss_all=list(),list()
for epoch in range(epochs):
    # Select a random batch of images
    idx = np.random.randint(0, X_train.shape[0], batch_size) #generate a
random number for selecting an image
    real_images = X_train[idx]

    # Generate fake images
    noise = np.random.normal(0, 1, (batch_size, 100)) # generate z
    fake_images = generator.predict(noise) # generator output for input z

    # Train discriminator
    d_loss_real = discriminator.train_on_batch(real_images,
np.ones(batch_size))
    d_loss_fake = discriminator.train_on_batch(fake_images,
np.zeros(batch_size))
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
    d_loss_all.append(d_loss[0])

    # Train generator
    noise = np.random.normal(0, 1, (batch_size, 100))
    g_loss = gan.train_on_batch(noise, np.ones(batch_size))
    g_loss_all.append(g_loss)

    # Print progress
    if epoch % 1000 == 0:
        print(f"Epoch: {epoch} \t Discriminator Loss: {d_loss[0]} \t
Generator Loss: {g_loss}")
```

Plot Loss Function

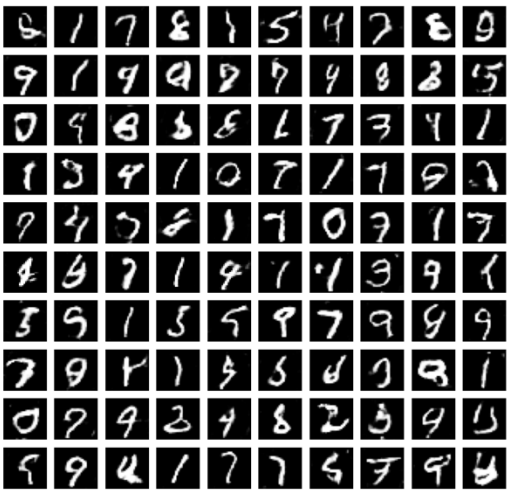
We have finished the training of GAN and let's see what accuracy the Generator is capable of to make Discriminator Fool.

```
plt.plot(d_loss_all)
plt.plot(g_loss_all)
plt.legend(('Discriminator','Generator'))
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```



Generate Images

```
# Generate images
noise = np.random.normal(0, 1, (100, 100))
generated_images = generator.predict(noise)

# Display generated images
plt.figure(figsize=(10, 10))
for i in range(100):
    plt.subplot(10, 10, i+1)
    plt.imshow(generated_images[i, :, :, 0], cmap='gray')
    plt.axis('off')
plt.tight_layout()
plt.show()
```



Conclusion

Generative Adversarial Networks (GANs) represent a powerful paradigm in the field of machine learning, offering diverse applications and functionalities. GANs have demonstrated remarkable capabilities in generating realistic data, enhancing image processing, and facilitating creative applications. Despite their effectiveness, challenges such as mode collapse and training instability persist, necessitating ongoing research efforts. Nevertheless, with proper understanding and implementation, GANs hold immense potential to revolutionize various domains, as exemplified by their practical utilization on the MNIST datasets.