**Parle Tilak Vidyalaya Associations**

# M. L. DAHANUKAR COLLEGE

**Vile-Parle (East), Mumbai – 400 057.**

**Practical Journal**

**DEEP LEARNING**

**Submitted by**

**Trupti Hiru Bhostekar**

**Seat No.:**

**M.Sc. [I.T.]-Information Technology Part II**

**2023 – 2024**

# INDEX

# Practical No 1

## AIM : Performing matrix multiplication and finding eigen vectors and eigen values using TensorFlow

Matrix multiplication and finding eigenvalues and eigenvectors are fundamental operations in many deep learning algorithms. TensorFlow, a powerful library for deep learning, can handle these operations efficiently.

**Code :**

```
# importing numpy library
import numpy as np

# create numpy 2d-array
m = np.array([[1, 2],[2, 3]])

print("Printing the Original square array:\n",m)
print()
print('****************************************')
print()
# finding eigenvalues and eigenvectors
w, v = np.linalg.eig(m)

# printing eigen values
print("Printing the Eigen values of the given square array:\n",w)
print()
# printing eigen vectors
print("Printing Right Eigen Vectors of the given square array:\n",v)
```

**Output :**

```
Printing the Original square array:
 [[1 2]
 [2 3]]

****************************************

Printing the Eigen values of the given square array:
 [-0.23606798  4.23606798]

Printing Right Eigen Vectors of the given square array:
 [[-0.85065081 -0.52573111]
 [ 0.52573111 -0.85065081]]
```

```
# importing numpy library
import numpy as np

# create numpy 2d-array
m = np.array([[1, 2, 3],[2, 3, 4],[4, 5, 6]])

print("Printing the Original square array:\n",m)
print()
print('**************************************')
print()

# finding eigenvalues and eigenvectors
w, v = np.linalg.eig(m)

# printing eigen values
print("Printing the Eigen values of the given square array:\n",w)
print()
# printing eigen vectors
print("Printing Right eigenvectors of the given square array:\n",v)
```

```
Printing the Original square array:
 [[1 2 3]
 [2 3 4]
 [4 5 6]]

**************************************

Printing the Eigen values of the given square array:
 [ 1.08309519e+01 -8.30951895e-01  1.01486082e-16]

Printing Right eigenvectors of the given square array:
 [[ 0.34416959  0.72770285  0.40824829]
 [ 0.49532111  0.27580256 -0.81649658]
 [ 0.79762415 -0.62799801  0.40824829]]
```

```
!pip install tensorflow
!pip install tensorflow[and-cuda]
import tensorflow as tf

# Let's see how we can compute the eigen vectors and values from a matrix
e_matrix_A = tf.random.uniform([2, 2], minval=3, maxval=10, dtype=tf.float32,
name="matrixA")
print("Matrix A: \n{}\n\n".format(e_matrix_A))
```

```
Matrix A:
[[6.218033  5.4523315]
 [4.8386693 8.46125  ]]
```

# Calculating the eigen values and vectors using tf.linalg.eigh, if you only want the values you can use eigvalsh

eigen_values_A, eigen_vectors_A = tf.linalg.eigh(e_matrix_A)
print("Eigen Vectors: \n{} \n\nEigen Values: \n{}\n".format(eigen_vectors_A, eigen_values_A))

```
Eigen Vectors:
[[-0.7828837 -0.6221681]
 [ 0.6221681 -0.7828837]]

Eigen Values:
[ 2.3726776 12.306605 ]
```

# Let's see how we can compute the eigen vectors and values from a matrix

e_matrix_A = tf.random.uniform([3, 3], minval=3, maxval=10, dtype=tf.float32, name="matrixA")
print("Matrix A: \n{}\n\n".format(e_matrix_A))

```
Matrix A:
[[7.8500776 9.986026  9.715893 ]
 [4.430363  9.160987  3.9983697]
 [3.3614936 5.726587  8.88068  ]]
```

# Calculating the eigen values and vectors using tf.linalg.eigh, if you only want the values you can use eigvalsh

eigen_values_A, eigen_vectors_A = tf.linalg.eigh(e_matrix_A)
print("Eigen Vectors: \n{} \n\nEigen Values: \n{}\n".format(eigen_vectors_A, eigen_values_A))

```
Eigen Vectors:
[[-0.29960304 -0.82127017  0.48554415]
 [ 0.76029336  0.10192359  0.6415338 ]
 [-0.57636106  0.56136155  0.5938697 ]]

Eigen Values:
[ 3.0739527  5.002572  17.815218 ]
```

# Practical No 2

## AIM : Solving XOR problem using deep feed forward network

Solving the XOR problem using a deep feed-forward neural network (also known as a multilayer perceptron, or MLP) is a classic example of using neural networks for a non-linearly separable problem. The XOR problem cannot be solved using a single layer of neurons but can be solved using a network with at least one hidden layer.

**Code :**

```
# importing Python library
import numpy as np
# define Unit Step Function
def unitStep(v):
    if v >= 0:
        return 1
    else:
        return 0

# design Perceptron Model
def perceptronModel(x, w, b):
    v = np.dot(w, x) + b
    y = unitStep(v)
    return y

# NOT Logic Function
# wNOT = -1, bNOT = 0.5
def NOT_logicFunction(x):
    wNOT = -1
    bNOT = 0.5
    return perceptronModel(x, wNOT, bNOT)

# AND Logic Function
# here w1 = wAND1 = 1,
# w2 = wAND2 = 1, bAND = -1.5

def AND_logicFunction(x):
```

```
    w = np.array([1, 1])
    bAND = -1.5
    return perceptronModel(x, w, bAND)

# OR Logic Function
# w1 = 1, w2 = 1, bOR = -0.5
def OR_logicFunction(x):
    w = np.array([1, 1])
    bOR = -0.5
    return perceptronModel(x, w, bOR)

# XOR Logic Function
# with AND, OR and NOT
# function calls in sequence
def XOR_logicFunction(x):
  y1 = AND_logicFunction(x)
  y2 = OR_logicFunction(x)
  y3 = NOT_logicFunction(y1)
  final_x = np.array([y2, y3])
  finalOutput = AND_logicFunction(final_x)
  y3 = NOT_logicFunction(y1)
  return finalOutput

# testing the Perceptron Model
test1 = np.array([0, 1])
test2 = np.array([1, 1])
test3 = np.array([0, 0])
test4 = np.array([1, 0])

print("XOR({}, {}) = {}".format(0, 1, XOR_logicFunction(test1)))
print("XOR({}, {}) = {}".format(1, 1, XOR_logicFunction(test2)))
print("XOR({}, {}) = {}".format(0, 0, XOR_logicFunction(test3)))
print("XOR({}, {}) = {}".format(1, 0, XOR_logicFunction(test4)))
```

**Output :**

```
XOR(0, 1) = 1
XOR(1, 1) = 0
XOR(0, 0) = 0
XOR(1, 0) = 1
```

# Practical No 3

## AIM : Implementing deep neural network for performing binary classification task.

The dataset we will use in this is the Sonar dataset.
This is a dataset that describes sonar chirp returns bouncing off different services.
The 60 input variables are the strength of the returns at different angles.
It is a binary classification problem that requires a model to differentiate rocks from metal cylinders.
It is a well-understood dataset.
All of the variables are continuous and generally in the range of 0 to 1.
The output variable is a string "M" for mine and "R" for rock, which will need to be converted to integers 1 and 0.
A benefit of using this dataset is that it is a standard benchmark problem.
This means that we have some idea of the expected skill of a good model.
Using cross-validation, a neural network should be able to achieve performance around 84% with an upper bound on accuracy for custom models at around 88%.

```
!pip uninstall tensorflow
!pip install tensorflow==2.12.0
import pandas as pd
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# load dataset
dataframe = pd.read_csv("sonar.all-data", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
```

```
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)

# baseline model
def create_baseline():
    # create model
    model = Sequential()
    model.add(Dense(60, input_dim=60, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

# evaluate model with standardized dataset
estimator = KerasClassifier(build_fn=create_baseline, epochs=100, batch_size=5, verbose=0)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(estimator, X, encoded_Y, cv=kfold)
print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

```
<ipython-input-9-c05b681d3cca>:3: DeprecationWarning: KerasClassifier is deprecated, use Sci-Keras (https://github.com/adriangb/scikeras) in
  estimator = KerasClassifier(build_fn=create_baseline, epochs=100, batch_size=5, verbose=0)
Baseline: 81.83% (11.76%)
```

```
# evaluate baseline model with standardized dataset
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_baseline, epochs=100, batch_size=5,
verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Standardized: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

```
<ipython-input-10-546fd92f22de>:4: DeprecationWarning: KerasClassifier is deprecated, use Sci-Keras (https://github.com/adriangb/scikeras) in
  estimators.append(('mlp', KerasClassifier(build_fn=create_baseline, epochs=100, batch_size=5, verbose=0)))
Standardized: 87.45% (5.46%)
```

```
# smaller model
def create_smaller():
    # create model
    model = Sequential()
```

```
    model.add(Dense(30, input_dim=60, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_smaller, epochs=100, batch_size=5,
verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Smaller: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

```
<ipython-input-11-2aa53080dfa6>:13: DeprecationWarning: KerasClassifier is deprecated, use Sci-Keras (https://github.com/adriangb/scikeras) inst
  estimators.append(('mlp', KerasClassifier(build_fn=create_smaller, epochs=100, batch_size=5, verbose=0)))
Smaller: 85.60% (9.17%)
```

```
# larger model
def create_larger():
    # create model
    model = Sequential()
    model.add(Dense(60, input_dim=60, activation='relu'))
    model.add(Dense(30, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_larger, epochs=100, batch_size=5,
verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Larger: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

```
<ipython-input-12-8b122404de97>:13: DeprecationWarning: KerasClassifier is deprecated, use Sci-Keras (https://github.com/adriangb/scikeras) ins
  estimators.append(('mlp', KerasClassifier(build_fn=create_larger, epochs=100, batch_size=5, verbose=0)))
Larger: 84.21% (6.67%)
```

# Practical No 4

## A] AIM : Using deep feed forward network with two hidden layers for performing classification and predicting the class

To build and use a deep feed-forward neural network for performing classification and predicting the class in a general deep learning problem, we'll use TensorFlow and Keras. Let's take a typical classification dataset, such as the Iris dataset, and demonstrate how to build, train, evaluate, and make predictions with a deep feed-forward neural network.

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data
3. Build the Model
4. Compile the Model
5. Train the Model
6. Evaluate the Model
7. Make Predictions

**Code :**

```
from keras.models import Sequential
from keras.layers import Dense
from sklearn.datasets import make_blobs
from sklearn.preprocessing import MinMaxScaler

X,Y=make_blobs(n_samples=100,centers=2,n_features=2,random_state=1)
scalar=MinMaxScaler()
scalar.fit(X)
X=scalar.transform(X)

model=Sequential()
model.add(Dense(4,input_dim=2,activation='relu'))
model.add(Dense(4,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam')
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 4)                 12

 dense_1 (Dense)             (None, 4)                 20

 dense_2 (Dense)             (None, 1)                 5

=================================================================
Total params: 37 (148.00 Byte)
Trainable params: 37 (148.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
model.fit(X,Y,epochs=100)   # u can use 150 epochs also…
```

```
4/4 [==============================] - 0s 3ms/step - loss: 0.2276
Epoch 96/100
4/4 [==============================] - 0s 3ms/step - loss: 0.2241
Epoch 97/100
4/4 [==============================] - 0s 3ms/step - loss: 0.2207
Epoch 98/100
4/4 [==============================] - 0s 3ms/step - loss: 0.2175
Epoch 99/100
4/4 [==============================] - 0s 5ms/step - loss: 0.2141
Epoch 100/100
4/4 [==============================] - 0s 3ms/step - loss: 0.2108
<keras.src.callbacks.History at 0x7ec2e4503ac0>
```

```
Xnew,Yreal=make_blobs(n_samples=3,centers=2,n_features=2,random_state=1)
Xnew=scalar.transform(Xnew)
Ynew=model.predict(Xnew)
for i in range(len(Xnew)):
 print("X=%s,Predicted=%s,Desired=%s"%(Xnew[i],Ynew[i],Yreal[i]))
```

**Output :**

```
1/1 [==============================] - 0s 88ms/step
X=[0.89337759 0.65864154],Predicted=[0.05559724],Desired=0
X=[0.29097707 0.12978982],Predicted=[0.61116236],Desired=1
X=[0.78082614 0.75391697],Predicted=[0.09069698],Desired=0
```

## B] AIM : Using deep feed forward network with two hidden layers for performing classification and predicting the probability of class

To perform classification and predict the probability of each class using a deep feed-forward network with two hidden layers, you can follow a similar approach as outlined previously. We will use the Iris dataset as an example and modify the steps to include predicting the probability of each class.

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data

3. Build the Model
4. Compile the Model
5. Train the Model
6. Evaluate the Model
7. Make Predictions and Predict Probability of Each Class

**Code :**

```
from keras.models import Sequential
from keras.layers import Dense
from sklearn.datasets import make_blobs
from sklearn.preprocessing import MinMaxScaler

X,Y=make_blobs(n_samples=100,centers=2,n_features=2,random_state=1)
scalar=MinMaxScaler()
scalar.fit(X)
X=scalar.transform(X)
model=Sequential()
model.add(Dense(4,input_dim=2,activation='relu'))
model.add(Dense(4,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam')
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 4)                 12

 dense_1 (Dense)             (None, 4)                 20

 dense_2 (Dense)             (None, 1)                 5

=================================================================
Total params: 37 (148.00 Byte)
Trainable params: 37 (148.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
model.fit(X,Y,epochs=200)
```

```
Epoch 197/200
4/4 [==============================] - 0s 6ms/step - loss: 0.0742
Epoch 198/200
4/4 [==============================] - 0s 6ms/step - loss: 0.0731
Epoch 199/200
4/4 [==============================] - 0s 7ms/step - loss: 0.0721
Epoch 200/200
4/4 [==============================] - 0s 5ms/step - loss: 0.0710
<keras.src.callbacks.History at 0x7f960c511e70>
```

```
Xnew,Yreal=make_blobs(n_samples=3,centers=2,n_features=2,random_state=1)
Xnew=scalar.transform(Xnew)
Yclass=model.predict(Xnew)

import numpy as np
def predict_prob(number):
  return [number[0],1-number[0]]

y_prob = np.array(list(map(predict_prob, model.predict(Xnew))))
y_prob
```

```
1/1 [==============================] - 0s 119ms/step
array([[0.1264445 , 0.8735555 ],
       [0.91967714, 0.08032286],
       [0.05721965, 0.94278035]])
```

```
for i in range(len(Xnew)):
 print("X=%s,Predicted_probability=%s,Predicted_class=%s"%(Xnew[i],y_prob[i],Yclass[i]))
```

```
X=[0.89337759 0.65864154],Predicted_probability=[0.1264445 0.8735555],Predicted_class=[0.1264445]
X=[0.29097707 0.12978982],Predicted_probability=[0.91967714 0.08032286],Predicted_class=[0.91967714]
X=[0.78082614 0.75391697],Predicted_probability=[0.05721965 0.94278035],Predicted_class=[0.05721965]
```

```
#second way
predict_prob=model.predict([Xnew])
predict_classes=np.argmax(predict_prob,axis=1)
predict_classes
```

```
1/1 [============================] - 0s 252ms/step
array([0, 0, 0])
```

# Practical No 5

## AIM : Evaluating feed forward deep network for regression using KFold cross validation

Evaluating a feed-forward deep network for regression using KFold cross-validation is a common approach to ensure that your model performs well across different subsets of the data. Here's how you can do this using TensorFlow and Keras.

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data
3. Define the Model
4. Evaluate Using KFold Cross-Validation

**Code :**

```
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt

# Model configuration
batch_size = 50
img_width, img_height, img_num_channels = 32, 32, 3
loss_function = sparse_categorical_crossentropy
no_classes = 100
no_epochs = 10   # you can increase it to 20,50,70, 100
optimizer = Adam()
verbosity = 1

# Load CIFAR-10 data
(input_train, target_train), (input_test, target_test) = cifar10.load_data()

# Determine shape of the data
input_shape = (img_width, img_height, img_num_channels)
```

```
# Parse numbers as floats
input_train = input_train.astype('float32')
input_test = input_test.astype('float32')

# Normalize data
input_train = input_train / 255
input_test = input_test / 255

# Create the model
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(no_classes, activation='softmax'))
model.summary()
```

Model: "sequential"

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d (Conv2D) | (None, 30, 30, 32) | 896 |
| max_pooling2d (MaxPooling2D) | (None, 15, 15, 32) | 0 |
| conv2d_1 (Conv2D) | (None, 13, 13, 64) | 18,496 |
| max_pooling2d_1 (MaxPooling2D) | (None, 6, 6, 64) | 0 |
| flatten (Flatten) | (None, 2304) | 0 |
| dense (Dense) | (None, 256) | 590,080 |
| dense_1 (Dense) | (None, 128) | 32,896 |
| dense_2 (Dense) | (None, 100) | 12,900 |

**Total params:** 655,268 (2.50 MB)
**Trainable params:** 655,268 (2.50 MB)
**Non-trainable params:** 0 (0.00 B)

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

# Fit data to model (this will take little time to train)
history = model.fit(input_train, target_train, batch_size=batch_size, epochs=no_epochs,
verbose=verbosity)
```

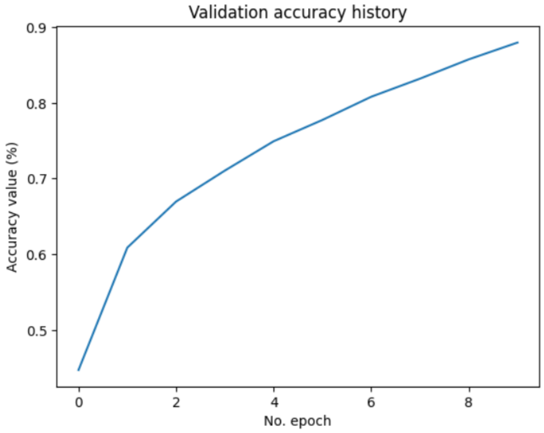
```
Epoch 1/10
1000/1000 [==============================] - 64s 63ms/step - loss: 1.5346 - accuracy: 0.4471
Epoch 2/10
1000/1000 [==============================] - 61s 61ms/step - loss: 1.1138 - accuracy: 0.6087
Epoch 3/10
1000/1000 [==============================] - 60s 60ms/step - loss: 0.9490 - accuracy: 0.6696
Epoch 4/10
1000/1000 [==============================] - 64s 64ms/step - loss: 0.8255 - accuracy: 0.7105
Epoch 5/10
1000/1000 [==============================] - 61s 61ms/step - loss: 0.7216 - accuracy: 0.7493
Epoch 6/10
1000/1000 [==============================] - 61s 61ms/step - loss: 0.6338 - accuracy: 0.7774
Epoch 7/10
1000/1000 [==============================] - 63s 63ms/step - loss: 0.5494 - accuracy: 0.8080
Epoch 8/10
1000/1000 [==============================] - 61s 61ms/step - loss: 0.4747 - accuracy: 0.8320
Epoch 9/10
1000/1000 [==============================] - 59s 59ms/step - loss: 0.4006 - accuracy: 0.8574
Epoch 10/10
1000/1000 [==============================] - 63s 63ms/step - loss: 0.3408 - accuracy: 0.8797
```

```
# Generate generalization metrics
score = model.evaluate(input_test, target_test, verbose=0)
print(f'Test loss: {score[0]} / Test accuracy: {score[1]}')
```

```
Test loss: 1.1294307708740234 / Test accuracy: 0.6931999921798706
```

```
# Visualize history
# Plot history: Loss
plt.plot(history.history['loss'])
plt.title('Validation loss history')
plt.ylabel('Loss value')
plt.xlabel('No. epoch')
plt.show()
```


```
# Plot history: Accuracy
plt.plot(history.history['accuracy'])
plt.title('Validation accuracy history')
plt.ylabel('Accuracy value (%)')
plt.xlabel('No. epoch')
plt.show()
```




```
# By Adding k fold cross validation
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import KFold
import numpy as np

# Model configuration
batch_size = 50
img_width, img_height, img_num_channels = 32, 32, 3
loss_function = sparse_categorical_crossentropy
no_classes = 100
no_epochs = 10
optimizer = Adam()
verbosity = 1
num_folds = 5

# Load CIFAR-10 data
(input_train, target_train), (input_test, target_test) = cifar10.load_data()

# Determine shape of the data
input_shape = (img_width, img_height, img_num_channels)

# Parse numbers as floats
```

```
input_train = input_train.astype('float32')
input_test = input_test.astype('float32')

# Normalize data
input_train = input_train / 255
input_test = input_test / 255

# Define per-fold score containers
acc_per_fold = []
loss_per_fold = []

# Merge inputs and targets
inputs = np.concatenate((input_train, input_test), axis=0)
targets = np.concatenate((target_train, target_test), axis=0)

# Define the K-fold Cross Validator
kfold = KFold(n_splits=num_folds, shuffle=True)

import tensorflow as tf
from tensorflow.keras.optimizers.legacy import SGD

tf.keras.optimizers.legacy.SGD(learning_rate=0.1)

# K-fold Cross Validation model evaluation
fold_no = 1
for train, test in kfold.split(inputs, targets):

  # Define the model architecture
  model = Sequential()
  model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Flatten())
  model.add(Dense(256, activation='relu'))
  model.add(Dense(128, activation='relu'))
  model.add(Dense(no_classes, activation='softmax'))
```

```
  optimizer = tf.keras.optimizers.legacy.Adam()

  # Compile the model
  model.compile(loss=loss_function,
                optimizer=optimizer,
                metrics=['accuracy'])

  # Generate a print
  print('------------------------------------------------------------------------')
  print(f'Training for fold {fold_no} ...')

  # Fit data to model
  history = model.fit(inputs[train], targets[train],
              batch_size=batch_size,
              epochs=no_epochs,
              verbose=verbosity)

  # Generate generalization metrics
  scores = model.evaluate(inputs[test], targets[test], verbose=0)
  print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]};
{model.metrics_names[1]} of {scores[1]*100}%')
  acc_per_fold.append(scores[1] * 100)
  loss_per_fold.append(scores[0])

  # Increase fold number
  fold_no = fold_no + 1
```

```
------------------------------------------------------------------------
Training for fold 1 ...
Epoch 1/10
960/960 [==============================] - 6s 4ms/step - loss: 1.5478 - accuracy: 0.4444
Epoch 2/10
960/960 [==============================] - 4s 4ms/step - loss: 1.1277 - accuracy: 0.5996
Epoch 3/10
960/960 [==============================] - 5s 5ms/step - loss: 0.9678 - accuracy: 0.6602
Epoch 4/10
960/960 [==============================] - 4s 4ms/step - loss: 0.8511 - accuracy: 0.7040
Epoch 5/10
960/960 [==============================] - 4s 4ms/step - loss: 0.7563 - accuracy: 0.7365
Epoch 6/10
960/960 [==============================] - 5s 5ms/step - loss: 0.6660 - accuracy: 0.7679
Epoch 7/10
960/960 [==============================] - 4s 4ms/step - loss: 0.5752 - accuracy: 0.7983
Epoch 8/10
960/960 [==============================] - 4s 4ms/step - loss: 0.5016 - accuracy: 0.8253
Epoch 9/10
960/960 [==============================] - 4s 5ms/step - loss: 0.4223 - accuracy: 0.8527
Epoch 10/10
960/960 [==============================] - 4s 4ms/step - loss: 0.3531 - accuracy: 0.8762
Score for fold 1: loss of 1.118696689605713; accuracy of 69.40000057220459%
```

```
# == Provide average scores ==
print('------------------------------------------------------------------------')
print('Score per fold')
for i in range(0, len(acc_per_fold)):
  print('------------------------------------------------------------------------')
  print(f'> Fold {i+1} - Loss: {loss_per_fold[i]} - Accuracy: {acc_per_fold[i]}%')
print('------------------------------------------------------------------------')
print('Average scores for all folds:')
print(f'> Accuracy: {np.mean(acc_per_fold)} (+- {np.std(acc_per_fold)})')
print(f'> Loss: {np.mean(loss_per_fold)}')
print('------------------------------------------------------------------------')
```

```
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 1.118696689605713 - Accuracy: 69.40000057220459%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 69.40000057220459 (+- 0.0)
> Loss: 1.118696689605713
------------------------------------------------------------------------
```

# Practical 6

## AIM : Implementing regularization to avoid overfitting in binary classification using TensorFlow

Implementing regularization is crucial for avoiding overfitting in deep learning models. For binary classification, common regularization techniques include L2 regularization (also known as weight decay) and dropout.

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data
3. Build the Model with Regularization
4. Compile the Model
5. Train the Model
6. Evaluate the Model
7. Make Predictions

**Code :**

```
from matplotlib import pyplot
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense

X,Y=make_moons(n_samples=100,noise=0.2,random_state=1)
n_train=30
trainX,testX=X[:n_train,:],X[n_train:]
trainY,testY=Y[:n_train],Y[n_train:]

print(trainX.shape)
print(trainY.shape)
print(testX.shape)
print(testY.shape)
```

```
(30, 2)
(30,)
(70, 2)
(70,)
```

```
model=Sequential()
```

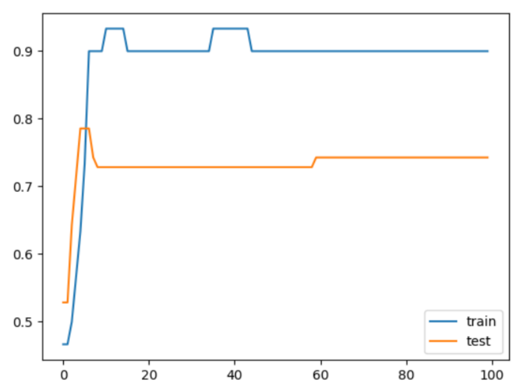
```
model.add(Dense(500,input_dim=2,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 500)               1500

 dense_1 (Dense)             (None, 1)                 501

=================================================================
Total params: 2001 (7.82 KB)
Trainable params: 2001 (7.82 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
history=model.fit(trainX,trainY,validation_data=(testX,testY),epochs=100)
```

```
1/1 [==============================] - 0s 295ms/step - loss: 0.3736 - accuracy: 0.9000 - val_loss: 0.4888 - val_accuracy: 0.7286
Epoch 31/100
1/1 [==============================] - 0s 134ms/step - loss: 0.3662 - accuracy: 0.9000 - val_loss: 0.4854 - val_accuracy: 0.7286
Epoch 32/100
1/1 [==============================] - 0s 145ms/step - loss: 0.3590 - accuracy: 0.9000 - val_loss: 0.4821 - val_accuracy: 0.7286
Epoch 33/100
1/1 [==============================] - 0s 157ms/step - loss: 0.3520 - accuracy: 0.9000 - val_loss: 0.4790 - val_accuracy: 0.7286
Epoch 34/100
1/1 [==============================] - 0s 210ms/step - loss: 0.3453 - accuracy: 0.9000 - val_loss: 0.4761 - val_accuracy: 0.7286
Epoch 35/100
1/1 [==============================] - 0s 91ms/step - loss: 0.3388 - accuracy: 0.9000 - val_loss: 0.4734 - val_accuracy: 0.7286
Epoch 36/100
```

```
pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()
```




# After 75 epochs it started overfitting by giving same validation accuracy on the test data, so let us use regularization technique

```
from keras.regularizers import l2
```

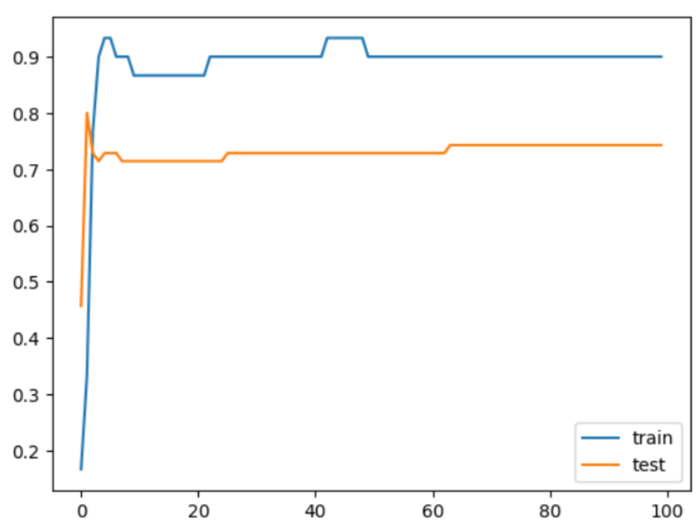
```
model=Sequential()
model.add(Dense(500,input_dim=2,activation='relu',kernel_regularizer=l2(0.001)))
model.add(Dense(1,activation='sigmoid'))
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_2 (Dense)             (None, 500)               1500

 dense_3 (Dense)             (None, 1)                 501

=================================================================
Total params: 2001 (7.82 KB)
Trainable params: 2001 (7.82 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
history=model.fit(trainX,trainY,validation_data=(testX,testY),epochs=100)
```

```
Epoch 72/100
1/1 [==============================] - 0s 40ms/step - loss: 0.2267 - accuracy: 0.9000 - val_loss: 0.4523 - val_accuracy: 0.7429
Epoch 73/100
1/1 [==============================] - 0s 45ms/step - loss: 0.2253 - accuracy: 0.9000 - val_loss: 0.4518 - val_accuracy: 0.7429
Epoch 74/100
1/1 [==============================] - 0s 45ms/step - loss: 0.2240 - accuracy: 0.9000 - val_loss: 0.4512 - val_accuracy: 0.7429
Epoch 75/100
1/1 [==============================] - 0s 43ms/step - loss: 0.2227 - accuracy: 0.9000 - val_loss: 0.4507 - val_accuracy: 0.7429
```

```
pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()
```



Lets apply l1 and l2 together to the model using below code

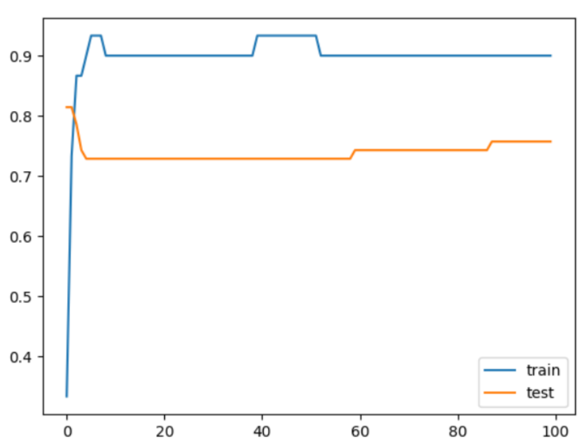
```
from keras.regularizers import l1_l2
model=Sequential()
model.add(Dense(500,input_dim=2,activation='relu',kernel_regularizer=l1_l2(l1=0.001,l2=0.001
)))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_4 (Dense)             (None, 500)               1500

 dense_5 (Dense)             (None, 1)                 501

=================================================================
Total params: 2001 (7.82 KB)
Trainable params: 2001 (7.82 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
history=model.fit(trainX,trainY,validation_data=(testX,testY),epochs=100)
```

```
Epoch 72/100
1/1 [==============================] - 0s 54ms/step - loss: 0.2829 - accuracy: 0.9000 - val_loss: 0.4936 - val_accuracy: 0.7429
Epoch 73/100
1/1 [==============================] - 0s 68ms/step - loss: 0.2815 - accuracy: 0.9000 - val_loss: 0.4926 - val_accuracy: 0.7429
Epoch 74/100
1/1 [==============================] - 0s 68ms/step - loss: 0.2801 - accuracy: 0.9000 - val_loss: 0.4916 - val_accuracy: 0.7429
Epoch 75/100
```

```
pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()
```



# Practical 7

## AIM : Implementing Text classification with an RNN

Implementing text classification with a Recurrent Neural Network (RNN) is a common task in natural language processing (NLP).

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data
3. Build the RNN Model
4. Compile the Model
5. Train the Model
6. Evaluate the Model
7. Make Predictions

```
import numpy as np
import tensorflow_datasets as tfds
import tensorflow as tf
tfds.disable_progress_bar()

import matplotlib.pyplot as plt

def plot_graphs(history, metric):
  plt.plot(history.history[metric])
  plt.plot(history.history['val_'+metric], '')
  plt.xlabel("Epochs")
  plt.ylabel(metric)
  plt.legend([metric, 'val_'+metric])

dataset, info = tfds.load('imdb_reviews', with_info=True,
                          as_supervised=True)
train_dataset, test_dataset = dataset['train'], dataset['test']
train_dataset.element_spec

for example, label in train_dataset.take(5):
  print('text: ', example.numpy())
  print('label: ', label.numpy())
```

```
text:  b"This was an absolutely terrible movie. Don't be lured in by Christoph
label:  0
text:  b'I have been known to fall asleep during films, but this is usually du
label:  0
text:  b'Mann photographs the Alberta Rocky Mountains in a superb fashion, and
label:  0
text:  b'This is the kind of film for a snowy Sunday afternoon when the rest o
label:  1
text:  b'As others have mentioned, all the women that go nude in this film are
label:  1
```

```
BUFFER_SIZE = 10000
BATCH_SIZE = 64

train_dataset =
train_dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
test_dataset = test_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)

for example, label in train_dataset.take(1):
  print('texts: ', example.numpy()[:3])
  print()
  print('labels: ', label.numpy()[:3])
```

```
texts:  [b"I expected alot from this movie. Kinda like Lee as a Naustradamous like caract
 b'I can\'t say whether the post-WWII British comedies produced at the Ealing Studios are
 b'Although I love this movie, I can barely watch it, it is so real. So, I put it on toni

labels:  [0 1 1]
```

Create the text encoder

```
VOCAB_SIZE = 1000
encoder = tf.keras.layers.TextVectorization(max_tokens=VOCAB_SIZE)
encoder.adapt(train_dataset.map(lambda text, label: text))

vocab = np.array(encoder.get_vocabulary())
vocab[:20]
```

```
array(['', '[UNK]', 'the', 'and', 'a', 'of', 'to', 'is', 'in', 'it', 'i',
       'this', 'that', 'br', 'was', 'as', 'for', 'with', 'movie', 'but'],
      dtype='<U14')
```

```
encoded_example = encoder(example)[:3].numpy()
encoded_example
```

```
array([[ 10, 855,   1, ...,   0,   0,   0],
       [ 10, 175, 130, ...,   0,   0,   0],
       [255,  10, 116, ...,   0,   0,   0]])
```

```
for n in range(3):
  print("Original: ", example[n].numpy())
  print("Round-trip: ", " ".join(vocab[encoded_example[n]]))
  print()
```

```
Original:  b"I expected alot from this movie. Kinda like Lee as a Naus
Round-trip:  i expected [UNK] from this movie [UNK] like lee as a [UNK

Original:  b'I can\'t say whether the post-WWII British comedies produ
Round-trip:  i cant say whether the [UNK] british [UNK] [UNK] at the [

Original:  b'Although I love this movie, I can barely watch it, it is
Round-trip:  although i love this movie i can [UNK] watch it it is so
```

Create the model

```
model = tf.keras.Sequential([
    encoder,
    tf.keras.layers.Embedding(
        input_dim=len(encoder.get_vocabulary()),
        output_dim=64,
        # Use masking to handle the variable sequence lengths
        mask_zero=True),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1)
])
```

```
print([layer.supports_masking for layer in model.layers])
```

```
[False, True, True, True, True]
```

# predict on a sample text without padding.

```
sample_text = ('The movie was cool. The animation and the graphics '
            'were out of this world. I would recommend this movie.')
predictions = model.predict(np.array([sample_text]))
print(predictions[0])
```

```
1/1 [==============================] - 4s 4s/step
[-0.00135332]
```

```
# predict on a sample text with padding

padding = "the " * 2000
predictions = model.predict(np.array([sample_text, padding]))
print(predictions[0])
```

```
1/1 [==============================] - 0s 84ms/step
[-0.00135332]
```

```
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
          optimizer=tf.keras.optimizers.Adam(1e-4),
          metrics=['accuracy'])
```

Train the model

```
history = model.fit(train_dataset, epochs=10,
               validation_data=test_dataset,
               validation_steps=30)
```

```
Epoch 1/10
391/391 [==============================] - 59s 125ms/step - loss: 0.64
Epoch 2/10
391/391 [==============================] - 28s 70ms/step - loss: 0.447
Epoch 3/10
391/391 [==============================] - 27s 69ms/step - loss: 0.369
Epoch 4/10
391/391 [==============================] - 25s 64ms/step - loss: 0.340
Epoch 5/10
391/391 [==============================] - 26s 65ms/step - loss: 0.324
Epoch 6/10
391/391 [==============================] - 25s 64ms/step - loss: 0.313
Epoch 7/10
391/391 [==============================] - 26s 65ms/step - loss: 0.306
Epoch 8/10
391/391 [==============================] - 25s 64ms/step - loss: 0.303
Epoch 9/10
391/391 [==============================] - 26s 65ms/step - loss: 0.300
Epoch 10/10
391/391 [==============================] - 26s 66ms/step - loss: 0.299
```

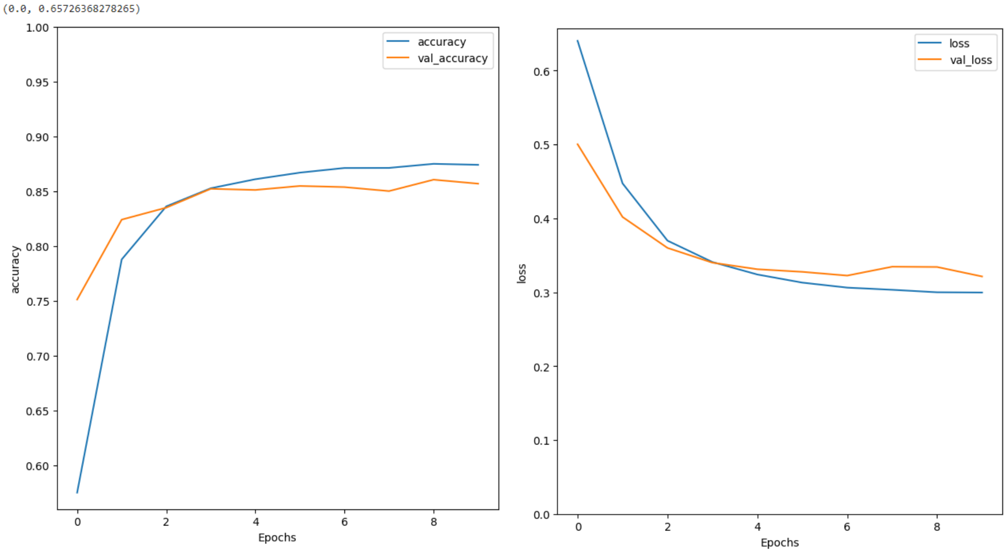
```
test_loss, test_acc = model.evaluate(test_dataset)
```

```
print('Test Loss:', test_loss)
print('Test Accuracy:', test_acc)
```

```
391/391 [==============================] - 12s 30ms/step - loss: 0.3155 - accuracy: 0.8617
Test Loss: 0.31554868817329407
Test Accuracy: 0.8616799712181091
```

```
plt.figure(figsize=(16, 8))
plt.subplot(1, 2, 1)
plot_graphs(history, 'accuracy')
plt.ylim(None, 1)
plt.subplot(1, 2, 2)
plot_graphs(history, 'loss')
plt.ylim(0, None)
```

```
(0.0, 0.65726368278265)
```




```
sample_text = ('The movie was cool. The animation and the graphics '
               'were out of this world. I would recommend this movie.')
predictions = model.predict(np.array([sample_text]))
```

```
1/1 [==============================] - 2s 2s/step
```

```
Predictions
```

```
array([[0.9359414]], dtype=float32)
```

```
model = tf.keras.Sequential([
    encoder,
    tf.keras.layers.Embedding(len(encoder.get_vocabulary()), 64, mask_zero=True),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64,  return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1)
])

model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(1e-4),
              metrics=['accuracy'])

history = model.fit(train_dataset, epochs=10,
                    validation_data=test_dataset,
                    validation_steps=30)
```

```
Epoch 1/10
391/391 [==============================] - 77s 151ms/step - los
Epoch 2/10
391/391 [==============================] - 47s 119ms/step - los
Epoch 3/10
391/391 [==============================] - 50s 127ms/step - los
Epoch 4/10
391/391 [==============================] - 47s 120ms/step - los
Epoch 5/10
391/391 [==============================] - 47s 120ms/step - los
Epoch 6/10
391/391 [==============================] - 47s 120ms/step - los
Epoch 7/10
391/391 [==============================] - 48s 122ms/step - los
Epoch 8/10
391/391 [==============================] - 47s 119ms/step - los
Epoch 9/10
391/391 [==============================] - 46s 118ms/step - los
Epoch 10/10
391/391 [==============================] - 47s 120ms/step - los
```

```
test_loss, test_acc = model.evaluate(test_dataset)

print('Test Loss:', test_loss)
print('Test Accuracy:', test_acc)
```

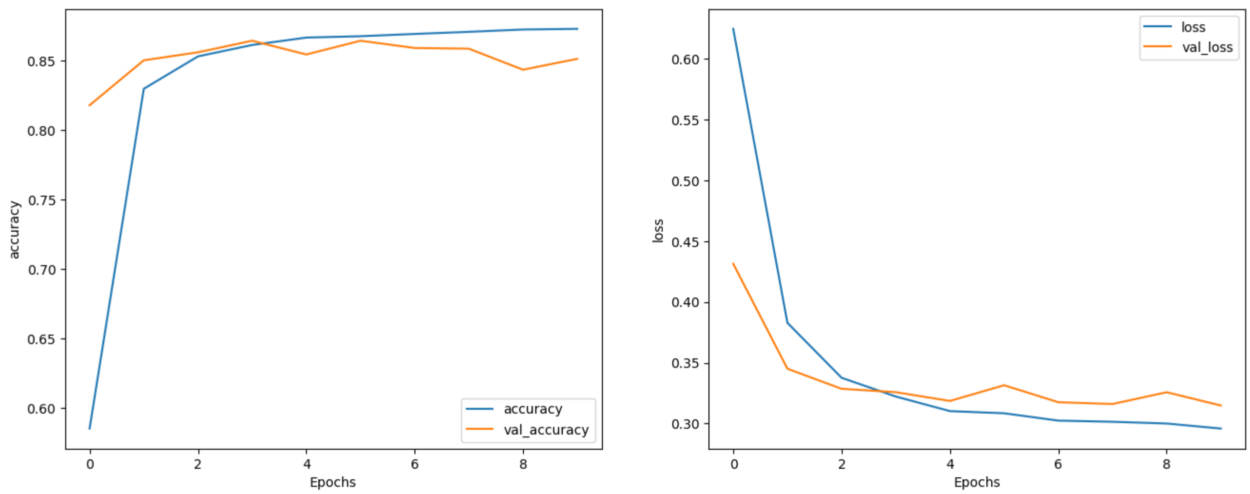
```
391/391 [==============================] - 20s 52ms/step - loss: 0.3130 - accuracy: 0.8564
Test Loss: 0.31295326352119446
Test Accuracy: 0.856440007686615
```

```
# predict on a sample text without padding.

sample_text = ('The movie was not good. The animation and the graphics '
               'were terrible. I would not recommend this movie.')
```

```
predictions = model.predict(np.array([sample_text]))
print(predictions)
```

```
1/1 [==============================] - 4s 4s/step
[[-1.9441454]]
```

```
plt.figure(figsize=(16, 6))
plt.subplot(1, 2, 1)
plot_graphs(history, 'accuracy')
plt.subplot(1, 2, 2)
plot_graphs(history, 'loss')
```



# Practical No 8

## AIM : Implementation of Autoencoders

Autoencoders are a type of neural network used for unsupervised learning. They are designed to encode input data into a lower-dimensional representation and then reconstruct the data from this representation. This can be useful for tasks like dimensionality reduction, feature learning, and data denoising.

Here's a step-by-step implementation of a simple autoencoder using TensorFlow and Keras.

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data
3. Build the Autoencoder Model
4. Compile the Model
5. Train the Model
6. Evaluate the Model
7. Visualize the Results

**Code :**

```
import keras
from keras import layers
# This is the size of our encoded representations
encoding_dim = 32  # 32 floats -> compression of factor 24.5, assuming the input is 784 floats

# This is our input image
input_img = keras.Input(shape=(784,))
# "encoded" is the encoded representation of the input
encoded = layers.Dense(encoding_dim, activation='relu')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded = layers.Dense(784, activation='sigmoid')(encoded)

# This model maps an input to its reconstruction
autoencoder = keras.Model(input_img, decoded)

#Let's also create a separate encoder model:
```

```
# This model maps an input to its encoded representation
encoder = keras.Model(input_img, encoded)

# This is our encoded (32-dimensional) input
encoded_input = keras.Input(shape=(encoding_dim,))
# Retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# Create the decoder model
decoder = keras.Model(encoded_input, decoder_layer(encoded_input))

#Now let's train our autoencoder to reconstruct MNIST digits.
#First, we'll configure our model to use a per-pixel binary crossentropy loss, and the Adam optimizer:

autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

#Let's prepare our input data. We're using MNIST digits, and we're discarding the labels (since we're only interested in encoding/decoding the input images).

from keras.datasets import mnist
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

```
# We will normalize all values between 0 and 1 and we will flatten the 28x28 images into vectors of size 784.

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print(x_train.shape)
print(x_test.shape)
```

```
(60000, 784)
(10000, 784)
```

```
# Now let's train our autoencoder for 50 epochs:
```

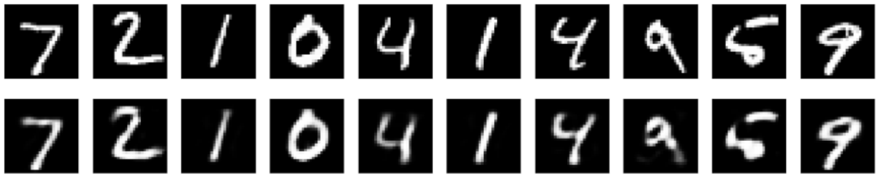
```
autoencoder.fit(x_train, x_train,
            epochs=50,
            batch_size=256,
            shuffle=True,
            validation_data=(x_test, x_test))
```

```
Epoch 1/50
235/235 [==============================] - 3s 11ms/step - loss: 0.2792 - val_loss: 0.1941
Epoch 2/50
235/235 [==============================] - 3s 13ms/step - loss: 0.1733 - val_loss: 0.1546
Epoch 3/50
235/235 [==============================] - 4s 19ms/step - loss: 0.1446 - val_loss: 0.1340
```

```
# Encode and decode some digits
# Note that we take them from the *test* set
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
```

```
313/313 [==============================] - 1s 1ms/step
313/313 [==============================] - 1s 1ms/step
```

```
# Use Matplotlib
import matplotlib.pyplot as plt

n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
   # Display original
   ax = plt.subplot(2, n, i + 1)
   plt.imshow(x_test[i].reshape(28, 28))
   plt.gray()
   ax.get_xaxis().set_visible(False)
   ax.get_yaxis().set_visible(False)

   # Display reconstruction
   ax = plt.subplot(2, n, i + 1 + n)
   plt.imshow(decoded_imgs[i].reshape(28, 28))
   plt.gray()
   ax.get_xaxis().set_visible(False)
   ax.get_yaxis().set_visible(False)
plt.show()
```



**Adding a sparsity constraint on the encoded representations**

In the previous example, the representations were only constrained by the size of the hidden layer (32). In such a situation, what typically happens is that the hidden layer is learning an approximation of PCA (principal component analysis). But another way to constrain the representations to be compact is to add a sparsity contraint on the activity of the hidden representations, so fewer units would "fire" at a given time. In Keras, this can be done by adding an activity_regularizer to our Dense layer:

```
from keras import regularizers
encoding_dim = 32
input_img = keras.Input(shape=(784,))
# Add a Dense layer with a L1 activity regularizer
encoded = layers.Dense(encoding_dim, activation='relu',
            activity_regularizer=regularizers.l1(10e-5))(input_img)
decoded = layers.Dense(784, activation='sigmoid')(encoded)

autoencoder = keras.Model(input_img, decoded)

#Let's also create a separate encoder model:
# This model maps an input to its encoded representation
encoder = keras.Model(input_img, encoded)

# This is our encoded (32-dimensional) input
encoded_input = keras.Input(shape=(encoding_dim,))
# Retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# Create the decoder model
decoder = keras.Model(encoded_input, decoder_layer(encoded_input))

#Now let's train our autoencoder to reconstruct MNIST digits.
```

#First, we'll configure our model to use a per-pixel binary crossentropy loss, and the Adam optimizer:

```
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

#Let's prepare our input data. We're using MNIST digits, and we're discarding the labels (since we're only interested in encoding/decoding the input images).

```
from keras.datasets import mnist
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()
```

# We will normalize all values between 0 and 1 and we will flatten the 28x28 images into vectors of size 784.

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print(x_train.shape)
print(x_test.shape)
```

```
(60000, 784)
(10000, 784)
```

# Now let's train our autoencoder for 50 epochs:

```
autoencoder.fit(x_train, x_train,
            epochs=50,
            batch_size=256,
            shuffle=True,
            validation_data=(x_test, x_test))
```

```
235/235 [==============================] - 5s 20ms/step - loss: 0.1024 - val_loss: 0.1012
Epoch 23/50
235/235 [==============================] - 3s 12ms/step - loss: 0.1022 - val_loss: 0.1009
Epoch 24/50
235/235 [==============================] - 4s 15ms/step - loss: 0.1020 - val_loss: 0.1007
Epoch 25/50
```

# Encode and decode some digits
# Note that we take them from the *test* set

```
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
```

```
313/313 [==============================] - 0s 1ms/step
313/313 [==============================] - 0s 1ms/step
```

```
# Use Matplotlib
import matplotlib.pyplot as plt

n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

**Deep autoencoder**

We do not have to limit ourselves to a single layer as encoder or decoder, we could instead use a stack of layers, such as:

```
input_img = keras.Input(shape=(784,))
encoded = layers.Dense(128, activation='relu')(input_img)
encoded = layers.Dense(64, activation='relu')(encoded)
encoded = layers.Dense(32, activation='relu')(encoded)

decoded = layers.Dense(64, activation='relu')(encoded)
decoded = layers.Dense(128, activation='relu')(decoded)
decoded = layers.Dense(784, activation='sigmoid')(decoded)

autoencoder = keras.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train,
            epochs=100,
            batch_size=256,
            shuffle=True,
            validation_data=(x_test, x_test))
```

```
Epoch 1/100
235/235 [==============================] - 5s 18ms/step - loss: 0.2466 - val_loss: 0.1719
Epoch 2/100
235/235 [==============================] - 5s 23ms/step - loss: 0.1538 - val_loss: 0.1374
Epoch 3/100
235/235 [==============================] - 5s 21ms/step - loss: 0.1315 - val_loss: 0.1238
```

```
# Encode and decode some digits
# Note that we take them from the *test* set
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
```

```
313/313 [==============================] - 1s 2ms/step
313/313 [==============================] - 0s 1ms/step
```

```
# Use Matplotlib
import matplotlib.pyplot as plt

n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
```

```
for i in range(n):
   # Display original
   ax = plt.subplot(2, n, i + 1)
   plt.imshow(x_test[i].reshape(28, 28))
   plt.gray()
   ax.get_xaxis().set_visible(False)
   ax.get_yaxis().set_visible(False)

   # Display reconstruction
   ax = plt.subplot(2, n, i + 1 + n)
   plt.imshow(decoded_imgs[i].reshape(28, 28))
   plt.gray()
   ax.get_xaxis().set_visible(False)
   ax.get_yaxis().set_visible(False)
plt.show()
```

# Practical No 9

## AIM : Implementation of convolutional neural network to predict numbers from number images

Implementing a Convolutional Neural Network (CNN) to predict numbers from images is a classic task in deep learning, often referred to as digit recognition. We'll use the MNIST dataset, which consists of 28x28 grayscale images of handwritten digits (0 to 9).

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data
3. Build the CNN Model
4. Compile the Model
5. Train the Model
6. Evaluate the Model
7. Make Predictions

**Code :**

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

```
X_train.shape
```

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

```
y_train.shape
```

```
(60000,)
```

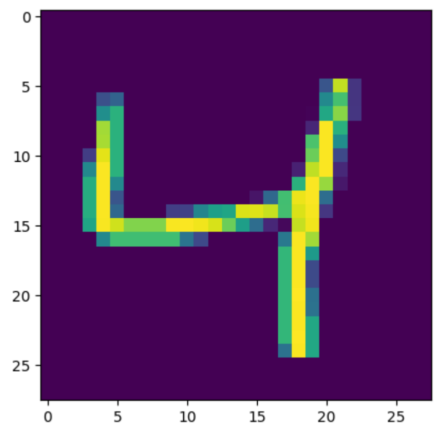
```
X_test.shape
```

```
(10000, 28, 28)
```

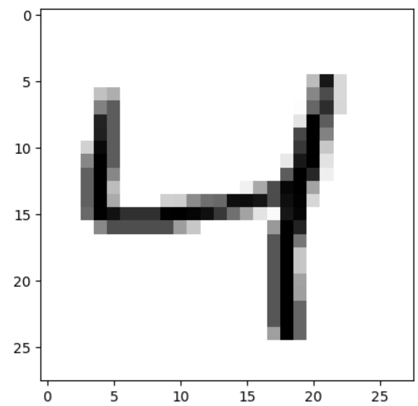
```
y_test.shape
```

```
(10000,)
```

```
import matplotlib.pyplot as plt
plt.imshow(X_train[2])
plt.show()
plt.imshow(X_train[2], cmap=plt.cm.binary)
```



<matplotlib.image.AxesImage at 0x7fdb3d698280>



X_train[2]

ndarray (28, 28) show data

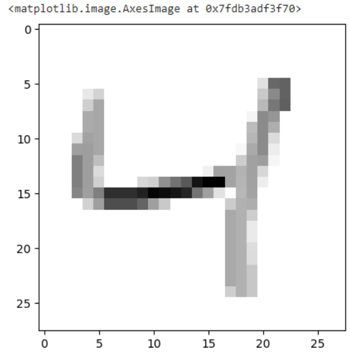
**Normalizing the data**

X_train = tf.keras.utils.normalize(X_train, axis=1)

X_test = tf.keras.utils.normalize(X_test, axis=1)

plt.imshow(X_train[2], cmap=plt.cm.binary)

<matplotlib.image.AxesImage at 0x7fdb3adf3f70>



print(X_train[2])

```
[[0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
```

import tensorflow as tf

import tensorflow.keras.layers  as KL

import tensorflow.keras.models  as KM

```
## Model
inputs = KL.Input(shape=(28, 28, 1))
c = KL.Conv2D(32, (3, 3), padding="valid", activation=tf.nn.relu)(inputs)
m = KL.MaxPool2D((2, 2), (2, 2))(c)
d = KL.Dropout(0.5)(m)
c = KL.Conv2D(64, (3, 3), padding="valid", activation=tf.nn.relu)(d)
m = KL.MaxPool2D((2, 2), (2, 2))(c)
d = KL.Dropout(0.5)(m)
c = KL.Conv2D(128, (3, 3), padding="valid", activation=tf.nn.relu)(d)
f = KL.Flatten()(c)
outputs = KL.Dense(10, activation=tf.nn.softmax)(f)
model = KM.Model(inputs, outputs)
model.summary()
model.compile(optimizer="adam", loss="sparse_categorical_crossentropy",
metrics=["accuracy"])
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2  (None, 13, 13, 32)        0
 D)

 dropout (Dropout)           (None, 13, 13, 32)        0

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 64)          0
 g2D)

 dropout_1 (Dropout)         (None, 5, 5, 64)          0

 conv2d_2 (Conv2D)           (None, 3, 3, 128)         73856

 flatten (Flatten)           (None, 1152)              0

 dense (Dense)               (None, 10)                11530

=================================================================
Total params: 104202 (407.04 KB)
Trainable params: 104202 (407.04 KB)
Non-trainable params: 0 (0.00 Byte)
```

```
model.fit(X_train, y_train, epochs=5)
test_loss, test_acc = model.evaluate(X_test, y_test)
print("Test Loss: {0} - Test Acc: {1}".format(test_loss, test_acc))
```

```
Epoch 1/5
1875/1875 [==============================] - 67s 33ms/step - loss: 0.2673 - accuracy: 0.9147
Epoch 2/5
1875/1875 [==============================] - 58s 31ms/step - loss: 0.0977 - accuracy: 0.9696
Epoch 3/5
1875/1875 [==============================] - 56s 30ms/step - loss: 0.0753 - accuracy: 0.9765
Epoch 4/5
1875/1875 [==============================] - 56s 30ms/step - loss: 0.0646 - accuracy: 0.9801
Epoch 5/5
1875/1875 [==============================] - 56s 30ms/step - loss: 0.0599 - accuracy: 0.9814
313/313 [==============================] - 3s 8ms/step - loss: 0.0338 - accuracy: 0.9896
Test Loss: 0.03380196541547775 - Test Acc: 0.9896000027656555
```

# Practical No 10

## AIM : Implementing Denoising of images using Autoencoder

Implementing denoising of images using an autoencoder involves training an autoencoder to remove noise from input images. Here's how you can do it step by step using TensorFlow and Keras:

Step-by-Step Implementation

1. Import Libraries
2. Load and Prepare the Data
3. Add Noise to the Images
4. Build the Autoencoder Model
5. Compile the Model
6. Train the Model
7. Evaluate the Model
8. Denoise Images

**Code :**

```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns
import warnings

warnings.filterwarnings('ignore')
from __future__ import print_function
from keras.models import Model
from keras.layers import Dense, Input
from keras.datasets import mnist
from keras.regularizers import l1
from keras.optimizers import Adam
```

**Utility Functions**

```
def plot_autoencoder_outputs(autoencoder, n, dims):
    decoded_imgs = autoencoder.predict(x_test)

    # number of example digits to show
    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(*dims))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        if i == n/2:
            ax.set_title('Original Images')

        # plot reconstruction
        ax = plt.subplot(2, n, i + 1 + n)
        plt.imshow(decoded_imgs[i].reshape(*dims))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        if i == n/2:
            ax.set_title('Reconstructed Images')
    plt.show()

def plot_loss(history):
    historydf = pd.DataFrame(history.history, index=history.epoch)
    plt.figure(figsize=(8, 6))
    historydf.plot(ylim=(0, historydf.values.max()))
    plt.title('Loss: %.3f' % history.history['loss'][-1])

def plot_compare_histories(history_list, name_list, plot_accuracy=True):
    dflist = []
    min_epoch = len(history_list[0].epoch)
    losses = []
```

```
    for history in history_list:
        h = {key: val for key, val in history.history.items() if not key.startswith('val_')}
        dflist.append(pd.DataFrame(h, index=history.epoch))
        min_epoch = min(min_epoch, len(history.epoch))
        losses.append(h['loss'][-1])

    historydf = pd.concat(dflist, axis=1)
    metrics = dflist[0].columns
    idx = pd.MultiIndex.from_product([name_list, metrics], names=['model', 'metric'])
    historydf.columns = idx

    plt.figure(figsize=(6, 8))
    ax = plt.subplot(211)
    historydf.xs('loss', axis=1, level='metric').plot(ylim=(0,1), ax=ax)
    plt.title("Training Loss: " + ' vs '.join([str(round(x, 3)) for x in losses]))

    if plot_accuracy:
        ax = plt.subplot(212)
        historydf.xs('acc', axis=1, level='metric').plot(ylim=(0,1), ax=ax)
        plt.title("Accuracy")
        plt.xlabel("Epochs")

    plt.xlim(0, min_epoch-1)
    plt.tight_layout()
```

**Deep Autoencoder**

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

print(x_train.shape)
print(x_test.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
(60000, 784)
(10000, 784)
```

```
input_size = 784
hidden_size = 128
code_size = 32

input_img = Input(shape=(input_size,))
hidden_1 = Dense(hidden_size, activation='relu')(input_img)
code = Dense(code_size, activation='relu')(hidden_1)
hidden_2 = Dense(hidden_size, activation='relu')(code)
output_img = Dense(input_size, activation='sigmoid')(hidden_2)

autoencoder = Model(input_img, output_img)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train, epochs=3)
```

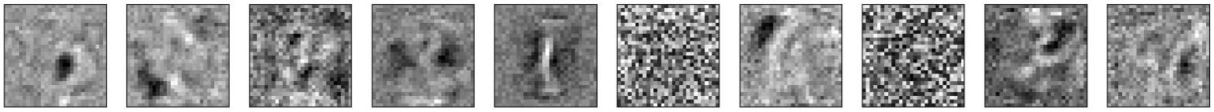
```
Epoch 1/3
1875/1875 [==============================] - 34s 15ms/step - loss: 0.1397
Epoch 2/3
1875/1875 [==============================] - 16s 8ms/step - loss: 0.1000
Epoch 3/3
1875/1875 [==============================] - 15s 8ms/step - loss: 0.0945
<keras.src.callbacks.History at 0x7ce17bb98490>
```

```
plot_autoencoder_outputs(autoencoder, 5, (28, 28))
```

```
313/313 [==============================] - 1s 4ms/step
```

```
weights = autoencoder.get_weights()[0].T

n = 10
plt.figure(figsize=(20, 5))
for i in range(n):
    ax = plt.subplot(1, n, i + 1)
    plt.imshow(weights[i+0].reshape(28, 28))
    ax.get_xaxis().set_visible(False)
```

```
    ax.get_yaxis().set_visible(False)
```



**Shallow Autoencoder**

```
input_size = 784
code_size = 32
input_img = Input(shape=(input_size,))
code = Dense(code_size, activation='relu')(input_img)
output_img = Dense(input_size, activation='sigmoid')(code)

autoencoder = Model(input_img, output_img)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.1613
Epoch 2/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.1045
Epoch 3/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0966
Epoch 4/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.0951
Epoch 5/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0946
<keras.src.callbacks.History at 0x7ce0f40b11b0>
```

```
plot_autoencoder_outputs(autoencoder, 5, (28, 28))
```

```
313/313 [==============================] - 1s 3ms/step
```

```
weights = autoencoder.get_weights()[0].T
n = 10
```

```
plt.figure(figsize=(20, 5))
for i in range(n):
    ax = plt.subplot(1, n, i + 1)
    plt.imshow(weights[i+20].reshape(28, 28))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```

Denoising Autoencoder

```
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()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    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()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    if i == n/2:
        ax.set_title('Noisy Input')
```

```
input_size = 784
hidden_size = 128
code_size = 32

input_img = Input(shape=(input_size,))
hidden_1 = Dense(hidden_size, activation='relu')(input_img)
code = Dense(code_size, activation='relu')(hidden_1)
hidden_2 = Dense(hidden_size, activation='relu')(code)
output_img = Dense(input_size, activation='sigmoid')(hidden_2)

autoencoder = Model(input_img, output_img)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train_noisy, x_train, epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 16s 8ms/step - loss: 0.1629
Epoch 2/10
1875/1875 [==============================] - 16s 8ms/step - loss: 0.1272
Epoch 3/10
1875/1875 [==============================] - 15s 8ms/step - loss: 0.1205
Epoch 4/10
1875/1875 [==============================] - 15s 8ms/step - loss: 0.1171
Epoch 5/10
```

```
n = 5
plt.figure(figsize=(10, 7))
images = autoencoder.predict(x_test_noisy)
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()
   ax.get_xaxis().set_visible(False)
   ax.get_yaxis().set_visible(False)
```

```
    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()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    if i == n/2:
        ax.set_title('Noisy Input')

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

```
313/313 [==============================] - 1s 3ms/step
```