**EXPERIMENT NO- 01**

**AIM:** Implementation of Classification with Multilayer Perceptron using Sckit-learn (MNIST Dataset)

**DESCRIPTION:**

**Modules used:**

NumPy: NumPy stands for Numerical Python. It is one of the basic Python Library that is used for creating arrays, filling null values, statistical calculations and computations. Pandas is built on the top of the NumPy library.

Matplotlib: Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

MLP Classifier: Multi-layer Perceptron (MLP) is a supervised learning algorithm that learns a function by training on a dataset, where m is the number of dimensions for input and o is the number of dimensions for output. Given a set of features X = x1,x2,,x3,…and a target y , it can learn

The process of training and evaluating a Multi-Layer Perceptron (MLP) classifier using the MNIST dataset, which consists of handwritten digit images.

After importing necessary libraries, the script loads the dataset, normalizes pixel values, and splits the data into training and testing sets.

An MLP model is created and trained using the training data. The model's predictions are then computed for the test data.

The script evaluates the model's performance by generating a confusion matrix and a classification report, which provides insights into its accuracy and precision for each digit class.

Additionally, a heatmap visualization of the confusion matrix is produced using the seaborn library, enhancing the understanding of the model's performance briefly. The code showcases a complete workflow for building, training, and assessing an MLP classifier for digit recognition.

**CODE:**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.neural\_network import MLPClassifier

from sklearn.metrics import classification\_report, confusion\_matrix

import seaborn as sns

# Load the MNIST dataset

from sklearn.datasets import fetch\_openml

mnist = fetch\_openml('mnist\_784', version=1)

X, y = mnist.data, mnist.target.astype(int)

# Normalize pixel values to the range [0, 1]

X /= 255.0

# Split the dataset into train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create an MLP model

mlp = MLPClassifier(hidden\_layer\_sizes=(100, 50), max\_iter=10, random\_state=42)

# Train the model

mlp.fit(X\_train, y\_train)

# Make predictions

predictions = mlp.predict(X\_test)

# Evaluate the model's performance

conf\_matrix = confusion\_matrix(y\_test, predictions)

print(conf\_matrix)

print(classification\_report(y\_test, predictions))

# Plot the confusion matrix heatmap using seaborn

plt.figure(figsize=(10, 8))

sns.heatmap(conf\_matrix, annot=True, cmap="Blues", fmt="d")

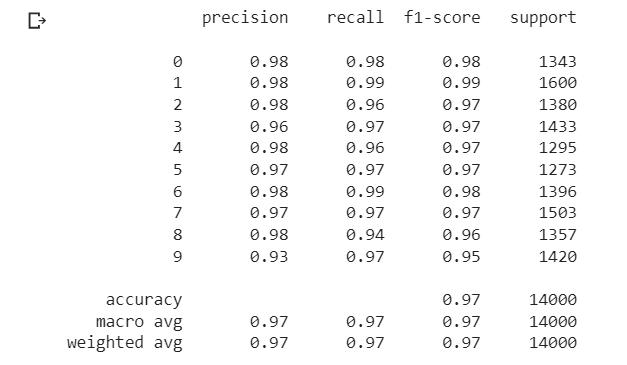
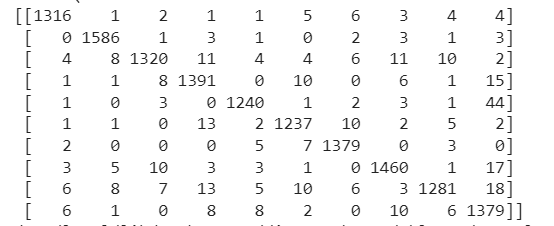
plt.xlabel("Predicted")

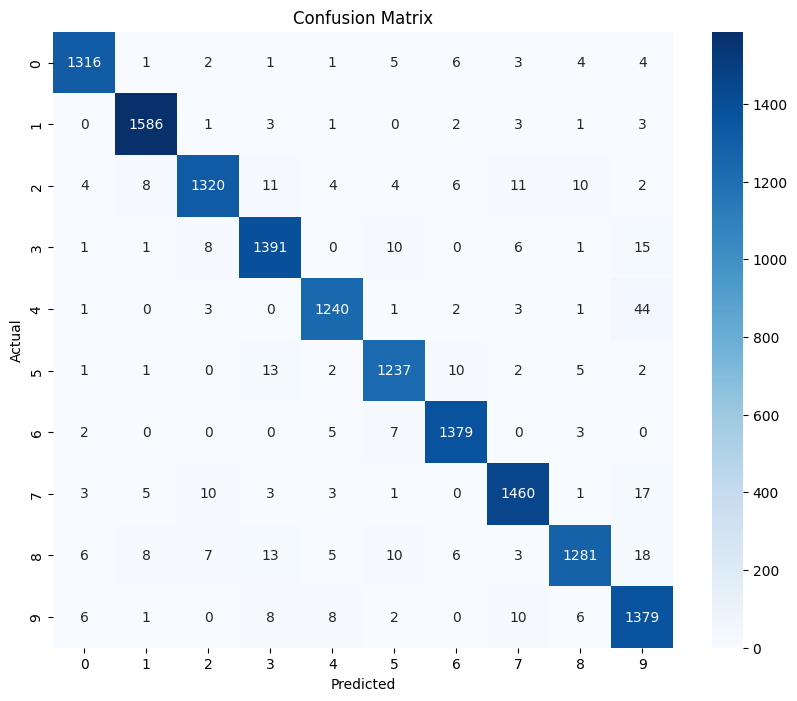
plt.ylabel("Actual")

plt.title("Confusion Matrix")

plt.show()

**OUTPUT:**





**EXPERIMENT NO-02**

**AIM:** Understanding of Deep learning Packages Basics: Tensorflow, Keras, Theano and PyTorch.

**DESCRIPTION:**

TensorFlow:

* Open-source deep learning framework by Google Brain.
* Provides both high-level and low-level APIs for building and deploying machine learning models.
* Uses computational graphs to define and execute operations.
* Widely used for production deployments due to its scalability and ecosystem.
* Allows distributed computing for training large models.
* Supports GPU acceleration for faster training.
* TensorFlow 2.0 and later versions incorporate the Keras high-level API as the official interface.

Keras:

* High-level neural networks API designed for rapid experimentation.
* Originally separate from TensorFlow but integrated into it from version 2.0 onward.
* Offers a user-friendly interface for building and training models.
* Helps researchers and developers prototype models quickly.
* Provides a clear and intuitive way to define neural network architectures.
* Can be used with TensorFlow, Theano (discontinued), and Microsoft Cognitive Toolkit (CNTK) backends.

Theano:

* Open-source numerical computation library for efficient mathematical expression evaluation.
* Primarily used for building neural network models.
* Development has been discontinued (as of my last update in September 2021).
* Utilized symbolic mathematical expressions for optimization.
* Was popular for its efficiency and performance gains, though other frameworks have since gained prominence.

PyTorch:

* Open-source deep learning framework by Facebook's AI Research lab (FAIR).
* Emphasizes dynamic computation graphs, making model construction more intuitive.
* Offers a flexible and easy-to-use interface for defining and training models.
* Well-suited for research and experimentation, as well as debugging.
* Supports GPU acceleration and provides strong integration with CUDA for efficient computation.
* Used for various machine learning tasks, from research to production.

**CODE:**

import tensorflow as tf

from tensorflow.keras.datasets import mnist

(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()

train\_images, test\_images = train\_images / 255.0, test\_images / 255.0

model = tf.keras.Sequential([

tf.keras.layers.Flatten(input\_shape=(28, 28)),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dropout(0.2),

tf.keras.layers.Dense(10, activation='softmax')

])

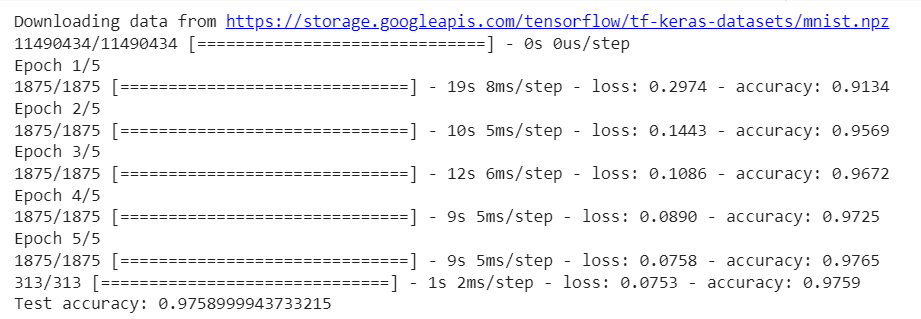
model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

model.fit(train\_images, train\_labels, epochs=5)

test\_loss, test\_acc = model.evaluate(test\_images, test\_labels)

print("Test accuracy:", test\_acc)

**OUTPUT:**



**CODE:**

import tensorflow as tf

from tensorflow.keras.datasets import mnist

(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()

train\_images, test\_images = train\_images / 255.0, test\_images / 255.0

model = tf.keras.Sequential([

tf.keras.layers.Flatten(input\_shape=(28, 28)),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dropout(0.2),

tf.keras.layers.Dense(10, activation='softmax')

])

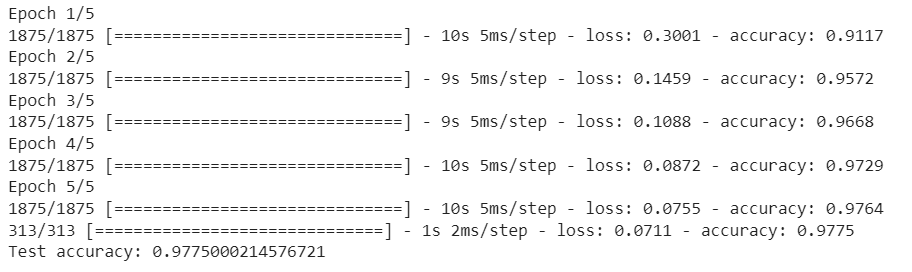
model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

model.fit(train\_images, train\_labels, epochs=5)

test\_loss, test\_acc = model.evaluate(test\_images, test\_labels)

print("Test accuracy:", test\_acc)

**OUTPUT:**



**CODE:**

import numpy as np

import theano

import theano.tensor as T

# Define symbolic variables

x = T.dscalar('x')

y = T.dscalar('y')

z = x + y

# Compile a function

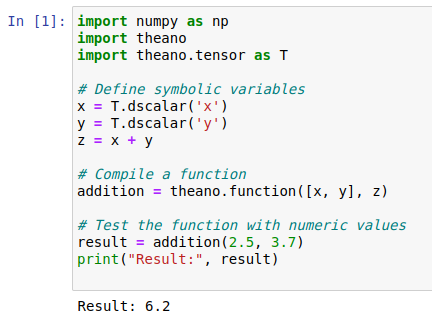
addition = theano.function([x, y], z)

# Test the function with numeric values

result = addition(2.5, 3.7)

print("Result:", result)

**OUTPUT:**



**CODE:**

# importing torch

import torch

# creating a tensors

t1=torch.tensor([1, 2, 3, 4])

t2=torch.tensor([[1, 2, 3, 4],

[5, 6, 7, 8],

[9, 10, 11, 12]])

# printing the tensors:

print("Tensor t1: \n", t1)

print("\nTensor t2: \n", t2)

# rank of tensors

print("\nRank of t1: ", len(t1.shape))

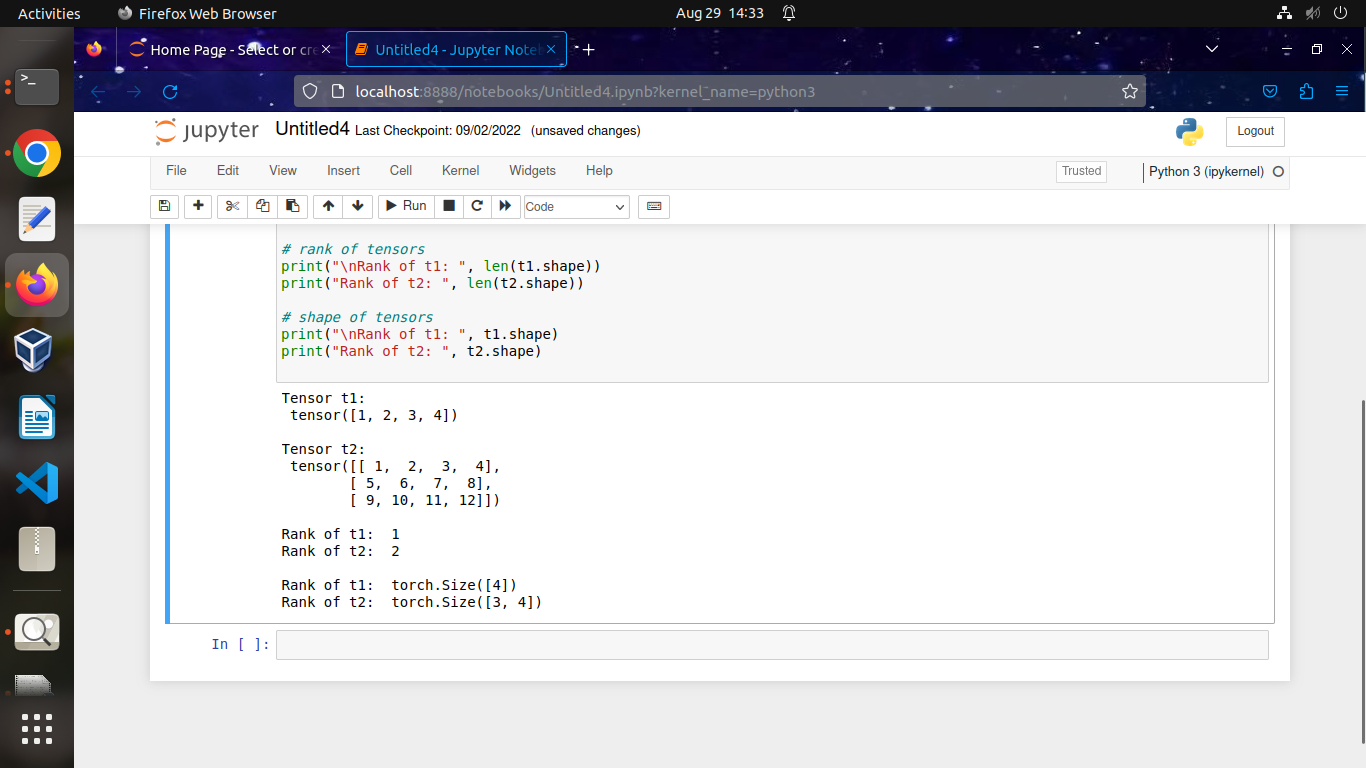
print("Rank of t2: ", len(t2.shape))

# shape of tensors

print("\nRank of t1: ", t1.shape)

print("Rank of t2: ", t2.shape)

**OUTPUT:**

****

**EXPERIMENT NO-03**

**AIM:** Improve the performance of Deep learning models with Hyper-Parameter Tuning.

**DESCRIPTION:**

H**yperparameters in Machine learning are those parameters that are explicitly defined by the user to control the learning process.**These hyperparameters are used to improve the learning of the model, and their values are set before starting the learning process of the model.

Here the prefix "hyper" suggests that the parameters are top-level parameters that are used in controlling the learning process. The value of the Hyperparameter is selected and set by the machine learning engineer before the learning algorithm begins training the model. **Hence, these are external to the model, and their values cannot be changed during the training process**.

**CODE:**

import numpy as np

import pandas as pd

import tensorflow as tf

from tensorflow import keras

from sklearn.model\_selection import train\_test\_split, GridSearchCV

from keras.wrappers.scikit\_learn import KerasClassifier

from sklearn.datasets import load\_iris

# Load the Iris dataset from scikit-learn

data = load\_iris()

X = data.data # Features

y = data.target # Target labels

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Define a function to create your neural network model

def create\_model(learning\_rate=0.01, num\_units=64):

model = keras.Sequential([

keras.layers.Dense(units=num\_units, activation='relu', input\_shape=(X\_train.shape[1],)),

keras.layers.Dense(units=num\_units, activation='relu'),

keras.layers.Dense(units=3, activation='softmax') # Multi-class classification

])

optimizer = keras.optimizers.Adam(learning\_rate=learning\_rate)

model.compile(optimizer=optimizer, loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

return model

# Create a KerasClassifier with your model function

model = KerasClassifier(build\_fn=create\_model, epochs=5, batch\_size=10)

# Define the hyperparameters you want to tune

param\_grid = {

'learning\_rate': [0.001, 0.01, 0.1],

'num\_units': [32, 64, 128]

}

# Perform hyperparameter tuning using GridSearchCV

grid\_search = GridSearchCV(estimator=model, param\_grid=param\_grid, cv=3, verbose=1)

grid\_result = grid\_search.fit(X\_train, y\_train)

# Print the best hyperparameters and corresponding performance

print(f"Best Parameters: {grid\_result.best\_params\_}")

print(f"Best Accuracy: {grid\_result.best\_score\_}")

# Train your final model with the best hyperparameters

best\_model = grid\_result.best\_estimator\_.model

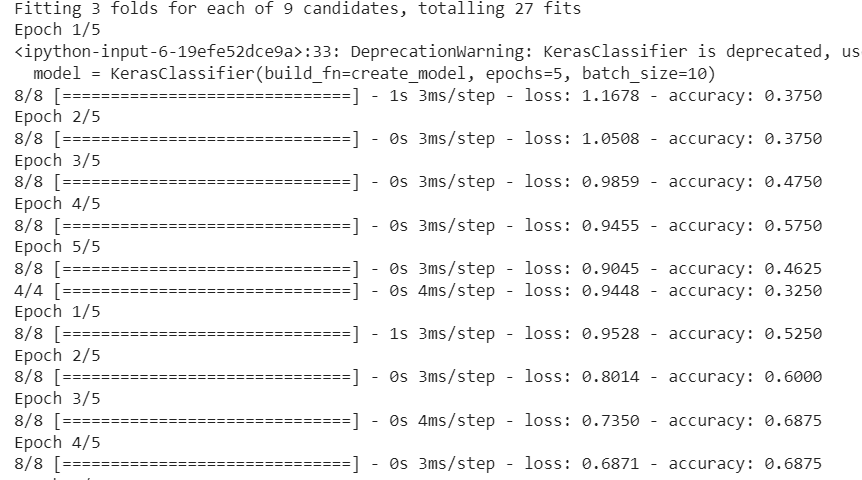
best\_model.fit(X\_train, y\_train, epochs=30, batch\_size=32)

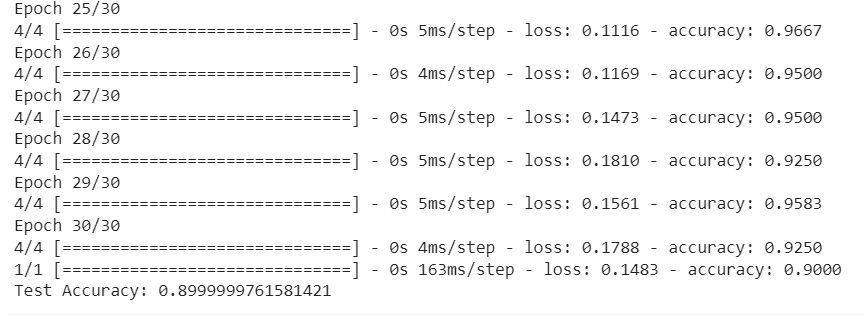
# Evaluate the final model on the test set

test\_loss, test\_accuracy = best\_model.evaluate(X\_test, y\_test)

print(f"Test Accuracy: {test\_accuracy}")

**OUTPUT:**

****

****

**EXPERIMENT NO-04**

**AIM:** IIlustrate the performance of various Optimization techniques of Gradient Descent(GD),Momentum Based GD,Nesterov Accelerated GD,Stochastic GD,AdaGrad,RMSProp,Adam

**DESCRIPTION:**

The various optimization techniques in deep learning, including Gradient Descent, Momentum-Based GD, Nesterov Accelerated GD, Stochastic GD, AdaGrad, RMSProp, and Adam, aim to efficiently update model parameters during training. Gradient Descent computes parameter updates based on the full dataset, while Momentum, Nesterov, and Stochastic GD introduce momentum terms to accelerate convergence, with Nesterov being an improved variant of Momentum. AdaGrad adapts learning rates individually for each parameter, RMSProp scales learning rates based on recent gradient magnitudes, and Adam combines the benefits of momentum and adaptive learning rates for faster and stable convergence on a wide range of tasks. Each technique has its strengths and may perform differently depending on the problem and dataset.

**CODE:**

**import numpy as np**

**import tensorflow as tf**

**import matplotlib.pyplot as plt**

**from tensorflow.keras.datasets import mnist**

**# Load the MNIST dataset**

**(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()**

**# Normalize pixel values to be between 0 and 1**

**X\_train, X\_test = X\_train / 255.0, X\_test / 255.0**

**# Define a function to create and compile a model**

**def create\_model(optimizer):**

**model = tf.keras.Sequential([**

**tf.keras.layers.Flatten(input\_shape=(28, 28)),**

**tf.keras.layers.Dense(128, activation='relu'),**

**tf.keras.layers.Dropout(0.2),**

**tf.keras.layers.Dense(10, activation='softmax')**

**])**

**model.compile(optimizer=optimizer,**

**loss='sparse\_categorical\_crossentropy',**

**metrics=['accuracy'])**

**return model**

**# Define different optimizers**

**optimizers = {**

**'SGD': tf.keras.optimizers.SGD(learning\_rate=0.01),**

**'Momentum': tf.keras.optimizers.SGD(learning\_rate=0.01, momentum=0.9),**

**'Nesterov': tf.keras.optimizers.SGD(learning\_rate=0.01, momentum=0.9, nesterov=True),**

**'AdaGrad': tf.keras.optimizers.Adagrad(learning\_rate=0.01),**

**'RMSProp': tf.keras.optimizers.RMSprop(learning\_rate=0.001),**

**'Adam': tf.keras.optimizers.Adam(learning\_rate=0.001)**

**}**

**# Initialize a dictionary to store accuracy history for each optimizer**

**accuracy\_history = {}**

**# Train and evaluate models with different optimizers**

**num\_epochs = 5**

**for optimizer\_name, optimizer in optimizers.items():**

**model = create\_model(optimizer)**

**history = model.fit(X\_train, y\_train, epochs=num\_epochs, verbose=1, validation\_data=(X\_test, y\_test))**

**accuracy\_history[optimizer\_name] = history.history['accuracy']**

**# Plot accuracy curves for each optimizer**

**plt.figure(figsize=(10, 6))**

**for optimizer\_name, accuracy\_values in accuracy\_history.items():**

**plt.plot(accuracy\_values, label=optimizer\_name)**

**plt.title('Accuracy Curves for Different Optimizers')**

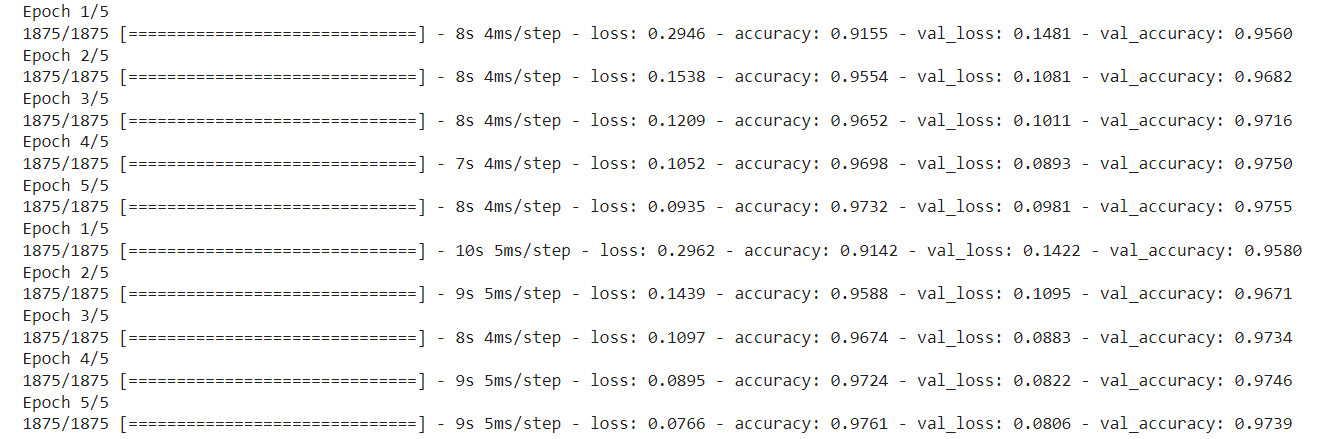
**plt.xlabel('Epochs')**

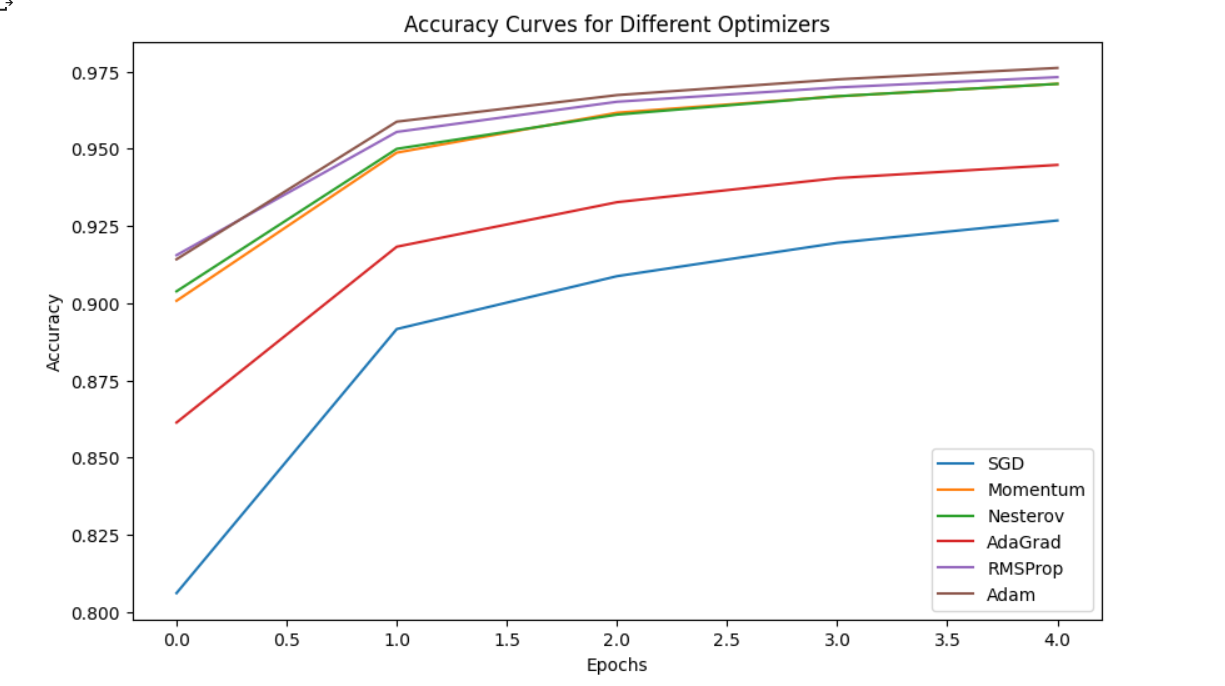
**plt.ylabel('Accuracy')**

**plt.legend()**

**plt.show()**

**OUTPUT:**

****

****

**EXPERIMENT NO-05**

**AIM:** Implementing of Denoising, sparse and contractive autoencoders.

**DESCRIPTION:**

Autoencoders are a type of artificial neural network used in machine learning and deep learning. They are designed to learn a compressed representation of input data, by encoding it into a smaller set of features, and then decoding it back into the original form.

A denoising autoencoder is a type of autoencoder that is designed to remove noise from input data. It works by learning a compressed representation of the input data, and then using this representation to reconstruct the original data without the noise. The process of training a denoising autoencoder involves adding noise to the input data, and then training the network to reconstruct the original data without the noise. The idea is that the network will learn to focus on the underlying structure of the data, rather than the noise, which will improve its ability to reconstruct clean data.

To achieve this, a denoising autoencoder typically uses a different loss function than a regular autoencoder, such as mean squared error (MSE) or binary cross- entropy. Additionally, the network architecture may include additional layers or other modifications to help the network learn to remove noise from the input.

**CODE:**

import numpy

import matplotlib.pyplot as plt

from keras.models import Sequential from keras.layers import Dense

from keras.datasets import mnist

(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()

X\_train.shape

X\_test.shape

plt.subplot(221)

plt.imshow(X\_train[0], cmap=plt.get\_cmap('gray')) plt.subplot(222)

plt.imshow(X\_train[1], cmap=plt.get\_cmap('gray'))

plt.subplot(223)

plt.imshow(X\_train[2], cmap=plt.get\_cmap('gray')) plt.subplot(224)

plt.imshow(X\_train[3], cmap=plt.get\_cmap('gray')) # show the plot

plt.show()

A group of numbers in black squares

Description automatically generated

num\_pixels = X\_train.shape[1] \* X\_train.shape[2]

X\_train = X\_train.reshape(X\_train.shape[0], num\_pixels).astype('float32') X\_test = X\_test.reshape(X\_test.shape[0], num\_pixels).astype('float32') X\_train = X\_train / 255

X\_test = X\_test / 255 X\_train.shape

X\_test.shape

noise\_factor = 0.2

x\_train\_noisy = X\_train + noise\_factor \* numpy.random.normal(loc=0.0, scale=1.0

, size=X\_train.shape)

x\_test\_noisy = X\_test + noise\_factor \* numpy.random.normal(loc=0.0, scale=1.0, s ize=X\_test.shape)

x\_train\_noisy = numpy.clip(x\_train\_noisy, 0., 1.)

x\_test\_noisy = numpy.clip(x\_test\_noisy, 0., 1.)

# create model model = Sequential()

model.add(Dense(500, input\_dim=num\_pixels, activation='relu')) model.add(Dense(300, activation='relu'))

model.add(Dense(100, activation='relu'))

model.add(Dense(300, activation='relu')) model.add(Dense(500, activation='relu')) model.add(Dense(784, activation='sigmoid'))

# Compile the model model.compile(loss='mean\_squared\_error', optimizer='adam')

# Training model

model.fit(x\_train\_noisy, X\_train, validation\_data=(x\_test\_noisy, X\_test), epochs=2

, batch\_size=200)

# Final evaluation of the model pred = model.predict(x\_test\_noisy) pred.shape

X\_test.shape

X\_test = numpy.reshape(X\_test, (10000,28,28)) \*255 pred = numpy.reshape(pred, (10000,28,28)) \*255

x\_test\_noisy = numpy.reshape(x\_test\_noisy, (-1,28,28)) \*255 plt.figure(figsize=(20, 4))

print("Test Images") for i in range(10,20,1):

plt.subplot(2, 10, i+1) plt.imshow(X\_test[i,:,:], cmap='gray') curr\_lbl = y\_test[i]

plt.title("(Label: " + str(curr\_lbl) + ")") plt.show()

plt.figure(figsize=(20, 4)) print("Test Images with Noise") for i in range(10,20,1):

plt.subplot(2, 10, i+1) plt.imshow(x\_test\_noisy[i,:,:], cmap='gray')

plt.show() plt.figure(figsize=(20, 4))

print("Reconstruction of Noisy Test Images")

for i in range(10,20,1):

plt.subplot(2, 10, i+1)

plt.imshow(pred[i,:,:], cmap='gray')

plt.show()

**OUTPUT:**

A number in a black box

Description automatically generated with medium confidence

**EXPERIMENT NO-06**

**AIM:** Evaluating the performance of the model using various Regularization Techniques.

**DESCRIPTION:**

L2 & L1 regularization

L1 and L2 are the most common types of regularization. These update the general cost function by adding another term known as the regularization term.

Cost function = Loss (say, binary cross entropy) + Regularization term

Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent.

However, this regularization term differs in L1 and L2. In L2, we have:



Here, lambda is the regularization parameter. It is the hyperparameter whose value is optimized for better results. L2 regularization is also known as weight decay as it forces the weights to decay towards zero (but not exactly zero).

In L1, we have:



Early stopping

Early stopping is a kind of cross-validation strategy where we keep one part of the training set as the validation set. When we see that the performance on the validation set is getting worse, we immediately stop the training on the model. This is known as early stopping.

Dropout

This is the one of the most interesting types of regularization techniques. It also produces very good results and is consequently the most frequently used regularization technique in the field of deep learning.

**CODE:**

import numpy as np import tensorflow as tf

from tensorflow import keras import matplotlib.pyplot as plt

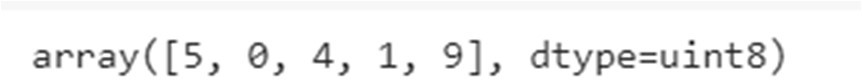
(X\_train,y\_train),(X\_test,y\_test)=keras.datasets.mnist.load\_data()

X\_train

A black and white sheet of paper with numbers and symbols

Description automatically generated

y\_train[:5]



X\_train.shape

A close up of numbers

Description automatically generated

X\_train[0].shape

A close up of a number

Description automatically generated

A screenshot of a computer

Description automatically generated

X\_train = X\_train / 255 X\_test = X\_test / 255

X\_train\_flat = X\_train.reshape(len(X\_train), 28\*28) X\_test\_flat = X\_test.reshape(len(X\_test), 28\*28)

X\_train\_flat.shape

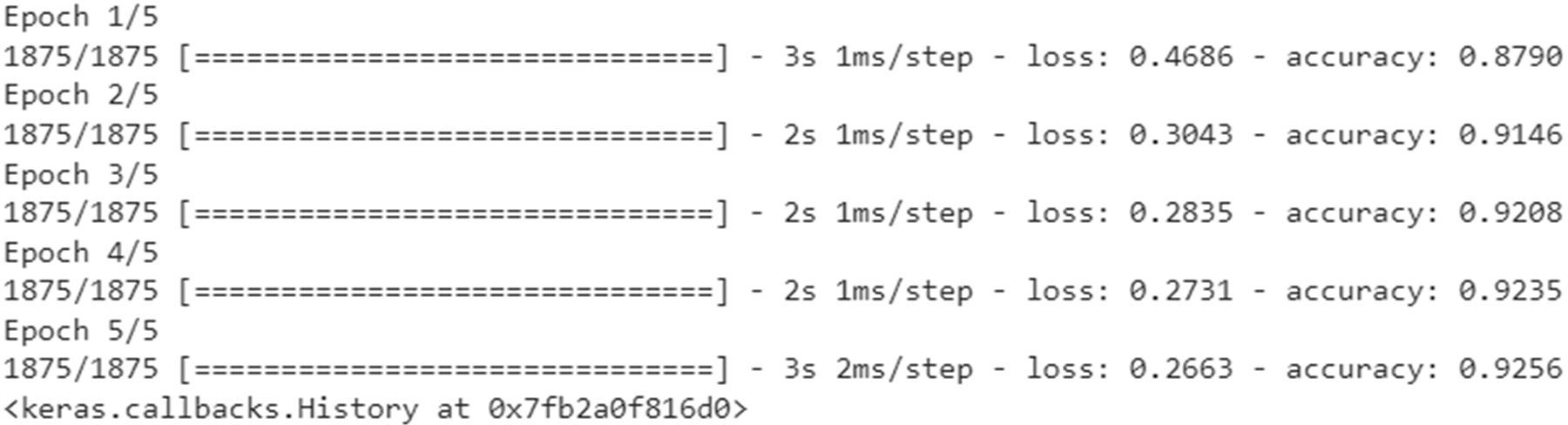
A close up of numbers

Description automatically generated

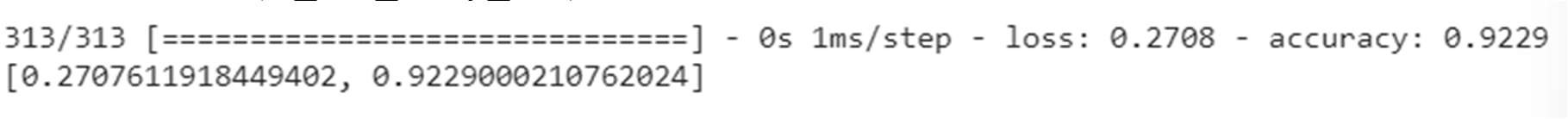
model=keras.Sequential([ keras.layers.Dense(10,input\_shape=(784,),activation='sigmoid')])

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

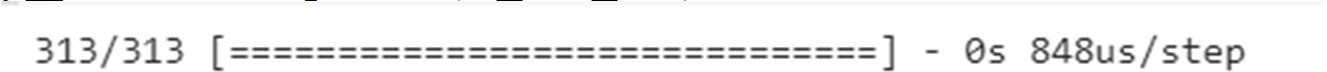
model.fit(X\_train\_flat,y\_train,epochs=5)



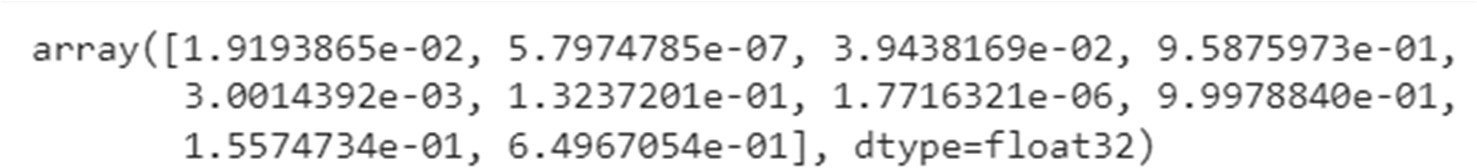
model.evaluate(X\_test\_flat,y\_test)



y\_pred=model.predict(X\_test\_flat)



y\_pred[0]



A screenshot of a computer program

Description automatically generated

A close up of a text

Description automatically generated

y\_pred\_labels=[np.argmax(i) for i in y\_pred] cm=tf.math.confusion\_matrix(labels=y\_test,predictions=y\_pred\_labels)

import seaborn as sns plt.figure(figsize=(10,8)) sns.heatmap(cm,annot=True,fmt='d')

A screenshot of a graph

Description automatically generated

from keras import regularizers model1=keras.Sequential([

keras.layers.Dense(100,input\_shape=(784,),activation='relu',kernel\_regularizer=regularizers.l2 (0.0001)),

keras.layers.Dense(10,activation='sigmoid')])

model1.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

model1.fit(X\_train\_flat,y\_train,epochs=5)

A group of black lines with numbers

Description automatically generated with medium confidence

model1.evaluate(X\_test\_flat,y\_test)



model2=keras.Sequential([ keras.layers.Dense(100,input\_shape=(784,),activation='relu',kernel\_regularizer=regularizers.l1

(0.0001)),

keras.layers.Dense(10,activation='sigmoid')

])

model2.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy']

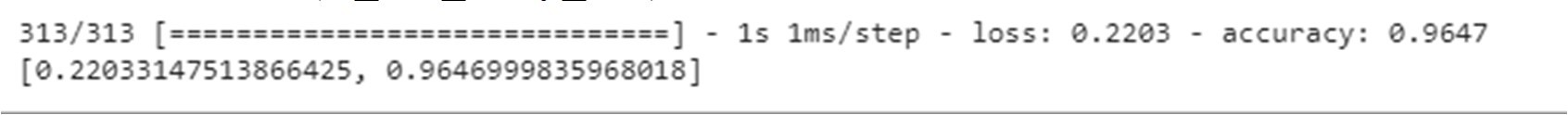
)

model2.fit(X\_train\_flat,y\_train,epochs=5)

A screenshot of a computer program

Description automatically generated

model2.evaluate(X\_test\_flat,y\_test)



from keras.layers.core import Dropout model3=keras.Sequential([

keras.layers.Dense(100,input\_shape=(784,),activation='relu'), Dropout(0.25), keras.layers.Dense(10,activation='sigmoid')

])

model3.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy']

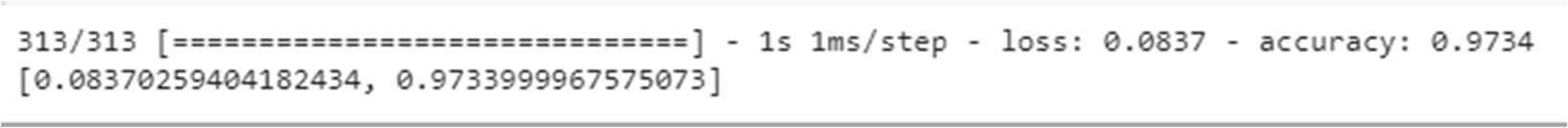
)

model3.fit(X\_train\_flat,y\_train,epochs=5)

A black and white text

Description automatically generated with medium confidence

model3.evaluate(X\_test\_flat,y\_test)



from keras.callbacks import EarlyStopping

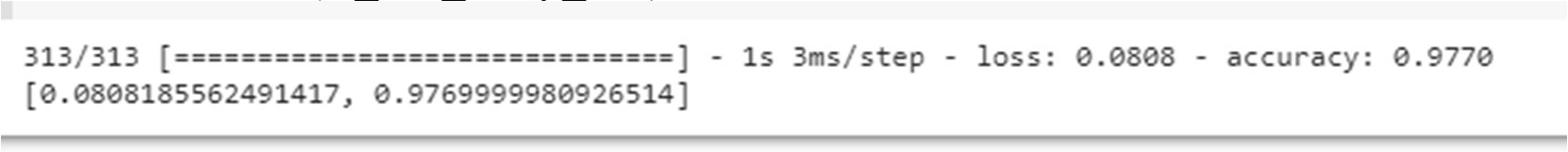
model3.fit(X\_train\_flat,y\_train,epochs=5,callbacks = [EarlyStopping(monitor='val\_acc', patience

=2)])

A white text with black text

Description automatically generated

model3.evaluate(X\_test\_flat,y\_test)



**EXPERIMENT NO-07**

**AIM:** Train a Deep Learning model to classify a given image using pretrained model of AlexNet,ZF-Net,VGGnet,GoogleNet,ResNet.

**DESCRIPTION:**

VGG introduced the concept of increasing the number of layers to improve accuracy. However, increasing the number of layers above 20 could prevent the model from converging. The main reason is the vanishing gradient problem—after too many folds, the learning rate is so low that the model’s weights cannot change.

Another issue is gradient explosion. A solution is gradient clipping, which involves “clipping” the error derivative to a certain threshold during backward propagation and using these clipped gradients to update the weights. When the error derivative is rescaled, weights are also rescaled, and this reduces the chance of an overflow or underflow that can lead to gradient explosion.

The Residual Network (ResNet) architecture uses the concept of skip connections, allowing inputs to “skip” some convolutional layers. The result is a significant reduction in training time and improved accuracy. After the model learns a given feature, it won’t attempt to learn it again—instead, it will focus on learning the new features. It’s a clever approach that can significantly improve model training.

**CODE:**

VGGNet 16

%matplotlib inline import numpy as np

import matplotlib.pyplot as plt from os import makedirs

from os.path import join, exists, expanduser

from tensorflow.keras.preprocessing import image

from tensorflow.keras.applications.vgg16 import VGG16,preprocess\_input

from tensorflow.keras.applications.imagenet\_utils import decode\_predictions fig, ax = plt.subplots(1, figsize=(12, 10))

img = image.load\_img('/content/butterfly.jpeg') img = image.img\_to\_array(img) ax.imshow(img / 255.)

ax.axis('off') plt.show()

vgg = VGG16(weights='imagenet')

img = image.load\_img('/content/butterfly.jpeg', target\_size=(224, 224)) img = image.img\_to\_array(img)

plt.imshow(img / 255.)

x = preprocess\_input(np.expand\_dims(img.copy(), axis=0)) preds = vgg.predict(x)

decode\_predictions(preds, top=3)

**OUTPUT:**

****

**CODE:**

ResNet50:

%matplotlib inline import numpy as np

import matplotlib.pyplot as plt from os import makedirs

from os.path import join, exists, expanduser

from tensorflow.keras.preprocessing import image

from tensorflow.keras.applications.resnet50 import ResNet50,preprocess\_input

from tensorflow.keras.applications.imagenet\_utils import decode\_predictions fig, ax = plt.subplots(1, figsize=(12, 10))

img = image.load\_img('/content/butterfly.jpeg') img = image.img\_to\_array(img) ax.imshow(img / 255.)

ax.axis('off')

plt.show()

resnet = ResNet50(weights='imagenet')

img = image.load\_img('/content/butterfly.jpeg', target\_size=(224, 224)) img = image.img\_to\_array(img)

plt.imshow(img / 255.)

x = preprocess\_input(np.expand\_dims(img.copy(), axis=0)) preds = resnet.predict(x)

decode\_predictions(preds, top=3)

**OUTPUT:**

A butterfly on a flower

Description automatically generated

**EXPERIMENT NO-08**

**AIM:** Implement of Deep learning model using guided backpropagation.

**DESCRIPTION:**

Guided Backpropagation is the combination of vanilla backpropagation at ReLUs and DeconvNets. ReLU is an activation function that deactivates the negative neurons. DeconvNets are simply the deconvolution and unpooling layers. We are only interested in knowing what image features the neuron detects. So when propagating the gradient, we set all the negative gradients to 0. We don’t care if a pixel “suppresses’’ (negative value) a neuron somewhere along the part to our neuron. Value in the filter map greater than zero signifies the pixel importance, which is overlapped with the input image to show which pixel from the input image contributed the most.

**CODE:**

import torch

from torch import nn

from torchvision import models, transforms

from PIL import Image

import matplotlib.pyplot as plt

class Guided\_backprop():

def \_\_init\_\_(self, model):

self.model = model

self.image\_reconstruction = None # store R0

self.activation\_maps = [] # store f1, f2, ...

self.model.eval()

self.register\_hooks()

def register\_hooks(self):

def first\_layer\_hook\_fn(module, grad\_in, grad\_out):

self.image\_reconstruction = grad\_in[0]

def forward\_hook\_fn(module, input, output):

self.activation\_maps.append(output)

def backward\_hook\_fn(module, grad\_in, grad\_out):

grad = self.activation\_maps.pop()

# for the forward pass, after the ReLU operation,

# if the output value is positive, we set the value to 1,

# and if the output value is negative, we set it to 0.

grad[grad > 0] = 1

# grad\_out[0] stores the gradients for each feature map,

# and we only retain the positive gradients

positive\_grad\_out = torch.clamp(grad\_out[0], min=0.0)

new\_grad\_in = positive\_grad\_out \* grad

return (new\_grad\_in,)

# AlexNet model

modules = list(self.model.features.named\_children())

# travese the modules，register forward hook & backward hook

# for the ReLU

for name, module in modules:

if isinstance(module, nn.ReLU):

module.register\_forward\_hook(forward\_hook\_fn)

module.register\_backward\_hook(backward\_hook\_fn)

# register backward hook for the first conv layer

first\_layer = modules[0][1]

first\_layer.register\_backward\_hook(first\_layer\_hook\_fn)

def visualize(self, input\_image, target\_class):

model\_output = self.model(input\_image)

self.model.zero\_grad()

pred\_class = model\_output.argmax().item()

grad\_target\_map = torch.zeros(model\_output.shape,

dtype=torch.float)

if target\_class is not None:

grad\_target\_map[0][target\_class] = 1

else:

grad\_target\_map[0][pred\_class] = 1

model\_output.backward(grad\_target\_map)

result = self.image\_reconstruction.data[0].permute(1,2,0)

return result.numpy()

def normalize(image):

norm = (image - image.mean())/image.std()

norm = norm \* 0.1

norm = norm + 0.5

norm = norm.clip(0, 1)

return norm

image = Image.open('./dog.jpg').convert('RGB')

transform = transforms.Compose([

transforms.Resize(224),

transforms.CenterCrop(224),

transforms.ToTensor(),

transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

])

tensor = transform(image).unsqueeze(0).requires\_grad\_()

model = models.alexnet(pretrained=True)

print('AlexNet Architecture:\n', '-'\*60, '\n', model, '\n', '-'\*60)

guided\_bp = Guided\_backprop(model)

result = guided\_bp.visualize(tensor, None)

result = normalize(result)

plt.imshow(result)

plt.show()

**OUTPUT:**

AlexNet Architecture:

------------------------------------------------------------

AlexNet(

(features): Sequential(

(0): Conv2d(3, 64, kernel\_size=(11, 11), stride=(4, 4), padding=(2, 2))

(1): ReLU(inplace=True)

(2): MaxPool2d(kernel\_size=3, stride=2, padding=0, dilation=1, ceil\_mode=False)

(3): Conv2d(64, 192, kernel\_size=(5, 5), stride=(1, 1), padding=(2, 2))

(4): ReLU(inplace=True)

(5): MaxPool2d(kernel\_size=3, stride=2, padding=0, dilation=1, ceil\_mode=False)

(6): Conv2d(192, 384, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(7): ReLU(inplace=True)

(8): Conv2d(384, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(9): ReLU(inplace=True)

(10): Conv2d(256, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

(11): ReLU(inplace=True)

(12): MaxPool2d(kernel\_size=3, stride=2, padding=0, dilation=1, ceil\_mode=False)

)

(avgpool): AdaptiveAvgPool2d(output\_size=(6, 6))

(classifier): Sequential(

(0): Dropout(p=0.5, inplace=False)

(1): Linear(in\_features=9216, out\_features=4096, bias=True)

(2): ReLU(inplace=True)

(3): Dropout(p=0.5, inplace=False)

(4): Linear(in\_features=4096, out\_features=4096, bias=True)

(5): ReLU(inplace=True)

(6): Linear(in\_features=4096, out\_features=1000, bias=True)

)

)

A dog with bright eyes

Description automatically generated

**EXPERIMENT NO-09**

**AIM:**  Implementation of language Modelling using RNN

**DESCRIPTION:**

A recurrent neural network (RNN) is a type of artificial neural network which uses sequential data or time series data. These deep learning algorithms are commonly used for ordinal or temporal problems, such as language translation, natural language processing (NLP), speech recognition, and image captioning; they are incorporated into popular applications such as Siri, voice search, and Google Translate.

A diagram of a machine

Description automatically generated

**CODE:**

import tensorflow as tf import numpy as np import os

import time

path\_to\_file = tf.keras.utils.get\_file('shakespeare.txt', 'https://storage.googleapis.com/download.t ensorflow.org/data/shakespeare.txt')

#READ THE DATA

# Read, then decode for py2 compat.

text = open(path\_to\_file, 'rb').read().decode(encoding='utf-8') # length of text is the number of characters in it print(f'Length of text: {len(text)} characters')

Length of text: 1115394 characters

# Take a look at the first 250 characters in text print(text[:250])

First Citizen:

Before we proceed any further, hear me speak.

All:

Speak, speak.

First Citizen:

You are all resolved rather to die than to famish?

All:

Resolved. resolved.

First Citizen:

First, you know Caius Marcius is chief enemy to the people.

# The unique characters in the file vocab = sorted(set(text)) print(f'{len(vocab)} unique characters')

65 unique characters

#PROCESS THE TEXT

example\_texts = ['abcdefg', 'xyz']

chars = tf.strings.unicode\_split(example\_texts, input\_encoding='UTF-8') chars

<tf.RaggedTensor [[b'a', b'b', b'c', b'd', b'e', b'f', b'g'], [b'x', b'y', b'z']]>

ids\_from\_chars = tf.keras.layers.StringLookup( vocabulary=list(vocab), mask\_token=None)

ids = ids\_from\_chars(chars) ids

<tf.RaggedTensor [[40, 41, 42, 43, 44, 45, 46], [63, 64, 65]]>

chars\_from\_ids = tf.keras.layers.StringLookup( vocabulary=ids\_from\_chars.get\_vocabulary(), invert=True, mask\_token=None)

chars = chars\_from\_ids(ids) chars

<tf.RaggedTensor [[b'a', b'b', b'c', b'd', b'e', b'f', b'g'], [b'x', b'y', b'z']]>

tf.strings.reduce\_join(chars, axis=-1).numpy() def text\_from\_ids(ids):

return tf.strings.reduce\_join(chars\_from\_ids(ids), axis=-1)

#THE PREDICTION TASK

all\_ids = ids\_from\_chars(tf.strings.unicode\_split(text, 'UTF-8')) all\_ids

<tf.Tensor: shape=(1115394,), dtype=int64, numpy=array([19, 48, 57, ..., 46, 9, 1])>

ids\_dataset = tf.data.Dataset.from\_tensor\_slices(all\_ids) for ids in ids\_dataset.take(10):

print(chars\_from\_ids(ids).numpy().decode('utf-8'))

F

i r s t

C

i t

i

seq\_length = 100

sequences = ids\_dataset.batch(seq\_length+1, drop\_remainder=True)

for seq in sequences.take(1):

print(chars\_from\_ids(seq))

tf.Tensor(

[b'F' b'i' b'r' b's' b't' b' ' b'C' b'i' b't' b'i' b'z' b'e' b'n' b':'

b'\n' b'B' b'e' b'f' b'o' b'r' b'e' b' ' b'w' b'e' b' ' b'p' b'r' b'o'

b'c' b'e' b'e' b'd' b' ' b'a' b'n' b'y' b' ' b'f' b'u' b'r' b't' b'h'

b'e' b'r' b',' b' ' b'h' b'e' b'a' b'r' b' ' b'm' b'e' b' ' b's' b'p'

b'e' b'a' b'k' b'.' b'\n' b'\n' b'A' b'l' b'l' b':' b'\n' b'S' b'p' b'e'

b'a' b'k' b',' b' ' b's' b'p' b'e' b'a' b'k' b'.' b'\n' b'\n' b'F' b'i'

b'r' b's' b't' b' ' b'C' b'i' b't' b'i' b'z' b'e' b'n' b':' b'\n' b'Y' b'o' b'u' b' '], shape=(101,), dtype=string)

for seq in sequences.take(5): print(text\_from\_ids(seq).numpy())

b'First Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou '

b'are all resolved rather to die than to famish?\n\nAll:\nResolved. resolved.\n\nFirst Citizen:\nFirst, you k'

b"now Caius Marcius is chief enemy to the people.\n\nAll:\nWe know't, we know't.\n\nFirst Citizen:\nLet us ki"

b"ll him, and we'll have corn at our own price.\nIs't a verdict?\n\nAll:\nNo more talking on't; let it be d"

b'one: away, away!\n\nSecond Citizen:\nOne word, good citizens.\n\nFirst Citizen:\nWe are accounted poor citi'

def split\_input\_target(sequence): input\_text = sequence[:-1] target\_text = sequence[1:] return input\_text, target\_text

split\_input\_target(list("Tensorflow"))

(['T', 'e', 'n', 's', 'o', 'r', 'f', 'l', 'o'],

['e', 'n', 's', 'o', 'r', 'f', 'l', 'o', 'w'])

dataset = sequences.map(split\_input\_target)

for input\_example, target\_example in dataset.take(1): print("Input :", text\_from\_ids(input\_example).numpy()) print("Target:", text\_from\_ids(target\_example).numpy())

Input : b'First Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou'

Target: b'irst Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou '

#CREATE TRAINING BATCHES

# Batch size BATCH\_SIZE = 64

# Buffer size to shuffle the dataset

# (TF data is designed to work with possibly infinite sequences,

# so it doesn't attempt to shuffle the entire sequence in memory. Instead, # it maintains a buffer in which it shuffles elements).

BUFFER\_SIZE = 10000

dataset = ( dataset

.shuffle(BUFFER\_SIZE)

.batch(BATCH\_SIZE, drop\_remainder=True)

.prefetch(tf.data.experimental.AUTOTUNE))

dataset

<\_PrefetchDataset element\_spec=(TensorSpec(shape=(64, 100), dtype=tf.int64, name=None), TensorSpec(shape=(64, 100), dtype=tf.int64, name=None))>

#BUILD THE MODEL

# Length of the vocabulary in StringLookup Layer vocab\_size = len(ids\_from\_chars.get\_vocabulary())

# The embedding dimension embedding\_dim = 256

# Number of RNN units rnn\_units = 1024

class MyModel(tf.keras.Model):

def init (self, vocab\_size, embedding\_dim, rnn\_units): super(). init (self)

self.embedding = tf.keras.layers.Embedding(vocab\_size, embedding\_dim) self.gru = tf.keras.layers.GRU(rnn\_units,

return\_sequences=True, return\_state=True)

self.dense = tf.keras.layers.Dense(vocab\_size)

def call(self, inputs, states=None, return\_state=False, training=False): x = inputs

x = self.embedding(x, training=training) if states is None:

states = self.gru.get\_initial\_state(x)

x, states = self.gru(x, initial\_state=states, training=training) x = self.dense(x, training=training)

if return\_state:

return x, states else:

return x

model = MyModel( vocab\_size=vocab\_size, embedding\_dim=embedding\_dim, rnn\_units=rnn\_units)

#TRY THE MODEL

model.summary()

A screenshot of a computer

Description automatically generated

sampled\_indices = tf.random.categorical(example\_batch\_predictions[0], num\_samples=1) sampled\_indices = tf.squeeze(sampled\_indices, axis=-1).numpy()

sampled\_indices

A number grid with numbers

Description automatically generated

print("Input:\n", text\_from\_ids(input\_example\_batch[0]).numpy()) print()

print("Next Char Predictions:\n", text\_from\_ids(sampled\_indices).numpy())

Input:

b't certain\nTo miseries enough; no hope to help you,\nBut as you shake off one to take another;\nNothing'

Next Char Predictions:

b"MGpuvYjKxhI MTNc H.[UNK]3hJopmkMvBtX.QN:p3efGIl'-

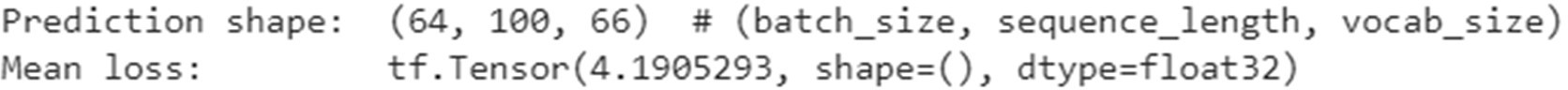
.[UNK]FYZwEl!Kvo!DrEwGNM[UNK]d.$LYzC IV:oI3PFz\ndqIfZ\npe'mFh3Fw$"

#ATTACH AN OPTIMIZER & A LOSS FUNCTION

loss = tf.losses.SparseCategoricalCrossentropy(from\_logits=True) example\_batch\_mean\_loss = loss(target\_example\_batch, example\_batch\_predictions)

print("Prediction shape: ", example\_batch\_predictions.shape, " # (batch\_size, sequence\_length, v ocab\_size)")

print("Mean loss: ", example\_batch\_mean\_loss)



tf.exp(example\_batch\_mean\_loss).numpy()

66.05775

model.compile(optimizer='adam', loss=loss)

#EXECUTE THE TRAINING

history = model.fit(dataset, epochs=20, callbacks=[checkpoint\_callback])

A screenshot of a computer

Description automatically generated

#GENERATE TEXT

class OneStep(tf.keras.Model):

def init (self, model, chars\_from\_ids, ids\_from\_chars, temperature=1.0): super(). init ()

self.temperature = temperature self.model = model self.chars\_from\_ids = chars\_from\_ids self.ids\_from\_chars = ids\_from\_chars

# Create a mask to prevent "[UNK]" from being generated. skip\_ids = self.ids\_from\_chars(['[UNK]'])[:, None] sparse\_mask = tf.SparseTensor(

# Put a -inf at each bad index. values=[-float('inf')]\*len(skip\_ids), indices=skip\_ids,

# Match the shape to the vocabulary dense\_shape=[len(ids\_from\_chars.get\_vocabulary())])

self.prediction\_mask = tf.sparse.to\_dense(sparse\_mask)

@tf.function

def generate\_one\_step(self, inputs, states=None):

# Convert strings to token IDs.

input\_chars = tf.strings.unicode\_split(inputs, 'UTF-8') input\_ids = self.ids\_from\_chars(input\_chars).to\_tensor()

# Run the model.

# predicted\_logits.shape is [batch, char, next\_char\_logits] predicted\_logits, states = self.model(inputs=input\_ids, states=states,

return\_state=True) # Only use the last prediction. predicted\_logits = predicted\_logits[:, -1, :]

predicted\_logits = predicted\_logits/self.temperature

# Apply the prediction mask: prevent "[UNK]" from being generated. predicted\_logits = predicted\_logits + self.prediction\_mask

# Sample the output logits to generate token IDs.

predicted\_ids = tf.random.categorical(predicted\_logits, num\_samples=1) predicted\_ids = tf.squeeze(predicted\_ids, axis=-1)

# Convert from token ids to characters predicted\_chars = self.chars\_from\_ids(predicted\_ids)

# Return the characters and model state. return predicted\_chars, states

one\_step\_model = OneStep(model, chars\_from\_ids, ids\_from\_chars) start = time.time()

states = None

next\_char = tf.constant(['ROMEO:']) result = [next\_char]

for n in range(1000):

next\_char, states = one\_step\_model.generate\_one\_step(next\_char, states=states) result.append(next\_char)

result = tf.strings.join(result) end = time.time()

print(result[0].numpy().decode('utf-8'), '\n\n' + '\_'\*80) print('\nRun time:', end - start)

A white text with black text

Description automatically generated

**EXPERIMENT NO-10**

**AIM:** Implementation of Encoder Decoder Models

**DESCRIPTION:**

The encoder-decoder model is a way of using recurrent neural networks for sequence-to- sequence prediction problems.

The overall structure of sequence-to-sequence model(encoder-decoder) which is commonly used is as shown below-

A diagram of a decoder

Description automatically generated

It consists of 3 parts: encoder, intermediate vector, and decoder.

Encoder-It accepts a single element of the input sequence at each time step, process it, collects information for that element and propagates it forward.

Intermediate vector- This is the final internal state produced from the encoder part of the model. It contains information about the entire input sequence to help the decoder make accurate predictions.

Decoder- given the entire sentence, it predicts an output at each time step

**CODE:**

import string import numpy as np

from keras.preprocessing.text import Tokenizer from keras.utils import pad\_sequences

from keras.models import Model

from keras.layers import LSTM, Input, TimeDistributed, Dense, Activation, RepeatVector, Emb edding

from keras.optimizers import Adam

from keras.losses import sparse\_categorical\_crossentropy

# Path to translation file

path\_to\_data = '/content/spa.txt'

# Read file

translation\_file = open(path\_to\_data,"r", encoding='utf-8') raw\_data = translation\_file.read()

translation\_file.close()

# Parse data

raw\_data = raw\_data.split('\n')

pairs = [sentence.split('\t') for sentence in raw\_data] pairs = pairs[1000:20000]

def clean\_sentence(sentence): # Lower case the sentence

lower\_case\_sent = sentence.lower() # Strip punctuation

string\_punctuation = string.punctuation + "¡" + '¿'

clean\_sentence = lower\_case\_sent.translate(str.maketrans('', '', string\_punctuation)) return clean\_sentence

def tokenize(sentences): # Create tokenizer

text\_tokenizer = Tokenizer() # Fit texts

text\_tokenizer.fit\_on\_texts(sentences)

return text\_tokenizer.texts\_to\_sequences(sentences), text\_tokenizer english\_sentences = [clean\_sentence(pair[0]) for pair in pairs]

spanish\_sentences = [clean\_sentence(pair[1]) for pair in pairs]

# Tokenize words

spa\_text\_tokenized, spa\_text\_tokenizer = tokenize(spanish\_sentences) eng\_text\_tokenized, eng\_text\_tokenizer = tokenize(english\_sentences)

print('Maximum length spanish sentence: {}'.format(len(max(spa\_text\_tokenized,key=len)))) print('Maximum length english sentence: {}'.format(len(max(eng\_text\_tokenized,key=len))))

# Check language length

spanish\_vocab = len(spa\_text\_tokenizer.word\_index) + 1 english\_vocab = len(eng\_text\_tokenizer.word\_index) + 1

print("Spanish vocabulary is of {} unique words".format(spanish\_vocab)) print("English vocabulary is of {} unique words".format(english\_vocab))

A black text on a white background

Description automatically generated

max\_spanish\_len = int(len(max(spa\_text\_tokenized,key=len))) max\_english\_len = int(len(max(eng\_text\_tokenized,key=len)))

spa\_pad\_sentence = pad\_sequences(spa\_text\_tokenized, max\_spanish\_len, padding = "post") eng\_pad\_sentence = pad\_sequences(eng\_text\_tokenized, max\_english\_len, padding = "post")

# Reshape data

spa\_pad\_sentence = spa\_pad\_sentence.reshape(\*spa\_pad\_sentence.shape, 1) eng\_pad\_sentence = eng\_pad\_sentence.reshape(\*eng\_pad\_sentence.shape, 1)

input\_sequence = Input(shape=(max\_spanish\_len,))

embedding = Embedding(input\_dim=spanish\_vocab, output\_dim=128,)(input\_sequence)

input\_sequence = Input(shape=(max\_spanish\_len,))

embedding = Embedding(input\_dim=spanish\_vocab, output\_dim=128,)(input\_sequence) encoder = LSTM(64, return\_sequences=False)(embedding)

input\_sequence = Input(shape=(max\_spanish\_len,))

embedding = Embedding(input\_dim=spanish\_vocab, output\_dim=128,)(input\_sequence) encoder = LSTM(64, return\_sequences=False)(embedding)

r\_vec = RepeatVector(max\_english\_len)(encoder)

input\_sequence = Input(shape=(max\_spanish\_len,))

embedding = Embedding(input\_dim=spanish\_vocab, output\_dim=128,)(input\_sequence) encoder = LSTM(64, return\_sequences=False)(embedding)

r\_vec = RepeatVector(max\_english\_len)(encoder)

decoder = LSTM(64, return\_sequences=True, dropout=0.2)(r\_vec)

input\_sequence = Input(shape=(max\_spanish\_len,))

embedding = Embedding(input\_dim=spanish\_vocab, output\_dim=128,)(input\_sequence) encoder = LSTM(64, return\_sequences=False)(embedding)

r\_vec = RepeatVector(max\_english\_len)(encoder)

decoder = LSTM(64, return\_sequences=True, dropout=0.2)(r\_vec) logits = TimeDistributed(Dense(english\_vocab))(decoder)

enc\_dec\_model = Model(input\_sequence, Activation('softmax')(logits)) enc\_dec\_model.compile(loss=sparse\_categorical\_crossentropy,

optimizer=Adam(1e-3), metrics=['accuracy'])

enc\_dec\_model.summary()

A screenshot of a computer program

Description automatically generated

model\_results = enc\_dec\_model.fit(spa\_pad\_sentence, eng\_pad\_sentence, batch\_size=30, epoch s=100)

A number of numbers on a white background

Description automatically generated

def logits\_to\_sentence(logits, tokenizer):

index\_to\_words = {idx: word for word, idx in tokenizer.word\_index.items()} index\_to\_words[0] = '<empty>'

return ' '.join([index\_to\_words[prediction] for prediction in np.argmax(logits, 1)]) index = 14

print("The english sentence is: {}".format(english\_sentences[index])) print("The spanish sentence is: {}".format(spanish\_sentences[index])) print('The predicted sentence is :')

print(logits\_to\_sentence(enc\_dec\_model.predict(spa\_pad\_sentence[index:index+1])[0], eng\_text

\_tokenizer))

A white background with black text

Description automatically generated