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.

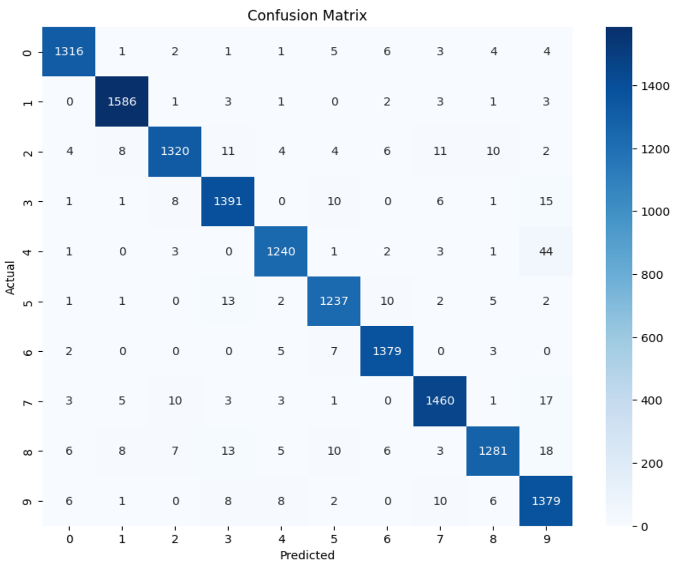
CODE:

```
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:

```
[[1316    1    2    1    1    5    6    3    4    4]
 [   0 1586    1    3    1    0    2    3    1    3]
 [   4    8 1320   11    4    4    6   11   10    2]
 [   1    1    8 1391    0   10    0    6    1   15]
 [   1    0    3    0 1240    1    2    3    1   44]
 [   1    1    0   13    2 1237   10    2    5    2]
 [   2    0    0    0    5    7 1379    0    3    0]
 [   3    5   10    3    3    1    0 1460    1   17]
 [   6    8    7   13    5   10    6    3 1281   18]
 [   6    1    0    8    8    2    0   10    6 1379]]
```

```
              precision    recall  f1-score   support

           0       0.98      0.98      0.98      1343
           1       0.98      0.99      0.99      1600
           2       0.98      0.96      0.97      1380
           3       0.96      0.97      0.97      1433
           4       0.98      0.96      0.97      1295
           5       0.97      0.97      0.97      1273
           6       0.98      0.99      0.98      1396
           7       0.97      0.97      0.97      1503
           8       0.98      0.94      0.96      1357
           9       0.93      0.97      0.95      1420

    accuracy                           0.97     14000
   macro avg       0.97      0.97      0.97     14000
weighted avg       0.97      0.97      0.97     14000
```



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:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
Epoch 1/5
1875/1875 [==============================] - 19s 8ms/step - loss: 0.2974 - accuracy: 0.9134
Epoch 2/5
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1443 - accuracy: 0.9569
Epoch 3/5
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1086 - accuracy: 0.9672
Epoch 4/5
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0890 - accuracy: 0.9725
Epoch 5/5
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0758 - accuracy: 0.9765
313/313 [==============================] - 1s 2ms/step - loss: 0.0753 - accuracy: 0.9759
Test accuracy: 0.9758999943733215
```

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:

```
Epoch 1/5
1875/1875 [==============================] - 10s 5ms/step - loss: 0.3001 - accuracy: 0.9117
Epoch 2/5
1875/1875 [==============================] - 9s 5ms/step - loss: 0.1459 - accuracy: 0.9572
Epoch 3/5
1875/1875 [==============================] - 9s 5ms/step - loss: 0.1088 - accuracy: 0.9668
Epoch 4/5
1875/1875 [==============================] - 10s 5ms/step - loss: 0.0872 - accuracy: 0.9729
Epoch 5/5
1875/1875 [==============================] - 10s 5ms/step - loss: 0.0755 - accuracy: 0.9764
313/313 [==============================] - 1s 2ms/step - loss: 0.0711 - accuracy: 0.9775
Test accuracy: 0.9775000214576721
```

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:

```
Result: 6.2
```

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:

```
Tensor t1:
 tensor([1, 2, 3, 4])

Tensor t2:
 tensor([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]])

Rank of t1:  1
Rank of t2:  2

Rank of t1:  torch.Size([4])
Rank of t2:  torch.Size([3, 4])
```

EXPERIMENT NO-03

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

DESCRIPTION:

Hyperparameters 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)
```

```python
print(f"Test Accuracy: {test_accuracy}")
```

OUTPUT:

```
Fitting 3 folds for each of 9 candidates, totalling 27 fits
Epoch 1/5
<ipython-input-6-19efe52dce9a>:33: DeprecationWarning: KerasClassifier is deprecated, us
  model = KerasClassifier(build_fn=create_model, epochs=5, batch_size=10)
8/8 [==============================] - 1s 3ms/step - loss: 1.1678 - accuracy: 0.3750
Epoch 2/5
8/8 [==============================] - 0s 3ms/step - loss: 1.0508 - accuracy: 0.3750
Epoch 3/5
8/8 [==============================] - 0s 3ms/step - loss: 0.9859 - accuracy: 0.4750
Epoch 4/5
8/8 [==============================] - 0s 3ms/step - loss: 0.9455 - accuracy: 0.5750
Epoch 5/5
8/8 [==============================] - 0s 3ms/step - loss: 0.9045 - accuracy: 0.4625
4/4 [==============================] - 0s 4ms/step - loss: 0.9448 - accuracy: 0.3250
Epoch 1/5
8/8 [==============================] - 1s 3ms/step - loss: 0.9528 - accuracy: 0.5250
Epoch 2/5
8/8 [==============================] - 0s 3ms/step - loss: 0.8014 - accuracy: 0.6000
Epoch 3/5
8/8 [==============================] - 0s 4ms/step - loss: 0.7350 - accuracy: 0.6875
Epoch 4/5
8/8 [==============================] - 0s 3ms/step - loss: 0.6871 - accuracy: 0.6875

Epoch 25/30
4/4 [==============================] - 0s 5ms/step - loss: 0.1116 - accuracy: 0.9667
Epoch 26/30
4/4 [==============================] - 0s 4ms/step - loss: 0.1169 - accuracy: 0.9500
Epoch 27/30
4/4 [==============================] - 0s 5ms/step - loss: 0.1473 - accuracy: 0.9500
Epoch 28/30
4/4 [==============================] - 0s 5ms/step - loss: 0.1810 - accuracy: 0.9250
Epoch 29/30
4/4 [==============================] - 0s 5ms/step - loss: 0.1561 - accuracy: 0.9583
Epoch 30/30
4/4 [==============================] - 0s 4ms/step - loss: 0.1788 - accuracy: 0.9250
1/1 [==============================] - 0s 163ms/step - loss: 0.1483 - accuracy: 0.9000
Test Accuracy: 0.8999999761581421
```

EXPERIMENT NO-04

AIM: Illustrate 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.

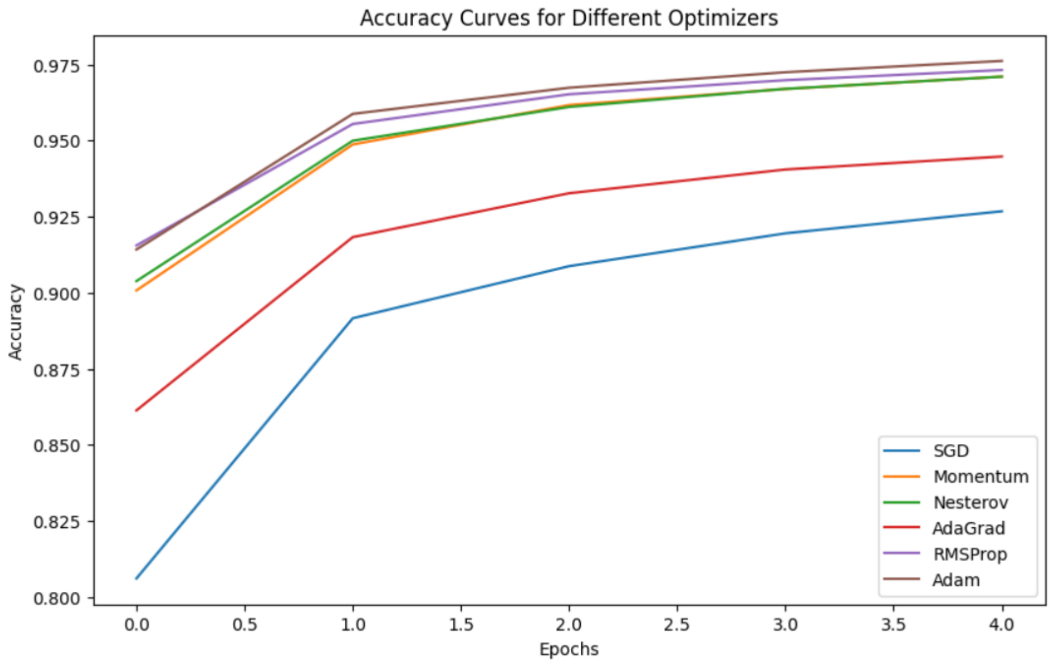
CODE:

```
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
```

```python
# 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:

```
Epoch 1/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.2946 - accuracy: 0.9155 - val_loss: 0.1481 - val_accuracy: 0.9560
Epoch 2/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.1538 - accuracy: 0.9554 - val_loss: 0.1081 - val_accuracy: 0.9682
Epoch 3/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.1209 - accuracy: 0.9652 - val_loss: 0.1011 - val_accuracy: 0.9716
Epoch 4/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.1052 - accuracy: 0.9698 - val_loss: 0.0893 - val_accuracy: 0.9750
Epoch 5/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.0935 - accuracy: 0.9732 - val_loss: 0.0981 - val_accuracy: 0.9755
Epoch 1/5
1875/1875 [==============================] - 10s 5ms/step - loss: 0.2962 - accuracy: 0.9142 - val_loss: 0.1422 - val_accuracy: 0.9580
Epoch 2/5
1875/1875 [==============================] - 9s 5ms/step - loss: 0.1439 - accuracy: 0.9588 - val_loss: 0.1095 - val_accuracy: 0.9671
Epoch 3/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.1097 - accuracy: 0.9674 - val_loss: 0.0883 - val_accuracy: 0.9734
Epoch 4/5
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0895 - accuracy: 0.9724 - val_loss: 0.0822 - val_accuracy: 0.9746
Epoch 5/5
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0766 - accuracy: 0.9761 - val_loss: 0.0806 - val_accuracy: 0.9739
```



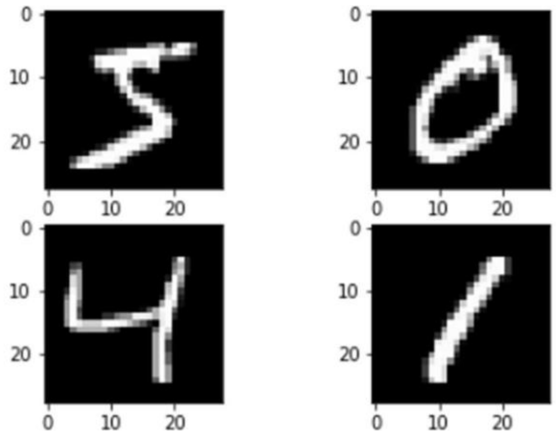
Accuracy Curves for Different Optimizers
Accuracy
Epochs
0.975
0.950
0.925
0.900
0.875
0.850
0.825
0.800
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
SGD
Momentum
Nesterov
AdaGrad
RMSProp
Adam

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)
```

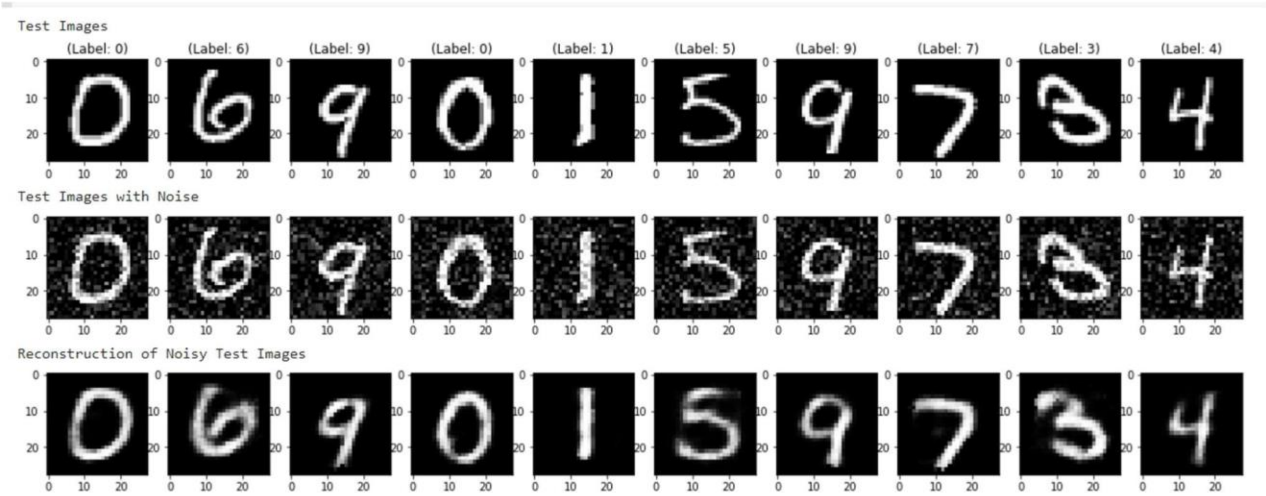
```
plt.imshow(X_train[1], cmap=plt.get_cmap('gray'))


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()
```



```
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:


Test Images
(Label: 0)
(Label: 6)
(Label: 9)
(Label: 0)
(Label: 1)
(Label: 5)
(Label: 9)
(Label: 7)
(Label: 3)
(Label: 4)
Test Images with Noise
Reconstruction of Noisy Test Images

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:

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum \|w\|^2$$

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:

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum \|w\|$$

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

```
array([[[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]]], dtype=uint8)
```

y_train[:5]

```
array([5, 0, 4, 1, 9], dtype=uint8)
```

X_train.shape

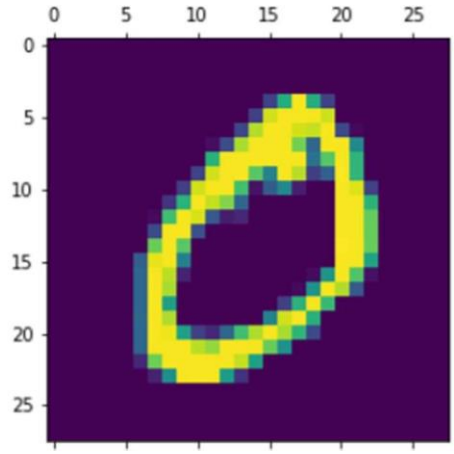
```
(60000, 28, 28)
```

X_train[0].shape

```
(28, 28)
```

```
plt.matshow(X_train[1])
```

```
<matplotlib.image.AxesImage at 0x7fb2a08df520>
```



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

```
(60000, 784)
```

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)

```
Epoch 1/5
1875/1875 [==============================] - 3s 1ms/step - loss: 0.4686 - accuracy: 0.8790
Epoch 2/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.3043 - accuracy: 0.9146
Epoch 3/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.2835 - accuracy: 0.9208
Epoch 4/5
1875/1875 [==============================] - 2s 1ms/step - loss: 0.2731 - accuracy: 0.9235
Epoch 5/5
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2663 - accuracy: 0.9256
<keras.callbacks.History at 0x7fb2a0f816d0>
```

model.evaluate(X_test_flat,y_test)

```
313/313 [==============================] - 0s 1ms/step - loss: 0.2708 - accuracy: 0.9229
[0.2707611918449402, 0.9229000210762024]
```

y_pred=model.predict(X_test_flat)

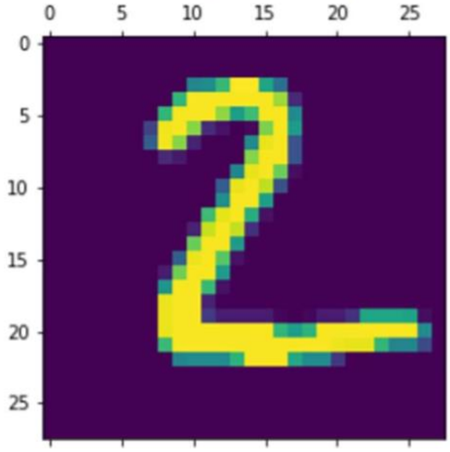
```
313/313 [==============================] - 0s 848us/step
```

y_pred[0]

```
array([1.9193865e-02, 5.7974785e-07, 3.9438169e-02, 9.5875973e-01,
       3.0014392e-03, 1.3237201e-01, 1.7716321e-06, 9.9978840e-01,
       1.5574734e-01, 6.4967054e-01], dtype=float32)
```

```
plt.matshow(X_test[1])
```

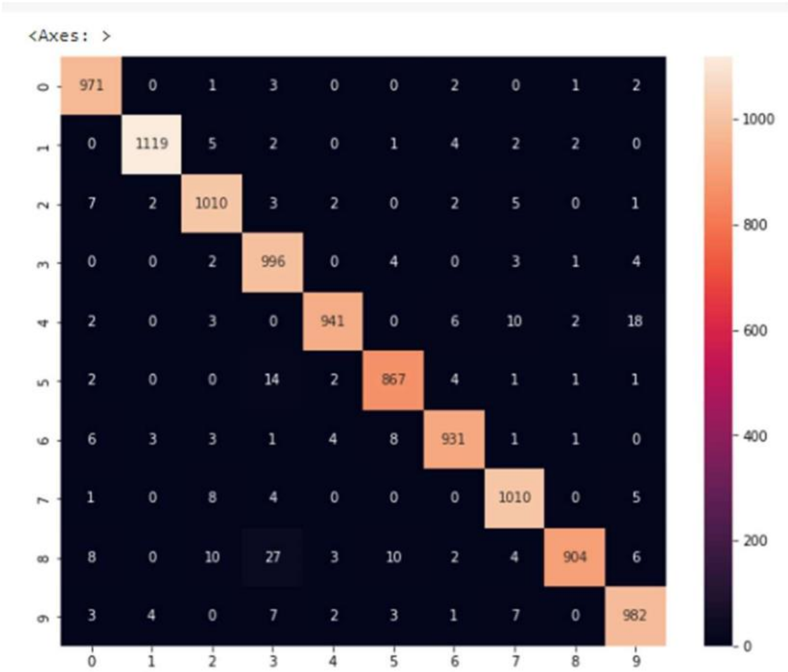
```
<matplotlib.image.AxesImage at 0x7fb27b72edf0>
```



```
np.argmax(y_pred[1])
```

2

```
y_pred_labels=[np.argmax(i) for i in y_pred]
cm=tf.math.confusion_matrix(labels=y_test,predictions=y_pred_la
bels)

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




```
from keras import regularizers model1=keras.Sequential([
keras.layers.Dense(100,input_shape=(784,),activation='relu',kern
el_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)
```

```
Epoch 1/5
1875/1875 [==============================] - 5s 2ms/step - loss: 0.2926 - accuracy: 0.9224
Epoch 2/5
1875/1875 [==============================] - 5s 2ms/step - loss: 0.1501 - accuracy: 0.9633
Epoch 3/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1196 - accuracy: 0.9730
Epoch 4/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.1048 - accuracy: 0.9779
Epoch 5/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0953 - accuracy: 0.9811
<keras.callbacks.History at 0x7fb26a66ffa0>
```

model1.evaluate(X_test_flat,y_test)

```
313/313 [==============================] - 1s 1ms/step - loss: 0.1096 - accuracy: 0.9757
[0.10955619812011719, 0.9757000207901001]
```

model2=keras.Sequential([
keras.layers.Dense(100,input_shape=(784,),activation='relu',kern
el_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)

```
Epoch 1/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.4520 - accuracy: 0.9159
Epoch 2/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2819 - accuracy: 0.9548
Epoch 3/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2368 - accuracy: 0.9654
Epoch 4/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2152 - accuracy: 0.9699
Epoch 5/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2012 - accuracy: 0.9720
<keras.callbacks.History at 0x7fb268d0c970>
```

model2.evaluate(X_test_flat,y_test)

```
313/313 [==============================] - 1s 1ms/step - loss: 0.2203 - accuracy: 0.9647
[0.22033147513866425, 0.9646999835968018]
```

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)
```

```
Epoch 1/5
1875/1875 [==============================] - 5s 2ms/step - loss: 0.3258 - accuracy: 0.9044
Epoch 2/5
1875/1875 [==============================] - 5s 2ms/step - loss: 0.1671 - accuracy: 0.9513
Epoch 3/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1305 - accuracy: 0.9617
Epoch 4/5
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1100 - accuracy: 0.9668
Epoch 5/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0952 - accuracy: 0.9706
<keras.callbacks.History at 0x7fb269bec9a0>
```

```
model3.evaluate(X_test_flat,y_test)
```

```
313/313 [==============================] - 1s 1ms/step - loss: 0.0837 - accuracy: 0.9734
[0.08370259404182434, 0.9733999967575073]
```

```
from keras.callbacks import EarlyStopping
model3.fit(X_train_flat,y_train,epochs=5,callbacks =
[EarlyStopping(monitor='val_acc', patience
=2)])
```

```
Epoch 1/5
1875/1875 [==============================] - ETA: 0s - loss: 0.0867 - accuracy: 0.9730WARNING:tensorflow:Early stopping
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0867 - accuracy: 0.9730
Epoch 2/5
1873/1875 [============================>.] - ETA: 0s - loss: 0.0783 - accuracy: 0.9745WARNING:tensorflow:Early stopping
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0782 - accuracy: 0.9745
Epoch 3/5
1861/1875 [============================>.] - ETA: 0s - loss: 0.0718 - accuracy: 0.9764WARNING:tensorflow:Early stopping
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0718 - accuracy: 0.9764
Epoch 4/5
1864/1875 [============================>.] - ETA: 0s - loss: 0.0685 - accuracy: 0.9782WARNING:tensorflow:Early stopping
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0684 - accuracy: 0.9782
Epoch 5/5
1872/1875 [============================>.] - ETA: 0s - loss: 0.0656 - accuracy: 0.9778WARNING:tensorflow:Early stopping
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0656 - accuracy: 0.9778
<keras.callbacks.History at 0x7fb269c0f8b0>
```

```
model3.evaluate(X_test_flat,y_test)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.0808 - accuracy: 0.9770
[0.0808185562491417, 0.9769999980926514]
```

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.

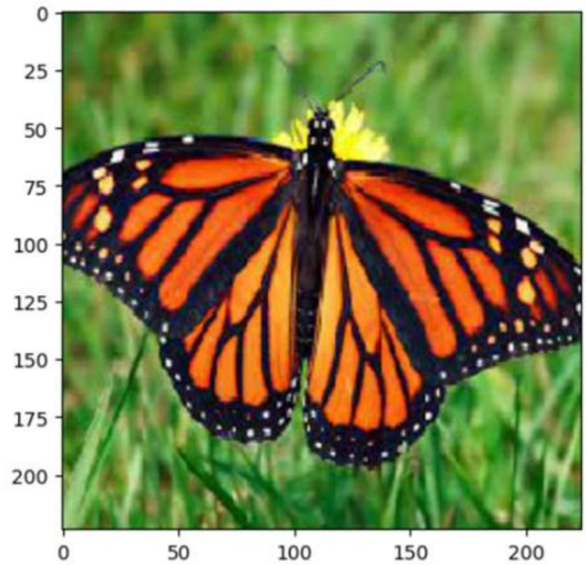
CODE:

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:

```
1/1 [==============================] - 1s 912ms/step
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/imagenet_class_index.json
35363/35363 [==============================] - 0s 0us/step
[[('n02279972', 'monarch', 0.9993399),
  ('n02281406', 'sulphur_butterfly', 0.00051474315),
  ('n02264363', 'lacewing', 9.5437914e-05)]]
```



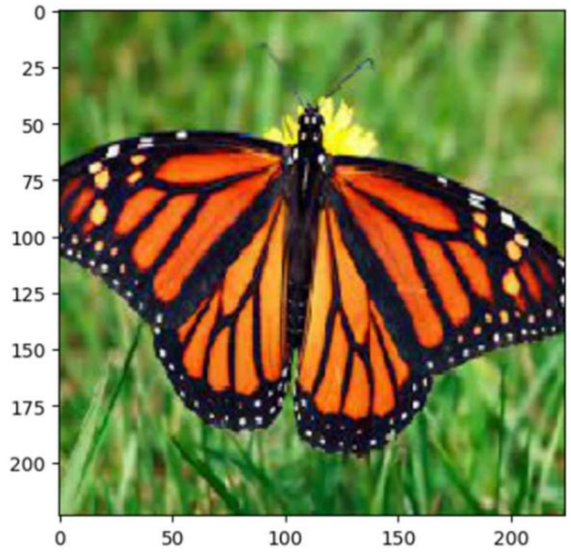
CODE:

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:

```
1/1 [==============================] - 1s 1s/step
[[('n02279972', 'monarch', 0.97757727),
  ('n02281406', 'sulphur_butterfly', 0.0059924624),
  ('n02277742', 'ringlet', 0.0056724995)]]
```



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.
```

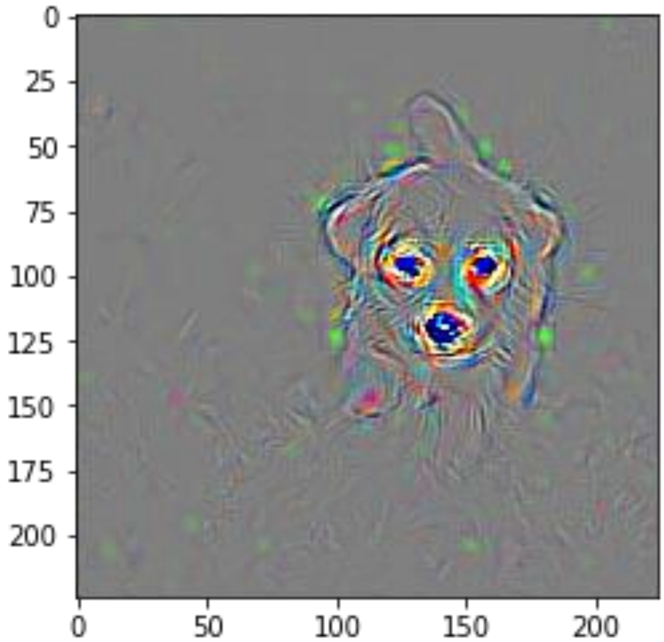
```
            grad[grad > 0] = 1

            # 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)
  )
)
```

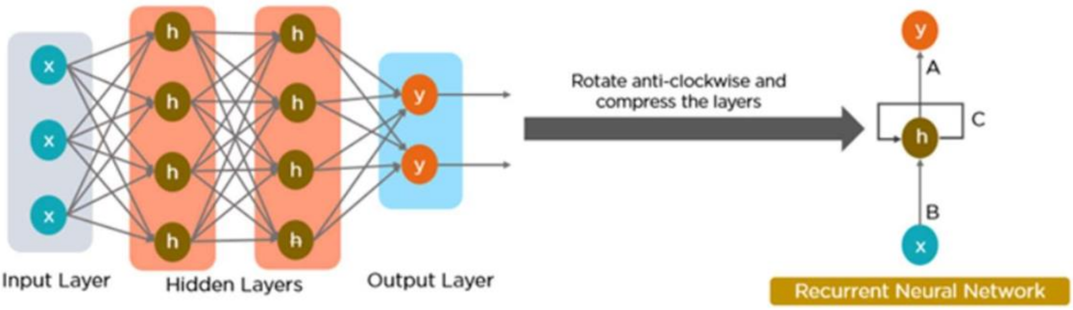


EXPERIMENT NO-09

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.



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()
```

```
Model: "my_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       multiple                  16896

 gru (GRU)                   multiple                  3938304

 dense (Dense)               multiple                  67650

=================================================================
Total params: 4,022,850
Trainable params: 4,022,850
Non-trainable params: 0
```

sampled_indices =
tf.random.categorical(example_batch_predictions[0],
num_samples=1) sampled_indices = tf.squeeze(sampled_indices,
axis=-1).numpy()

sampled_indices

```
array([26, 20, 55, 60, 61, 38, 49, 24, 63, 47, 22,  2, 26, 33, 27, 42,  2,
       21,  9,  0, 10, 47, 23, 54, 55, 52, 50, 26, 61, 15, 59, 37,  9, 30,
       27, 11, 55, 10, 44, 45, 20, 22, 51,  6,  8,  9,  0, 19, 38, 39, 62,
       18, 51,  3, 24, 61, 54,  3, 17, 57, 18, 62, 20, 27, 26,  0, 43,  9,
        4, 25, 38, 65, 16,  2, 22, 35, 11, 54, 22, 10, 29, 19, 65,  1, 43,
       56, 22, 45, 39,  1, 55, 44,  6, 52, 19, 47, 10, 19, 62,  4])
```

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)

```
Prediction shape:  (64, 100, 66)  # (batch_size, sequence_length, vocab_size)
Mean loss:         tf.Tensor(4.1905293, shape=(), dtype=float32)
```

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])

```
Epoch 1/20
172/172 [==============================] - 16s 63ms/step - loss: 2.6883
Epoch 2/20
172/172 [==============================] - 12s 60ms/step - loss: 1.9707
Epoch 3/20
172/172 [==============================] - 12s 57ms/step - loss: 1.6944
Epoch 4/20
172/172 [==============================] - 13s 59ms/step - loss: 1.5365
Epoch 5/20
172/172 [==============================] - 11s 57ms/step - loss: 1.4389
Epoch 6/20
172/172 [==============================] - 12s 59ms/step - loss: 1.3731
Epoch 7/20
172/172 [==============================] - 12s 58ms/step - loss: 1.3213
Epoch 8/20
172/172 [==============================] - 11s 57ms/step - loss: 1.2764
Epoch 9/20
172/172 [==============================] - 12s 57ms/step - loss: 1.2353
Epoch 10/20
172/172 [==============================] - 13s 58ms/step - loss: 1.1965
Epoch 11/20
172/172 [==============================] - 12s 58ms/step - loss: 1.1564
Epoch 12/20
172/172 [==============================] - 11s 56ms/step - loss: 1.1151
Epoch 13/20
172/172 [==============================] - 11s 57ms/step - loss: 1.0714
Epoch 14/20
172/172 [==============================] - 11s 57ms/step - loss: 1.0249
Epoch 15/20
172/172 [==============================] - 11s 57ms/step - loss: 0.9751
Epoch 16/20
172/172 [==============================] - 12s 60ms/step - loss: 0.9241
Epoch 17/20
172/172 [==============================] - 12s 58ms/step - loss: 0.8721
Epoch 18/20
172/172 [==============================] - 11s 57ms/step - loss: 0.8196
Epoch 19/20
172/172 [==============================] - 11s 57ms/step - loss: 0.7688
Epoch 20/20
172/172 [==============================] - 11s 57ms/step - loss: 0.7217
```

#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
```

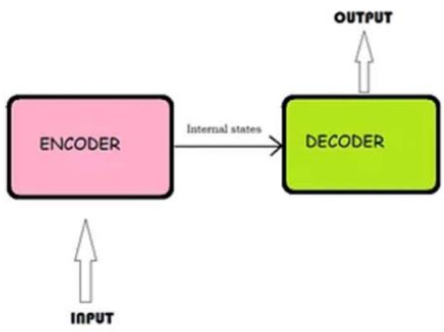
```
one_step_model = OneStep(model, chars_from_ids,
ids_from_chars) start = time.time()
states = None
next_char = tf.constant(['ROMEO:']) result = [next_char]

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)
```

```
ROMEO:
Come, well; we'll be he not with blow.
If the queen hath slain my kinning,
O, being myself, ourselves will have
Only men to draw our tongues it instance:
Yet I being but abidet, the heavens Took him;
For canst thou speak me father; for I'll weip, and wido:
And I, to look upon thy body, hate must call
Ky hangs live an opposite, or breath,
All in another's shrew and pule to do't.

BIANCA:
Tell her, Lord Hastings, lady: healthy hath got him mad.
Here's no tongue to purchase, or I here promised.
We must find into my dum.

HASTINGS:
So, so would me! 'tis give again to you.

CORIOLANUS:
Tut,
The lie, now--
Seens it is in France, of came?

MENENIUS:
Would I not have their goods best inconstant?

ADRIAN:
You shall not burden you; who it is smile,
I think, indeed, that so my state haste.

CORIOLANUS:
Like a most noble father of a father
With green ballad for a peril to myself
To London answer me. Go, say Kate, I pray thee, come unto
Verona bourning honour. Earth of many Margare
As king of Rich

________________________________________________________________________________

Run time: 3.1975724697113037
```

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-



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))
```

```
Maximum length spanish sentence: 9
Maximum length english sentence: 5
Spanish vocabulary is of 7230 unique words
English vocabulary is of 3724 unique words
```

```
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()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 9)]               0

 embedding_4 (Embedding)     (None, 9, 128)            925440

 lstm_4 (LSTM)               (None, 64)                49408

 repeat_vector_2 (RepeatVect  (None, 5, 64)            0
 or)

 lstm_5 (LSTM)               (None, 5, 64)             33024

 time_distributed (TimeDistr  (None, 5, 3724)          242060
 ibuted)

 activation (Activation)     (None, 5, 3724)           0

=================================================================
Total params: 1,249,932
Trainable params: 1,249,932
Non-trainable params: 0
```

```
model_results = enc_dec_model.fit(spa_pad_sentence,
eng_pad_sentence, batch_size=30, epoch s=100)
```

```
634/634 [==============================] - 22s 35ms/step - loss: 0.2581 - accuracy: 0.9214
Epoch 99/100
634/634 [==============================] - 23s 37ms/step - loss: 0.2583 - accuracy: 0.9208
Epoch 100/100
634/634 [==============================] - 25s 39ms/step - loss: 0.2558 - accuracy: 0.9218
```

```
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))
```

```
The english sentence is: stay away
The spanish sentence is: fuera
The predicted sentence is :
1/1 [==============================] - 1s 749ms/step
im away <empty> <empty> <empty>
```