INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
Dundigal - 500 043, Hyderabad, Telangana

DEEP LEARNING LABORATORY								
VII Semester: CSE (AI&ML)								
Course Code	**Category**	**Hours / Week**			**Credits**	**Maximum Marks**		
		L	**T**	**P**	**C**	**CIA**	**SEE**	**Total**
ACAD31	**Core**	0	0	2	1	40	60	100
Contact Classes: Nil	**Tutorial Classes: Nil**	**Practical Classes: 45**				**Total Classes: 45**		
Prerequisite: Python Programming								

I. COURSE OVERVIEW:

This course will introduce students will get practical implementations of various deep learning models using Python and the PyTorch library. Recommended lectures are: Machine Learning, and basic courses on Linear Algebra, Analysis, Probability & Statistics. While it is not a hard requirement, basic knowledge of Python will be greatly helpful.

II. COURSE OBJECTIVES:

The students will try to learn:

I. The theoretical foundations, algorithms and methodologies of Neural Network.
II. The design of single and multi-layer feed-forward deep networks and tune various hyper-parameters.
III. The provide the role of practical knowledge in handling and analyzing real world applications.
IV. To describe the role of neural networks in engineering, artificial intelligence, and cognitive modeling.

III. COURSE OUTCOMES:

At the end of the course students should be able to:

CO1 Understand the concepts of machine learning algorithms and deep neural networks for solving domain specific problems.
CO2 Select the concept of multi-layer perceptron and use the regularization technique to prevent the problem of model overfitting.
CO3 Make use of the activation functions and interpret how the performance of a CNN model varies during various classification tasks.
CO4 Develop the auto encoders and decoders to reconstruct the initial data from noisy data and visualize the inputs with encoded representations.
CO5 Construct a simple possible autoencoder and configure the model using different entropies and optimizers for dimensionality reduction.
CO6 Examine the recurrent neural networks and analyze how they predict the next possible outcomes for a given input.

DEEP NEURAL NETWORKS LABORATORY COURSE CONTENT

S.No	Topic Name	Page No.
1.	**Build a Machine Learning Model** a. Contextualize Machine Learning b. Explore the data and choose the type of algorithm c. Prepare and Clean the dataset. d. Split the prepared dataset and perform cross validation.	5
2.	**Build a Multi-Layer Perception** a. Load the HR dataset and perform data pre-processing like label encoding. b. Divide the dataset into training and testing to assess the model performance. c. Build a classification model using various parameters. d. Make prediction and evaluate the model.	10
3.	**Build a Feedforward Neural Network** a. Import the libraries to load IRIS dataset. b. Prepare the training and testing datasets and initialize weights. c. Build a back propagation model and create a loop by updating the weights in each iteration. d. Calculate the accuracy cum loss and plot the mean squared error and accuracy graphs.	14
4.	**Activation Functions** a. Implement the code to visualize the binary step activation function. b. Implement the code to visualize the linear activation function. c. Implement the code to visualize the sigmoid activation function. d. Implement the code to visualize the Tanh activation function. e. Implement the code to visualize the ReLU activation function. f. Implement the code to visualize the Softmax activation function.	20
5.	**Unsupervised Training of Neural Networks** a. Consider the case study of unsupervised deep learning on the MNIST dataset. b. Define the problem and organize a photo gallery. c. Arrange on the photos based on time and location. d. Extract semantic meaning from the images.	30
6.	**Auto Encoders** a. Import all the libraries and begin with simple auto encoder. b. Implement the code to develop a deep CNN auto encoder. c. Show how the model performs using denoising auto encoder.	34
7.	**Convolutional Neural Networks** a. Download and prepare the CIFAR10 dataset. b. Perform the dataset verifications and create the convolutional base. c. Add dense layers to the convolutional base. d. Compile, train and evaluate the model.	47
8.	**Fooling Neural Network**	51

a. Design and develop a neural network using two important parameters like weights and biases.

b. Implement the code for W has 10 rows of 784 elements each, where w[n] contains the weights for digits n.

c. Reshape the elements into 28 x 28 -pixel images each time we display them.

9. Performance Evaluation 55

a. Select any dataset and develop a CNN model.

b. Perform the image classification for binary datasets.

c. Evaluate the model performance using metrics like confusion matrix and

d. Submit the classification report by including accuracy and loss.

10. Stochastic Encoders and Decoders 59

a. Develop a simple and basic possible autoencoder.

b. Create a separate encoder and decoder model and reconstruct the MNIST digits.

c. Configure the model using binary cross entropy and Adam optimizer.

d. Visualize the reconstructed inputs and encoded representations.

11. Test Model 71

a. Perform the data preparation to fit for the model.

b. Create a RNN model which can take in the input sequence.

c. Train the model and check the accuracy on training data.

d. Check the loss on the validation data and perform the actual training.

12. Recognize Text In Images 77

a. Understand the memorization task to test the capabilities of LSTM.

b. Test the learned temporal dependency capability of LSTM.

c. Implement some arithmetic tasks to evaluate the interpretation capability of LSTM.

IV. COURSE CONTENT:

EXERCISES FOR FOUNDATIONS OF DEEP LEARNING LABORATORY

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

1. Getting Started Exercises

1.1 A Practical Introduction to Deep Neural Network

The Deep Neural Networks Laboratory offers a hands-on exploration of cutting-edge techniques in deep learning, utilizing powerful tools like Python, PyTorch, TensorFlow, and Keras. Throughout the exercises, students will gain practical experience in building, training, and optimizing neural networks, leveraging Python's versatile ecosystem for data manipulation and model development. PyTorch provides dynamic computation graphs and intuitive debugging, while TensorFlow, coupled with its Keras API, enables scalable and efficient deployment of deep learning models. This lab equips students with the foundational skills to implement advanced neural networks across various applications, bridging theoretical concepts with real-world practice.

Input: Iris dataset.

Output: Data frame filled with missing values, value zero or with any other value.

Explanation:

1. **Loading and Exploring the Dataset:**
    - The Iris dataset is loaded from scikit-learn.
    - A DataFrame is created for easier exploration and visualization of the data.
2. **Model Selection and Training:**
    - The dataset is split into training and testing sets.
    - A logistic regression model is trained on the data.
3. **Model Evaluation:**
    - The model's performance is evaluated using classification metrics such as accuracy, precision, recall, and F1-score.
4. **Interpreting Results:**
    - The model coefficients are printed to understand the influence of each feature on the predictions.

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...	...
145	146	6.7	3.0	5.2	2.3	Iris-virginica
146	147	6.3	2.5	5.0	1.9	Iris-virginica
147	148	6.5	3.0	5.2	2.0	Iris-virginica
148	149	6.2	3.4	5.4	2.3	Iris-virginica
149	150	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 6 columns

Hint:

```
# import necessary libraries
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, cross_val_score,
StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt
import seaborn as sns
")

#1. Contextualizing Machine learning using the IRIS dataset for Classification

# Load the IRIS Dataset

# Load the IRIS dataset
iris = load_iris()
X = pd.DataFrame(iris.data, columns=iris.feature_names)
y = pd.Series(iris.target, name='species')

# Description of the dataset
print(f"Dataset Description:\n{iris.DESCR}")
print(f"Features:\n{X.head()}")
print(f"Target:\n{y.head()}")
```

	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 5 columns

Try: Write the code using Boston Housing

1.2 Choose an algorithm

Explore the IRIS dataset to understand the distribution of the features (sepal length, sepal width, petal length, and petal width) and check for any patterns that differentiate the species. This helps in deciding that a classification algorithm is appropriate).

Input: Iris dataset

Output: I('The accuracy of the SVM using Petals is:', 0.97777777777777775)

('The accuracy of the SVM using Sepals is:', 0.80000000000000004

Explanation: Load the dataset first before loading the variables. Implement the code for Pearson Correlation and observe the results. If the value is near to 1 that means those two features are correlated and we can drop any one of them.

Hint:

```
# Data exploration
# Importing the libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

sns.pairplot(pd.concat([X, y], axis=1), hue='species')
plt.show()

# Correlation matrix
correlation = X.corr()
sns.heatmap(correlation, annot=True, cmap='coolwarm')
plt.title('Feature Correlation Matrix')
# Choosing a simple Decision Tree Classifier for classification
```

Try: Write the code to implement with Logistic Regression and Decision Tree

1.3 Prepare and clean the data.

For the IRIS dataset, this step is straightforward as it is already clean with no missing values or categorical variables. Standardizing or normalizing the numerical features might still be useful depending on the algorithm chosen.

Input: IRIS dataset.

Output: Normalized dataset ready for modeling.

Explanation:

The dataset is standardized to bring all features to a similar scale. A check for null values confirms the dataset is clean.

Hint:

```
# Preparing the data: Standardization
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# No missing values or categorical variables in the IRIS dataset
# Check for any null values (just as a good practice)
print(X.isnull().sum())
```

Try: Write the code to implement the one hot encoding and dump encoding.

1.4 Split the Prepared data set and perform cross-validation

Split the IRIS dataset into 80% training and 20% testing. Use Stratified K-Fold cross-validation with K=5 to ensure each fold has a balanced representation of all three species.

To train the model and next test the model we have to split the entire dataset into train and test sets. In that, the training dataset is used to train the model and the test dataset is to test the model which has been trained with the training dataset.

Import train_test_split to split the data into train and test datasets.

Input: IRIS dataset

Output: Rescaled dataset with outcomes proportionality.

Explanation: The dataset is split into training and testing sets. Cross-validation (Stratified K-Fold) is used to validate the model's performance

Hint:

```
# import the necessary libraries
import pandas as pd
import scipy
import numpy as np
from sklearn.preprocessing import MinMaxScaler
import seaborn as sns
import matplotlib.pyplot as plt

# Load the dataset
df = pd.read_csv('csv file path')
```

```
train, test = train_test_split(iris, test_size=0.3) # our main data split into
train and test
# the attribute test_size=0.3 splits the data into 70% and 30% ratio. train=70%
and test=30%
print(train.shape)
print(test.shape)

train_X = train[['SepalLengthCm','SepalWidthCm','PetalLengthCm','PetalWidthCm']] #
taking the training data features
train_y = train.Species # output of the training data

test_X = test[['SepalLengthCm','SepalWidthCm','PetalLengthCm','PetalWidthCm']] #
taking test data feature
test_y = test.Species  # output value of the test data

train_X.head()
```

Try: Apply Normalization and Standardization techniques on the above same dataset and generate the heat map.

1.5 Deploy the model

Deploying the model involves setting it up to make predictions on new data.

Input: IRS Dataset (csv file)

	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 5 columns

Output:

Explanation: The model is trained on the training set, evaluated on the test set, and saved for deployment. An example of loading the model and making a prediction is also included.

Hint:

```
# Train the model
model.fit(X_train, y_train)


# Evaluate the model on the test set
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)


print(f"Test Accuracy: {accuracy}")
```

```
print("Classification Report:")
print(classification_report(y_test, y_pred, target_names=iris.target_names))

# Save the model for deployment
import joblib

# Save the trained model
joblib.dump(model, 'iris_decision_tree_model.pkl')
print("Model saved as 'iris_decision_tree_model.pkl'.")

# Example of loading the model and making predictions
loaded_model = joblib.load('iris_decision_tree_model.pkl')
example_data = np.array([[5.1, 3.5, 1.4, 0.2]])  # Example input: features of an iris flower
example_data_scaled = scaler.transform(example_data)
prediction = loaded_model.predict(example_data_scaled)

print(f"Predicted species: {iris.target_names[prediction[0]]}")
```

Try: Output the training and testing values with Boston Housing dataset. Visualize the results with different plots.

2. Multilayer Perceptron Neural Network

2.1 Data Pre-Processing(Label Encoding)

Multi-Layer Perceptron (MLP) is the simplest type of artificial neural network. It is a combination of multiple perceptron models. Perceptrons are inspired by the human brain and try to simulate its functionality to solve problems. In MLP, these perceptrons are highly interconnected and parallel in nature. This parallelization helpful in faster computation. Perceptron was introduced by Frank Rosenblatt in 1950. It has the capability to learn complex things just like the human brain. Perceptron network consists of three units: Sensory Unit (Input Unit), Associator Unit (Hidden Unit), and Response Unit (Output Unit)

Input: HR Dataset.

Output:

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years	Departments	salary
0	0.38	0.53	2	157	3	0	1	0	sales	low
1	0.80	0.86	5	262	6	0	1	0	sales	medium
2	0.11	0.88	7	272	4	0	1	0	sales	medium
3	0.72	0.87	5	223	5	0	1	0	sales	low
4	0.37	0.52	2	159	3	0	1	0	sales	low

Explanation:

Lots of machine learning algorithms require numerical input data, so you need to represent categorical columns in a numerical column. In order to encode this data, you could map each value to a number. e.g. Salary column's value can be represented as low:0, medium:1, and high:2. This process is known as label encoding. In sklearn, we can do this using LabelEncode. Perform employee churn prediction using

Multi-Layer Perceptron. Employee churn prediction helps us in designing better employee retention plans and improving employee satisfaction.

Import the preprocessing module and create the Label Encoder object. By using this object fit and transform "salary" and "Departments" column into the numeric column.

Hint:

```
Import numpy as np
import pandas as pd

# Load the data
data = pd.reas_csv("HR_comma_sep.csv")
data.head()
```

```
# Import LabelEncoder
from sklearn import preprocessing


# Creating labelEncoder
le = preprocessing.LabelEncoder()


# Converting string labels into numbers.
data['salary']=le.fit_transform(data['salary'])
data['Departments ']=le.fit_transform(data['Departments '])
```

2.2 Split the dataset

In order to assess the model performance, we need to divide the dataset into a training set and test set. Let's split the dataset by using function train_test_split(). Pass basically 3 parameters features, target and test_set_size.

Input: HR Dataset

Output: Dataset with removed features whose variance doesn't meet the threshold.

Explanation:

Data reduction is also called dimensionality reduction. This reduces the size of data by encoding mechanisms. It can be lossy or lossless. If after reconstruction from compressed data, original data can be retrieved, such reduction is called lossless reduction else it is called lossy reduction. The two effective methods of dimensionality reduction are: Wavelet transforms and PCA (Principal Component Analysis).

PCA is sensitive to the relative scaling of the original variables.

Hint:

```
url="iris dataset file path'
# load dataset into Pandas Dataframe

df.head( )
```

	sepal length	sepal width	petal length	petal width	target
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Hint:

```
# Spliting data into Feature and

X=data[['satisfaction_level',          'last_evaluation',          'number_project',
'average_montly_hours',            'time_spend_company',           'Work_accident',
'promotion_last_5years', 'Departments ', 'salary']]

y=data['left']

# Import train_test_split function
from sklearn.model_selection import train_test_split

# Split dataset into training set and test set

X_train,   X_test,   y_train,   y_test   =   train_test_split(X,   y,   test_size=0.3,
random_state=42)  # 70% training and 30% test
```

Try: Imagine you have 1000 predictor features and 1 target feature in a machine learning problem. You must select the 100 most important features based on the relationship between input features and the target features. Develop the code to state whether the problem belongs to dimensionality reduction or not.

2.3 Build Classification Model using Parameters

Let's build an employee churn prediction model. Here, our objective is to predict churn using MLPClassifier.

First, import the MLPClassifier module and create MLP Classifier object using MLPClassifier() function. Then, fit your model on the train set using fit() and perform prediction on the test set using predict()

.

Input: HR dataset

Output: Different data plots

Explanation:

Parameters:

- **hidden_layer_sizes:** it is a tuple where each element represents one layer and its value represents the number of neurons on each hidden layer.
- **learning_rate_init:** It used to controls the step-size in updating the weights.
- **activation:** Activation function for the hidden layer. Examples, identity, logistic, tanh, and relu. by default, relu is used as an activation function.

- **random_state:** It defines the random number for weights and bias initialization.
- **verbose:** It used to print progress messages to standard output

Hint:

```
# Import MLPClassifer
from sklearn.neural_network import MLPClassifier

# Create model object
clf = MLPClassifier(hidden_layer_sizes=(6,5),
                    random_state=5,
                    verbose=True,
                    learning_rate_init=0.01)

# Fit data onto the model
clf.fit(X_train,y_train
```

Try:
Develop the code using other classifier and other parameters.

2.4 Make Prediction and Evaluate the Model

Make predictions on the test dataset and assess model accuracy based on available actual labels of the test dataset.

Input: HR dataset

Output: 0.9386666666666666

Hint:

```
# Make prediction on test dataset
ypred=clf.predict(X_test)

# Import accuracy score
from sklearn.metrics import accuracy_score

# Calcuate accuracy
accuracy_score(y_test,ypred)
```

Try:
Develop the code to plot the visualizations like histogram, scatter plot, line chart, bar chart and other interactive data visualizations for different datasets.

3. FEED-FORWARD NEURAL NETWORK

3.1 Load IRIS dataset for Feed-Forward Neural Network

An artificial feed-forward neural network (also known as multilayer perceptron) trained with backpropagation is an old machine learning technique that was developed in order to have

machines that can mimic the brain. Neural networks were the focus of a lot of machine learning research during the 1980s and early 1990s but declined in popularity during the late 1990s. Since 2010, neural networks have experienced a resurgence in popularity due to improvements in computing speed and the availability of massive amounts of data with which to train large-scale neural networks. In the real world, neural networks have been used to recognize speech, caption images, and even help self-driving cars learn how to park autonomously.

Input: IRIS dataset

Output:

Explanation: These lines import the necessary libraries and define the constants used throughout the code.
The normalize( ) function scales the dataset's attribute values to a range of 0 to 1. The get_five_stratified_folds( ) function creates five stratified folds for cross-validation.

Hint:

```
import pandas as pd  # Import Pandas library
import numpy as np  # Import Numpy library
from random import shuffle, seed, random  # Import shuffle(), seed(), and
random() from the random module
from collections import Counter  # Used for counting
from math import exp  # Import exp() function from the math module

# File name: neural_network.py
# Author: Addison Sears-Collins

ALGORITHM_NAME = "Feedforward Neural Network With Backpropagation"
SEPARATOR = ","  # Separator for the data set (e.g. "\t" for tab data)#   with
backpropagation (also called multilayer perceptron)
def normalize(dataset):
    # Generate a list of the column names
    column_names = list(dataset)

    # For every column except the actual class column
    for col in range(len(column_names) - 1):
        temp = dataset[column_names[col]]  # Go column by column
        minimum = temp.min()  # Get the minimum of the column
        maximum = temp.max()  # Get the maximum of the column

        # Normalize all values in the column so that they are between 0 and 1
        dataset[column_names[col]] = (dataset[column_names[col]] - minimum) /
(maximum - minimum)

    return dataset
```

Hint:

```
def get_five_stratified_folds(dataset):
    # Create five empty folds
    five_folds = [[] for _ in range(5)]

    # Get the number of columns in the data set
    class_column = len(dataset[0]) - 1

    # Shuffle the data randomly
    shuffle(dataset)

    # Generate a list of the unique class values and their counts
    classes = [instance[class_column] for instance in dataset]
    unique_classes = list(Counter(classes).keys())

    # Distribute instances among folds based on their class
    for unique_class in unique_classes:
        counter = 0
        for instance in dataset:
            if unique_class == instance[class_column]:
                five_folds[counter].append(instance)
                counter = (counter + 1) % 5

    # Shuffle each fold
    for fold in five_folds:
        shuffle(fold)

    return five_folds
```

Try: Write the code to import Soybean(small) dataset.

3.2 Prepare train and test datasets and initialize weights

The initialize_neural_net() function initializes the neural network with random weights. The train_neural_net() function trains the neural network over a set number of epochs
Input: IRIS dataset

Output: Datasets are divided into trained dataset to train the model and test dataset to evaluate the performance of the model using test dataset which the model has never seen before. By evaluating the model on the test set, ensure it has learnt to generalize the patterns.

Hint:

```
# Initialization
def weighted_sum_of_inputs(weights, inputs):
    return sum(w * inp for w, inp in zip(weights[:-1], inputs)) + weights[-1]
```

```
def sigmoid(x):
    return 1.0 / (1.0 + exp(-x))

def forward_propagate(network, instance):
    inputs = instance

    for layer in network:
        new_inputs = []
        for node in layer:
            weighted_sum = weighted_sum_of_inputs(node['weights'], inputs)
            node['output'] = sigmoid(weighted_sum)
            new_inputs.append(node['output'])
        inputs = new_inputs

    return inputs

# Train the neural network
def train_neural_net(network, training_set, learning_rate, no_epochs,
no_outputs):
    for epoch in range(no_epochs):
        for instance in training_set:
            outputs = forward_propagate(network, instance)
            actual_output = [0 for _ in range(no_outputs)]
            actual_output[int(instance[-1])] = 1
            back_propagate(network, actual_output)
            update_weights(network, instance, learning_rate)
```

Try: Write the implementation with soybean dataset.

3.3 Build a Back Propagation Model and update weights.

The back_propagate() function adjusts weights by calculating deltas (errors) using the backpropagation algorithm

Input: IRIS dataset

Output: Gradients for Layer1

Explanation:
The update_weights() function updates weights in the neural network using calculated deltas and a learning rate

Hint:

```
def sigmoid_derivative(output):
    return output * (1.0 - output)

def back_propagate(network, actual):
    # Iterate in reverse order (i.e. starts from the output layer)
    for i in reversed(range(len(network))):
```

```
        layer = network[i]
        errors = []

        if i != len(network) - 1:  # Hidden layer
            for j in range(len(layer)):
                error = sum(node['weights'][j] * node['delta'] for node in
network[i + 1])
                errors.append(error)
        else:  # Output layer
            for j, node in enumerate(layer):
                errors.append(actual[j] - node['output'])

        for j, node in enumerate(layer):
            node['delta'] = errors[j] * sigmoid_derivative(node['output'])
# update weights
def update_weights(network, instance, learning_rate):
    for layer_index, layer in enumerate(network):
        inputs = instance[:-1] if layer_index == 0 else [node['output'] for node
in network[layer_index - 1]]

        for node in layer:
            for j in range(len(inputs)):
                node['weights'][j] += learning_rate * node['delta'] * inputs[j]
            node['weights'][-1] += learning_rate * node['delta']
```

Try: Build a back propagation model for GLASS dataset.

3.4 Prediction and Accuracy Calculation

The predict_class() function predicts the class of a given instance, and calculate_accuracy() computes the accuracy of predictions

Input: IRIS dataset

Output: Predicted class labels.

Explanation:
Test Set predictions and Main functions.

Hint:

```
# Prediction
def predict_class(network, instance):
    outputs = forward_propagate(network, instance)
    return outputs.index(max(outputs))
# Calculate accuracy
def calculate_accuracy(actual, predicted):
    correct = sum(1 for a, p in zip(actual, predicted) if a == p)
    return (correct / len(actual)) * 100.0
```

```
def get_test_set_predictions(training_set, test_set, learning_rate, no_epochs,
no_hidden_layers, no_nodes_per_hidden_layer):
    no_inputs = len(training_set[0]) - 1
    no_outputs = len(set(instance[-1] for instance in training_set))

    network = initialize_neural_net(no_inputs, no_hidden_layers,
no_nodes_per_hidden_layer, no_outputs)
    train_neural_net(network, training_set, learning_rate, no_epochs,
no_outputs)

    predictions = [predict_class(network, instance) for instance in test_set]
    return network, predictions

# Main method
def main():
    LEARNING_RATE = 0.1
    NO_EPOCHS = 1000
    NO_HIDDEN_LAYERS = 1
    NO_NODES_PER_HIDDEN_LAYER = 8

    print("Welcome to the " + ALGORITHM_NAME + " Program!\n")

    data_path = "vote.txt"
    pd_data_set = pd.read_csv(data_path, sep=SEPARATOR)
    pd_data_set = normalize(pd_data_set)

    dataset_as_list = pd_data_set.values.tolist()
    seed(1)

    fv_folds = get_five_stratified_folds(dataset_as_list)
    scores = []

    for experiment_counter, fold in enumerate(fv_folds):
        print(f"\nRunning Experiment {experiment_counter}...\n")

        training_set = sum([f for i, f in enumerate(fv_folds) if i !=
experiment_counter], [])
        test_set = [list(instance) for instance in fold]

        neural_net, predicted_values = get_test_set_predictions(
            training_set, test_set, LEARNING_RATE, NO_EPOCHS,
            NO_HIDDEN_LAYERS, NO_NODES_PER_HIDDEN_LAYER)

        actual_values = [instance[-1] for instance in fold]
        accuracy = calculate_accuracy(actual_values, predicted_values)
        scores.append(accuracy)

        print(f"Experiment {experiment_counter} Trained Neural Network\n")
```

```
        for layer in neural_net:
            print(layer)
        print()

    print("Experiments Completed.\n")
    print(f"{ALGORITHM_NAME} Summary Statistics\n")
    print(f"Accuracy Statistics for All 5 Experiments: {scores}\n")
    print(f"Classification Accuracy: {sum(scores)/float(len(scores)):.3f}%")
```

Try: Calculate the loss and plot the mean squared error and accuracy graphs.

4. Activation Functions

4.1 Binary Step Activation Function

Without an activation function, a neural network is a linear regression model that doesn't perform complex tasks. It can learn and carry out more complex tasks by performing non-linear transformations.

Input: Generate values linearly spaced between -10 and 10. These values serve as the input to the activation function**.** If the node input value is less than 0, it returns 0 as output else it returns 1.

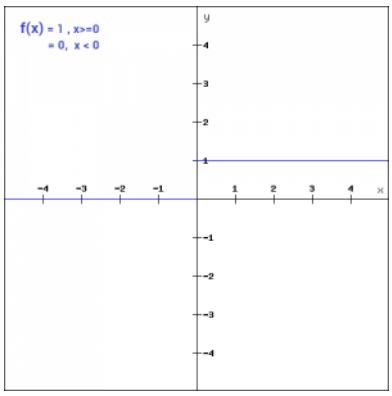
Output: (1,0)

Explanation:

The binary step function is an activation function used in some neural networks, and it can be defined as:

f(x) = 1, x>=0

= 0, x<0



Visualization: The results are plotted using 'matplotlib' to show how the binary step activation function works. The x-axis represents the input values, and the y-axis shows the output after applying the binary step function.

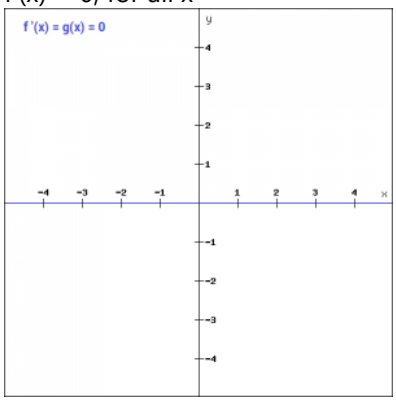
This is the simplest activation function, which can be implemented with a single if-else condition in python **Hint:**

```
#Import numpy and pyplot from matplotlib
import matplotlib.pyplot as plt
```

```
# Define the binary step activation function

def binary_step(x):
    if x<0:
        return 0
    else:
        return 1

binary_step(5), binary_step(-1)

# Generate input data
```

The binary step function can be used as an activation function while creating a binary classifier. As you can imagine, this function will not be useful when there are multiple classes in the target variable. That is one of the limitations of binary step function.
Moreover, the gradient of the step function is zero which causes a hindrance in the back propagation process. That is, if you calculate the derivative of f(x) with respect to x, it comes out to be 0.

f'(x) = 0, for all x



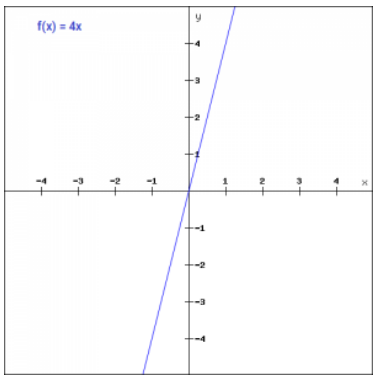
Hint:

```
f'(x) = 0, for all xprint("Mean Squared Error (MSE) = ", mse)
```

4.2 Linear Activation Function

We saw the problem with the step function, the gradient of the function became zero. This is because there is no component of x in the binary step function. Instead of a binary function, we can use a linear function. We can define the function as-

f(x)=ax



Input: ('x'): Here the activation is proportional to the input. The variable 'a' in this case can be any constant value. Let's quickly define the function in python.

Output: (16, -8).

Explanation:

What do you think will be the derivative is this case? When we differentiate the function with respect to x, the result is the coefficient of x, which is a constant.

Although the gradient here does not become zero, but it is a constant which does not depend upon the input value x at all. This implies that the weights and biases will be updated during the backpropagation process but the updating factor would be the same.

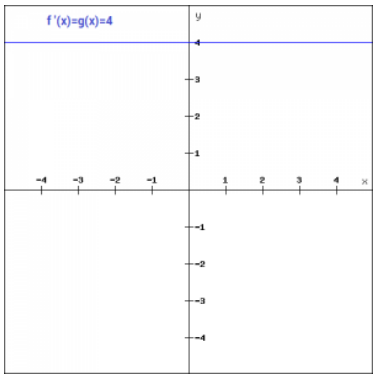
In this scenario, the neural network will not really improve the error since the gradient is the same for every iteration. The network will not be able to train well and capture the complex patterns from the data. Hence, linear function might be ideal for simple tasks where interpretability is highly desired.

Hint:

```
import numpy as np
import matplotlib.pyplot as plt

# define the linear activation function
def linear_function(x):
    return 4*x

linear_function(4), linear_function(-2)
```

Hint:

```
#Import The libraries


f'(x) = a
```

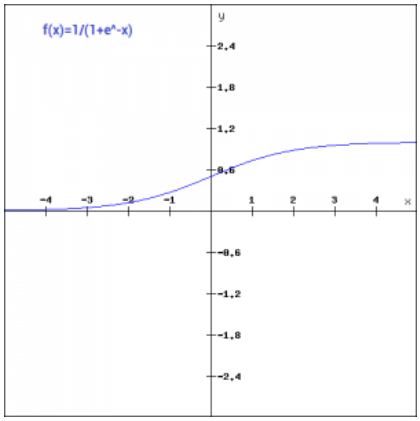


Try: Use the same code and apply on different datasets and do the comparative analysis for different performance metrics.

4.3 Sigmoid Activation Function

The next activation function in deep learning that we are going to look at is the Sigmoid activation function. It is one of the most widely used non-linear activation function. Sigmoid transforms the values between the range 0 and 1. Here is the mathematical expression for sigmoid-

$$f(x) = \frac{1}{1+e^{-x}}$$



Input: Here the activation is proportional to the input. The variable 'a' in this case can be any constant value. Represents the sigmoid function applied to input value "x"

Output: (0.9990889488055994, 2.7894680920908113e-10). The result is divided by 1 plus the exponential term, which ensures the output always falls between 0 and 1.

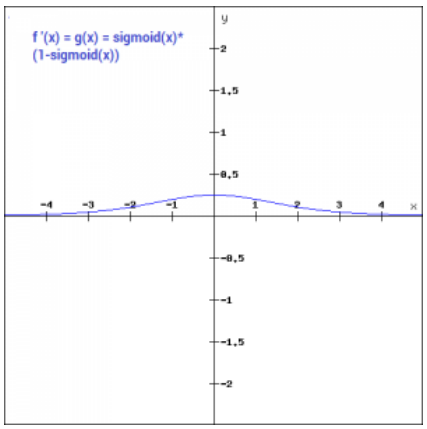
Explanation:

Unlike the binary step and linear functions, sigmoid is a non-linear function. multiple There are multiple neurons having sigmoid function as their activation function, the output is non linear as well. Here is the python code for defining the function in python

- σ(x): Represents the sigmoid function applied to input value "x".
- e^(-x): The exponential function with input "-x".
- 1 / (1 + ...): The result is divided by 1 plus the exponential term, which ensures the output always falls between 0 and 1.

Hint:

```
#Import libraries
import numpy as np

def sigmoid_function(x):
    z = (1/(1 + np.exp(-x)))
    return z

sigmoid_function(7),sigmoid_function(-22)
```



The gradient values are significant for range -3 and 3 but the graph gets much flatter in other regions. This implies that for values greater than 3 or less than -3, will have very small gradients. As the gradient value approaches zero, the network is not really learning.

The sigmoid function is not symmetric around zero. So output of all the neurons will be of the same sign. This can be addressed by scaling the sigmoid function which is exactly what happens in the tanh function.

4.4 Tanh Activation Function

The tanh function is very similar to the sigmoid function. The only difference is that it is symmetric around the origin. The range of values in this case is from -1 to 1. Thus the inputs to the next layers will not always be of the same sign. The tanh function is defined as-

The tanh activation function is defined as:

$$\tanh(x)=2\text{sigmoid}(2x)-1$$

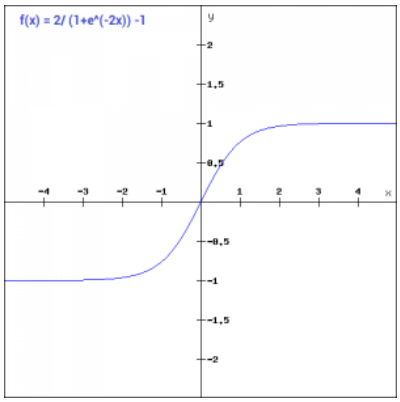
$$f(x) = \tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

Input: generate 400 values linearly spaced between -10 and 10 .

Output: Smooth 'S' shaped curve.

Explanation:

The results are plotted using 'matplotlib'. The x-axis represents the input values and the y-axis shows the output after applying the tanh activation function.

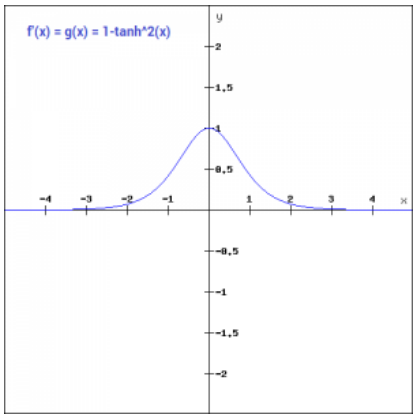


tanh(x) = 2/(1+e^(-2x)) -1

Hint:

```
# Import necessary modules
from numpy as np
from matplotlib.pyplot as plt

# define the tanh activation function
 def tanh_function(x):
    z = (2/(1 + np.exp(-2*x))) -1
    return z

# generate input data
 tanh_function(0.5), tanh_function(-1)
```

```
The range of values is between -1 to 1. Apart from that, all other properties of
tanh function are the same as that of the sigmoid function. Similar to sigmoid,
the tanh function is continuous and differentiable at all points.
```



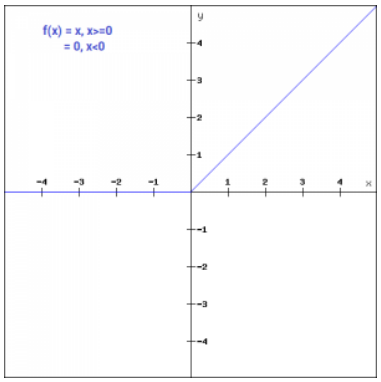
The gradient of the tanh function is steeper as compared to the sigmoid function. You might be wondering, how will we decide which activation function to choose? Usually tanh is preferred over the sigmoid function since it is zero centered and the gradients are not restricted to move in a certain direction.

4.5 RELU Activation Function

The ReLU Activation function is another non-linear activation function that has gained popularity in the deep learning domain. ReLU stands for Rectified Linear Unit. The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time.

This means that the neurons will only be deactivated if the output of the linear transformation is less than 0. The plot below will help you understand this better-

fx) = max(0,x)



Input: For the negative input values, the result is zero, that means the neuron does not get activated.

Output: (7, 0)

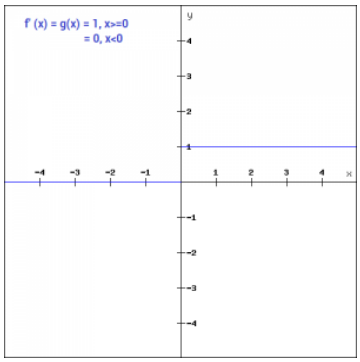
Explanation: Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh function. Here is the python function for ReLU.

The results are plotted using 'matplotlib'. The x-axis represents the input values, and the y-axis shows the output after applying the ReLU activation function.

The gradient of the ReLU function.

f'(x) = 1, x>=0

= 0, x<0



Hint:

```
#import necessary libraries
import numpy as np
import matplotlib.pyplot as plt

#define the ReLU activation function
  def relu_function(x):
    if x<0:
        return 0
    else:
        return x
# Apply the ReLU activation function to the input data
  relu_function(7), relu_function(-7)
```

Try: Instead of defining the Relu function as 0 for negative values of x, define it as an extremely small linear component of x**.** Write the python code for the same.

4.6 SOFTMAX Activation Function

The Softmax function is commonly used in the output layer of neural networks for classification tasks. Softmax function is often described as a combination of multiple sigmoids. We know that sigmoid returns values between 0 and 1, which can be treated as probabilities of a data point belonging to a particular class. Thus sigmoid is widely used for binary classification problems.
Mathematically, the Softmax function is defined as:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

Where z = [z1, z2, …..,zk] is the input vector.

- σ(z)i is the ith component of the output vector after applying Softmax.
- K is the number of classes or dimensions

Input: Generate a 3 x 3 matrix of input values representing three different sets of scores. Each row represents a set of logits (e.g from different classes).

Output: array([0.08021815, 0.11967141, 0.80011044])

Explanation: The function computes the exponential of each input value and normalizes the result to produce a probability distribution. The softmax function can be used for multiclass classification problems.
This function returns the probability for a datapoint belonging to each individual class. While building a network for a multiclass problem, the output layer would have as many neurons as the number of classes in the target. For instance if you have three classes, there would be three neurons in the output layer. Suppose you got the output from the neurons as [1.2 , 0.9 , 0.75].
Applying the softmax function over these values, you will get the following result – [0.42 , 0.31, 0.27]. These represent the probability for the data point belonging to each class.

The sum of all the values is 1.

Hint:

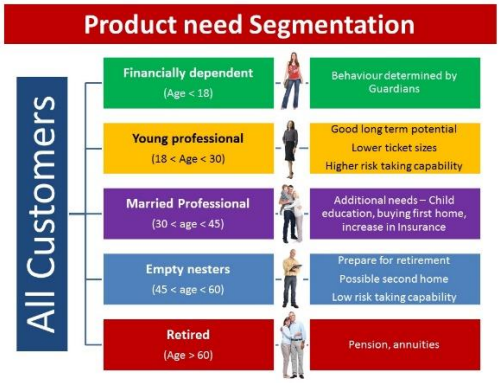
```
# import necessary libraries

import numpy as np
import matplotlib.pyplot as plt

#define the softmax activation function

def softmax_function(x):
    z = np.exp(x)
    z_ = z/z.sum()
    return z_
# Apply the Softmax activation function to the input data

softmax_function([0.8, 1.2, 3.1])

# Apply the Softmax activation function to the input data
   y = softmax_function(x)

# Vislualize the function
   for i in range(y.shape[0]):


      plt.show()
```

5. Unsupervised Training of Neural Networks

5.1 Case Study of Unsupervised Deep Learning

During my first project, I was working with a bank's marketing division. The director asked me to meet with him to talk about a data science project. I was eager to get started and was hoping to find a problem that would allow me to implement my skills and improve the customer's experience. The meeting started promptly. The director told me that the bank had a lot of data about its customers, but it didn't know how to use it to improve its operations. He wanted to use data science to enhance the company's marketing efforts.

Let's take an example to better understand this concept. Let's say a bank wants to divide its customers so that they can recommend the right products to them. They can do this in a data-driven way – by segmenting the customers on the basis of their ages and then deriving insights from these segments. This would help the bank give better product recommendations to their customers, thus increasing customer satisfaction.



Input: MNIST dataset.

Output: Figure out which of the images are similar to each other and cluster them into groups.

Explanation: An Auto Encoder with encoding set to 10 is trained on images of cats, each of size 100×100. So the input dimension is 10,000, and the Auto Encoder has to represent all this information in a vector of size 10.
Use a deep learning architecture called **Auto Encoders.** Train the algorithm to recreate what is learnt. Use a much smaller representation phase to recreate it.

```
# import all necessary libraries
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
from keras.models import Model
from keras.layers import Input, Dense
from keras import regularizers

# Load MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()

# Normalize the data to a range of 0 to 1
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

# Flatten the 28x28 images into vectors of size 784
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

input_dim = x_train.shape[1]

# Define the structure of the autoencoder
input_layer = Input(shape=(input_dim,))
encoded = Dense(128, activation='relu', activity_regularizer=regularizers.l1(10e-
5))(input_layer)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(32, activation='relu')(encoded)
decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
```

```
decoded = Dense(input_dim, activation='sigmoid')(decoded)

# Create the autoencoder model
autoencoder = Model(input_layer, decoded)

# Compile the model
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

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

Try: Write the code implementation by taking different datasets.

5.2 Problem Defining and Photo Gallery Organizing

Another **Problem Definition:**

The problem is to use unsupervised deep learning techniques to analyze and organize the images in the MNIST dataset. Specifically, we aim to:
Reduce the dimensionality of the data using techniques like autoencoders. Cluster the images based on similarities in the latent space. Reconstruct or generate new images based on learned features (optional).

Photo gallery Organization:

For organizing images, we can group them based on:
Clusters of Similar Digits: Use an unsupervised clustering algorithm, such as k-means, to group images of similar handwritten digits (e.g., group all '1's, '2's, etc., without using the labels).
Visual Embedding Space: Use techniques like t-SNE or PCA to project high-dimensional image data into 2D or 3D space for better visualization of clusters.
Input: X = np.array([[0, 1, 0, 1],
[1, 0, 1, 1],
[0, 0, 0, 1],
[1, 0, 1, 0]])
y = np.array([0, 1, 0, 1])

Output: Feature counts:
{0: array([0, 1, 0, 2]), 1: array([2, 0, 2, 1])}

Explanation: Here, the autoencoder has been trained and can now be used to visualize reconstructed images and their latent representations.

Hint:

```
# Visualize the MNIST dataset (gallery of images)
n = 10  # Number of digits to display
plt.figure(figsize=(20, 4))
for i in range(n):
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
    plt.title("Original")
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

Try: Produce the code to build Gaussian naive Bayes classifiers using Iris data.

5.3 Arrange the Photos based on Time and Location.

Though the MNIST dataset doesn't have time or location metadata, we can interpret this part in the following way: Time (Temporal Organization) and Location (Spatial organization).
Since MNIST lacks explicit time and location information, we can interpret the "location" as being in the latent space. Below, we use t-SNE to visualize the 2D representation of the encoded images in the latent space
Input: MNIST Dataset

Input: MNIST Dataset
Output:
Explanation:
Time (Temporal Organization): If we think of time in terms of training progression, we could organize the gallery based on how images evolve over the course of the training of a model. For example, autoencoders or GANs could be used to visualize how reconstruction improves with each training epoch.

Location (Spatial Organization): Location can refer to the latent space representation of the images. After reducing dimensionality using autoencoders or another technique, the latent vectors (which represent the compressed information of the images) can be plotted in a 2D or 3D space, effectively "arranging" them based on their location in this abstract, learned feature space.
Hint:

```
# Flatten the latent representations and apply t-SNE for visualization
from sklearn.manifold import TSNE

# Get the encoder part of the autoencoder to reduce the dimensionality
encoder = Model(input_layer, encoded)

# Encode the test images to extract latent features
x_test_encoded = encoder.predict(x_test)

# Use t-SNE to further reduce the latent features to 2D for visualization
x_test_tsne = TSNE(n_components=2).fit_transform(x_test_encoded)

# Plot the 2D t-SNE results
plt.scatter(x_test_tsne[:,            0],           x_test_tsne[:,            1],
c=np.argmax(autoencoder.predict(x_test), axis=1), cmap='tab10')
plt.colorbar()
plt.title("t-SNE visualization of the latent space")
plt.show()
```

5.4 Extract Semantic Meaning from the Images

Semantic Meaning Extraction:
In unsupervised learning, semantic meaning typically refers to the discovery of latent factors that explain the variations in the data. Here's how you could extract semantic meaning from the MNIST images:
Latent Space Representation:

Use an autoencoder to extract a compressed version of each image (the latent vector). These latent vectors can reveal the core features of each digit (e.g., certain patterns in strokes that differentiate a '1' from a '7')

Clustering for Semantic Groups: Once the latent space is created, apply clustering algorithms (e.g., k-means) to group similar images. Each cluster can be interpreted as containing images that share similar high-level features (e.g., all images that look like the digit '8' might form one cluster, but there may be sub-clusters for more rounded or less rounded '8's).

Dimensionality Reduction: Apply t-SNE or PCA on the latent space to visualize how different digits are organized. This gives a sense of the "semantic" differences between various digits in terms of stroke width, orientation, etc.
Feature Interpolation: In models like VAEs, you can interpolate between latent vectors to generate intermediate images. This process can reveal how the model understands the progression from one digit to another, effectively extracting semantic meaning from the transitions.

Input: MNIST Dataset

Output:
Explanation:
Explore the latent space to understand the semantics behind image clustering. Here, we'll use k-means clustering to group similar images.
Hint:

```
from sklearn.cluster import KMeans

# Apply k-means clustering on the latent space
# Visualizing the reconstructed images (semantic meaning)
decoded_imgs = autoencoder.predict(x_test)

n = 10  # Number of digits to display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original images
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
    plt.title("Original")
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstructed images
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
    plt.title("Reconstructed")
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

6. Auto Encoders

6.1 Simple Auto encoder

"Autoencoding" is a data compression algorithm where the compression and decompression functions are 1) data-specific, 2) lossy, and 3) learned automatically from examples rather than engineered by a human. Additionally, in almost all contexts where the term "autoencoder" is used, the compression and decompression functions are implemented with neural networks.

Input: A batch of images of size: torch.Size(N_batch, 1, 28,28)

N_batch represents the number of images in the batch.

Output: For each image, An accurate reconstruction of that image. Return N_batch images.

The output should be of shape torch.Size([N_batch, 1, 28, 28]).

Explanation:

Given an image as input, our model should reconstruct the same identical image.

- So, given an image I, we intend to minimise the distance between I and its reconstruction I' from the auto-encoder.
- In PyTorch, the loss torch.nn.MSELoss calculates the mean squared error between two tensors (in this case: between two batches images). This is the loss we will use to compare original images with their reconstructions.
- In the end, by minimising the mean squared error loss, we intend to provide more accurate reconstructions.

The __init__ method

- As before, here we define all our layers.
- We consider that both the encoder and the decoder have 3 Linear operations (with hidden layers of size 64)

```
# import necessary libraries
Import torch
class MNISTAutoEncoder(torch.nn.Module):
    def __init__(self, feature_space_dimensionality):
        super().__init__()
        self.linear_encoder_1 = torch.nn.Linear(in_features=1 * 28 * 28, out_features=64)
        self.linear_encoder_2 = torch.nn.Linear(in_features=64, out_features=64)
        self.linear_encoder_final = torch.nn.Linear(in_features=64, out_features=feature_space_dimensionality)

        self.linear_decoder_1 = torch.nn.Linear(in_features=feature_space_dimensionality, out_features=64)
        self.linear_decoder_2 = torch.nn.Linear(in_features=64, out_features=64)
        self.linear_decoder_final = torch.nn.Linear(in_features=64, out_features=1 * 28 * 28)
```

The forward method

There is one subtlety here. We said that our model should output reconstructions of the original images. As a consequence, the output should be of shape (N_batch, 1, 28, 28).

```
def forward(self, tensor_images):

        # Flattening images
        x = tensor_images.view(-1, 1 * 28 * 28)

        # Encoder --------------------

        # Encoder - Layer 1
        x = self.linear_encoder_1(x)
```

```
        x = torch.relu(x)

        # Encoder - Layer 2
        x = self.linear_encoder_2(x)
        x = torch.relu(x)

        # Encoder - Final Layer
        encoding = self.linear_encoder_final(x)
        # Note that there is no activation function here

        # Decoder --------------------

        # Decoder - Layer 1
        x = self.linear_decoder_1(encoding)
        x = torch.relu(x)

        # Decoder - Layer 2
        x = self.linear_decoder_2(x)
        x = torch.relu(x)

        # Decoder - Final Layer
        reconstruction = self.linear_decoder_final(x)



        # Putting reconstruction on the right shape (as original images)
        reconstruction = reconstruction.view(-1, 1, 28, 28)

        return reconstruction

transaction data = pd.read csv('transaction data.csv')
```

get_encoding

get_encoding simply corresponds to the encoder operations of the forward method.

```
def get_encoding(self, tensor_images):

        # Flattening images
        x = tensor_images.view(-1, 1 * 28 * 28)

        # Encoder --------------------

        # Encoder - Layer 1
        x = self.linear_encoder_1(x)
        x = torch.relu(x)

        # Encoder - Layer 2
        x = self.linear_encoder_2(x)
        x = torch.relu(x)

        # Encoder - Final Layer
        encoding = self.linear_encoder_final(x)
        # Note that there is no activation function here

        return encoding

transaction data = pd.read csv('transaction data.csv')
```

Test the Model

```
import torch

mnist_auto_encoder = MNISTAutoEncoder()
n_batch = 42
random_tensor = torch.randn(size=(n_batch, 1, 28, 28))
print(mnist_auto_encoder(random_tensor).size())

transaction data = pd.read csv('transaction data.csv')
```

Try: Write the code for the same using keras library.

6.2 Deep CNN Autoencoder

The images are of size 28 x 28 x 1 or a 784-dimensional vector. You convert the image matrix to an array, rescale it between 0 and 1, reshape it so that it's of size 28 x 28 x 1, and feed this as an input to the network.
Also, you will use a batch size of 128 using a higher batch size of 256 or 512 is also preferable it all depends on the system you train your model. It contributes heavily in determining the learning parameters and affects the prediction accuracy. You will train your network for 50 epochs.

Input: shape is (28, 28, 1) which corresponds to grayscale image of size 28x28.

Output:

Explanation:

Encoder : The first layer will have 32 filters of size 3 x 3, followed by a downsampling (max-pooling) layer,

- The second layer will have 64 filters of size 3 x 3, followed by another downsampling layer,
- The final layer of encoder will have 128 filters of size 3 x 3.

Decoder

- The first layer will have 128 filters of size 3 x 3 followed by a upsampling layer,/li>
- The second layer will have 64 filters of size 3 x 3 followed by another upsampling layer,
- The final layer of encoder will have 1 filter of size 3 x 3.

The max-pooling layer will downsample the input by two times each time you use it, while the upsampling layer will upsample the input by two times each time it is used.

Hint:

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D
from tensorflow.keras.models import Model

batch_size = 128
epochs = 50
inChannel = 1
x, y = 28, 28
input_img = Input(shape = (x, y, inChannel))

# Define decoder layers
def autoencoder(input_img):
```

```
    #encoder
    #input = 28 x 28 x 1 (wide and thin)
    conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img) #28 x
28 x 32
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1) #14 x 14 x 32
    conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(pool1) #14 x 14
x 64
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2) #7 x 7 x 64
    conv3 = Conv2D(128, (3, 3), activation='relu', padding='same')(pool2) #7 x 7 x
128 (small and thick)

    #decoder
    conv4 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv3) #7 x 7 x
128
    up1 = UpSampling2D((2,2))(conv4) # 14 x 14 x 128
    conv5 = Conv2D(64, (3, 3), activation='relu', padding='same')(up1) # 14 x 14 x
64
    up2 = UpSampling2D((2,2))(conv5) # 28 x 28 x 64
    decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(up2) # 28 x
28 x 1
    return decoded

autoencoder = Model(input_img, autoencoder(input_img))
autoencoder.compile(loss='mean_squared_error', optimizer = RMSprop())
```

Let's visualize the layers that you created in the above step by using the summary function, this will show number of parameters (weights and biases) in each layer and also the total parameters in your model.

```
    autoencoder.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 28, 28, 32)        320
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 14, 14, 64)        18496
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 7, 7, 128)         73856
_________________________________________________________________
conv2d_28 (Conv2D)           (None, 7, 7, 128)         147584
_________________________________________________________________
up_sampling2d_9 (UpSampling2 (None, 14, 14, 128)       0
_________________________________________________________________
conv2d_29 (Conv2D)           (None, 14, 14, 64)        73792
_________________________________________________________________
up_sampling2d_10 (UpSampling (None, 28, 28, 64)        0
_________________________________________________________________
```

```
conv2d_30 (Conv2D)           (None, 28, 28, 1)         577
=================================================================
Total params: 314,625
Trainable params: 314,625
Non-trainable params: 0
_________________________________________________________________
```

The model trains for 50 epochs. The fit() function will return a history object; By storying the result of this function in fashion_train, you can use it later to plot the loss function plot between training and validation which will help you to analyze your model's performance visually.

Train the model.

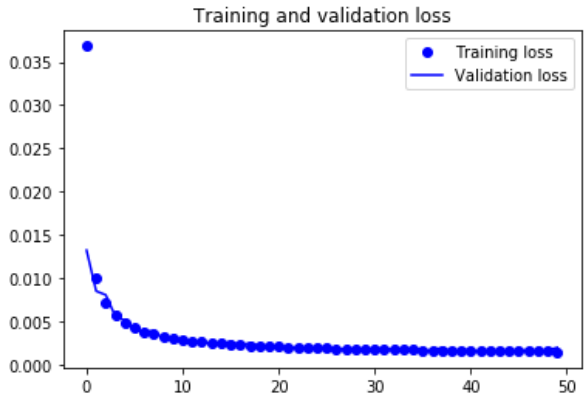
```
autoencoder_train = autoencoder.fit(train_X, train_ground, batch_size =batch_size, epochs=epochs, verbose=1,validation_data=(valid_X, valid_ground))
```

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/50
48000/48000 [==============================] - 16s - loss: 0.0368 - val_loss: 0.0132
Epoch 2/50
48000/48000 [==============================] - 15s - loss: 0.0101 - val_loss: 0.0085
Epoch 3/50
48000/48000 [==============================] - 15s - loss: 0.0071 - val_loss: 0.0081
Epoch 4/50
48000/48000 [==============================] - 15s - loss: 0.0057 - val_loss: 0.0056
Epoch 5/50
48000/48000 [==============================] - 15s - loss: 0.0048 - val_loss: 0.0051
Epoch 6/50
48000/48000 [==============================] - 15s - loss: 0.0043 - val_loss: 0.0039
Epoch 7/50
48000/48000 [==============================] - 15s - loss: 0.0038 - val_loss: 0.0039
Epoch 8/50
48000/48000 [==============================] - 15s - loss: 0.0035 - val_loss: 0.0039
Epoch 9/50
48000/48000 [==============================] - 15s - loss: 0.0032 - val_loss: 0.0030
Epoch 10/50
48000/48000 [==============================] - 15s - loss: 0.0030 - val_loss: 0.0029
Epoch 11/50
48000/48000 [==============================] - 15s - loss: 0.0029 - val_loss: 0.0026
Epoch 12/50
48000/48000 [==============================] - 15s - loss: 0.0027 - val_loss: 0.0025
Epoch 13/50
48000/48000 [==============================] - 15s - loss: 0.0026 - val_loss: 0.0028
Epoch 14/50
48000/48000 [==============================] - 15s - loss: 0.0025 - val_loss: 0.0022
Epoch 15/50
48000/48000 [==============================] - 15s - loss: 0.0024 - val_loss: 0.0024
Epoch 16/50
48000/48000 [==============================] - 16s - loss: 0.0023 - val_loss: 0.0027
Epoch 17/50
48000/48000 [==============================] - 15s - loss: 0.0023 - val_loss: 0.0022
```

```
Epoch 18/50
48000/48000 [==============================] - 16s - loss: 0.0022 - val_loss: 0.0025
Epoch 19/50
48000/48000 [==============================] - 16s - loss: 0.0022 - val_loss: 0.0022
Epoch 20/50
48000/48000 [==============================] - 15s - loss: 0.0021 - val_loss: 0.0022
Epoch 21/50
48000/48000 [==============================] - 16s - loss: 0.0021 - val_loss: 0.0020
Epoch 22/50
48000/48000 [==============================] - 16s - loss: 0.0020 - val_loss: 0.0019
Epoch 23/50
48000/48000 [==============================] - 16s - loss: 0.0020 - val_loss: 0.0021
Epoch 24/50
48000/48000 [==============================] - 16s - loss: 0.0020 - val_loss: 0.0018
Epoch 25/50
48000/48000 [==============================] - 16s - loss: 0.0019 - val_loss: 0.0020
Epoch 26/50
48000/48000 [==============================] - 16s - loss: 0.0019 - val_loss: 0.0020
Epoch 27/50
48000/48000 [==============================] - 16s - loss: 0.0019 - val_loss: 0.0017
Epoch 28/50
48000/48000 [==============================] - 16s - loss: 0.0018 - val_loss: 0.0018
Epoch 29/50
48000/48000 [==============================] - 16s - loss: 0.0018 - val_loss: 0.0019
Epoch 30/50
48000/48000 [==============================] - 16s - loss: 0.0018 - val_loss: 0.0017
Epoch 31/50
48000/48000 [==============================] - 16s - loss: 0.0018 - val_loss: 0.0019
Epoch 32/50
48000/48000 [==============================] - 16s - loss: 0.0017 - val_loss: 0.0018
Epoch 33/50
48000/48000 [==============================] - 16s - loss: 0.0017 - val_loss: 0.0017
Epoch 34/50
48000/48000 [==============================] - 15s - loss: 0.0017 - val_loss: 0.0018
Epoch 35/50
48000/48000 [==============================] - 15s - loss: 0.0017 - val_loss: 0.0019
Epoch 36/50
48000/48000 [==============================] - 15s - loss: 0.0017 - val_loss: 0.0016
Epoch 37/50
48000/48000 [==============================] - 15s - loss: 0.0017 - val_loss: 0.0017
Epoch 38/50
48000/48000 [==============================] - 15s - loss: 0.0016 - val_loss: 0.0019
Epoch 39/50
48000/48000 [==============================] - 15s - loss: 0.0016 - val_loss: 0.0015
Epoch 40/50
48000/48000 [==============================] - 15s - loss: 0.0016 - val_loss: 0.0017
Epoch 41/50
48000/48000 [==============================] - 15s - loss: 0.0016 - val_loss: 0.0017
Epoch 42/50
48000/48000 [==============================] - 15s - loss: 0.0016 - val_loss: 0.0014
Epoch 43/50
48000/48000 [==============================] - 15s - loss: 0.0016 - val_loss: 0.0018
```

```
Epoch 44/50
48000/48000 [==============================] - 15s - loss: 0.0016 - val_loss: 0.0015
Epoch 45/50
48000/48000 [==============================] - 15s - loss: 0.0016 - val_loss: 0.0014
Epoch 46/50
48000/48000 [==============================] - 16s - loss: 0.0015 - val_loss: 0.0016
Epoch 47/50
48000/48000 [==============================] - 16s - loss: 0.0015 - val_loss: 0.0017
Epoch 48/50
48000/48000 [==============================] - 16s - loss: 0.0015 - val_loss: 0.0015
Epoch 49/50
48000/48000 [==============================] - 16s - loss: 0.0015 - val_loss: 0.0016
Epoch 50/50
48000/48000 [==============================] - 16s - loss: 0.0015 - val_loss: 0.0020
```

```
loss = autoencoder_train.history['loss']
val_loss = autoencoder_train.history['val_loss']
epochs = range(epochs)
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```



the validation loss and the training loss both are in sync. It shows that your model is not overfitting: the validation loss is decreasing and not increasing, and there rarely any gap between training and validation loss.

Therefore, you can say that your model's generalization capability is good.

Finally, it's time to reconstruct the test images using the predict() function of Keras and see how well your model is able reconstruct on the test data.

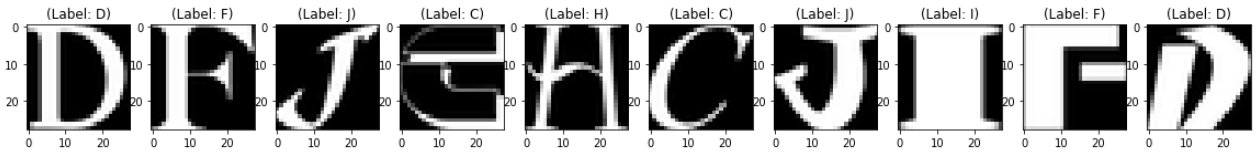
Predicting on Test Data

You will be predicting the trained model on the complete 10,000 test images and plot few of the reconstructed images to visualize how well your model is able to reconstruct the test images.

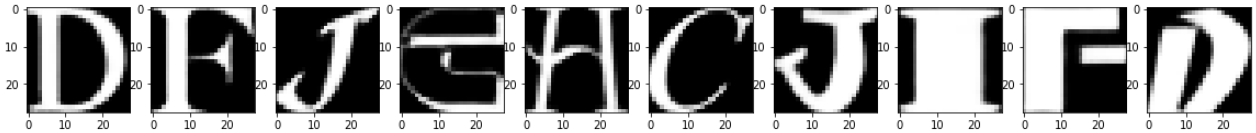
```
pred = autoencoder. predict(test_data)
pred.shape
(10000, 28, 28, 1)
```

```
plt.figure(figsize=(20, 4))
print("Test Images")
for i in range(10):
    plt.subplot(2, 10, i+1)
    plt.imshow(test_data[i, ..., 0], cmap='gray')
    curr_lbl = test_labels[i]
    plt.title("(Label: " + str(label_dict[curr_lbl]) + ")")
plt.show()
plt.figure(figsize=(20, 4))
print("Reconstruction of Test Images")
for i in range(10):
    plt.subplot(2, 10, i+1)
    plt.imshow(pred[i, ..., 0], cmap='gray')
plt.show()
```

Test Images



Reconstruction of Test Images



Try:

6.3 Denoising Autoencoder

A denoising autoencoder tries to learn a representation (latent-space or bottleneck) that is robust to noise. You add noise to an image and then feed the noisy image as an input to the enooder part of your network. The encoder part of the autoencoder transforms the image into a different space that tries to preserve the alphabets but removes the noise.

During training, you define a loss function, similar to the root mean squared error that you had defined earlier in convolutional autoencoder. At every iteration of the training, the network will compute a loss between the noisy image outputted by the decoder and the ground truth (denoisy image) and will also try to minimize that loss or difference between the reconstructed image and the original noise-free image. In other words, the network will learn a 7 x 7 x 128 space that will be noise free encodings of the data that you will train your network on.

Input: MNIST dataset

Output: Removal of noise from the images.

Explanation:

Let's first define a noise factor which is a hyperparameter. The noise factor is multiplied with a random matrix that has a mean of 0.0 and standard deviation of 1.0. This matrix will draw samples from normal (Gaussian) distribution. The shape of the random normal array will be similar to the shape of the data you will be adding the noise.

For simplicity, let's understand it with an example: The variable train_X has a shape of 48000 x 28 x 28 x 1. Hence, the random normal array will also have a shape similar to train_X only then you will be able to add two arrays since then they will have same dimensions.

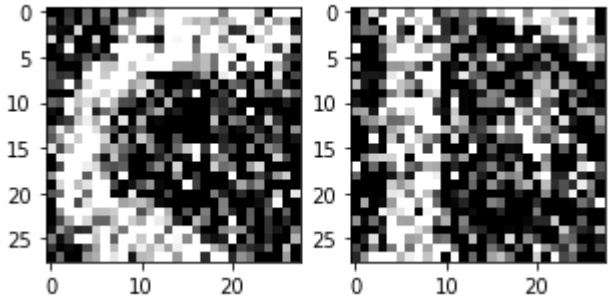
Hint:

```
noise_factor = 0.5
x_train_noisy = train_X + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=train_X.shape)
x_valid_noisy = valid_X + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=valid_X.shape)
x_test_noisy = test_data + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=test_data.shape)
x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_valid_noisy = np.clip(x_valid_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)encoder layer3 = Conv2D(8, (3, 3),
activation='relu', padding='same'))
```

np.clip() will threshold all the negative values to zero and all the values greater than one to one. Since, you want your pixel values to be between zero and one. And there are chances that after introducing noise into the data the pixel values range can change a bit, so to be on the safer side it's a good practice to clip the pixel values.

Visualizing the noisy images

```
plt.figure(figsize=[5,5])

# Display the first image in training data
plt.subplot(121)
curr_img = np.reshape(x_train_noisy[1], (28,28))
plt.imshow(curr_img, cmap='gray')

# Display the first image in testing data
plt.subplot(122)
curr_img = np.reshape(x_test_noisy[1], (28,28))
plt.imshow(curr_img, cmap='gray')

<matplotlib.image.AxesImage at 0x7f3ec6b20e48>
```



Now that you have the noisy data, you can feed it into the network and see how the noise is "magically" removed from the images!

Denoising Autoencoder Network

The autoencoder is divided into two parts: the encoder and the decoder. The architecture that you're going to construct will look like the following:

Encoder

- The first layer will have 32-3 x 3 filters followed by a downsampling (max-pooling) layer,
- The second layer will have 64-3 x 3 filters followed by another downsampling layer,
- The final layer of encoder will have 128-3 x 3 filters.

Decoder

- The first layer will have 128-3 x 3 filters followed by a upsampling layer,
- The second layer will have 64-3 x 3 filters followed by another upsampling layer,
- The final layer of encoder will have 1-3 x 3 filters.

Hint:

```
batch_size = 128
epochs = 20
inChannel = 1
x, y = 28, 28
input_img = Input(shape = (x, y, inChannel))
```

Hint:

```
def autoencoder(input_img):
    #encoder
    conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
    conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(pool1)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
    conv3 = Conv2D(128, (3, 3), activation='relu', padding='same')(pool2)


#decoder

    conv4 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv3)
    up1 = UpSampling2D((2,2))(conv4)
    conv5 = Conv2D(64, (3, 3), activation='relu', padding='same')(up1)
    up2 = UpSampling2D((2,2))(conv5)
    decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(up2)
    return decodedfalse positives + false negatives)



autoencoder = Model(input_img, autoencoder(input_img))
autoencoder.compile(loss='mean_squared_error', optimizer = RMSprop())
```

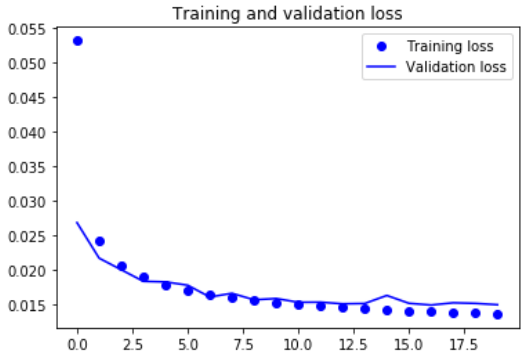
Training:

If you remember while training the convolutional autoencoder, you had fed the training images twice since the input and the ground truth were both same. However, in denoising autoencoder, you feed the noisy images as an input while your ground truth remains the denoisy images on which you had applied the noise. Only then the network will be able to compute a loss between the noisy and the true denoisy images.

```
autoencoder_train = autoencoder.fit(x_train_noisy, train_X,
batch_size=batch_size,epochs=epochs,verbose=1,validation_data=(x_valid_noisy,
valid_X))
```

Train versus Validation Loss Plot

```
loss = autoencoder_train.history['loss']
val_loss = autoencoder_train.history['val_loss']
epochs = range(epochs)
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```




```
# F1 score

F1_score = 2 * (precision * recall) / (precision + recall)
```

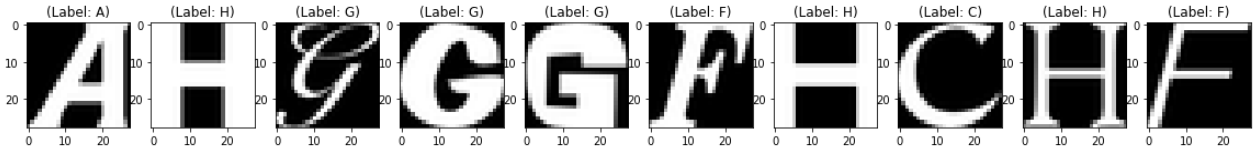
From the above plot, you can derive some intuition that the model is overfitting at some epochs while being in sync for most of the time. You can definitely try to improve the performance of the model by introducing some complexity into it so that the loss can reduce more, try training it for more epochs and then decide.

Predicting on Test dataset

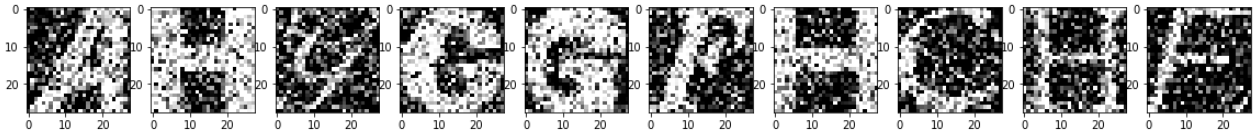
```
pred = autoencoder.predict(x_test_noisy)



plt.figure(figsize=(20, 4))
print("Test Images")
for i in range(10,20,1):
    plt.subplot(2, 10, i+1)
    plt.imshow(test_data[i, ..., 0], cmap='gray')
```

```
    curr_lbl = test_labels[i]

    plt.title("(Label: " + str(label_dict[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, ..., 0], 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, ..., 0], cmap='gray')
plt.show()
```

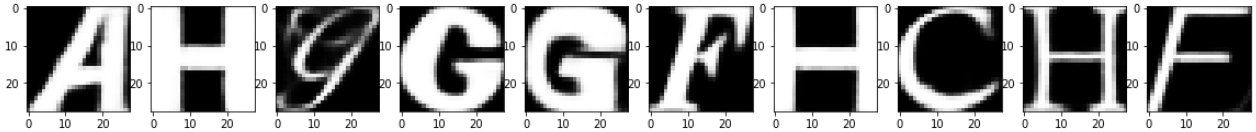
Test Images



Test Images with Noise



Reconstruction of Noisy Test Images



Both It look like that, within just 20 epoch, a fairly simple network architecture removed the noise from the test images.

Try: Implement the code for at least three different datasets and compare the results.

7. Convolutional Neural Networks

7.1 Preparation of CIFAR10 Dataset.

This demonstrates the training of a simple Convolutional Neural Network(CNN) to classify CIFAR (CIFAR -10 and CIFAR-100)are labeled subsets of 80 million of the tiny images dataset) images. This uses the Keras Sequential API, creating and training your model.

Input: CIFAR-10 dataset

Download and prepare the CIFAR-10 dataset
The CIFAR-10 dataset contains 60,000 color images in 10 classes, with 6,000 images in each class. The dataset is divided into 50,000 training images and 10,000 testing images. The classes are mutually exclusive and there is no overlap between them.

Output: Dataset is prepared

Hint:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
# Normalize pixel values to be between 0 and 1.

train_images, test_images = train_images/255.0, test_images /255.0
#Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
```

7.2 Data Verification

Let's define the convolutional base of the model using three 2D convolutional layers with ReLU activation functions, followed by max pooling layers. The convolutional layer has 32 filters, each with a size of 3x3.

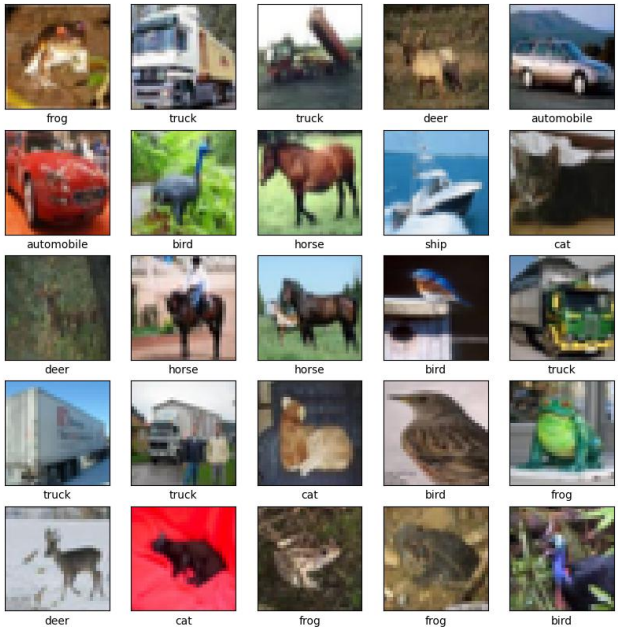
The second convolutional layer has 64 filters, also with a size of 3x3. The third convolutional layer has 64 filters, with a size of 3x3. The max pooling layers have a pool size of 2x2.

To verify that the dataset looks correct, let's plot the first 25 images from the training set and display the class name below each image.

Input: Image of size 32x32x3
Output: A feature map of size 4x4x64.
Hint:

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

# Perform the dataset split process and apply the DecisionTreeClassifier with a random state 0.

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplots(5, 5, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i])
```

```
    # The CIFAR labels happen to be arrays for extra index.
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```



Try: Produce a code to with varying imaging size and number of images.

7.3 Creating the Convolutional base

Define the convolutional base using a common pattern: a stack of Conv2D and MaxPooling2D layers.
Input: Tensors of shape(image_height, image_width, color_channels), ignoring the batch size.

Output: Output of every Conv2D and MaxPooling2D layer should be a 3D tensor of shape (height, width, channels). The width and height dimensions tend to shrink as you go deeper in the network.

Explanation:

Configure CNN to process inputs of shape (32, 32, 3), which is the format of CIFAR images. Pass the argument input_shape to first_layer. The number of output channels for each Conv2D layer is controlled by the first argument (e.g, 32 or 64). Typically, as the width and height shrink, add more output channels in each Conv2D layer.

Hint:

```
# Importing Modules

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation = ‘relu’, input_shape = (32, 32,
3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation = ‘relu’))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation = ‘relu’))

#display the architecture of the model
model.summary()

Model: “sequential”
```

```
Total params: 56, 320 (220.00 KB)

Trainable params: 56, 320(220.00 KB)

Non-trainable params: 0(0.00 B)
```

7.4 Add Dense Layers on Top

To complete the model, the last output tensor from the convolutional base (of shape (4, 4, 64)) is fed into one or more Dense layers to perform classification. Flatten or unroll the 3D output to 1D, then add one or more Dense layers on top. CIFAR has 10 output cases, use a final Dense layer with 10 outputs.

Input: Vectors (1D)

Output: The network summary should be showed as (4, 4, 64) outputs are flattened into vectors of shape (1024) before going through two Dense layers.

Explanation:
Adding Dense layers on top of a Convolutional Neural Network (CNN) serves to integrate the high-level features extracted by the convolutional layers and make final predictions. Dense layers, being fully connected, help in mapping these features to the output space, which can be useful for classification or regression tasks. Essentially, they act as the decision-making layers of the network, enabling the CNN to learn complex relationships in the data.

Hint:

```
# importing libraries
model.add(layers.Flatten())
model.add(layers.Dense(64, activation = ‘relu’))
model.add(layers.Dense(10))

model.summary()

model.”sequential”

Total params: 122, 570(478.79 KB)

Trainable params: 122,570(478.79 KB)

Non-trainable params: 0.00 KB
```

7.5 Compile and Train the Model

In K-Means, we do what is called “hard labeling”, where we simply add the label of the maximum probability. However, certain data points that exist at the boundary of clusters may simply have similar probabilities of being on either cluster. In such circumstances, we look at all the probabilities instead of the max probability. This is known as “soft labeling”. The goal of this exercise is to implement the clustering technique to perform the ‘hard labelling’.

Input: CIFAR-10 images

Output: Visualize the different clusters with different colors.

Hint:

```
Import tensorflow as tf
from tensorflow.keras import datasets, layers, models
```

```
#Load and preprocess the CIFAR-10 dataset
(X_train, y_train), (X_test, y_test) = datasets.cifar10.load_data()


# Normalize the pixel values to be between 0 and 1.

X_train, X_test = X_train/255.0, X_test/255.0

#Define the CNN model
model = models.Sequential([layers.Conv2D(32, (3, 3), activation = 'relu',
input_shape=(32, 32, 3)),
layers.MaxPooling2D((2, 2)),
layers.Conv2D(64, (3, 3)), activation = 'relu'),
layers.MaxPooling2D(2, 2)),
layers.Flatten(),
layers.Dense(64, activation = 'relu'),
layers.Dense(64, activation = 'softmax')
])

# Compile the model

model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy', metrics =
['accuracy'])

# Train the model
history = model.fit(X_train, y_train, epochs = 10, validation_data = (X_test, y_test))


# Evaluate the model
test_loss, test_acc = model.evaluate(X_test, y_test, verbose = 2)
print(f"Test accuracy: {test_acc:.4f}")
```

Explanation:

The Model definition: The model is built using a 'sequential ' API. It starts with three convolutional layers ('Conv2D'), each followed by a max-pooling layer ('MaxPooling2D') to reduce the spatial dimensions.

The Flatten layer converts the 2D feature maps to a 1D vector, which is then fed in to the Dense layers.

The first Dense layer with 64 neurons and ReLU activation acts as a fully connected layer to learn complex patterns. The final Dense layer with 10 neurons and softmax activation produces the output probabilities for each of the classes.

Model Compilation**:** The model is compiled using Adam optimizer, sparse_categorical_crossentropy as the loss function, and accuracy as the metric.

Model Training: The model is trained for 10 epochs using the training data and validated on the test data after each epoch.

8. Fooling Neural Network

8.1 Designing and developing a neural network

Neural Networks are like the workhorses of deep learning. With enough data and computational power, they can be used to solve most of the problems in deep learning, speech recognition, and more.
Let us design and develop a neural network using two parameters, weights and biases.

Input: 4 Samples with three feature values, Value 1: Represent the presence or absence of a feature (0 or 1).

Value 2: In similar nature
Value 3: a random value

Output: A set of corresponding outputs that the network is trained to predict.
A probability (e.g., the likelihood of a binary classification).
A decision boundary (e.g., if the output is greater than 0.5, classify as 1; otherwise, classify as 0

Explanation:
Weights and Biases Initialization: Random values are assigned to weights and biases.
Forward Propagation: Input passes through the network, and activations are computed using the sigmoid function.

Backward Propagation: Errors are computed, and weights and biases are updated using the derivatives of the activation function.

Training: The training loop adjusts weights and biases over multiple epochs.

Hint:

```
import numpy as np

#Define activation functions and their derivatives

  def sigmoid(x):
    return 1/(1 + np.exp(-x))

  def sigmoid_derivative(x):
    return x * (1 - x)

   # Initialize parameters
  Input_size = 3    # Number of input neurons
  Input_size = 4    # Number of neurons in the hidden layer.
  Input_size = 1    # Number of output neurons

    #Initialize weights and biases
np.rondom.seed(42) = # For reproducibility
weights_input_hidden = np.random.rand(input_size, hidden_size)   #Weights
bias_hidden = np.random.rand(hidden_size)  # Bias for hidden layer
weights_hidden_output = np.random.rand(hidden_size, output_size)  # Weights
bias_output = np.random.rand(output_size)   # Bias for output layer

# Forward Propagation

  def forward_propagation(X):
    # Calculate input for hidden layer
    hidden_input = np.dot(X, weights_input_hidden) + bias_hidden
    hidden_output = sigmoid(hidden_input) # Activation function


    # Calculate input for output layer
     final_input = np.dot(hidden_output, weights_hidden_output) + bias_output
     output = sigmoid(final_input) # Activation function
         return hidden_output, output

    #Back Propagation and weight update

   def backward_propagation(X, y, hidden_output, output, learning_rate=0.1):
```

```
    # Calculate the error
    output_error = y - output  # Error in output
    output_delta = output_error * sigmoid_derivative(output)  # Derivative of sigmoid

# Error in hidden layer
    hidden_error = output_delta.dot(weights_hidden_output.T)
    hidden_delta = hidden_error * sigmoid_derivative(hidden_output)

# Update weights and biases
    global weights_hidden_output, bias_output, weights_input_hidden, bias_hidden
    weights_hidden_output += hidden_output.T.dot(output_delta) * learning_rate
    bias_output += np.sum(output_delta, axis=0) * learning_rate
    weights_input_hidden += X.T.dot(hidden_delta) * learning_rate
    bias_hidden += np.sum(hidden_delta, axis=0) * learning_rate

# Training the neural network
   def train(X, y, epochs=10000):
     for epoch in range(epochs):
        hidden_output, output = forward_propagation(X)
        backward_propagation(X, y, hidden_output, output)

# Example input (X) and output (y)
X = np.array([[0, 0, 1],
              [1, 1, 1],
              [1, 0, 1],
              [0, 1, 1]])  # Input data
y = np.array([[0],
              [1],
              [1],
              [0]])  # Output data

# Train the neural network
train(X, y)

# Testing the neural network
_, output = forward_propagation(X)
print("Predicted Output:\n", output)
```

Try: Design and develop a neural network with MNIST dataset.

8.2 Implementing Weight Matrix for Digit Classification

Imagine a bank that uses automated systems to process handwritten checks. Each check contains handwritten digits representing the check amount that needs to be recognized accurately.
The goal is to set up a weight matrix (w) for a neural network that classifies handwritten digits (0-9). Each digit is represented as an image of 28x28 pixels, flattened into a vector of 784 features (pixels). The weight matrix w helps the neural network learn the patterns associated with each digit.

Input: **Input Vector (X):**

- The input is a vector of 784 features, corresponding to the 28x28 pixel grayscale image of a handwritten digit.
- Each pixel is represented by a numerical value (typically 0-255 for grayscale intensity), which gets flattened into a 1-dimensional array.

- Example: For a 28x28 image of the digit "3", the input could look like [0, 0, 12, ..., 255, 34, 0], where each number corresponds to the pixel intensity

Output: The output is a vector of 10 values (one for each digit class 0-9).

Each value in the output vector represents the network's confidence score (logit) for each digit class.

These scores are typically passed through a softmax function to convert them into probabilities that sum to 1. Example: If the network predicts the digit "3", the output might look like [0.01, 0.03, 0.05, 0.85, 0.02, 0.01, 0.01, 0.01, 0.005, 0.005], where the highest value (0.85) corresponds to the class '3'.

Explanation:

Initialization:

Initialize w with random values using np.random.randn, which generates values from a standard normal distribution. This is a common practice when starting neural network training.

Weights for Each Digit: Each row in w corresponds to a digit from 0 to 9. For instance, w[0] contains the weights associated with the digit 0.

Usage: Use these weights in a neural network's forward pass to calculate the output (logits) for each digit, followed by applying a softmax function to obtain probabilities.

Hint:

```
# Import the Modules like cluster hierarchy, linkage, and dendrogram

import matplotlib.pyplot as plt
import numpy as np

# Number of classes (digits 0-9)

num_classes = 10

# Number of input features (28x28 pixels flattened into 784 features)
input_size = 784

# Initialize the weights matrix with random values

# Each row of w corresponds to the weights for a digit class (0 through 9)

w = np.random.randn(num_classes, input_size)

# Example: accessing the weights for digit '0'

weights_for_digit_0 = w[0]

# Display the shape to confirm it's 10x784
print("Shape of weights matrix:", w.shape)

# Example of printing weights for digit '0'
print("Weights for digit 0 (first 10 elements):", weights_for_digit_0[:10])
```

Try: Design and develop implementation of weight matrices for digit classification using Digits dataset from Scikit-learn.

8.6 Reshaping digits using Neural Networks

Suppose you are developing a neural network model to classify handwritten digits, such as those used by automated postal services to recognize ZIP codes on envelopes. The images in the dataset are stored as flattened vectors with 784 elements (28x28 pixels). Before feeding these images into the neural network, you need to visualize them to verify their correctness and understand the raw input data.

Input: Customer/Iris Dataset

Output: Visualization showing the implementation of DBSCAN Clustering

Explanation:
Density-based spatial clustering of applications with noise, or DBSCAN, is a popular clustering algorithm used as a replacement for k-means in predictive analytics. To run it doesn't require an input for the number of clusters, but it does need to tune two other parameters.

Hint:

```
# Importing Modules

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist

# Load MNIST Dataset
# x_train contains the images, y_train contains the labels
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Select an image to display (e.g., the first image in the training set)dbscan =
image_index = 0  # You can change this index to display different images
image = x_train[image_index]  # The image is already in 28x28 shape for MNIST


# If you have a 784-element flattened vector, use:
# image = x_train[image_index].reshape(28, 28)


# Display the image

plt.imshow(image, cmap='gray')
plt.title(f"Label: {y_train[image_index]}")
plt.axis('off')  # Hide axis
plt.show()
```

Try: Use flattened image to digit classification.

9. Performance Evaluation

9.1 Developing CNN Model

Performance metrics are crucial for evaluating algorithms like regression and classification, as the choice of metric directly impacts the interpretation of results. For instance, in an email spam detection model,

using accuracy might be misleading if spam is rare, as the model could achieve high accuracy by predicting "not spam" for all emails. Instead, precision would focus on minimizing false positives (misclassifying important emails as spam), while recall would prioritize identifying all spam emails, even if it means catching some legitimate ones as spam. The F1 score balances precision and recall, making it ideal when both false positives and false negatives are important. Selecting the right metric ensures the model aligns with the specific goals of the application, leading to more meaningful and effective outcomes.

Input: CIFAR - 10 dataset

Output: Visualize the 2D projection with two components of PCA graph.

Explanation:
A Convolutional Neural Network (CNN) is ideal for image classification tasks. Below is an example of how you can build a CNN using TensorFlow and Keras for binary classification:

Hint:

```
# import all necessary libraries

from tensorflow.keras.datasets import cifar10
import numpy as np
import tensorflow as tf
# Load CIFAR-10 dataset

(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Select only class 3 (cats) and class 5 (dogs)

train_mask = np.isin(y_train, [3, 5]).flatten()
test_mask = np.isin(y_test, [3, 5]).flatten()


x_train, y_train = x_train[train_mask], y_train[train_mask]
x_test, y_test = x_test[test_mask], y_test[test_mask]


# Convert labels to binary (cat: 0, dog: 1)

y_train = np.where(y_train == 3, 0, 1)
y_test = np.where(y_test == 3, 0, 1)


# Normalize image data

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
```

Hint:

```
# import all necessary libraries

from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
```

```
# Initialize the CNN
model = Sequential()

# Add layers
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))  # Input
shape is 32x32x3 for CIFAR-10
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

# Add dense layers
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))  # Sigmoid for binary classification

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

# Print the model summary
model.summary()
```

9.2 Image Classification for Binary dataset

To train the CNN, you can use a dataset generator or load images directly.

Input: Iris Dataset
Output: Visualization showing the four features of Iris dataset by applying t-SNE model

Explanation:
The train the model using the following code. Split the training set further into validation data for evaluation during training.

Hint:

```
# Image preprocessing and augmentation
# Split training data for validation
from sklearn.model_selection import train_test_split
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2,
random_state=42)

# Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=32,
validation_data=(x_val, y_val))
```

Try: Make use of Scikit-learn implementation of PCA and perform the dimension

9.3 Evaluation Using Confusion Matrix

To train the CNN, you can use a dataset generator or load images directly.

Input: CIFAR – 10 Dataset
Output: Visualization showing the four features of Iris dataset by applying t-SNE model

Explanation:
One of the unsupervised learning methods for visualization is **t-distributed stochastic neighbor embedding, or t-SNE.** It maps high-dimensional space into a two or three-dimensional space which can then be visualized. Specifically, it models each high-dimensional object by a two- or three-dimensional point in such a way that similar objects are modeled by nearby points and dissimilar objects are modeled by distant points with high probability.

Hint:

```
from sklearn.metrics import confusion_matrix, classification_report
import numpy as np

# Make predictions on the test set
y_pred = model.predict(x_test)
y_pred = np.round(y_pred).astype(int).flatten()

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)

# Classification Report
cr = classification_report(y_test, y_pred, target_names=['Cat', 'Dog'])
print("Classification Report:")
print(cr)
```

9.4 Accuracy and Loss in Classification Report

To train the CNN, you can use a dataset generator or load images directly.

Input: CIFAR Dataset
Output: Visualization showing the four features of Iris dataset by applying t-SNE model

Explanation:

Confusion Matrix: Showing the number of true positives, false positives, true negatives, and false negatives.

Classification Report: Including metrics like precision, recall, F1-score.

Accuracy and Loss Plot: A plot showing how accuracy and loss changed during training.

Hint:

```
from sklearn.metrics import confusion_matrix, classification_report
import matplotlib.pyplot as plt

# Plot Accuracy
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
```

```
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy')
plt.legend()

# Plot Loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss')
plt.legend()

plt.show()plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.title('Accuracy')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.title('Loss')
plt.show()
```

Try: Write the implementation code with other dataset or customized dataset.

10. Stochastic Encoders and Decoders

10.1 Develop a simple and basic possible autoencoder.

Compressing and reconstructing images of handwritten digits (0-9) from the MNIST dataset. This is useful for tasks like image denoising or dimensionality reduction for visualization.

Input: Images of handwritten digits (28x28 grayscale images).

Output: Reconstructed images that should look similar to the original images.

Explanation:

Data Preparation**:** The MNIST dataset is loaded and normalized, and images are reshaped for the neural network. An encoder with a single hidden layer (64 neurons) compresses the input data, and a decoder reconstructs the images.

Training**:** The autoencoder is trained using the mean squared error loss function to minimize the difference between input and reconstructed images.

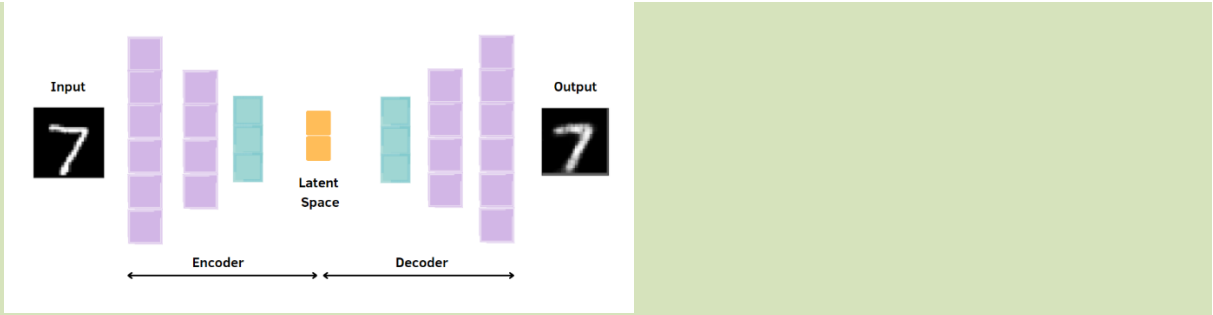
Hint:

```
# Import necessary modules
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Flatten, Reshape
from tensorflow.keras.datasets import mnist
```

```
import matplotlib.pyplot as plt

# Load the MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Flatten the images for the fully connected network
x_train = x_train.reshape((len(x_train), 28 * 28))
x_test = x_test.reshape((len(x_test), 28 * 28))


# Define the autoencoder architecture
input_img = Input(shape=(784,))

# Encoder
encoded = Dense(64, activation='relu')(input_img)
# Decoder
decoded = Dense(784, activation='sigmoid')(encoded)

# Autoencoder Model
autoencoder = Model(input_img, decoded)

# Compile the model
autoencoder.compile(optimizer='adam', loss='mse')

# Train the model
autoencoder.fit(x_train, x_train,
                epochs=20,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))

# Predict and visualize the results
decoded_imgs = autoencoder.predict(x_test)
# Reshape the images back to 28x28 for visualization
x_test = x_test.reshape((len(x_test), 28, 28))
decoded_imgs = decoded_imgs.reshape((len(decoded_imgs), 28, 28))

# Display original and reconstructed images
n = 10  # Number of images to display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i], cmap='gray')
    plt.title("Original")
    plt.axis('off')

# Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i], cmap='gray')
    plt.title("Reconstructed")
    plt.axis('off')

plt.show()
```

Yields below output.

```
# Output:
```



Try: Develop a similar encoder and decoder using Fashion MNIST.

10.2 Encoder and Decoder and Reconstruct MNIST digits

Using the encoder to compress images of handwritten digits into a smaller, latent space representation, and the decoder to reconstruct the images back to their original form. Develop separate encoder and decoder models to compress and reconstruct handwritten digit images from the MNIST dataset. This approach allows for more flexible usage of the encoder and decoder, such as using the encoder for dimensionality reduction and the decoder for data generation

Input: Images of handwritten digits (28x28 grayscale images)
Output: Reconstructed images that should closely resemble the original images.

Explanation:

The encoder reduces the dimensionality of the input data, creating a compact representation (latent space).The decoder takes the latent space representation and reconstructs it back to the original image size.The autoencoder is trained by connecting the encoder and decoder and minimizing the reconstruction error between the input and output. Use the trained encoder to encode test images and the decoder to reconstruct them..

Hint:

```
# import necessary libraries

import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt
```

Hint:

```
# Load the MNIST dataset

(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
```

Hint:

```
# Flatten the images for the fully connected network
x_train = x_train.reshape((len(x_train), 28 * 28))
x_test = x_test.reshape((len(x_test), 28 * 28))
```

Tying this together, the complete example of preparing the semi-supervised learning dataset is listed below.

Hint:

```
# Define the encoder

input_img = Input(shape=(784,))
encoded = Dense(64, activation='relu')(input_img)
encoder = Model(input_img, encoded)

# Define the decoder

encoded_input = Input(shape=(64,))
decoded = Dense(784, activation='sigmoid')(encoded_input)
decoder = Model(encoded_input, decoded)
# Define the autoencoder model by linking encoder and decoder

autoencoder_input = Input(shape=(784,))
encoded_img = encoder(autoencoder_input)
decoded_img = decoder(encoded_img)
autoencoder = Model(autoencoder_input, decoded_img)

# Compile the model

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

# Train the autoencoder
```

```
autoencoder.fit(x_train, x_train,
                epochs=20,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test),
                verbose=0)
```

Hint:

```
# Encode and decode some images from the test set

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

# Reshape images back to 28x28 for visualization
x_test = x_test.reshape((len(x_test), 28, 28))
decoded_imgs = decoded_imgs.reshape((len(decoded_imgs), 28, 28))
```

Hint:

```
...

# Display original and reconstructed images
n = 10  # Number of images to display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i], cmap='gray')
```

```
    plt.title("Original")
    plt.axis('off')


# Display reconstruction
ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i], cmap='gray')
    plt.title("Reconstructed")
    plt.axis('off')

plt.show()
```

Try: Create encoder and decoder for LSUN(Large-scale Scene Understanding) and reconstruct the scenes.

10.3 Configure the Model using Binary Cross Entropy and Adam Optimizer

Consider the following problem of given sets of unlabeled observations, each set with known label proportions, predict the labels of another set of observations, possibly with known label proportions. This problem occurs in areas like e-commerce, politics, spam filtering and improper content detection. We present consistent estimators which can reconstruct the correct labels with high probability in a uniform convergence sense. The goal of this exercise is to generate the estimate labels for the training set using consistent estimator

Input: MNIST dataset

Output: Reconstructed images by the VAE (Variational Autoencoder)

Explanation:
Maps input images to a distribution in the latent space, producing mean (mu) and log-variance (logvar). Decoder: Reconstructs the images from samples drawn from the latent space distribution. Samples from the latent space distribution using the reparameterization trick. Combines binary cross-entropy for reconstruction loss and KL divergence for regularizing the latent space. Trains the model by minimizing the combined loss function. Show a comparison between original and reconstructed images.

Hint:

```
# import necessary libraries

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
```

Hint:

```
# Define the Encoder, Decoder, and VAE Models

class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.fc1 = nn.Linear(784, 400)
```

```
        self.fc21 = nn.Linear(400, 20)  # Mean of latent space
        self.fc22 = nn.Linear(400, 20)  # Log-variance of latent space

    def forward(self, x):
        h1 = torch.relu(self.fc1(x))
        z_mean = self.fc21(h1)
        z_logvar = self.fc22(h1)
        return z_mean, z_logvar

class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.fc3 = nn.Linear(20, 400)
        self.fc4 = nn.Linear(400, 784)

    def forward(self, z):
        h3 = torch.relu(self.fc3(z))
        x_reconstructed = torch.sigmoid(self.fc4(h3))  # Sigmoid for binary output
        return x_reconstructed

class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def forward(self, x):
        mu, logvar = self.encoder(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        x_reconstructed = self.decoder(z)
        return x_reconstructed, mu, logvarore*100))
```

```
# Define the Loss Function

def loss_function(recon_x, x, mu, logvar):
    BCE = nn.functional.binary_cross_entropy(recon_x, x.view(-1, 784),
reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD

# Set Up DataLoader for MNIST

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))])

train_dataset = torchvision.datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
#Initialize Model, Optimizer, and Training Loop
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
model = VAE().to(device)

optimizer = optim.Adam(model.parameters(), lr=1e-3)

def train(model, dataloader, optimizer, epochs=5):
    model.train()
    for epoch in range(epochs):
        epoch_loss = 0
        for batch_idx, (data, _) in enumerate(dataloader):
            data = data.to(device)
            optimizer.zero_grad()
            recon_batch, mu, logvar = model(data)
            loss = loss_function(recon_batch, data, mu, logvar)
            loss.backward()
            epoch_loss += loss.item()
            optimizer.step()
            if batch_idx % 100 == 0:
                print(f'Epoch {epoch} [{batch_idx *
len(data)}/{len(dataloader.dataset)}] Loss: {loss.item()}')
        print(f'Epoch {epoch} Average Loss: {epoch_loss / len(dataloader.dataset)}')


train(model, train_loader, optimizer)



# Visualize

def visualize_reconstruction(model, dataloader, num_samples=10):
    model.eval()
    with torch.no_grad():
        data, _ = next(iter(dataloader))
        data = data.to(device)
        recon_batch, _, _ = model(data)
        data = data.cpu().numpy()
        recon_batch = recon_batch.cpu().numpy()
        fig, axes = plt.subplots(2, num_samples, figsize=(10, 2))

        for i in range(num_samples):
            axes[0, i].imshow(data[i].reshape(28, 28), cmap='gray')
            axes[0, i].axis('off')
            axes[1, i].imshow(recon_batch[i].reshape(28, 28), cmap='gray')
            axes[1, i].axis('off')
        plt.show()
visualize_reconstruction(model, train_loader)
```

Try: Compare the Loss functions by configuring the model with Adam Optimizer.

10.4 Visualize the reconstructed inputs and encoded representations

To visualize the reconstructed inputs and encoded representations in a Variational Autoencoder (VAE), we can create a visualization using Python and libraries like PyTorch and Matplotlib. This involves the following steps:

1. **Visualizing Reconstructed Inputs**: We compare the original images with the reconstructed images from the VAE.

2. **Visualizing Encoded Representations**: We plot the latent space representations (encoded vectors) to observe how the model has organized the data in the latent space.

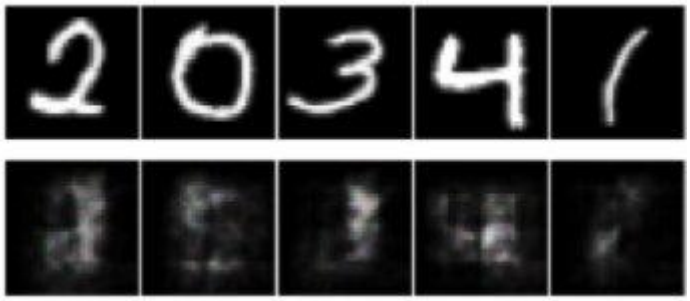
Input: Test dataset
Output: A plot comparing the original images (top row) with their reconstructions (bottom row)
Explanation:
Reconstructed Inputs:

- The top row displays the original images, and the bottom row displays the images reconstructed by the VAE. A good reconstruction indicates that the VAE effectively learns to encode and decode images.

Encoded Representations:

- The scatter plot shows the 2-dimensional latent space representations of the input images. Each point corresponds to an image, and the color indicates its class label (e.g., digits 0-9).

- This visualization helps you understand how well the VAE has organized the data in the latent space. Ideally, similar images (e.g., same digits) should cluster together, showing that the VAE has learned meaningful latent representations.



Hint:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt


# Define the encoder, decoder, and VAE classes (as provided earlier)

class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.fc1 = nn.Linear(784, 400)
        self.fc21 = nn.Linear(400, 20)  # Mean of latent space
        self.fc22 = nn.Linear(400, 20)  # Log-variance of latent space

    def forward(self, x):
        h1 = torch.relu(self.fc1(x))
        z_mean = self.fc21(h1)
        z_logvar = self.fc22(h1)
        return z_mean, z_logvar
```

```
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.fc3 = nn.Linear(20, 400)
        self.fc4 = nn.Linear(400, 784)

    def forward(self, z):
        h3 = torch.relu(self.fc3(z))
        x_reconstructed = torch.sigmoid(self.fc4(h3))  # Sigmoid for binary output
        return x_reconstructed

class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def forward(self, x):
        mu, logvar = self.encoder(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        x_reconstructed = self.decoder(z)
        return x_reconstructed, mu, logvar
# Load MNIST data

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))])
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
download=True, transform=transform)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=True)
# Initialize model and optimizer
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = VAE().to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
# Load a pre-trained model or train the model (code for training would be similar
to previous examples)
# Visualize reconstructed inputs and encoded representations
def visualize_reconstructions(model, dataloader):
    model.eval()
    with torch.no_grad():
        data, _ = next(iter(dataloader))
        data = data.to(device)
        recon_data, mu, logvar = model(data)

    # Plot original images

    fig, axes = plt.subplots(2, 8, figsize=(15, 4))
        for i in range(8):
            axes[0, i].imshow(data[i].cpu().view(28, 28), cmap='gray')
            axes[0, i].axis('off')
            axes[0, i].set_title('Original')

    # Plot reconstructed images
```

```
for i in range(8):
            axes[1, i].imshow(recon_data[i].cpu().view(28, 28), cmap='gray')
            axes[1, i].axis('off')
            axes[1, i].set_title('Reconstructed')

        plt.show()

def visualize_latent_space(model, dataloader):
    model.eval()
    z_list = []
    labels_list = []
    with torch.no_grad():
        for data, labels in dataloader:
            data = data.to(device)
            mu, logvar = model.encoder(data.view(-1, 784))
            z = model.reparameterize(mu, logvar)
            z_list.append(z.cpu())
            labels_list.append(labels)

    z = torch.cat(z_list)
    labels = torch.cat(labels_list)

    plt.figure(figsize=(10, 8))
    plt.scatter(z[:, 0], z[:, 1], c=labels, cmap='viridis', alpha=0.7)
    plt.colorbar()
    plt.xlabel('Latent Dimension 1')
    plt.ylabel('Latent Dimension 2')
    plt.title('Latent Space Visualization')
    plt.show()
    # Example: Visualize reconstructions and latent space
    visualize_reconstructions(model, test_loader)
    visualize_latent_space(model, test_loader)
```

Try: Make use of at least three other datasets and do the comparative study on the results obtained.

10.5 Label Propagation Learning a Complex Structure

The decision boundaries generated by SVM, Label Spreading, and Self-training on a dataset of iris were compared. The results show that these methods can learn good boundaries even if there are only a few labeled pieces of information. Note that training with 100% of the data is not included since it is functionally identical to doing so with the SVC. The goal of this exercise is to demonstrate the label spreading and self-training can learn good boundaries.

Input: make_circles dataset

Output: Obtain 100% with self-trained data

Explanation:
Example of LabelPropagation learning a complex internal structure to demonstrate “manifold learning”. The outer circle should be labeled “red” and the inner circle “blue”. Because both label groups lie inside their own distinct shape, we can see that the labels propagate correctly around the circle.

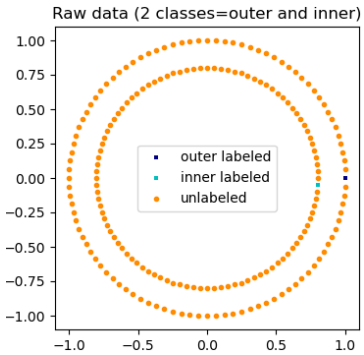
We generate a dataset with two concentric circles. In addition, a label is associated with each sample of the dataset that is: 0 (belonging to the outer circle), 1 (belonging to the inner circle), and -1 (unknown). Here, all labels but two are tagged as unknown.

Hint:

```
import numpy as np

from sklearn.datasets import make_circles

n_samples = 200
X, y = make_circles(n_samples=n_samples, shuffle=False)
outer, inner = 0, 1
labels = np.full(n_samples, -1.0)
labels[0] = outer
labels[-1] = inner
```

Hint:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(4, 4))
plt.scatter(
    X[labels == outer, 0],
    X[labels == outer, 1],
    color="navy",
    marker="s",
    lw=0,
    label="outer labeled",
    s=10,
)
plt.scatter(
    X[labels == inner, 0],
    X[labels == inner, 1],
    color="c",
    marker="s",
    lw=0,
    label="inner labeled",
    s=10,
)
plt.scatter(
    X[labels == -1, 0],
    X[labels == -1, 1],
    color="darkorange",
    marker=".",
    label="unlabeled",
)
plt.legend(scatterpoints=1, shadow=False, loc="center")
_ = plt.title("Raw data (2 classes=outer and inner)")
```



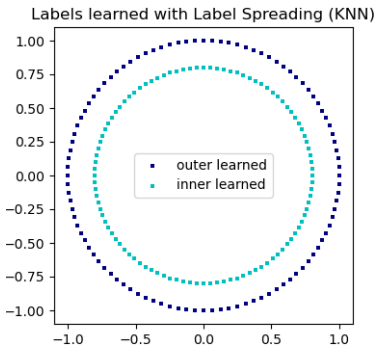
The aim of **LabelSpreading** is to associate a label to sample where the label is initially unknown.

```
from sklearn.semi_supervised import LabelSpreading

label_spread = LabelSpreading(kernel="knn", alpha=0.8)
label_spread.fit(X, labels)
LabelSpreading(alpha=0.8, kernel='knn')
```

Now, we can check which labels have been associated with each sample when the label was unknown.

Hint:

```
output_labels = label_spread.transduction_
output_label_array = np.asarray(output_labels)
outer_numbers = np.where(output_label_array == outer)[0]
inner_numbers = np.where(output_label_array == inner)[0]

plt.figure(figsize=(4, 4))
plt.scatter(
    X[outer_numbers, 0],
    X[outer_numbers, 1],
    color="navy",
    marker="s",
    lw=0,
    s=10,
    label="outer learned",
)
plt.scatter(
    X[inner_numbers, 0],
    X[inner_numbers, 1],
    color="c",
    marker="s",
    lw=0,
    s=10,
    label="inner learned",
)
plt.legend(scatterpoints=1, shadow=False, loc="center")
plt.title("Labels learned with Label Spreading (KNN)")
plt.show()
```



Try: Implement the code to perform the efficient non-parametric function induction in semi-supervised learning.

11. Test Model

11.1 Data Preparation for the RNN model

Recurrent Neural Networks (RNNs) are neural network architecture that is suited for handling sequential datasets. Sequential datasets are these types of datasets that has some forms of sequence such as: sound(a sequence of sound waves), texts (sequence of words or characters), and time series.

When you are working with sequential data (say texts), you want to preserve the sequence of the information.

Take an example: Given the sentence Today, I took my cat for a [....], can you predict the next word? In order for a machine to predict the next word in that sentence, there should be a way to maintain the flow and the sequence of the words, not only from the input to output, but also having a backward connection at each time step. That is where the notion of recurrence comes from.

Input: Imdb movie reviews dataset.

Output:

Explanation:

The conventional feedforward networks have a constant flow of information: The data goes straight from the input to output, whereas in RNNs, there is a feedback loop of data at each time step. Also, the input dimension of data to be fed to feedforward network is fixed, whereas RNNs can handle the data of variable input lengths/dimension.

With that said, RNNs possesses some disadvantages. Because they have a small memory, they can not be able to track long term sequences. Also, they suffer from vanishing and exploding gradients. And these disadvantages motivates the other version of recurrent networks called Long Short Term Memories (LSTMs)

Hint:

```
import tensorflow as tf
from tensorflow import keras
import numpy as n
```

Hint:

```
max_features = 20000
sequence_length = 200

(train_data, train_labels), (val_data, val_labels) =
keras.datasets.imdb.load_data(num_words=max_features)
```

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz

17465344/17464789 [==============================] - 0s 0us/step

17473536/17464789 [==============================] - 0s 0us/step

Loading the training and validation data from Keras datasets

```
train_data[:2]
```

Output: array([1, 0, 0, 1, 0, 0, 1, 0, 1, 0])

array([list([1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670, 2, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2, 336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147, 2025, 19, 14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38,

76, 15, 13, 1247, 4, 22, 17, 515, 17, 12, 16, 626, 18, 19193, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 66, 3785, 33, 4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135, 48, 25, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 10311, 8, 4, 107, 117, 5952, 15, 256, 4, 2, 7, 3766, 5, 723, 36, 71, 43, 530, 476, 26, 400, 317, 46, 7, 4, 12118, 1029, 13, 104, 88, 4, 381, 15, 297, 98, 32, 2071, 56, 26, 141, 6, 194, 7486, 18, 4, 226, 22, 21, 134, 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92, 25, 104, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16, 4472, 113, 103, 32, 15, 16, 5345, 19, 178, 32]),

list([1, 194, 1153, 194, 8255, 78, 228, 5, 6, 1463, 4369, 5012, 134, 26, 4, 715, 8, 118, 1634, 14, 394, 20, 13, 119, 954, 189, 102, 5, 207, 110, 3103, 21, 14, 69, 188, 8, 30, 23, 7, 4, 249, 126, 93, 4, 114, 9, 2300, 1523, 5, 647, 4, 116, 9, 35, 8163, 4, 229, 9, 340, 1322, 4, 118, 9, 4, 130, 4901, 19, 4, 1002, 5, 89, 29, 952, 46, 37, 4, 455, 9, 45, 43, 38, 1543, 1905, 398, 4, 1649, 26, 6853, 5, 163, 11, 3215, 10156, 4, 1153, 9, 194, 775, 7, 8255, 11596, 349, 2637, 148, 605, 15358, 8003, 15, 123, 125, 68, 2, 6853, 15, 349, 165, 4362, 98, 5, 4, 228, 9, 43, 2, 1157, 15, 299, 120, 5, 120, 174, 11, 220, 175, 136, 50, 9, 4373, 228, 8255, 5, 2, 656, 245, 2350, 5, 4, 9837, 131, 152, 491, 18, 2, 32, 7464, 1212, 14, 9, 6, 371, 78, 22, 625, 64, 1382, 9, 8, 168, 145, 23, 4, 1690, 15, 16, 4, 1355, 5, 28, 6, 52, 154, 462, 33, 89, 78, 285, 16, 145, 95])],

dtype=object)

Hint:

```
train_labels[:10]
```

Try: Write and execute the code for Street View House Numbers (SVNH) dataset. It requires minimum data preprocessing. It is similar to MNIST dataset but has more labeled data (over 600,000 labeled images). The SVNH dataset has been collected from house numbers viewed in Google Street View.
The Imdb dataset from Keras dataset is already prepared. The only thing left to do is to pad the tokenized sequences in order that all sentences/reviews have the same length, length equivalent to sequence_length

Preparing the Dataset.

Hint:

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

train_data = pad_sequences(train_data, maxlen=sequence_length)
val_data = pad_sequences(val_data, maxlen=sequence_length)
```

11.2 Create a RNN Model

The basic RNN is implemented in Keras as a layer. There are more recurrent layers (LSTMs and GRUs) that we will see later, but for now we are going to use SimpleRNN to classify the movie reviews.

Here is the overview of the Keras SimpleRNN layer.

Input: Imdb dataset from Keras dataset

Output: The most efficient path to reach its destination by a bot

Explanation:

- **units**: A positive integer that denotes the dimension of the output.
- **activation**: The activation function to use. The default activation function is tanh.
- **dropout**: A float between 0 and 1 denoting the percentage of units or neurons to drop for the linear transformation of the input.
- **recurrent_dropout**: A float between 0 and 1 denoting the percentage of units or neurons to drop for the linear transformation of the recurrent state.

- **return_state**: True or False. Denotes whether to return the last state in addition to the output.
- **go_backwards**: True or False. Denotes whether to process the input sequence backwards and return the reversed sequence.

That's a few notes about the important parameters. For more about the SimpleRNN layer, there is a well-documented help at keras.io

Hint:

```
tf.keras.layers.SimpleRNN(
    units,
    activation="tanh",
    dropout=0.0,
    recurrent_dropout=0.0,
    return_sequences=False,
    return_state=False,
    go_backwards=False
)

simple_rnn = tf.keras.Sequential([

  # Add an embedding layer to convert input tokens into a feature vector
  tf.keras.layers.Embedding(input_dim=max_features, output_dim=32,
input_length=sequence_length),
  tf.keras.layers.SimpleRNN(32, return_sequences=True),
  tf.keras.layers.SimpleRNN(16, return_sequences=True),
  tf.keras.layers.Dense(10, activation='relu'),
  tf.keras.layers.Dense(1, activation='sigmoid')
])
# Compiling the model

simple_rnn.compile(optimizer='adam', loss='binary_crossentropy',
metrics='accuracy')
```

11.3 Train the Model

Problem Statement: The basic RNN is implemented in Keras as a layer. Training a Recurrent Neural Network (RNN) involves feeding sequential data into the model, where each step depends on the previous one due to its memory-like structure. The model learns to capture temporal dependencies by updating its weights using backpropagation through time (BPTT

Input: Imdb Movie Review dataset

Output: Epoch 1/3

782/782 [==============================] - 486s 618ms/step - loss: 0.6143 - accuracy: 0.6478 - val_loss: 0.5477 - val_accuracy: 0.7082

Epoch 2/3

782/782 [==============================] - 483s 617ms/step - loss: 0.5089 - accuracy: 0.7352 - val_loss: 0.5863 - val_accuracy: 0.6944

Epoch 3/3

782/782 [==============================] - 481s 615ms/step - loss: 0.6491 - accuracy: 0.6101 - val_loss: 0.6275 - val_accuracy: 0.638.

Explanation: Compile the model using Binary Cross-entropy and an optimizer like Adam. Train the model using the training dataset, with validation data for monitoring performance.

```
#Hint:
# Training the model

history = simple_rnn.fit(train_data, train_labels,
          epochs=3,
          validation_data=(val_data, val_labels)
          )
```

Hint:

```
#Checking the accuracy on the training data.
class Agent:
  import matplotlib.pyplot as plt
        #print("Total rewards so far= %.3f "%self.total_rewards)
```

Checking the accuracy on training data.

```
  # function to plot accuracy and loss

def plot_acc_loss(history):

  model_history = history.history
  acc = model_history['accuracy']
  val_acc = model_history['val_accuracy']
  loss = model_history['loss']
  val_loss = model_history['val_loss']
  epochs = history.epoch

  plt.figure(figsize=(10,5))
  plt.plot(epochs, acc, 'r', label='Training Accuracy')
  plt.plot(epochs, val_acc, 'g', label='Validation Accuracy')
  plt.title('Training and validation accuracy')
  plt.xlabel('Epochs')
  plt.ylabel('Accuracy')
  plt.legend(loc=0)
```

Output:

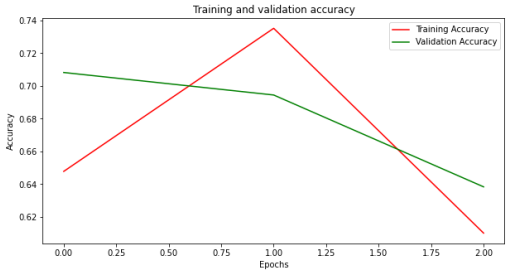
Try: Write and execute the code for Street View House Numbers (SVNH) dataset. It requires minimum data preprocessing. It is similar to MNIST dataset but has more labeled data (over 600,000 labeled images). The SVNH dataset has been collected from house numbers viewed in Google Street View.

11.4 Check the Loss on the Validation Data

The concept of reinforcement learning is a type of machine learning that allows agents to learn to behave in an environment by giving them rewards. The agent can interact with the environment and act based on its current condition. The goal of this exercise is to implement reinforcement learning is to

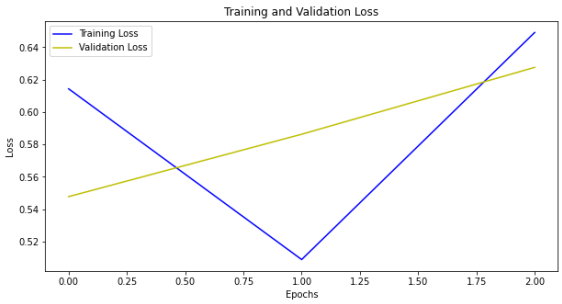
help agents develop a policy that will allow them to maximize their reward over time. This type of machine learning is commonly used in various fields such as gaming and robotics. It can help improve an agent's decision-making skills.

Input: Imdb Movie Review dataset



Output:

<Figure size 432x288 with 0 Axes>



Explanation:

Step 1: Preparing the Imdb Movie Review dataset: Prepare the dataset which contains the preprocessed text data labeled as positive or negative. The reviews are tokenized and padded to ensure uniform input length before feeding input into the RNN.

Step 2: Build a Sequential RNN model using Keras. Add an Embedding layer to represent words, followed by an LSTM (Long Short-Term Memory) layer to capture sequence information, and a Dense output layer for classification.

Step 3: Training the model: Compile the model using binary cross-entropy loss and an optimizer like Adam. Train the model using the training dataset, with validation data for monitoring performance

Step 4: After training, evaluate the model's performance by checking accuracy on the training data using the evaluate method.

Step 5: This process loads the data, builds an RNN model, trains it, and checks the accuracy on the training dataset

Hint:

```
# Create a new figure with plt.figure()
  plt.figure()

  plt.figure(figsize=(10,5))
  plt.plot(epochs, loss, 'b', label='Training Loss')
  plt.plot(epochs, val_loss, 'y', label='Validation Loss')
```

```
  plt.title('Training and Validation Loss')
  plt.xlabel('Epochs')
  plt.ylabel('Loss')
  plt.legend(loc=0)
  plt.show()    agent.step(obj)
  print("Total reward is %.3f "%agent.total_rewards)


plot_acc_loss(history)
```

Try: Write and execute the code for Street View House Numbers (SVNH) dataset. It requires minimum data preprocessing. It is similar to MNIST dataset but has more labeled data (over 600,000 labeled images). The SVNH dataset has been collected from house numbers viewed in Google Street View.

12. Recognize Text In Images

12.1 Memorization Task to test the capabilities of LSTM

Long Short-Term Memory Networks or LSTM in deep learning, is a sequential neural network that allows information to persist. It is a special type of Recurrent Neural Network which is capable of handling the vanishing gradient problem faced by RNN. LSTM Model can be implemented in Python using the Keras library.

Let's say while watching a video, you remember the previous scene, or while reading a book, you know what happened in the earlier chapter. RNNs work similarly; they remember the previous information and use it for processing the current input. The shortcoming of RNN is they cannot remember long-term dependencies due to vanishing gradient. LSTMs are explicitly designed to avoid long-term dependency problems. Unlike traditional neural networks, LSTM incorporates feedback connections, allowing it to process entire sequences of data, not just individual data points. This makes it highly effective in understanding and predicting patterns in sequential data like time series, text, and speech.

Input: Algorithm

Output: Sequence Learning

Explanation:

LSTM (Long Short-Term Memory) is a recurrent neural network (RNN) architecture widely used in Deep Learning. It excels at capturing long-term dependencies, making it ideal for sequence prediction tasks. Sequence prediction is different from traditional classification and regression problems.

It requires that you take the order of observations into account and that you use models like Long Short-Term Memory (LSTM) recurrent neural networks that have memory and that can learn any temporal dependence between observations.

It is critical to apply LSTMs to learn how to use them on sequence prediction problems, and for that, you need a suite of well-defined problems that allow you to focus on different problem types and framings. It is critical so that you can build up your intuition for how sequence prediction problems are different and how sophisticated models like LSTMs can be used to address them.
It is divided into sections:

1. Sequence Learning Problem.
2. Value Memorization
3. Echo Random Integer

4. Echo Random Subsequences

5. Sequence Classification.

The sequence problems were designed with a few properties in mind:

- **Narrow**. To focus on one aspect of the sequence prediction, such as memory or function approximation.
- **Scalable**. To be made more or less difficult along the chosen narrow focus.
- **Reframed**. Two or more framings of the each problem are presented to support the exploration of different algorithm learning capabilities.

1. Sequence Learning Problem

In this problem, a sequence of contiguous real values between 0.0 and 1.0 are generated. Given one or more time steps of past values, the model must predict the next item in the sequence.

Hint:

```
# import necessary libraries
from numpy import array
# generate a sequence of real values between 0 and 1.
def generate_sequence(length=10):
	return array([i/float(length) for i in range(length)])

print(generate_sequence())
```

This could be framed as a memorization challenge where given the observation at the previous time step, the model must predict the next value:

X (samples), y

0.0, 0.1
0.1, 0.2
0.2, 0.3
...

The network could memorize the input-output pairs, which is quite boring, but would demonstrate the function approximation capability of the network.

The problem could be framed as randomly chosen contiguous subsequences as input time steps and the next value in the sequence as output.

X (timesteps), y
0.4, 0.5, 0.6, 0.7
0.0, 0.2, 0.3, 0.4
0.3, 0.4, 0.5, 0.6
...

This would require the network to learn either to add a fixed value to the last seen observation or to memorize all possible subsequences of the generated problem. This framing of the problem would be

modeled as a many-to-one sequence prediction problem. This is an easy problem that tests primitive features of sequence learning. This problem could be solved by a multilayer Perceptron network

2. Value Memorization

   The problem is to remember the first value in the sequence and to repeat it at the end of the sequence. This problem is based on "Experiment 2" used to demonstrate LSTMs in the 1997 paper Long Short Term Memory. This can be framed as a one-step prediction problem. Given one value in the sequence, the model must predict the next value in the sequence. For example, given a value of "0" as an input, the model must predict the value "1".

Consider the following two sequences of 5 integers:

3, 0, 1, 2, 3

4, 0, 1, 2, 4

```
# To generate two sequences
def generate_sequence(length=5):
      return [i for i in range(length)]
# sequence 1
seq1 = generate_sequence()
seq1[0] = seq1[-1] = seq1[-2]
print(seq1)
# sequence 2
seq1 = generate_sequence()
seq1[0] = seq1[-1]
print(seq1
```

Output:

[3, 1, 2, 3, 3]

[4, 1, 2, 3, 4]

The integers could be normalized, or more preferably one hot encoded. The patterns introduce a wrinkle in that there is conflicting information between the two sequences and that the model must know the context of each one-step prediction (e.g. the sequence it is currently predicting) in order to correctly predict each full sequence.

We can see that the first value of the sequence is repeated as the last value of the sequence. This is the indicator that provides context to the model as to which sequence it is working on.

The conflict is the transition from the second to last items in each sequence. In sequence one, a "2" is given as an input and a "3" must be predicted, whereas in sequence two, a "2" is given as input and a "4" must be predicted.

```
# check if a point is within the bounds of the search

Sequence 1:

X (samples), y
...
1,                      2
2,                      3

Sequence 2:
```

```
X (samples), y
...
1,                          2
2,                          4
```

This wrinkle is important to prevent the model from memorizing each single-step input-output pair of values in each sequence, as a sequence unaware model may be inclined to do.

This framing would be modeled as a one-to-one sequence prediction problem. his is a problem that a multilayer Perceptron and other non-recurrent neural networks cannot learn. The first value in the sequence must be remembered across multiple samples.

This problem could be framed as providing the entire sequence except the last value as input time steps and predicting the final value.

X (timesteps),	y
3, 0, 1, 2,	3
4, 0, 1, 2,	4

Each time step is still shown to the network one at a time, but the network must remember the value at the first time step. The difference is, the network can better learn the difference between the sequence, and between long sequences via backpropagation through time.

This framing of the problem would be modeled as a many-to-one sequence prediction problem.

Again, this problem could not be learned by a multilayer Perceptron

13.1 Learned Temporal Dependency Capability of LSTM

3. Echo Random Integer

In this problem, random sequences of integers are generated. The model must remember an integer at a specific lag time and echo it at the end of the sequence.

For example, a random sequence of 10 integers may be:

Input: 5, 3, 2, 1, 9, 9, 2, 7, 1, 6
Output: [47, 69, 76, 9, 71, 87, 8, 16, 32, 81]

Explanation:

The problem may be framed as echoing the value at the 5th time step, in this case 9. The code below will generate random sequences of integers.

Hint:

```
from random import randint

# generate a sequence of random numbers in [0, 99]

def generate_sequence(length=10):
```

```
      return [randint(0, 99) for _ in range(length)]
print(generate_sequence())
```

The integers can be normalized, but more preferably a one hot encoding can be used.
A simple framing of this problem is to echo the current input value.
1yhat(t) = f(X(t))
For example:
1X (timesteps), y

25, 3, 2, 1, 9, 9

This trivial problem can easily be solved by a multilayer Perceptron and could be used for calibration or diagnostics of a test harness.

A more challenging framing of the problem is to echo the value at the previous time step.

1yhat(t) = f(X(t-1))

For example::

1X (timesteps), y

25, 3, 2, 1, 9, 1
This is a problem that cannot be solved by a multilayer Perceptron.
The index to echo can be pushed further back in time, putting more demand on the LSTMs memory.
Unlike the "Value Memorization" problem above, a new sequence would be generated each training epoch. This would require that the model learn a generalization echo solution rather than memorize a specific sequence or sequences of random numbers.
In both cases, the problem would be modeled as a many-to-one sequence prediction problem.

Try: Write and implement the same concept with the Next-Step Prediction in a Number sequence.

14.1 Evaluation of interpretation capability of LSTM

Problem Statement:

Sequence Classification

Let us take above example, The problem is defined as a sequence of random values between 0 and 1. This sequence is taken as input for the problem with each number provided one per timestep.

A binary label (0 or 1) is associated with each input. The output values are all 0. Once the cumulative sum of the input values in the sequence exceeds a threshold, then the output value flips from 0 to 1.

A threshold of 1/4 the sequence length is used

Input: (x)

0.63144003 0.29414551 0.91587952 0.95189228 0.32195638 0.60742236 0.83895793 0.18023048 0.84762691 0.29165514

Output: (y)

0 0 0 1 1 1 1 1 1 1

Explanation:

This is a sequence classification problem that can be modeled as one-to-one. State is required to interpret past time steps to correctly predict when the output sequence flips from 0 to 1.

Hint:

```
# import the necessary libraries
from random import random
from numpy import array
from numpy import cumsum

# create a sequence classification instance
def get_sequence(n_timesteps):
	# create a sequence of random numbers in [0,1]
	X = array([random() for _ in range(n_timesteps)])
	# calculate cut-off value to change class values
	limit = n_timesteps/4.0
	# determine the class outcome for each item in cumulative sequence
	y = array([0 if x < limit else 1 for x in cumsum(X)])
	return X, y

X, y = get_sequence(10)
print(X)
print(y)...
```

Output:

[0.31102339 0.66591885 0.7211718 0.78159441 0.50496384 0.56941485

0.60775583 0.36833139 0.180908 0.80614878]

[0 0 0 0 1 1 1 1 1 1]

Try: Implement for Echo Random Subsequences.

13. Final Notes

The only way to learn programming is program, program, and program on challenging problems. The problems in this tutorial are certainly NOT challenging. There are tens of thousands of challenging problems available – used in training for various programming contests. Check out these sites:

- Industry-Curated Hackathon: Machine Hack (AI Hackathons | MachineHack Generative AI)
- Kaggle Competition for Data Science (Kaggle: Your Machine Learning and Data Science Community)
- Brainstorming Data Science Challenges (Contests | Analytics Vidhya).
- Code Lab Online Data Science Challenges (CodaLab - Competitions).
- Coding Challenges Platform (https://www.topcoder.com/challenges).
- Data Driven Science Online Competition (Competitions (drivendata.org))
- New Science Competition (https://www.icfpconference.org/)
- New Algorithmic Competition (AIcrowd)
- Tianchi Big Data Science Competition (4 Data Science Competition Platforms Other Than Kaggle | by Edwin Tan | Towards Data Science)
- Data Visualization Competition (Iron Viz | Win or learn—you can't lose (tableau.com))

- Other Competitions (ML Contests)

Student must have any one of the following certifications:

1. LearnBay – Machine Learning Course for Professions
2. AWS Certified Machine Learning - Specialty Certification
3. Andrew Ng's Machine Learning Specialization
4. IBM Machine Learning Professional Certification
5. Google Professional Machine Learning Engineer Certification
6. University of Washington Machine Learning Specialization

V. REFERENCE BOOKS:

1. Mehryar Van den Oord, A., Kalchbrenner, N., Espeholt, L., Vinyals, O.,Graves, A., et al. (2016). Conditional image generation with pixelCNN decoders. In Advances in Neural Information Processing Systems.
2. Sutskever, I., Vinyals, O., and Le, Q. V. "Sequence to sequence learning with neural networks, In Advances in neural information processing system", 2014.
3. Avrim Blum, John Hopcroft, Ravindran Kannan, "Foundations of Data Science", Cambridge UniversityPress, 2020.

VI. WEB REFERENCES:

1. http://paulorauber.com/slides/deep_learning_lab.pdf
2. https:// www.geeksforgeeks.org/python-keras-keras-utils-to_categorical
3. https://www. geeksforgeeks.org/deep-convolutional-gan-with-keras/?ref=rp
4. https://tinyurl.com/yk4clsot
5. https://www.analyticsvidhya.com/blog/2018/12/guide-convolutional-neural-network-cnn/
6. https://cs.nyu.edu/~mohri/mlu1