योगः कर्मसु कौशलम्
IN PURSUIT OF PERFECTION

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS

Grade A++ Accredited Institution by NAAC

NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE
An ISO 9001:2015 Certified Institution

SCHOOL OF ENGINEERING & TECHNOLOGY

BTECH Programme: AI&DS

Course Title: Fundamentals of Deep Learning Lab

Course Code: AIDS304P

Submitted To: Dr. Dimple Tiwari

Submitted By: Rohit Kumar Saxena

Enrollment no.: 03817711922

VIPS

योगः कर्मसु कौशलम्

IN PURSUIT OF PERFECTION

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS

Grade A++ Accredited Institution by NAAC

NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;

Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE

An ISO 9001:2015 Certified Institution

SCHOOL OF ENGINEERING & TECHNOLOGY

VISION OF INSTITUTE

To be an educational institute that empowers the field of engineering to build a sustainable future by providing quality education with innovative practices that supports people, planet and profit.

MISSION OF INSTITUTE

To groom the future engineers by providing value-based education and awakening students' curiosity, nurturing creativity and building capabilities to enable them to make significant contributions to the world.

योगः कर्मसु कौशलम्
IN PURSUIT OF PERFECTION

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS
Grade A++ Accredited Institution by NAAC
NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE
An ISO 9001:2015 Certified Institution
SCHOOL OF ENGINEERING & TECHNOLOGY

INDEX

S.No	Experiment Name	Date	Marks			Remark	Updated Marks	Faculty Signature
			Laboratory Assessment (15 Marks)	Class Participation (5 Marks)	Viva (5 Marks)			

योगः कर्मसु कौशलम्
IN PURSUIT OF PERFECTION

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS
Grade A++ Accredited Institution by NAAC
NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE
An ISO 9001:2015 Certified Institution
SCHOOL OF ENGINEERING & TECHNOLOGY

INDEX

S.No	Experiment Name	Date	Marks			Remark	Updated Marks	Faculty Signature
			Laboratory Assessment (15 Marks)	Class Participation (5 Marks)	Viva (5 Marks)			

Experiment 1

Aim: To explore the basic features of Tensorflow and Keras packages in Python.

Objectives:

- Understand the foundational functionalities of TensorFlow and Keras for deep learning.
- Explore how to build, compile, and train basic neural network models using Keras.

Theory:

TensorFlow and Keras are powerful libraries in Python that form the backbone of deep learning model implementation. They provide a comprehensive ecosystem for building, training, and deploying machine learning models. These libraries are pivotal for both academic and practical experiments in the field of deep learning.

1. TensorFlow:

TensorFlow, developed by Google, is an open-source machine learning framework designed for high-performance numerical computations. It enables researchers and developers to efficiently create and train machine learning models across various platforms. TensorFlow's core strength lies in its ability to handle tensors, which are multi-dimensional arrays that form the foundation of all computations in deep learning.

Key features of TensorFlow include:

- **Automatic Differentiation**: TensorFlow computes gradients automatically, simplifying the process of backpropagation for complex neural networks.

- **GPU/TPU Acceleration**: TensorFlow optimally utilizes hardware resources like GPUs and TPUs for faster training and inference.

- **Scalability**: Its distributed training capabilities allow it to handle large-scale models and datasets across multiple devices and servers.

- **Versatility**: TensorFlow supports both low-level control for custom operations and high-level APIs like Keras for simplicity.

2. Keras:

Keras is a high-level neural network API integrated with TensorFlow. It is designed to be user-friendly while still providing powerful features for advanced customization. Keras reduces the complexity of building deep learning models by offering intuitive syntax and ready-to-use components for common tasks.

Some of the features of Keras include:

- **Simplified Model Building**: Keras offers Sequential and Functional APIs to define models effortlessly.

- **Ease of Use**: It includes pre-built layers, loss functions, optimizers, and metrics, allowing for rapid prototyping.
- **Modularity**: Every component in Keras is independent, making it easy to adapt and customize.
- **Integration**: Since it is built on TensorFlow, it inherits the performance and scalability benefits of TensorFlow.

3. Core Concepts:

- **Tensors**:
  The primary data structure in TensorFlow, tensors represent multi-dimensional arrays that enable efficient numerical computation. Operations on tensors are highly optimized for both CPU and GPU execution.
- **Layers**:
  Layers are the building blocks of a deep learning model. In Keras, layers perform specific transformations, such as linear computations, activation functions, or normalization, to process input data progressively.
- **Compilation and Training**:
  Keras provides a streamlined workflow for compiling models, training them on datasets, and evaluating their performance. Functions like compile(), fit(), and evaluate() abstract the complexities of training, making the process user-friendly.
- **Backpropagation and Optimization**:
  TensorFlow and Keras automate the process of calculating gradients and updating model parameters using optimizers like SGD, Adam, or RMSprop. This minimizes loss functions and improves model accuracy over successive iterations.

4. Regression and Classification with TensorFlow and Keras

- **Regression**:
  In regression tasks, the model predicts continuous outputs. TensorFlow and Keras simplify the creation of regression models by providing built-in loss functions (e.g., Mean Squared Error) and metrics to evaluate model performance.
- **Classification**:
  For classification tasks, the model predicts categorical outputs, such as labels for data points. TensorFlow and Keras support multi-class and binary classification through layers like Dense, activation functions like softmax or sigmoid, and loss functions like categorical_crossentropy.

By exploring TensorFlow and Keras, learners gain hands-on experience with key deep learning principles. This experiment lays the groundwork for understanding tensor operations, building neural networks, and solving real-world problems through regression and classification tasks. These tools bridge the gap between theoretical concepts and practical applications in the realm of deep learning.

Code:

```
# Regression

import tensorflow as tf
import numpy as np
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test =
train_test_split(housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(X_train_full,
y_train_full, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)

model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(30, activation='relu',
input_shape=X_train.shape[1:]),
    tf.keras.layers.Dense(1)
])

model.compile(loss='mean_squared_error',
optimizer=tf.keras.optimizers.SGD(learning_rate=0.01))

history = model.fit(X_train, y_train, epochs=20,
validation_data=(X_valid, y_valid))

mse_test = model.evaluate(X_test, y_test)

print(f"Test MSE: {mse_test}")
```

```
# Classification

import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris

iris = load_iris()
```

```
X = iris.data
y = iris.target

y = tf.keras.utils.to_categorical(y, num_classes=3)

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

model = tf.keras.Sequential([
    tf.keras.layers.Dense(10, activation='relu', input_shape=(4,)),
    tf.keras.layers.Dense(3, activation='softmax')
])

model.compile(optimizer='sgd',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=1)

loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy:.4f}")
```

Output:

```
# Regression
```

```
Epoch 1/20
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning:
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
363/363 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 1.5180 - val_loss: 9.0862
Epoch 2/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 0.5275 - val_loss: 11.1998
Epoch 3/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 0.4372 - val_loss: 2.4182
Epoch 4/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.4664 - val_loss: 0.3828
Epoch 5/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.4045 - val_loss: 0.3711
Epoch 6/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 0.3895 - val_loss: 0.3606
Epoch 7/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 0.3724 - val_loss: 0.3563
Epoch 8/20
363/363 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 0.3806 - val_loss: 0.3601
Epoch 9/20
```

```
162/162 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.3398
Test MSE: 0.3419281840324402
```

```
# Classification
```

```
Epoch 1/50
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning:
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5217 - loss: 1.0527
Epoch 2/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5646 - loss: 0.9997
Epoch 3/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5396 - loss: 0.9901
Epoch 4/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.5167 - loss: 0.9830
Epoch 5/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5408 - loss: 0.9570
Epoch 6/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.5148 - loss: 0.9633
Epoch 7/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.5619 - loss: 0.9037
Epoch 8/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5925 - loss: 0.8811
Epoch 9/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5794 - loss: 0.8873
Epoch 10/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.5354 - loss: 0.8884
Epoch 11/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5835 - loss: 0.8282
Epoch 12/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6044 - loss: 0.8249
Epoch 13/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5698 - loss: 0.8236
Epoch 14/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6148 - loss: 0.7863
Epoch 15/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.6119 - loss: 0.7857
```

```
Test Loss: 0.5009
Test Accuracy: 0.8333
```

Learning Outcomes:

- Understand how to build and train regression and classification models using TensorFlow and Keras.
- Gain insights into key evaluation metrics like Mean Squared Error (MSE) for regression and Accuracy or F1-Score for classification tasks.
- Develop the ability to preprocess data and apply activation functions, optimizers, and loss functions suitable for different problem domains.

Experiment 2

Aim: Implementation of ANN model for regression and classification problem in Python.

Objectives:

- To implement an ANN model in Python for regression and evaluate its performance using MSE.
- To implement an ANN model in Python for classification and assess its accuracy using performance metrics.

Theory:

Introduction to Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANNs) are computational models inspired by the human brain's structure and functioning. They consist of layers of interconnected nodes (neurons) which process and transform input data through various layers to produce an output. The power of ANNs lies in their ability to learn from data, identify patterns, and generalize predictions, making them highly suitable for various tasks such as regression, classification, and time series forecasting.

ANNs are composed of three primary types of layers:

1. **Input Layer**: Receives the raw input data.
2. **Hidden Layers**: Contain neurons that process the data using weights and activation functions.
3. **Output Layer**: Provides the final result or prediction.

Each neuron in the hidden layers and output layer processes inputs from previous layers and passes them through an activation function, which introduces non-linearity into the model. The weights of the connections between neurons are learned during training by using optimization techniques like gradient descent.

Regression Problems with ANN

In regression tasks, the objective is to predict a continuous numerical output based on input features. For example, predicting house prices based on location, size, and other features is a typical regression problem.

The basic architecture of an ANN for regression consists of:

- **Input Layer**: Takes in feature data (e.g., square footage, number of bedrooms).
- **Hidden Layers**: Processes the data through neurons.
- **Output Layer**: Produces a continuous output, which could be a single value (e.g., price).

To train an ANN for regression, the model adjusts its weights to minimize the difference between the predicted output and the actual target value using a loss function like Mean Squared Error (MSE). The training process involves backpropagation, where errors are propagated backward through the network, and weights are updated to reduce the loss.

Key Steps in ANN Regression:

1. **Data Preprocessing**: Normalization and scaling of data are crucial for ensuring efficient learning.
2. **Model Construction**: A neural network model is constructed with appropriate layers and activation functions.
3. **Training the Model**: The model is trained using a dataset, adjusting weights based on the error at each step.
4. **Evaluation**: The model's performance is evaluated using metrics like MSE or Root Mean Squared Error (RMSE).

Python libraries like Keras and TensorFlow are often used to implement ANNs for regression, as they provide simple APIs for defining and training neural networks.

Classification Problems with ANN

In classification tasks, the goal is to categorize data into distinct classes or categories. For example, classifying images as either "cat" or "dog" or predicting whether a customer will buy a product based on their behavior are typical classification problems.

The structure of an ANN for classification is similar to that of regression, but with key differences in the output layer:

- **Input Layer**: Receives features of the data (e.g., image pixels, demographic features).
- **Hidden Layers**: Processes the data through neurons.
- **Output Layer**: Uses a softmax or sigmoid activation function to output class probabilities. In binary classification, the output layer consists of a single neuron with a sigmoid activation function. For multi-class classification, a softmax activation function is used.

For classification tasks, the model is trained using a loss function like **Cross-Entropy Loss**, which measures the difference between predicted probabilities and actual class labels. During training, the model learns to adjust its weights to reduce this loss.

Key Steps in ANN Classification:

1. **Data Preprocessing**: This involves encoding categorical variables (e.g., one-hot encoding for labels) and normalizing input features.
2. **Model Construction**: The model is designed with hidden layers and an output layer with appropriate activation functions (sigmoid for binary classification or softmax for multi-class).
3. **Training the Model**: The neural network is trained using a labeled dataset, and the weights are adjusted through backpropagation.
4. **Evaluation**: The performance is evaluated using accuracy, precision, recall, F1-score, or AUC (for binary classification).

Challenges in ANN Models

While ANNs are powerful, they come with several challenges:

1. **Overfitting**: ANN models, especially deep networks with many layers, can easily overfit the training data, leading to poor generalization. Regularization techniques such as dropout, L2 regularization, or early stopping can help mitigate overfitting.
2. **Data Quality**: ANN models are sensitive to the quality and quantity of data. Proper data preprocessing, including handling missing values, scaling, and feature engineering, is crucial.
3. **Model Tuning**: The performance of an ANN heavily depends on hyperparameters like the number of hidden layers, neurons, learning rate, and batch size. Grid search or random search methods are often employed to optimize these hyperparameters.
4. **Computational Complexity**: Training deep neural networks can be computationally expensive, requiring powerful hardware (e.g., GPUs) and careful resource management.

Python Libraries for Implementing ANN

Several Python libraries make it easy to implement ANN models for both regression and classification:

1. **TensorFlow/Keras**: These are the most popular frameworks for building and training neural networks. Keras provides a high-level API that is user-friendly, while TensorFlow provides more flexibility and control for advanced users.
2. **PyTorch**: Another powerful deep learning framework that offers flexibility and dynamic computation graphs, making it suitable for research and development.
3. **Scikit-learn**: Although primarily used for traditional machine learning algorithms, Scikit-learn provides simple neural network implementations (MLPRegressor and MLPClassifier) that are useful for smaller-scale tasks.

Conclusion

ANNs are versatile tools for both regression and classification tasks in machine learning. By leveraging their ability to model complex relationships and learn from data, they can achieve high accuracy in predicting continuous values and categorizing data into distinct classes. Python, with its rich ecosystem of libraries like Keras and TensorFlow, provides a powerful environment for implementing, training, and evaluating ANN models. However, successful deployment requires careful attention to data quality, model tuning, and computational resources to avoid common pitfalls like overfitting and high computational costs.

Code:

```
#Regression

import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
df = pd.DataFrame(housing.data, columns=housing.feature_names)
df['target'] = housing.target

df.head()
df=df.dropna()

X=df.iloc[:,:-1] ## independent features
y=df.iloc[:,-1] ## dependent features

X.isnull()
y.isnull()

sns.pairplot(df)

df.corr()

corrmat = df.corr()
top_corr_features = corrmat.index
plt.figure(figsize=(20,20))
g=sns.heatmap(df[top_corr_features].corr(),annot=True,cmap="RdYlGn")

corrmat.index

from sklearn.ensemble import ExtraTreesRegressor
import matplotlib.pyplot as plt
model = ExtraTreesRegressor()
model.fit(X,y)

X.head()

print(model.feature_importances_)

feat_importances = pd.Series(model.feature_importances_,
index=X.columns)
```

```
feat_importances.nlargest(5).plot(kind='barh')
plt.show()
sns.distplot(y)

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=0)

import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LeakyReLU,PReLU,ELU
from keras.layers import Dropout

NN_model = Sequential()

# The Input Layer :
NN_model.add(Dense(128, kernel_initializer='normal',input_dim =
X_train.shape[1], activation='relu'))

# The Hidden Layers :
NN_model.add(Dense(256, kernel_initializer='normal',activation='relu'))
NN_model.add(Dense(256, kernel_initializer='normal',activation='relu'))
NN_model.add(Dense(256, kernel_initializer='normal',activation='relu'))

# The Output Layer :
NN_model.add(Dense(1, kernel_initializer='normal',activation='linear'))

# Compile the network :
NN_model.compile(loss='mean_absolute_error', optimizer='adam',
metrics=['mean_absolute_error'])
NN_model.summary()

# Fitting the ANN to the Training set
model_history=NN_model.fit(X_train, y_train,validation_split=0.33,
batch_size = 10, epochs = 100)

prediction=NN_model.predict(X_test)

y_test

sns.distplot(y_test.values.reshape(-1,1)-prediction)
plt.scatter(y_test,prediction)

from sklearn import metrics
print('MAE:', metrics.mean_absolute_error(y_test, prediction))
print('MSE:', metrics.mean_squared_error(y_test, prediction))
print('RMSE:', np.sqrt(metrics.mean_squared_error(y_test, prediction)))
```

```python
# Classification

import numpy as np
import pandas as pd
import tensorflow as tf

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data
y = iris.target

y = tf.keras.utils.to_categorical(y, num_classes=3)

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

ann = tf.keras.models.Sequential()

ann.add(tf.keras.layers.Dense(units=6,activation="relu"))
ann.add(tf.keras.layers.Dense(units=6,activation="relu"))
ann.add(tf.keras.layers.Dense(units=3,activation="sigmoid"))

ann.compile(optimizer="adam",loss="categorical_crossentropy",metrics=['
accuracy'])
ann.fit(X_train,y_train,batch_size=32,epochs = 100)

loss, accuracy = ann.evaluate(X_test, y_test, verbose=0)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy:.4f}")
```

Output:

```
#Regression
df.head()
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	target
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422

```
X.isnull()
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False
...	...	...	...	...	...	...	...	...
20635	False	False	False	False	False	False	False	False
20636	False	False	False	False	False	False	False	False
20637	False	False	False	False	False	False	False	False
20638	False	False	False	False	False	False	False	False
20639	False	False	False	False	False	False	False	False

20640 rows × 8 columns

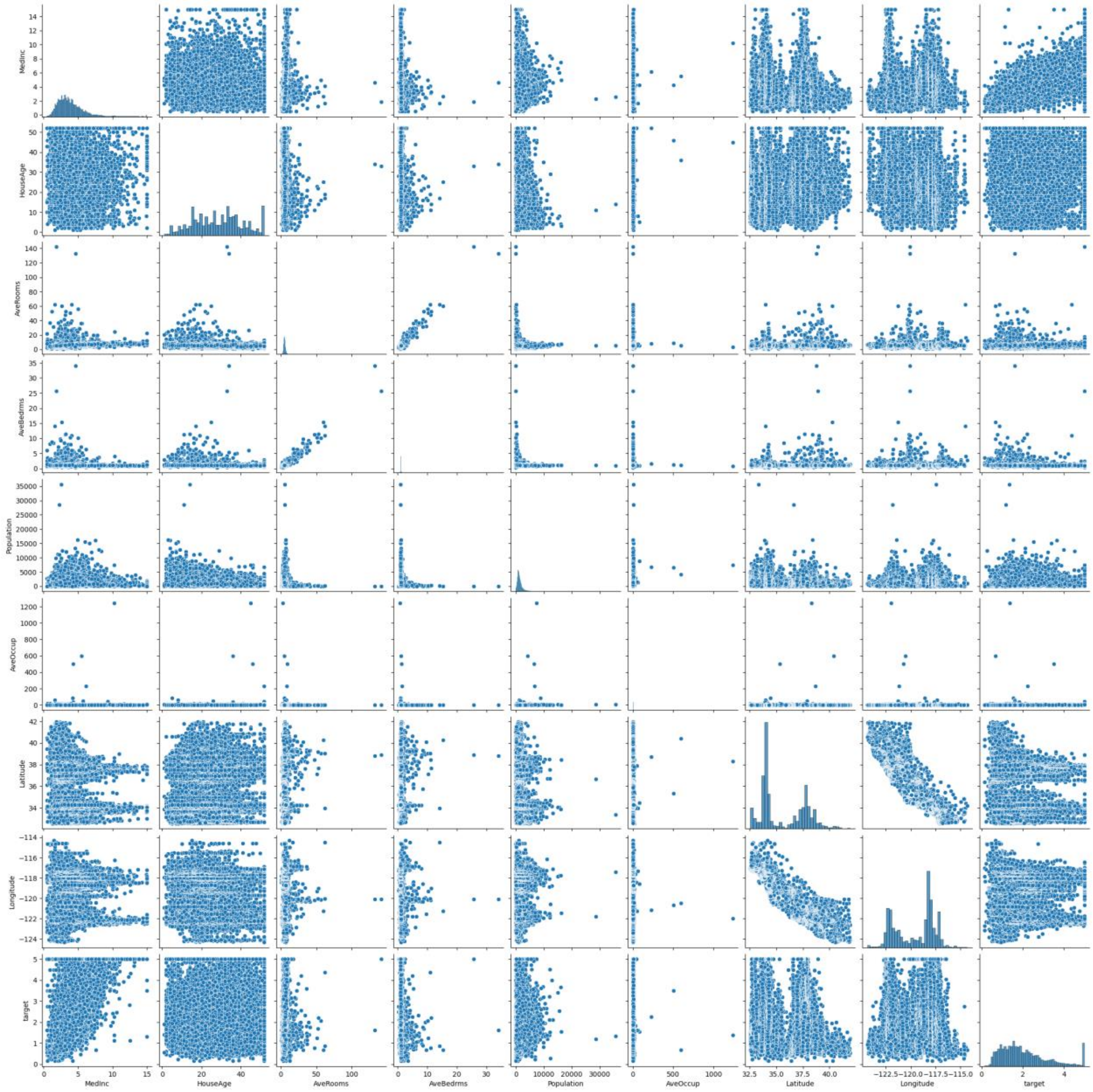
```
y.isnull()
```

	target
0	False
1	False
2	False
3	False
4	False
...	...
20635	False
20636	False
20637	False
20638	False
20639	False

20640 rows × 1 columns

dtype: bool

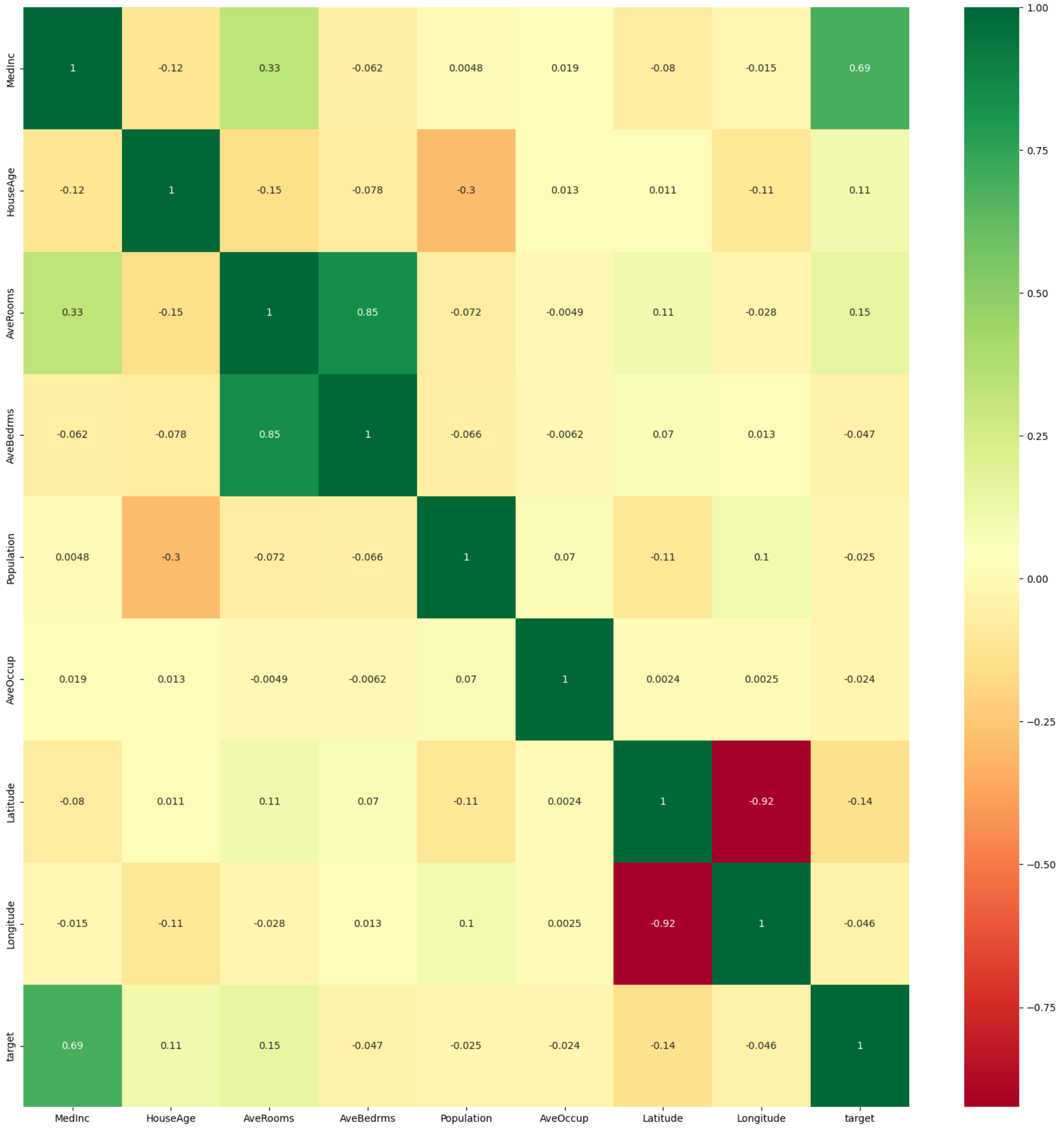
```
sns.pairplot(df)
```



```
df.corr()
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	target
MedInc	1.000000	-0.119034	0.326895	-0.062040	0.004834	0.018766	-0.079809	-0.015176	0.688075
HouseAge	-0.119034	1.000000	-0.153277	-0.077747	-0.296244	0.013191	0.011173	-0.108197	0.105623
AveRooms	0.326895	-0.153277	1.000000	0.847621	-0.072213	-0.004852	0.106389	-0.027540	0.151948
AveBedrms	-0.062040	-0.077747	0.847621	1.000000	-0.066197	-0.006181	0.069721	0.013344	-0.046701
Population	0.004834	-0.296244	-0.072213	-0.066197	1.000000	0.069863	-0.108785	0.099773	-0.024650
AveOccup	0.018766	0.013191	-0.004852	-0.006181	0.069863	1.000000	0.002366	0.002476	-0.023737
Latitude	-0.079809	0.011173	0.106389	0.069721	-0.108785	0.002366	1.000000	-0.924664	-0.144160
Longitude	-0.015176	-0.108197	-0.027540	0.013344	0.099773	0.002476	-0.924664	1.000000	-0.045967
target	0.688075	0.105623	0.151948	-0.046701	-0.024650	-0.023737	-0.144160	-0.045967	1.000000

```
sns.heatmap()
```




```
corrmat.index
```

```
Index(['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population', 'AveOccup',
       'Latitude', 'Longitude', 'target'],
      dtype='object')
```

```
model.fit(X,y)
```

```
▾ ExtraTreesRegressor
ExtraTreesRegressor()
```

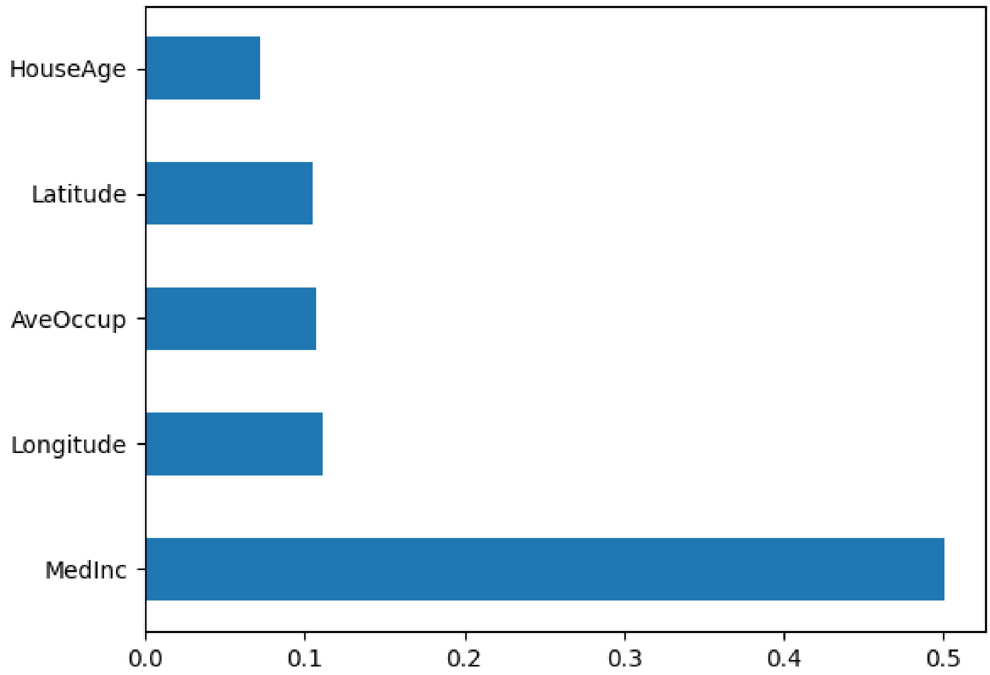
```
X.head()
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25

```
print(model.feature_importances_)
```

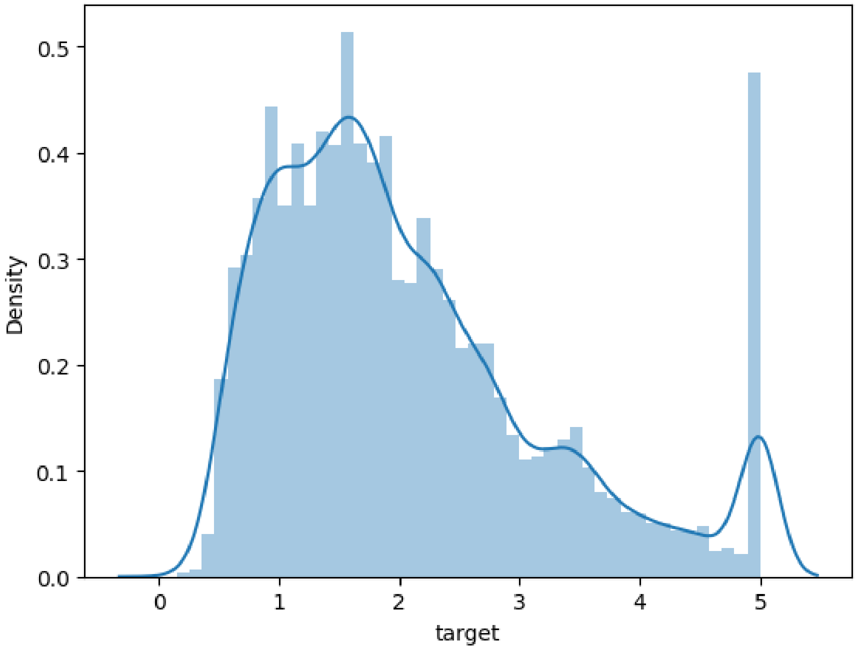
```
[0.50077469 0.07189236 0.04195345 0.03438947 0.0276384  0.10722586
 0.10460644 0.11151933]
```

```
plt.show()
```



```
sns.distplot(y)
```

```
<ipython-input-42-0f415a98584e>:1: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(y)
<Axes: xlabel='target', ylabel='Density'>
```




```
NN_model.fit()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 128)	1,152
dense_1 (Dense)	(None, 256)	33,024
dense_2 (Dense)	(None, 256)	65,792
dense_3 (Dense)	(None, 256)	65,792
dense_4 (Dense)	(None, 1)	257

```
 Total params: 166,017 (648.50 KB)
 Trainable params: 166,017 (648.50 KB)
 Non-trainable params: 0 (0.00 B)
Epoch 1/100
968/968 ━━━━━━━━━━━━━━━━━━━━ 10s 7ms/step - loss: 1.3826 - mean_absolute_error: 1.3826 - val_loss: 0.7857 - val_mean_absolute_error: 0.7857
Epoch 2/100
968/968 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - loss: 0.7961 - mean_absolute_error: 0.7961 - val_loss: 0.7636 - val_mean_absolute_error: 0.7636
Epoch 3/100
968/968 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - loss: 0.6541 - mean_absolute_error: 0.6541 - val_loss: 0.6083 - val_mean_absolute_error: 0.6083
Epoch 4/100
968/968 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - loss: 0.6483 - mean_absolute_error: 0.6483 - val_loss: 0.5825 - val_mean_absolute_error: 0.5825
Epoch 5/100
968/968 ━━━━━━━━━━━━━━━━━━━━ 8s 8ms/step - loss: 0.6072 - mean_absolute_error: 0.6072 - val_loss: 0.5607 - val_mean_absolute_error: 0.5607
Epoch 6/100
968/968 ━━━━━━━━━━━━━━━━━━━━ 6s 6ms/step - loss: 0.5924 - mean_absolute_error: 0.5924 - val_loss: 0.5867 - val_mean_absolute_error: 0.5867
Epoch 7/100
968/968 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - loss: 0.5857 - mean_absolute_error: 0.5857 - val_loss: 0.5459 - val_mean_absolute_error: 0.5459
Epoch 8/100
968/968 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - loss: 0.5931 - mean_absolute_error: 0.5931 - val_loss: 0.5405 - val_mean_absolute_error: 0.5405
Epoch 9/100
968/968 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - loss: 0.5673 - mean_absolute_error: 0.5673 - val_loss: 0.5310 - val_mean_absolute_error: 0.5310
Epoch 10/100
968/968 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - loss: 0.5688 - mean_absolute_error: 0.5688 - val_loss: 0.5931 - val_mean_absolute_error: 0.5931
```

```
prediction=NN_model.predict(X_test)
```

```
194/194 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step
```

```
y_test
```

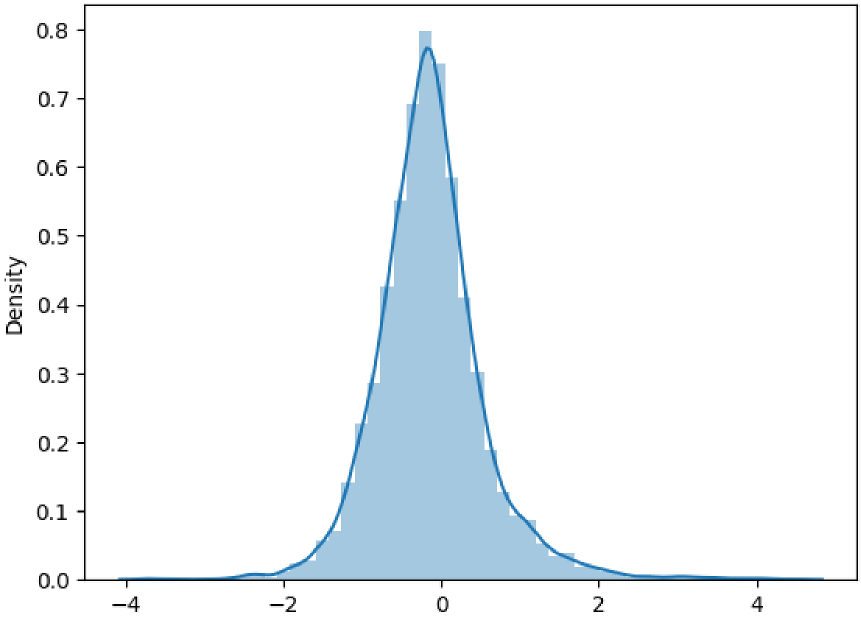
	target
14740	1.369
10101	2.413
20566	2.007
2670	0.725
15709	4.600
...	...
19681	0.740
12156	1.773
10211	3.519
2445	0.925
17914	2.983

6192 rows × 1 columns

dtype: float64

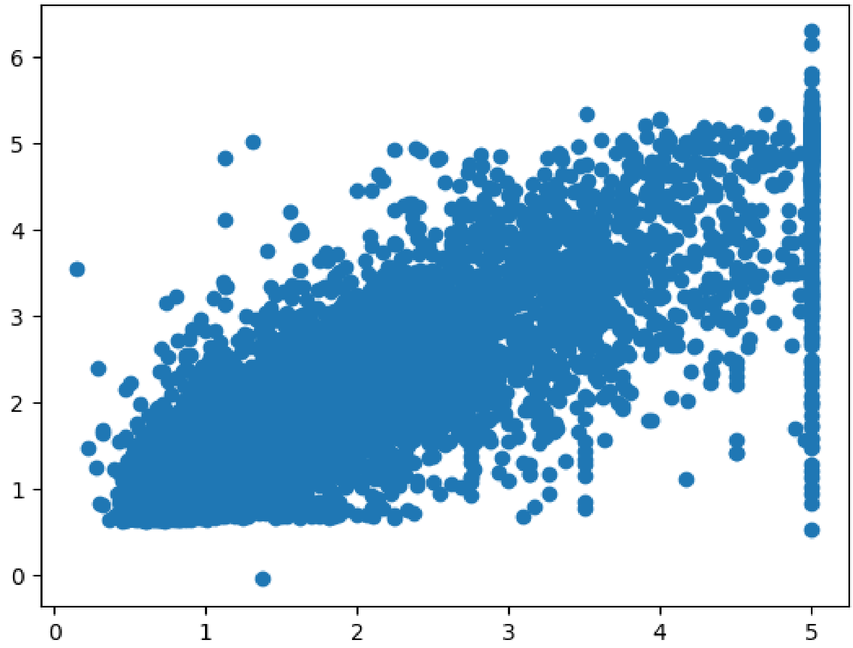
```
sns.distplot(y_test.values.reshape(-1,1)-prediction)
```

```
<ipython-input-50-be70e624d0c6>:1: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(y_test.values.reshape(-1,1)-prediction)
<Axes: ylabel='Density'>
```



```
plt.scatter(y_test,prediction)
```

```
<matplotlib.collections.PathCollection at 0x7e1038f835d0>
```




```
# MAE, MSE and RMSE
```

```
MAE: 0.517750902395806
MSE: 0.50007160344142
RMSE: 0.7071574106529748
```

```
#Classification
```

```
Shape of : X_train , y_train
```

```
(120, 4)
(120, 3)
```

```
ann.fit()
```

```
Epoch 1/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.4062 - loss: 1.2876
Epoch 2/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.3973 - loss: 1.2885
Epoch 3/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.3546 - loss: 1.2926
Epoch 4/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.3921 - loss: 1.3113
Epoch 5/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.3696 - loss: 1.2849
Epoch 6/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.4546 - loss: 1.2412
Epoch 7/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.4563 - loss: 1.2322
Epoch 8/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.4862 - loss: 1.2163
Epoch 9/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.4948 - loss: 1.1950
Epoch 10/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.4448 - loss: 1.2024
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.7485 - loss: 0.7116
Epoch 91/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.7281 - loss: 0.7048
Epoch 92/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.7825 - loss: 0.6821
Epoch 93/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.7919 - loss: 0.6674
Epoch 94/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8152 - loss: 0.6559
Epoch 95/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8329 - loss: 0.6476
Epoch 96/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8144 - loss: 0.6330
Epoch 97/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8102 - loss: 0.6436
Epoch 98/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8217 - loss: 0.6148
Epoch 99/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.7956 - loss: 0.6270
Epoch 100/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8269 - loss: 0.5879
<keras.src.callbacks.history.History at 0x7e1038c37850>
```

```
#Loss and accuracy
```

```
Test Loss: 0.5241
Test Accuracy: 0.9333
```

Learning Outcomes:

- Gain the ability to build and train ANN models for both regression and classification tasks in Python.
- Learn to preprocess data and evaluate model performance using key metrics like MSE, accuracy, and F1-score.

Experiment 3

Aim: Implementation of Convolution Neural Network for MRI Data Set in Python.

Objectives:

- To implement CNN model in Python on MRI images.
- To implement CNN model for classifying images and evaluate its performance using performance metrics.

Theory:

Introduction to Deep Learning and CNNs

Deep learning is a subset of machine learning that focuses on training artificial neural networks to learn from large amounts of data. Among deep learning architectures, **Convolutional Neural Networks (CNNs)** are particularly effective for analyzing image-based data, making them suitable for medical imaging tasks such as MRI (Magnetic Resonance Imaging) analysis.

MRI scans provide detailed internal body structures, helping in medical diagnosis, treatment planning, and research. However, manually interpreting MRI images is time-consuming and subject to variability. CNNs help automate and improve MRI analysis by detecting patterns, classifying abnormalities, and segmenting regions of interest.

Convolutional Neural Networks (CNNs) in Medical Imaging

CNNs are specialized deep learning models designed for image processing tasks. They use **convolutional layers** to extract spatial features from images, making them efficient in identifying patterns and structures within MRI scans. The fundamental components of a CNN include:

- **Convolutional Layers:** Apply filters to input images to extract features such as edges, textures, and shapes.
- **Pooling Layers:** Reduce the dimensionality of the data while preserving important features, improving computational efficiency.
- **Fully Connected Layers:** Flatten the extracted features and connect them to output neurons for classification or segmentation tasks.
- **Activation Functions (ReLU, Softmax, Sigmoid):** Introduce non-linearity into the model to improve learning capability.

CNNs have significantly advanced medical imaging applications, enabling automated tumor detection, brain segmentation, and disease classification from MRI datasets.

MRI Dataset and Its Importance

MRI datasets contain grayscale images that represent different tissues and structures within the human body. These datasets are crucial for training CNN models in various medical applications, such as:

- **Brain Tumor Classification** (Normal vs. Tumor)
- **Alzheimer's Disease Detection**
- **Organ Segmentation**

MRI datasets usually require preprocessing steps, including normalization, resizing, and augmentation, to improve model performance.

Implementation of CNN for MRI Dataset in Python

Python, with deep learning libraries such as **TensorFlow** and **Keras**, provides an efficient framework for implementing CNNs. The basic steps involved in developing a CNN for MRI image classification include:

1. **Loading the MRI Dataset:** Using libraries like `Pandas`, `NumPy`, or `TensorFlow datasets`.
2. **Preprocessing the Data:** Resizing images, normalizing pixel values, and augmenting the dataset.
3. **Building the CNN Model:** Defining layers such as convolutional, pooling, and fully connected layers.
4. **Compiling and Training the Model:** Using an optimizer (SGD, Adam) and loss function (categorical cross-entropy).
5. **Evaluating the Model:** Testing the model on unseen MRI images and assessing its accuracy.

Applications of CNNs in MRI Analysis

- **Brain Tumor Detection:** CNNs can classify MRI images as normal or containing tumors with high accuracy.
- **Alzheimer's Disease Prediction:** Identifying structural changes in brain MRI scans to diagnose neurodegenerative diseases.
- **Medical Image Segmentation:** Extracting regions of interest, such as tumors or organs, from MRI scans for precise analysis.

Conclusion

The implementation of a Convolutional Neural Network (CNN) for MRI image classification is a powerful deep learning application in medical imaging. CNNs efficiently learn spatial hierarchies of features and can significantly improve disease detection accuracy. With Python-based frameworks like TensorFlow and Keras, researchers and medical professionals can develop robust models to analyze MRI scans, contributing to advancements in medical diagnostics and healthcare.

Code:

```
import pandas as pd
import numpy as np
import cv2
import os
import random
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.image import imread

import tensorflow as tf
import tensorflow_hub as hub
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

from keras.models import Sequential
from keras.layers import Conv2D,Activation,MaxPooling2D,Dense,Flatten,
Dropout
from keras.losses import BinaryCrossentropy
try:
    from keras.optimizer import Adam
except:
    from tensorflow.keras.optimizers import Adam

from sklearn.model_selection import train_test_split
from sklearn.metrics import auc, roc_curve, log_loss, roc_auc_score,
auc, roc_curve, accuracy_score, confusion_matrix, cohen_kappa_score


import warnings
warnings.filterwarnings('ignore')


MAIN_DIR = "/content/brain_tumor_dataset/"

# control reproducibility with random seeds
seed_value = 1
np.random.seed(seed_value)
random.seed(seed_value)
tf.random.set_seed(seed_value)

# Define location of dataset
folder_no = MAIN_DIR + '/no'
folder_yes = MAIN_DIR + '/yes'

# Collect image paths
image_paths = []
```

```
for filepath in [folder_no, folder_yes]:
    for f in os.listdir(filepath)[:3]:  # Take the first 3 images per
folder
        image_paths.append(os.path.join(filepath, f))

fig, axes = plt.subplots(1, len(image_paths), figsize=(15, 5))

for ax, img_path in zip(axes, image_paths):
    image = imread(img_path)
    ax.imshow(image)

plt.tight_layout()
plt.show()

filepath = MAIN_DIR
images, labels, filenames = [], [], []

for gt in ['yes','no']:
    filepath_gt = filepath + gt
    for f in os.listdir(filepath_gt):
        filename = filepath_gt + '/' + f
        if '._' not in filename: # metadata files created by macos

            # load image
            photo =
tf.keras.utils.load_img(path=filename,target_size=(200, 200))

            # load image pixels
            image = imread(filename)
            resized_image = cv2.resize(image,dsize=(200,200))

            # in case of grayscale images
            if len(np.shape(resized_image)) > 2:
                # convert the image from COLOR_BGR2GRAY
                resized_image = cv2.cvtColor(resized_image,
cv2.COLOR_BGR2GRAY)

            # store to array
            images.append(resized_image)
            filenames.append(filename)

            if gt=='yes':
                label = 1
                labels.append(label)
            elif gt=='no':
                label = 0
                labels.append(label)
```

```
gtFr =
pd.DataFrame(labels).rename(columns={0:'gt'}).reset_index(drop=False).r
ename(columns={'index':'pID'})
gtFr['filename'] = filenames

# split into training and testing sets, stratifying by gt for equal
representation
trainFr, testFr = train_test_split(gtFr, test_size=0.2,
stratify=gtFr['gt'])

# store training/testing indices
trainFr['set'] = 'train'
testFr['set'] = 'test'

gtFr2 = pd.concat([trainFr,testFr],axis=0)
gtFr2.set_index(['filename']).to_csv('base_model_train_test.csv')

files_train = []
X_train = []
y_train = []


for idx, row in trainFr.iterrows():
    # load image pixels
    image = imread(row['filename'])

    # resize image to standard 200x200
    resized_image = cv2.resize(image,dsize=(200,200))

    # in case of grayscale images
    if len(np.shape(resized_image)) > 2:
        # convert the image from COLOR_BGR2GRAY
        resized_image = cv2.cvtColor(resized_image, cv2.COLOR_BGR2GRAY)

    # store to array
    X_train.append(resized_image)
    y_train.append(row['gt'])
    files_train.append(row['filename'])


# reshape data to fit model
X_train = np.array(X_train).reshape(len(trainFr),200,200,1)
y_train = np.array(y_train)

files_test = []
X_test = []
y_test = []
```

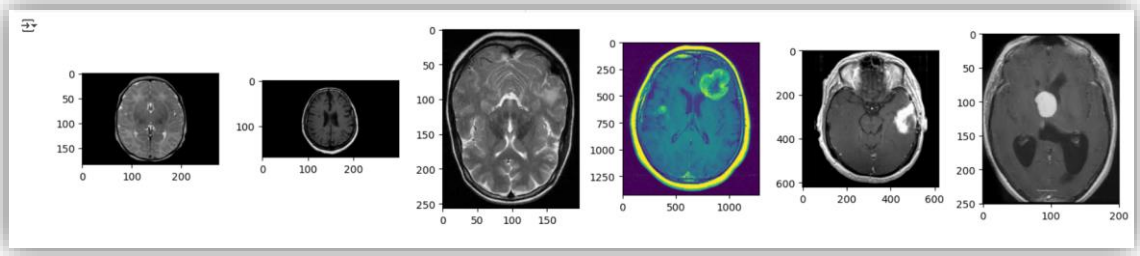
```
for idx, row in testFr.iterrows():
    # load image pixels
    image = imread(row['filename'])

    # resize image to standard 200x200
    resized_image = cv2.resize(image,dsize=(200,200))

    # in case of grayscale images
    if len(np.shape(resized_image)) > 2:
        # convert the image from COLOR_BGR2GRAY
        resized_image = cv2.cvtColor(resized_image, cv2.COLOR_BGR2GRAY)

    # store to array
    X_test.append(resized_image)
    y_test.append(row['gt'])
    files_test.append(row['filename'])

# reshape data to fit model
X_test = np.array(X_test).reshape(len(testFr),200,200,1)
y_test = np.array(y_test)

# create model
model = Sequential()

# add convolutional layer
model.add(Conv2D(64, kernel_size=3, activation='relu',
input_shape=(200,200,1))) # input is of shape (N, C) or (N, C, L) where
N is the batch size as before. However what does the C and L denote
here? It seems that C = number of features, L = number of channels,

# add max pooling layer
model.add(MaxPooling2D(pool_size=(2, 2)))

# add another convolutional layer
model.add(Conv2D(32, kernel_size=3, activation='relu'))

# add another max pooling layer
model.add(MaxPooling2D(pool_size=(2, 2)))

# flatten output (connect convolutional layers and dense layers)
model.add(Flatten())

# add a dense layer with 128 neurons and ReLU activation
model.add(Dense(128, activation='relu'))

# add dense layer
model.add(Dense(1, activation='sigmoid'))
```

```
# compile model using accuracy to measure model performance
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

history = model.fit(X_train, y_train, validation_data=(X_test, y_test),
epochs=30)

# Function to plot loss and accuracy in a single row
def plot_curves(history):
    """
    Returns loss and accuracy curves in a single row.
    """
    loss = history.history["loss"]
    val_loss = history.history["val_loss"]
    accuracy = history.history["accuracy"]
    val_accuracy = history.history["val_accuracy"]
    epochs = range(len(loss))

    # Create a single row with 2 plots (loss and accuracy)
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Plot loss
    axes[0].plot(epochs, loss, label="training_loss")
    axes[0].plot(epochs, val_loss, label="val_loss")
    axes[0].set_title("Loss")
    axes[0].set_xlabel("Epochs")
    axes[0].legend()

    # Plot accuracy
    axes[1].plot(epochs, accuracy, label="training_accuracy")
    axes[1].plot(epochs, val_accuracy, label="val_accuracy")
    axes[1].set_title("Accuracy")
    axes[1].set_xlabel("Epochs")
    axes[1].legend()

    plt.tight_layout()  # Adjust layout for better spacing
    plt.show()

plot_curves(history)

# make predictions
predictions = model.predict(X_test)

# store as probabilities
probabilities = [p[0] for p in predictions]
testFr['pred'] = probabilities

testFr.set_index(['pID']).to_csv('base_model_predictions.csv')
```

```
dataPos = testFr[testFr['gt']==1]['pred'].values
dataNeg = testFr[testFr['gt']==0]['pred'].values
dataAll = np.concatenate((dataPos, dataNeg))
lblArr = np.zeros(len(dataAll), dtype=bool)
lblArr[0:len(dataPos)] = True

fpr, tpr, thresholds = roc_curve(lblArr, dataAll, pos_label=True)
roc_auc = auc(fpr, tpr)

# invert comparison if (ROC<0.5) required
if roc_auc<0.5:
    lblArr = ~lblArr
    fpr, tpr, thresholds = roc_curve(lblArr, dataAll, pos_label=True)
    roc_auc = auc(fpr, tpr)
    print('inverting labels')

print('ROC AUC: {:0.2f}'.format(roc_auc))

# calculate best cut-off based on distance to top corner of ROC curve
distArr = np.sqrt(np.power(fpr, 2) + np.power((1 - tpr), 2))
cutoffIdx = np.argsort(distArr)[0]
cutoffTh = thresholds[cutoffIdx]

print('Cutoff threshold that maximizes sens/spec:
{:0.2f}'.format(cutoffTh))

lblOut = dataAll >= cutoffTh

acc = accuracy_score(lblArr, lblOut)
print('Accuracy: {:0.2f}'.format(acc))

sens = tpr[cutoffIdx]
print('Sensitivity: {:0.2f}'.format(sens))

spec = 1 - fpr[cutoffIdx]
print('Specificity: {:0.2f}'.format(spec))

kappa = cohen_kappa_score(lblOut, lblArr)
print('Cohen kappa score: {:0.2f}'.format(kappa))
```

Output:

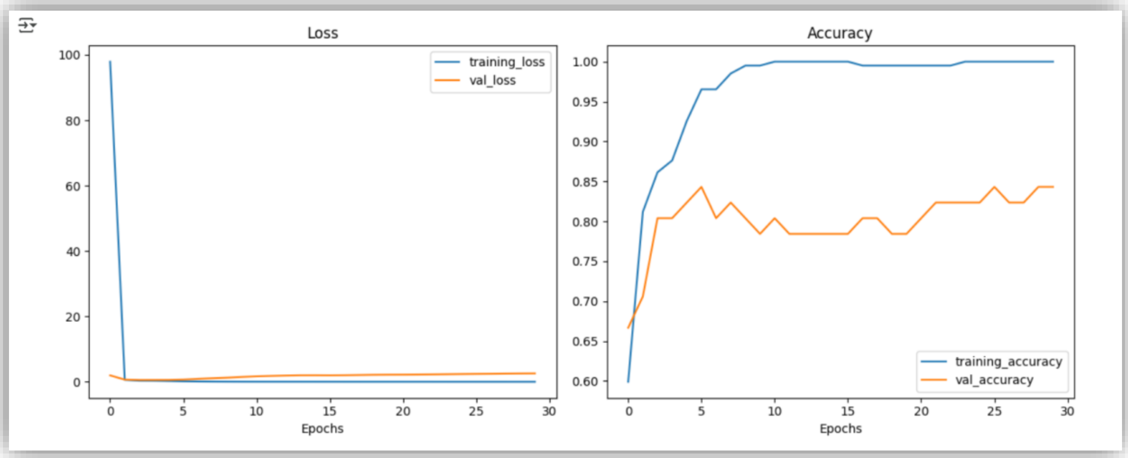
```
# First 3 images per folder
```



```
model.fit()
```

```
Epoch 1/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 23s 3s/step - accuracy: 0.6289 - loss: 109.7322 - val_accuracy: 0.6667 - val_loss: 1.9671
Epoch 2/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 46s 4s/step - accuracy: 0.8392 - loss: 0.5811 - val_accuracy: 0.7059 - val_loss: 0.6620
Epoch 3/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 19s 3s/step - accuracy: 0.8475 - loss: 0.3732 - val_accuracy: 0.8039 - val_loss: 0.5536
Epoch 4/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 21s 3s/step - accuracy: 0.8424 - loss: 0.3469 - val_accuracy: 0.8039 - val_loss: 0.5629
Epoch 5/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 27s 3s/step - accuracy: 0.9068 - loss: 0.2688 - val_accuracy: 0.8235 - val_loss: 0.5739
Epoch 6/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 33s 3s/step - accuracy: 0.9627 - loss: 0.1645 - val_accuracy: 0.8431 - val_loss: 0.6756
Epoch 7/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 20s 3s/step - accuracy: 0.9596 - loss: 0.0938 - val_accuracy: 0.8039 - val_loss: 0.9022
Epoch 8/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 21s 3s/step - accuracy: 0.9855 - loss: 0.0554 - val_accuracy: 0.8235 - val_loss: 1.0873
Epoch 9/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 21s 3s/step - accuracy: 0.9973 - loss: 0.0370 - val_accuracy: 0.8039 - val_loss: 1.2648
Epoch 10/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 20s 2s/step - accuracy: 0.9973 - loss: 0.0269 - val_accuracy: 0.7843 - val_loss: 1.4849
Epoch 11/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 18s 2s/step - accuracy: 1.0000 - loss: 0.0211 - val_accuracy: 0.8039 - val_loss: 1.6774
Epoch 12/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 18s 3s/step - accuracy: 1.0000 - loss: 0.0188 - val_accuracy: 0.7843 - val_loss: 1.8015
Epoch 13/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 21s 3s/step - accuracy: 1.0000 - loss: 0.0178 - val_accuracy: 0.7843 - val_loss: 1.9148
Epoch 14/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 22s 3s/step - accuracy: 1.0000 - loss: 0.0170 - val_accuracy: 0.7843 - val_loss: 1.9916
Epoch 15/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 18s 3s/step - accuracy: 1.0000 - loss: 0.0161 - val_accuracy: 0.7843 - val_loss: 2.0005
Epoch 16/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 20s 3s/step - accuracy: 1.0000 - loss: 0.0142 - val_accuracy: 0.7843 - val_loss: 1.9772
Epoch 17/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 18s 2s/step - accuracy: 0.9951 - loss: 0.0112 - val_accuracy: 0.8039 - val_loss: 2.0106
Epoch 18/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 22s 3s/step - accuracy: 0.9951 - loss: 0.0097 - val_accuracy: 0.8039 - val_loss: 2.0763
Epoch 19/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 20s 3s/step - accuracy: 0.9951 - loss: 0.0093 - val_accuracy: 0.7843 - val_loss: 2.1431
Epoch 20/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 20s 3s/step - accuracy: 0.9951 - loss: 0.0083 - val_accuracy: 0.7843 - val_loss: 2.1772
Epoch 21/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 22s 2s/step - accuracy: 0.9951 - loss: 0.0074 - val_accuracy: 0.8039 - val_loss: 2.1870
Epoch 22/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 20s 3s/step - accuracy: 0.9951 - loss: 0.0065 - val_accuracy: 0.8235 - val_loss: 2.2192
Epoch 23/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 21s 3s/step - accuracy: 0.9951 - loss: 0.0056 - val_accuracy: 0.8235 - val_loss: 2.2601
Epoch 24/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 20s 3s/step - accuracy: 1.0000 - loss: 0.0046 - val_accuracy: 0.8235 - val_loss: 2.3132
Epoch 25/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 20s 2s/step - accuracy: 1.0000 - loss: 0.0037 - val_accuracy: 0.8235 - val_loss: 2.3698
Epoch 26/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 21s 3s/step - accuracy: 1.0000 - loss: 0.0028 - val_accuracy: 0.8431 - val_loss: 2.4179
Epoch 27/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 17s 3s/step - accuracy: 1.0000 - loss: 0.0022 - val_accuracy: 0.8235 - val_loss: 2.4562
Epoch 28/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 19s 3s/step - accuracy: 1.0000 - loss: 0.0016 - val_accuracy: 0.8235 - val_loss: 2.5058
Epoch 29/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 19s 2s/step - accuracy: 1.0000 - loss: 0.0013 - val_accuracy: 0.8431 - val_loss: 2.5441
Epoch 30/30
7/7 ━━━━━━━━━━━━━━━━━━━━ 21s 3s/step - accuracy: 1.0000 - loss: 9.7393e-04 - val_accuracy: 0.8431 - val_loss: 2.5706
```

```
plot_curves(history)
```




```
# Evaluation Metrics
```

```
ROC AUC: 0.84
Cutoff threshold that maximizes sens/spec: 0.75
Accuracy: 0.84
Sensitivity: 0.90
Specificity: 0.75
Cohen kappa score: 0.67
```

Learning Outcomes:

- Gain the ability to build and train CNN models for classification tasks on images in Python.
- Learnt to preprocess and scale image data to train CNN model and evaluate model performance using key metrics like accuracy, sensitivity and kappa score.

Experiment 4

Aim: Implementation of Autoencoders for dimensionality reduction in Python.

Objectives:

- To implement Autoencoders in Python for dimensionality reduction.
- To train an Autoencoder using TensorFlow and Keras and analyze its reconstruction ability.

Theory:

Introduction to Deep Learning and Dimensionality Reduction

Deep learning is a subset of machine learning that enables computers to learn from large datasets using artificial neural networks. One of the critical challenges in machine learning is dealing with **high-dimensional data**, which can lead to increased computational costs and reduced model performance due to the "curse of dimensionality." **Dimensionality reduction** is a technique used to transform high-dimensional data into a lower-dimensional representation while preserving important features.

Among various dimensionality reduction techniques, **Autoencoders** have gained significant importance. Autoencoders are a type of neural network architecture used to learn efficient data representations in an **unsupervised manner**. They compress input data into a **lower-dimensional latent space** and then reconstruct it back, capturing the most critical features of the data.

What are Autoencoders?

An **Autoencoder (AE)** is a neural network designed to encode input data into a lower-dimensional space (encoding) and then reconstruct the original input (decoding). It consists of two main components:

1. **Encoder:** Reduces the input dimensions by learning compressed representations.
2. **Decoder:** Reconstructs the input data from the compressed representation.

The objective of an autoencoder is to minimize the **reconstruction loss**, typically measured using **Mean Squared Error (MSE)** or **Binary Cross-Entropy Loss**.

Applications of Autoencoders in Dimensionality Reduction

Autoencoders are widely used for dimensionality reduction and feature extraction, especially in cases where linear methods like PCA (Principal Component Analysis) are insufficient. Some key applications include:

- **Data Compression:** Efficiently encoding high-dimensional data into a compact representation.
- **Noise Reduction (Denoising Autoencoders):** Removing noise from images or signals by learning a cleaner representation.

- **Anomaly Detection:** Identifying rare patterns by reconstructing normal data patterns and detecting deviations.
- **Feature Extraction:** Learning meaningful representations for downstream machine learning tasks.

Implementation of Autoencoders for Dimensionality Reduction in Python

Autoencoders can be implemented using **TensorFlow** and **Keras** in Python. The basic steps include:

1. **Loading the Dataset:** Common datasets such as **MNIST** or **CIFAR-10** are used to train autoencoders.
2. **Preprocessing the Data:** Normalization and reshaping of input images.
3. **Building the Autoencoder Model:**
    - A simple autoencoder consists of an **encoder (Dense layers)** followed by a **decoder (Dense layers)**.
4. **Training the Autoencoder:** Using an **Adam optimizer** and **Mean Squared Error (MSE) loss**.
5. **Evaluating the Performance:** Checking the quality of reconstructed images and analyzing dimensionality reduction.

Advantages of Autoencoders for Dimensionality Reduction

Compared to traditional techniques like PCA, autoencoders offer several benefits:

- **Non-linearity Handling:** Autoencoders can learn complex, non-linear relationships in data.
- **Better Representation Learning:** Captures hierarchical and meaningful features.
- **Scalability:** Works efficiently on large datasets without significant performance degradation.
- **Feature Learning without Labels:** Unlike supervised learning, autoencoders require no labeled data.

Conclusion

Autoencoders provide an efficient way to perform **dimensionality reduction**, especially for high-dimensional datasets. They are widely used in feature extraction, denoising, and anomaly detection tasks. With deep learning frameworks like **TensorFlow and Keras**, implementing autoencoders in Python has become straightforward. Their ability to learn efficient representations without supervision makes them a powerful tool in modern machine learning and deep learning applications.

Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, LeakyReLU
from tensorflow.keras.optimizers import RMSprop
from sklearn.manifold import TSNE

# Load Dataset
file_path = "train.csv"  # Change if needed
df = pd.read_csv(file_path)
print(df.head())

# EDA
print(df.info())
print(df.describe())
print(df.isnull().sum().sum(), "missing values")

# Distribution of Target Variable
sns.histplot(df['target'], bins=30, kde=True)
plt.title("Target Variable Distribution")
plt.show()

# Remove ID and extract features
df.drop(columns=['ID_code'], inplace=True)
X = df.drop(columns=['target']).values  # Features
y = df['target'].values  # Target

# Normalize Data
scaler = StandardScaler()
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Handle NaN or Inf values
X_train = np.nan_to_num(X_train)
X_test = np.nan_to_num(X_test)

# Autoencoder Architecture
encoding_dim = 10  # Dimension reduction size
input_dim = X_train.shape[1]
```

```
input_layer = Input(shape=(input_dim,))
encoded = Dense(64)(input_layer)
encoded = LeakyReLU(alpha=0.1)(encoded)
encoded = Dense(32)(encoded)
encoded = LeakyReLU(alpha=0.1)(encoded)
encoded = Dense(encoding_dim, activation='linear')(encoded)

decoded = Dense(32)(encoded)
decoded = LeakyReLU(alpha=0.1)(decoded)
decoded = Dense(64)(decoded)
decoded = LeakyReLU(alpha=0.1)(decoded)
decoded = Dense(input_dim, activation='linear')(decoded)

# Define Autoencoder
autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer=RMSprop(learning_rate=0.0001),
loss='mae')

# Train Autoencoder
autoencoder.fit(X_train, X_train, epochs=50, batch_size=256,
shuffle=True, validation_data=(X_test, X_test))

# Extract Encoder
encoder = Model(input_layer, encoded)
X_train_encoded = encoder.predict(X_train)
X_test_encoded = encoder.predict(X_test)

# t-SNE Visualization
tsne = TSNE(n_components=2, random_state=42)
X_embedded = tsne.fit_transform(X_test_encoded)
plt.scatter(X_embedded[:, 0], X_embedded[:, 1],
c=y_test[:len(X_embedded)], cmap='coolwarm', alpha=0.5)
plt.colorbar()
plt.title("t-SNE Visualization of Encoded Features")
plt.show()

print("Original feature shape:", X_train.shape)
print("Reduced feature shape:", X_train_encoded.shape)
```

Output:

```
# Load dataset - df.head()
```

```
  ID_code  target    var_0   var_1    var_2   var_3    var_4   var_5   var_6  \
0 train_0       0   8.9255 -6.7863  11.9081  5.0930  11.4607 -9.2834  5.1187
1 train_1       0  11.5006 -4.1473  13.8588  5.3890  12.3622  7.0433  5.6208
2 train_2       0   8.6093 -2.7457  12.0805  7.8928  10.5825 -9.0837  6.9427
3 train_3       0  11.0604 -2.1518   8.9522  7.1957  12.5846 -1.8361  5.8428
4 train_4       0   9.8369 -1.4834  12.8746  6.6375  12.2772  2.4486  5.9405

    var_7  ...  var_190  var_191  var_192  var_193  var_194  var_195  \
0 18.6266  ...   4.4354   3.9642   3.1364   1.6910  18.5227  -2.3978
1 16.5338  ...   7.6421   7.7214   2.5837  10.9516  15.4305   2.0339
2 14.6155  ...   2.9057   9.7905   1.6704   1.6858  21.6042   3.1417
3 14.9250  ...   4.4666   4.7433   0.7178   1.4214  23.0347  -1.2706
4 19.2514  ...  -1.4905   9.5214  -0.1508   9.1942  13.2876  -1.5121

  var_196  var_197  var_198  var_199
0  7.8784   8.5635  12.7803  -1.0914
1  8.1267   8.7889  18.3560   1.9518
2 -6.5213   8.2675  14.7222   0.3965
3 -2.9275  10.2922  17.9697  -8.9996
4  3.9267   9.5031  17.9974  -8.8104

[5 rows x 202 columns]
```

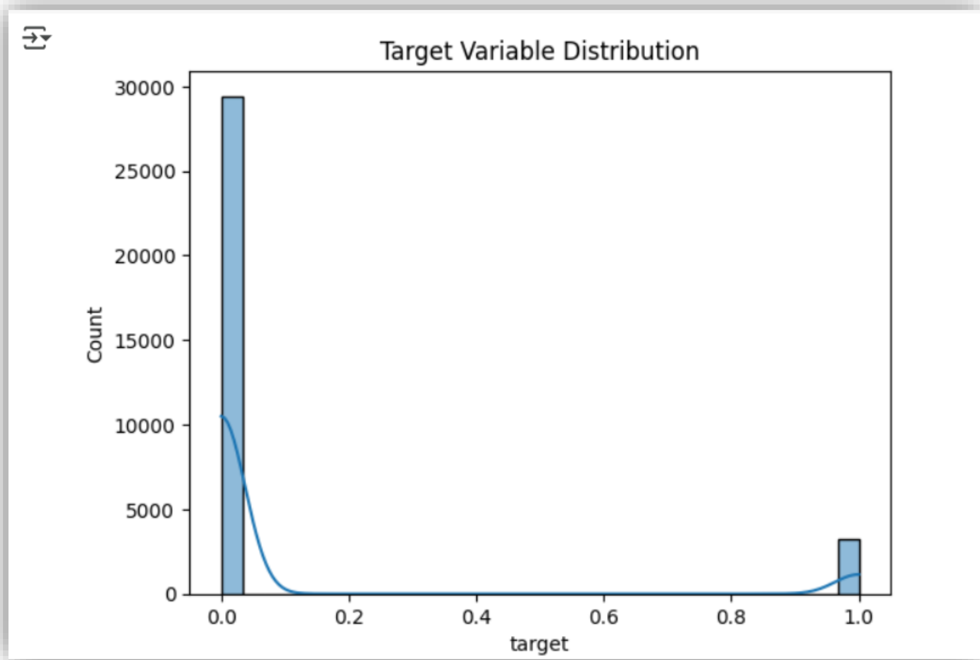
```
# EDA
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32639 entries, 0 to 32638
Columns: 202 entries, ID_code to var_199
dtypes: float64(200), int64(1), object(1)
memory usage: 50.3+ MB
None
             target         var_0         var_1         var_2         var_3  \
count  32639.000000  32639.000000  32639.000000  32639.000000  32639.000000
mean       0.098410     10.657325     -1.656794     10.699796      6.790028
std        0.297873      3.051383      4.068756      2.624799      2.050666
min        0.000000      0.597900    -13.960900      2.898200     -0.040200
25%        0.000000      8.426950     -4.782650      8.733400      5.234850
50%        0.000000     10.506700     -1.634500     10.546800      6.828400
75%        0.000000     12.739650      1.338500     12.489050      8.331100
max        1.000000     19.325900     10.335600     18.347700     12.977300

              var_4         var_5         var_6         var_7         var_8  \
count  32639.000000  32639.000000  32639.000000  32639.000000  32639.000000
mean      11.084600     -5.073672      5.406663     16.571058      0.282771
std        1.630488      7.873897      0.868575      3.416607      3.324497
min        5.918800    -29.013300      2.385700      5.749400     -9.991100
25%        9.884800    -11.216150      4.765500     13.956550     -2.285150
50%       11.105400     -4.877700      5.381800     16.501000      0.364600
75%       12.280650      0.937500      6.002600     19.113800      2.919400
max       16.671400     17.251600      8.355600     27.597700      9.482200
```

```
              ...        var_190       var_191       var_192       var_193  \
count         ...   32638.000000  32638.000000  32638.000000  32638.000000
mean          ...       3.212316      7.449308      1.930695      3.305760
std           ...       4.566513      3.018351      1.475990      4.000176
min           ...     -11.906900     -2.343000     -3.566800    -10.173300
25%           ...      -0.089450      5.159075      0.904125      0.554625
50%           ...       3.209900      7.368550      1.918050      3.365950
75%           ...       6.390175      9.524075      2.944275      6.157350
max           ...      18.078900     16.409400      7.647600     16.782600
```

```
            var_194       var_195       var_196       var_197       var_198  \
count  32638.000000  32638.000000  32638.000000  32638.000000  32638.000000
mean      18.006183     -0.159380      2.307928      8.915007     15.877706
std        3.141888      1.423152      5.451356      0.921477      3.001748
min        8.694400     -5.048100    -13.328200      6.047600      6.644800
25%       15.637000     -1.185575     -1.946275      8.260700     13.846600
50%       17.971950     -0.193200      2.413000      8.895600     15.915950
75%       20.427700      0.807375      6.538500      9.598075     18.083100
max       27.528400      4.255700     18.321500     12.000400     25.442200

            var_199
count  32638.000000
mean      -3.371388
std       10.403095
min      -36.325100
25%      -11.254825
50%       -2.883850
75%        4.751800
max       26.468800

[8 rows x 201 columns]
78 missing values
```

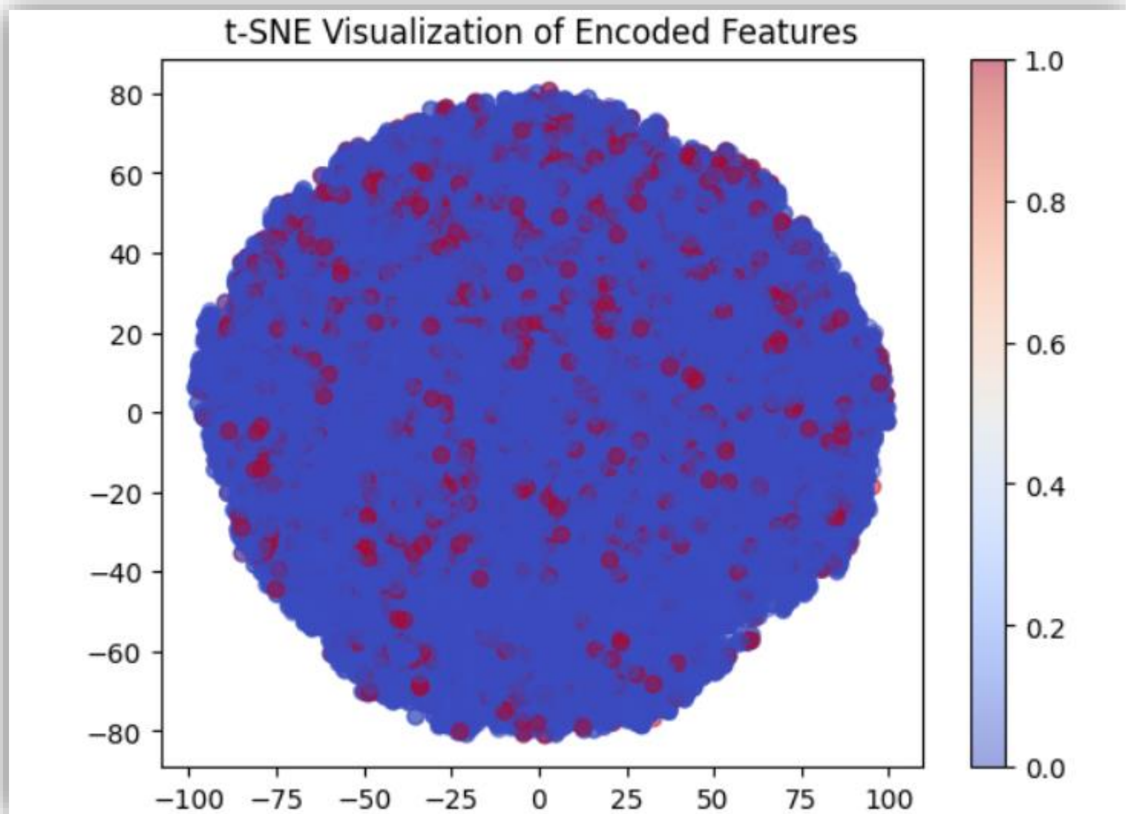
```
# Distribution of Target Variable
```



```
# autoencoder.fit()
```

```
Epoch 1/50
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - loss: 0.8247 - val_loss: 0.8208
Epoch 2/50
625/625 ━━━━━━━━━━━━━━━━━━━━ 6s 7ms/step - loss: 0.8202 - val_loss: 0.8188
Epoch 3/50
625/625 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.8179 - val_loss: 0.8159
Epoch 4/50
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - loss: 0.8150 - val_loss: 0.8127
Epoch 5/50
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - loss: 0.8119 - val_loss: 0.8094
Epoch 6/50
625/625 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 0.8084 - val_loss: 0.8065
Epoch 7/50
625/625 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 0.8058 - val_loss: 0.8045
Epoch 8/50
625/625 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 0.8041 - val_loss: 0.8032
Epoch 9/50
625/625 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - loss: 0.8028 - val_loss: 0.8023
Epoch 10/50
625/625 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 0.8021 - val_loss: 0.8017
```

```
# t-SNE Visualization
```



```
# Original vs Reduced
```

```
Original feature shape: (160000, 200)
Reduced feature shape: (160000, 10)
```

Learning Outcomes:

- Understand the concept of autoencoders and their role in dimensionality reduction.
- Learn to preprocess and normalize high-dimensional data for efficient training.

Experiment 5

Aim: Improving Autocoder's Performance using convolution layers in Python (MNIST Dataset to be utilized).

Objectives:

- To implement Autoencoders in Python on Image Dataset.
- To train an Autoencoder using TensorFlow and Keras and analyze its reconstruction ability.

Theory:

1. Introduction

Autoencoders are a type of artificial neural network used for unsupervised learning of efficient codings in data. They are primarily utilized for dimensionality reduction, denoising, and feature extraction. In this experiment, we apply autoencoders to the MNIST dataset, a widely used benchmark dataset for handwritten digit recognition. We first implement a basic autoencoder using fully connected layers and later enhance its performance by employing convolutional layers.

2. Autoencoders: Concept and Working Mechanism

Autoencoders consist of two main components:

1. **Encoder**: This part compresses the input data into a lower-dimensional representation (latent space).
2. **Decoder**: This part reconstructs the input data from the compressed representation.

Mathematically, an autoencoder maps an input to a compressed representation using an encoding function , and then reconstructs using a decoding function.

The training objective is to minimize the reconstruction loss between and , usually measured by Mean Squared Error (MSE) or Binary Cross-Entropy (BCE).

3. Application of Autoencoders on MNIST Dataset

3.1 Dataset Preparation

The MNIST dataset consists of 60,000 training images and 10,000 test images of handwritten digits (0-9), each of size 28x28 pixels. The data is normalized to the range to improve training efficiency.

3.2 Fully Connected Autoencoder Implementation

A simple autoencoder with fully connected layers is constructed as follows:

- **Encoder**: Input (784 dimensions) → Dense(128, ReLU) → Dense(64, ReLU) → Latent Space (32 dimensions)
- **Decoder**: Dense(64, ReLU) → Dense(128, ReLU) → Output (784 dimensions, Sigmoid)

After training with Adam optimizer and MSE loss function, the autoencoder is able to reconstruct the input images reasonably well. However, due to the dense architecture, spatial relationships in images are not efficiently captured.

4. Improving Autoencoder Performance with Convolutional Layers

To improve performance, a Convolutional Autoencoder (CAE) is implemented. Unlike fully connected layers, convolutional layers maintain spatial hierarchies, making them better suited for image reconstruction tasks.

4.1 Convolutional Autoencoder Architecture

- **Encoder**: Conv2D(32 filters, 3x3, ReLU) → MaxPooling2D(2x2) → Conv2D(64 filters, 3x3, ReLU) → MaxPooling2D(2x2)
- **Decoder**: Conv2DTranspose(64 filters, 3x3, ReLU) → UpSampling2D(2x2) → Conv2DTranspose(32 filters, 3x3, ReLU) → UpSampling2D(2x2) → Conv2DTranspose(1 filter, 3x3, Sigmoid)

4.2 Performance Comparison

The CAE significantly improves the quality of reconstructed images compared to the fully connected autoencoder. The main advantages include:

- **Preservation of spatial features**, reducing loss of information.
- **Faster convergence** due to fewer trainable parameters.
- **Better generalization** for unseen images.

5. Conclusion

Autoencoders provide a powerful method for image compression and feature extraction. While fully connected autoencoders serve as a baseline, convolutional autoencoders significantly enhance performance by leveraging spatial hierarchies. Future work can include Variational Autoencoders (VAEs) for more structured latent spaces and application of these techniques to real-world datasets beyond MNIST.

Code:

```
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

(X_train, _), (X_test, _) = tf.keras.datasets.fashion_mnist.load_data()

X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0

X_train_flat = X_train.reshape(-1, 28 * 28)
X_test_flat = X_test.reshape(-1, 28 * 28)

(X_train, y_train), (X_test, y_test) =
tf.keras.datasets.fashion_mnist.load_data()

print(f"Train set shape: {X_train.shape}, Labels: {y_train.shape}")
print(f"Test set shape: {X_test.shape}, Labels: {y_test.shape}")

class_labels = [
    "T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
    "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"
]

plt.figure(figsize=(10, 6))
for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.imshow(X_train[i], cmap="gray")
    plt.title(class_labels[y_train[i]])
    plt.axis("off")
plt.tight_layout()
plt.show()

class_counts = pd.Series(y_train).value_counts().sort_index()

plt.figure(figsize=(8, 5))
sns.barplot(x=class_counts.index, y=class_counts.values,
palette="coolwarm")
plt.xticks(ticks=range(10), labels=class_labels, rotation=45)
plt.xlabel("Class Labels")
plt.ylabel("Count")
plt.title("Distribution of Fashion MNIST Classes")
plt.show()
plt.figure(figsize=(8, 5))
```

```
sns.histplot(X_train.flatten(), bins=50, kde=True, color="purple")
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")
plt.title("Pixel Intensity Distribution in Fashion MNIST")
plt.show()
# Convert images to tuples for easy comparison
unique_images = set([tuple(image.flatten()) for image in X_train])

print(f"Total Images: {len(X_train)}")
print(f"Unique Images: {len(unique_images)}")
print(f"Duplicate Images: {len(X_train) - len(unique_images)}")

input_dim = 28 * 28  # 784 pixels
encoding_dim = 64  # Reduced dimension

# Define the autoencoder
input_layer = layers.Input(shape=(input_dim,))
encoded = layers.Dense(128, activation='relu')(input_layer)
encoded = layers.Dense(encoding_dim, activation='relu')(encoded)

decoded = layers.Dense(128, activation='relu')(encoded)
decoded = layers.Dense(input_dim, activation='sigmoid')(decoded)  #
Output same shape as input

autoencoder = models.Model(input_layer, decoded)
encoder = models.Model(input_layer, encoded)  # Encoder for
dimensionality reduction

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

# Train the model
autoencoder.fit(X_train_flat, X_train_flat, epochs=20, batch_size=256,
validation_data=(X_test_flat, X_test_flat))

# Reshape for CNN (add channel dimension)
X_train = X_train.reshape(-1, 28, 28, 1)
X_test = X_test.reshape(-1, 28, 28, 1)

# Convolutional Autoencoder
input_layer = layers.Input(shape=(28, 28, 1))

# Encoder
x = layers.Conv2D(32, (3, 3), activation='relu',
padding='same')(input_layer)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
```

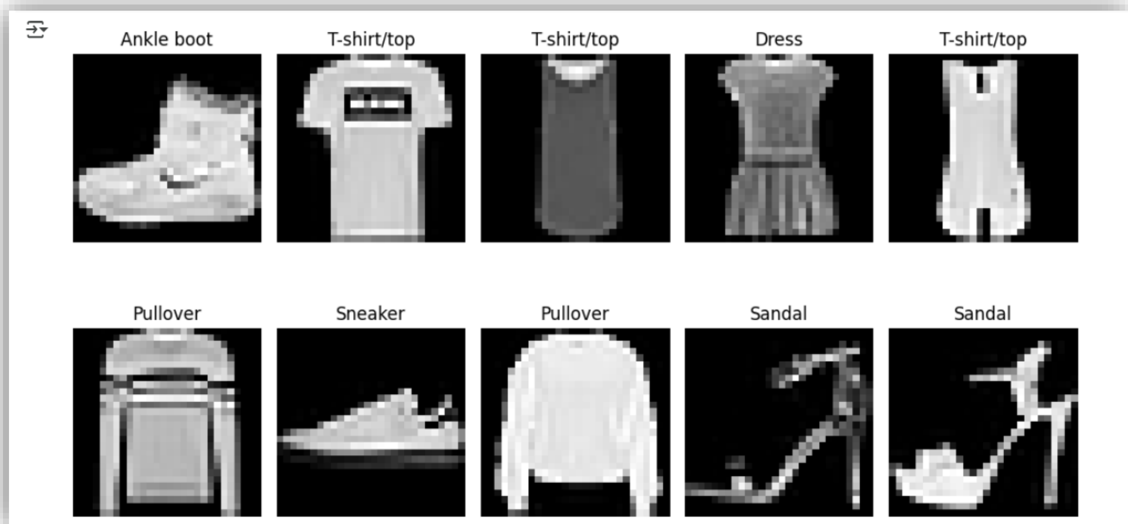
```
encoded = layers.Conv2D(128, (3, 3), activation='relu',
padding='same')(x)

# Decoder
x = layers.Conv2DTranspose(64, (3, 3), activation='relu',
padding='same')(encoded)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2DTranspose(32, (3, 3), activation='relu',
padding='same')(x)
x = layers.UpSampling2D((2, 2))(x)

decoded = layers.Conv2DTranspose(1, (3, 3), activation='sigmoid',
padding='same')(x)

# Define model
autoencoder = models.Model(input_layer, decoded)

# Compile and train
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

autoencoder.fit(X_train, X_train, epochs=20, batch_size=256,
validation_data=(X_test, X_test))

encoder = models.Model(input_layer, encoded)
X_encoded = encoder.predict(X_test)
print("Encoded shape:", X_encoded.shape)

n = 10  # Number of images to display
decoded_imgs = autoencoder.predict(X_test)

plt.figure(figsize=(20, 4))
for i in range(n):
    # Original images
    plt.subplot(2, n, i + 1)
    plt.imshow(X_test[i].reshape(28, 28), cmap='gray')
    plt.axis('off')

    # Reconstructed images
    plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
    plt.axis('off')

plt.show()
```

Output:

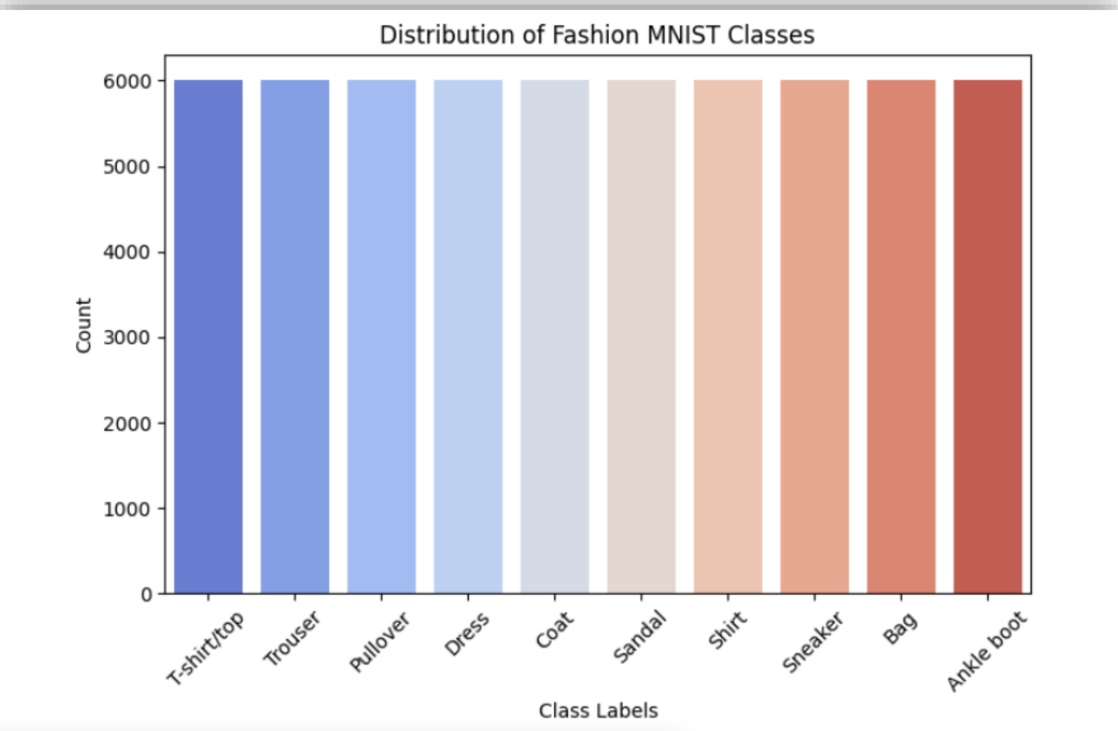
```
# Check dataset shape
```

```
Train set shape: (60000, 28, 28), Labels: (60000,)
Test set shape: (10000, 28, 28), Labels: (10000,)
```

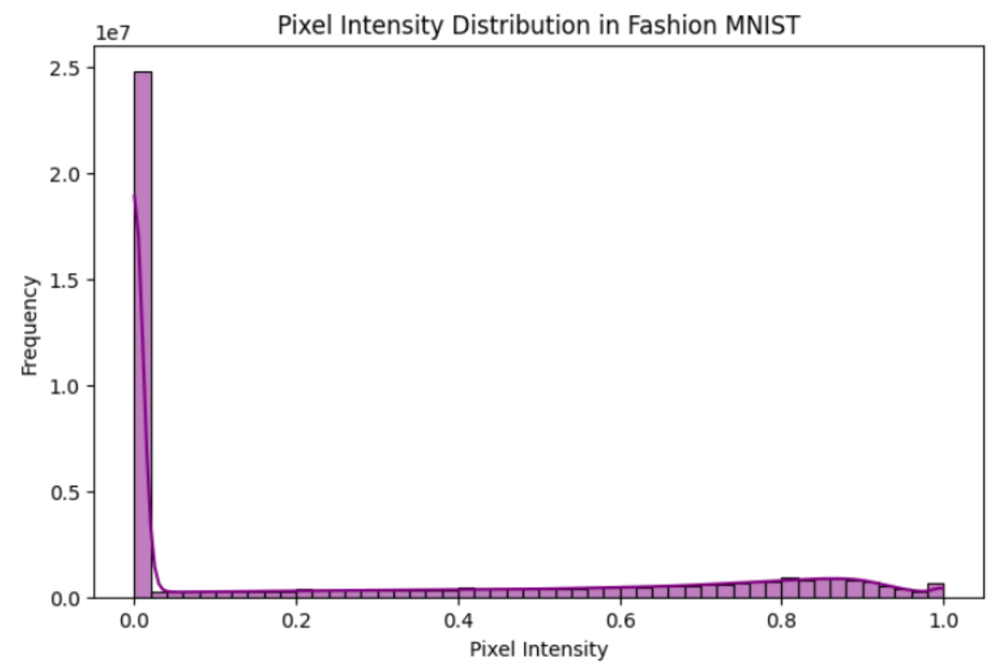
```
# Plot sample images with labels
```




```
# Distribution of Fashion MNIST Classes
```



```
# Pixel Intensity Distribution in Fashion MNIST
```




```
# Image count
```

```
Total Images: 60000
Unique Images: 60000
Duplicate Images: 0
```

```
# Train the model
```

```
Epoch 1/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - loss: 0.0732 - val_loss: 0.0230
Epoch 2/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0216 - val_loss: 0.0182
Epoch 3/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0175 - val_loss: 0.0162
Epoch 4/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - loss: 0.0156 - val_loss: 0.0148
Epoch 5/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - loss: 0.0145 - val_loss: 0.0140
Epoch 6/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - loss: 0.0136 - val_loss: 0.0132
Epoch 7/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - loss: 0.0129 - val_loss: 0.0130
Epoch 8/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 0.0125 - val_loss: 0.0121
Epoch 9/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - loss: 0.0119 - val_loss: 0.0119
Epoch 10/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 0.0115 - val_loss: 0.0114
```

```
# Convolutional Autoencoder
```

```
Epoch 1/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 104s 435ms/step - loss: 0.3735 - val_loss: 0.2807
Epoch 2/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 144s 444ms/step - loss: 0.2760 - val_loss: 0.2716
Epoch 3/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 142s 446ms/step - loss: 0.2690 - val_loss: 0.2681
Epoch 4/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 104s 444ms/step - loss: 0.2646 - val_loss: 0.2646
Epoch 5/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 140s 436ms/step - loss: 0.2629 - val_loss: 0.2624
Epoch 6/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 142s 436ms/step - loss: 0.2604 - val_loss: 0.2609
Epoch 7/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 142s 437ms/step - loss: 0.2585 - val_loss: 0.2594
Epoch 8/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 102s 436ms/step - loss: 0.2576 - val_loss: 0.2584
Epoch 9/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 142s 434ms/step - loss: 0.2564 - val_loss: 0.2573
Epoch 10/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 141s 432ms/step - loss: 0.2553 - val_loss: 0.2564
```

```
# Encoded shape:
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step
Encoded shape: (10000, 7, 7, 128)
```

```
# Original vs Reconstructed
```

Learning Outcomes:

- Understand the concept of autoencoders and their application in image reconstruction.
- Learn to preprocess and normalize the MNIST dataset for improved training efficiency.