Neural Networks and Deep Learning Lab

DEPARTMENT OF COMPUTER ENGINEERING & APPLICATIONS

Institute of Engineering & Technology

Lab File

Subject Name: Neural Networks and Deep Learning Lab

Subject Code: BCSE0452

Name: Ananya Agrawal

University Roll No.: 2115800002

Class Roll No.: 01

Course: B.Tech. CSE Hons.

Year: III Semester: V

Submitted to: Dr. Gaurav Bathla

Department of Computer Engineering & Applications

Neural Networks and Deep Learning Lab Experiment List

Experiment 1: Implementation of Artificial Neural Network using Input layer, hidden layers, and output layer. (using softmax for output layer)

An artificial neural network (ANN) is a computational model inspired by the biological structure and function of the human brain. It consists of interconnected nodes, called neurons, arranged in layers. Information flows forward from the input layer to the output layer through the hidden layers. Each neuron performs a weighted sum of its inputs and applies an activation function to the result.

Here are the key components and their functions:

- **Input layer:** Receives the input data and distributes it to the neurons in the first hidden layer.
- **Hidden layers:** Perform computations and extract features from the input data. Each neuron in a hidden layer combines its inputs from the previous layer, applies a non-linear activation function (e.g., ReLU, sigmoid) to introduce non-linearity, and then transmits the output to the next layer.
- **Output layer:** Generates the final output of the network. For multi-class classification, the output layer typically uses a softmax activation function. This function transforms the weighted sum of inputs into probabilities representing the likelihood of each class.

Softmax Activation Function:

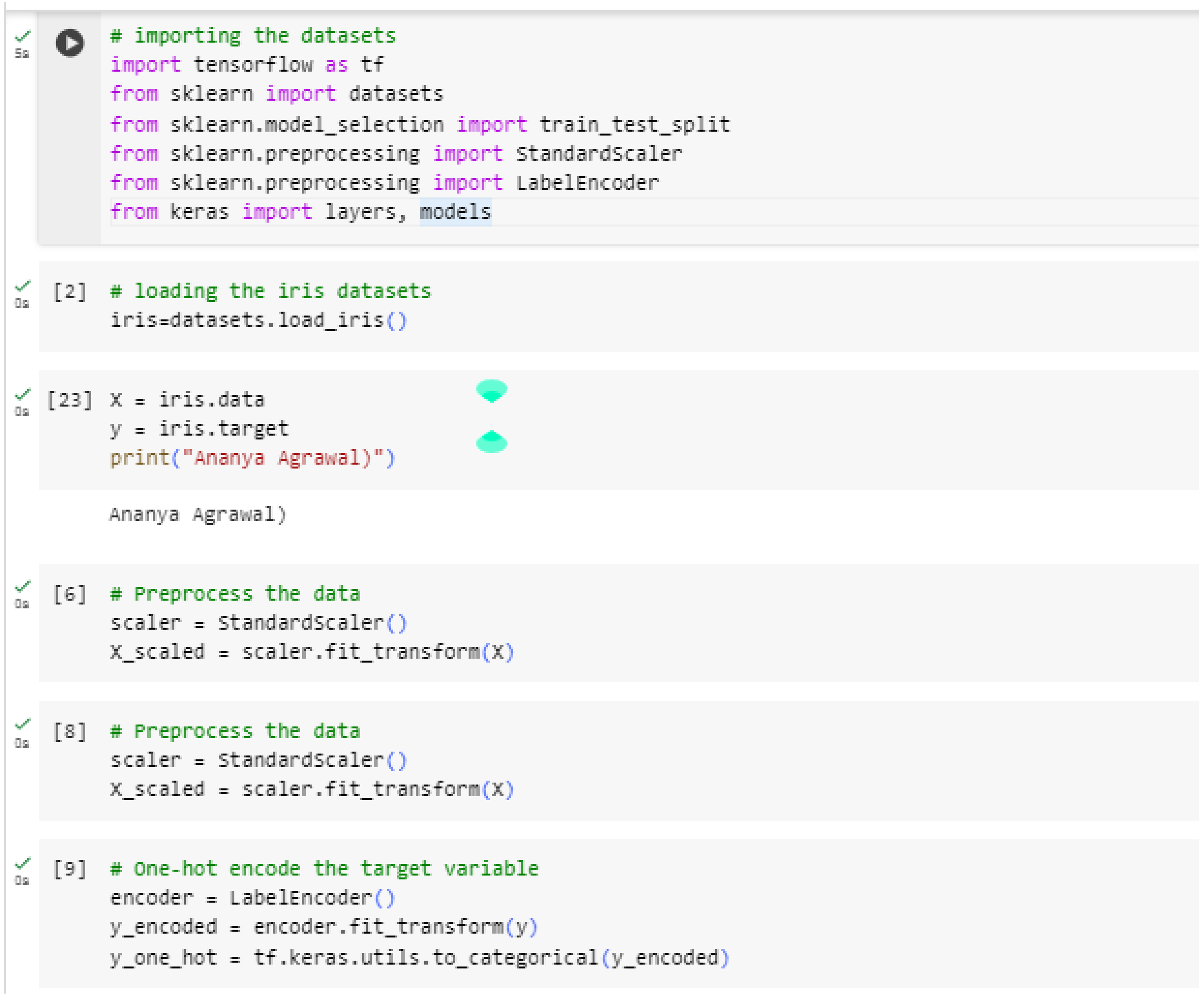
The softmax function is a popular choice for the output layer of ANNs used for multi-class classification. It takes a vector of real numbers as input and outputs a probability distribution over C classes (where C is the number of classes).

Here's the formula for softmax:

σ(x)_i = e^x_i / Σ_j e^x_j

where:

- σ(x)_i is the probability of the i-th class.
- x_i is the weighted sum of inputs for the i-th neuron in the output layer.
- Σ_j is the sum over all classes.



```python
# importing the datasets
import tensorflow as tf
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from keras import layers, models
```

[2]
```python
# loading the iris datasets
iris=datasets.load_iris()
```

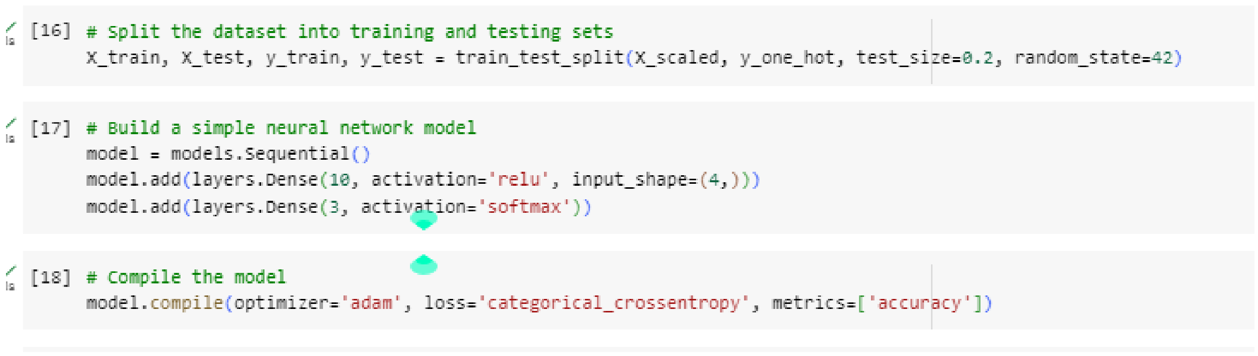
[23]
```python
X = iris.data
y = iris.target
print("Ananya Agrawal)")
```

```
Ananya Agrawal)
```

[6]
```python
# Preprocess the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

[8]
```python
# Preprocess the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

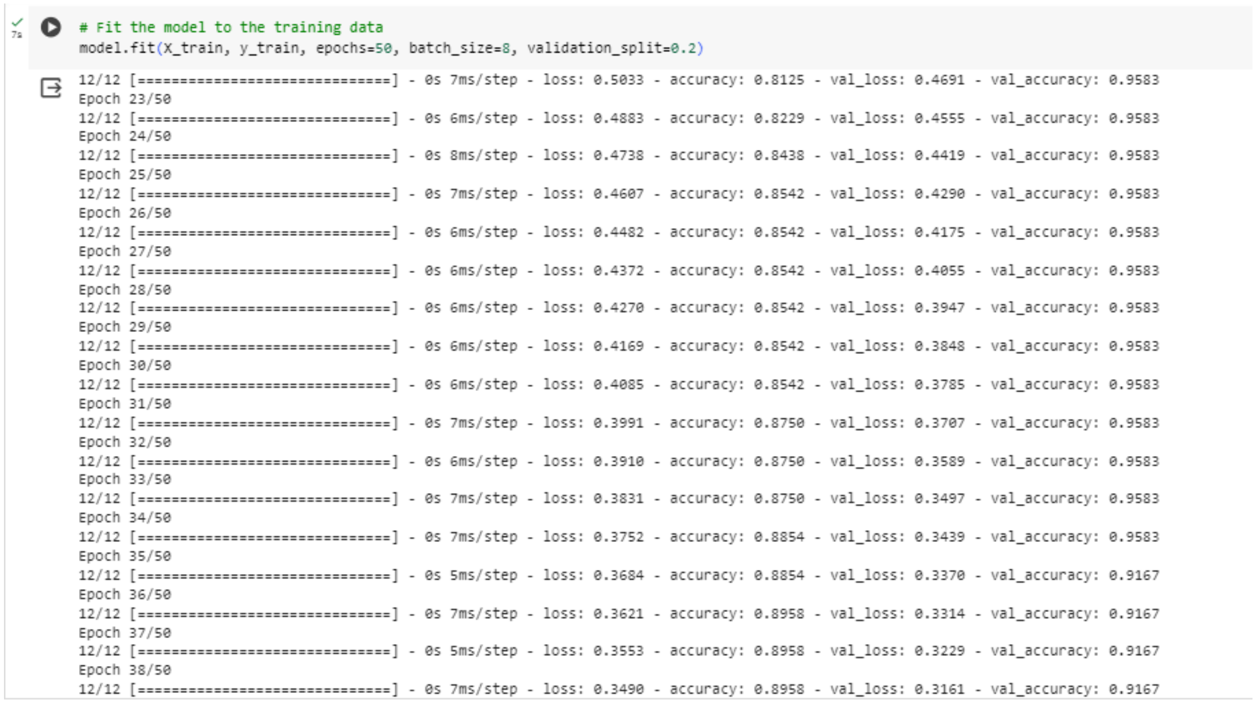
[9]
```python
# One-hot encode the target variable
encoder = LabelEncoder()
y_encoded = encoder.fit_transform(y)
y_one_hot = tf.keras.utils.to_categorical(y_encoded)
```



[16]
```python
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_one_hot, test_size=0.2, random_state=42)
```

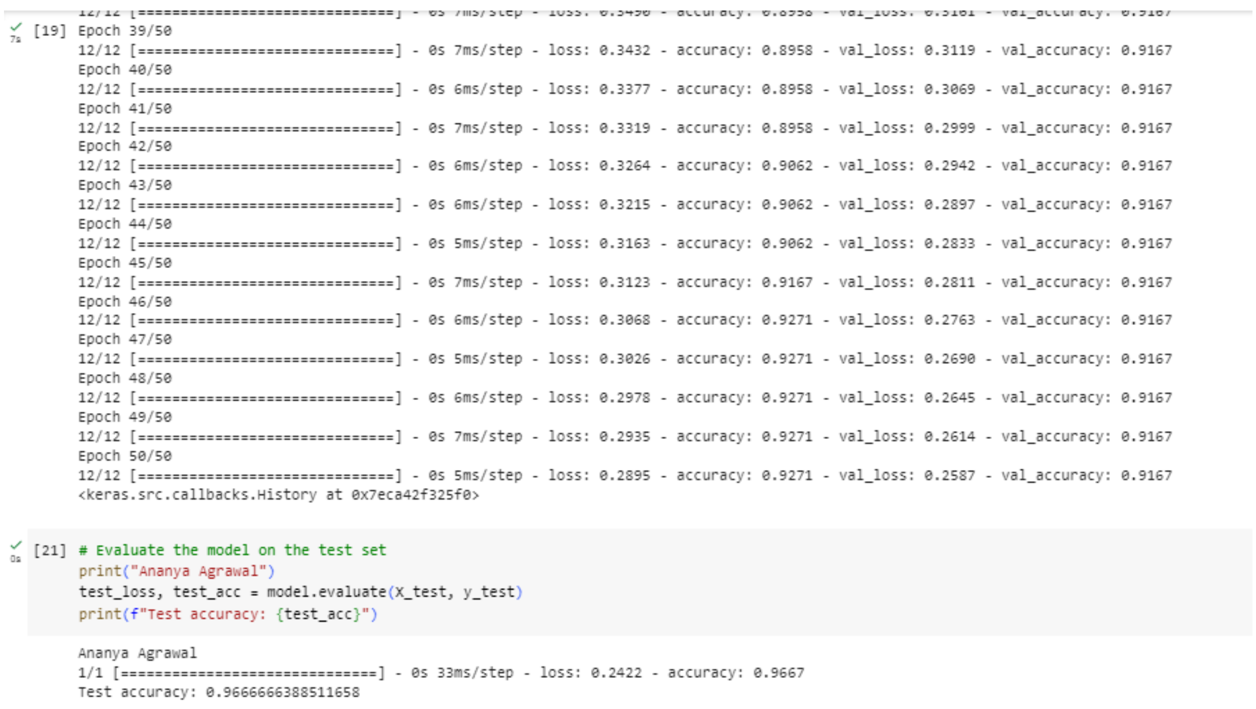
[17]
```python
# Build a simple neural network model
model = models.Sequential()
model.add(layers.Dense(10, activation='relu', input_shape=(4,)))
model.add(layers.Dense(3, activation='softmax'))
```

[18]
```python
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```



```
# Fit the model to the training data
model.fit(X_train, y_train, epochs=50, batch_size=8, validation_split=0.2)
```

```
12/12 [==============================] - 0s 7ms/step - loss: 0.5033 - accuracy: 0.8125 - val_loss: 0.4691 - val_accuracy: 0.9583
Epoch 23/50
12/12 [==============================] - 0s 6ms/step - loss: 0.4883 - accuracy: 0.8229 - val_loss: 0.4555 - val_accuracy: 0.9583
Epoch 24/50
12/12 [==============================] - 0s 8ms/step - loss: 0.4738 - accuracy: 0.8438 - val_loss: 0.4419 - val_accuracy: 0.9583
Epoch 25/50
12/12 [==============================] - 0s 7ms/step - loss: 0.4607 - accuracy: 0.8542 - val_loss: 0.4290 - val_accuracy: 0.9583
Epoch 26/50
12/12 [==============================] - 0s 6ms/step - loss: 0.4482 - accuracy: 0.8542 - val_loss: 0.4175 - val_accuracy: 0.9583
Epoch 27/50
12/12 [==============================] - 0s 6ms/step - loss: 0.4372 - accuracy: 0.8542 - val_loss: 0.4055 - val_accuracy: 0.9583
Epoch 28/50
12/12 [==============================] - 0s 6ms/step - loss: 0.4270 - accuracy: 0.8542 - val_loss: 0.3947 - val_accuracy: 0.9583
Epoch 29/50
12/12 [==============================] - 0s 6ms/step - loss: 0.4169 - accuracy: 0.8542 - val_loss: 0.3848 - val_accuracy: 0.9583
Epoch 30/50
12/12 [==============================] - 0s 6ms/step - loss: 0.4085 - accuracy: 0.8542 - val_loss: 0.3785 - val_accuracy: 0.9583
Epoch 31/50
12/12 [==============================] - 0s 7ms/step - loss: 0.3991 - accuracy: 0.8750 - val_loss: 0.3707 - val_accuracy: 0.9583
Epoch 32/50
12/12 [==============================] - 0s 6ms/step - loss: 0.3910 - accuracy: 0.8750 - val_loss: 0.3589 - val_accuracy: 0.9583
Epoch 33/50
12/12 [==============================] - 0s 7ms/step - loss: 0.3831 - accuracy: 0.8750 - val_loss: 0.3497 - val_accuracy: 0.9583
Epoch 34/50
12/12 [==============================] - 0s 7ms/step - loss: 0.3752 - accuracy: 0.8854 - val_loss: 0.3439 - val_accuracy: 0.9583
Epoch 35/50
12/12 [==============================] - 0s 5ms/step - loss: 0.3684 - accuracy: 0.8854 - val_loss: 0.3370 - val_accuracy: 0.9167
Epoch 36/50
12/12 [==============================] - 0s 7ms/step - loss: 0.3621 - accuracy: 0.8958 - val_loss: 0.3314 - val_accuracy: 0.9167
Epoch 37/50
12/12 [==============================] - 0s 5ms/step - loss: 0.3553 - accuracy: 0.8958 - val_loss: 0.3229 - val_accuracy: 0.9167
Epoch 38/50
12/12 [==============================] - 0s 7ms/step - loss: 0.3490 - accuracy: 0.8958 - val_loss: 0.3161 - val_accuracy: 0.9167
```



```
12/12 [==============================] - 0s 7ms/step - loss: 0.3490 - accuracy: 0.8958 - val_loss: 0.3161 - val_accuracy: 0.9167
```

```
[19] Epoch 39/50
12/12 [==============================] - 0s 7ms/step - loss: 0.3432 - accuracy: 0.8958 - val_loss: 0.3119 - val_accuracy: 0.9167
Epoch 40/50
12/12 [==============================] - 0s 6ms/step - loss: 0.3377 - accuracy: 0.8958 - val_loss: 0.3069 - val_accuracy: 0.9167
Epoch 41/50
12/12 [==============================] - 0s 7ms/step - loss: 0.3319 - accuracy: 0.8958 - val_loss: 0.2999 - val_accuracy: 0.9167
Epoch 42/50
12/12 [==============================] - 0s 6ms/step - loss: 0.3264 - accuracy: 0.9062 - val_loss: 0.2942 - val_accuracy: 0.9167
Epoch 43/50
12/12 [==============================] - 0s 6ms/step - loss: 0.3215 - accuracy: 0.9062 - val_loss: 0.2897 - val_accuracy: 0.9167
Epoch 44/50
12/12 [==============================] - 0s 5ms/step - loss: 0.3163 - accuracy: 0.9062 - val_loss: 0.2833 - val_accuracy: 0.9167
Epoch 45/50
12/12 [==============================] - 0s 7ms/step - loss: 0.3123 - accuracy: 0.9167 - val_loss: 0.2811 - val_accuracy: 0.9167
Epoch 46/50
12/12 [==============================] - 0s 6ms/step - loss: 0.3068 - accuracy: 0.9271 - val_loss: 0.2763 - val_accuracy: 0.9167
Epoch 47/50
12/12 [==============================] - 0s 5ms/step - loss: 0.3026 - accuracy: 0.9271 - val_loss: 0.2690 - val_accuracy: 0.9167
Epoch 48/50
12/12 [==============================] - 0s 6ms/step - loss: 0.2978 - accuracy: 0.9271 - val_loss: 0.2645 - val_accuracy: 0.9167
Epoch 49/50
12/12 [==============================] - 0s 7ms/step - loss: 0.2935 - accuracy: 0.9271 - val_loss: 0.2614 - val_accuracy: 0.9167
Epoch 50/50
12/12 [==============================] - 0s 5ms/step - loss: 0.2895 - accuracy: 0.9271 - val_loss: 0.2587 - val_accuracy: 0.9167
<keras.src.callbacks.History at 0x7eca42f325f0>
```

```
[21] # Evaluate the model on the test set
print("Ananya Agrawal")
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"Test accuracy: {test_acc}")
```

```
Ananya Agrawal
1/1 [==============================] - 0s 33ms/step - loss: 0.2422 - accuracy: 0.9667
Test accuracy: 0.9666666388511658
```

Experiment 2: Implementation of Artificial Neural Network using Input layer, hidden layers, and output layer. (using sigmoid for binary classification)

A single-layer perceptron is the simplest type of artificial neural network (ANN) with only one hidden layer containing one neuron. It is a powerful tool for binary classification problems, where the output can only be one of two values (e.g., 0 or 1, True or False).

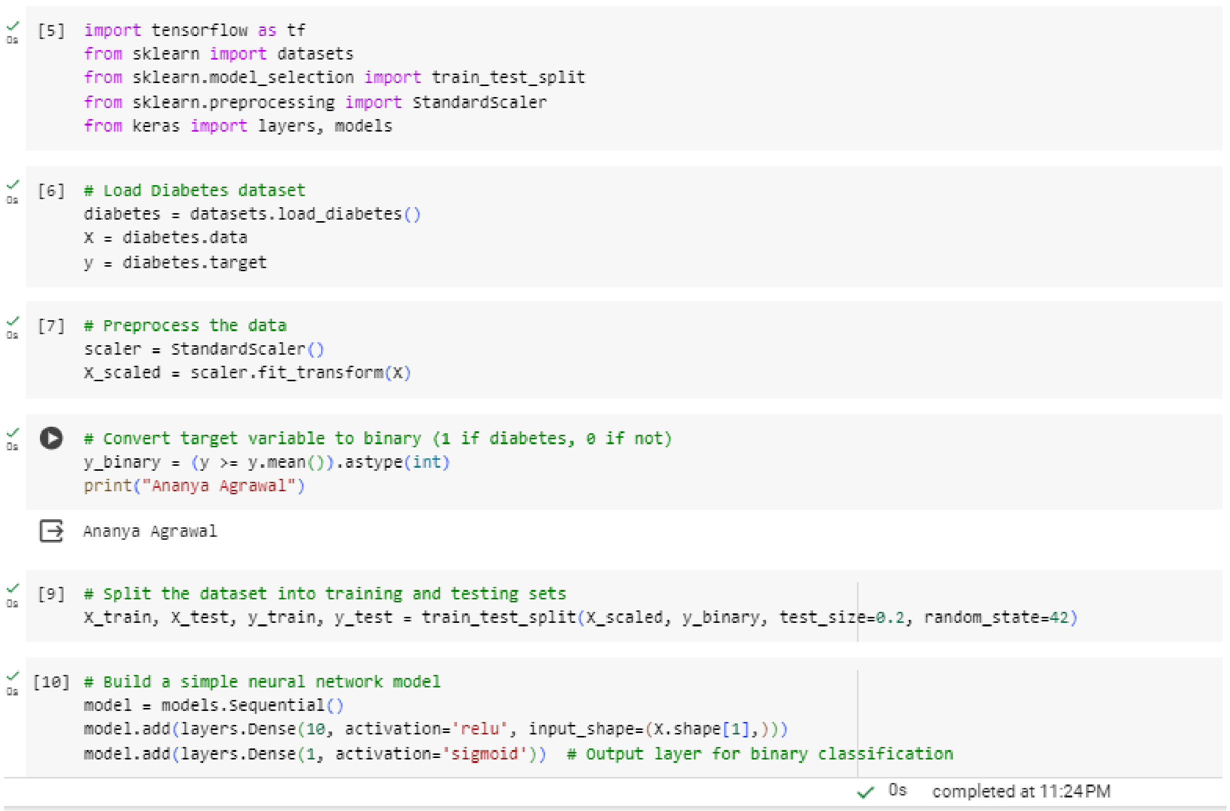
Here's how it works:

1. **Input:** Receives the data points represented by vectors of features.
2. **Weighted sum:** The single neuron in the hidden layer takes a weighted sum of all input features. Each feature value is multiplied by its corresponding weight, and the results are summed up.
3. **Bias:** A bias term is added to the weighted sum to shift the activation function and adjust the decision boundary.
4. **Sigmoid activation function:** The sum is then passed through the sigmoid activation function. This function maps the sum to a value between 0 and 1, representing the probability of the input belonging to one of the two classes.
5. **Output:** The output of the network is the value obtained after applying the sigmoid function. It represents the predicted class for the input data point.

Sigmoid Activation Function:

The sigmoid function plays a crucial role in binary classification by converting the weighted sum into a probability. It has the following properties:

- **Output range:** 0 to 1.
- **S-shaped curve:** The function gradually increases from 0 to 1 as the input value increases.
- **Differentiable:** Allows for efficient gradient-based optimization algorithms for learning the network weights.



```
[5]  import tensorflow as tf
     from sklearn import datasets
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import StandardScaler
     from keras import layers, models
```

```
[6]  # Load Diabetes dataset
     diabetes = datasets.load_diabetes()
     X = diabetes.data
     y = diabetes.target
```

```
[7]  # Preprocess the data
     scaler = StandardScaler()
     X_scaled = scaler.fit_transform(X)
```

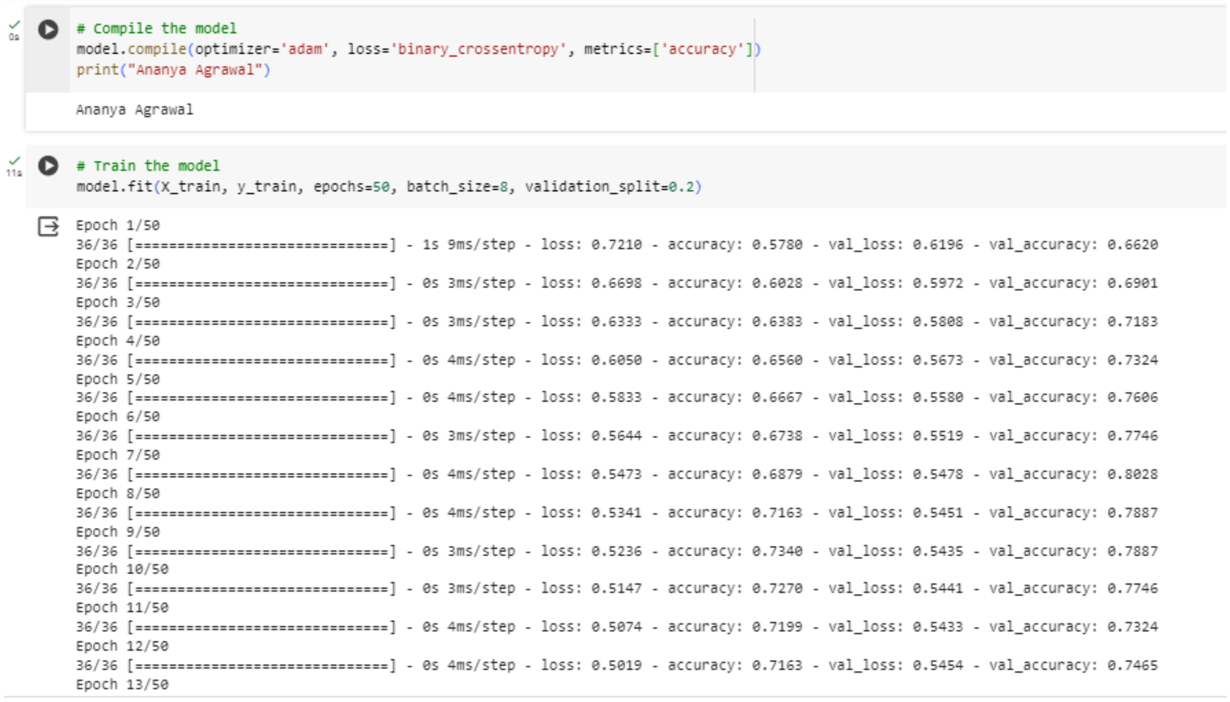
```
     # Convert target variable to binary (1 if diabetes, 0 if not)
     y_binary = (y >= y.mean()).astype(int)
     print("Ananya Agrawal")
```

```
Ananya Agrawal
```

```
[9]  # Split the dataset into training and testing sets
     X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_binary, test_size=0.2, random_state=42)
```

```
[10] # Build a simple neural network model
     model = models.Sequential()
     model.add(layers.Dense(10, activation='relu', input_shape=(X.shape[1],)))
     model.add(layers.Dense(1, activation='sigmoid'))  # Output layer for binary classification
```

✓ 0s completed at 11:24 PM

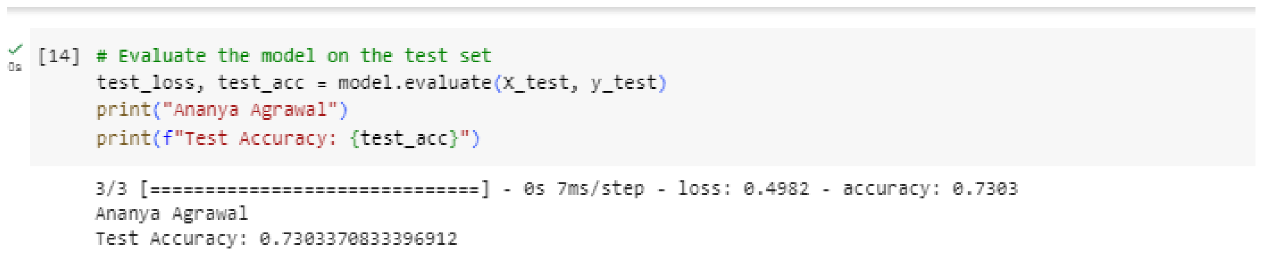


```
# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
print("Ananya Agrawal")
```

```
Ananya Agrawal
```

```
# Train the model
model.fit(X_train, y_train, epochs=50, batch_size=8, validation_split=0.2)
```

```
Epoch 1/50
36/36 [==============================] - 1s 9ms/step - loss: 0.7210 - accuracy: 0.5780 - val_loss: 0.6196 - val_accuracy: 0.6620
Epoch 2/50
36/36 [==============================] - 0s 3ms/step - loss: 0.6698 - accuracy: 0.6028 - val_loss: 0.5972 - val_accuracy: 0.6901
Epoch 3/50
36/36 [==============================] - 0s 3ms/step - loss: 0.6333 - accuracy: 0.6383 - val_loss: 0.5808 - val_accuracy: 0.7183
Epoch 4/50
36/36 [==============================] - 0s 4ms/step - loss: 0.6050 - accuracy: 0.6560 - val_loss: 0.5673 - val_accuracy: 0.7324
Epoch 5/50
36/36 [==============================] - 0s 4ms/step - loss: 0.5833 - accuracy: 0.6667 - val_loss: 0.5580 - val_accuracy: 0.7606
Epoch 6/50
36/36 [==============================] - 0s 3ms/step - loss: 0.5644 - accuracy: 0.6738 - val_loss: 0.5519 - val_accuracy: 0.7746
Epoch 7/50
36/36 [==============================] - 0s 4ms/step - loss: 0.5473 - accuracy: 0.6879 - val_loss: 0.5478 - val_accuracy: 0.8028
Epoch 8/50
36/36 [==============================] - 0s 4ms/step - loss: 0.5341 - accuracy: 0.7163 - val_loss: 0.5451 - val_accuracy: 0.7887
Epoch 9/50
36/36 [==============================] - 0s 3ms/step - loss: 0.5236 - accuracy: 0.7340 - val_loss: 0.5435 - val_accuracy: 0.7887
Epoch 10/50
36/36 [==============================] - 0s 3ms/step - loss: 0.5147 - accuracy: 0.7270 - val_loss: 0.5441 - val_accuracy: 0.7746
Epoch 11/50
36/36 [==============================] - 0s 4ms/step - loss: 0.5074 - accuracy: 0.7199 - val_loss: 0.5433 - val_accuracy: 0.7324
Epoch 12/50
36/36 [==============================] - 0s 4ms/step - loss: 0.5019 - accuracy: 0.7163 - val_loss: 0.5454 - val_accuracy: 0.7465
Epoch 13/50
```



```
[14] # Evaluate the model on the test set
     test_loss, test_acc = model.evaluate(X_test, y_test)
     print("Ananya Agrawal")
     print(f"Test Accuracy: {test_acc}")
```

```
3/3 [==============================] - 0s 7ms/step - loss: 0.4982 - accuracy: 0.7303
Ananya Agrawal
Test Accuracy: 0.7303370833396912
```

Experiment 3: Comparing accuracies of various optimisers (Gradient Descent, Stochastic Gradient Descent, Gradient Descent with Momentum, Adagrad, RMSProp, Adam)

The goal is to compare the performance of different optimization algorithms when training a neural network model. Here's the theory related to this experiment:

Optimization Algorithms in Neural Networks:

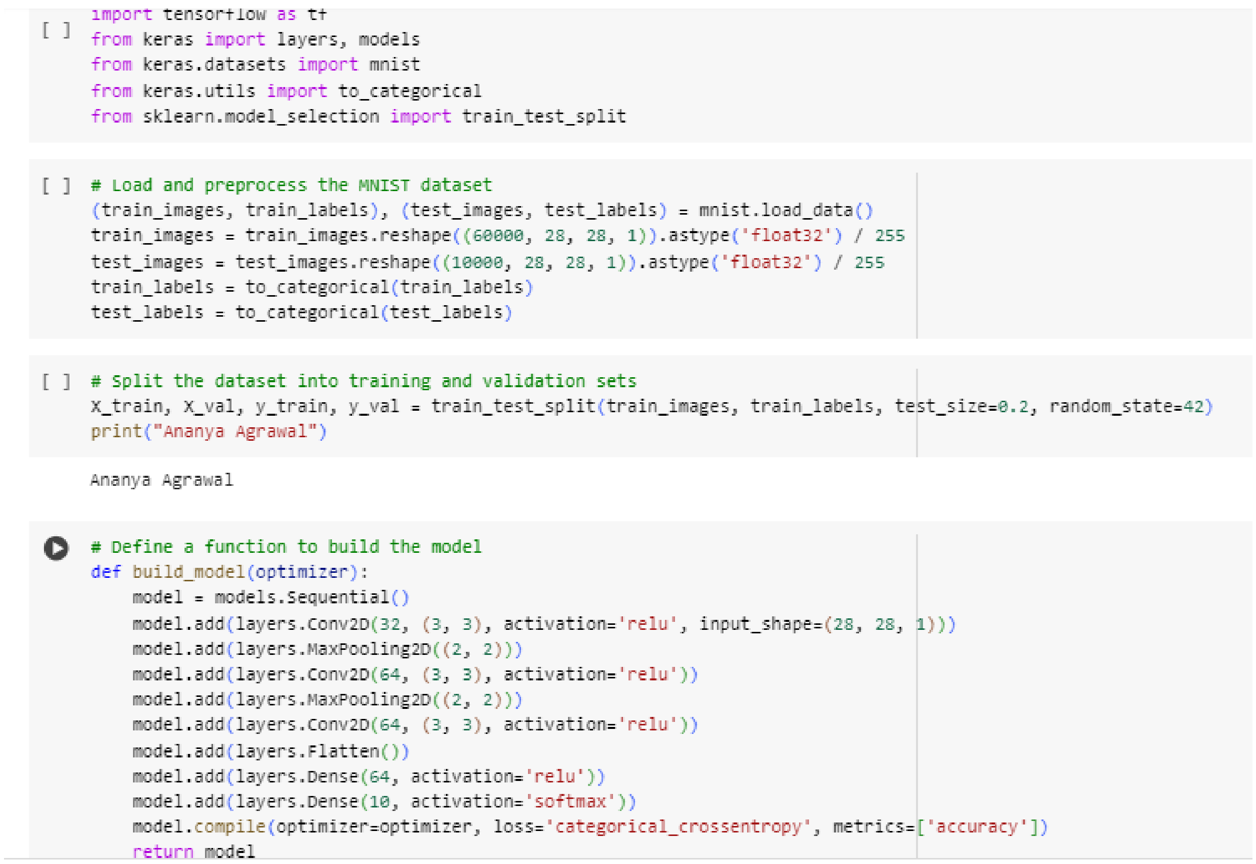
1. **Gradient Descent (GD):**
    - It's a fundamental optimization algorithm used to minimize the loss function by adjusting model parameters in the direction of steepest descent of the gradient.
    - Vanilla GD computes gradients using the entire dataset, which can be slow for large datasets.
2. **Stochastic Gradient Descent (SGD):**
    - SGD is a variation of GD that updates model parameters based on gradients computed for individual training examples or small batches.
    - It converges faster than GD but can exhibit noisy updates.
3. **Gradient Descent with Momentum:**
    - It improves upon SGD by adding momentum—a method to accelerate convergence and smooth out the update process.
    - Momentum accumulates a fraction of past gradients to continue moving in the same direction, reducing oscillations.
4. **Adagrad:**
    - Adagrad adjusts the learning rate adaptively for each parameter based on the historical squared gradients.
    - It performs larger updates for infrequent parameters and smaller updates for frequent ones.
5. **RMSProp:**
    - RMSProp addresses Adagrad's overly aggressive learning rate decay by using a moving average of squared gradients.

- It divides the learning rate by a running average of recent magnitudes of gradients, which helps to stabilize the learning process.

6. **Adam (Adaptive Moment Estimation):**

    - Adam combines the concepts of momentum and RMSProp by maintaining both a momentum term and a running average of gradients' magnitudes.

    - It adapts the learning rates for each parameter individually and often performs well across various types of neural network architectures.

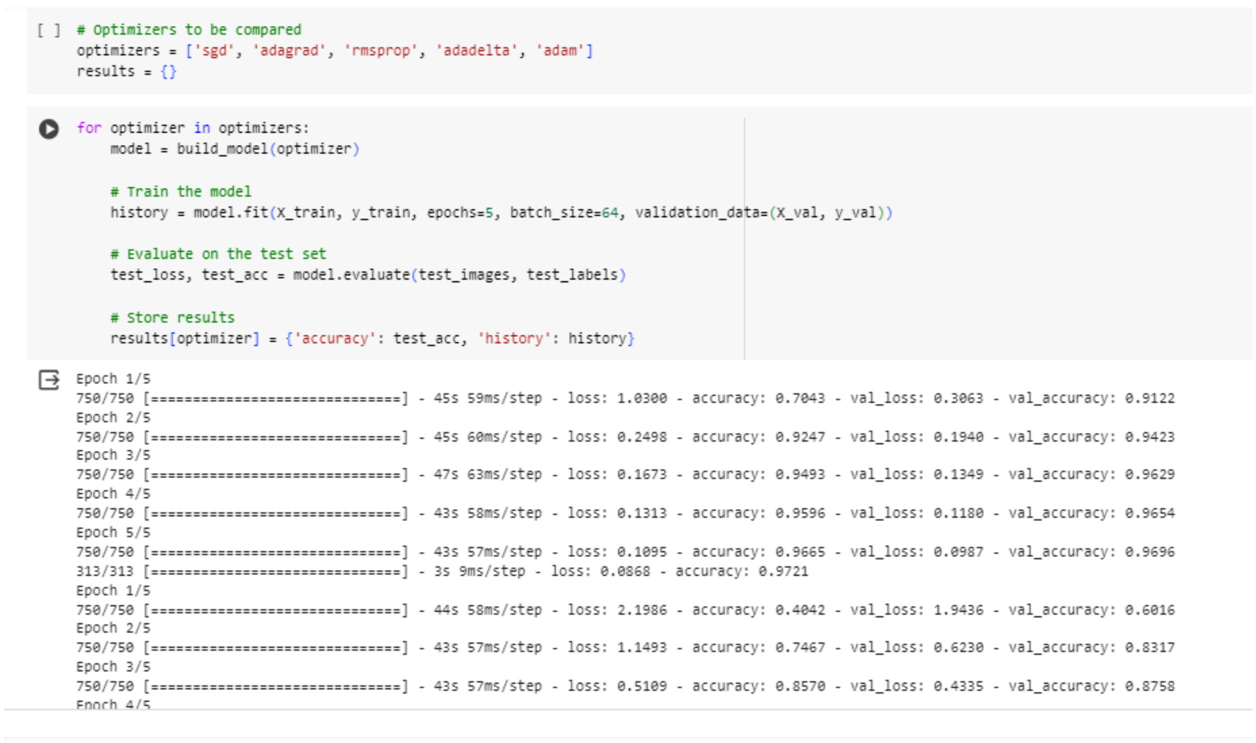


```
[ ] import tensorflow as tf
    from keras import layers, models
    from keras.datasets import mnist
    from keras.utils import to_categorical
    from sklearn.model_selection import train_test_split
```

```
[ ] # Load and preprocess the MNIST dataset
    (train_images, train_labels), (test_images, test_labels) = mnist.load_data()
    train_images = train_images.reshape((60000, 28, 28, 1)).astype('float32') / 255
    test_images = test_images.reshape((10000, 28, 28, 1)).astype('float32') / 255
    train_labels = to_categorical(train_labels)
    test_labels = to_categorical(test_labels)
```

```
[ ] # Split the dataset into training and validation sets
    X_train, X_val, y_train, y_val = train_test_split(train_images, train_labels, test_size=0.2, random_state=42)
    print("Ananya Agrawal")

    Ananya Agrawal
```

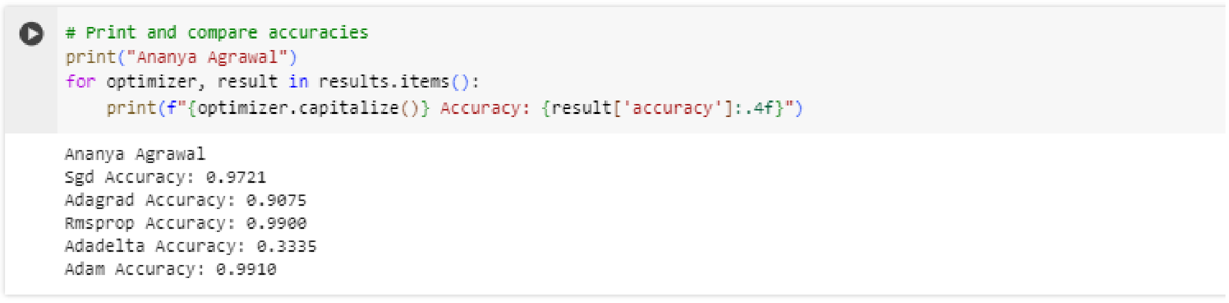
```
    # Define a function to build the model
    def build_model(optimizer):
        model = models.Sequential()
        model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
        model.add(layers.MaxPooling2D((2, 2)))
        model.add(layers.Conv2D(64, (3, 3), activation='relu'))
        model.add(layers.MaxPooling2D((2, 2)))
        model.add(layers.Conv2D(64, (3, 3), activation='relu'))
        model.add(layers.Flatten())
        model.add(layers.Dense(64, activation='relu'))
        model.add(layers.Dense(10, activation='softmax'))
        model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
        return model
```



```
[ ] # Optimizers to be compared
    optimizers = ['sgd', 'adagrad', 'rmsprop', 'adadelta', 'adam']
    results = {}
```

```
for optimizer in optimizers:
    model = build_model(optimizer)

    # Train the model
    history = model.fit(X_train, y_train, epochs=5, batch_size=64, validation_data=(X_val, y_val))

    # Evaluate on the test set
    test_loss, test_acc = model.evaluate(test_images, test_labels)

    # Store results
    results[optimizer] = {'accuracy': test_acc, 'history': history}
```

```
Epoch 1/5
750/750 [==============================] - 45s 59ms/step - loss: 1.0300 - accuracy: 0.7043 - val_loss: 0.3063 - val_accuracy: 0.9122
Epoch 2/5
750/750 [==============================] - 45s 60ms/step - loss: 0.2498 - accuracy: 0.9247 - val_loss: 0.1940 - val_accuracy: 0.9423
Epoch 3/5
750/750 [==============================] - 47s 63ms/step - loss: 0.1673 - accuracy: 0.9493 - val_loss: 0.1349 - val_accuracy: 0.9629
Epoch 4/5
750/750 [==============================] - 43s 58ms/step - loss: 0.1313 - accuracy: 0.9596 - val_loss: 0.1180 - val_accuracy: 0.9654
Epoch 5/5
750/750 [==============================] - 43s 57ms/step - loss: 0.1095 - accuracy: 0.9665 - val_loss: 0.0987 - val_accuracy: 0.9696
313/313 [==============================] - 3s 9ms/step - loss: 0.0868 - accuracy: 0.9721
Epoch 1/5
750/750 [==============================] - 44s 58ms/step - loss: 2.1986 - accuracy: 0.4042 - val_loss: 1.9436 - val_accuracy: 0.6016
Epoch 2/5
750/750 [==============================] - 43s 57ms/step - loss: 1.1493 - accuracy: 0.7467 - val_loss: 0.6230 - val_accuracy: 0.8317
Epoch 3/5
750/750 [==============================] - 43s 57ms/step - loss: 0.5109 - accuracy: 0.8570 - val_loss: 0.4335 - val_accuracy: 0.8758
Epoch 4/5
```



```
# Print and compare accuracies
print("Ananya Agrawal")
for optimizer, result in results.items():
    print(f"{optimizer.capitalize()} Accuracy: {result['accuracy']:.4f}")
```

```
Ananya Agrawal
Sgd Accuracy: 0.9721
Adagrad Accuracy: 0.9075
Rmsprop Accuracy: 0.9900
Adadelta Accuracy: 0.3335
Adam Accuracy: 0.9910
```

Experiment 4: Implementation of Dropout Layers

Dropout Layers in Neural Networks:

1. **Purpose of Dropout:**
   - Dropout is a regularization technique used during training in neural networks to prevent overfitting.
   - It works by randomly deactivating (dropping out) a fraction of neurons during each training iteration.
2. **Implementation of Dropout:**
   - Dropout is applied by inserting Dropout layers into the neural network architecture.
   - During training, each neuron in the Dropout layer has a probability (usually between 0.2 and 0.5) of being "dropped out" or set to zero.
3. **Functionality:**
   - Dropout introduces randomness during training, effectively making the network less sensitive to the specific weights of neurons.
   - It forces the network to learn more robust and generalized features, reducing reliance on a specific set of neurons.
4. **Preventing Overfitting:**
   - By dropping out neurons, Dropout prevents the network from relying too heavily on certain neurons or co-adapting to them.
   - It helps in creating an ensemble of different network architectures within a single model, reducing overfitting.
5. **Training vs. Testing:**
   - During inference or testing, the entire network is used (no neurons are dropped out), and instead, the weights are scaled to compensate for the dropped neurons during training.

In 1
```
import tensorflow as tf
from keras import layers, models
from keras.datasets import mnist
from keras.utils import to_categorical
```
Executed at 2023.12.10 20:02:19 in 5s 873ms

In 2
```
# Load MNIST dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```
Executed at 2023.12.10 20:02:20 in 190ms

In 3
```
# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```
Executed at 2023.12.10 20:02:20 in 170ms

In 4
```
# Add a channel dimension to the images (MNIST is grayscale)
train_images = train_images.reshape((60000, 28, 28, 1))
test_images = test_images.reshape((10000, 28, 28, 1))
```
Executed at 2023.12.10 20:02:20 in 16ms

In 5
```
# One-hot encode the labels
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```
Executed at 2023.12.10 20:02:20 in 15ms

In 6
```
# Define the CNN model with dropout layers
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.25))  # Dropout layer
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.25))  # Dropout layer
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))  # Dropout layer
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```
Executed at 2023.12.10 20:02:20 in 139ms

In 7
```
# Compile the model
model.compile(optimizer='adam',
             loss='categorical_crossentropy',
             metrics=['accuracy'])
```
Executed at 2023.12.10 20:02:20 in 17ms

In 8
```
# Train the model
model.fit(train_images, train_labels, epochs=10, batch_size=64, validation_data=(test_images, test_labels))
```
Executed at 2023.12.10 20:04:19 in 1m 59s 157ms

```
Epoch 1/10
938/938 [==============================] - 12s 12ms/step - loss: 0.3096 - accuracy: 0.8993 - val_loss: 0.0603 - val_accuracy: 0.9816
Epoch 2/10
938/938 [==============================] - 12s 13ms/step - loss: 0.0958 - accuracy: 0.9709 - val_loss: 0.0407 - val_accuracy: 0.9869
Epoch 3/10
938/938 [==============================] - 13s 13ms/step - loss: 0.0754 - accuracy: 0.9766 - val_loss: 0.0288 - val_accuracy: 0.9906
Epoch 4/10
938/938 [==============================] - 11s 12ms/step - loss: 0.0638 - accuracy: 0.9803 - val_loss: 0.0318 - val_accuracy: 0.9900
Epoch 5/10
938/938 [==============================] - 12s 12ms/step - loss: 0.0594 - accuracy: 0.9815 - val_loss: 0.0276 - val_accuracy: 0.9904
Epoch 6/10
938/938 [==============================] - 12s 12ms/step - loss: 0.0522 - accuracy: 0.9842 - val_loss: 0.0241 - val_accuracy: 0.9919
Epoch 7/10
938/938 [==============================] - 12s 13ms/step - loss: 0.0485 - accuracy: 0.9848 - val_loss: 0.0255 - val_accuracy: 0.9920
```

Out 8
```
<keras.callbacks.History at 0x1bab29273a0>
```

In 9
```
# Evaluate the model
print("Anik Roy")
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy: {test_acc}')
```
Executed at 2023.12.10 20:04:20 in 894ms

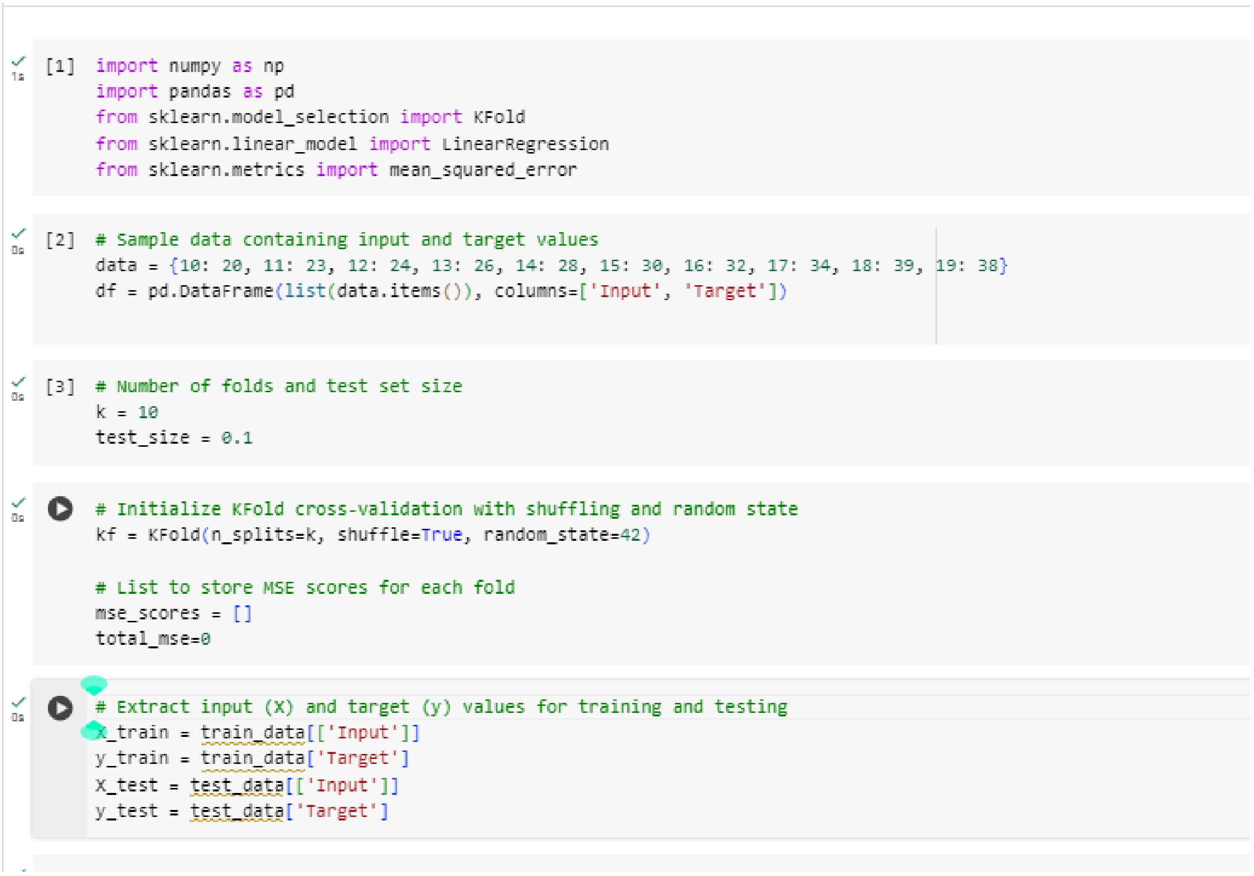
```
313/313 [==============================] - 1s 3ms/step - loss: 0.0231 - accuracy: 0.9921
Test accuracy: 0.9921000003814697
```

Experiment 5: Implementation of K-Fold cross validation

K-Fold Cross-Validation is a technique used to evaluate the performance and robustness of a machine learning model.

Process:

1. **Data Splitting:**
    - The dataset is divided into 'K' subsets/folds of approximately equal size.
    - One fold is used for validation, and the remaining (K-1) folds are used for training the model.
2. **Training and Validation:**
    - The model is trained 'K' times, each time using a different fold as the validation set and the rest for training.
    - For each iteration, the model's performance metrics are evaluated using the validation fold.
3. **Performance Metrics:**
    - Metrics such as accuracy, precision, recall, or F1 score are computed for each iteration.
    - The average performance across all 'K' folds is often used to represent the model's general performance.



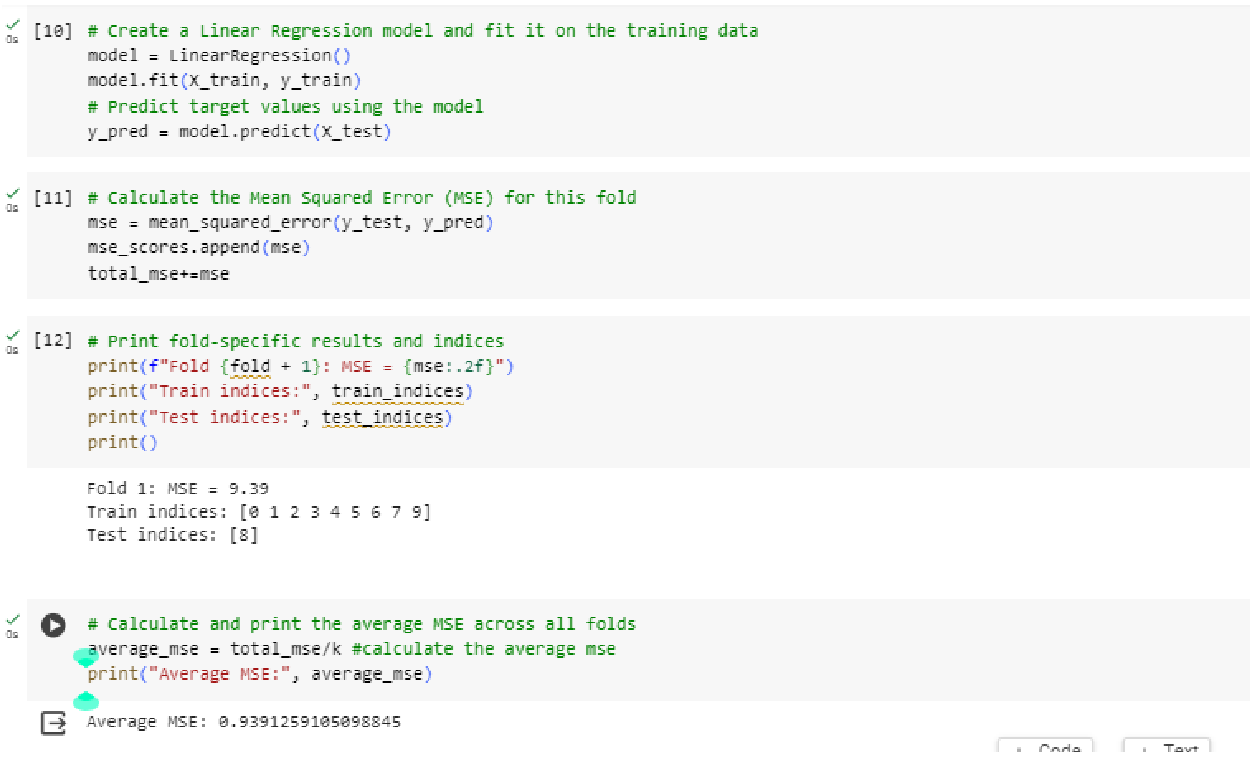
[1]
```python
import numpy as np
import pandas as pd
from sklearn.model_selection import KFold
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
```

[2]
```python
# Sample data containing input and target values
data = {10: 20, 11: 23, 12: 24, 13: 26, 14: 28, 15: 30, 16: 32, 17: 34, 18: 39, 19: 38}
df = pd.DataFrame(list(data.items()), columns=['Input', 'Target'])
```

[3]
```python
# Number of folds and test set size
k = 10
test_size = 0.1
```

```python
# Initialize KFold cross-validation with shuffling and random state
kf = KFold(n_splits=k, shuffle=True, random_state=42)

# List to store MSE scores for each fold
mse_scores = []
total_mse=0
```

```python
# Extract input (X) and target (y) values for training and testing
X_train = train_data[['Input']]
y_train = train_data['Target']
X_test = test_data[['Input']]
y_test = test_data['Target']
```



[10]
```python
# Create a Linear Regression model and fit it on the training data
model = LinearRegression()
model.fit(X_train, y_train)
# Predict target values using the model
y_pred = model.predict(X_test)
```

[11]
```python
# Calculate the Mean Squared Error (MSE) for this fold
mse = mean_squared_error(y_test, y_pred)
mse_scores.append(mse)
total_mse+=mse
```

[12]
```python
# Print fold-specific results and indices
print(f"Fold {fold + 1}: MSE = {mse:.2f}")
print("Train indices:", train_indices)
print("Test indices:", test_indices)
print()
```

```
Fold 1: MSE = 9.39
Train indices: [0 1 2 3 4 5 6 7 9]
Test indices: [8]
```

```python
# Calculate and print the average MSE across all folds
average_mse = total_mse/k #calculate the average mse
print("Average MSE:", average_mse)
```

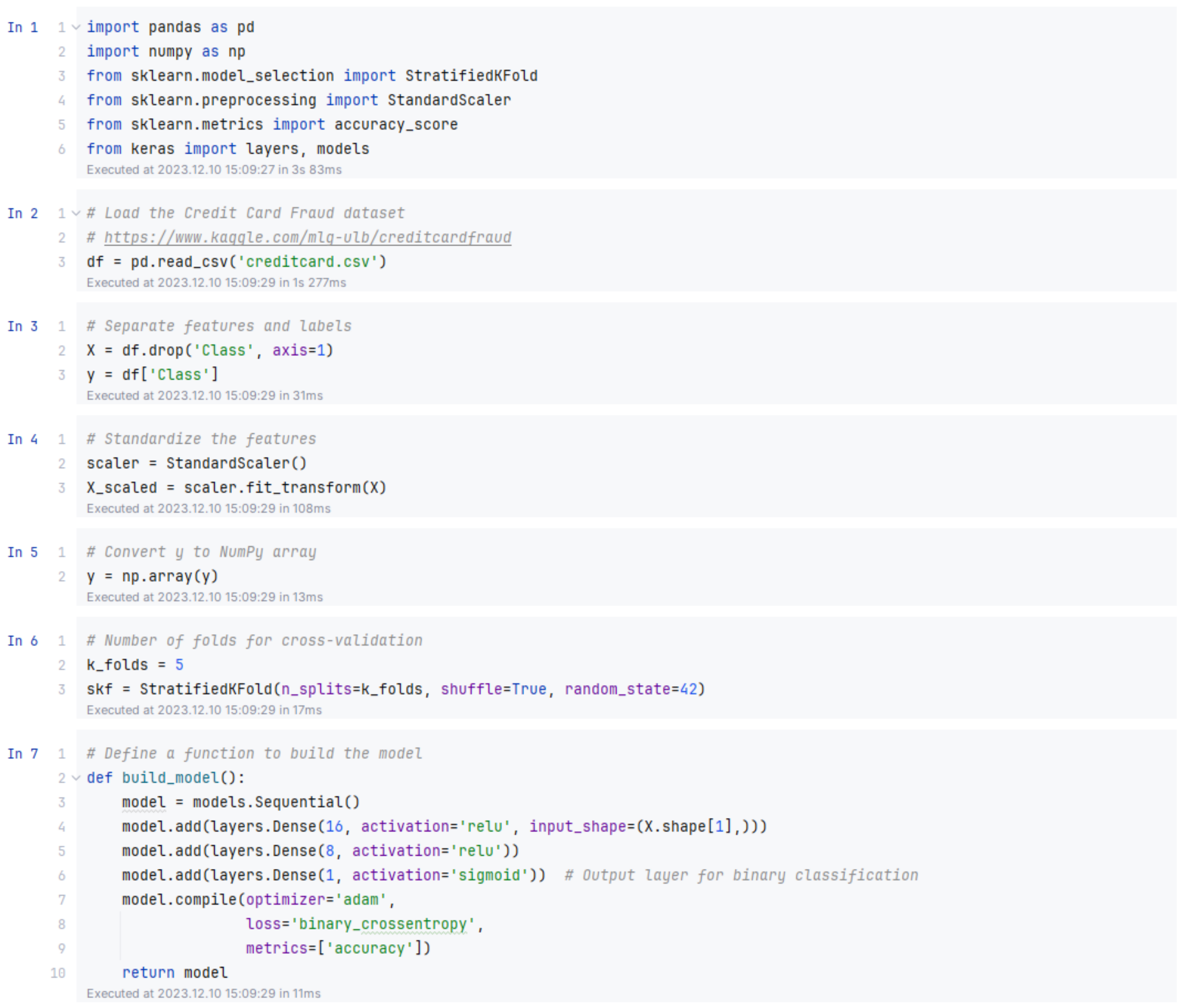
```
Average MSE: 0.9391259105098845
```

Experiment 6: Implementation of Stratified K-Fold cross validation

Stratified K-Fold Cross-Validation is a technique used to validate the robustness and performance of a machine learning model, especially in cases where the dataset is imbalanced.

Process:

1. **Data Splitting:**
    - Similar to K-Fold Cross-Validation, the dataset is divided into 'K' subsets/folds.
    - However, in Stratified K-Fold, the division is done in such a way that each fold maintains the same class distribution as the original dataset. This is crucial in cases of class imbalance.
2. **Training and Validation:**
    - The model is trained 'K' times, where each fold is used once as the validation set and the rest for training.
    - The training and validation procedure remains the same as in regular K-Fold CV.
3. **Performance Metrics:**
    - Performance metrics like accuracy, precision, recall, or F1 score are calculated for each iteration across the 'K' folds.



In 1
```python
import pandas as pd
import numpy as np
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from keras import layers, models
```
Executed at 2023.12.10 15:09:27 in 3s 83ms

In 2
```python
# Load the Credit Card Fraud dataset
# https://www.kaggle.com/mlg-ulb/creditcardfraud
df = pd.read_csv('creditcard.csv')
```
Executed at 2023.12.10 15:09:29 in 1s 277ms

In 3
```python
# Separate features and labels
X = df.drop('Class', axis=1)
y = df['Class']
```
Executed at 2023.12.10 15:09:29 in 31ms

In 4
```python
# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```
Executed at 2023.12.10 15:09:29 in 108ms

In 5
```python
# Convert y to NumPy array
y = np.array(y)
```
Executed at 2023.12.10 15:09:29 in 13ms

In 6
```python
# Number of folds for cross-validation
k_folds = 5
skf = StratifiedKFold(n_splits=k_folds, shuffle=True, random_state=42)
```
Executed at 2023.12.10 15:09:29 in 17ms

In 7
```python
# Define a function to build the model
def build_model():
    model = models.Sequential()
    model.add(layers.Dense(16, activation='relu', input_shape=(X.shape[1],)))
    model.add(layers.Dense(8, activation='relu'))
    model.add(layers.Dense(1, activation='sigmoid'))  # Output layer for binary classification
    model.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model
```
Executed at 2023.12.10 15:09:29 in 11ms

```
In 8
# Perform stratified k-fold cross-validation
fold_results = []

for train_index, test_index in skf.split(X_scaled, y):
    X_train, X_test = X_scaled[train_index], X_scaled[test_index]
    y_train, y_test = y[train_index], y[test_index]

    model = build_model()

    # Train the model
    model.fit(X_train, y_train, epochs=5, batch_size=64, verbose=0)

    # Evaluate on the test set
    y_pred = model.predict(X_test)
    y_pred_binary = np.round(y_pred).flatten()

    # Calculate accuracy for the fold
    fold_accuracy = accuracy_score(y_test, y_pred_binary)
    fold_results.append(fold_accuracy)
```
Executed at 2023.12.10 15:11:14 in 1m 44s 755ms

```
1781/1781 [==============================] - 2s 826us/step
1781/1781 [==============================] - 2s 896us/step
1781/1781 [==============================] - 2s 852us/step
1781/1781 [==============================] - 2s 842us/step
1781/1781 [==============================] - 2s 874us/step
```

```
average_accuracy = sum(fold_results) / k_folds
print(f'Average Cross-Validation Accuracy: {average_accuracy:.4f}')
```
Executed at 2023.12.10 15:11:14 in 13ms

```
Average Cross-Validation Accuracy: 0.9994
```

Experiment 7: Implementation of Recurrent Neural Network for time series analysis

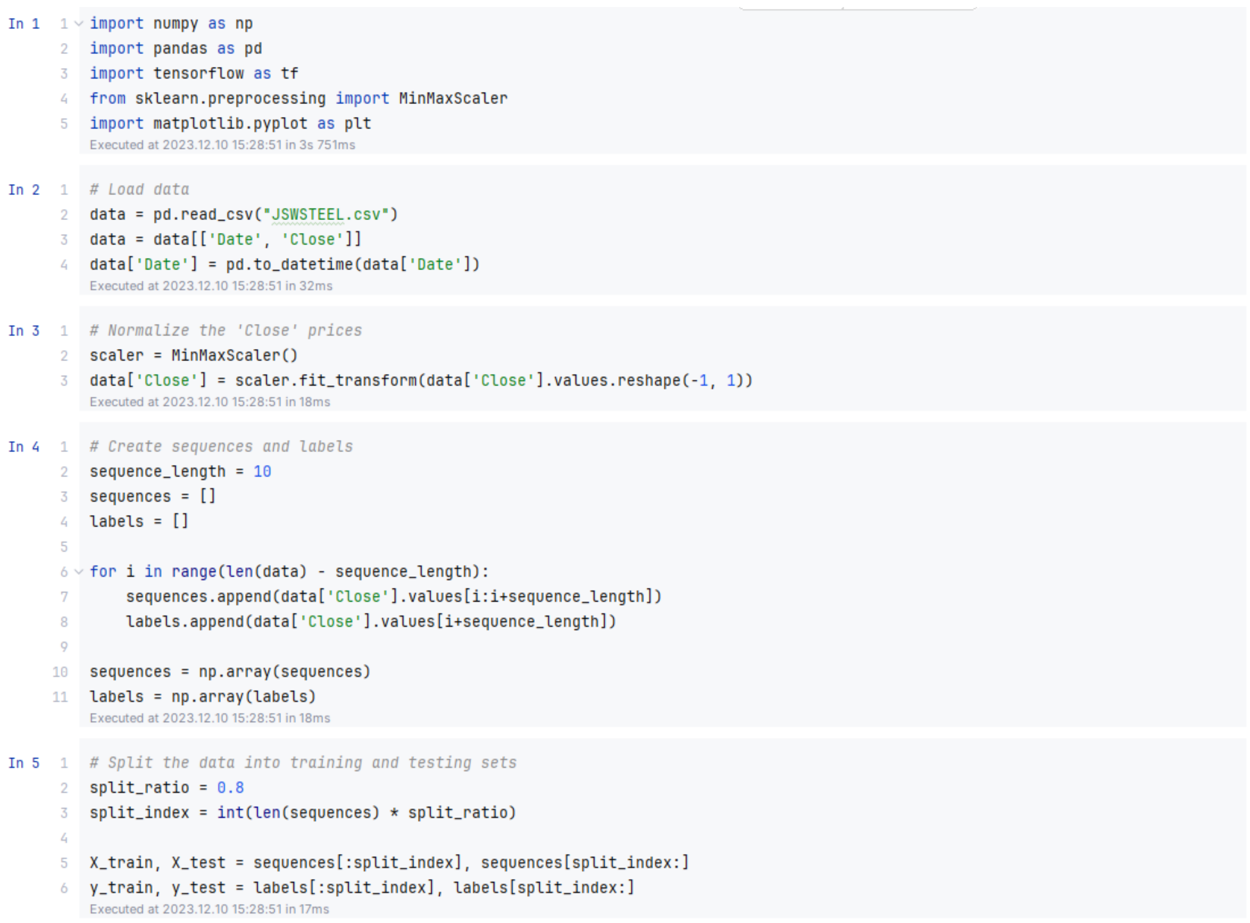
A Recurrent Neural Network (RNN) is a type of neural network designed to work with sequential data, making it particularly effective for time series analysis, natural language processing, and sequential data prediction tasks.

Key Concepts:

1. **Sequential Data Handling:**
    - RNNs are built to handle sequential data where the order of the data points matters. They can process input sequences of varying lengths.
2. **Memory Retention:**
    - RNNs have a memory element that retains information about past inputs using hidden states. This memory enables them to capture patterns or dependencies within sequential data.
3. **Recurrent Connections:**
    - These networks employ recurrent connections that allow information to persist. Each neuron in an RNN is connected to itself, enabling it to consider previous inputs while processing the current one.
4. **Time Series Analysis:**
    - For time series tasks, RNNs can predict future values based on historical data. They excel in capturing temporal dependencies and patterns like seasonality and trends.

Objective:

The objective of implementing an RNN for time series analysis is to showcase the network's ability to learn from sequential data, capture temporal patterns, and make predictions about future values based on the historical input. This experiment aims to demonstrate the RNN's effectiveness in handling time series data by learning and generalizing patterns for forecasting or classification tasks.



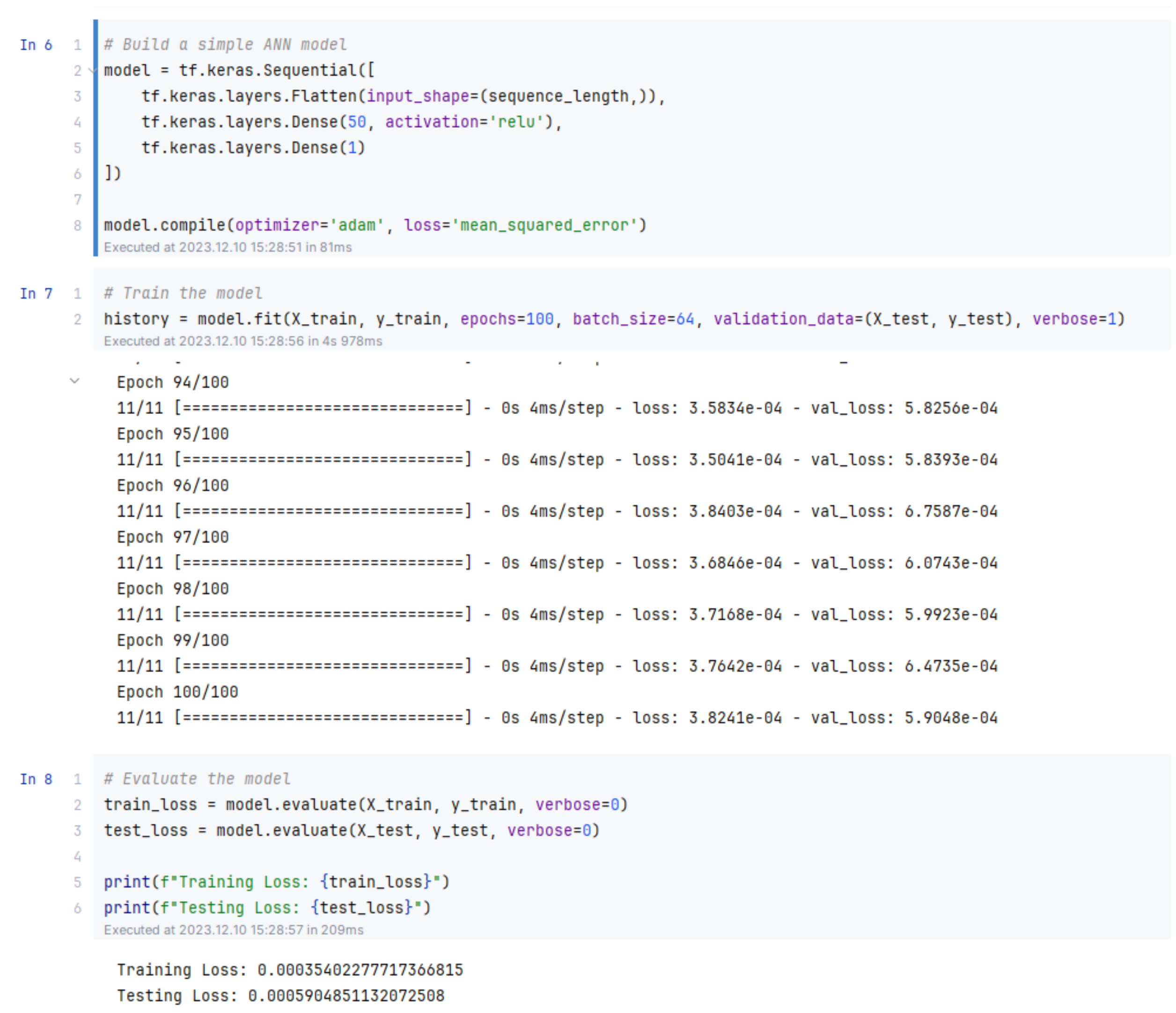
In 1
```python
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt
```
Executed at 2023.12.10 15:28:51 in 3s 751ms

In 2
```python
# Load data
data = pd.read_csv("JSWSTEEL.csv")
data = data[['Date', 'Close']]
data['Date'] = pd.to_datetime(data['Date'])
```
Executed at 2023.12.10 15:28:51 in 32ms

In 3
```python
# Normalize the 'Close' prices
scaler = MinMaxScaler()
data['Close'] = scaler.fit_transform(data['Close'].values.reshape(-1, 1))
```
Executed at 2023.12.10 15:28:51 in 18ms

In 4
```python
# Create sequences and labels
sequence_length = 10
sequences = []
labels = []

for i in range(len(data) - sequence_length):
    sequences.append(data['Close'].values[i:i+sequence_length])
    labels.append(data['Close'].values[i+sequence_length])

sequences = np.array(sequences)
labels = np.array(labels)
```
Executed at 2023.12.10 15:28:51 in 18ms

In 5
```python
# Split the data into training and testing sets
split_ratio = 0.8
split_index = int(len(sequences) * split_ratio)

X_train, X_test = sequences[:split_index], sequences[split_index:]
y_train, y_test = labels[:split_index], labels[split_index:]
```
Executed at 2023.12.10 15:28:51 in 17ms



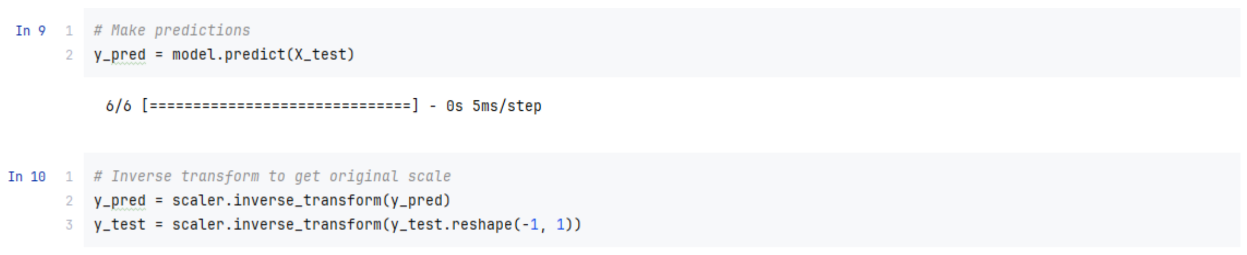
```
In 6
# Build a simple ANN model
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(sequence_length,)),
    tf.keras.layers.Dense(50, activation='relu'),
    tf.keras.layers.Dense(1)
])

model.compile(optimizer='adam', loss='mean_squared_error')
Executed at 2023.12.10 15:28:51 in 81ms
```

```
In 7
# Train the model
history = model.fit(X_train, y_train, epochs=100, batch_size=64, validation_data=(X_test, y_test), verbose=1)
Executed at 2023.12.10 15:28:56 in 4s 978ms
```

```
Epoch 94/100
11/11 [==============================] - 0s 4ms/step - loss: 3.5834e-04 - val_loss: 5.8256e-04
Epoch 95/100
11/11 [==============================] - 0s 4ms/step - loss: 3.5041e-04 - val_loss: 5.8393e-04
Epoch 96/100
11/11 [==============================] - 0s 4ms/step - loss: 3.8403e-04 - val_loss: 6.7587e-04
Epoch 97/100
11/11 [==============================] - 0s 4ms/step - loss: 3.6846e-04 - val_loss: 6.0743e-04
Epoch 98/100
11/11 [==============================] - 0s 4ms/step - loss: 3.7168e-04 - val_loss: 5.9923e-04
Epoch 99/100
11/11 [==============================] - 0s 4ms/step - loss: 3.7642e-04 - val_loss: 6.4735e-04
Epoch 100/100
11/11 [==============================] - 0s 4ms/step - loss: 3.8241e-04 - val_loss: 5.9048e-04
```

```
In 8
# Evaluate the model
train_loss = model.evaluate(X_train, y_train, verbose=0)
test_loss = model.evaluate(X_test, y_test, verbose=0)

print(f"Training Loss: {train_loss}")
print(f"Testing Loss: {test_loss}")
Executed at 2023.12.10 15:28:57 in 209ms
```

```
Training Loss: 0.00035402277717366815
Testing Loss: 0.0005904851132072508
```

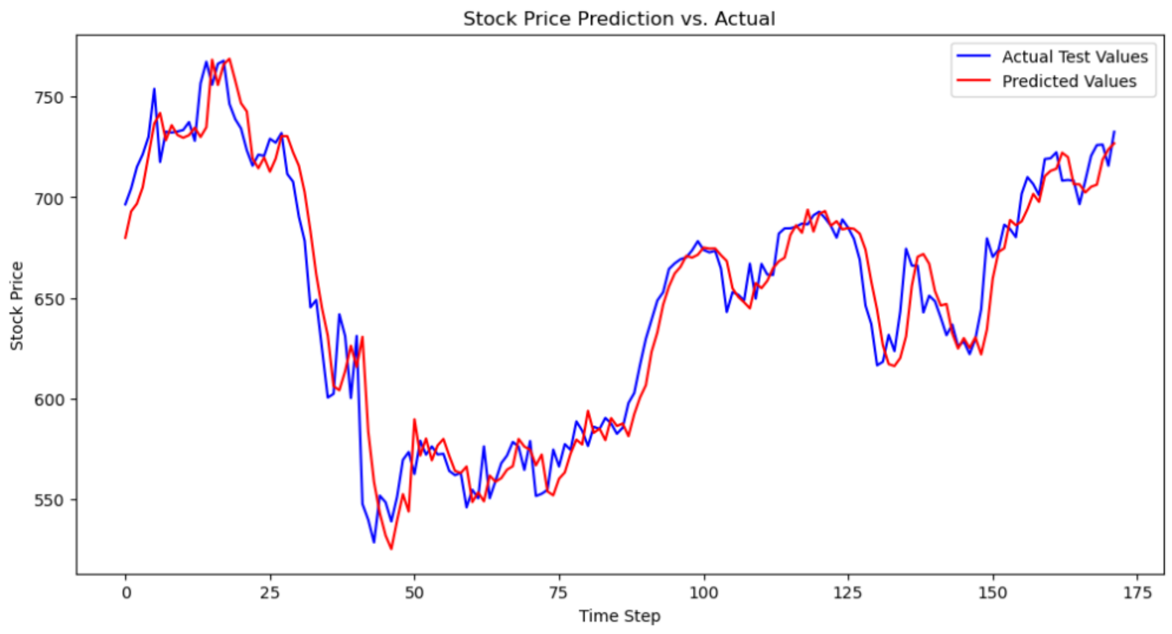


```
In 9
# Make predictions
y_pred = model.predict(X_test)
```

```
6/6 [==============================] - 0s 5ms/step
```

```
In 10
# Inverse transform to get original scale
y_pred = scaler.inverse_transform(y_pred)
y_test = scaler.inverse_transform(y_test.reshape(-1, 1))
```

```
plt.figure(figsize=(12, 6))
plt.plot(y_test, label='Actual Test Values', color='blue')
plt.plot(y_pred, label='Predicted Values', color='red')
plt.title('Stock Price Prediction vs. Actual')
plt.xlabel('Time Step')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```

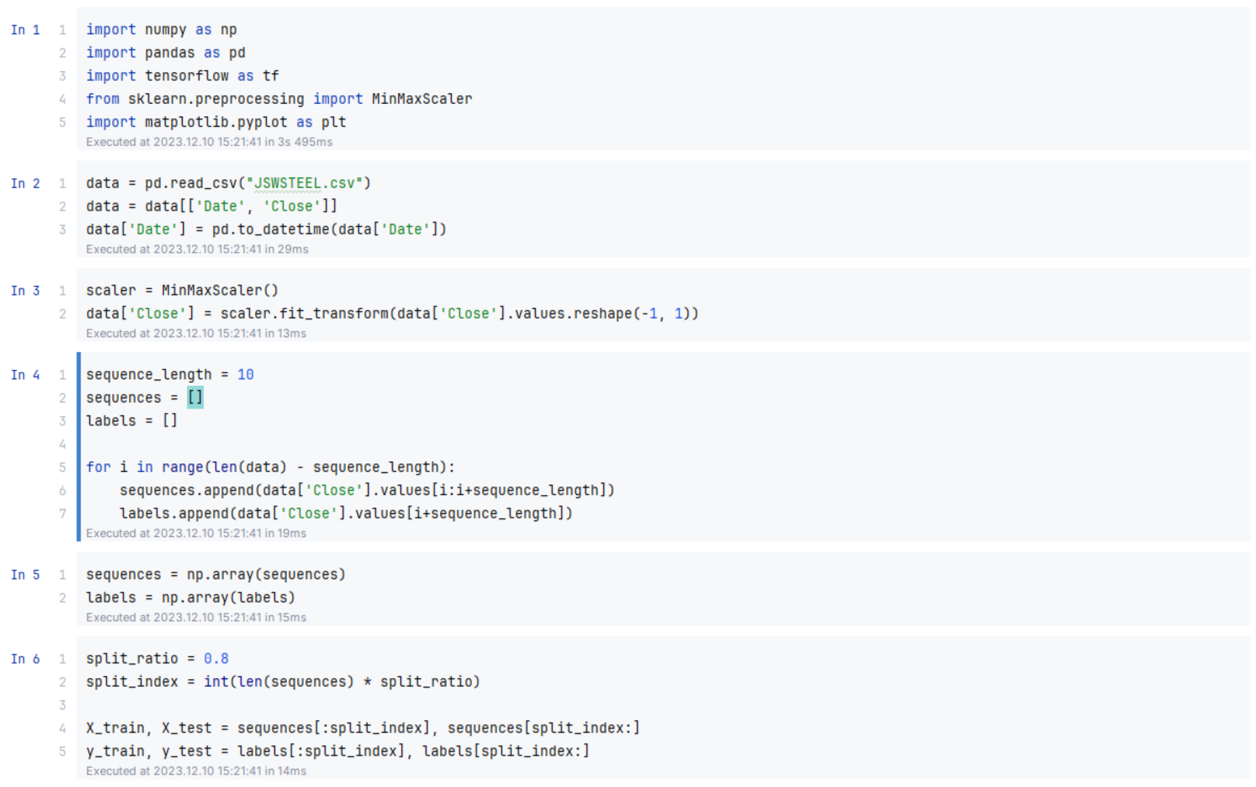


Experiment 8: Implementation of Long Short-Term Memory (LSTM) model.

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) architecture specifically designed to address the vanishing gradient problem in traditional RNNs and capture long-range dependencies in sequential data.

Key Aspects of LSTM:

1. **Memory Cells:**
    - LSTMs utilize memory cells that enable them to retain information over long sequences. These cells have an internal structure that helps in remembering or forgetting information as required.
2. **Gates:**
    - LSTMs have three gates: the input gate, forget gate, and output gate.
        - Input Gate: Controls the flow of new information into the cell state.
        - Forget Gate: Decides what information to discard or forget from the cell state.
        - Output Gate: Determines what information to output based on the current input and the memory of the cell state.
3. **Long-Term Dependencies:**
    - LSTMs are capable of learning long-range dependencies in sequential data. They can retain important information over a long period without the loss of context, making them suitable for tasks that require understanding of long-term patterns.
4. **Training:**
    - LSTMs are trained using backpropagation through time (BPTT), similar to other recurrent networks. However, the unique architecture of LSTMs with gating mechanisms enables better learning of sequences by mitigating the vanishing gradient problem.



In 1
```python
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt
```
Executed at 2023.12.10 15:21:41 in 3s 495ms

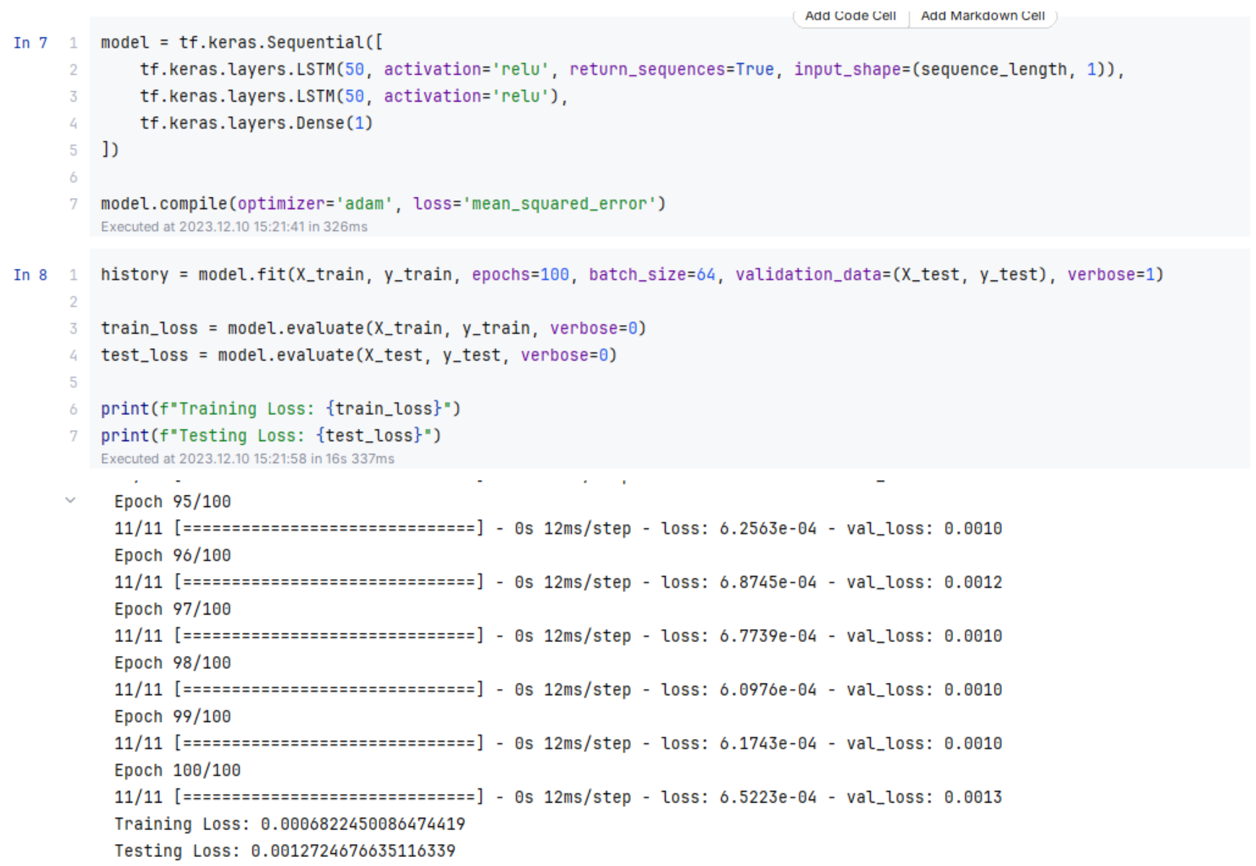
In 2
```python
data = pd.read_csv("JSWSTEEL.csv")
data = data[['Date', 'Close']]
data['Date'] = pd.to_datetime(data['Date'])
```
Executed at 2023.12.10 15:21:41 in 29ms

In 3
```python
scaler = MinMaxScaler()
data['Close'] = scaler.fit_transform(data['Close'].values.reshape(-1, 1))
```
Executed at 2023.12.10 15:21:41 in 13ms

In 4
```python
sequence_length = 10
sequences = []
labels = []

for i in range(len(data) - sequence_length):
    sequences.append(data['Close'].values[i:i+sequence_length])
    labels.append(data['Close'].values[i+sequence_length])
```
Executed at 2023.12.10 15:21:41 in 19ms

In 5
```python
sequences = np.array(sequences)
labels = np.array(labels)
```
Executed at 2023.12.10 15:21:41 in 15ms

In 6
```python
split_ratio = 0.8
split_index = int(len(sequences) * split_ratio)

X_train, X_test = sequences[:split_index], sequences[split_index:]
y_train, y_test = labels[:split_index], labels[split_index:]
```
Executed at 2023.12.10 15:21:41 in 14ms

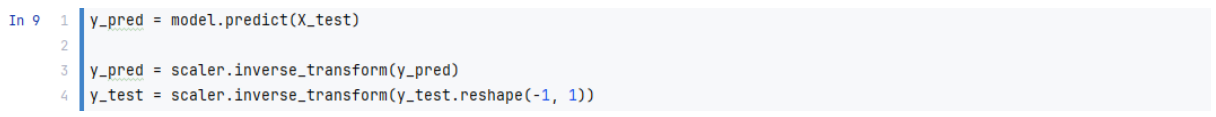


Add Code Cell | Add Markdown Cell

In 7
```python
model = tf.keras.Sequential([
    tf.keras.layers.LSTM(50, activation='relu', return_sequences=True, input_shape=(sequence_length, 1)),
    tf.keras.layers.LSTM(50, activation='relu'),
    tf.keras.layers.Dense(1)
])

model.compile(optimizer='adam', loss='mean_squared_error')
```
Executed at 2023.12.10 15:21:41 in 326ms

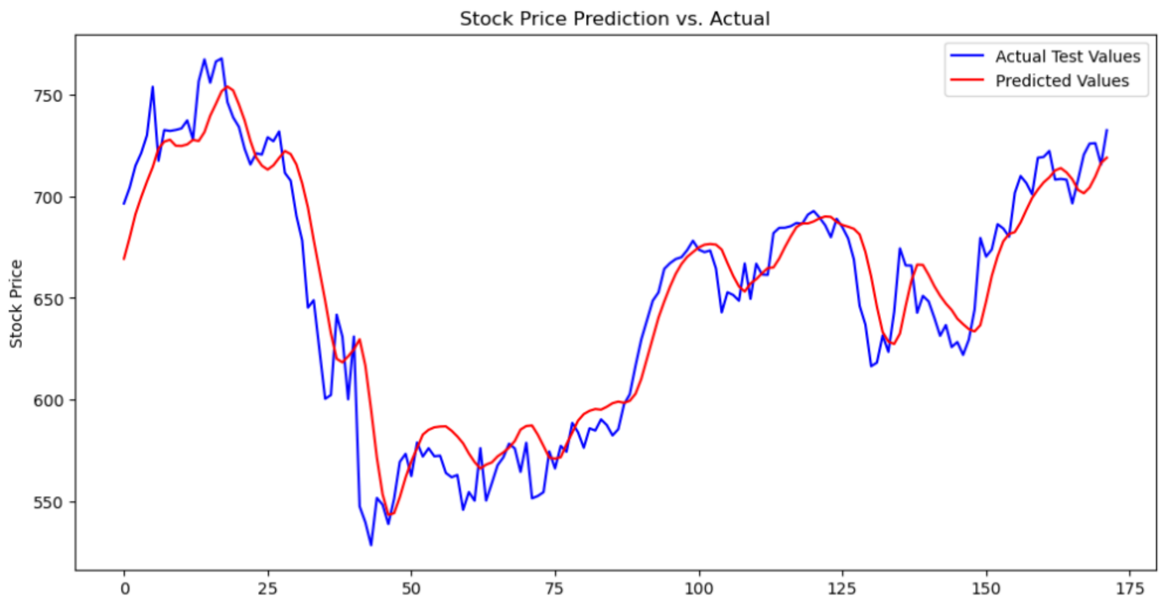
In 8
```python
history = model.fit(X_train, y_train, epochs=100, batch_size=64, validation_data=(X_test, y_test), verbose=1)

train_loss = model.evaluate(X_train, y_train, verbose=0)
test_loss = model.evaluate(X_test, y_test, verbose=0)

print(f"Training Loss: {train_loss}")
print(f"Testing Loss: {test_loss}")
```
Executed at 2023.12.10 15:21:58 in 16s 337ms

```
Epoch 95/100
11/11 [==============================] - 0s 12ms/step - loss: 6.2563e-04 - val_loss: 0.0010
Epoch 96/100
11/11 [==============================] - 0s 12ms/step - loss: 6.8745e-04 - val_loss: 0.0012
Epoch 97/100
11/11 [==============================] - 0s 12ms/step - loss: 6.7739e-04 - val_loss: 0.0010
Epoch 98/100
11/11 [==============================] - 0s 12ms/step - loss: 6.0976e-04 - val_loss: 0.0010
Epoch 99/100
11/11 [==============================] - 0s 12ms/step - loss: 6.1743e-04 - val_loss: 0.0010
Epoch 100/100
11/11 [==============================] - 0s 12ms/step - loss: 6.5223e-04 - val_loss: 0.0013
Training Loss: 0.0006822450086474419
Testing Loss: 0.0012724676635116339
```



In 9
```python
y_pred = model.predict(X_test)

y_pred = scaler.inverse_transform(y_pred)
y_test = scaler.inverse_transform(y_test.reshape(-1, 1))
```

```python
plt.figure(figsize=(12, 6))
plt.plot(y_test, label='Actual Test Values', color='blue')
plt.plot(y_pred, label='Predicted Values', color='red')
plt.title('Stock Price Prediction vs. Actual')
plt.xlabel('Time Step')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```
Executed at 2023.12.10 16:06:51 in 20ms

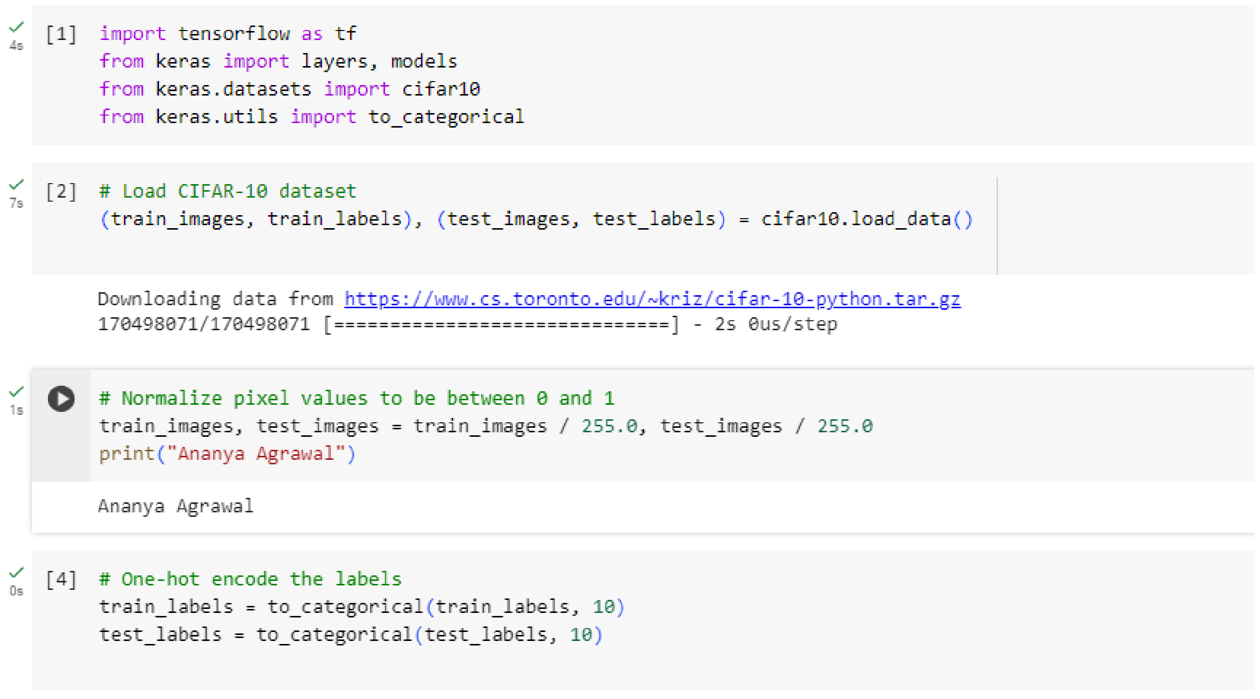


Experiment 9: Implementation of Convolutional Neural Network using CIFAR10 dataset.

A Convolutional Neural Network (CNN) is a type of deep neural network primarily used for image recognition, classification, and computer vision tasks. It's designed to automatically and adaptively learn spatial hierarchies of features from input images.

Key Aspects of CNNs:

1. **Convolutional Layers:**
    - CNNs comprise multiple convolutional layers that apply filters or kernels to input images. These filters detect various features like edges, textures, and patterns across the image.
2. **Pooling Layers:**
    - Pooling layers (e.g., MaxPooling) reduce the dimensionality of the convolved feature maps while retaining the most important information. They downsample the features, reducing computational complexity.
3. **Activation Functions:**
    - Commonly used activation functions like ReLU (Rectified Linear Unit) introduce non-linearity into the network, aiding in learning complex patterns within the data.
4. **Fully Connected Layers:**
    - After the convolutional and pooling layers, fully connected layers are employed for classification. They combine all the features learned by previous layers for final decision making.



```
[1] import tensorflow as tf
    from keras import layers, models
    from keras.datasets import cifar10
    from keras.utils import to_categorical
```

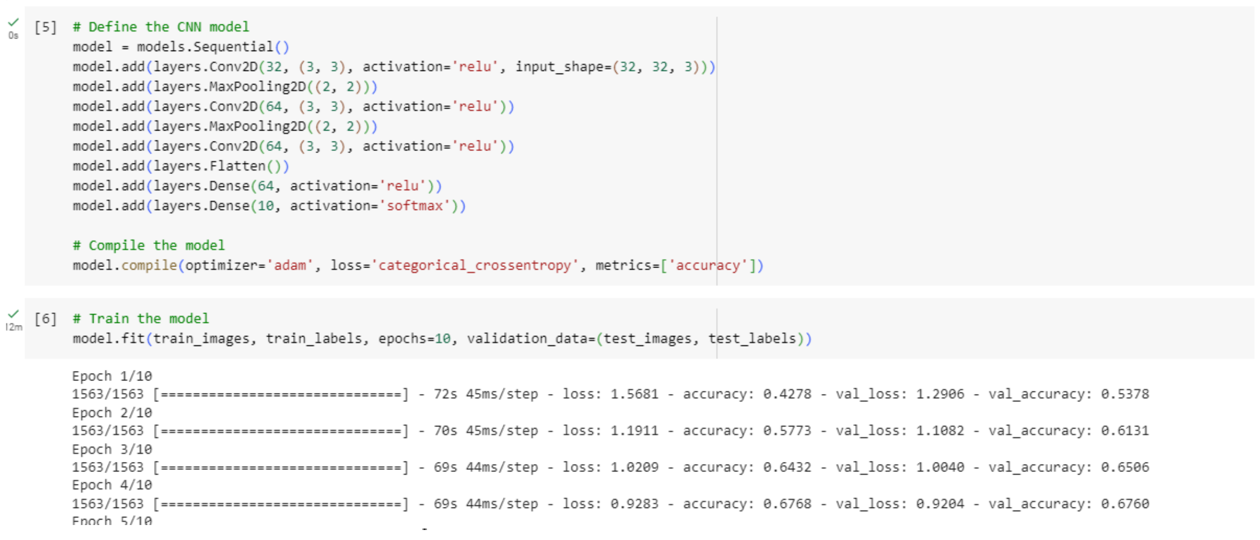
```
[2] # Load CIFAR-10 dataset
    (train_images, train_labels), (test_images, test_labels) = cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 2s 0us/step
```

```
# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
print("Ananya Agrawal")
```

```
Ananya Agrawal
```

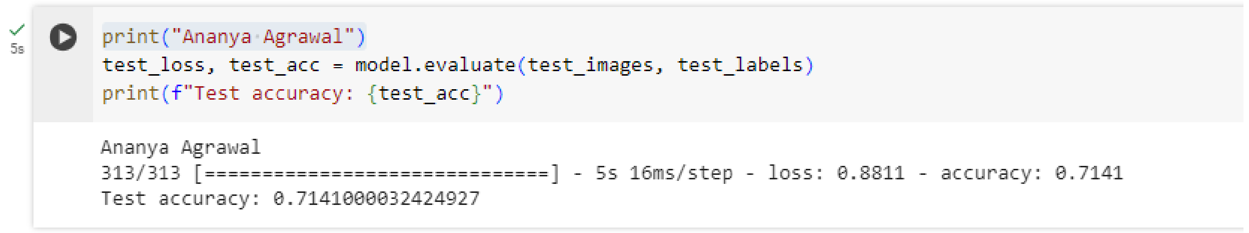
```
[4] # One-hot encode the labels
    train_labels = to_categorical(train_labels, 10)
    test_labels = to_categorical(test_labels, 10)
```



```
[5] # Define the CNN model
    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'))
    model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(10, activation='softmax'))

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

```
[6] # Train the model
    model.fit(train_images, train_labels, epochs=10, validation_data=(test_images, test_labels))
```

```
Epoch 1/10
1563/1563 [==============================] - 72s 45ms/step - loss: 1.5681 - accuracy: 0.4278 - val_loss: 1.2906 - val_accuracy: 0.5378
Epoch 2/10
1563/1563 [==============================] - 70s 45ms/step - loss: 1.1911 - accuracy: 0.5773 - val_loss: 1.1082 - val_accuracy: 0.6131
Epoch 3/10
1563/1563 [==============================] - 69s 44ms/step - loss: 1.0209 - accuracy: 0.6432 - val_loss: 1.0040 - val_accuracy: 0.6506
Epoch 4/10
1563/1563 [==============================] - 69s 44ms/step - loss: 0.9283 - accuracy: 0.6768 - val_loss: 0.9204 - val_accuracy: 0.6760
Epoch 5/10
```



```
print("Ananya Agrawal")
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"Test accuracy: {test_acc}")
```

```
Ananya Agrawal
313/313 [==============================] - 5s 16ms/step - loss: 0.8811 - accuracy: 0.7141
Test accuracy: 0.7141000032424927
```

Experiment 10: Implementation of pre-trained model for Image Processing and comparing accuracies

Pre-trained models are neural network architectures that have been trained on large datasets like ImageNet for tasks such as image classification, object detection, or segmentation.

Key Points:

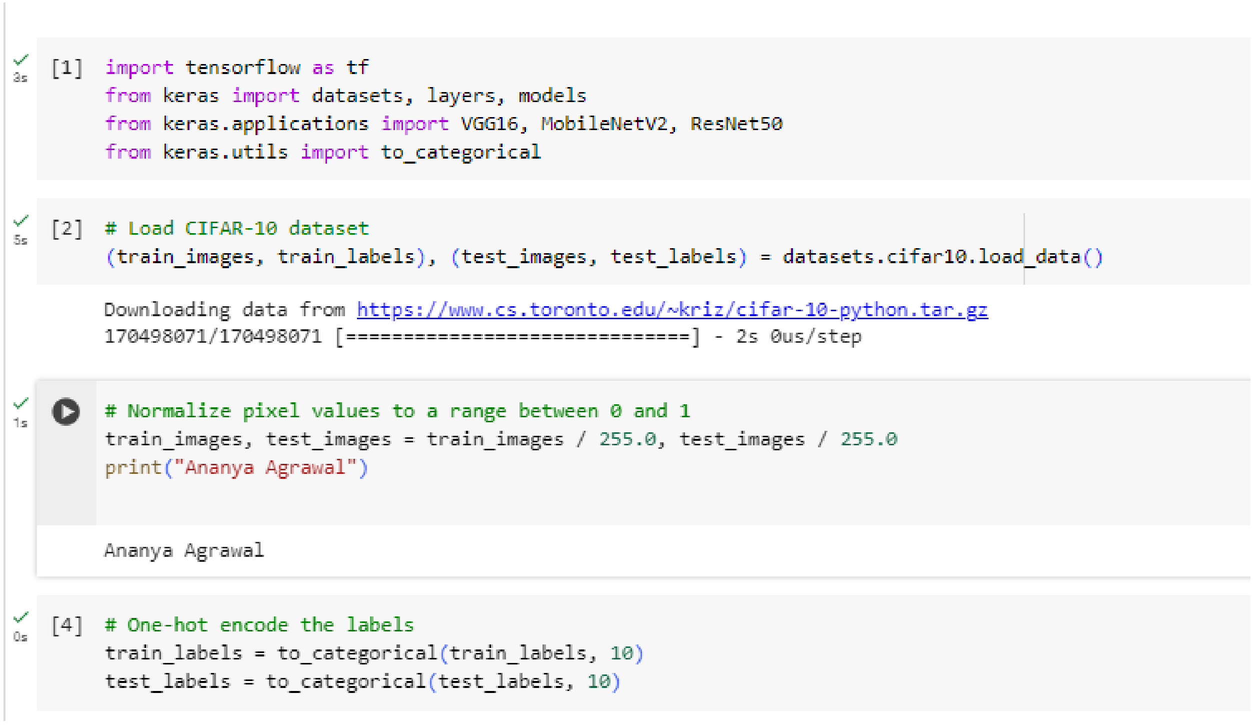
1. **Pre-trained Models:**
    - These models are trained on vast datasets, learning features and patterns that generalize well to various image-related tasks.
    - Popular pre-trained models include VGG, ResNet, Inception, and MobileNet, among others.
2. **Transfer Learning:**
    - Leveraging pre-trained models involves using their learned representations as a starting point for a new task or dataset.
    - This approach is known as transfer learning, where the pre-trained model's knowledge is fine-tuned or used as a feature extractor for a different dataset.
3. **Fine-tuning and Feature Extraction:**
    - Fine-tuning entails retraining some or all parts of the pre-trained model's layers on a new dataset to adapt it to the specific task.
    - Feature extraction involves using the pre-trained model's layers as a fixed feature extractor and adding new layers on top for the desired task.
4. **Comparing Accuracies:**
    - In this experiment, the accuracies of different pre-trained models on a specific image-related task (like classification or object detection) are compared.
    - The models' performances are evaluated on the same test dataset, and their accuracies or other evaluation metrics are analyzed comparatively.

5. **Applications:**

    - Pre-trained models offer a shortcut for building accurate models with less data and computational resources. They find extensive use in various applications like image classification, object detection, style transfer, and more.

Experiment Objective:

The objective is to showcase the effectiveness of pre-trained models in image-related tasks by comparing their accuracies. By using these models as a starting point, the experiment aims to demonstrate how pre-trained architectures, with or without fine-tuning, perform on a specific task or dataset. This comparison helps in identifying which pre-trained model architecture performs better for the given image processing task.

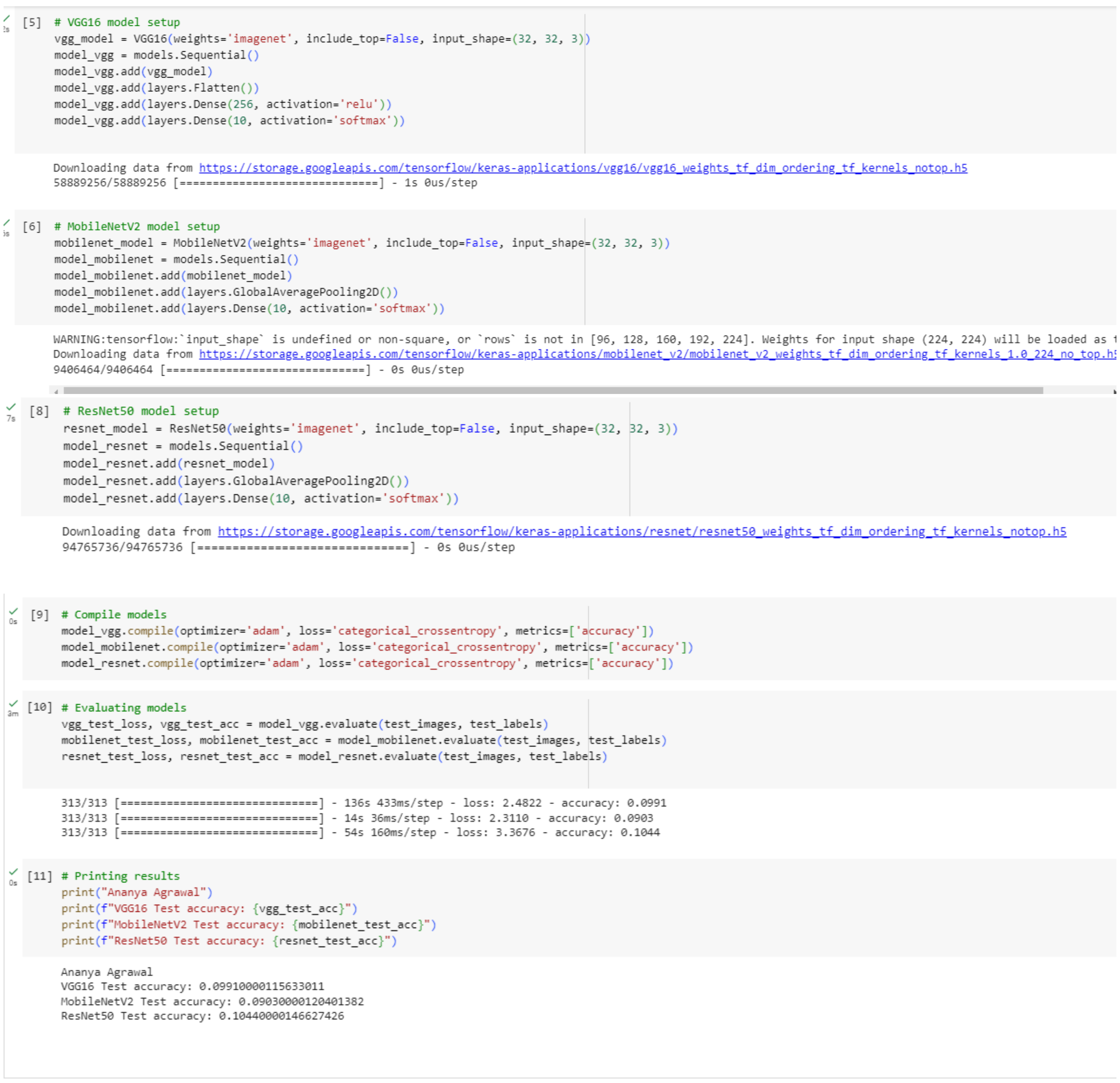


```
[1] import tensorflow as tf
    from keras import datasets, layers, models
    from keras.applications import VGG16, MobileNetV2, ResNet50
    from keras.utils import to_categorical
```

```
[2] # Load CIFAR-10 dataset
    (train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

    Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
    170498071/170498071 [==============================] - 2s 0us/step
```

```
    # Normalize pixel values to a range between 0 and 1
    train_images, test_images = train_images / 255.0, test_images / 255.0
    print("Ananya Agrawal")

    Ananya Agrawal
```

```
[4] # One-hot encode the labels
    train_labels = to_categorical(train_labels, 10)
    test_labels = to_categorical(test_labels, 10)
```



```
[5]  # VGG16 model setup
     vgg_model = VGG16(weights='imagenet', include_top=False, input_shape=(32, 32, 3))
     model_vgg = models.Sequential()
     model_vgg.add(vgg_model)
     model_vgg.add(layers.Flatten())
     model_vgg.add(layers.Dense(256, activation='relu'))
     model_vgg.add(layers.Dense(10, activation='softmax'))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 [==============================] - 1s 0us/step
```

```
[6]  # MobileNetV2 model setup
     mobilenet_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(32, 32, 3))
     model_mobilenet = models.Sequential()
     model_mobilenet.add(mobilenet_model)
     model_mobilenet.add(layers.GlobalAveragePooling2D())
     model_mobilenet.add(layers.Dense(10, activation='softmax'))
```

```
WARNING:tensorflow:`input_shape` is undefined or non-square, or `rows` is not in [96, 128, 160, 192, 224]. Weights for input shape (224, 224) will be loaded as t
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_224_no_top.h5
9406464/9406464 [==============================] - 0s 0us/step
```

```
[8]  # ResNet50 model setup
     resnet_model = ResNet50(weights='imagenet', include_top=False, input_shape=(32, 32, 3))
     model_resnet = models.Sequential()
     model_resnet.add(resnet_model)
     model_resnet.add(layers.GlobalAveragePooling2D())
     model_resnet.add(layers.Dense(10, activation='softmax'))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5
94765736/94765736 [==============================] - 0s 0us/step
```

```
[9]  # Compile models
     model_vgg.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
     model_mobilenet.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
     model_resnet.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
[10] # Evaluating models
     vgg_test_loss, vgg_test_acc = model_vgg.evaluate(test_images, test_labels)
     mobilenet_test_loss, mobilenet_test_acc = model_mobilenet.evaluate(test_images, test_labels)
     resnet_test_loss, resnet_test_acc = model_resnet.evaluate(test_images, test_labels)
```

```
313/313 [==============================] - 136s 433ms/step - loss: 2.4822 - accuracy: 0.0991
313/313 [==============================] - 14s 36ms/step - loss: 2.3110 - accuracy: 0.0903
313/313 [==============================] - 54s 160ms/step - loss: 3.3676 - accuracy: 0.1044
```

```
[11] # Printing results
     print("Ananya Agrawal")
     print(f"VGG16 Test accuracy: {vgg_test_acc}")
     print(f"MobileNetV2 Test accuracy: {mobilenet_test_acc}")
     print(f"ResNet50 Test accuracy: {resnet_test_acc}")
```

```
Ananya Agrawal
VGG16 Test accuracy: 0.09910000115633011
MobileNetV2 Test accuracy: 0.09030000120401382
ResNet50 Test accuracy: 0.10440000146627426
```