Deep Learning

Lab 1

Nayyab Malik
(BSAI-127)

Assigned By

Mam Iqra Nasem

Faculty of Engineering & CS

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

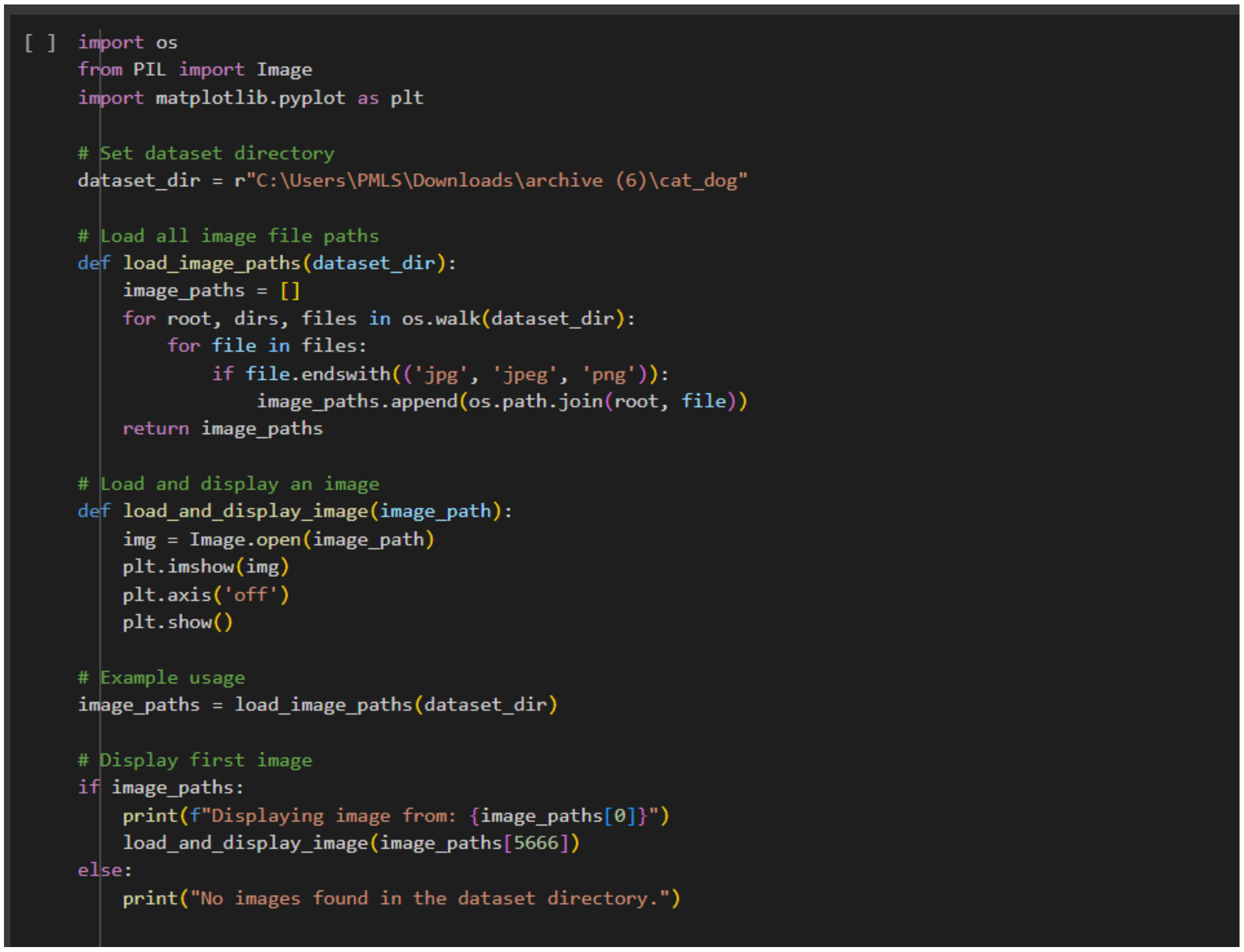
Submission Date: 31th October, 2024

Classification Task using ANN

In deep learning, a classification task involves training a model to categorize input data into predefined classes.

1. Problem Definition and Data Collection:

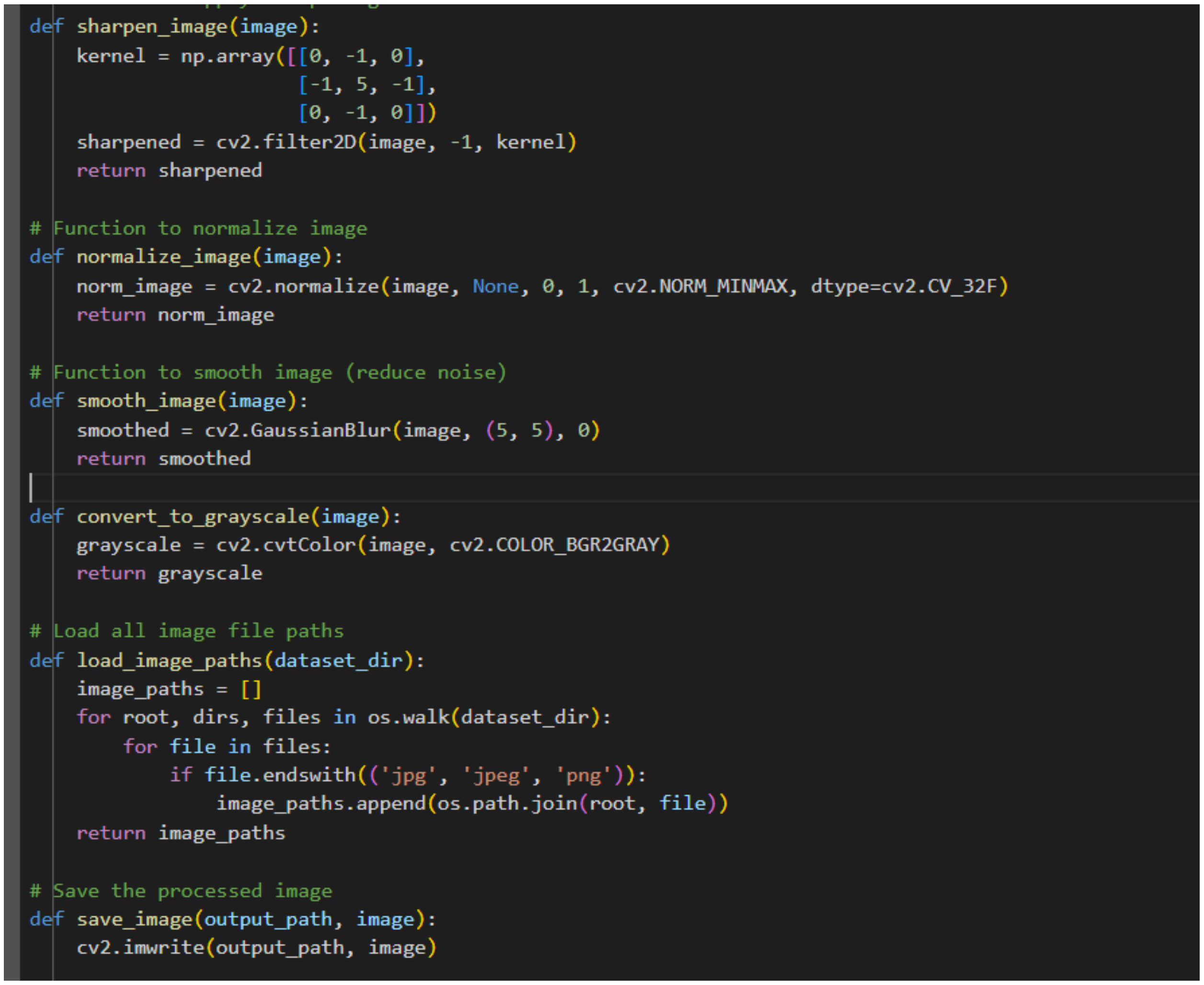
- Define the problem and decide on the categories the model should classify (e.g., dog vs. cat).
- Collect and label data according to these categories (e.g., labeled images for image classification).



```
[ ]  import os
     from PIL import Image
     import matplotlib.pyplot as plt

     # Set dataset directory
     dataset_dir = r"C:\Users\PMLS\Downloads\archive (6)\cat_dog"

     # Load all image file paths
     def load_image_paths(dataset_dir):
         image_paths = []
         for root, dirs, files in os.walk(dataset_dir):
             for file in files:
                 if file.endswith(('jpg', 'jpeg', 'png')):
                     image_paths.append(os.path.join(root, file))
         return image_paths

     # Load and display an image
     def load_and_display_image(image_path):
         img = Image.open(image_path)
         plt.imshow(img)
         plt.axis('off')
         plt.show()

     # Example usage
     image_paths = load_image_paths(dataset_dir)

     # Display first image
     if image_paths:
         print(f"Displaying image from: {image_paths[0]}")
         load_and_display_image(image_paths[5666])
     else:
         print("No images found in the dataset directory.")
```

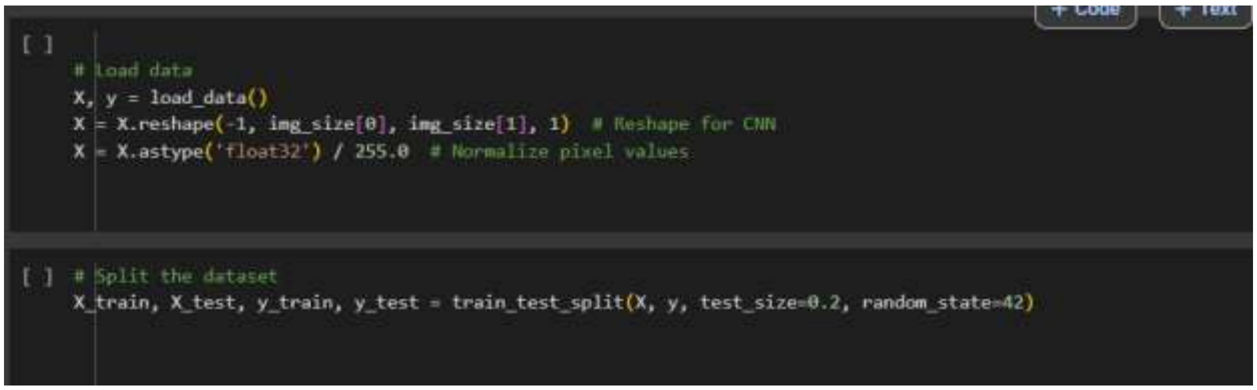
2. Data Preprocessing:

- **Data Cleaning**: Remove or fix corrupted samples by applying sharpening, background removal etc.
- **Normalization/Standardization**: Scale input features to a consistent range (e.g., 0–1 or -1–1).



```python
def sharpen_image(image):
    kernel = np.array([[0, -1, 0],
                       [-1, 5, -1],
                       [0, -1, 0]])
    sharpened = cv2.filter2D(image, -1, kernel)
    return sharpened

# Function to normalize image
def normalize_image(image):
    norm_image = cv2.normalize(image, None, 0, 1, cv2.NORM_MINMAX, dtype=cv2.CV_32F)
    return norm_image

# Function to smooth image (reduce noise)
def smooth_image(image):
    smoothed = cv2.GaussianBlur(image, (5, 5), 0)
    return smoothed

def convert_to_grayscale(image):
    grayscale = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    return grayscale

# Load all image file paths
def load_image_paths(dataset_dir):
    image_paths = []
    for root, dirs, files in os.walk(dataset_dir):
        for file in files:
            if file.endswith(('jpg', 'jpeg', 'png')):
                image_paths.append(os.path.join(root, file))
    return image_paths

# Save the processed image
def save_image(output_path, image):
    cv2.imwrite(output_path, image)
```

3. **<u>Train Test Split</u>:**
   Divide the dataset into training, validation, and test sets.



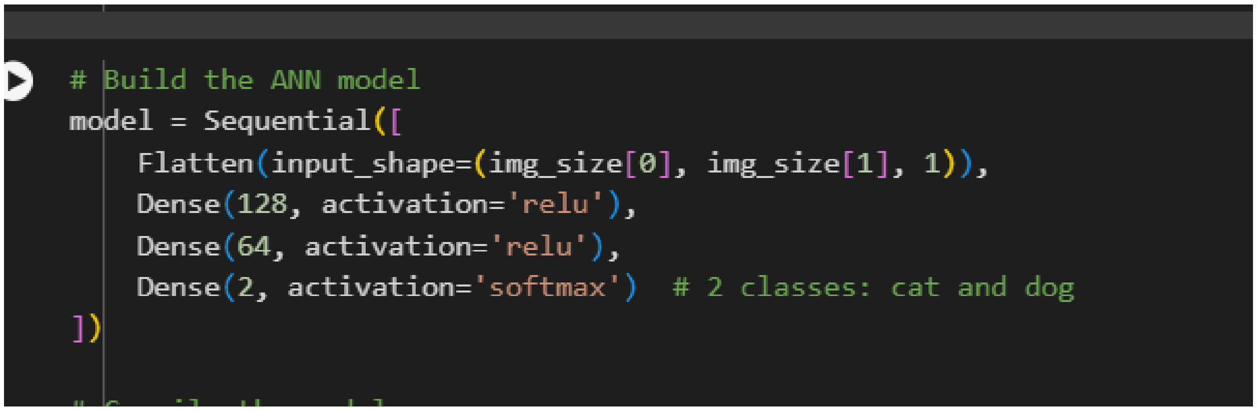
```python
[ ]
    # Load data
    X, y = load_data()
    X = X.reshape(-1, img_size[0], img_size[1], 1)  # Reshape for CNN
    X = X.astype('float32') / 255.0  # Normalize pixel values
```

```python
[ ] # Split the dataset
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

4. **<u>Model Selection</u>:**
   ANNs consist of fully connected (dense) layers. For a basic ANN architecture:

- **Input Layer**: Matches the number of features in your data.

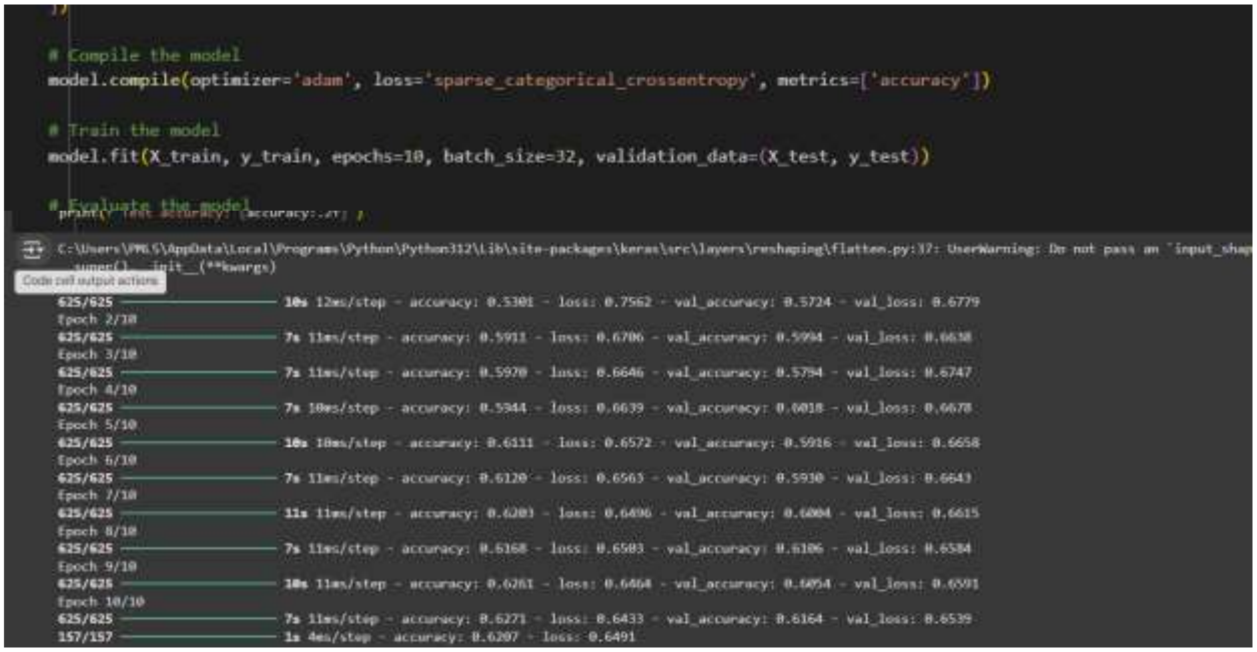
- **Hidden Layers**: Add one or more dense layers with activation functions like ReLU. More layers and neurons can allow the model to capture complex patterns.
- **Output Layer**:
  - For binary classification: Use a single neuron with a sigmoid activation function (output between 0 and 1).
  - For multi-class classification: Use a number of neurons equal to the number of classes with a softmax activation function (outputs probabilities for each class).



```
# Build the ANN model
model = Sequential([
    Flatten(input_shape=(img_size[0], img_size[1], 1)),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(2, activation='softmax')  # 2 classes: cat and dog
])
```

5. <u>Model Training</u>:

- **Compile the Model**: Specify the optimizer, loss function, and metrics (e.g., accuracy).
- **Train the Model**: Use a batch size that balances training speed and stability (e.g., 32 or 64) and train for multiple epochs.
- **Validation**: Monitor the validation loss and accuracy to prevent overfitting. Use techniques like early stopping to halt training when validation performance stops improving.

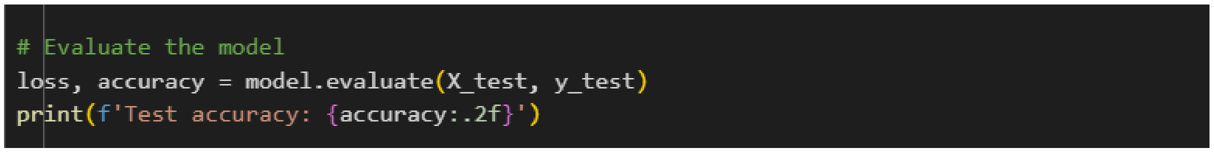


```
# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))

# Evaluate the model
```

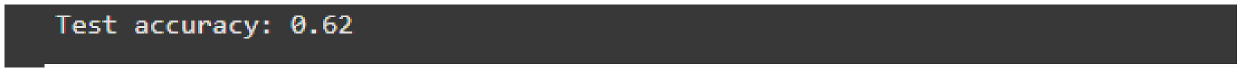
```
C:\Users\PMLS\AppData\Local\Programs\Python\Python312\Lib\site-packages\keras\src\layers\reshaping\flatten.py:37: UserWarning: Do not pass an `input_shap
  super().__init__(**kwargs)
625/625 ━━━━━━━━━━━━━━━━━━━━ 10s 12ms/step - accuracy: 0.5301 - loss: 0.7562 - val_accuracy: 0.5724 - val_loss: 0.6779
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - accuracy: 0.5911 - loss: 0.6706 - val_accuracy: 0.5994 - val_loss: 0.6638
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - accuracy: 0.5970 - loss: 0.6646 - val_accuracy: 0.5794 - val_loss: 0.6747
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.5944 - loss: 0.6639 - val_accuracy: 0.6018 - val_loss: 0.6678
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 10s 10ms/step - accuracy: 0.6111 - loss: 0.6572 - val_accuracy: 0.5916 - val_loss: 0.6658
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - accuracy: 0.6120 - loss: 0.6563 - val_accuracy: 0.5930 - val_loss: 0.6643
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 11s 11ms/step - accuracy: 0.6203 - loss: 0.6496 - val_accuracy: 0.6004 - val_loss: 0.6615
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - accuracy: 0.6168 - loss: 0.6503 - val_accuracy: 0.6106 - val_loss: 0.6584
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 10s 11ms/step - accuracy: 0.6261 - loss: 0.6464 - val_accuracy: 0.6054 - val_loss: 0.6591
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 7s 11ms/step - accuracy: 0.6271 - loss: 0.6433 - val_accuracy: 0.6164 - val_loss: 0.6539
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6207 - loss: 0.6491
```

6. Evaluating the Model:

- Use the test set to evaluate the trained ANN on unseen data.
- Calculate metrics like **accuracy**, **precision**, **recall**, and **F1 score**.



```python
# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f'Test accuracy: {accuracy:.2f}')
```



```
Test accuracy: 0.62
```

Deep Learning

Lab 1

Nayyab Malik
(BSAI-127)

Assigned By

Mam Iqra Nasem

Faculty of Engineering & CS

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

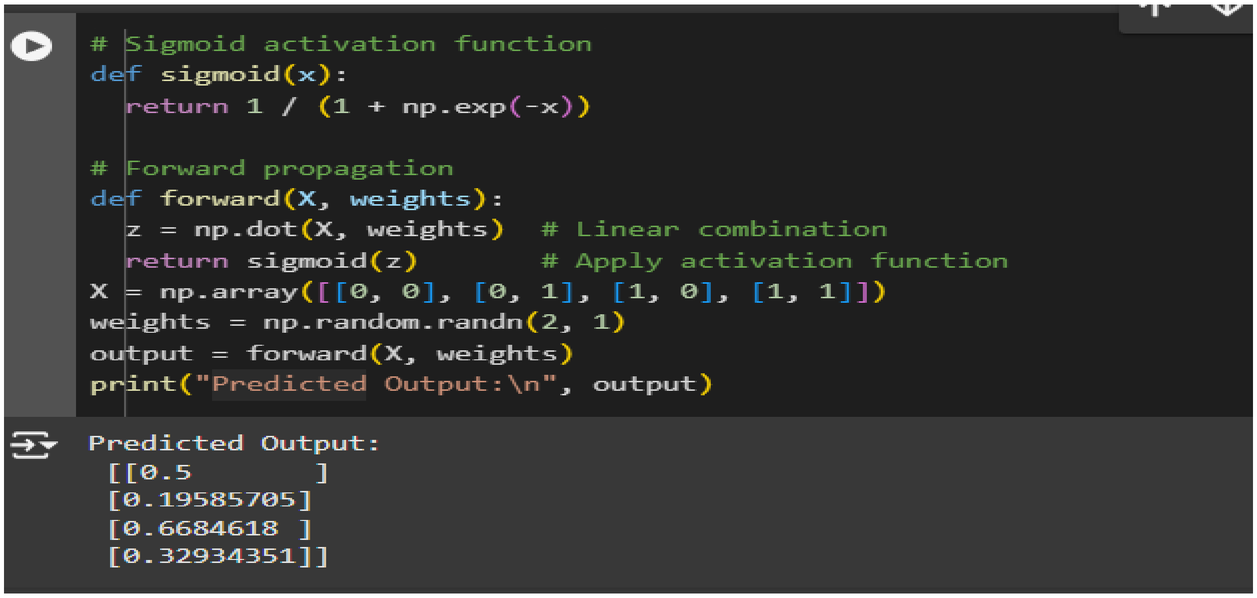
Submission Date: 31th October, 2024

Gradient Descent and it’s Working

Gradient Descent is an optimization algorithm used to minimize the loss function in machine learning and deep learning models. Its goal is to find the optimal parameters (weights) for a model that yield the lowest possible loss, thereby improving model accuracy.

1. Initialization:

- Start with Initial Parameters: Begin with randomly initialized model parameters (weights and biases). These parameters will be updated iteratively predict output.

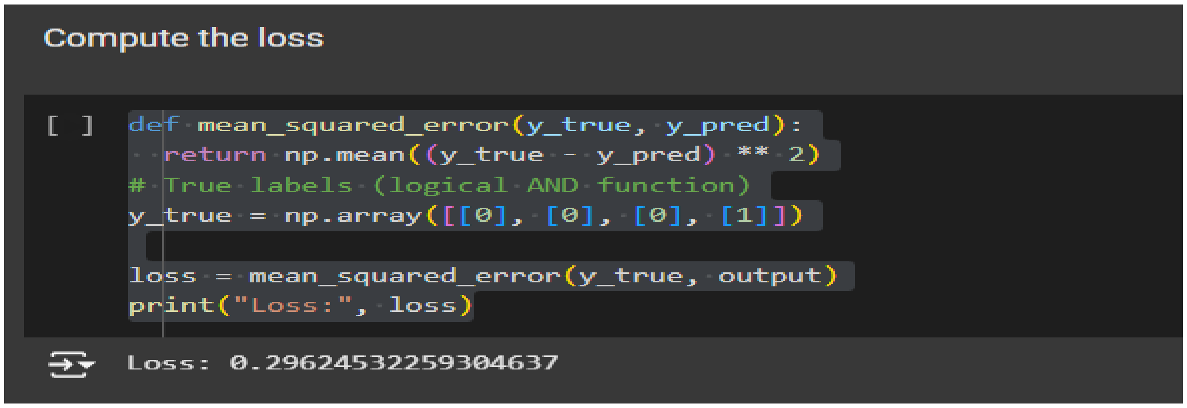


```
# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Forward propagation
def forward(X, weights):
    z = np.dot(X, weights)  # Linear combination
    return sigmoid(z)       # Apply activation function
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
weights = np.random.randn(2, 1)
output = forward(X, weights)
print("Predicted Output:\n", output)
```

```
Predicted Output:
 [[0.5       ]
 [0.19585705]
 [0.6684618 ]
 [0.32934351]]
```

2. Compute the Loss Function:

- Forward Pass: Pass input data through the model to compute predictions.
- Calculate Loss: Compare the predictions with the actual values using a loss function, such as Mean Squared Error for regression or Cross-Entropy for classification. The loss quantifies how far off the predictions are from the true values.



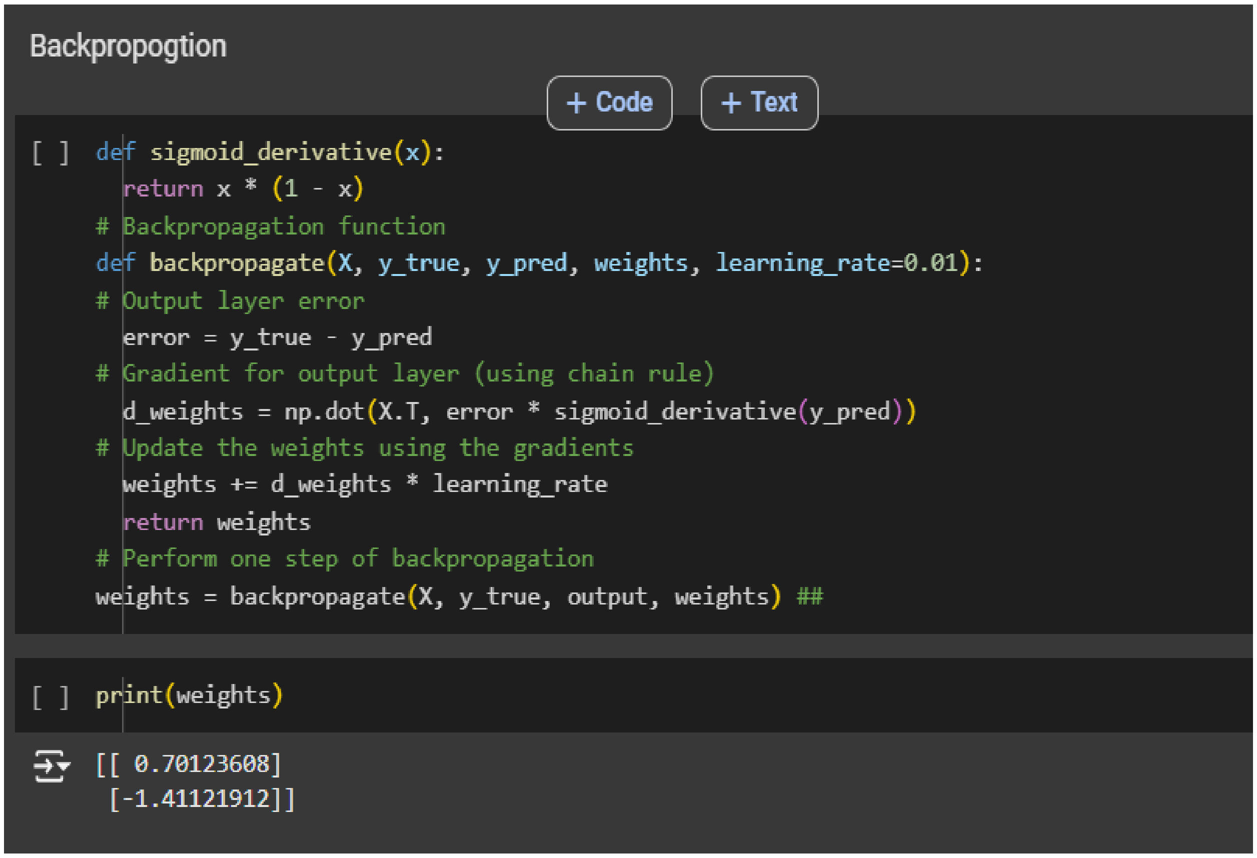
Compute the loss

```
def mean_squared_error(y_true, y_pred):
  return np.mean((y_true - y_pred) ** 2)
# True labels (logical AND function)
y_true = np.array([[0], [0], [0], [1]])

loss = mean_squared_error(y_true, output)
print("Loss:", loss)
```

```
Loss: 0.29624532259304637
```

3. Calculate the Gradient:

- Backpropagation: Perform backpropagation to calculate the gradient, which is the derivative of the loss with respect to each parameter. The gradient represents the direction and rate at which the loss function changes with respect to changes in each parameter.
- Direction of Steepest Ascent: The gradient points in the direction where the loss increases most steeply. However, we want to minimize the loss, so we'll move in the opposite direction.



```
[ ] def sigmoid_derivative(x):
      return x * (1 - x)
    # Backpropagation function
    def backpropagate(X, y_true, y_pred, weights, learning_rate=0.01):
    # Output layer error
      error = y_true - y_pred
    # Gradient for output layer (using chain rule)
      d_weights = np.dot(X.T, error * sigmoid_derivative(y_pred))
    # Update the weights using the gradients
      weights += d_weights * learning_rate
      return weights
    # Perform one step of backpropagation
    weights = backpropagate(X, y_true, output, weights) ##
```

```
[ ] print(weights)
```

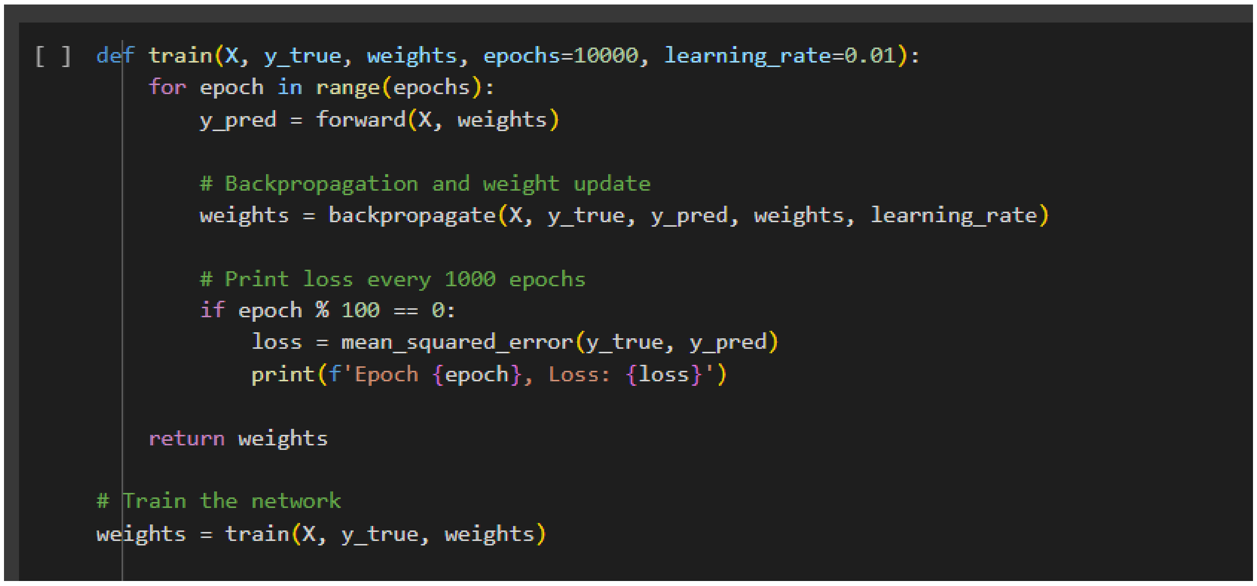
```
[[ 0.70123608]
 [-1.41121912]]
```

4. Update the Parameters:

- Move in the Opposite Direction of the Gradient: Adjust each parameter by taking a small step in the opposite direction of the gradient. The amount of the step is controlled by a hyperparameter called the learning rate.
- Update Formula:
  - For a weight w:

$$w=w-\alpha * \partial L/\partial w$$

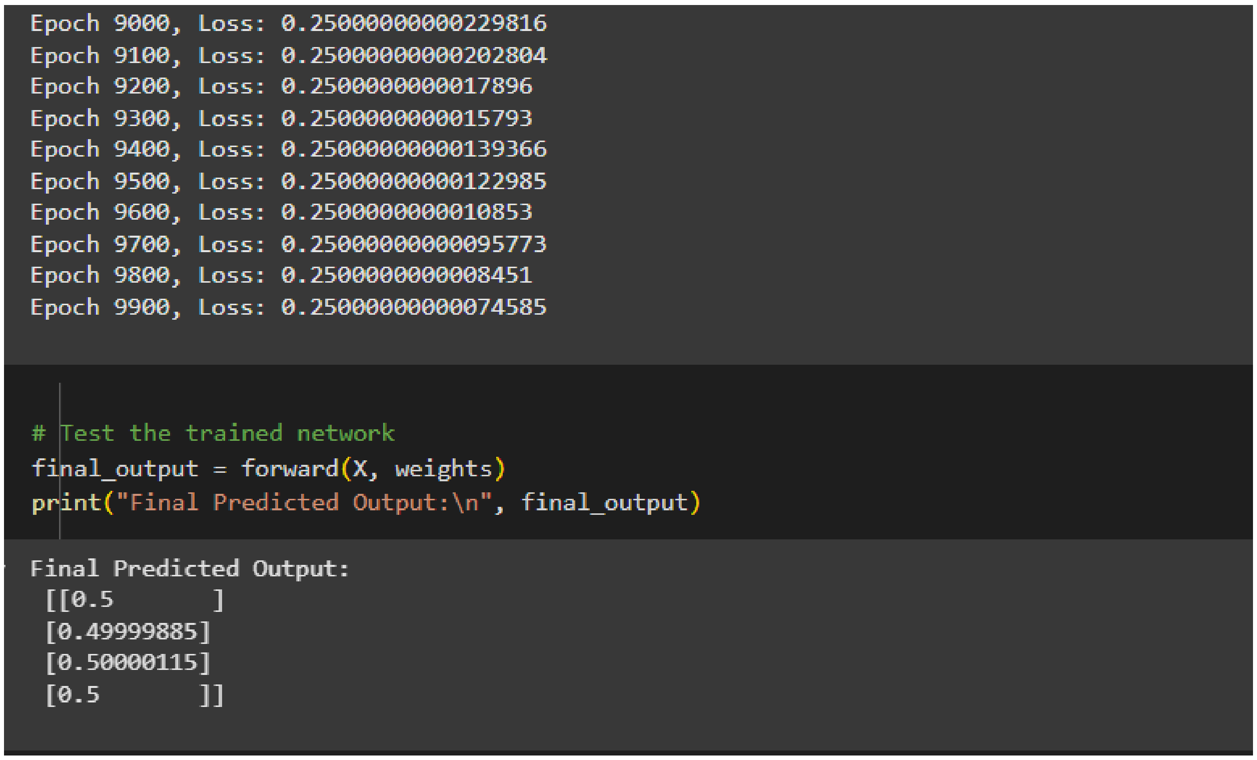
Here, α alpha is the learning rate ∂L/∂w is the gradient of the loss with respect to w.



```
[ ] def train(X, y_true, weights, epochs=10000, learning_rate=0.01):
        for epoch in range(epochs):
            y_pred = forward(X, weights)

            # Backpropagation and weight update
            weights = backpropagate(X, y_true, y_pred, weights, learning_rate)

            # Print loss every 1000 epochs
            if epoch % 100 == 0:
                loss = mean_squared_error(y_true, y_pred)
                print(f'Epoch {epoch}, Loss: {loss}')

        return weights

    # Train the network
    weights = train(X, y_true, weights)
```

5. Iterate Until Convergence:

- Repeat steps 2–4 for multiple iterations (epochs) until the loss stops decreasing significantly, or until the model reaches a pre-defined number of epochs.
- Convergence: This is the point at which the parameters no longer change much with additional iterations, ideally indicating that the model has found the minimum loss.



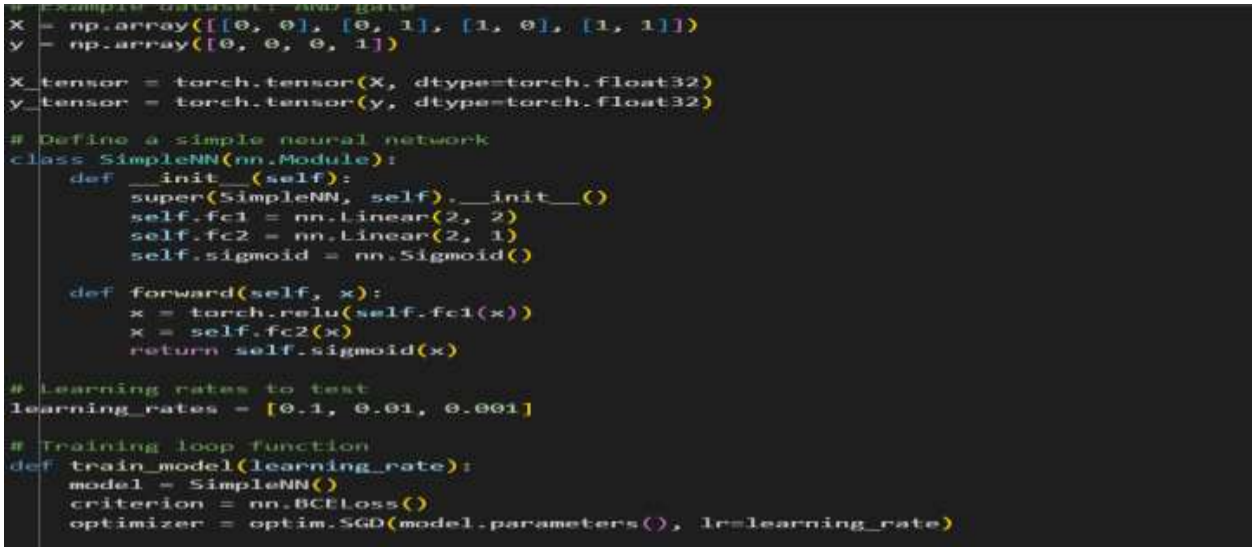
```
Epoch 9000, Loss: 0.25000000000229816
Epoch 9100, Loss: 0.25000000000202804
Epoch 9200, Loss: 0.2500000000017896
Epoch 9300, Loss: 0.2500000000015793
Epoch 9400, Loss: 0.25000000000139366
Epoch 9500, Loss: 0.25000000000122985
Epoch 9600, Loss: 0.2500000000010853
Epoch 9700, Loss: 0.25000000000095773
Epoch 9800, Loss: 0.2500000000008451
Epoch 9900, Loss: 0.25000000000074585
```

```
# Test the trained network
final_output = forward(X, weights)
print("Final Predicted Output:\n", final_output)
```

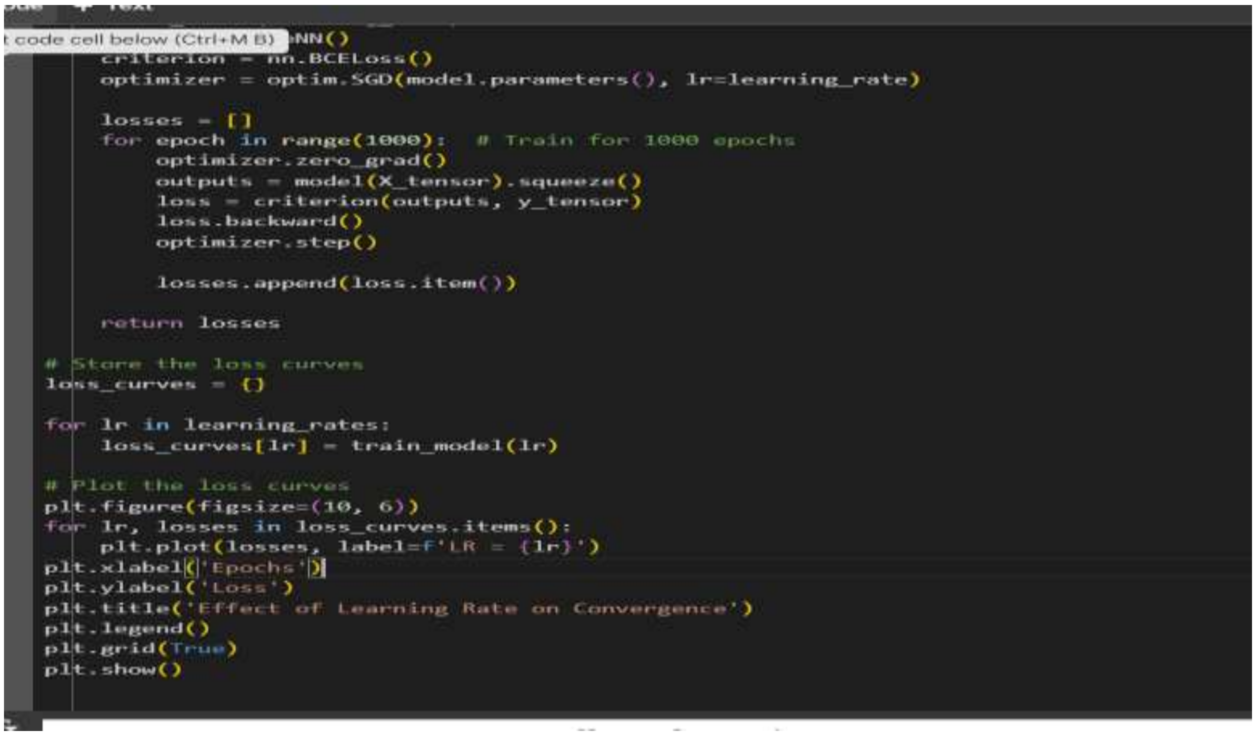
```
Final Predicted Output:
 [[0.5       ]
 [0.49999885]
 [0.50000115]
 [0.5       ]]
```

Lab Experiment 1: Effect of Learning Rate on Convergence

The learning rate determines the size of the steps taken during the optimization process (gradient descent) to minimize the loss function. It is a hyperparameter that controls how much the weights in the network are adjusted with respect to the loss gradient.



```python
# Example dataset: AND gate
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])

X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.float32)

# Define a simple neural network
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(2, 2)
        self.fc2 = nn.Linear(2, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return self.sigmoid(x)

# Learning rates to test
learning_rates = [0.1, 0.01, 0.001]

# Training loop function
def train_model(learning_rate):
    model = SimpleNN()
    criterion = nn.BCELoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

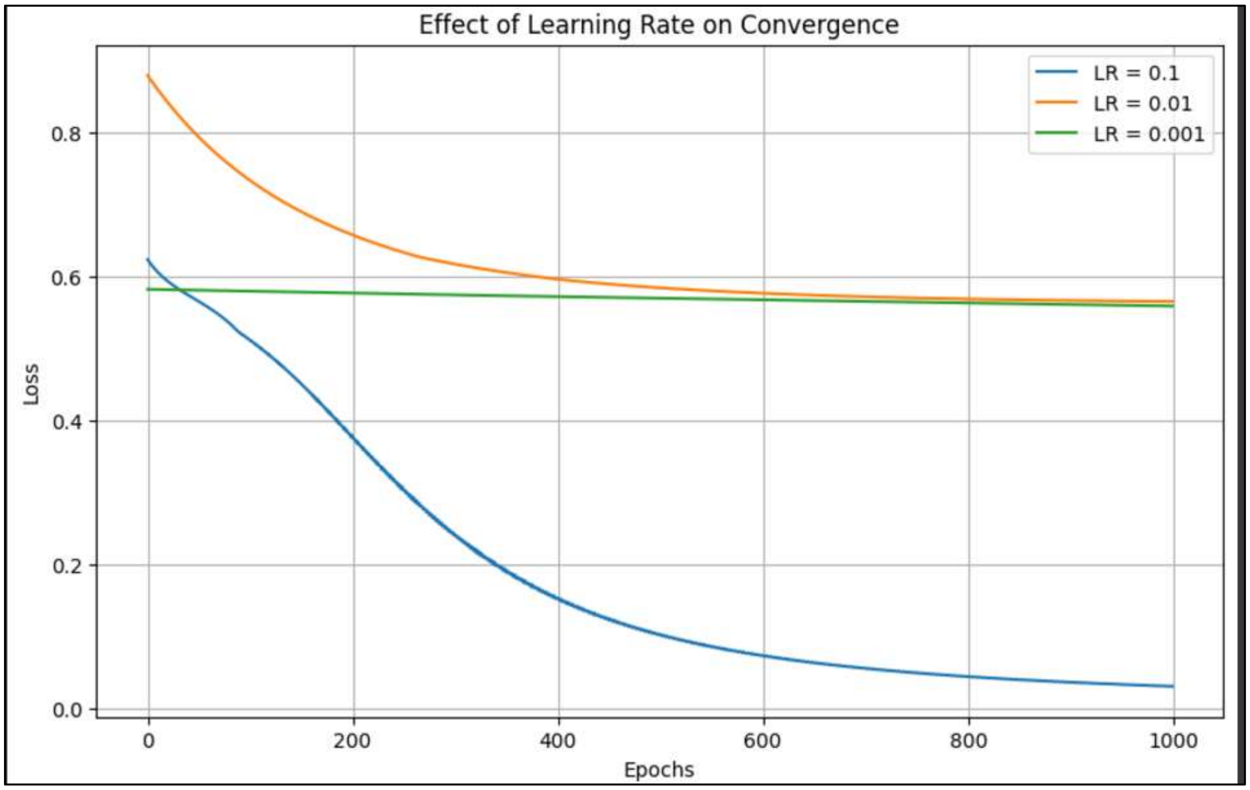


```python
t code cell below (Ctrl+M B) NN()
    criterion = nn.BCELoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    losses = []
    for epoch in range(1000):  # Train for 1000 epochs
        optimizer.zero_grad()
        outputs = model(X_tensor).squeeze()
        loss = criterion(outputs, y_tensor)
        loss.backward()
        optimizer.step()

        losses.append(loss.item())

    return losses

# Store the loss curves
loss_curves = {}

for lr in learning_rates:
    loss_curves[lr] = train_model(lr)

# Plot the loss curves
plt.figure(figsize=(10, 6))
for lr, losses in loss_curves.items():
    plt.plot(losses, label=f'LR = {lr}')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Effect of Learning Rate on Convergence')
plt.legend()
plt.grid(True)
plt.show()
```

1. **Learning Rate = 0.1 (Blue Line)**:
   - The loss decreases rapidly in the initial epochs but then shows signs of oscillation or even divergence. This is likely because the learning rate is too high, causing the

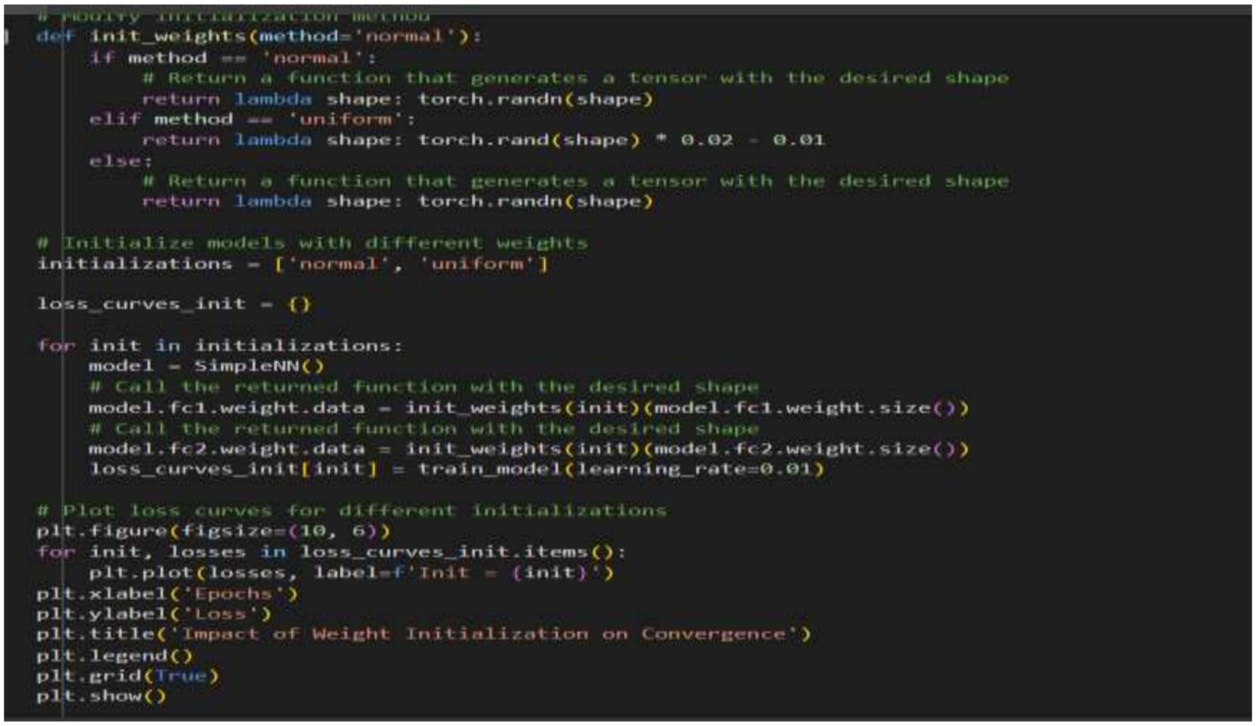
model to make large updates to the weights, overshooting the optimal point and preventing smooth convergence.

2. **Learning Rate = 0.01 (Orange Line)**:
   - This learning rate provides a more stable descent compared to the 0.1 rate. The loss decreases steadily and seems to stabilize after a number of epochs. This suggests that a moderate learning rate helps the network converge without overshooting the optimal values.
3. **Learning Rate = 0.001 (Green Line)**:
   - With this low learning rate, the convergence is much slower. The loss decreases very slowly, which might indicate that the updates to the weights are too small, making the optimization process less efficient. However, the model is more likely to converge to the global minimum, avoiding oscillations or divergence.



Lab Experiment 2: Impact of Initial Weights on Convergence

The initial weights are the starting values of the parameters (weights and biases) of the neural network. Proper initialization can affect how fast and effectively the network converges



```
# Modify initialization method
def init_weights(method='normal'):
    if method == 'normal':
        # Return a function that generates a tensor with the desired shape
        return lambda shape: torch.randn(shape)
    elif method == 'uniform':
        return lambda shape: torch.rand(shape) * 0.02 - 0.01
    else:
        # Return a function that generates a tensor with the desired shape
        return lambda shape: torch.randn(shape)

# Initialize models with different weights
initializations = ['normal', 'uniform']

loss_curves_init = {}

for init in initializations:
    model = SimpleNN()
    # Call the returned function with the desired shape
    model.fc1.weight.data = init_weights(init)(model.fc1.weight.size())
    # Call the returned function with the desired shape
    model.fc2.weight.data = init_weights(init)(model.fc2.weight.size())
    loss_curves_init[init] = train_model(learning_rate=0.01)

# Plot loss curves for different initializations
plt.figure(figsize=(10, 6))
for init, losses in loss_curves_init.items():
    plt.plot(losses, label=f'Init = {init}')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Impact of Weight Initialization on Convergence')
plt.legend()
plt.grid(True)
plt.show()
```

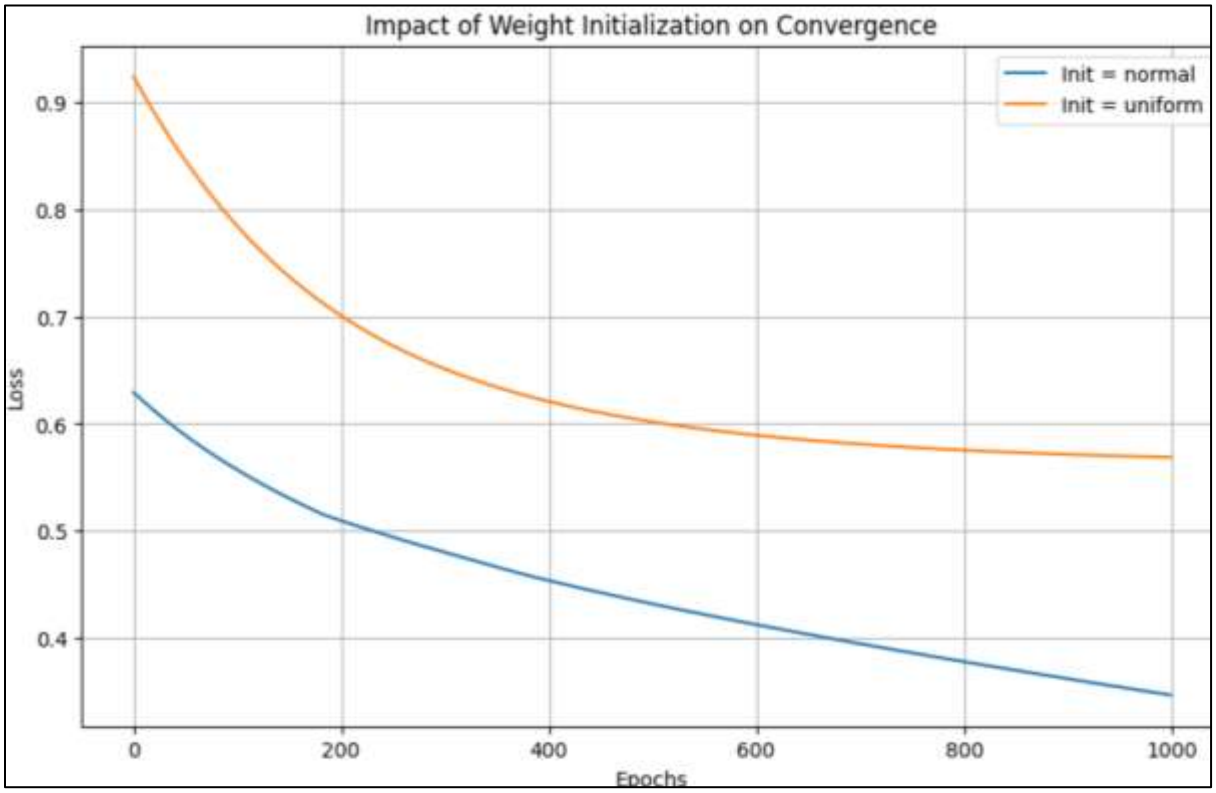
1. **Normal Initialization:** Weights are initialized using a normal distribution.
2. **Uniform Initialization:** Weights are initialized using a uniform distribution.

Interpretation

- **Convergence:** The x-axis represents the number of training epochs, and the y-axis indicates the loss value. A lower loss implies better model performance.
- **Loss Curves:** Both curves demonstrate a general downward trend, suggesting that the model is learning and improving over time.
- **Comparison:** The "Init = normal" curve consistently has a lower loss value than the "Init = uniform" curve. This implies that normal initialization leads to faster convergence and potentially superior performance.

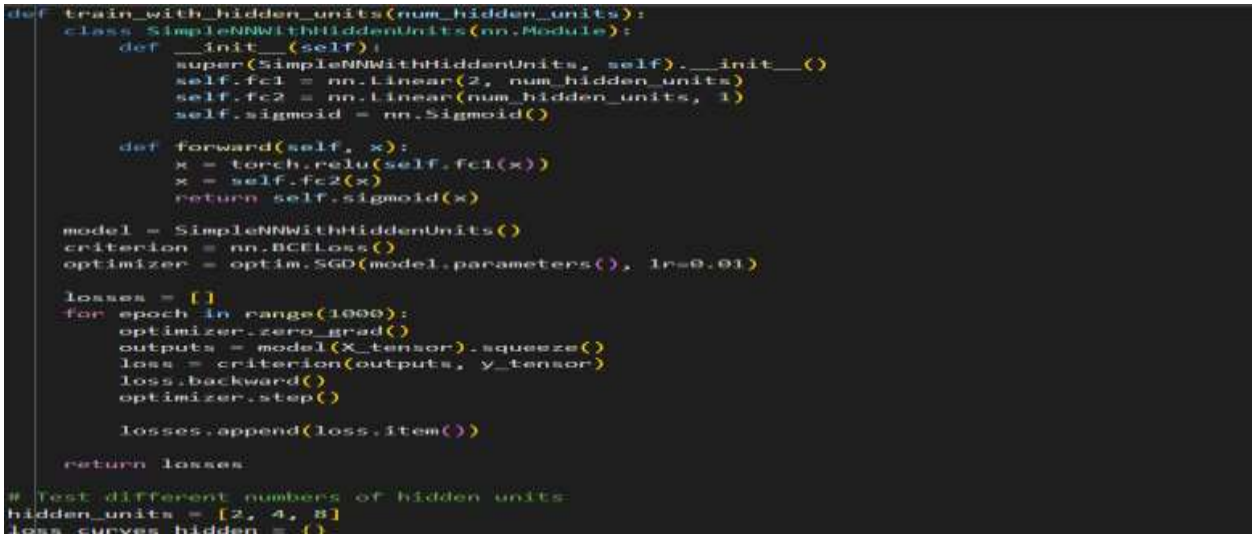
Possible Reasons for Normal Initialization's Success

- **Variance:** Normal initialization might offer a better balance of variance in the initial weights, which can aid the model in exploring the solution space more efficiently.
- **Vanishing/Exploding Gradients:** Normal initialization might be less susceptible to the vanishing or exploding gradient problem, which can hinder training.

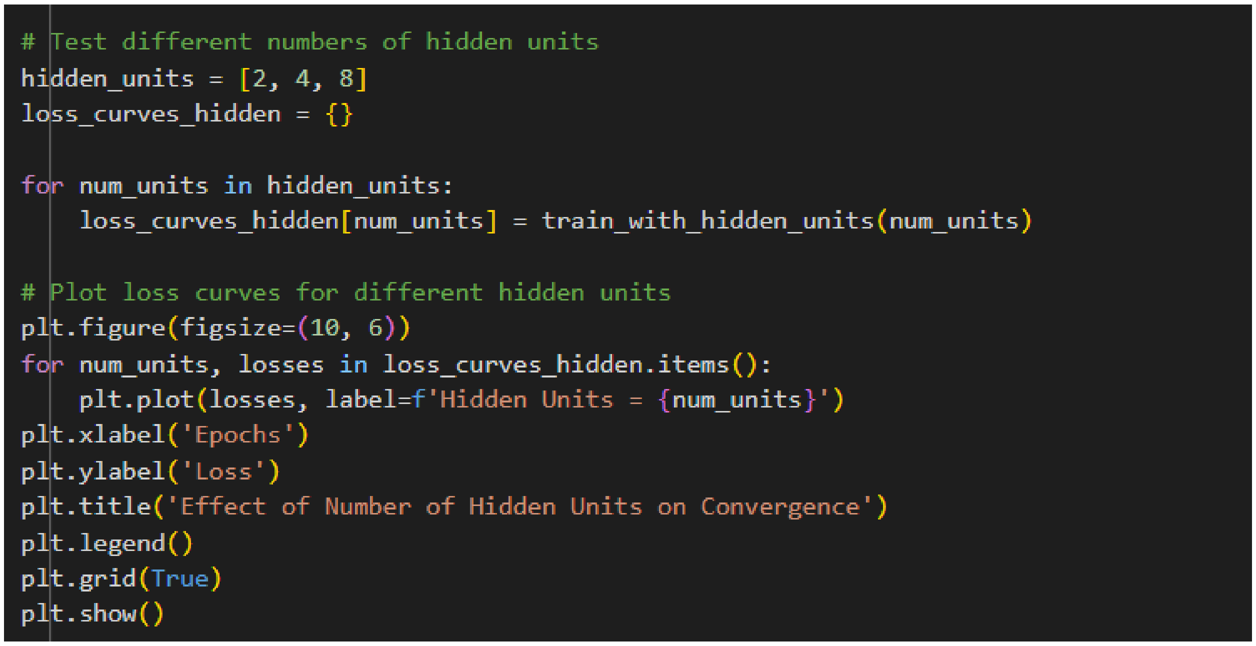


Lab Experiment 3: **<u>Effect of Number of Hidden Units</u>**

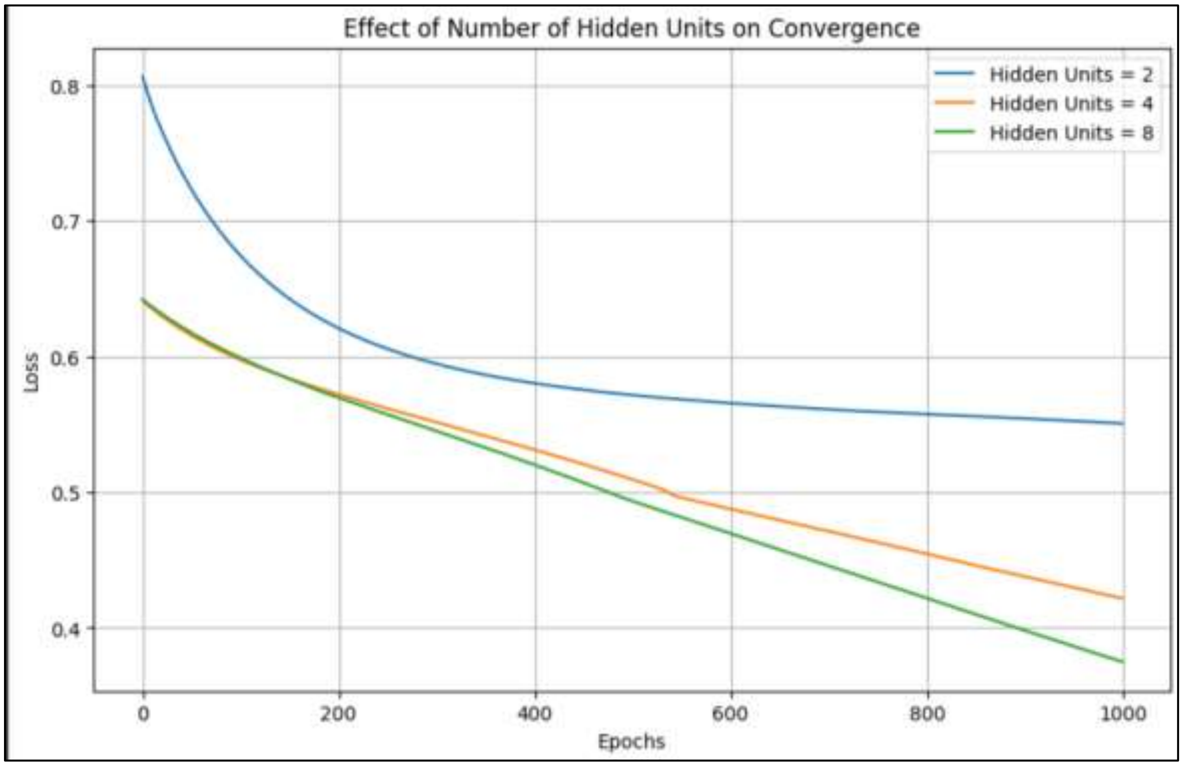
Hidden layers are the layers between the input and output layers in a neural network. They contain neurons (also called units) that apply transformations to the input data using learned weights and biases.



```
def train_with_hidden_units(num_hidden_units):
    class SimpleNNWithHiddenUnits(nn.Module):
        def __init__(self):
            super(SimpleNNWithHiddenUnits, self).__init__()
            self.fc1 = nn.Linear(2, num_hidden_units)
            self.fc2 = nn.Linear(num_hidden_units, 1)
            self.sigmoid = nn.Sigmoid()

        def forward(self, x):
            x = torch.relu(self.fc1(x))
            x = self.fc2(x)
            return self.sigmoid(x)

    model = SimpleNNWithHiddenUnits()
    criterion = nn.BCELoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01)

    losses = []
    for epoch in range(1000):
        optimizer.zero_grad()
        outputs = model(X_tensor).squeeze()
        loss = criterion(outputs, y_tensor)
        loss.backward()
        optimizer.step()

        losses.append(loss.item())

    return losses

# Test different numbers of hidden units
hidden_units = [2, 4, 8]
loss_curves_hidden = {}
```



```
# Test different numbers of hidden units
hidden_units = [2, 4, 8]
loss_curves_hidden = {}

for num_units in hidden_units:
    loss_curves_hidden[num_units] = train_with_hidden_units(num_units)

# Plot loss curves for different hidden units
plt.figure(figsize=(10, 6))
for num_units, losses in loss_curves_hidden.items():
    plt.plot(losses, label=f'Hidden Units = {num_units}')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Effect of Number of Hidden Units on Convergence')
plt.legend()
plt.grid(True)
plt.show()
```



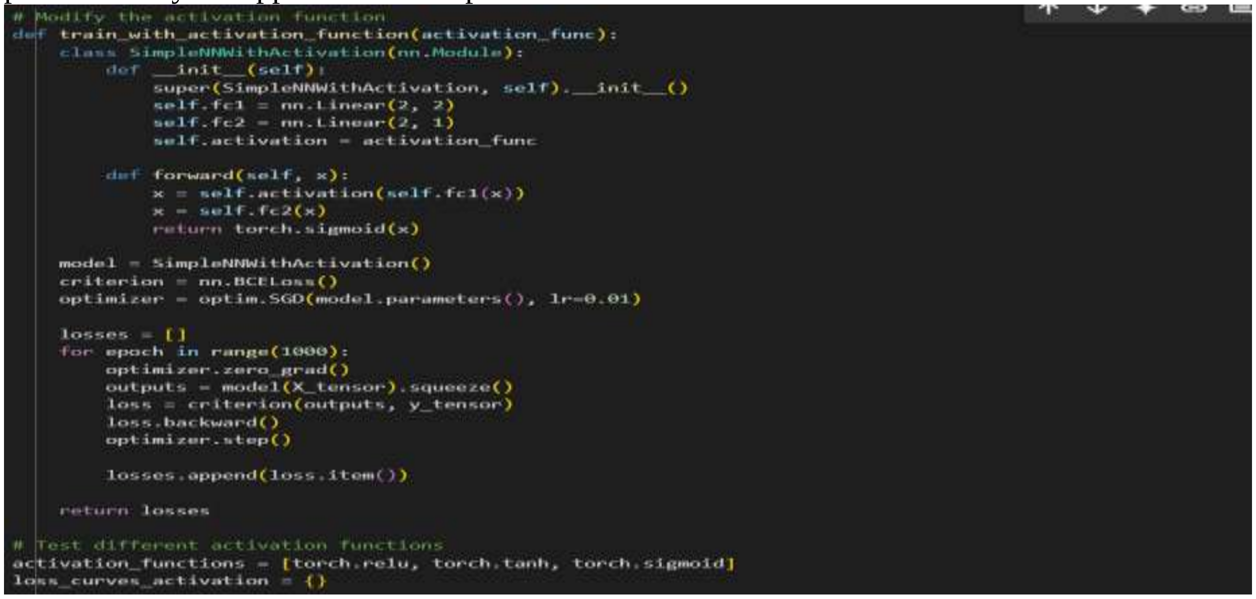
Interpretation

- **Convergence:** The x-axis represents the number of training epochs, and the y-axis indicates the loss value. A lower loss implies better model performance.

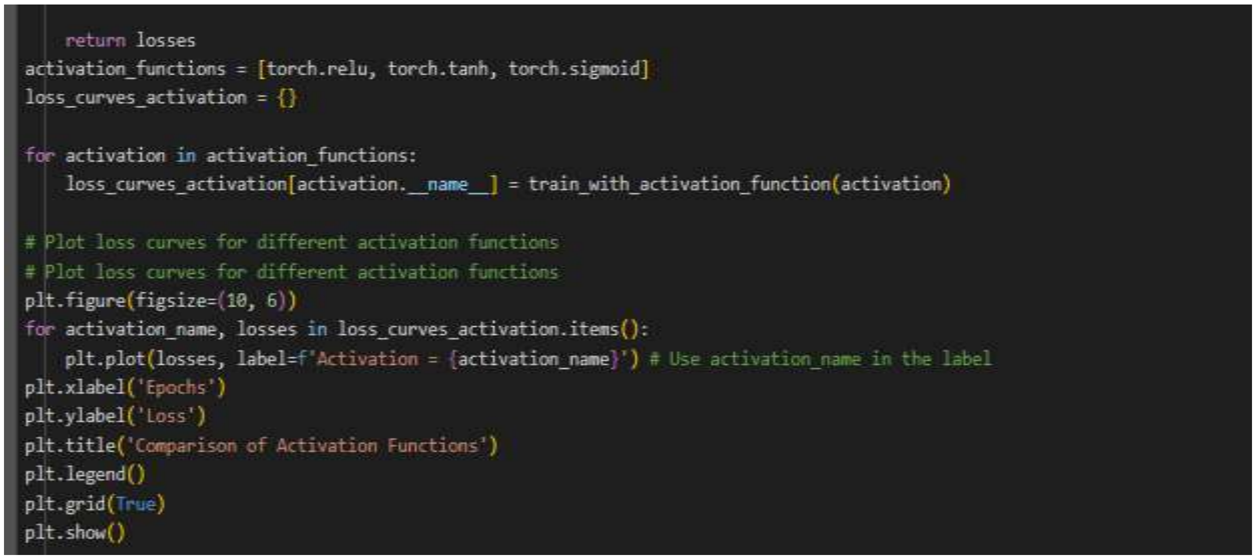
- **Loss Curves:** All three curves show a general downward trend, suggesting that the model is learning and improving over time.
- **Comparison:**
    - **Hidden Units = 2:** The curve starts with a higher loss and converges more slowly.
    - **Hidden Units = 4:** The curve starts with a lower loss and converges faster than the 2-unit model.
    - **Hidden Units = 8:** The curve starts with the lowest loss and converges the fastest among the three.

Lab Experiment 4: Comparing Activation Functions

Activation functions introduce non-linearity into the network, enabling it to learn complex patterns. They are applied to the output of each neuron.



```python
# Modify the activation function
def train_with_activation_function(activation_func):
    class SimpleNNWithActivation(nn.Module):
        def __init__(self):
            super(SimpleNNWithActivation, self).__init__()
            self.fc1 = nn.Linear(2, 2)
            self.fc2 = nn.Linear(2, 1)
            self.activation = activation_func

        def forward(self, x):
            x = self.activation(self.fc1(x))
            x = self.fc2(x)
            return torch.sigmoid(x)

    model = SimpleNNWithActivation()
    criterion = nn.BCELoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01)

    losses = []
    for epoch in range(1000):
        optimizer.zero_grad()
        outputs = model(X_tensor).squeeze()
        loss = criterion(outputs, y_tensor)
        loss.backward()
        optimizer.step()

        losses.append(loss.item())

    return losses

# Test different activation functions
activation_functions = [torch.relu, torch.tanh, torch.sigmoid]
loss_curves_activation = {}
```

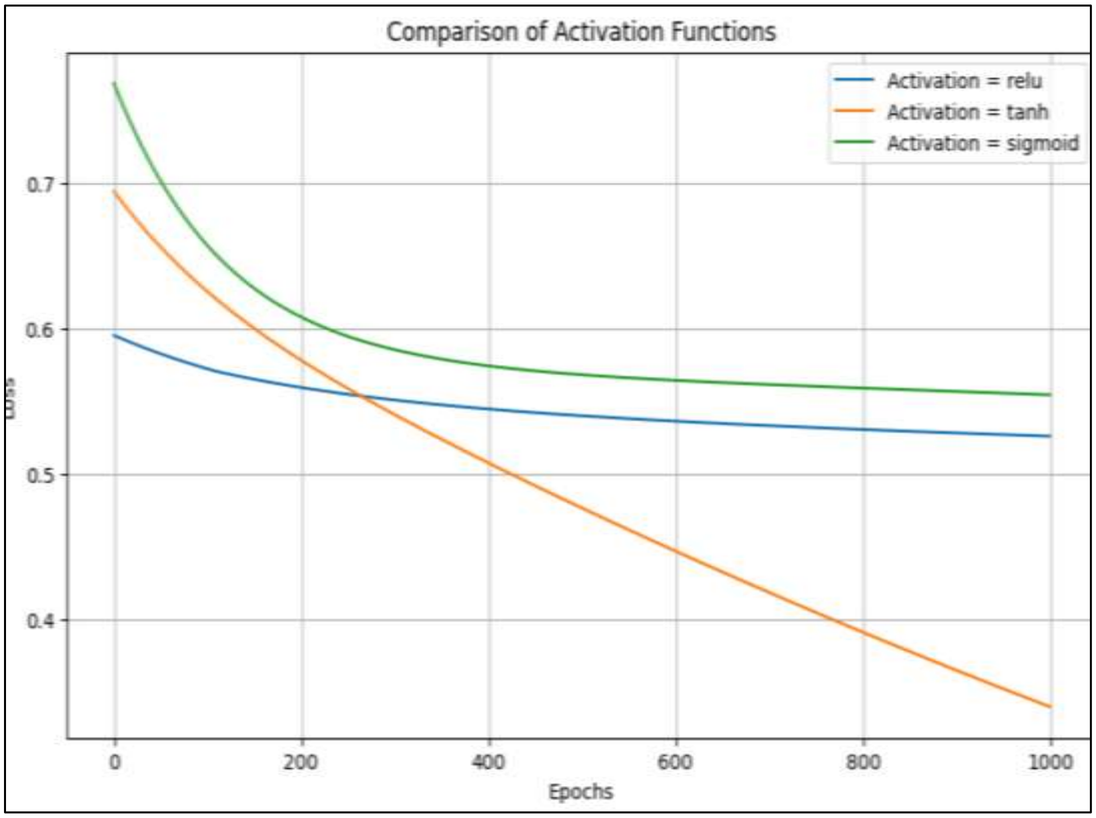


```python
    return losses
activation_functions = [torch.relu, torch.tanh, torch.sigmoid]
loss_curves_activation = {}

for activation in activation_functions:
    loss_curves_activation[activation.__name__] = train_with_activation_function(activation)

# Plot loss curves for different activation functions
# Plot loss curves for different activation functions
plt.figure(figsize=(10, 6))
for activation_name, losses in loss_curves_activation.items():
    plt.plot(losses, label=f'Activation = {activation_name}') # Use activation_name in the label
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Comparison of Activation Functions')
plt.legend()
plt.grid(True)
plt.show()
```

1. **ReLU (Rectified Linear Unit):** A piecewise linear function that outputs the input directly if it's positive, and zero otherwise.
2. **Tanh (Hyperbolic Tangent):** A smooth function that outputs values between -1 and 1.
3. **Sigmoid:** A smooth function that outputs values between 0 and 1.

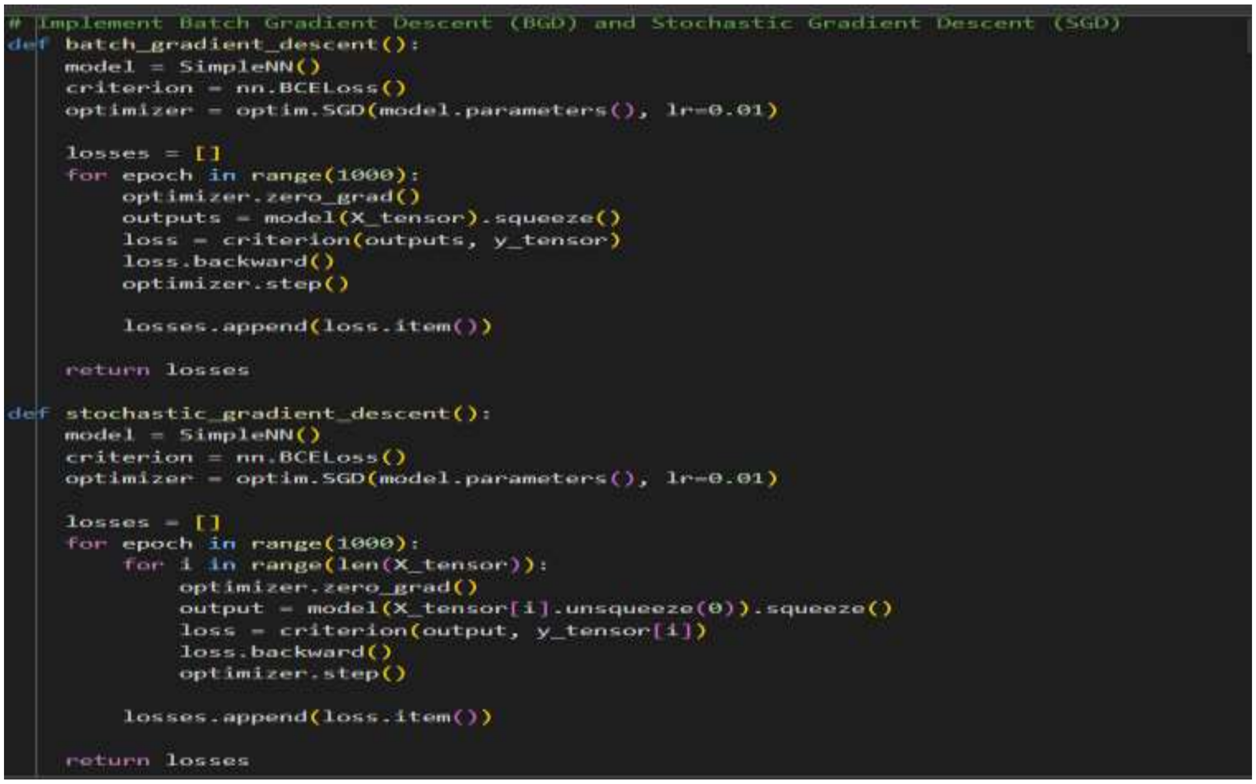
Interpretation

- **Convergence:** The x-axis represents the number of training epochs, and the y-axis indicates the loss value. A lower loss implies better model performance.
- **Loss Curves:** All three curves show a general downward trend, suggesting that the model is learning and improving over time.
- **Comparison:**
    - **ReLU:** The ReLU curve converges the fastest and reaches the lowest loss value, indicating that it's a highly effective activation function for this particular model and dataset.
    - **Tanh:** The Tanh curve converges at a moderate pace and reaches a slightly higher loss value compared to ReLU.
    - **Sigmoid:** The Sigmoid curve converges the slowest and reaches the highest loss value among the three, suggesting that it's less suitable for this specific scenario.

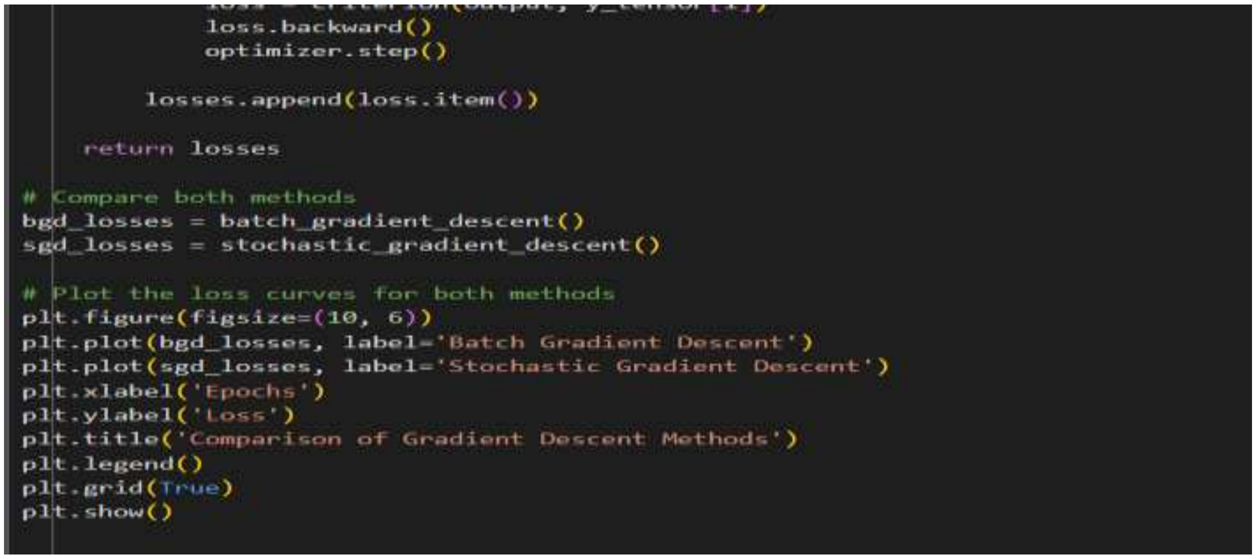


Lab Experiment 5: Batch Gradient Descent vs. Stochastic Gradient Descent

Both **Batch Gradient Descent (BGD)** and **Stochastic Gradient Descent (SGD)** are optimization algorithms used for training machine learning models, particularly neural networks, by minimizing the loss function. They are both used to update model parameters (weights) based on the gradient of the loss with respect to those parameters, but they differ in how they compute and apply the gradient updates.



```
# Implement Batch Gradient Descent (BGD) and Stochastic Gradient Descent (SGD)
def batch_gradient_descent():
    model = SimpleNN()
    criterion = nn.BCELoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01)

    losses = []
    for epoch in range(1000):
        optimizer.zero_grad()
        outputs = model(X_tensor).squeeze()
        loss = criterion(outputs, y_tensor)
        loss.backward()
        optimizer.step()

        losses.append(loss.item())

    return losses

def stochastic_gradient_descent():
    model = SimpleNN()
    criterion = nn.BCELoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01)

    losses = []
    for epoch in range(1000):
        for i in range(len(X_tensor)):
            optimizer.zero_grad()
            output = model(X_tensor[i].unsqueeze(0)).squeeze()
            loss = criterion(output, y_tensor[i])
            loss.backward()
            optimizer.step()

        losses.append(loss.item())

    return losses
```

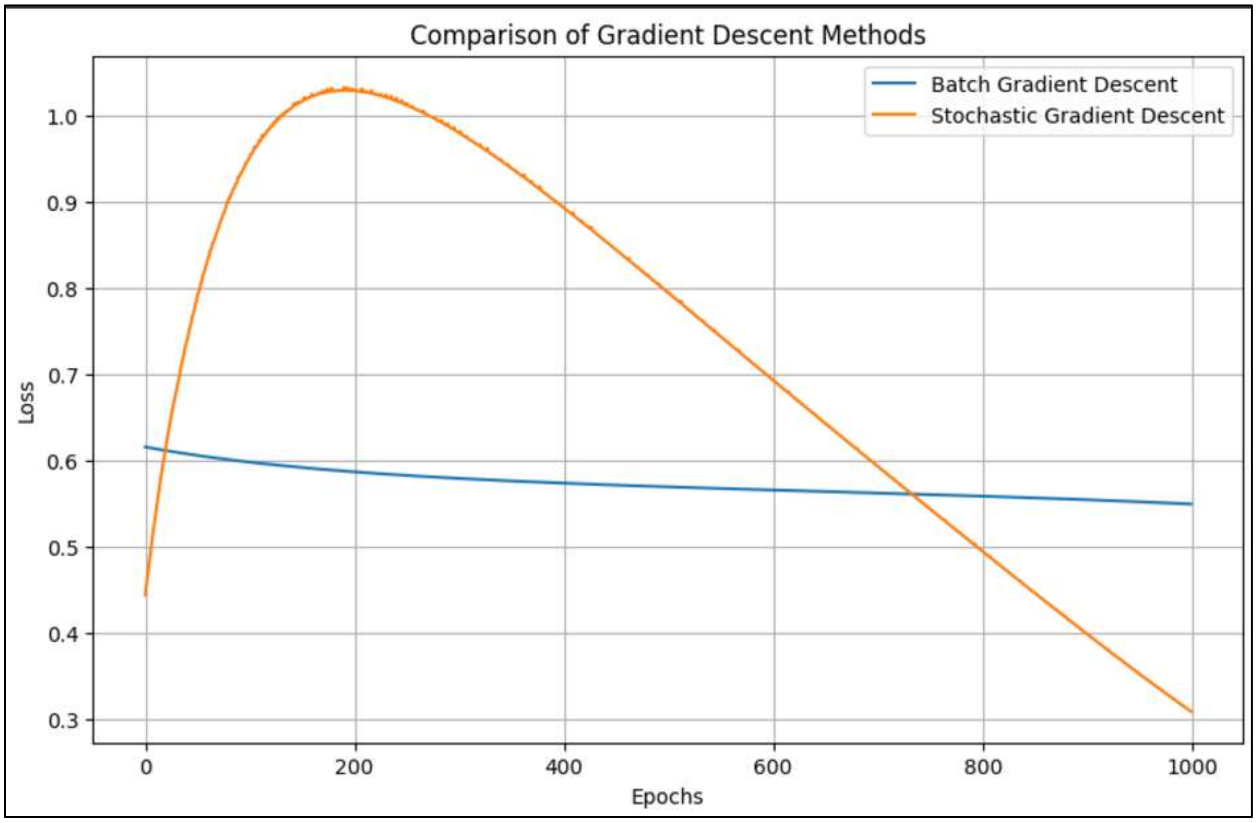


```
            loss.backward()
            optimizer.step()

        losses.append(loss.item())

    return losses

# Compare both methods
bgd_losses = batch_gradient_descent()
sgd_losses = stochastic_gradient_descent()

# Plot the loss curves for both methods
plt.figure(figsize=(10, 6))
plt.plot(bgd_losses, label='Batch Gradient Descent')
plt.plot(sgd_losses, label='Stochastic Gradient Descent')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Comparison of Gradient Descent Methods')
plt.legend()
plt.grid(True)
plt.show()
```

1. **Batch Gradient Descent (BGD):** Updates the model parameters using the gradient calculated from the entire training dataset.
2. **Stochastic Gradient Descent (SGD):** Updates the model parameters using the gradient calculated from a single randomly selected training example (or a small batch).

Interpretation

- **Convergence:** The x-axis represents the number of training epochs, and the y-axis indicates the loss value. A lower loss implies better model performance.
- **Loss Curves:** Both curves show a general downward trend, suggesting that the model is learning and improving over time.
- **Comparison:**
    - **BGD:** The BGD curve converges smoothly and steadily, but it can be computationally expensive for large datasets.
    - **SGD:** The SGD curve is more noisy and fluctuates more, but it can converge faster than BGD, especially in the early stages of training.



Deep Learning

Lab 2

Nayyab Malik
(BSAI-127)

Assigned By

Mam Iqra Nasem

Faculty of Engineering & CS

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

Submission Date: 31th October, 2024

Loss Function

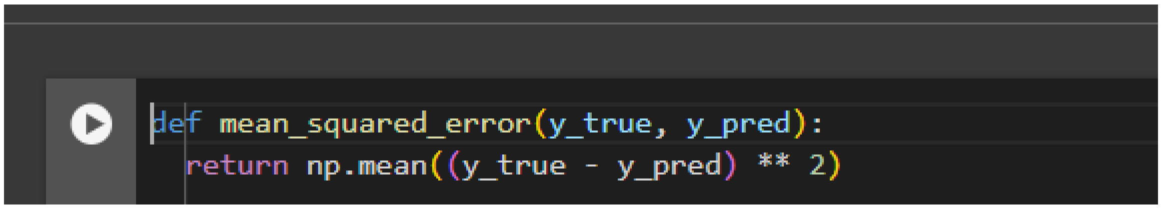
A loss function is a metric used to evaluate how well a model's predictions match the actual data. In supervised learning, the loss function calculates the "error" between the predicted and actual values. During training, the model uses this feedback to adjust its parameters (weights and biases) in order to reduce the loss, thereby improving accuracy. Lower loss indicates a better fit of the model to the data.

1. Mean Squared Error (MSE):

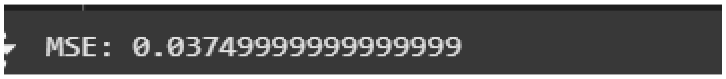
Mean Squared Error is a common loss function for regression tasks. It calculates the average of the squares of the differences between predicted values and actual values. The squaring aspect makes it sensitive to large errors (outliers), as larger errors have a greater impact due to the squaring operation.

Properties of MSE*:*

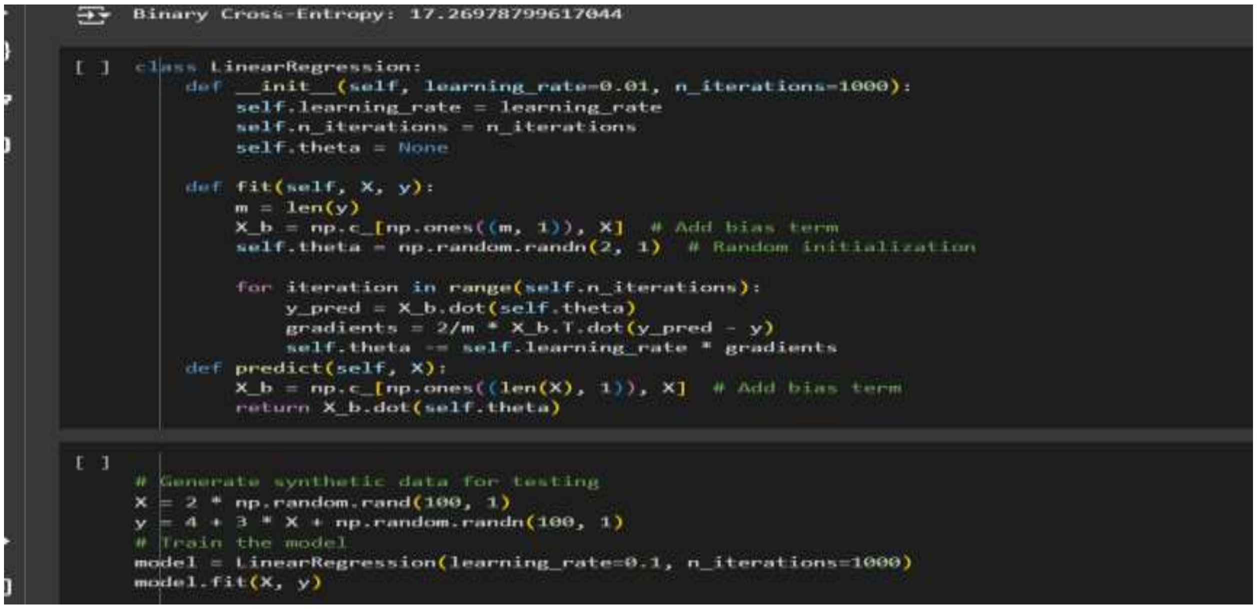
- **Outlier Sensitivity**: Since errors are squared, MSE penalizes larger errors more heavily.
- **Smooth Gradient**: The squaring provides a smooth gradient, making it easier for optimization algorithms to minimize.



```python
def mean_squared_error(y_true, y_pred):
  return np.mean((y_true - y_pred) ** 2)
```



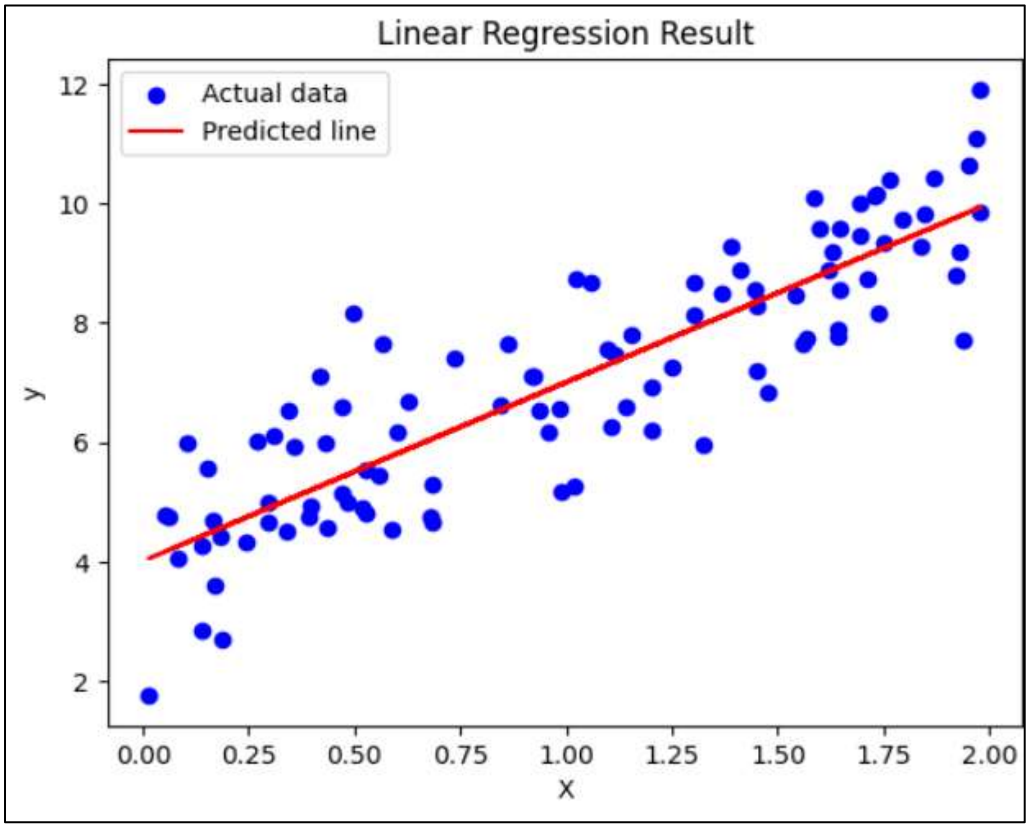
```
MSE: 0.037499999999999999
```

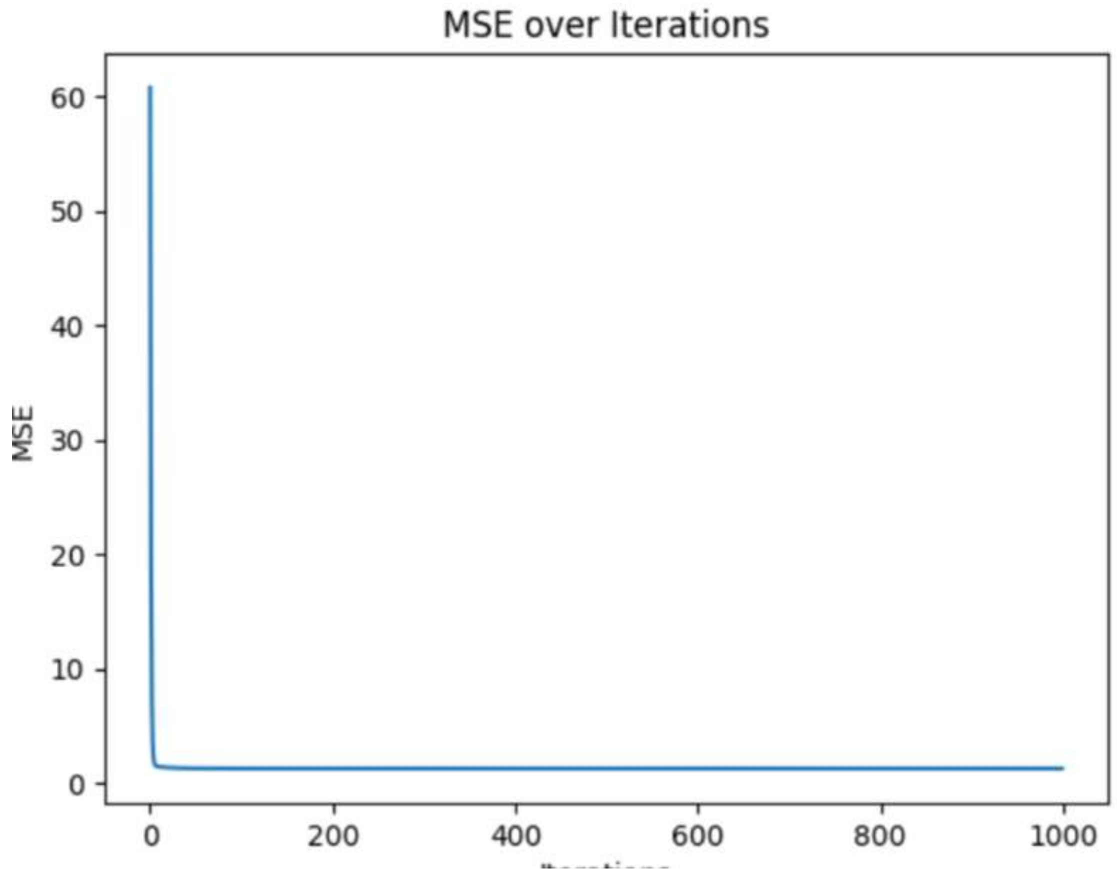


```
Binary Cross-Entropy: 17.26978799617044
```

```python
class LinearRegression:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.theta = None

    def fit(self, X, y):
        m = len(y)
        X_b = np.c_[np.ones((m, 1)), X]  # Add bias term
        self.theta = np.random.randn(2, 1)  # Random initialization

        for iteration in range(self.n_iterations):
            y_pred = X_b.dot(self.theta)
            gradients = 2/m * X_b.T.dot(y_pred - y)
            self.theta -= self.learning_rate * gradients
    def predict(self, X):
        X_b = np.c_[np.ones((len(X), 1)), X]  # Add bias term
        return X_b.dot(self.theta)
```

```python
# Generate synthetic data for testing
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
# Train the model
model = LinearRegression(learning_rate=0.1, n_iterations=1000)
model.fit(X, y)
```


Linear Regression Result
Actual data
Predicted line
y
X
12
10
8
6
4
2
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

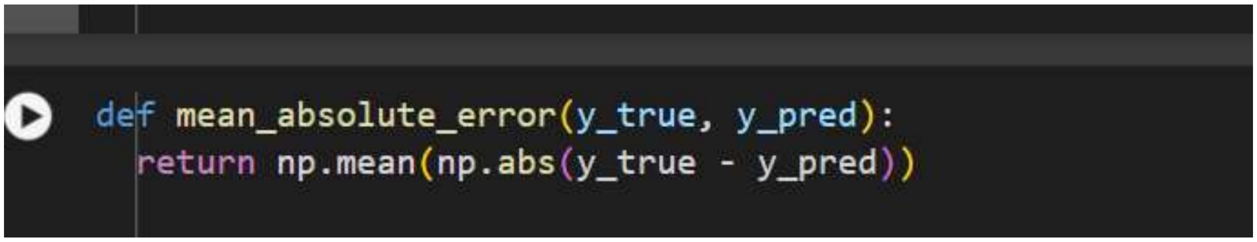

MSE over Iterations
MSE
60
50
40
30
20
10
0
0
200
400
600
800
1000

2. <u>Mean Absolute Error</u> (MAE):

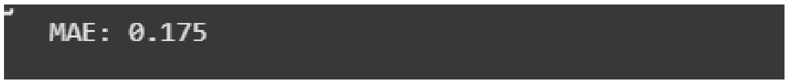
Mean Absolute Error is another common loss function for regression. It calculates the average of the absolute differences between the predicted and actual values. Unlike MSE, MAE does not square the differences, which makes it less sensitive to outliers.

Properties of MAE:

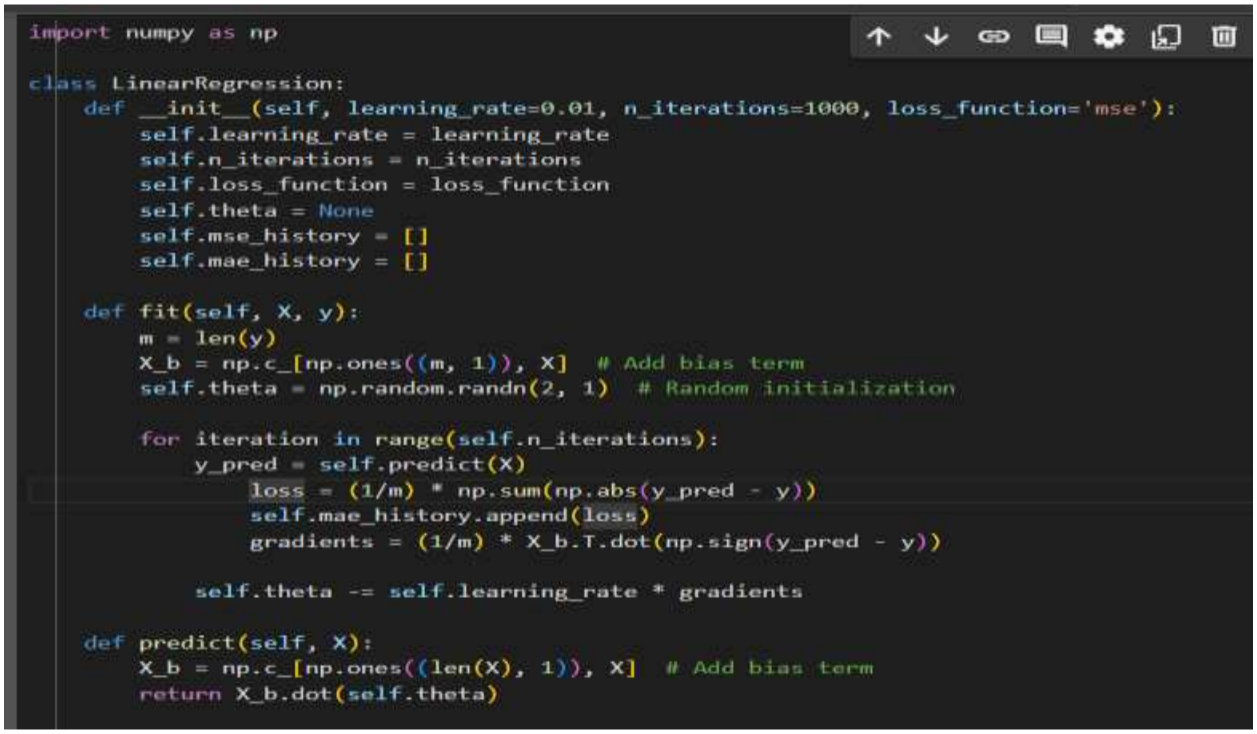
- **Less Sensitive to Outliers**: Errors are treated equally, whether large or small, which helps in minimizing the effect of outliers.
- **Non-Smooth Gradient**: The absolute value function is not smooth at zero, so the gradient can be less stable during optimization.



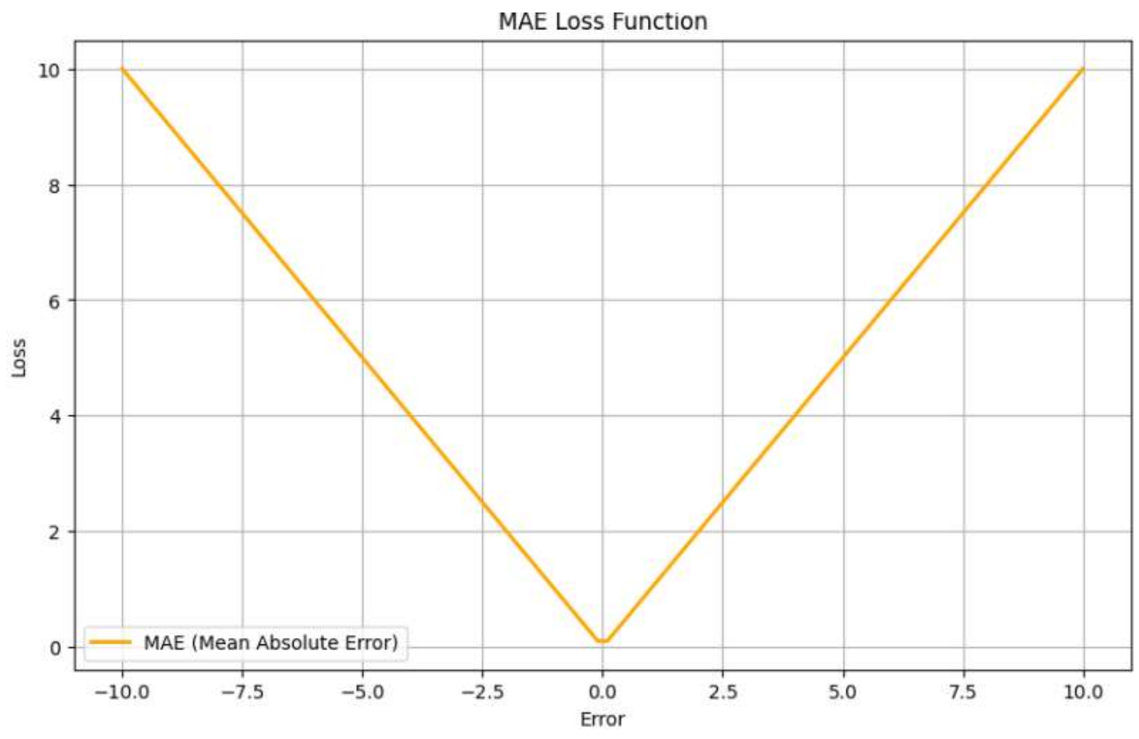
```
def mean_absolute_error(y_true, y_pred):
  return np.mean(np.abs(y_true - y_pred))
```



```
MAE: 0.175
```



```
import numpy as np

class LinearRegression:
    def __init__(self, learning_rate=0.01, n_iterations=1000, loss_function='mse'):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.loss_function = loss_function
        self.theta = None
        self.mse_history = []
        self.mae_history = []

    def fit(self, X, y):
        m = len(y)
        X_b = np.c_[np.ones((m, 1)), X]  # Add bias term
        self.theta = np.random.randn(2, 1)  # Random initialization

        for iteration in range(self.n_iterations):
            y_pred = self.predict(X)
                loss = (1/m) * np.sum(np.abs(y_pred - y))
                self.mae_history.append(loss)
                gradients = (1/m) * X_b.T.dot(np.sign(y_pred - y))

            self.theta -= self.learning_rate * gradients

    def predict(self, X):
        X_b = np.c_[np.ones((len(X), 1)), X]  # Add bias term
        return X_b.dot(self.theta)
```

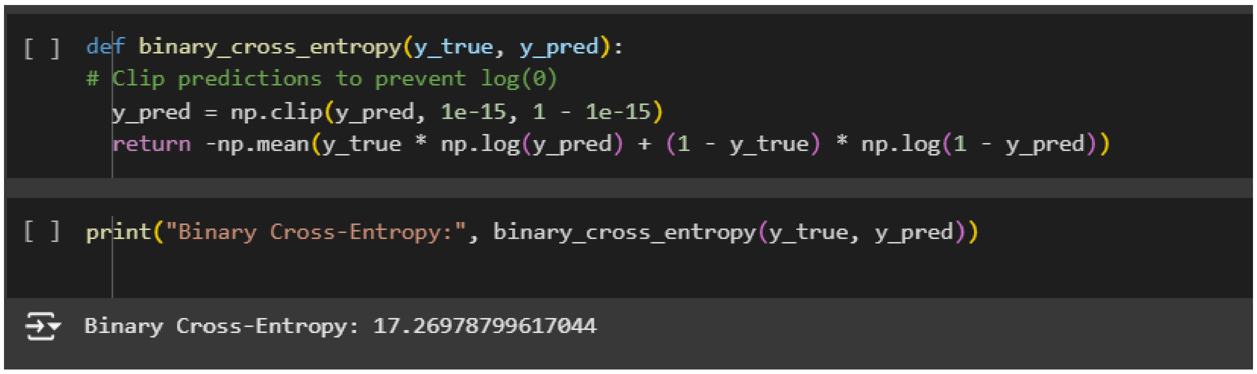


3. Binary Cross-Entropy:

Binary Cross-Entropy is a common loss function for binary classification tasks (where there are only two classes, e.g., 0 and 1). It measures the difference between the predicted probability and the actual label. Cross-entropy loss increases as the predicted probability diverges from the actual label.

Properties of Binary Cross-Entropy:

- **Suitable for Probabilistic Outputs**: It works best with outputs between 0 and 1, representing probabilities.
- **Sensitive to Large Errors**: Large differences between predicted probabilities and actual labels will result in high loss values.



```
[ ] def binary_cross_entropy(y_true, y_pred):
    # Clip predictions to prevent log(0)
      y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)
      return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))
```

```
[ ] print("Binary Cross-Entropy:", binary_cross_entropy(y_true, y_pred))
```

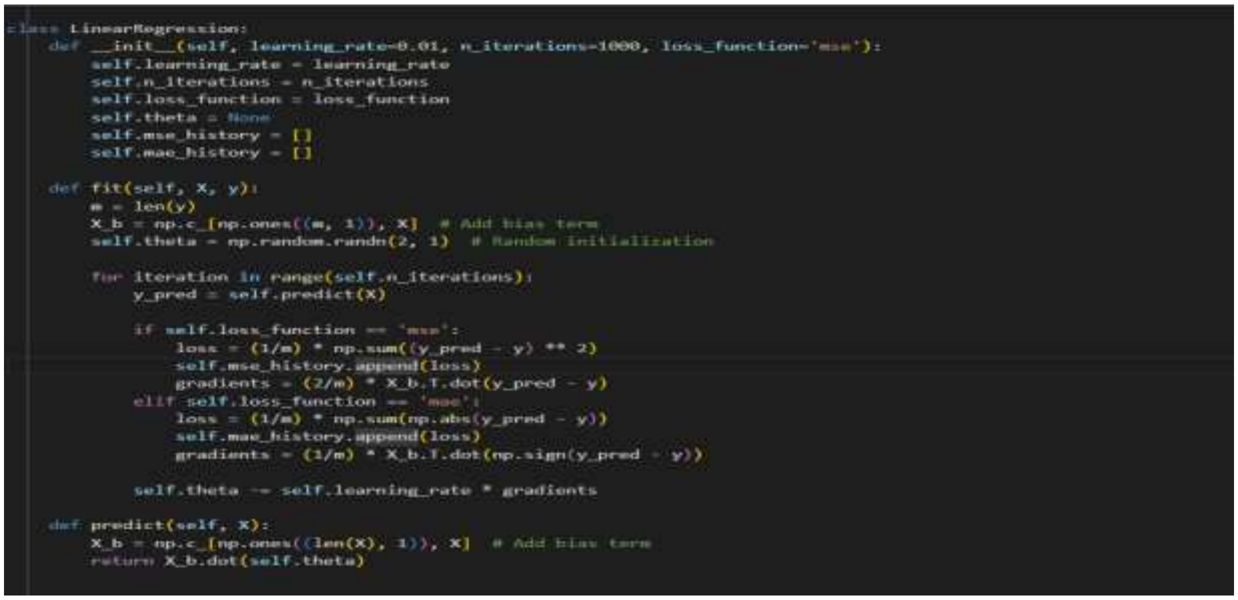
```
Binary Cross-Entropy: 17.26978799617044
```

Comparison between MSE and MAE:

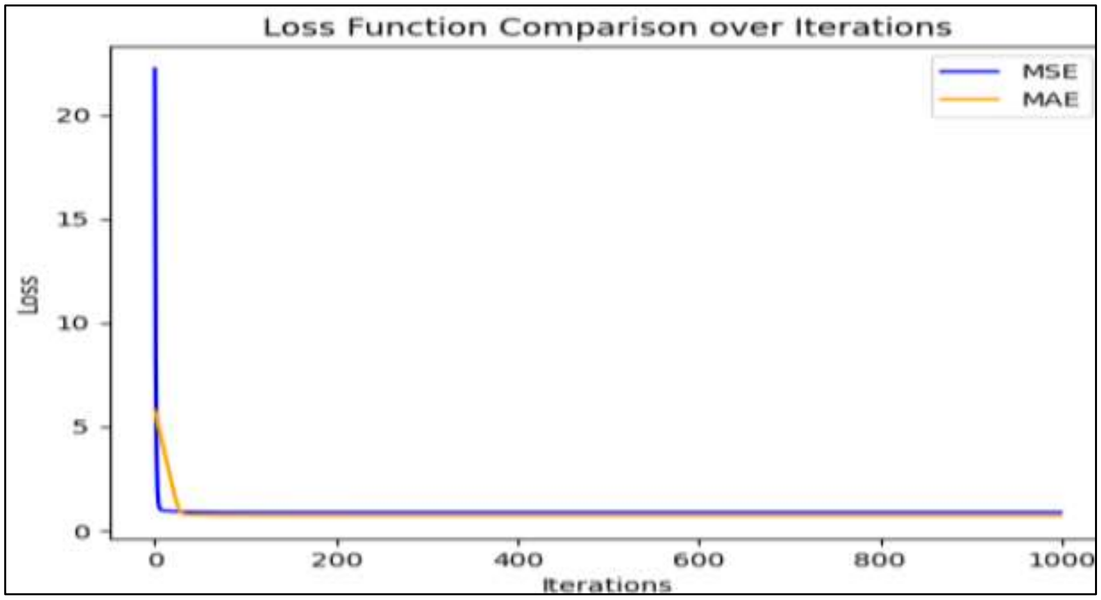
To understand the differences between MSE and MAE, let's consider their behavior on a graph.

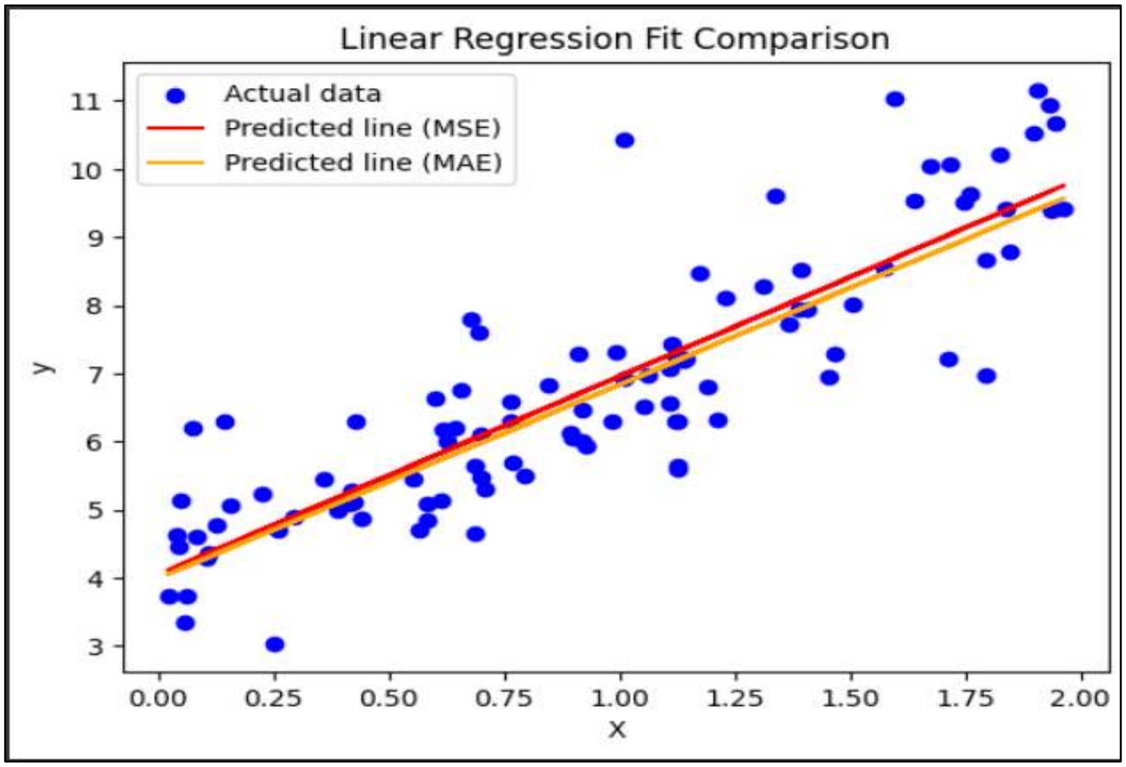
Graphical Comparison:

1. **MSE Graph**: The MSE graph shows a quadratic growth as the error increases. This is because errors are squared, making MSE more sensitive to large errors.
2. **MAE Graph**: The MAE graph grows linearly with the error, making it less sensitive to larger errors. This leads to a more balanced approach when handling data with potential outliers.



```python
class LinearRegression:
    def __init__(self, learning_rate=0.01, n_iterations=1000, loss_function='mse'):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.loss_function = loss_function
        self.theta = None
        self.mse_history = []
        self.mae_history = []

    def fit(self, X, y):
        m = len(y)
        X_b = np.c_[np.ones((m, 1)), X]  # Add bias term
        self.theta = np.random.randn(2, 1)  # Random initialization

        for iteration in range(self.n_iterations):
            y_pred = self.predict(X)

            if self.loss_function == 'mse':
                loss = (1/m) * np.sum((y_pred - y) ** 2)
                self.mse_history.append(loss)
                gradients = (2/m) * X_b.T.dot(y_pred - y)
            elif self.loss_function == 'mae':
                loss = (1/m) * np.sum(np.abs(y_pred - y))
                self.mae_history.append(loss)
                gradients = (1/m) * X_b.T.dot(np.sign(y_pred - y))

            self.theta -= self.learning_rate * gradients

    def predict(self, X):
        X_b = np.c_[np.ones((len(X), 1)), X]  # Add bias term
        return X_b.dot(self.theta)
```




Linear Regression Fit Comparison
Actual data
Predicted line (MSE)
Predicted line (MAE)
y
x
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
3
4
5
6
7
8
9
10
11

Deep Learning

Lab 3

Nayyab Malik
(BSAI-127)

Assigned By

Mam Iqra Nasem

Faculty of Engineering & CS

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

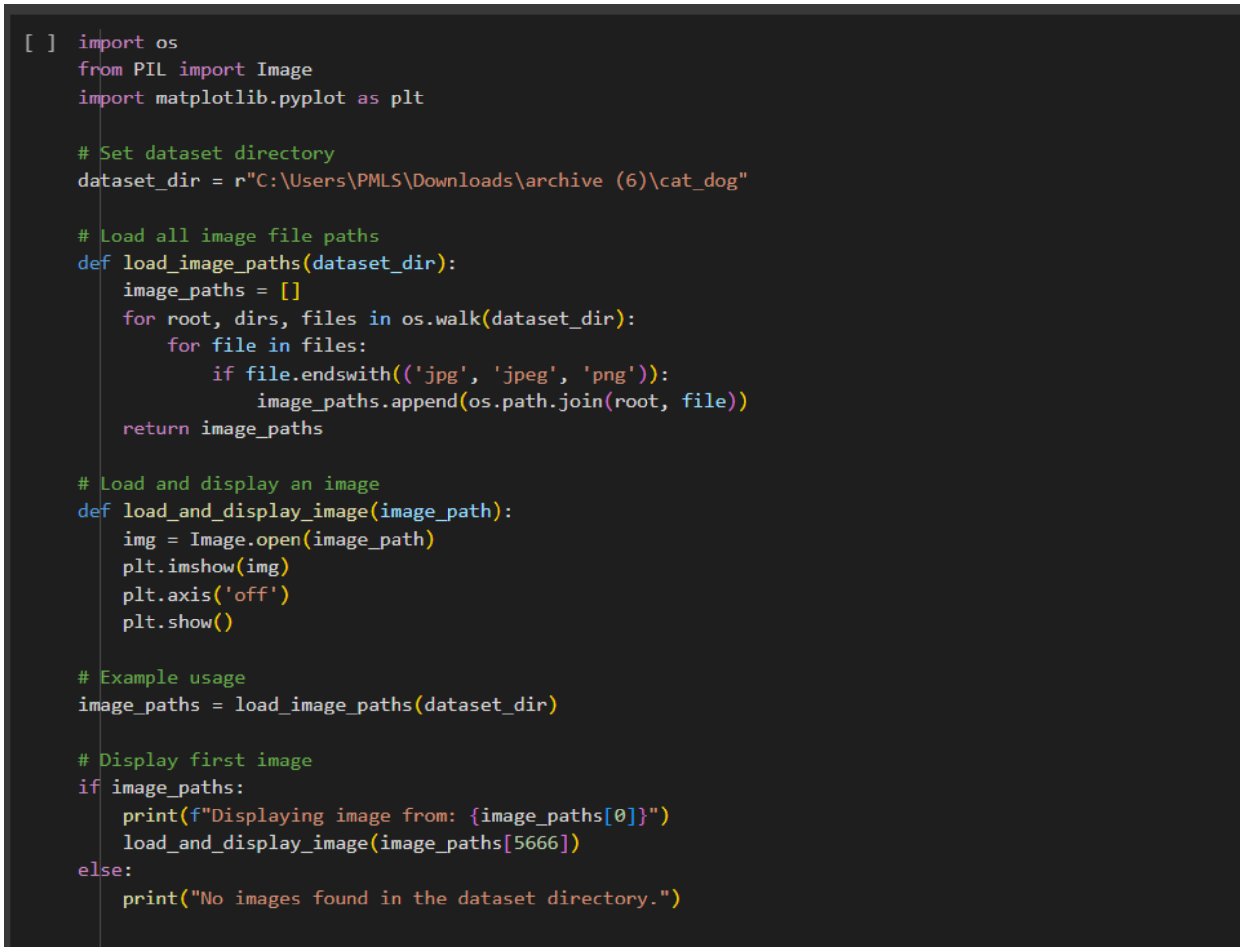
Submission Date: 31th October, 2024

Classification Task using CNN

In deep learning, a classification task involves training a model to categorize input data into predefined classes.

1. Problem Definition and Data Collection:

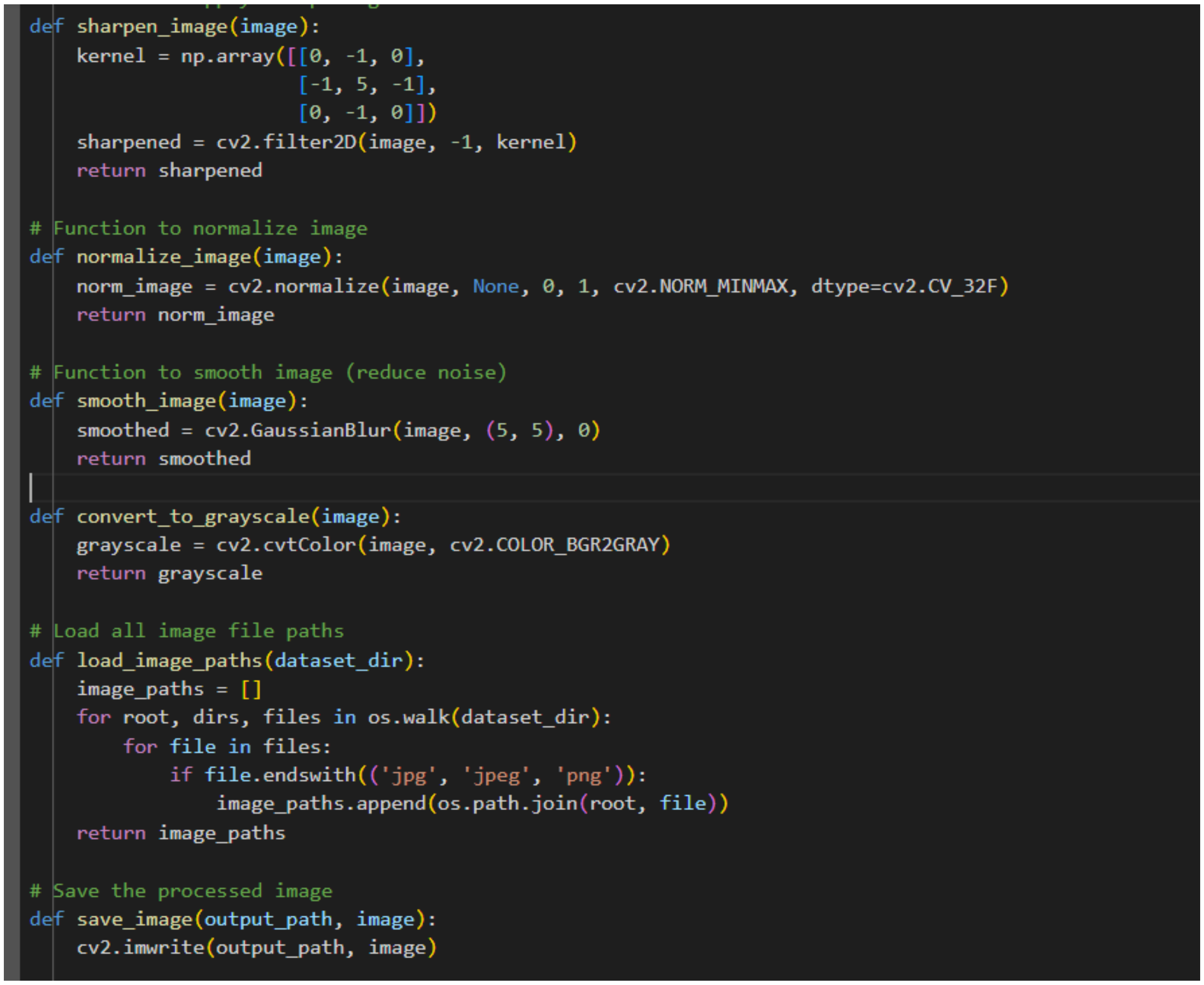
- Define the problem and decide on the categories the model should classify (e.g., dog vs. cat).
- Collect and label data according to these categories (e.g., labeled images for image classification).



```
[ ]  import os
     from PIL import Image
     import matplotlib.pyplot as plt

     # Set dataset directory
     dataset_dir = r"C:\Users\PMLS\Downloads\archive (6)\cat_dog"

     # Load all image file paths
     def load_image_paths(dataset_dir):
         image_paths = []
         for root, dirs, files in os.walk(dataset_dir):
             for file in files:
                 if file.endswith(('jpg', 'jpeg', 'png')):
                     image_paths.append(os.path.join(root, file))
         return image_paths

     # Load and display an image
     def load_and_display_image(image_path):
         img = Image.open(image_path)
         plt.imshow(img)
         plt.axis('off')
         plt.show()

     # Example usage
     image_paths = load_image_paths(dataset_dir)

     # Display first image
     if image_paths:
         print(f"Displaying image from: {image_paths[0]}")
         load_and_display_image(image_paths[5666])
     else:
         print("No images found in the dataset directory.")
```

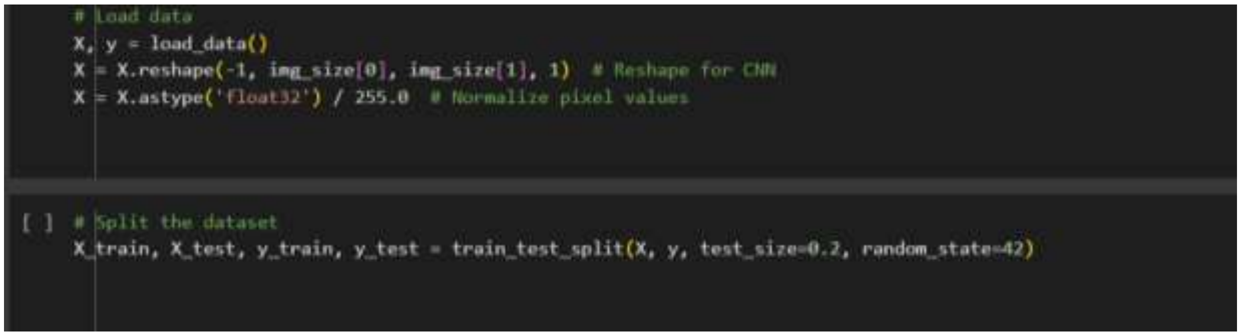
2. Data Preprocessing:

- **Data Cleaning**: Remove or fix corrupted samples by applying sharpening, background removal etc.
- **Normalization/Standardization**: Scale input features to a consistent range (e.g., 0–1 or -1–1).



```python
def sharpen_image(image):
    kernel = np.array([[0, -1, 0],
                       [-1, 5, -1],
                       [0, -1, 0]])
    sharpened = cv2.filter2D(image, -1, kernel)
    return sharpened

# Function to normalize image
def normalize_image(image):
    norm_image = cv2.normalize(image, None, 0, 1, cv2.NORM_MINMAX, dtype=cv2.CV_32F)
    return norm_image

# Function to smooth image (reduce noise)
def smooth_image(image):
    smoothed = cv2.GaussianBlur(image, (5, 5), 0)
    return smoothed

def convert_to_grayscale(image):
    grayscale = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    return grayscale

# Load all image file paths
def load_image_paths(dataset_dir):
    image_paths = []
    for root, dirs, files in os.walk(dataset_dir):
        for file in files:
            if file.endswith(('jpg', 'jpeg', 'png')):
                image_paths.append(os.path.join(root, file))
    return image_paths

# Save the processed image
def save_image(output_path, image):
    cv2.imwrite(output_path, image)
```

3. **<u>Train Test Split</u>:**
   Divide the dataset into training, validation, and test sets.



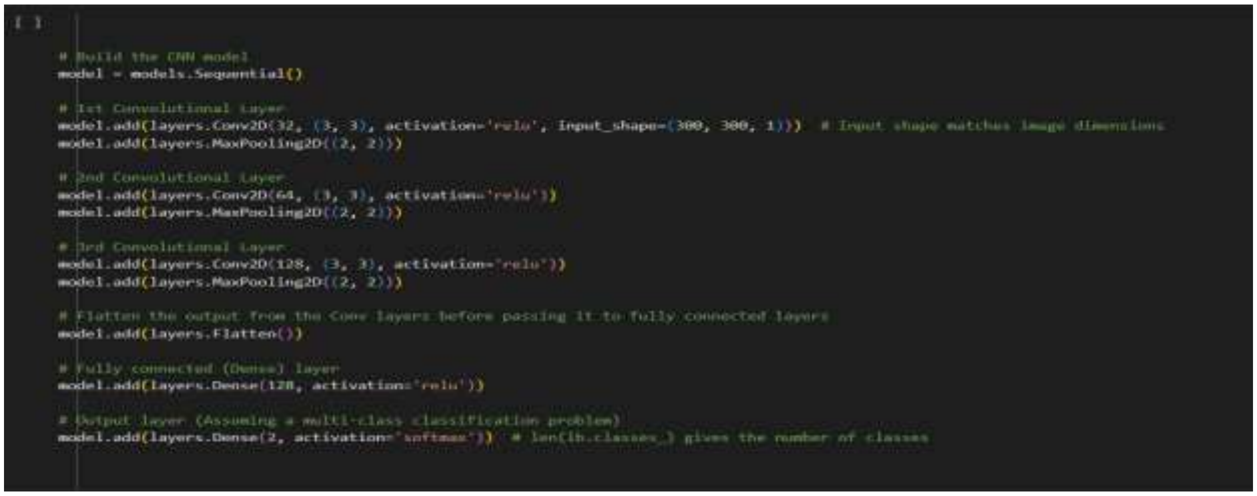
```python
# Load data
X, y = load_data()
X = X.reshape(-1, img_size[0], img_size[1], 1)  # Reshape for CNN
X = X.astype('float32') / 255.0  # Normalize pixel values
```

```python
[ ] # Split the dataset
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

4. **<u>CNN Model</u>:**

   This is where we build a simple CNN model. We'll use several convolutional layers followed by pooling layers, then flatten the output and use dense (fully connected) layers to classify the images.

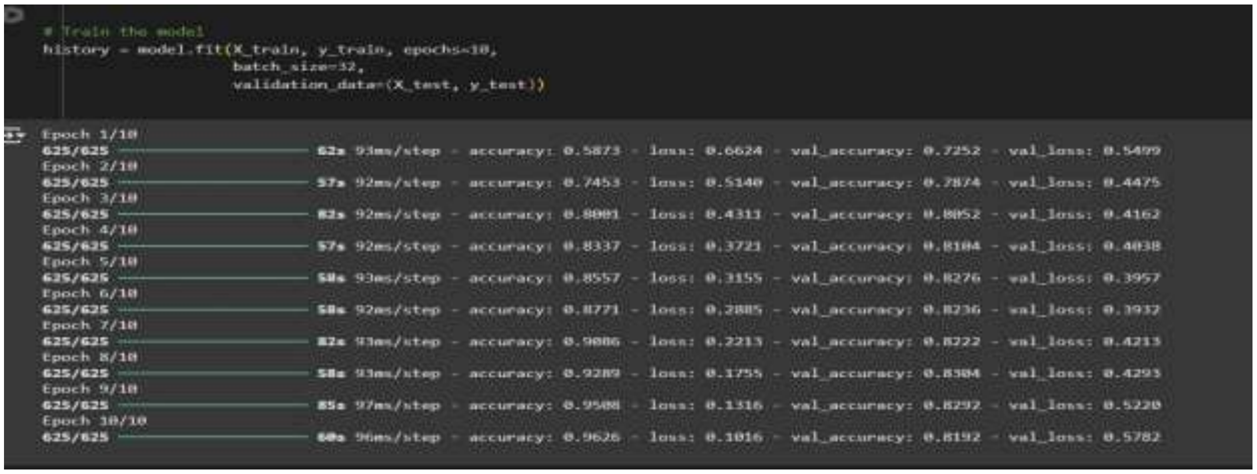
- **Convolutional Layers (Conv2D)**: These layers apply filters to the input image to extract important features such as edges, textures, and shapes.
- **Pooling Layers (MaxPooling2D)**: These reduce the spatial dimensions of the feature maps, lowering computational load and making the network invariant to small translations.
- **Flatten Layer**: This layer reshapes the 2D matrix from the previous layer into a 1D vector, preparing it for the dense layers.
- **Dense Layers**: These fully connected layers analyze the features extracted by convolutional layers and produce class scores.
- **Output Layer**: The final dense layer has 10 neurons (for 10 classes) and produces the raw class scores.



```
# Build the CNN model
model = models.Sequential()

# 1st Convolutional Layer
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(300, 300, 1)))  # Input shape matches image dimensions
model.add(layers.MaxPooling2D((2, 2)))

# 2nd Convolutional Layer
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

# 3rd Convolutional Layer
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

# Flatten the output from the Conv layers before passing it to fully connected layers
model.add(layers.Flatten())

# Fully connected (Dense) layer
model.add(layers.Dense(128, activation='relu'))

# Output layer (Assuming a multi-class classification problem)
model.add(layers.Dense(2, activation='softmax'))  # len(lb.classes_) gives the number of classes
```

5. **<u>Model Training</u>:**

- **Compile the Model**: Specify the optimizer, loss function, and metrics (e.g., accuracy).
- **Train the Model**: Use a batch size that balances training speed and stability (e.g., 32 or 64) and train for multiple epochs.
- **Validation**: Monitor the validation loss and accuracy to prevent overfitting. Use techniques like early stopping to halt training when validation performance stops improving.

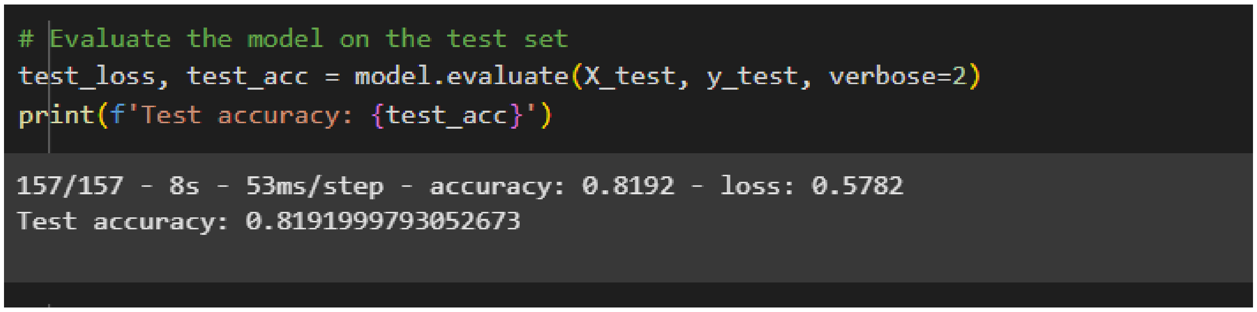


```
# Train the model
history = model.fit(X_train, y_train, epochs=10,
                    batch_size=32,
                    validation_data=(X_test, y_test))
```

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 62s 93ms/step - accuracy: 0.5873 - loss: 0.6624 - val_accuracy: 0.7252 - val_loss: 0.5499
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 57s 92ms/step - accuracy: 0.7453 - loss: 0.5140 - val_accuracy: 0.7874 - val_loss: 0.4475
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 82s 92ms/step - accuracy: 0.8001 - loss: 0.4311 - val_accuracy: 0.8052 - val_loss: 0.4162
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 57s 92ms/step - accuracy: 0.8337 - loss: 0.3721 - val_accuracy: 0.8104 - val_loss: 0.4038
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 58s 93ms/step - accuracy: 0.8557 - loss: 0.3155 - val_accuracy: 0.8276 - val_loss: 0.3957
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 58s 92ms/step - accuracy: 0.8771 - loss: 0.2885 - val_accuracy: 0.8236 - val_loss: 0.3932
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 82s 93ms/step - accuracy: 0.9086 - loss: 0.2213 - val_accuracy: 0.8222 - val_loss: 0.4213
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 58s 93ms/step - accuracy: 0.9289 - loss: 0.1755 - val_accuracy: 0.8304 - val_loss: 0.4293
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 85s 97ms/step - accuracy: 0.9508 - loss: 0.1316 - val_accuracy: 0.8292 - val_loss: 0.5220
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 60s 96ms/step - accuracy: 0.9626 - loss: 0.1016 - val_accuracy: 0.8192 - val_loss: 0.5782
```

6. <u>Evaluating the Model</u>:

- Use the test set to evaluate the trained CNN on unseen data.



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

```
157/157 - 8s - 53ms/step - accuracy: 0.8192 - loss: 0.5782
Test accuracy: 0.8191999793052673
```

Deep Learning

Lab 4

Nayyab Malik
(BSAI-127)

Assigned By

Mam Iqra Nasem

Faculty of Engineering & CS

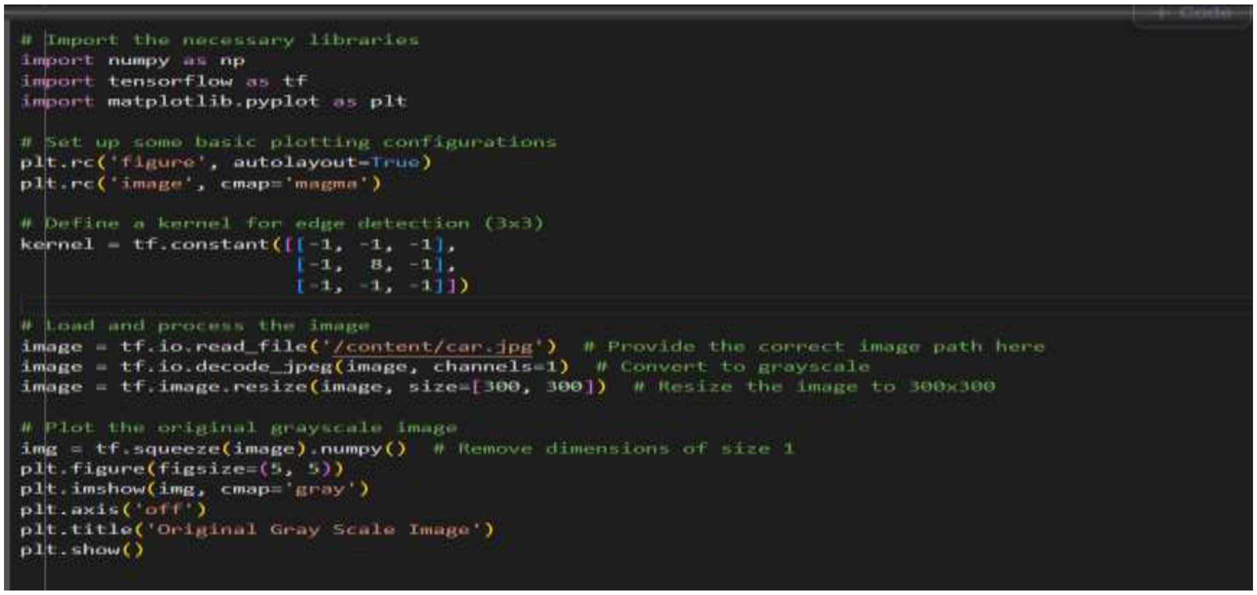
DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

Submission Date: 31th October, 2024

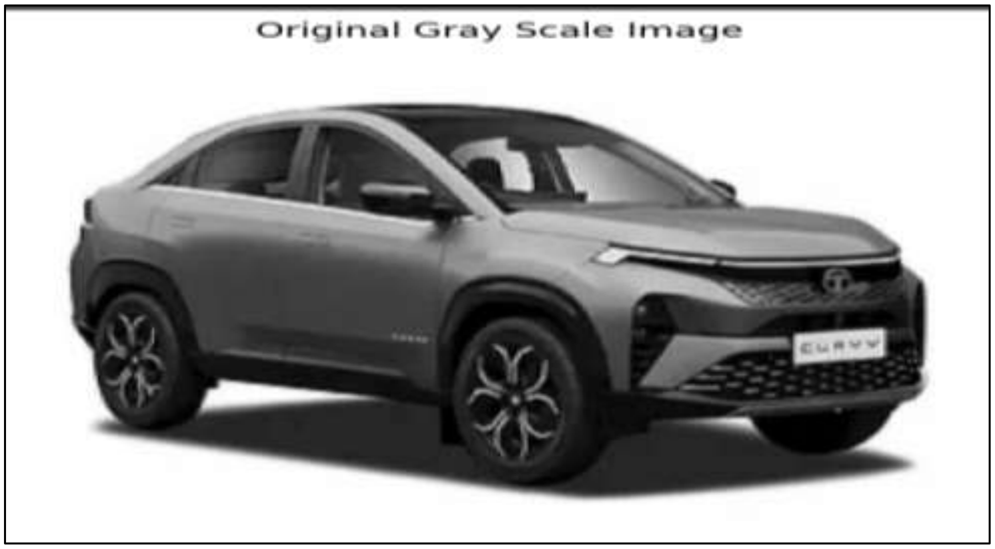
CNN Pattern

2. **Grayscale Image Loading and Preprocessing:**
   - **Loading Image**: The code reads an image file from a specified path and decodes it to grayscale.
   - **Resizing Image**: The grayscale image is resized to 300x300 pixels, allowing for uniform processing.
   - **Plotting Original Image**: The plt.imshow function is used to visualize the grayscale image, showing what the original image looks like before applying any filters.



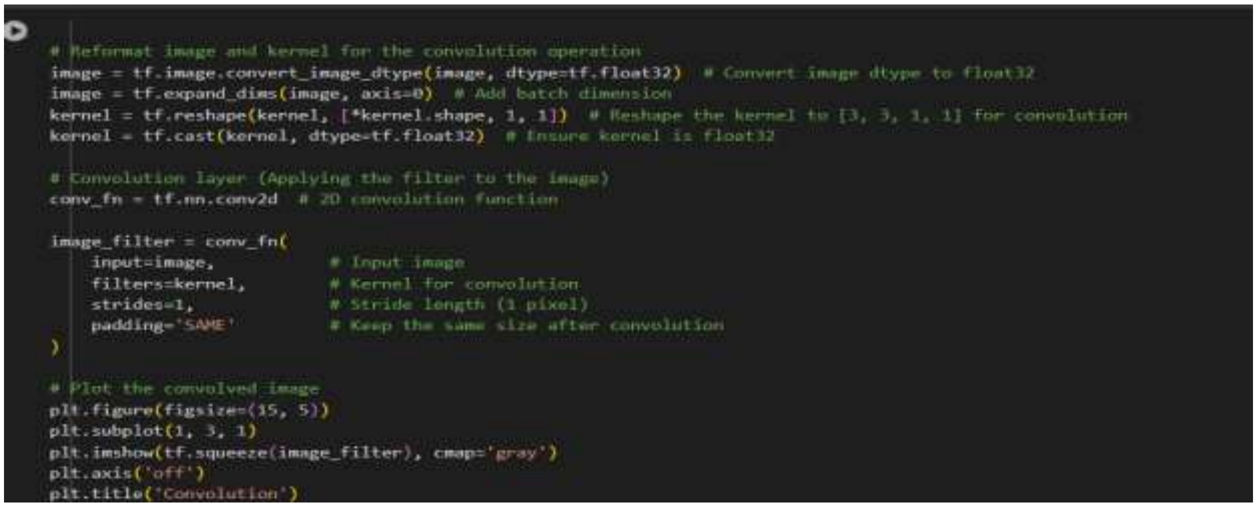
```python
# Import the necessary libraries
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# Set up some basic plotting configurations
plt.rc('figure', autolayout=True)
plt.rc('image', cmap='magma')

# Define a kernel for edge detection (3x3)
kernel = tf.constant([[-1, -1, -1],
                      [-1,  8, -1],
                      [-1, -1, -1]])

# Load and process the image
image = tf.io.read_file('/content/car.jpg')  # Provide the correct image path here
image = tf.io.decode_jpeg(image, channels=1)  # Convert to grayscale
image = tf.image.resize(image, size=[300, 300])  # Resize the image to 300x300

# Plot the original grayscale image
img = tf.squeeze(image).numpy()  # Remove dimensions of size 1
plt.figure(figsize=(5, 5))
plt.imshow(img, cmap='gray')
plt.axis('off')
plt.title('Original Gray Scale Image')
plt.show()
```

This preprocessing step ensures that the image is suitable for convolutional operations by having a uniform size and single color channel (grayscale).

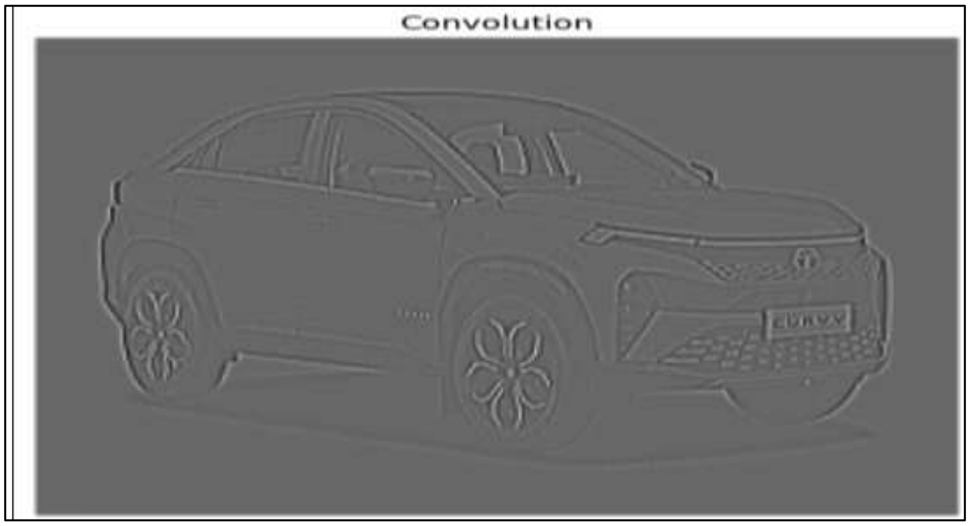


2. <u>Convolution with Edge Detection Filter</u>:

- **Convolution Filter (Edge Detection Kernel)**:
  - A 3x3 edge-detection kernel is defined as kernel. This kernel emphasizes pixel intensity differences, which highlight edges.
  - Each element in this kernel represents the weight for surrounding pixels during convolution.
  - The kernel's configuration (centered positive weight with surrounding negative weights) is designed to detect edges by emphasizing intensity changes between neighboring pixels.
- **Convolution Operation**:
  - The convolution operation (tf.nn.conv2d) is applied to the grayscale image, using this edge-detection kernel.
  - **Result**: This highlights the edges in the image by intensifying areas with rapid pixel value changes.
  - **Plotting**: The convolved image (edges highlighted) is displayed, giving a view of detected edges.

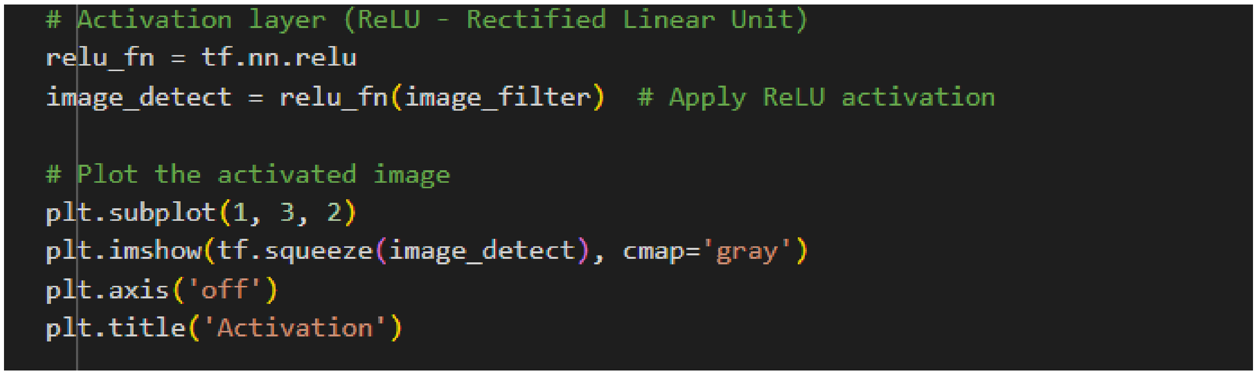


```
# Reformat image and kernel for the convolution operation
image = tf.image.convert_image_dtype(image, dtype=tf.float32)  # Convert image dtype to float32
image = tf.expand_dims(image, axis=0)  # Add batch dimension
kernel = tf.reshape(kernel, [*kernel.shape, 1, 1])  # Reshape the kernel to [3, 3, 1, 1] for convolution
kernel = tf.cast(kernel, dtype=tf.float32)  # Ensure kernel is float32

# Convolution layer (Applying the filter to the image)
conv_fn = tf.nn.conv2d  # 2D convolution function

image_filter = conv_fn(
    input=image,           # Input image
    filters=kernel,        # Kernel for convolution
    strides=1,             # Stride length (1 pixel)
    padding='SAME'         # Keep the same size after convolution
)

# Plot the convolved image
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.imshow(tf.squeeze(image_filter), cmap='gray')
plt.axis('off')
plt.title('Convolution')
```

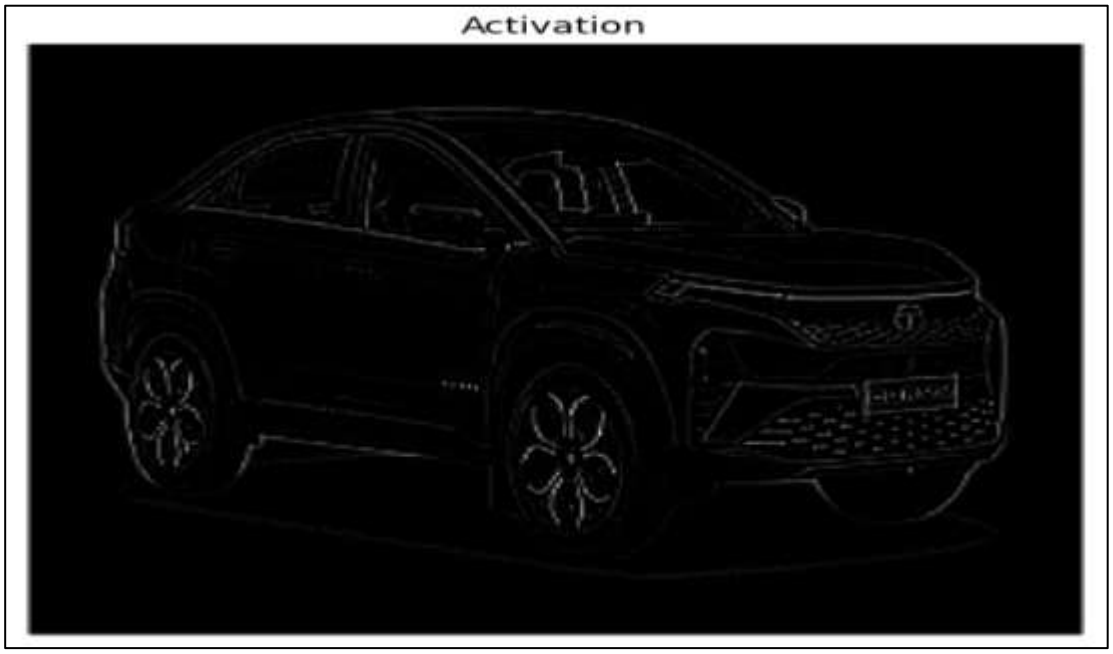


3. <u>**Activation Function (ReLU)**</u>:

- **ReLU (Rectified Linear Unit) Activation**:
    - The ReLU activation function (tf.nn.relu) is applied to the convolved image.
    - **Purpose**: ReLU sets all negative pixel values to zero, ensuring that only the edges (positive values) remain visible.
    - **Effect**: This helps retain the most prominent edges while removing lower-intensity edges.
    - **Plotting**: The activated image is displayed, showing the enhanced edges after ReLU application.



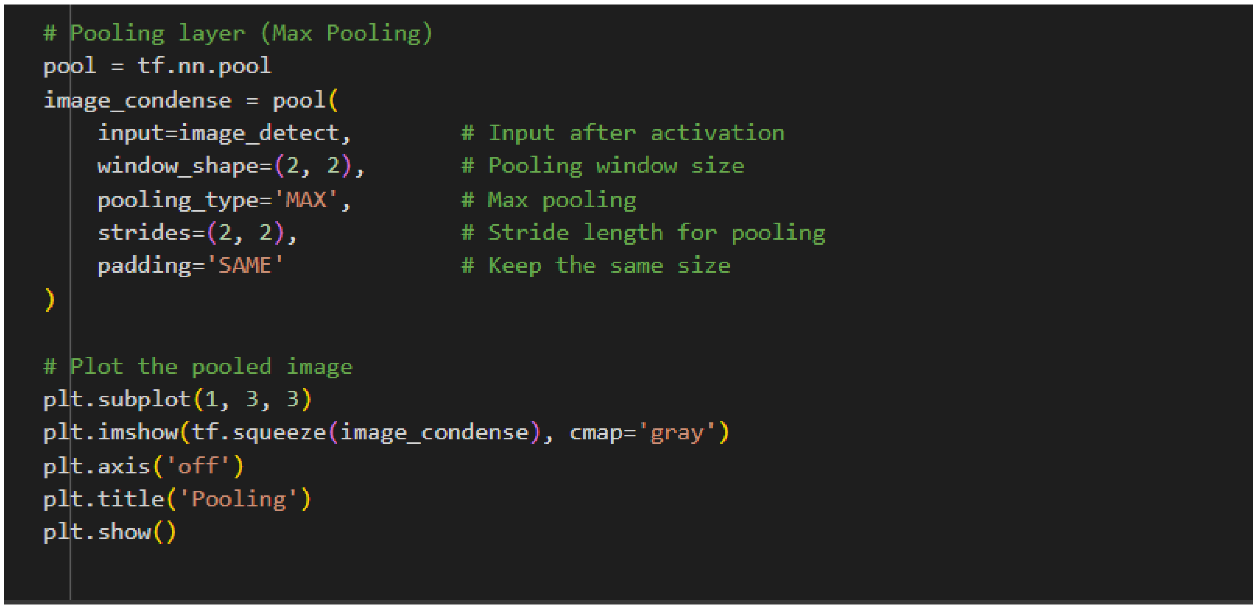
```
# Activation layer (ReLU - Rectified Linear Unit)
relu_fn = tf.nn.relu
image_detect = relu_fn(image_filter)  # Apply ReLU activation

# Plot the activated image
plt.subplot(1, 3, 2)
plt.imshow(tf.squeeze(image_detect), cmap='gray')
plt.axis('off')
plt.title('Activation')
```



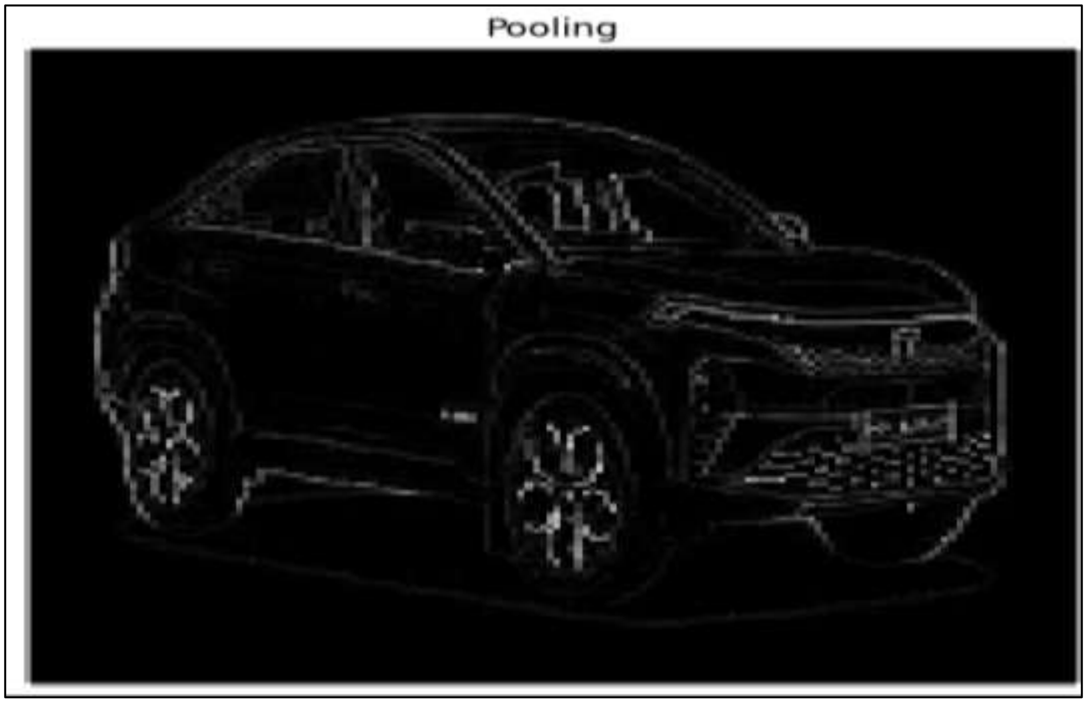
4. <u>**Pooling Layer**</u> **(Max Pooling):**

- **Max Pooling**:
    - Max pooling (tf.nn.pool) is applied to the activated image using a 2x2 window and stride of 2.

- **Purpose**: Max pooling reduces the image size while retaining the most significant features within each 2x2 window, effectively condensing the image.
- **Effect**: This results in a smaller, more condensed representation of the edges, making the image easier for CNNs to process while preserving essential features.
- **Plotting**: The pooled image is displayed, showing a condensed version of the edge-detected image



```
# Pooling layer (Max Pooling)
pool = tf.nn.pool
image_condense = pool(
    input=image_detect,         # Input after activation
    window_shape=(2, 2),        # Pooling window size
    pooling_type='MAX',         # Max pooling
    strides=(2, 2),             # Stride length for pooling
    padding='SAME'              # Keep the same size
)

# Plot the pooled image
plt.subplot(1, 3, 3)
plt.imshow(tf.squeeze(image_condense), cmap='gray')
plt.axis('off')
plt.title('Pooling')
plt.show()
```



Deep Learning

Lab 4+5

Nayyab Malik
(BSAI-127)

Assigned By

Mam Iqra Nasem

Faculty of Engineering & CS

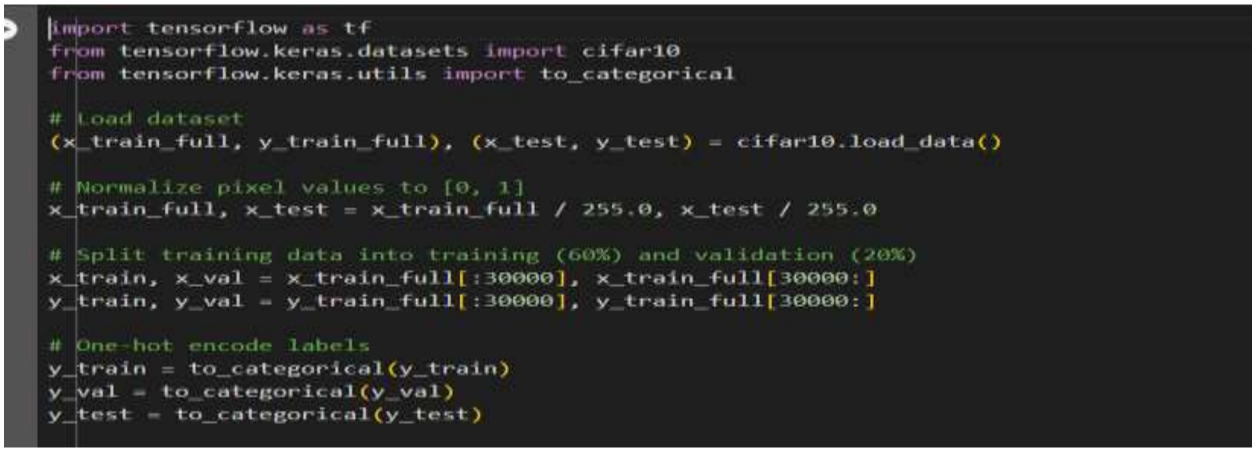
DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

Submission Date: 31th October, 2024

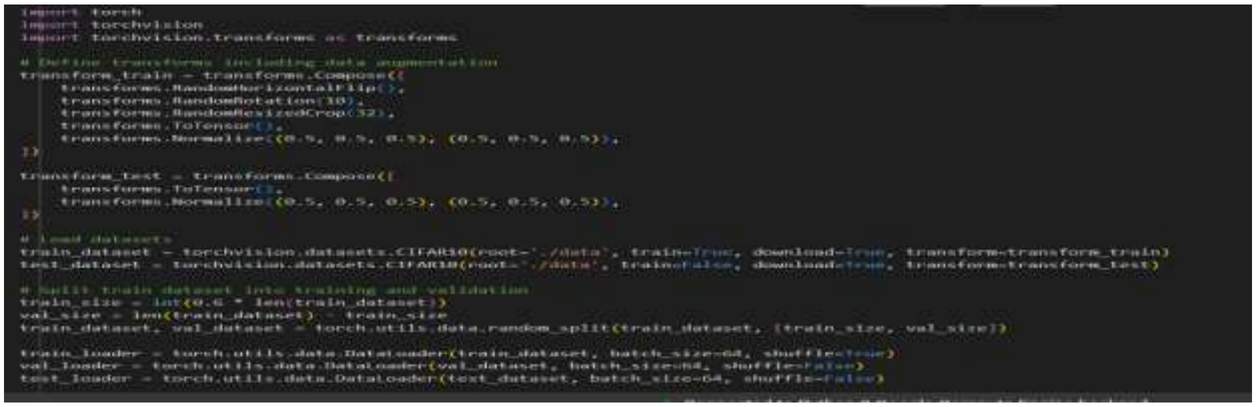
Step 1: Data Preparation

1. **Load the Dataset**:
   - Use TensorFlow or PyTorch to load the CIFAR-10 dataset.
   - Split the dataset into:
     - Training Set (60%)
     - Validation Set (20%)
     - Test Set (20%)



```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical

# Load dataset
(x_train_full, y_train_full), (x_test, y_test) = cifar10.load_data()

# Normalize pixel values to [0, 1]
x_train_full, x_test = x_train_full / 255.0, x_test / 255.0

# Split training data into training (60%) and validation (20%)
x_train, x_val = x_train_full[:30000], x_train_full[30000:]
y_train, y_val = y_train_full[:30000], y_train_full[30000:]

# One-hot encode labels
y_train = to_categorical(y_train)
y_val = to_categorical(y_val)
y_test = to_categorical(y_test)
```

2. **Data Augmentation**:
   - Apply data augmentation techniques to the training set:
     - Random horizontal flip
     - Random rotation
     - Random zoom
     - Random cropping

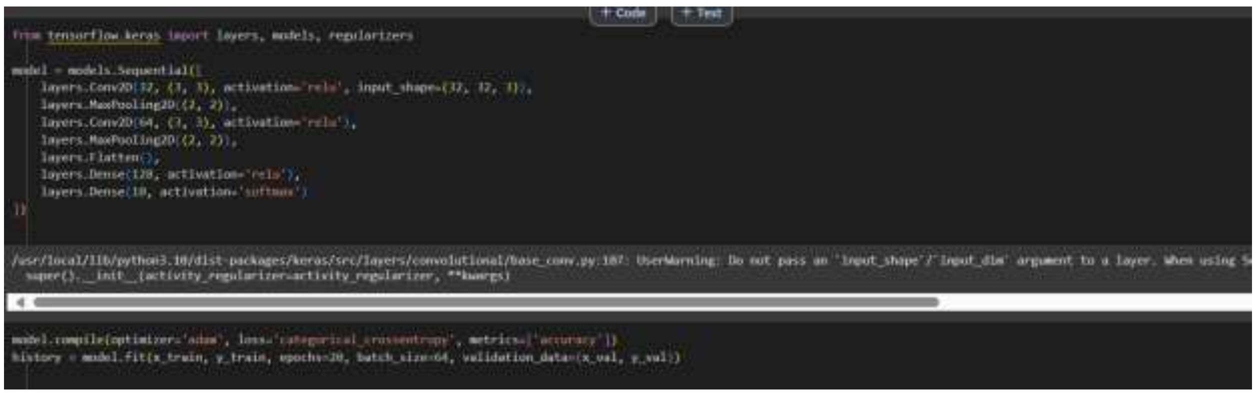


```
import torch
import torchvision
import torchvision.transforms as transforms

# Define transforms including data augmentation
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.RandomResizedCrop(32),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

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

# Load datasets
train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform_test)

# Split train dataset into training and validation
train_size = int(0.6 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_dataset, val_dataset = torch.utils.data.random_split(train_dataset, [train_size, val_size])

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=64, shuffle=False)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64, shuffle=False)
```

Step 2: Build the CNN Model

1. **Model Architecture**:
   - Build a simple CNN with the following layers:
     - Input Layer (e.g., input shape of (32, 32, 3))
     - Convolutional Layer (32 filters, 3x3 kernel, ReLU activation)

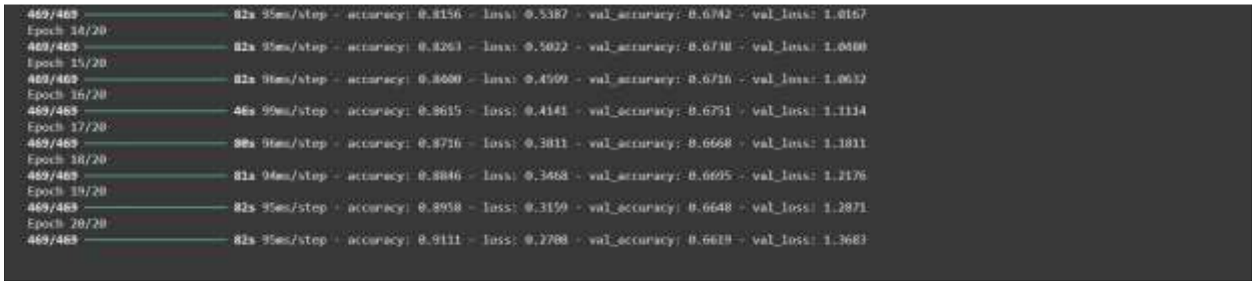
- Max Pooling Layer (2x2)
- Convolutional Layer (64 filters, 3x3 kernel, ReLU activation)
- Max Pooling Layer (2x2)
- Flatten Layer
- Fully Connected (Dense) Layer (128 units, ReLU activation)
- Output Layer (10 units, Softmax activation)



```
from tensorflow.keras import layers, models, regularizers

model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

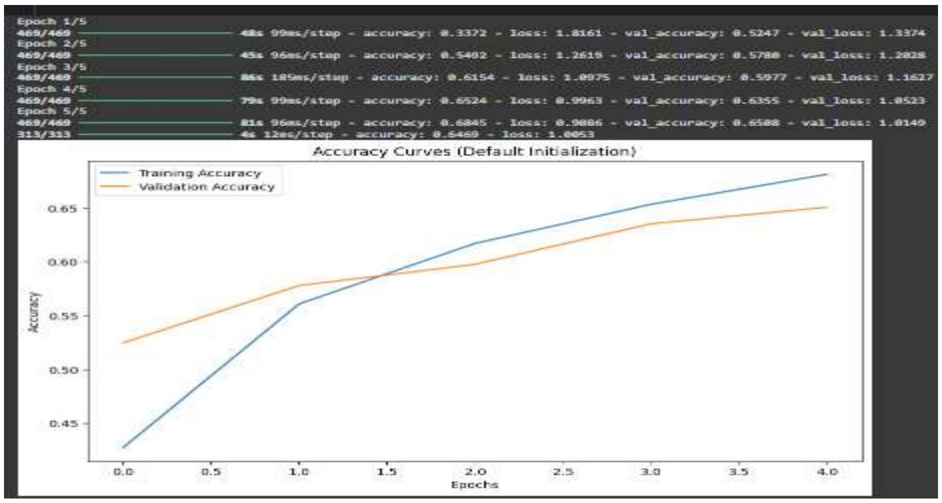
```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using S
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
history = model.fit(x_train, y_train, epochs=20, batch_size=64, validation_data=(x_val, y_val))
```

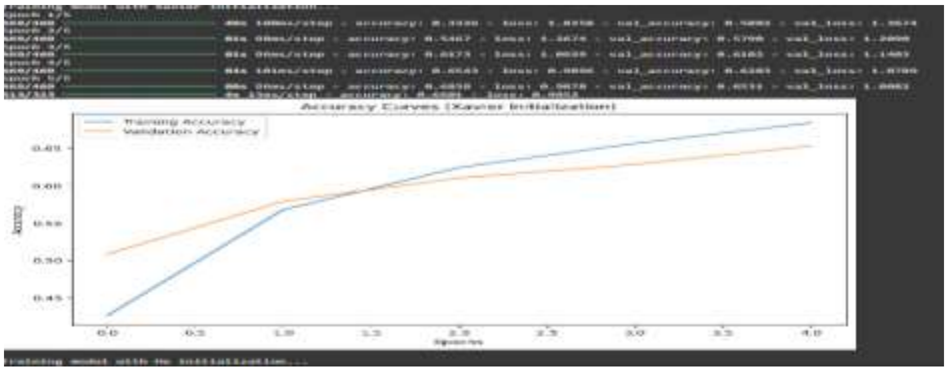


```
469/469 ━━━━━━━━━━━━━━━━━━━━ 82s 95ms/step - accuracy: 0.8156 - loss: 0.5387 - val_accuracy: 0.6742 - val_loss: 1.0167
Epoch 14/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 82s 95ms/step - accuracy: 0.8263 - loss: 0.5022 - val_accuracy: 0.6738 - val_loss: 1.0480
Epoch 15/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 82s 96ms/step - accuracy: 0.8400 - loss: 0.4599 - val_accuracy: 0.6716 - val_loss: 1.0632
Epoch 16/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 46s 99ms/step - accuracy: 0.8615 - loss: 0.4141 - val_accuracy: 0.6751 - val_loss: 1.1134
Epoch 17/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 80s 96ms/step - accuracy: 0.8716 - loss: 0.3811 - val_accuracy: 0.6668 - val_loss: 1.1811
Epoch 18/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 81s 94ms/step - accuracy: 0.8846 - loss: 0.3468 - val_accuracy: 0.6695 - val_loss: 1.2176
Epoch 19/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 82s 95ms/step - accuracy: 0.8958 - loss: 0.3159 - val_accuracy: 0.6648 - val_loss: 1.2871
Epoch 20/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 82s 95ms/step - accuracy: 0.9111 - loss: 0.2708 - val_accuracy: 0.6619 - val_loss: 1.3683
```

2. **Weight Initialization**:
   - Experiment with different weight initialization methods:
     - Default initialization
     - Xavier (Glorot) Initialization
     - He Initialization

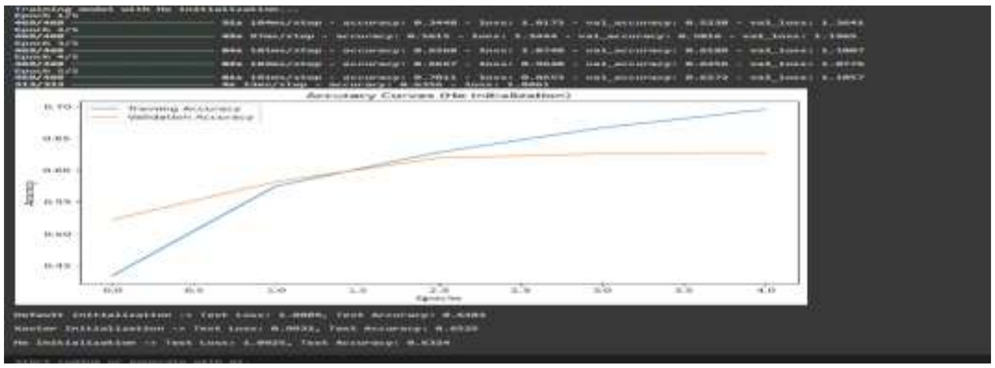


```
Epoch 1/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 48s 99ms/step - accuracy: 0.3372 - loss: 1.8161 - val_accuracy: 0.5247 - val_loss: 1.3374
Epoch 2/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 45s 96ms/step - accuracy: 0.5402 - loss: 1.2619 - val_accuracy: 0.5780 - val_loss: 1.2028
Epoch 3/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 86s 105ms/step - accuracy: 0.6154 - loss: 1.0975 - val_accuracy: 0.5977 - val_loss: 1.1627
Epoch 4/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 79s 99ms/step - accuracy: 0.6524 - loss: 0.9963 - val_accuracy: 0.6355 - val_loss: 1.0523
Epoch 5/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 81s 96ms/step - accuracy: 0.6845 - loss: 0.9086 - val_accuracy: 0.6508 - val_loss: 1.0149
313/313 ━━━━━━━━━━━━━━━━━━━━ 4s 12ms/step - accuracy: 0.6469 - loss: 1.0053
```



```
Training model with Xavier initialization...
Epoch 1/5
400/400 ━━━━━━━━━━━━━━━━━━━━ 40s 100ms/step - accuracy: 0.3930 - loss: 1.8550 - val_accuracy: 0.5083 - val_loss: 1.3574
Epoch 2/5
400/400 ━━━━━━━━━━━━━━━━━━━━ 41s 98ms/step - accuracy: 0.5467 - loss: 1.2674 - val_accuracy: 0.5700 - val_loss: 1.2000
Epoch 3/5
400/400 ━━━━━━━━━━━━━━━━━━━━ 41s 99ms/step - accuracy: 0.6173 - loss: 1.0939 - val_accuracy: 0.6183 - val_loss: 1.1403
Epoch 4/5
400/400 ━━━━━━━━━━━━━━━━━━━━ 40s 101ms/step - accuracy: 0.6543 - loss: 0.9806 - val_accuracy: 0.6385 - val_loss: 1.0700
Epoch 5/5
400/400 ━━━━━━━━━━━━━━━━━━━━ 40s 99ms/step - accuracy: 0.6850 - loss: 0.9078 - val_accuracy: 0.6533 - val_loss: 1.0003
313/313 ━━━━━━━━━━━━━━━━━━━━ 4s 13ms/step - accuracy: 0.6500 - loss: 0.9852
```

```
Training model with He initialization...
```

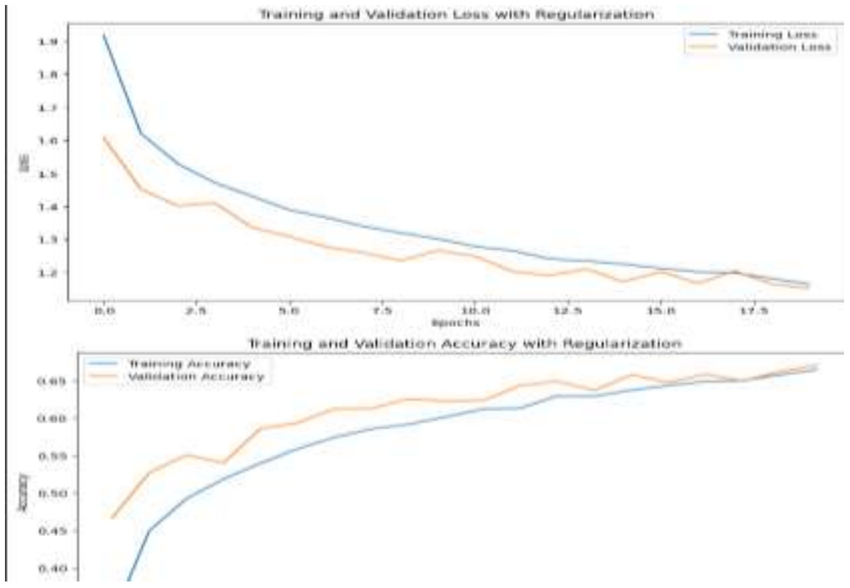


```
Training model with He initialization...
Epoch 1/5
400/400 ━━━━━━━━━━━━━━━━━━━━ 51s 104ms/step - accuracy: 0.3440 - loss: 1.8173 - val_accuracy: 0.5330 - val_loss: 1.3041
Epoch 2/5
400/400 ━━━━━━━━━━━━━━━━━━━━ 40s 97ms/step - accuracy: 0.5615 - loss: 1.2444 - val_accuracy: 0.5816 - val_loss: 1.1909
Epoch 3/5
400/400 ━━━━━━━━━━━━━━━━━━━━ 40s 101ms/step - accuracy: 0.6300 - loss: 1.0740 - val_accuracy: 0.6100 - val_loss: 1.1007
Epoch 4/5
400/400 ━━━━━━━━━━━━━━━━━━━━ 40s 100ms/step - accuracy: 0.6667 - loss: 0.9600 - val_accuracy: 0.6250 - val_loss: 1.0776
Epoch 5/5
400/400 ━━━━━━━━━━━━━━━━━━━━ 40s 101ms/step - accuracy: 0.7011 - loss: 0.8653 - val_accuracy: 0.6272 - val_loss: 1.1057
313/313 ━━━━━━━━━━━━━━━━━━━━ 4s 13ms/step - accuracy: 0.6350 - loss: 1.0003
```

```
Default Initialization -> Test Loss: 1.0000, Test Accuracy: 0.6400
Xavier Initialization -> Test Loss: 0.9932, Test Accuracy: 0.6520
He Initialization -> Test Loss: 1.0025, Test Accuracy: 0.6324
```

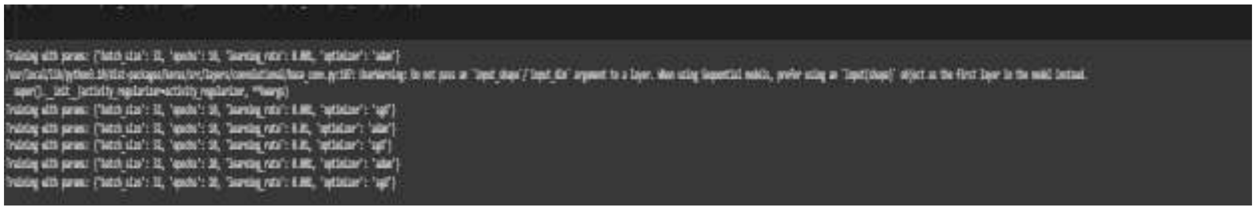
Step 4: Implement Regularization

1. **Apply L2 Regularization**:
   - Add L2 regularization to the convolutional and/or dense layers.
   - Re-train the model and observe the impact on performance.
2. **Introduce Dropout**:
   - Add dropout layers after the dense layers (e.g., dropout rate of 0.5).
   - Re-train the model and compare the results to the previous models.



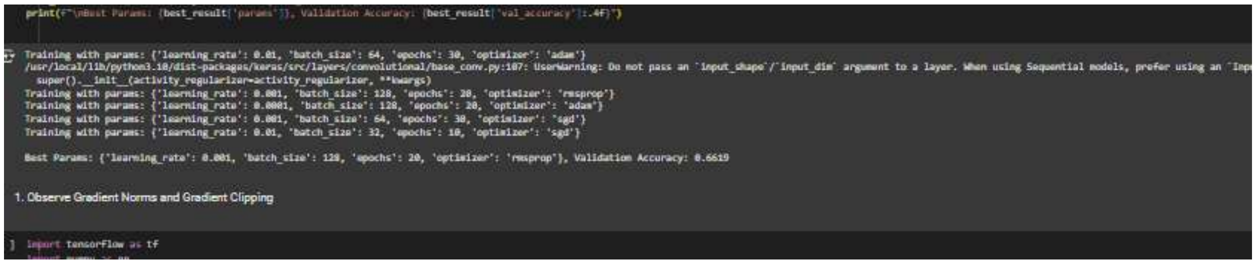
Step 5: Hyperparameter Tuning

1. **Identify Hyperparameters**:
   - Select hyperparameters to tune (e.g., learning rate, batch size, number of epochs).



```
Training with params: {'batch_size': 32, 'epochs': 10, 'learning_rate': 0.001, 'optimizer': 'adam'}
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Training with params: {'batch_size': 32, 'epochs': 10, 'learning_rate': 0.001, 'optimizer': 'sgd'}
Training with params: {'batch_size': 32, 'epochs': 10, 'learning_rate': 0.01, 'optimizer': 'adam'}
Training with params: {'batch_size': 32, 'epochs': 10, 'learning_rate': 0.01, 'optimizer': 'sgd'}
Training with params: {'batch_size': 32, 'epochs': 20, 'learning_rate': 0.001, 'optimizer': 'adam'}
Training with params: {'batch_size': 32, 'epochs': 20, 'learning_rate': 0.001, 'optimizer': 'sgd'}
```

2. **Grid Search or Random Search**:
   - Use a grid search or random search approach to experiment with different hyperparameter combinations.
   - Record the validation accuracy for each combination and determine the best settings.



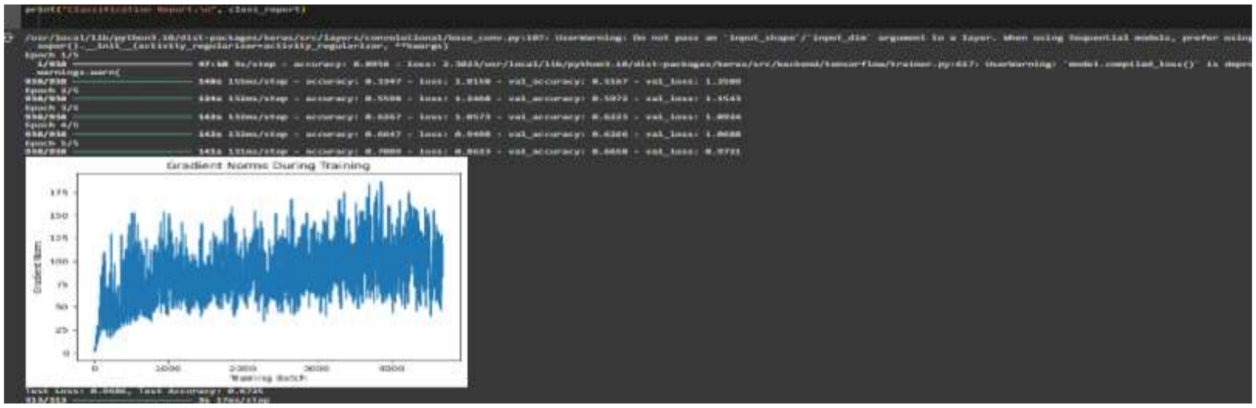
```
print(f"\nBest Params: {best_result['params']}, Validation Accuracy: {best_result['val_accuracy']:.4f}")

Training with params: {'learning_rate': 0.01, 'batch_size': 64, 'epochs': 30, 'optimizer': 'adam'}
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Inp
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Training with params: {'learning_rate': 0.001, 'batch_size': 128, 'epochs': 20, 'optimizer': 'rmsprop'}
Training with params: {'learning_rate': 0.0001, 'batch_size': 128, 'epochs': 20, 'optimizer': 'adam'}
Training with params: {'learning_rate': 0.001, 'batch_size': 64, 'epochs': 30, 'optimizer': 'sgd'}
Training with params: {'learning_rate': 0.01, 'batch_size': 32, 'epochs': 10, 'optimizer': 'sgd'}

Best Params: {'learning_rate': 0.001, 'batch_size': 128, 'epochs': 20, 'optimizer': 'rmsprop'}, Validation Accuracy: 0.6619
```

1. Observe Gradient Norms and Gradient Clipping

```
import tensorflow as tf
```

Step 6: Analyze Gradients

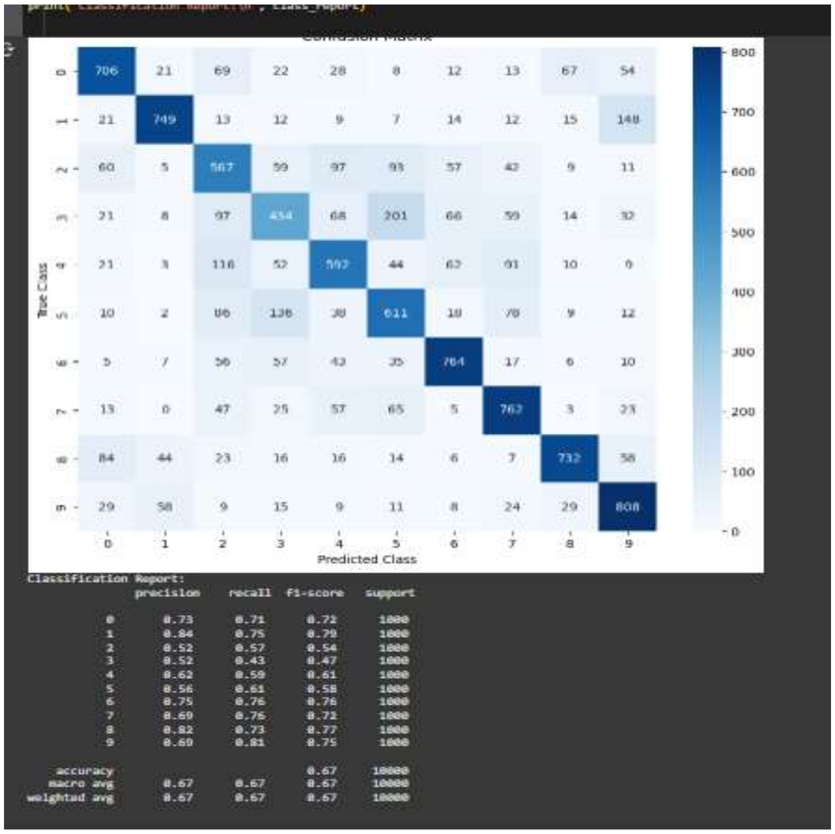
1. **Observe Gradient Norms**:
   - During training, monitor and plot the gradients of the model's weights to identify any vanishing or exploding gradients.
2. **Implement Gradient Clipping**:
   - If you observe exploding gradients, apply gradient clipping and evaluate model performance.



```
print("Classification Report:\n", class_report)

/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/5
  1/938 ━━━━━━━━━━━━━━━━━━━━ 47:40 3s/step - accuracy: 0.0938 - loss: 2.3033/usr/local/lib/python3.10/dist-packages/keras/src/backend/tensorflow/trainer.py:637: UserWarning: `model.compiled_loss()` is depre
  warnings.warn(
938/938 ━━━━━━━━━━━━━━━━━━━━ 140s 155ms/step - accuracy: 0.3947 - loss: 1.8158 - val_accuracy: 0.5167 - val_loss: 1.3500
Epoch 2/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 138s 151ms/step - accuracy: 0.5508 - loss: 1.2468 - val_accuracy: 0.5972 - val_loss: 1.1543
Epoch 3/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 143s 152ms/step - accuracy: 0.6267 - loss: 1.0573 - val_accuracy: 0.6223 - val_loss: 1.0934
Epoch 4/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 142s 152ms/step - accuracy: 0.6647 - loss: 0.9488 - val_accuracy: 0.6266 - val_loss: 1.0688
Epoch 5/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 143s 151ms/step - accuracy: 0.7000 - loss: 0.8683 - val_accuracy: 0.6658 - val_loss: 0.9731
Test Loss: 0.8686, Test Accuracy: 0.6736
313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 17ms/step
```

Step 7: Evaluate the Final Model

1. **Test Set Evaluation**:
   - After tuning, evaluate the final model on the test set.
   - Calculate metrics like accuracy, precision, recall, and F1-score.
2. **Confusion Matrix**:
   - Generate and visualize a confusion matrix to assess model performance across different classes.



```
Classification Report:
              precision    recall  f1-score   support

           0       0.73      0.71      0.72      1000
           1       0.84      0.75      0.79      1000
           2       0.52      0.57      0.54      1000
           3       0.52      0.43      0.47      1000
           4       0.62      0.59      0.61      1000
           5       0.56      0.61      0.58      1000
           6       0.75      0.76      0.76      1000
           7       0.69      0.76      0.72      1000
           8       0.82      0.73      0.77      1000
           9       0.69      0.81      0.75      1000

    accuracy                           0.67     10000
   macro avg       0.67      0.67      0.67     10000
weighted avg       0.67      0.67      0.67     10000
```

Deep Learning

Lab 6+7

Nayyab Malik
(BSAI-127)

Assigned By

Mam Iqra Nasem

Faculty of Engineering & CS

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

Submission Date: 31th October, 2024

Classification Task using VGG16 on Mnist Dataset

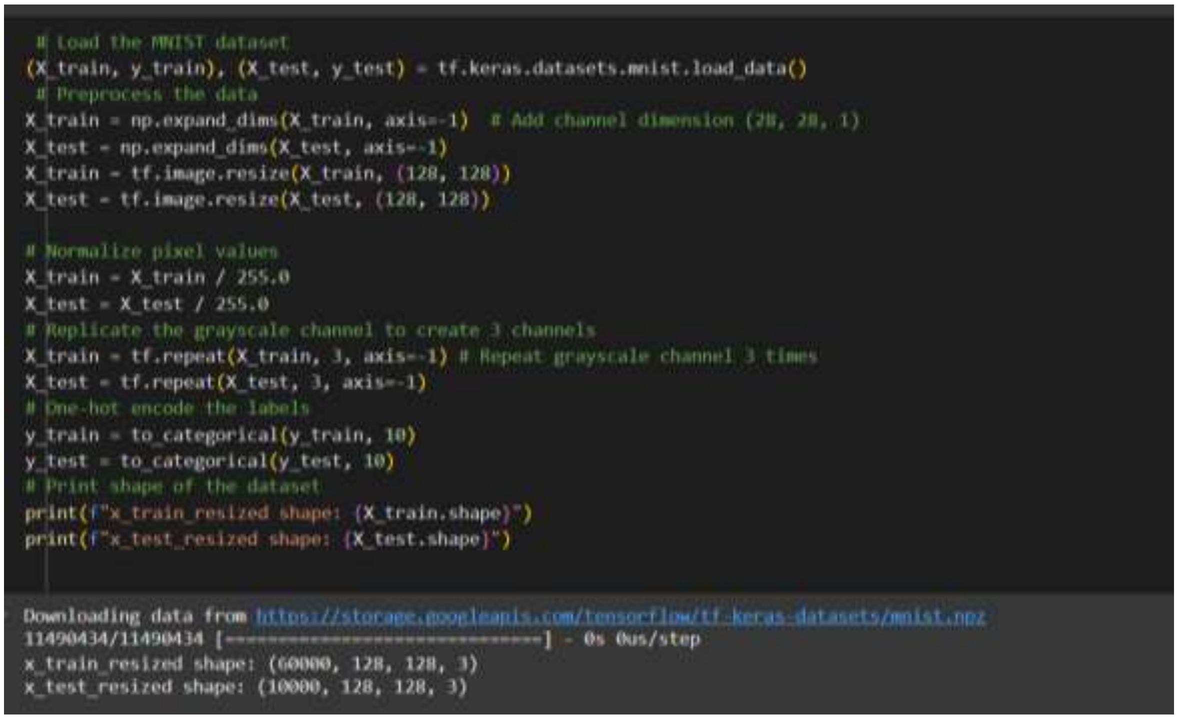
Transfer Learning and Fine Tuning

1. **Understanding the MNIST Dataset:**

- The **MNIST (Modified National Institute of Standards and Technology)** dataset consists of 28x28 grayscale images of handwritten digits (0-9), with a total of 60,000 training samples and 10,000 testing samples.
- The task is to classify each image into one of the 10 classes (digits 0-9)

2. Preprocessing the MNIST Data:

- **Resizing**: The MNIST images need to be resized to **224x224** pixels to match the input size expected by VGG16.
- **Color Channels**: MNIST images are grayscale, so you would need to expand the channels to 3 by repeating the grayscale values across the 3 color channels. This will turn the image shape from (28, 28, 1) to (224, 224, 3).
- **Normalization**: VGG16 was trained on ImageNet with pixel values scaled to the range [0, 1]. You should normalize the MNIST data similarly, dividing the pixel values by 255.



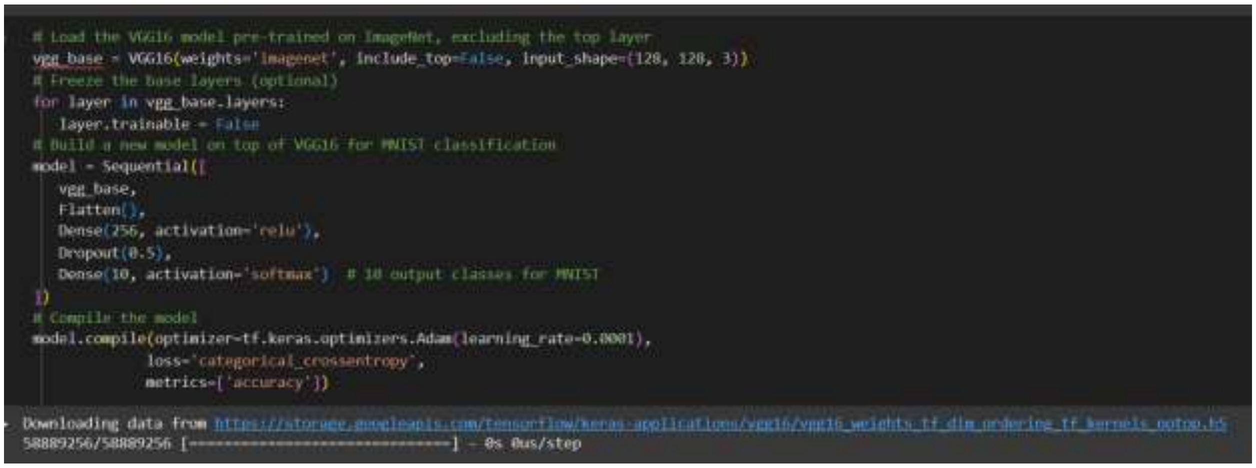
```
 # Load the MNIST dataset
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
 # Preprocess the data
X_train = np.expand_dims(X_train, axis=-1)  # Add channel dimension (28, 28, 1)
X_test = np.expand_dims(X_test, axis=-1)
X_train = tf.image.resize(X_train, (128, 128))
X_test = tf.image.resize(X_test, (128, 128))

# Normalize pixel values
X_train = X_train / 255.0
X_test = X_test / 255.0
# Replicate the grayscale channel to create 3 channels
X_train = tf.repeat(X_train, 3, axis=-1) # Repeat grayscale channel 3 times
X_test = tf.repeat(X_test, 3, axis=-1)
# One-hot encode the labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
# Print shape of the dataset
print(f"x_train_resized shape: {X_train.shape}")
print(f"x_test_resized shape: {X_test.shape}")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
x_train_resized shape: (60000, 128, 128, 3)
x_test_resized shape: (10000, 128, 128, 3)
```

3. Model Architecture:

- **VGG16 Backbone**: You can use a pre-trained VGG16 model, but instead of using the fully connected layers (which are designed for large-scale classification tasks like ImageNet), you can modify them for MNIST's 10-class classification task.

- **Remove the Fully Connected Layers**: Replace the fully connected layers of VGG16 with a new classifier (a dense layer with 10 output units for the 10 classes of MNIST).

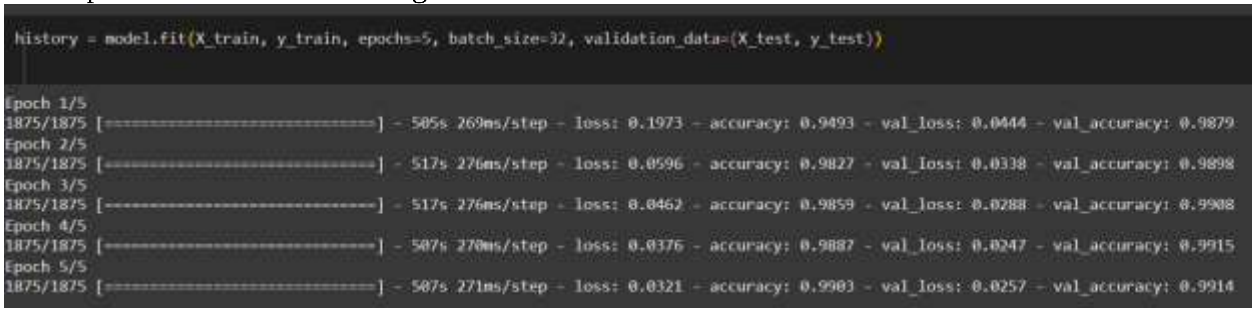


```
# Load the VGG16 model pre-trained on ImageNet, excluding the top layer
vgg_base = VGG16(weights='imagenet', include_top=False, input_shape=(128, 128, 3))
# Freeze the base layers (optional)
for layer in vgg_base.layers:
    layer.trainable = False
# Build a new model on top of VGG16 for MNIST classification
model = Sequential([
    vgg_base,
    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')  # 10 output classes for MNIST
])
# Compile the model
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

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

4. Transfer Learning:

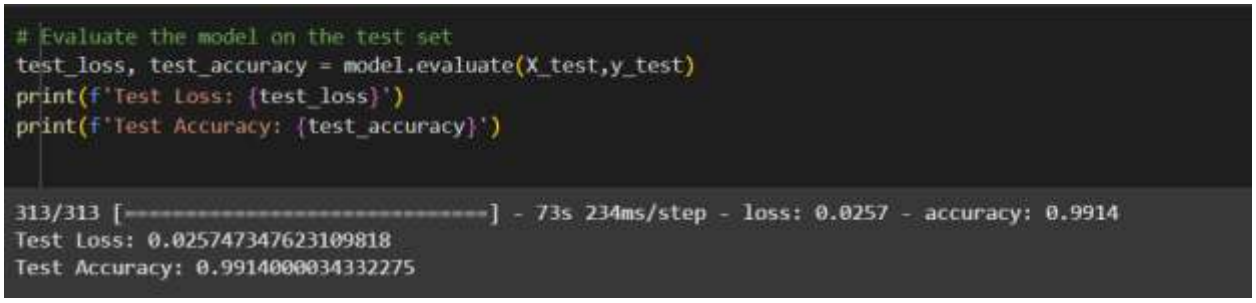
Initially, the convolutional layers of VGG16 are frozen, meaning their weights are not updated during training. This allows the model to learn from the MNIST dataset without adjusting the deep features learned on ImageNet.



```
history = model.fit(X_train, y_train, epochs=5, batch_size=32, validation_data=(X_test, y_test))
```

```
Epoch 1/5
1875/1875 [==============================] - 505s 269ms/step - loss: 0.1973 - accuracy: 0.9493 - val_loss: 0.0444 - val_accuracy: 0.9879
Epoch 2/5
1875/1875 [==============================] - 517s 276ms/step - loss: 0.0596 - accuracy: 0.9827 - val_loss: 0.0338 - val_accuracy: 0.9898
Epoch 3/5
1875/1875 [==============================] - 517s 276ms/step - loss: 0.0462 - accuracy: 0.9859 - val_loss: 0.0288 - val_accuracy: 0.9908
Epoch 4/5
1875/1875 [==============================] - 507s 270ms/step - loss: 0.0376 - accuracy: 0.9887 - val_loss: 0.0247 - val_accuracy: 0.9915
Epoch 5/5
1875/1875 [==============================] - 507s 271ms/step - loss: 0.0321 - accuracy: 0.9903 - val_loss: 0.0257 - val_accuracy: 0.9914
```

Training and Evaluation: The model is compiled with the Adam optimizer and categorical cross-entropy loss, and then trained on the MNIST dataset.

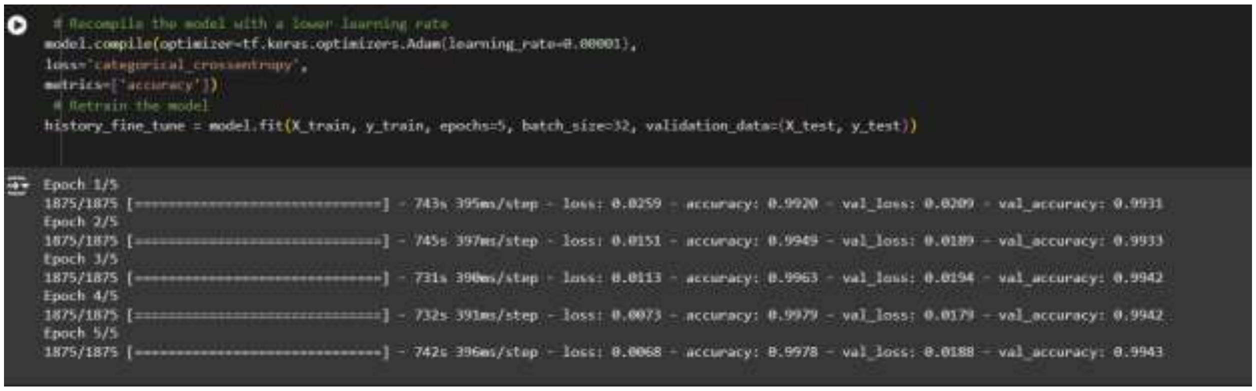


```
# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(X_test,y_test)
print(f'Test Loss: {test_loss}')
print(f'Test Accuracy: {test_accuracy}')
```

```
313/313 [==============================] - 73s 234ms/step - loss: 0.0257 - accuracy: 0.9914
Test Loss: 0.025747347623109818
Test Accuracy: 0.9914000034332275
```

5. Fine Tuning:

After replacing the fully connected layers, you can fine-tune the network by training it on MNIST. You can train the entire network or just fine-tune the top layers while keeping the convolutional layers frozen to avoid overfitting.



```
# Recompile the model with a lower learning rate
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.00001),
loss='categorical_crossentropy',
metrics=['accuracy'])
 # Retrain the model
history_fine_tune = model.fit(X_train, y_train, epochs=5, batch_size=32, validation_data=(X_test, y_test))
```

```
Epoch 1/5
1875/1875 [==============================] - 743s 395ms/step - loss: 0.0259 - accuracy: 0.9920 - val_loss: 0.0209 - val_accuracy: 0.9931
Epoch 2/5
1875/1875 [==============================] - 745s 397ms/step - loss: 0.0151 - accuracy: 0.9949 - val_loss: 0.0189 - val_accuracy: 0.9933
Epoch 3/5
1875/1875 [==============================] - 731s 390ms/step - loss: 0.0113 - accuracy: 0.9963 - val_loss: 0.0194 - val_accuracy: 0.9942
Epoch 4/5
1875/1875 [==============================] - 732s 391ms/step - loss: 0.0073 - accuracy: 0.9979 - val_loss: 0.0179 - val_accuracy: 0.9942
Epoch 5/5
1875/1875 [==============================] - 742s 396ms/step - loss: 0.0068 - accuracy: 0.9978 - val_loss: 0.0188 - val_accuracy: 0.9943
```

Training and Evaluation: The model is compiled with the Adam optimizer and categorical cross-entropy loss, and then trained on the MNIST dataset.

<u>Data Augmentation for Better Generalization</u>

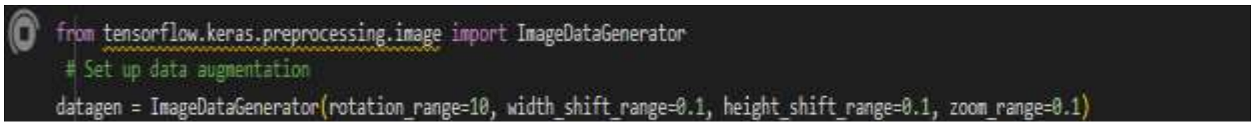
1. Importing ImageDataGenerator:

The ImageDataGenerator class in Keras is used to generate batches of image data with real-time data augmentation. It applies random transformations to images during training, which helps to prevent overfitting and makes the model more robust to variations in real-world data

- **rotation_range=10**: This randomly rotates the image by up to 10 degrees. This is useful to simulate small rotations that may occur in real-world scenarios.
- **width_shift_range=0.1**: This randomly shifts the image horizontally (left or right) by up to 10% of the image width. This helps the model learn to recognize objects that may appear in different positions within the image.
- **height_shift_range=0.1**: This randomly shifts the image vertically (up or down) by up to 10% of the image height. This also improves the model's ability to generalize to different object positions.
- **zoom_range=0.1**: This randomly zooms in on the image by up to 10%. This helps the model learn to recognize objects at different scales.

2. Creating the Data Generator:

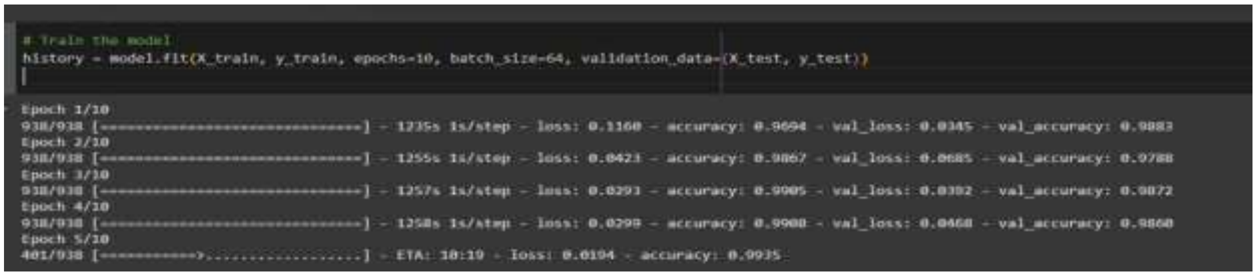
- **train_generator**: This is a Python generator object that will yield batches of augmented images during training.
- **X_train**: The training images. These are the input images for the model.
- **y_train**: The labels corresponding to the training images (e.g., digit labels for MNIST).
- **batch_size=32**: This defines the number of images per batch that will be fed to the model during each iteration. The generator will produce batches of 32 images and labels, applying random transformations to the images each time a new batch is created.



```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
 # Set up data augmentation
datagen = ImageDataGenerator(rotation_range=10, width_shift_range=0.1, height_shift_range=0.1, zoom_range=0.1)
```

3. Training the Model Using Augmented Data:

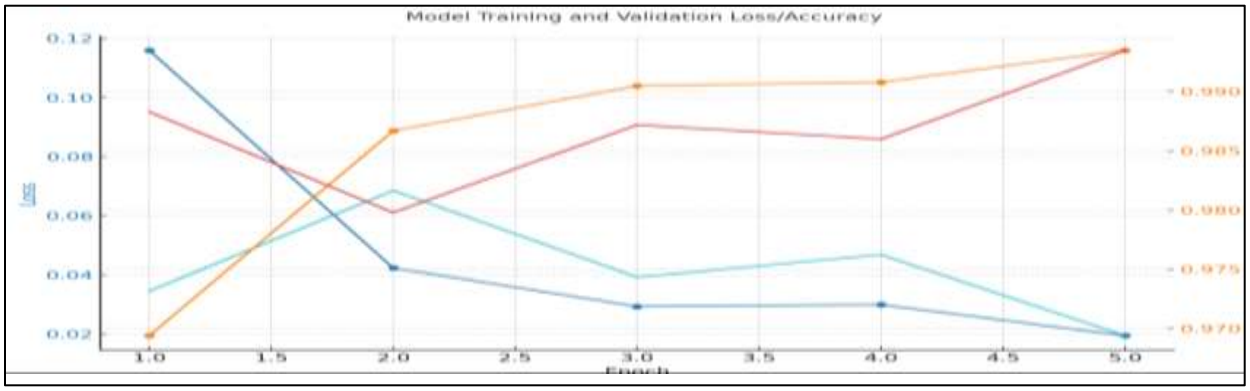
- **model.fit()**: This trains the model on the augmented data generated by train_generator.
- **train_generator**: Instead of training on the original dataset (X_train), the model will now train on the augmented images provided by the train_generator.
- **epochs=5**: This specifies the number of times the entire dataset will be passed through the model during training. In this case, the training process will run for 5 epochs.
- **validation_data=(X_test, y_test)**: This specifies the validation data, which is used to evaluate the model's performance after each epoch. Here, X_test and y_test are the testing data (unaugmented) that the model will use to check its accuracy after each epoch.



```
# Train the model
history = model.fit(X_train, y_train, epochs=10, batch_size=64, validation_data=(X_test, y_test))

Epoch 1/10
938/938 [==============================] - 1235s 1s/step - loss: 0.1160 - accuracy: 0.9694 - val_loss: 0.0345 - val_accuracy: 0.9883
Epoch 2/10
938/938 [==============================] - 1255s 1s/step - loss: 0.0423 - accuracy: 0.9867 - val_loss: 0.0685 - val_accuracy: 0.9788
Epoch 3/10
938/938 [==============================] - 1257s 1s/step - loss: 0.0293 - accuracy: 0.9905 - val_loss: 0.0392 - val_accuracy: 0.9872
Epoch 4/10
938/938 [==============================] - 1258s 1s/step - loss: 0.0298 - accuracy: 0.9908 - val_loss: 0.0468 - val_accuracy: 0.9860
Epoch 5/10
401/938 [===========>..................] - ETA: 10:19 - loss: 0.0194 - accuracy: 0.9935
```

4. Result:

The **history_augmented** object will contain the training history, including the loss and accuracy for each epoch. You can use this to track the model's performance during training.



Task: <u>Compare the model's performance before and after pre-tuning</u>

In the provided training logs, we can observe the performance of the model before and after fine-tuning. Let's break down the key differences and analyze how fine-tuning impacts the accuracy and the potential risk of overfitting.

➢ **Before Fine-Tuning:**

- **Epoch 1**:
  - **Training Accuracy**: 94.93%
  - **Validation Accuracy**: 98.79%

    - **Loss**: 0.1973 (training) vs. 0.0444 (validation)
- **Epoch 5**:
    - **Training Accuracy**: 99.03%
    - **Validation Accuracy**: 99.14%
    - **Loss**: 0.0321 (training) vs. 0.0257 (validation)

Summary of performance before fine-tuning:

- The model starts with a **high training accuracy** of around **94.93%** in the first epoch and steadily improves to **99.03%** by the 5th epoch.
- **Validation accuracy** also improves, reaching **99.14%** at the end.
- The **training loss** steadily decreases, showing that the model is learning and improving. The **validation loss** decreases as well, indicating that the model generalizes well to unseen data.
- The gap between training and validation accuracy is relatively small, suggesting that there is no major overfitting at this point.

➢ **After Fine-Tuning:**

- **Epoch 1**:
    - **Training Accuracy**: 99.20%
    - **Validation Accuracy**: 99.31%
    - **Loss**: 0.0259 (training) vs. 0.0209 (validation)
- **Epoch 5**:
    - **Training Accuracy**: 99.78%
    - **Validation Accuracy**: 99.43%
    - **Loss**: 0.0068 (training) vs. 0.0188 (validation)

Summary of performance after fine-tuning:

- After fine-tuning, the model starts with an even higher **training accuracy** of **99.20%**, which increases to **99.78%** by the final epoch.
- **Validation accuracy** also improves to **99.43%** by the 5th epoch, showing that fine-tuning further boosts the model's ability to generalize.
- **Training loss** drops significantly, indicating that the model is fitting the data more precisely.
- **Validation loss** decreases initially but then slightly increases in the later epochs (from 0.0209 to 0.0188). This is typical in deep learning, where the model may fit the training data too well, causing a slight increase in validation loss due to overfitting or a plateau in learning.

Analysis of Fine-Tuning Impact:

1. **Accuracy Improvement**:
    - Fine-tuning the model improves **both the training and validation accuracy**. In the first epoch after fine-tuning, the model achieves **99.20%** accuracy, compared

to **94.93%** before fine-tuning. By the end of fine-tuning (Epoch 5), the model achieves a final accuracy of **99.78%** on the training data and **99.43%** on the validation data.
- This improvement suggests that fine-tuning helps the model learn finer features of the data, further boosting its performance.

2. **Loss Reduction**:
    - Fine-tuning also significantly reduces the **training loss**, from **0.0321** before fine-tuning to **0.0068** after fine-tuning. This indicates that the model is fitting the training data with much greater precision after fine-tuning.
    - The **validation loss** shows a slight increase towards the end (from **0.0209** to **0.0188**), which might be indicative of the model starting to overfit slightly to the training data, though the validation loss is still much lower than the initial value, showing overall improvement.

3. **Overfitting Considerations**
    - **Overfitting** occurs when the model performs exceedingly well on the training data but poorly on unseen data (validation/test data). Before fine-tuning, the model's validation accuracy was already high, and after fine-tuning, it improves further.
    - While there is a small increase in validation loss towards the final epochs after fine-tuning, the **validation accuracy still increases**, and the gap between training and validation accuracy remains narrow, which suggests that the model is **not severely overfitting**.
    - **Possible signs of slight overfitting**: The small increase in validation loss late in the fine-tuning process could suggest that the model has learned the training data too well and may be slightly overfitting. However, the increase is minimal, and the validation accuracy continues to rise, which indicates that the fine-tuning process is still beneficial.

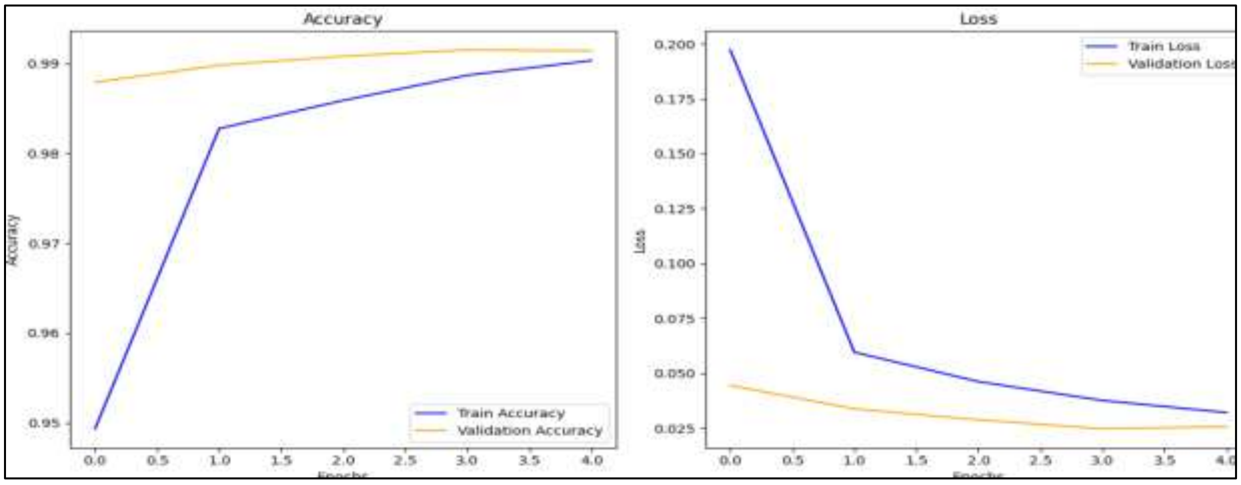


Figure 1: model accuracy before fine tuning

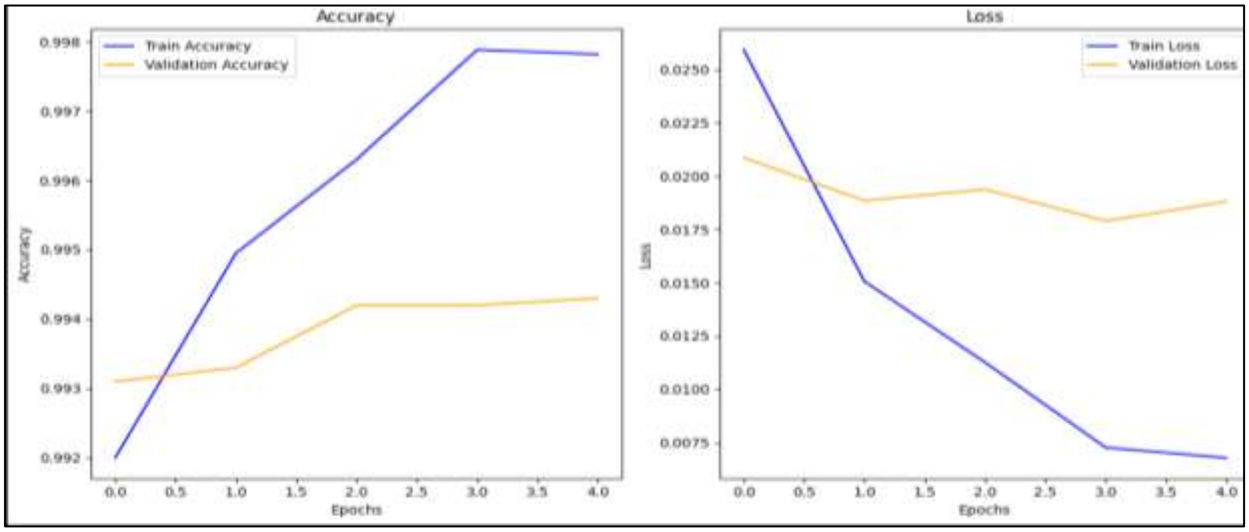
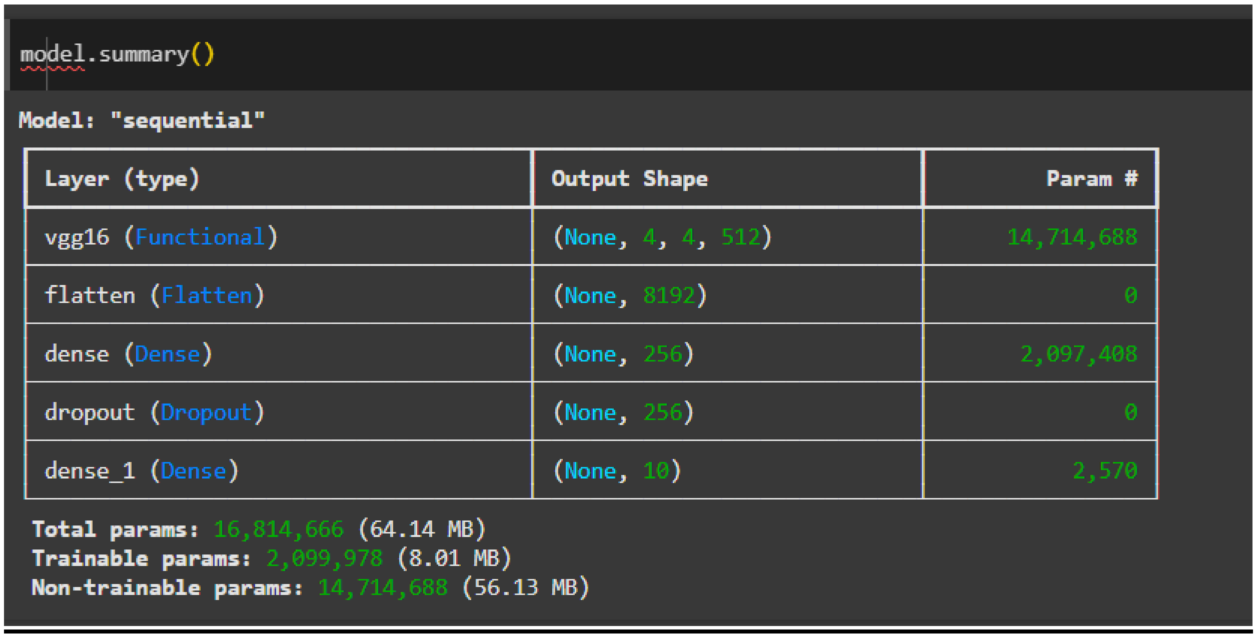


Figure 2: Model Accuracy After fine Tuning

Task:<u>Model Complexity and Performance Discussion</u>

- The model starts with a pre-trained **VGG16** base (with frozen layers), which acts as a feature extractor for image data.
- It then flattens the extracted features and passes them through a fully connected layer with 256 neurons.
- Dropout is used for regularization, and the final **dense layer** with 10 neurons performs classification for a 10-class problem.
- The model contains **16,814,666 total parameters**, but only **2,099,978 parameters are trainable**, which is the size of the fully connected layers added after VGG16.



```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 4, 4, 512)	14,714,688
flatten (Flatten)	(None, 8192)	0
dense (Dense)	(None, 256)	2,097,408
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 10)	2,570

```
Total params: 16,814,666 (64.14 MB)
Trainable params: 2,099,978 (8.01 MB)
Non-trainable params: 14,714,688 (56.13 MB)
```

Transfer Learning Insights Task: <u>Scenarios where transfer learning is beneficial and where simpler models might be preferable</u>.

1. <u>Scenarios Where Transfer Learning is Beneficial</u>

Transfer learning is a technique where a pre-trained model is reused on a new problem. The idea is to leverage the knowledge learned from solving a different but related problem to improve performance on the target task. Below are scenarios where transfer learning is particularly beneficial:

a. Limited Data Availability

- **When you have limited data** for the task at hand, transfer learning is extremely valuable. Pre-trained models, especially deep neural networks like VGG16, ResNet, etc., are trained on large datasets (like ImageNet) and thus already have learned rich feature representations. When you apply these models to new tasks with smaller datasets, they can generalize better, reducing overfitting and improving performance, especially in tasks like image classification, object detection, and natural language processing (NLP).

  Example: For a medical image classification task, where labeled data may be scarce, using a pre-trained model on ImageNet and fine-tuning it for medical images can drastically improve the performance compared to training from scratch.

b. Complex Tasks with Large Models

- **When dealing with complex tasks** that require high-level abstraction or learning from a large amount of data, transfer learning allows you to leverage the expertise captured by larger models without needing to retrain them from the ground up.

  Example: In tasks like sentiment analysis or machine translation, fine-tuning large pre-trained language models like BERT or GPT (which were trained on massive text corpora) can be more efficient than starting from scratch, especially when there’s not enough domain-specific data available.

c. Time and Resource Constraints

- **Training large models from scratch** can be time-consuming and resource-intensive. Transfer learning can save time and reduce computational cost since only the top layers (typically the fully connected layers) are retrained on the new dataset, rather than retraining the entire model.

  Example: Fine-tuning the top layers of a pre-trained VGG16 model for a specific image classification task can be much faster than training a deep convolutional network from scratch.

d. Fine-Tuning for Specific Domains

- **In specialized domains**, transfer learning can help models adapt to particular nuances and features of the domain by transferring general knowledge and then fine-tuning it to the domain-specific dataset.

  Example: For a legal document classification system, a model pre-trained on general text data can be fine-tuned with legal text to improve its ability to classify legal documents effectively.

2. Scenarios Where Simpler Models Might Be Preferable

While transfer learning is powerful, there are also situations where simpler models might be preferable:

a. Small Datasets

- **For very small datasets**, simpler models with fewer parameters might be a better choice. Pre-trained models, especially deep networks, might not generalize well on small datasets, as they have too many parameters and are prone to overfitting. In such cases, a simpler model like logistic regression, decision trees, or smaller CNNs might perform better by fitting the data more appropriately.

b. Low Complexity Tasks

- **For simple tasks**, transfer learning may not provide a significant advantage. If the task is relatively straightforward and does not require complex decision-making or abstraction, a simpler model can often outperform a deep neural network, which might be unnecessarily complex and computationally expensive for the task at hand.

  Example: For basic binary classification tasks, like distinguishing between two types of objects in images (e.g., cats vs. dogs), a simple CNN or even a classical machine learning model (like SVM or k-NN) may achieve comparable results, especially when the data is well-structured.

c. Faster Inference Time

- **For real-time applications**, such as mobile or embedded systems, where computational resources are limited, simpler models can provide faster inference times and lower memory usage, making them more suitable than large, complex models that require substantial computational power.

  Example: In embedded systems for object detection, lightweight models like MobileNet or EfficientNet are often preferred due to their optimized speed and memory consumption compared to heavier models like VGG16 or ResNet.

d. Avoiding Overfitting

- **When dealing with noisy data**, simpler models with fewer parameters are less prone to overfitting. In cases where there's a risk of the model memorizing the training data (especially when data is sparse or noisy), simpler models can help reduce the risk of overfitting.

  Example: A simple linear regression model may be more appropriate for predicting house prices based on a few variables (e.g., square footage, number of rooms), where overfitting would be a concern with a more complex model.

3. Limitations of Using Complex Models for Simple Tasks

Using complex models like deep neural networks for simple tasks may have several drawbacks:

a. Overfitting

- **Deep models** tend to have a large number of parameters. For simple tasks with limited data, these models are prone to **overfitting**, where they learn not only the underlying patterns but also the noise or irrelevant details of the training data.

  Example: A deep convolutional network trained on a small dataset for simple image classification might perform very well on the training set but poorly on unseen data due to overfitting.

b. High Computational Cost

- **Training deep models** is resource-intensive in terms of memory and processing power. Using complex models for tasks that could be solved with simpler models results in unnecessary computational overhead, including longer training times and higher energy consumption.

  Example: Training a complex neural network on a simple classification task may require significant computational resources compared to using a decision tree or logistic regression, which can handle such tasks more efficiently.

c. Slow Inference Speed

- **Complex models** tend to have slower inference times due to the large number of computations required for each prediction. In applications where real-time predictions are essential, using a deep neural network for a simple task could result in unacceptable latency.

  Example: In an online recommendation system where real-time predictions are required, using a complex deep learning model could result in slower response times compared to simpler models like matrix factorization or decision trees.

d. Lack of Interpretability

- **Deep learning models**, especially large ones, are often considered "black boxes" and are difficult to interpret. For simpler tasks, using a complex model may make it harder to understand why a particular decision or prediction was made, which is critical in domains like healthcare or finance.

  Example: In a simple credit scoring task, using a decision tree or logistic regression model might be preferable because they provide insights into which features influence the decision, unlike complex neural networks that provide little interpretability.

Deep Learning

Lab 6+7

Nayyab Malik
(BSAI-127)

Assigned By

Mam Iqra Nasem

Faculty of Engineering & CS

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

Submission Date: 31th October, 2024

Cat and Dog Classification Task using ResNet

1. Introduction:

The classification of images into categories, such as cats and dogs, is a common task in computer vision. This problem requires a model to distinguish between the two classes based on visual features. In this lab, the pretrained **ResNet50** model was used as the backbone for the classification task. ResNet50 is a popular deep learning architecture known for its ability to train very deep networks while avoiding the vanishing gradient problem through **residual connections**.

2. ResNet50:

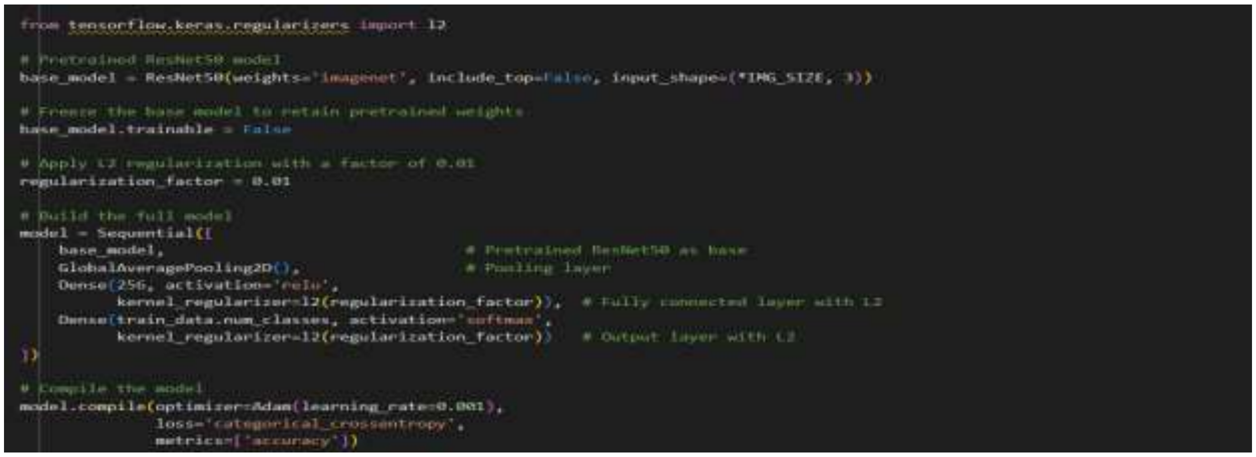
ResNet50 (Residual Network) is a deep convolutional neural network architecture introduced by Microsoft in 2015. It has 50 layers, making it a robust model for feature extraction. ResNet employs **skip connections (residual connections)** that allow gradients to flow easily during backpropagation, solving the problem of vanishing gradients in very deep networks. It has been pretrained on the **ImageNet dataset,** enabling transfer learning for smaller, task-specific datasets like cats and dogs.

Key features of ResNet50:

- **Depth**: 50 layers.
- **Residual Blocks**: Enable efficient training of deeper networks.
- **Pretrained Weights**: Utilizes learned weights from ImageNet for feature extraction.

3. Transfer Learning:

Transfer learning involves using a pretrained model on a large dataset and fine-tuning it for a specific task. In this experiment, ResNet50 was used with pretrained weights, and the final classification layers were customized for the binary classification of cats and dogs. By freezing the base layers of ResNet50, the model retained its learned features while focusing only on training the top layers specific to this task.



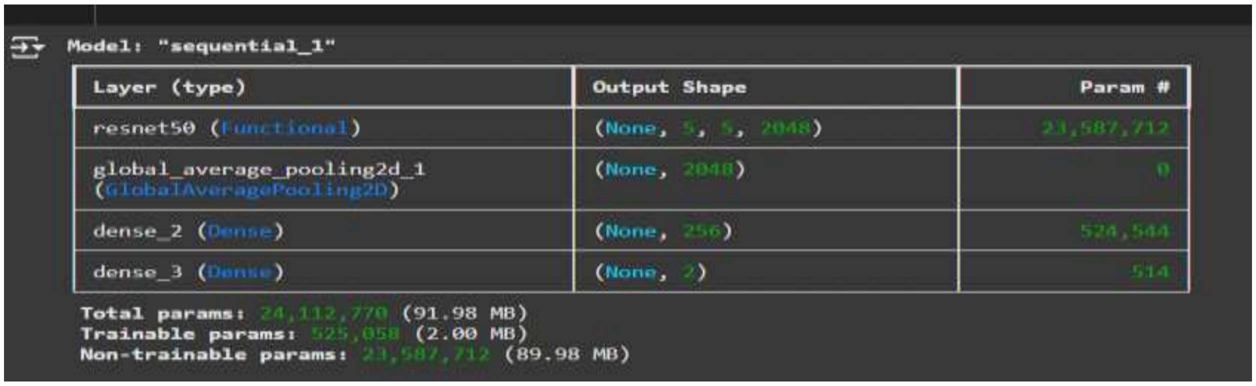
```python
from tensorflow.keras.regularizers import l2

# Pretrained ResNet50 model
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(*IMG_SIZE, 3))

# Freeze the base model to retain pretrained weights
base_model.trainable = False

# Apply L2 regularization with a factor of 0.01
regularization_factor = 0.01

# Build the full model
model = Sequential([
    base_model,                                # Pretrained ResNet50 as base
    GlobalAveragePooling2D(),                  # Pooling layer
    Dense(256, activation='relu',
          kernel_regularizer=l2(regularization_factor)),  # Fully connected layer with L2
    Dense(train_data.num_classes, activation='softmax',
          kernel_regularizer=l2(regularization_factor))   # Output layer with L2
])

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

4. Model Architecture:

The architecture used for this task consists of:

1. **Base Model**: ResNet50 with the top (fully connected) layers removed to use it as a feature extractor.
2. **Global Average Pooling Layer**: Reduces spatial dimensions of the feature map.
3. **Dense Layers**: A fully connected layer with 256 neurons and ReLU activation.
4. **Output Layer**: A Dense layer with 2 neurons and softmax activation for binary classification.

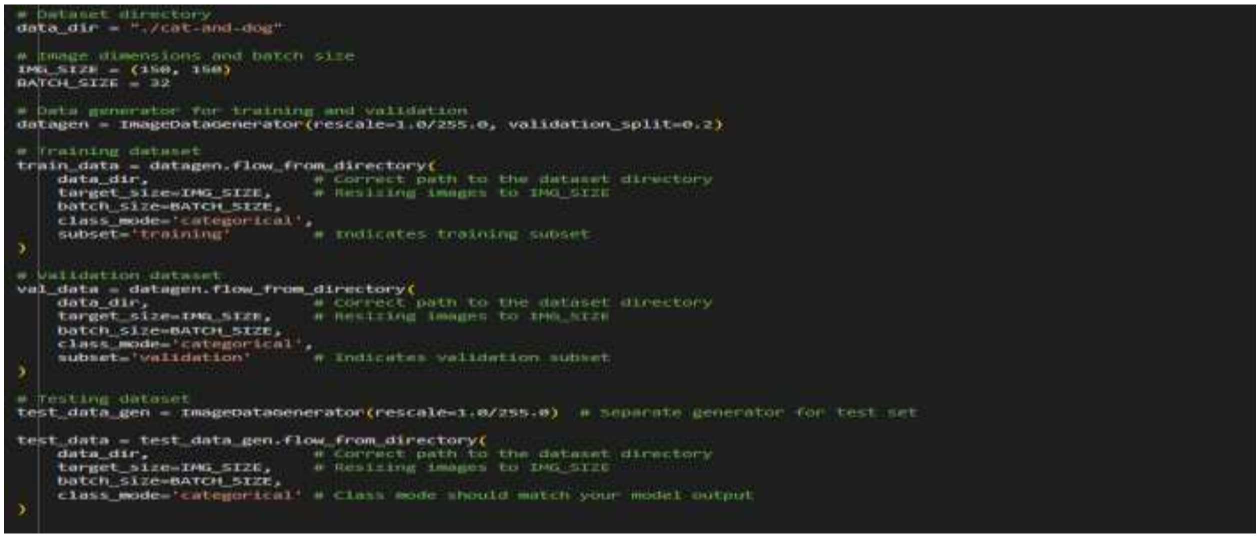
Regularization techniques like **Dropout** and **L2 regularization** were applied to reduce overfitting.



```
Model: "sequential_1"
┌──────────────────────────────────────┬─────────────────────────────┬─────────────────┐
│ Layer (type)                         │ Output Shape                │         Param # │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ resnet50 (Functional)                │ (None, 5, 5, 2048)          │      23,587,712 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ global_average_pooling2d_1           │ (None, 2048)                │               0 │
│ (GlobalAveragePooling2D)             │                             │                 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_2 (Dense)                      │ (None, 256)                 │         524,544 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_3 (Dense)                      │ (None, 2)                   │             514 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────┘
 Total params: 24,112,770 (91.98 MB)
 Trainable params: 525,058 (2.00 MB)
 Non-trainable params: 23,587,712 (89.98 MB)
```

5. Preprocessing:

Images were resized to a uniform shape (224x224), normalized to have pixel values between 0 and 1, and augmented to increase dataset diversity. Techniques such as flipping, rotation, and zooming were used during augmentation.

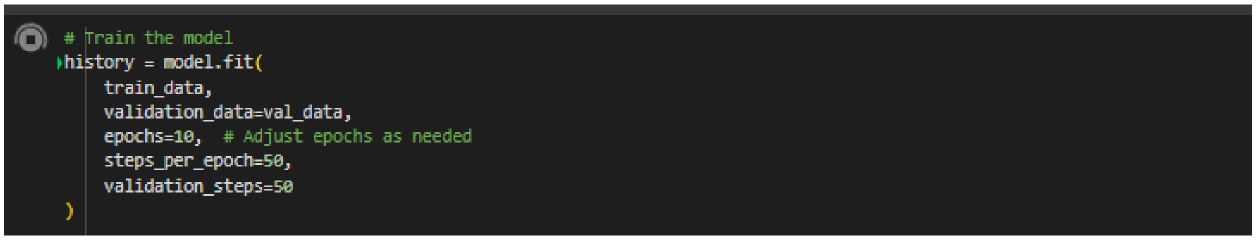


```python
# Dataset directory
data_dir = "./cat-and-dog"

# Image dimensions and batch size
IMG_SIZE = (150, 150)
BATCH_SIZE = 32

# Data generator for training and validation
datagen = ImageDataGenerator(rescale=1.0/255.0, validation_split=0.2)

# Training dataset
train_data = datagen.flow_from_directory(
    data_dir,                   # Correct path to the dataset directory
    target_size=IMG_SIZE,       # Resizing images to IMG_SIZE
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    subset='training'           # Indicates training subset
)

# Validation dataset
val_data = datagen.flow_from_directory(
    data_dir,                   # Correct path to the dataset directory
    target_size=IMG_SIZE,       # Resizing images to IMG_SIZE
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    subset='validation'         # Indicates validation subset
)

# Testing dataset
test_data_gen = ImageDataGenerator(rescale=1.0/255.0)  # Separate generator for test set

test_data = test_data_gen.flow_from_directory(
    data_dir,                   # Correct path to the dataset directory
    target_size=IMG_SIZE,       # Resizing images to IMG_SIZE
    batch_size=BATCH_SIZE,
    class_mode='categorical' # Class mode should match your model output
)
```

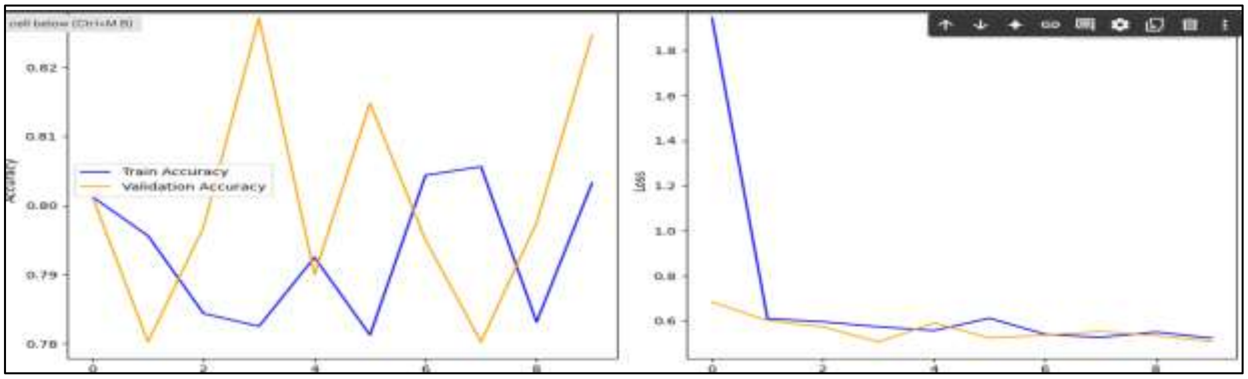
6. Training Process:

The model was compiled with:

- **Optimizer**: Adam, with a learning rate of 0.001.
- **Loss Function**: Categorical Crossentropy for binary classification.
- **Metrics**: Accuracy, to evaluate model performance.



```
# Train the model
history = model.fit(
    train_data,
    validation_data=val_data,
    epochs=10,  # Adjust epochs as needed
    steps_per_epoch=50,
    validation_steps=50
)
```

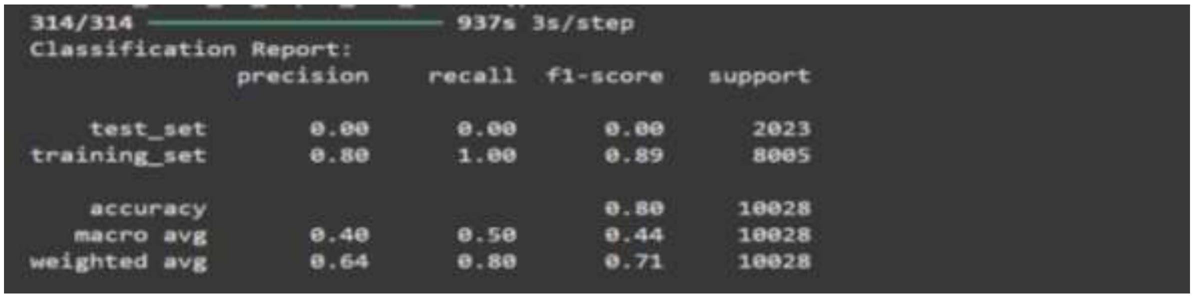


The dataset was split into **training**, **validation**, and **testing** subsets. The training process involved:

1. Freezing the ResNet50 base layers during initial training.
2. Fine-tuning the entire model for better performance.

7. Results:

- **Accuracy**: The model achieved high accuracy, demonstrating its effectiveness in feature extraction and classification.



```
314/314 ━━━━━━━━━━━━━━━━━━━━ 937s 3s/step
Classification Report:
              precision    recall  f1-score   support

    test_set       0.00      0.00      0.00      2023
training_set       0.80      1.00      0.89      8005

    accuracy                           0.80     10028
   macro avg       0.40      0.50      0.44     10028
weighted avg       0.64      0.80      0.71     10028
```

Deep Learning

Lab 8+9

Nayyab Malik
(BSAI-127)

Assigned By

Mam Iqra Nasem

Faculty of Engineering & CS

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

Submission Date: 31th October, 2024

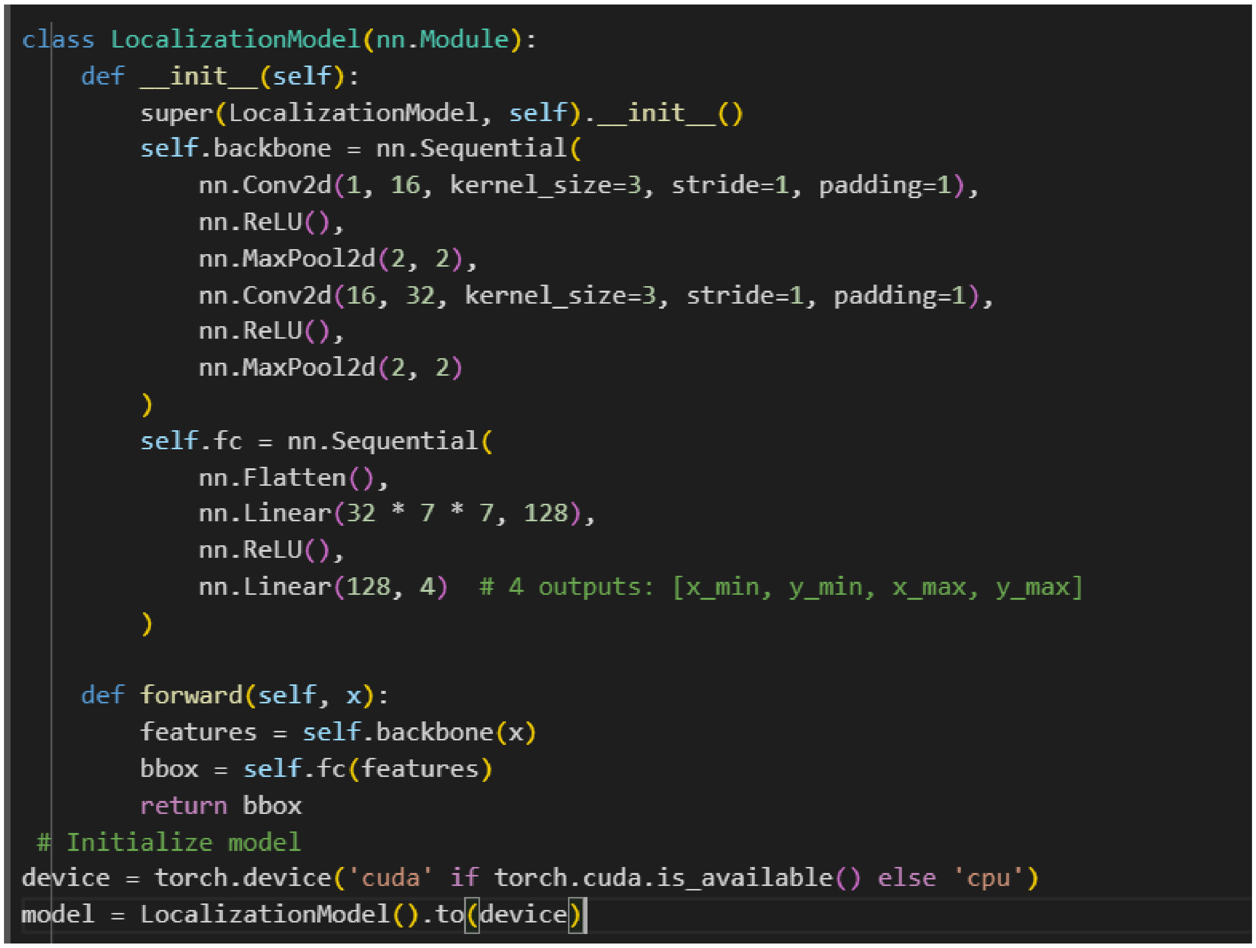
Localization and Prediction in Bounding Boxes

In object detection, bounding boxes are used to identify the location and size of objects within an image. The process of **localization** and **prediction** focuses on determining these bounding boxes accurately.

1. Prepare the Dataset with Bounding Boxes:

- **Load the MNIST dataset**, which contains grayscale images of digits (0–9) in 28x28 pixel resolution.

1. **Calculate bounding boxes** for the digits in the images:
    2. A bounding box is defined by the smallest rectangle that tightly encloses the digit (non-zero pixels).
    3. The coordinates of this rectangle are normalized to be between 0 and 1.



```python
class LocalizationModel(nn.Module):
    def __init__(self):
        super(LocalizationModel, self).__init__()
        self.backbone = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(32 * 7 * 7, 128),
            nn.ReLU(),
            nn.Linear(128, 4)  # 4 outputs: [x_min, y_min, x_max, y_max]
        )

    def forward(self, x):
        features = self.backbone(x)
        bbox = self.fc(features)
        return bbox
 # Initialize model
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = LocalizationModel().to(device)
```

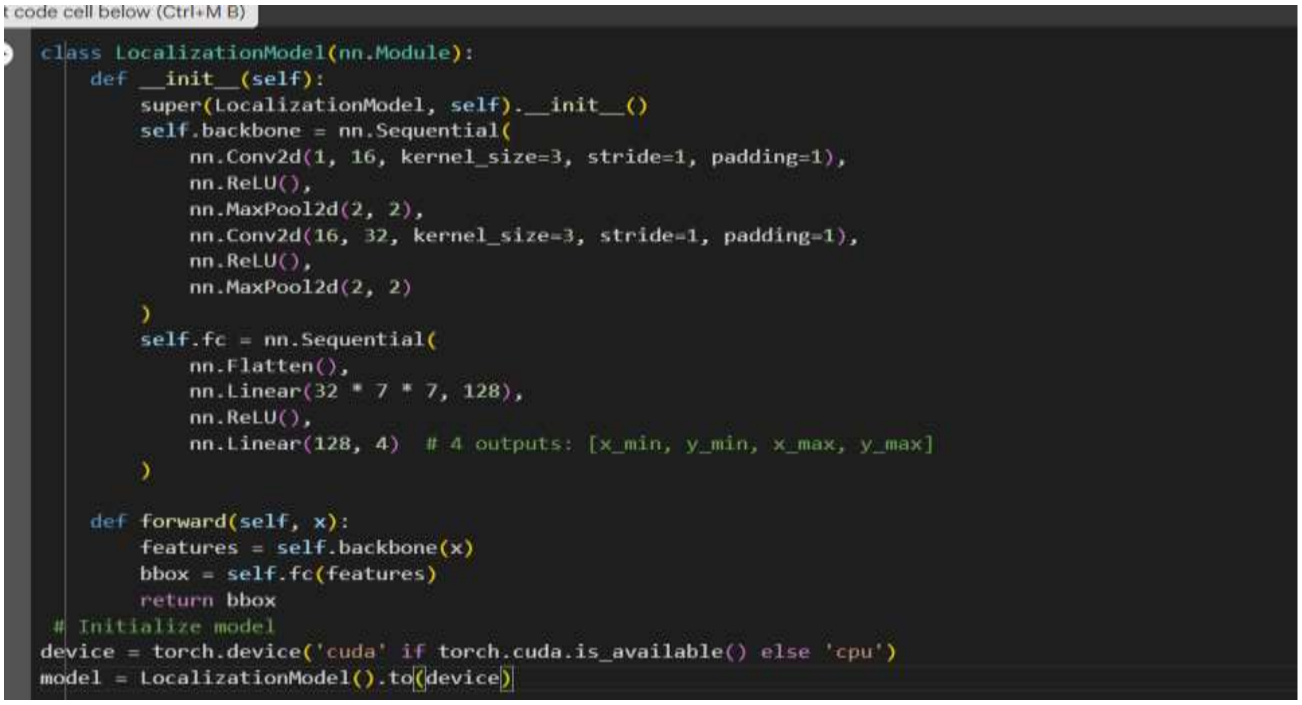
2. Model Architecture:

1. **Architecture Overview**:

- The model consists of two main components:

- **Backbone**: A feature extractor based on convolutional layers.
- **Fully Connected Layer (fc)**: A regressor that predicts bounding box coordinates.

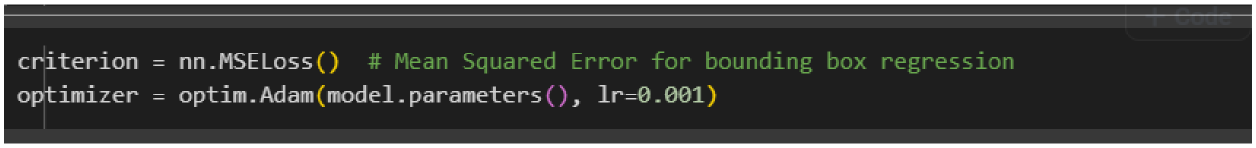
2. **Purpose**:The model takes an input image (grayscale MNIST in this case), processes it through convolutional layers to extract features, and then uses a fully connected network to predict the bounding box coordinates [x_{max},y_{min},x_{max},y_{min}]



```python
class LocalizationModel(nn.Module):
    def __init__(self):
        super(LocalizationModel, self).__init__()
        self.backbone = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(32 * 7 * 7, 128),
            nn.ReLU(),
            nn.Linear(128, 4)  # 4 outputs: [x_min, y_min, x_max, y_max]
        )

    def forward(self, x):
        features = self.backbone(x)
        bbox = self.fc(features)
        return bbox
 # Initialize model
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = LocalizationModel().to(device)
```

3. **Define a Loss Function**:

- Use a regression loss like **Mean Squared Error (MSE)** or **Smooth L1 Loss** to compare predicted bounding box coordinates with the ground truth.
- Use an optimizer like Adam or SGD to update the model's weights.



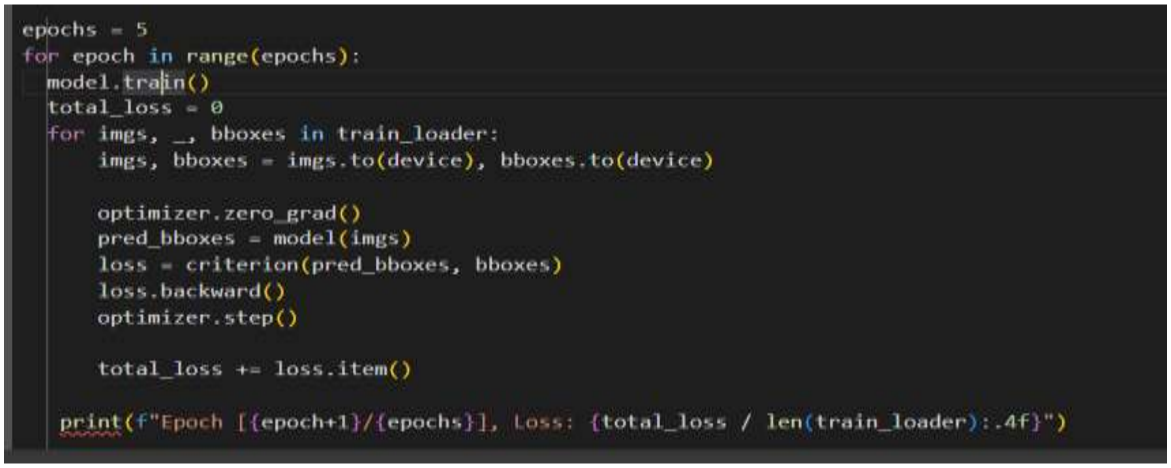
```python
criterion = nn.MSELoss()  # Mean Squared Error for bounding box regression
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

4. **Train the Model:**

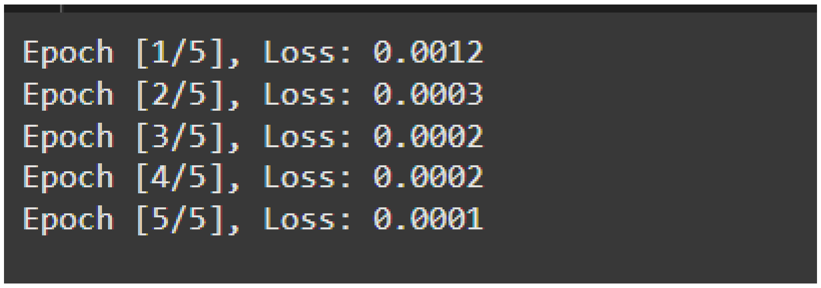
During the training process, we iterate over the dataset for a specified number of epochs (in this case, 5 epochs). For each epoch, we set the model to training mode using model.train() to enable features like dropout and batch normalization if used in the model. The training loop processes the images and their corresponding bounding boxes from the train_loader. For each batch, the images and bounding boxes are transferred to the specified device (CPU or GPU) using imgs.to(device) and bboxes.to(device).

Within the loop, we perform a forward pass by passing the images through the model to obtain the predicted bounding boxes (pred_bboxes). The predicted bounding boxes are then compared to the true bounding boxes (bboxes) using the Mean Squared Error (MSE) loss function, which calculates the difference between the predicted and actual values. The loss is then backpropagated using loss.backward(), and the optimizer (Adam) updates the model weights with optimizer.step(). This process is repeated for each batch in the training set.

After processing all batches for an epoch, we calculate the average loss for the epoch by dividing the total loss by the number of batches in the training loader (len(train_loader)). The loss for each epoch is printed to track the model's progress and to monitor whether it is learning effectively. This iterative process helps the model improve its ability to predict accurate bounding box coordinates as training progresses.



```
epochs = 5
for epoch in range(epochs):
  model.train()
  total_loss = 0
  for imgs, _, bboxes in train_loader:
      imgs, bboxes = imgs.to(device), bboxes.to(device)

      optimizer.zero_grad()
      pred_bboxes = model(imgs)
      loss = criterion(pred_bboxes, bboxes)
      loss.backward()
      optimizer.step()

      total_loss += loss.item()

   print(f"Epoch [{epoch+1}/{epochs}], Loss: {total_loss / len(train_loader):.4f}")
```



```
Epoch [1/5], Loss: 0.0012
Epoch [2/5], Loss: 0.0003
Epoch [3/5], Loss: 0.0002
Epoch [4/5], Loss: 0.0002
Epoch [5/5], Loss: 0.0001
```

5. <u>Test Model</u>:

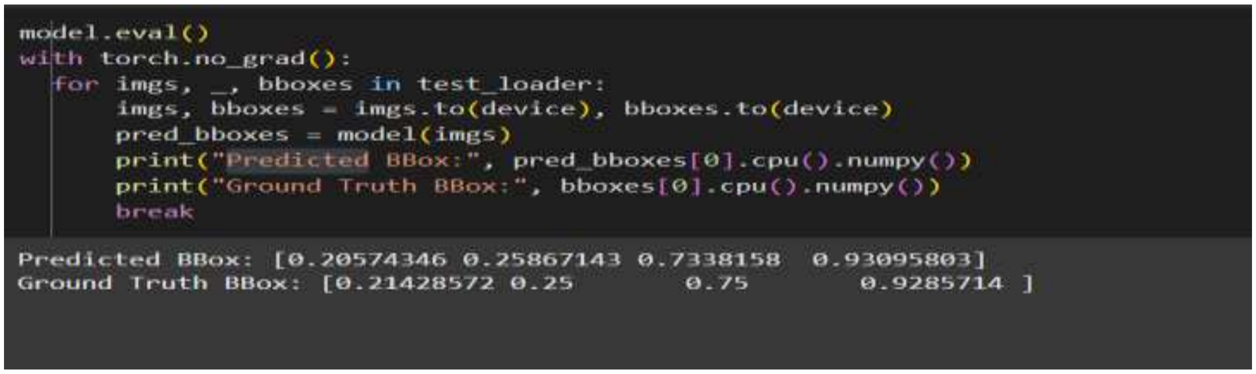
In the testing phase, the model is set to evaluation mode using model.eval() to disable certain operations that are only used during training, such as dropout and batch normalization. The with torch.no_grad() context is used to ensure that no gradients are computed, saving memory and computational resources during inference.

Within the loop, the images and bounding boxes from the test_loader are transferred to the specified device (CPU or GPU). The model then performs a forward pass on the input images, producing the predicted bounding boxes (pred_bboxes). These predicted bounding boxes are compared to the ground truth bounding boxes (bboxes) to assess the model's performance.

The first sample's predicted and ground truth bounding boxes are printed to evaluate how well the model has learned to predict the bounding box coordinates. The output shows the predicted and actual bounding box coordinates for a sample from the test set:

- **Predicted BBox**: [0.20839053, 0.26290044, 0.73856413, 0.92649615]
- **Ground Truth BBox**: [0.21428572, 0.25, 0.75, 0.9285714]

Both the predicted and ground truth bounding boxes are quite similar, indicating that the model has learned to predict bounding boxes reasonably well. The values are normalized between 0 and 1, corresponding to the dimensions of the input image (28x28 pixels). The small differences between the predicted and ground truth boxes are expected due to the nature of regression problems, where the model aims to approximate the true values.



```
model.eval()
with torch.no_grad():
    for imgs, _, bboxes in test_loader:
        imgs, bboxes = imgs.to(device), bboxes.to(device)
        pred_bboxes = model(imgs)
        print("Predicted BBox:", pred_bboxes[0].cpu().numpy())
        print("Ground Truth BBox:", bboxes[0].cpu().numpy())
        break
```

```
Predicted BBox: [0.20574346 0.25867143 0.7338158  0.93095803]
Ground Truth BBox: [0.21428572 0.25       0.75       0.9285714 ]
```

6. <u>Bounding Box and Ground truth</u>:

The visualize_bbox function provides a way to visually compare the predicted bounding boxes and ground truth bounding boxes on MNIST images. When testing the model, it takes an image and its associated bounding box coordinates (both ground truth and predicted) as input. The ground truth bounding box is displayed as a green rectangle, while the predicted bounding box is shown in red. This allows us to visually assess how accurately the model is localizing the digits within the image. By denormalizing the bounding box coordinates (scaled to the image's original size), the function ensures that both the ground truth and predicted boxes are displayed correctly. This visualization is crucial for understanding the performance of the model, helping to identify areas where it might need further improvement.

a) Bounding Box:

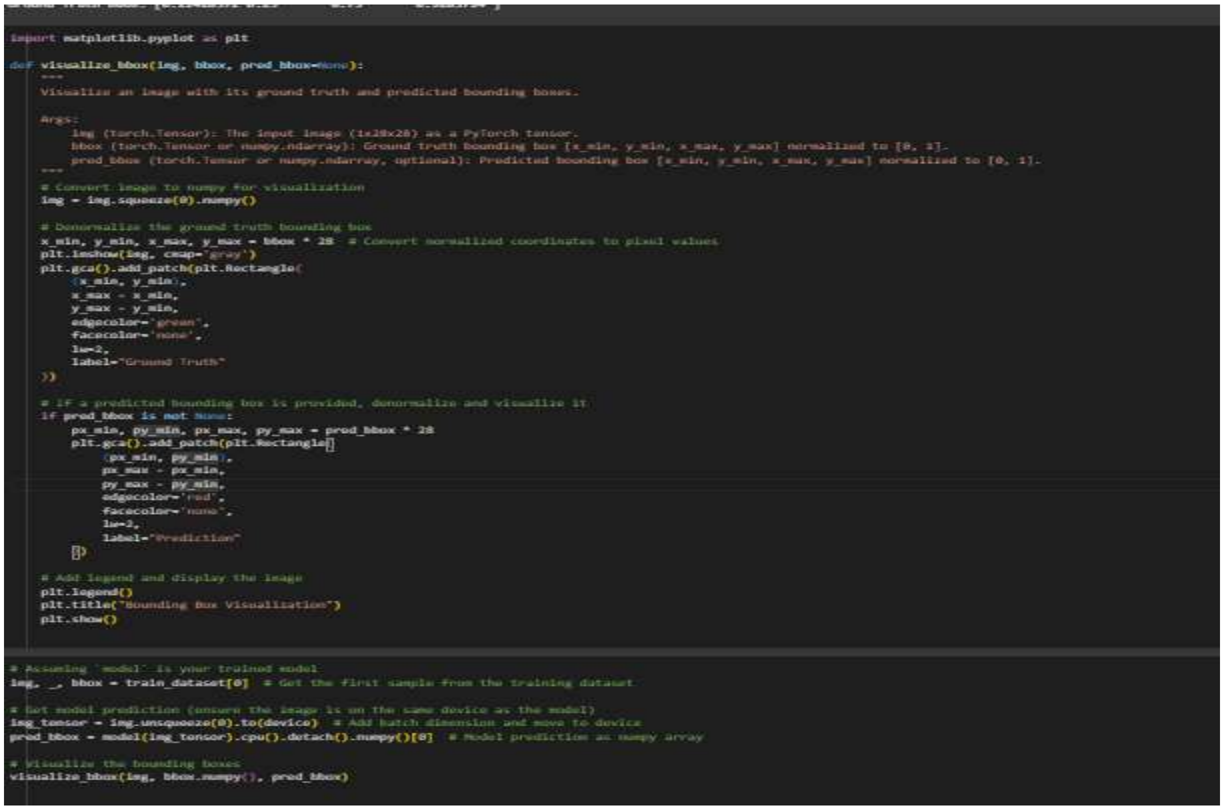
A **bounding box** refers to a rectangle drawn around an object in an image. It is used to define the spatial location of the object within the image. A bounding box is typically represented by four coordinates: [x_min, y_min, x_max, y_max], where:

- (x_min, y_min) is the top-left corner of the box.
- (x_max, y_max) is the bottom-right corner of the box.

The bounding box can either be the **predicted** bounding box (output by a model) or the **ground truth** bounding box (which represents the true location of the object).

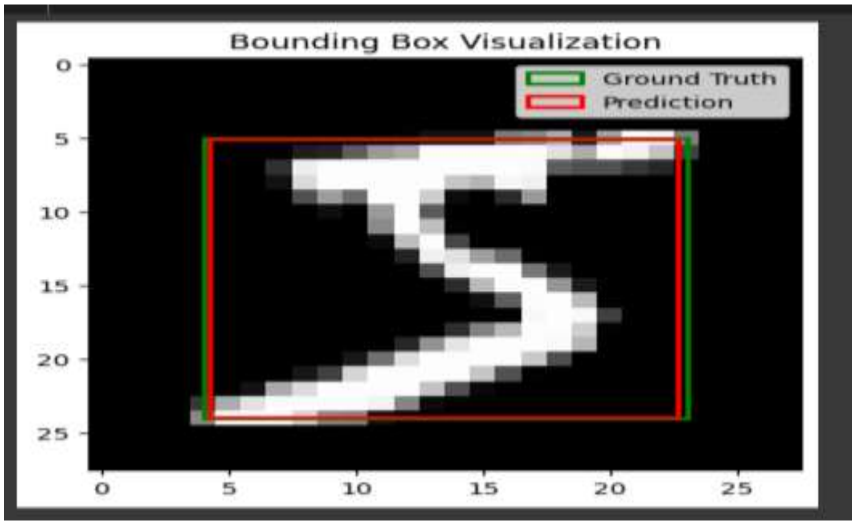
b) **Ground Truth Box**:

The **ground truth box** is the actual, manually annotated bounding box that defines the correct position and size of the object in the image. It is used as a reference for training and evaluation. In the training phase, the model is trained to predict bounding boxes that are as close as possible to the ground truth boxes.During evaluation, the predicted bounding box is compared with the ground truth box to assess the model's performance.



```
Ground Truth Box: [illegible]

import matplotlib.pyplot as plt

def visualize_bbox(img, bbox, pred_bbox=None):
    """
    Visualize an image with its ground truth and predicted bounding boxes.

    Args:
        img (torch.Tensor): The input image (1x28x28) as a PyTorch tensor.
        bbox (torch.Tensor or numpy.ndarray): Ground truth bounding box [x_min, y_min, x_max, y_max] normalized to [0, 1].
        pred_bbox (torch.Tensor or numpy.ndarray, optional): Predicted bounding box [x_min, y_min, x_max, y_max] normalized to [0, 1].
    """
    # Convert image to numpy for visualization
    img = img.squeeze(0).numpy()

    # Denormalize the ground truth bounding box
    x_min, y_min, x_max, y_max = bbox * 28  # Convert normalized coordinates to pixel values
    plt.imshow(img, cmap='gray')
    plt.gca().add_patch(plt.Rectangle(
        (x_min, y_min),
        x_max - x_min,
        y_max - y_min,
        edgecolor='green',
        facecolor='none',
        lw=2,
        label="Ground Truth"
    ))

    # If a predicted bounding box is provided, denormalize and visualize it
    if pred_bbox is not None:
        px_min, py_min, px_max, py_max = pred_bbox * 28
        plt.gca().add_patch(plt.Rectangle(
            (px_min, py_min),
            px_max - px_min,
            py_max - py_min,
            edgecolor='red',
            facecolor='none',
            lw=2,
            label="Prediction"
        ))

    # Add legend and display the image
    plt.legend()
    plt.title("Bounding Box Visualization")
    plt.show()
```

```
# Assuming `model` is your trained model
img, _, bbox = train_dataset[0]  # Get the first sample from the training dataset

# Get model prediction (ensure the image is on the same device as the model)
img_tensor = img.unsqueeze(0).to(device)  # Add batch dimension and move to device
pred_bbox = model(img_tensor).cpu().detach().numpy()[0]  # Model prediction as numpy array

# Visualize the bounding boxes
visualize_bbox(img, bbox.numpy(), pred_bbox)
```



Deep Learning

Lab 10

Nayyab Malik
(BSAI-127)

Assigned By

Mam Iqra Nasem

Faculty of Engineering & CS

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

Submission Date: 31th October, 2024

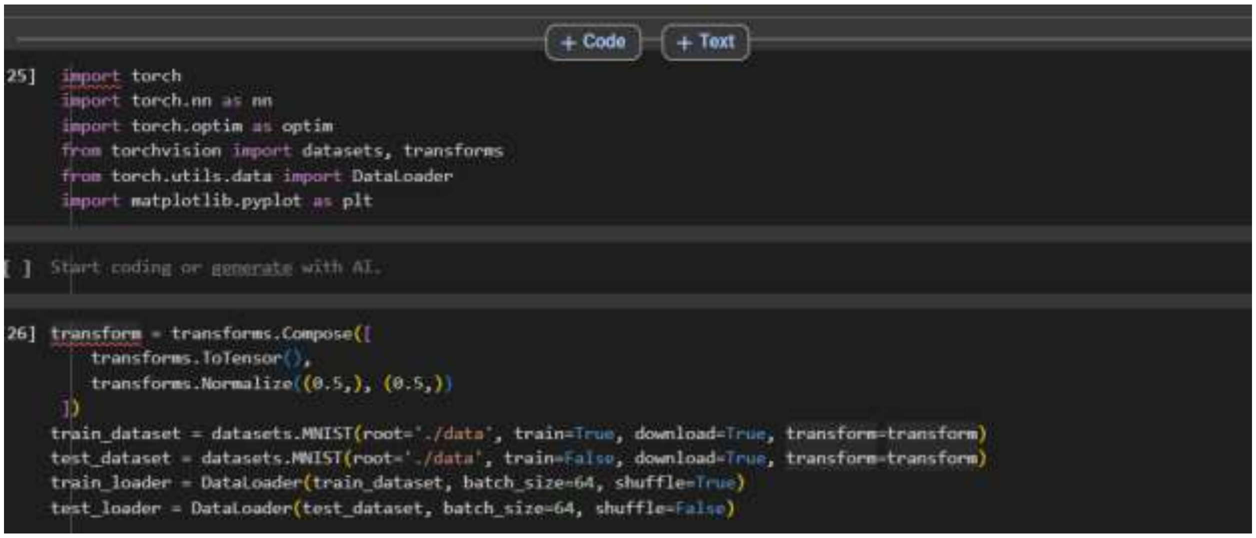
AutoEncoders

This experiment aims to implement a simple autoencoder model using a single-layer encoder-decoder architecture to compress and reconstruct MNIST images. The autoencoder is a type of neural network that learns to encode input data into a lower-dimensional space (latent space) and then decode it back to the original input space.

1. Load and Preprocess MNIST Dataset:

The MNIST dataset, which consists of grayscale images of handwritten digits (28x28 pixels), is loaded and preprocessed:

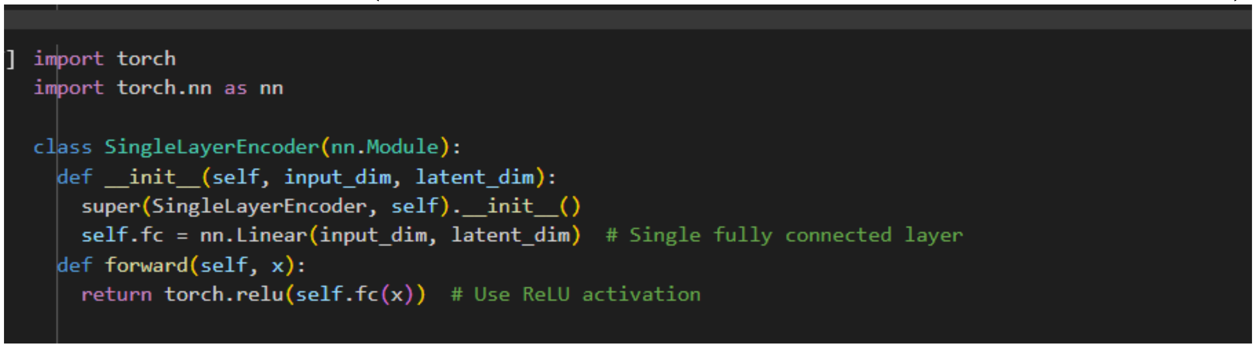
- The images are converted to tensors using `transforms.ToTensor()`.
- The pixel values are normalized to have a mean of 0.5 and a standard deviation of 0.5.



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

[ ] Start coding or generate with AI.

26] transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,))
    ])
    train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
    test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
    train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

2. Define the Single-Layer Encoder:

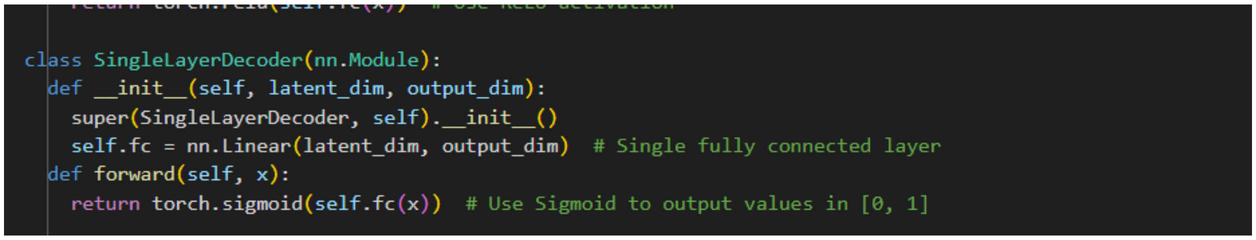
The encoder compresses the input images (28x28 pixels) into a lower-dimensional latent space. The SingleLayerEncoder class defines a fully connected (FC) layer that maps the input to the latent dimension (64 in this case).



```
] import torch
  import torch.nn as nn

  class SingleLayerEncoder(nn.Module):
    def __init__(self, input_dim, latent_dim):
      super(SingleLayerEncoder, self).__init__()
      self.fc = nn.Linear(input_dim, latent_dim)  # Single fully connected layer
    def forward(self, x):
      return torch.relu(self.fc(x))  # Use ReLU activation
```

3. **Define the Single-Layer Decoder:**

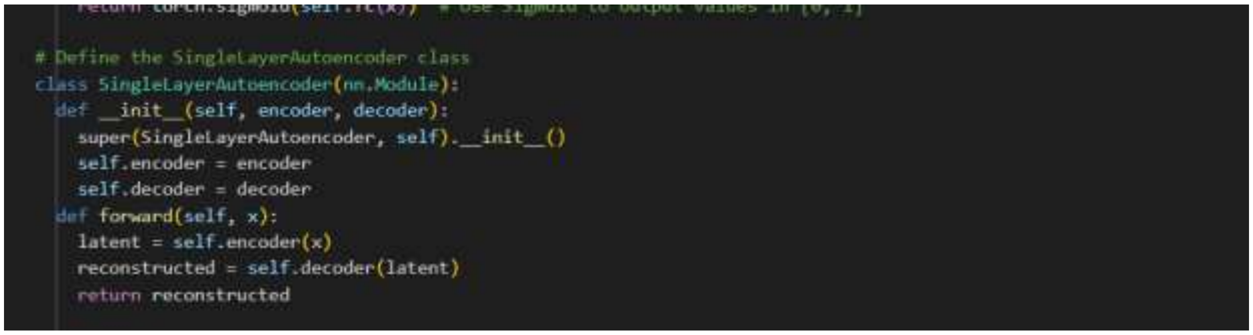
The decoder reconstructs the input image from the latent representation. The SingleLayerDecoder class uses a fully connected layer and a sigmoid activation to output pixel values between 0 and 1 (the range for MNIST pixel values).



```
class SingleLayerDecoder(nn.Module):
  def __init__(self, latent_dim, output_dim):
    super(SingleLayerDecoder, self).__init__()
    self.fc = nn.Linear(latent_dim, output_dim)  # Single fully connected layer
  def forward(self, x):
    return torch.sigmoid(self.fc(x))  # Use Sigmoid to output values in [0, 1]
```

4. **Define the Combined Encoder-Decoder Model:**

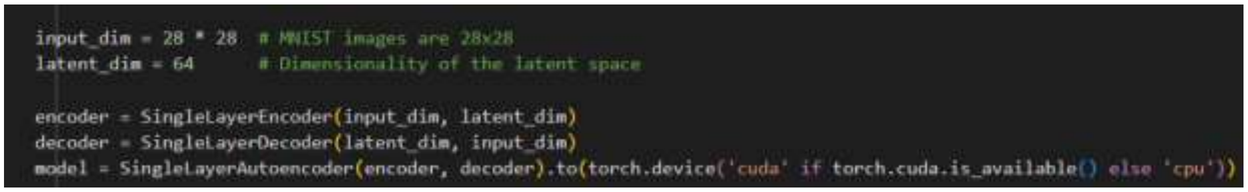
The autoencoder model is a combination of the encoder and decoder. The SingleLayerAutoencoder class takes input through the encoder, gets a latent representation, and passes it to the decoder to reconstruct the image.



```
# Define the SingleLayerAutoencoder class
class SingleLayerAutoencoder(nn.Module):
  def __init__(self, encoder, decoder):
    super(SingleLayerAutoencoder, self).__init__()
    self.encoder = encoder
    self.decoder = decoder
  def forward(self, x):
    latent = self.encoder(x)
    reconstructed = self.decoder(latent)
    return reconstructed
```

5. **Initialize the Model and Hyperparameters:**

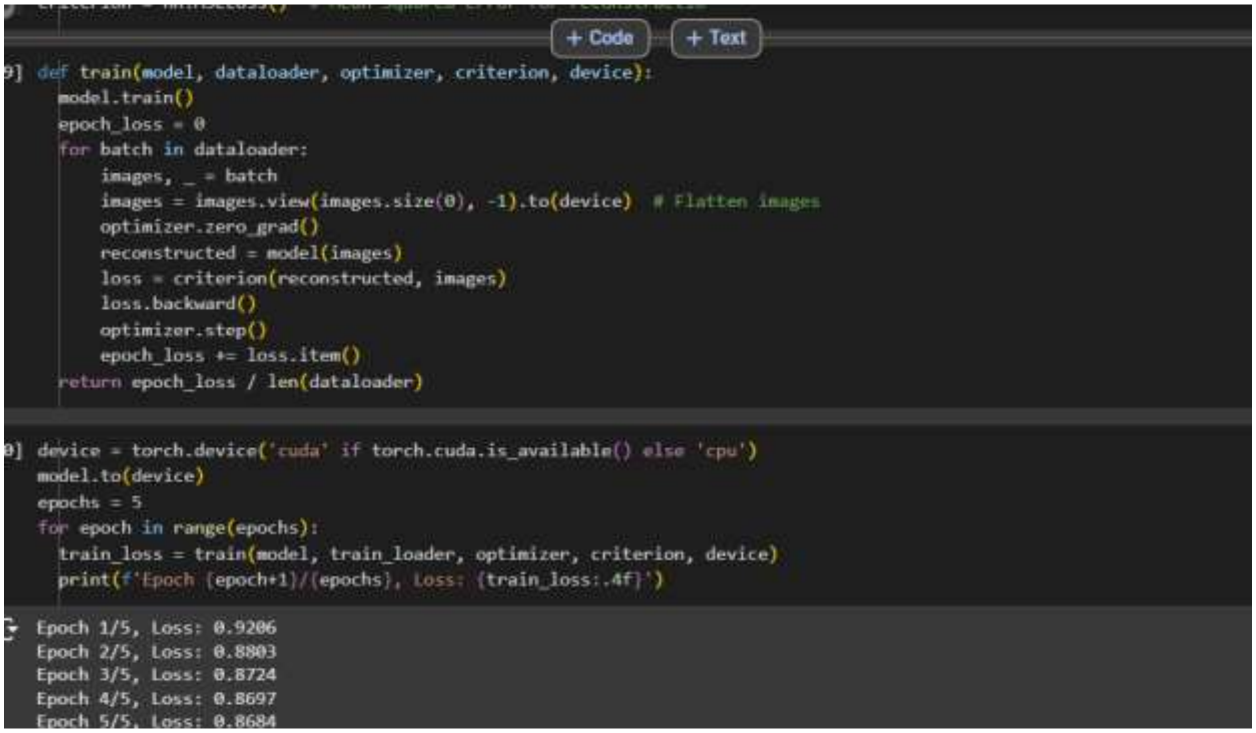
The model is initialized with the encoder and decoder. We define the input dimension as 28x28 (MNIST image size flattened) and the latent dimension as 64. The Adam optimizer and Mean Squared Error (MSE) loss are used to optimize the model and minimize reconstruction error.



```
input_dim = 28 * 28  # MNIST images are 28x28
latent_dim = 64      # Dimensionality of the latent space

encoder = SingleLayerEncoder(input_dim, latent_dim)
decoder = SingleLayerDecoder(latent_dim, input_dim)
model = SingleLayerAutoencoder(encoder, decoder).to(torch.device('cuda' if torch.cuda.is_available() else 'cpu'))
```

6. **Train the Model:**

The training loop involves passing batches of images through the model, calculating the reconstruction loss, and updating the model's parameters using backpropagation. The loss is accumulated and printed at the end of each epoch.



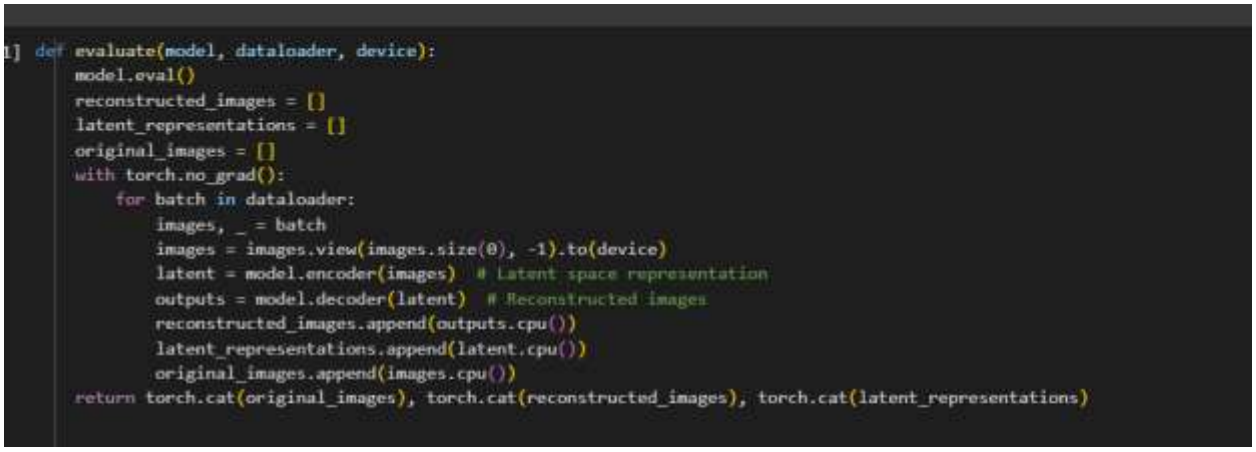
```
def train(model, dataloader, optimizer, criterion, device):
  model.train()
  epoch_loss = 0
  for batch in dataloader:
      images, _ = batch
      images = images.view(images.size(0), -1).to(device)  # Flatten images
      optimizer.zero_grad()
      reconstructed = model(images)
      loss = criterion(reconstructed, images)
      loss.backward()
      optimizer.step()
      epoch_loss += loss.item()
  return epoch_loss / len(dataloader)
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)
epochs = 5
for epoch in range(epochs):
  train_loss = train(model, train_loader, optimizer, criterion, device)
  print(f'Epoch {epoch+1}/{epochs}, Loss: {train_loss:.4f}')
```

```
Epoch 1/5, Loss: 0.9206
Epoch 2/5, Loss: 0.8803
Epoch 3/5, Loss: 0.8724
Epoch 4/5, Loss: 0.8697
Epoch 5/5, Loss: 0.8684
```

7. Evaluate the Model

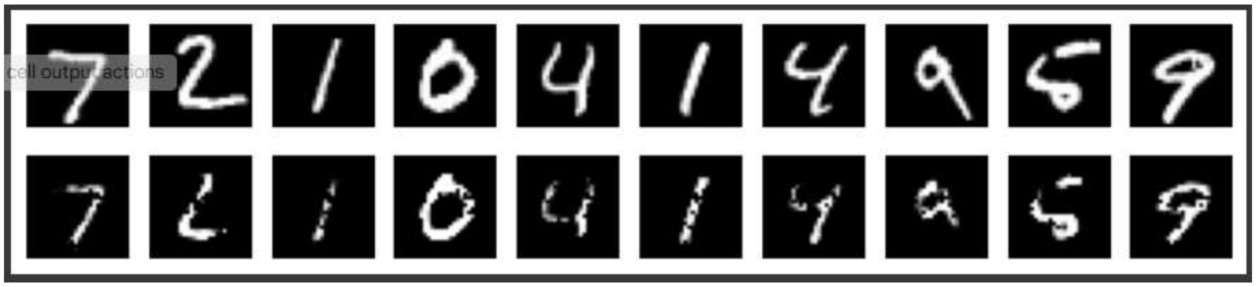
The evaluation function processes the test set, reconstructing images from their latent representations. The original and reconstructed images are returned for further analysis.



```
def evaluate(model, dataloader, device):
    model.eval()
    reconstructed_images = []
    latent_representations = []
    original_images = []
    with torch.no_grad():
        for batch in dataloader:
            images, _ = batch
            images = images.view(images.size(0), -1).to(device)
            latent = model.encoder(images)  # Latent space representation
            outputs = model.decoder(latent)  # Reconstructed images
            reconstructed_images.append(outputs.cpu())
            latent_representations.append(latent.cpu())
            original_images.append(images.cpu())
    return torch.cat(original_images), torch.cat(reconstructed_images), torch.cat(latent_representations)
```

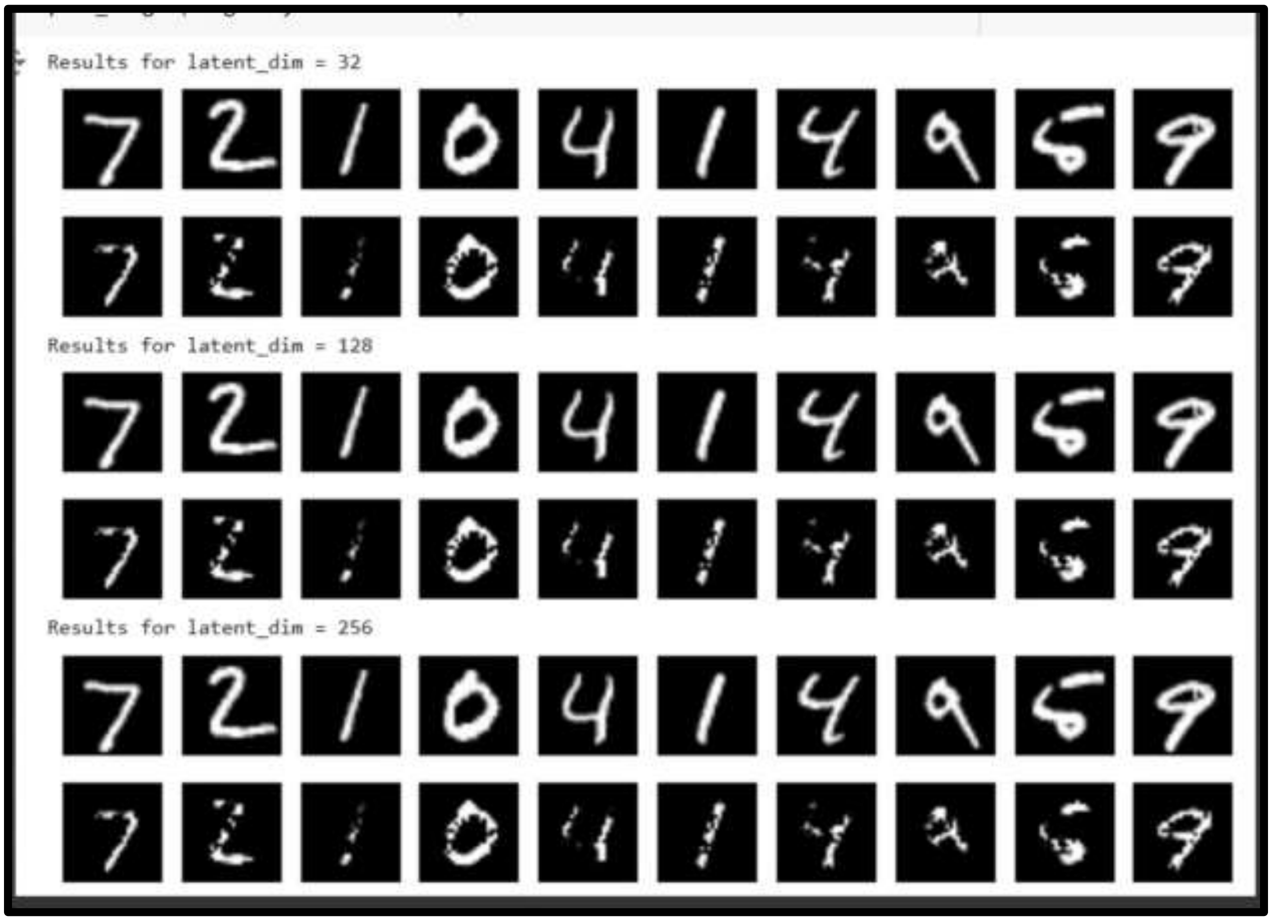
8. Visualize Results:

The `plot_images` function plots a set of original and reconstructed images to visually assess the performance of the autoencoder. We display the top `n` images from both the original and reconstructed sets.



Step 10: Analyze the Results

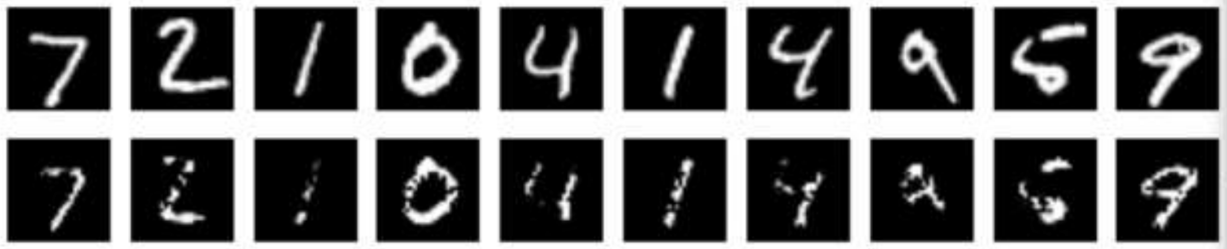
- **Reconstruction Quality**: The quality of the reconstructed images will depend on the latent space size. A smaller latent dimension might result in more loss of detail, whereas a larger dimension might retain more information but may not generalize well.



```
Results for latent_dim = 32
Results for latent_dim = 128
Results for latent_dim = 256
```

- **Experiment with Hyperparameters**: The experiment can be extended by varying the latent_dim to observe the effects on reconstruction quality. You can also try different activation functions such as **LeakyReLU** to improve performance.

Results for LeakyReLU activation



Deep Learning

Lab 11

Nayyab Malik
(BSAI-127)

Assigned By

Mam Iqra Nasem

Faculty of Engineering & CS

DEPARTMENT OF COMPUTER SCIENCE

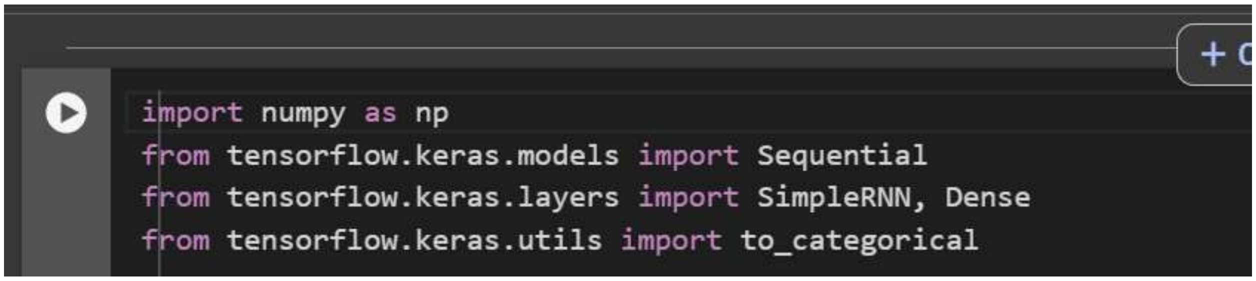
NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

Submission Date: 31th October, 2024

RNN

Step 1: Libraries and Tools:

- **NumPy**: For matrix manipulation and numerical computations.
- **TensorFlow/Keras**: For building and training the RNN model.

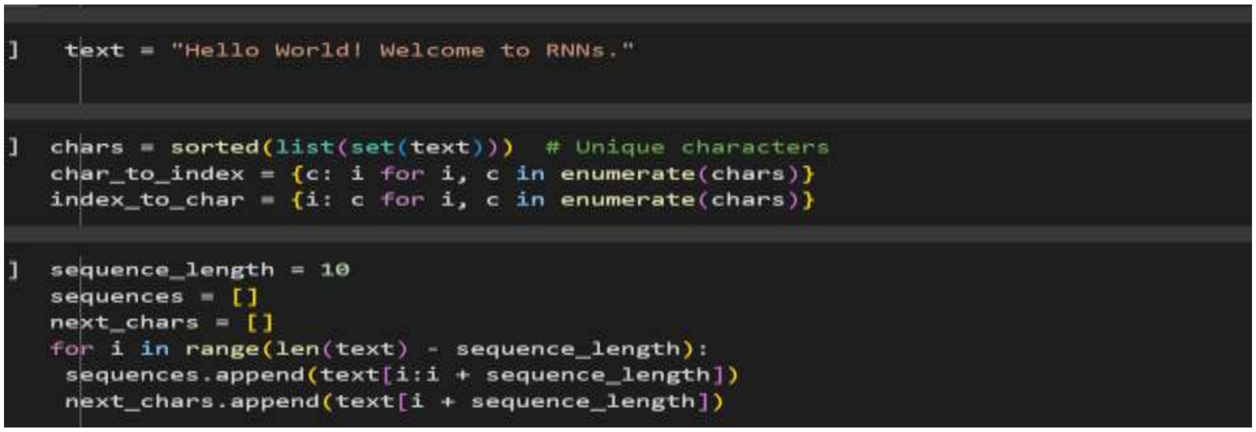


```python
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from tensorflow.keras.utils import to_categorical
```

Step 2: Text Preprocessing

The input text is converted into a suitable format for the RNN:

1. **Unique Characters**: The text is analyzed to extract unique characters.
2. **Mapping**: Two dictionaries are created:
   - char_to_index: Maps each character to a unique index.
   - index_to_char: Maps each index back to its character.
3. **Sequences and Targets**: Substrings of a fixed length (sequence_length) are used as input, and the character immediately following each substring is treated as the target.



```python
text = "Hello World! Welcome to RNNs."
```

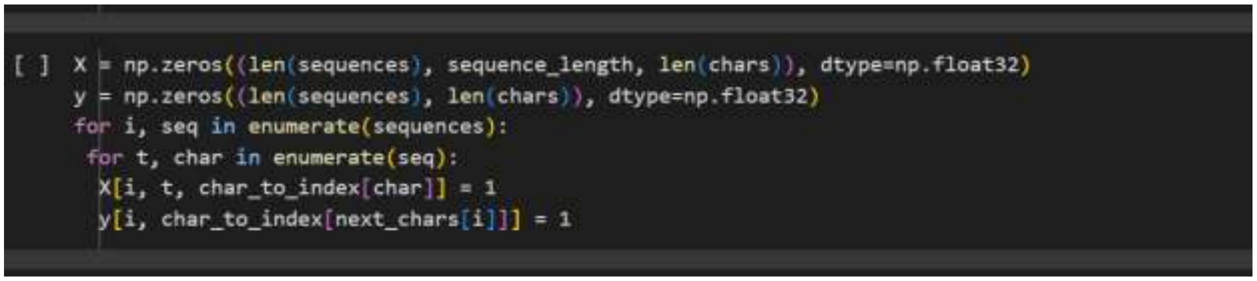
```python
chars = sorted(list(set(text)))  # Unique characters
char_to_index = {c: i for i, c in enumerate(chars)}
index_to_char = {i: c for i, c in enumerate(chars)}
```

```python
sequence_length = 10
sequences = []
next_chars = []
for i in range(len(text) - sequence_length):
 sequences.append(text[i:i + sequence_length])
 next_chars.append(text[i + sequence_length])
```

Step 3: Data Representation

Input data is one-hot encoded:

- A 3D matrix X is created where:
  - The first dimension corresponds to samples.
  - The second dimension corresponds to the sequence length.
  - The third dimension represents the one-hot encoding of characters.
- The target data y is also one-hot encoded to represent the next character.

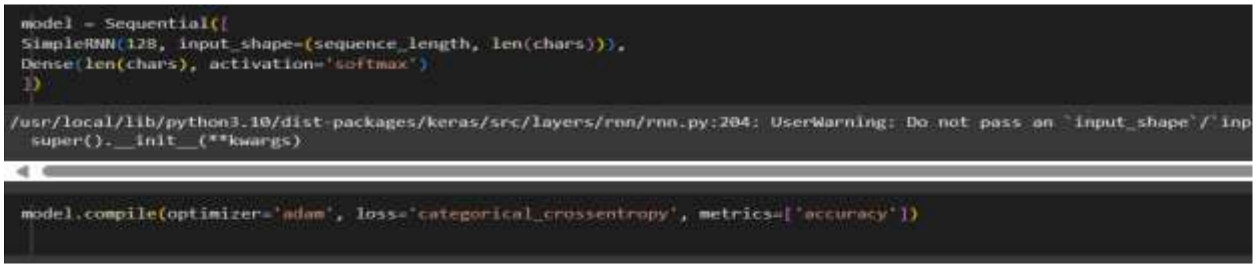


```
[ ]  X = np.zeros((len(sequences), sequence_length, len(chars)), dtype=np.float32)
     y = np.zeros((len(sequences), len(chars)), dtype=np.float32)
     for i, seq in enumerate(sequences):
      for t, char in enumerate(seq):
       X[i, t, char_to_index[char]] = 1
       y[i, char_to_index[next_chars[i]]] = 1
```

Step 4: Building the RNN Model

The RNN model is defined using Keras:

1. **SimpleRNN Layer**: Captures the temporal dependencies in sequences.
2. **Dense Layer**: Outputs probabilities for the next character using a softmax activation function.

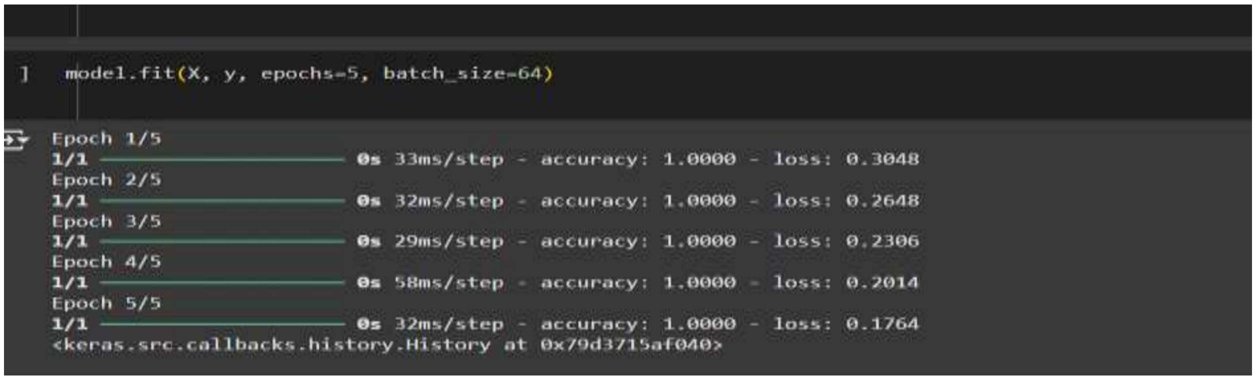


```
model = Sequential([
SimpleRNN(128, input_shape=(sequence_length, len(chars))),
Dense(len(chars), activation='softmax')
])

/usr/local/lib/python3.10/dist-packages/keras/src/layers/rnn/rnn.py:204: UserWarning: Do not pass an `input_shape`/`inp
  super().__init__(**kwargs)

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

Step 5: Training the Model

The model is compiled and trained using:

- **Loss Function**: Categorical crossentropy, as this is a multi-class classification problem.
- **Optimizer**: Adam, for efficient gradient updates.
- **Metrics**: Accuracy, to monitor the model's performance.

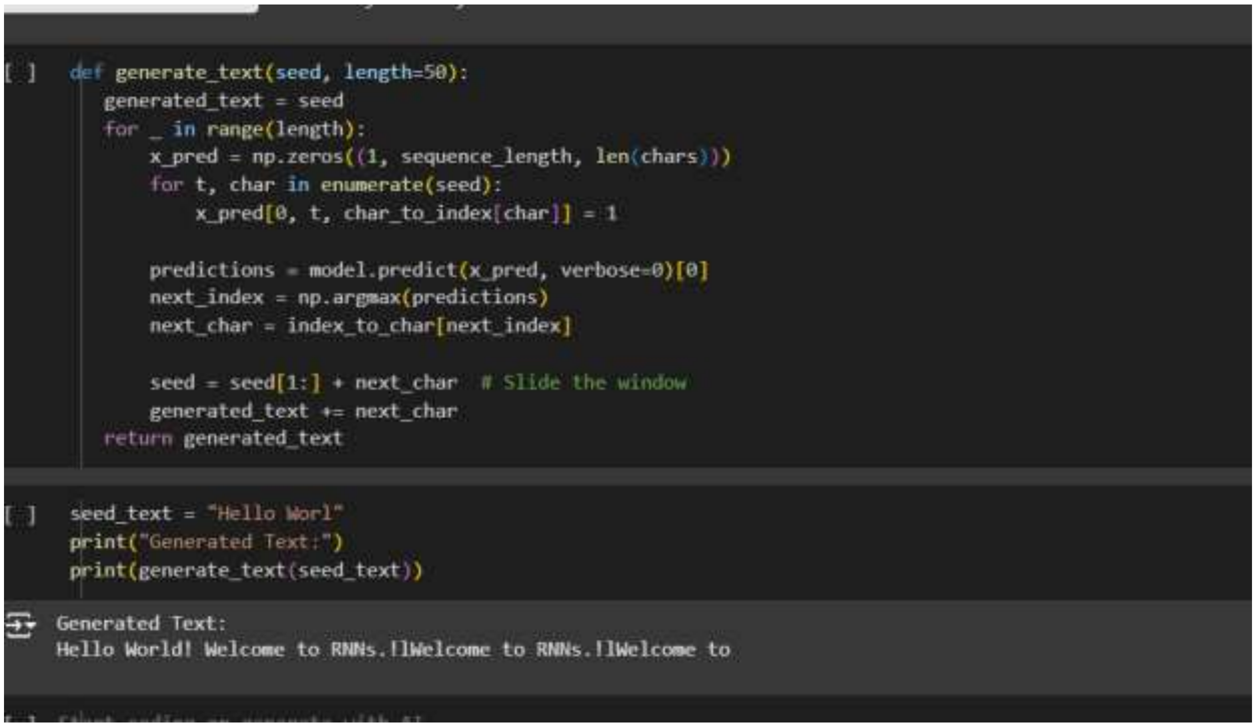


```
]  model.fit(X, y, epochs=5, batch_size=64)

Epoch 1/5
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - accuracy: 1.0000 - loss: 0.3048
Epoch 2/5
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - accuracy: 1.0000 - loss: 0.2648
Epoch 3/5
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 1.0000 - loss: 0.2306
Epoch 4/5
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - accuracy: 1.0000 - loss: 0.2014
Epoch 5/5
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - accuracy: 1.0000 - loss: 0.1764
<keras.src.callbacks.history.History at 0x79d3715af040>
```

Step 6: Generating Text

The trained model generates new text:

1. **Seed Input**: A substring of the original text is used as the starting point.
2. **Prediction**: The model predicts the next character repeatedly by sliding a window over the generated text.
3. **Output**: A sequence of generated text is returned.



```
def generate_text(seed, length=50):
    generated_text = seed
    for _ in range(length):
        x_pred = np.zeros((1, sequence_length, len(chars)))
        for t, char in enumerate(seed):
            x_pred[0, t, char_to_index[char]] = 1

        predictions = model.predict(x_pred, verbose=0)[0]
        next_index = np.argmax(predictions)
        next_char = index_to_char[next_index]

        seed = seed[1:] + next_char  # Slide the window
        generated_text += next_char
    return generated_text
```

```
seed_text = "Hello Worl"
print("Generated Text:")
print(generate_text(seed_text))
```

```
Generated Text:
Hello World! Welcome to RNNs.!lWelcome to RNNs.!lWelcome to
```

Deep Learning

Lab 12+13

Nayyab Malik
(BSAI-127)

Assigned By

Mam Iqra Nasem

Faculty of Engineering & CS

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

Submission Date: 31th October, 2024

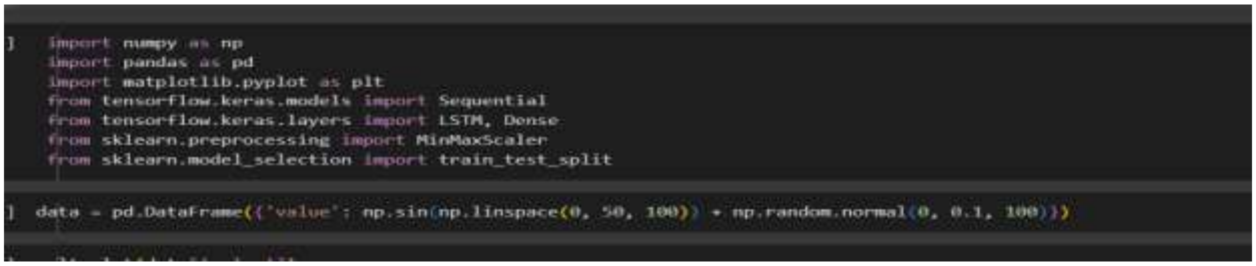
LSTM

This project demonstrates how to use a Long Short-Term Memory (LSTM) model for time series forecasting. The data is processed, normalized, and used to train an LSTM network, which predicts future values in the sequence. The training process is evaluated through loss curves, and the model's performance is visualized by comparing predicted values with true values.

Step 1: Data Preparation

We start with generating a synthetic time series dataset:

- The dataset consists of sine wave values with added random noise to simulate real-world data variations.
- The time series data is visualized using Matplotlib to ensure its integrity.

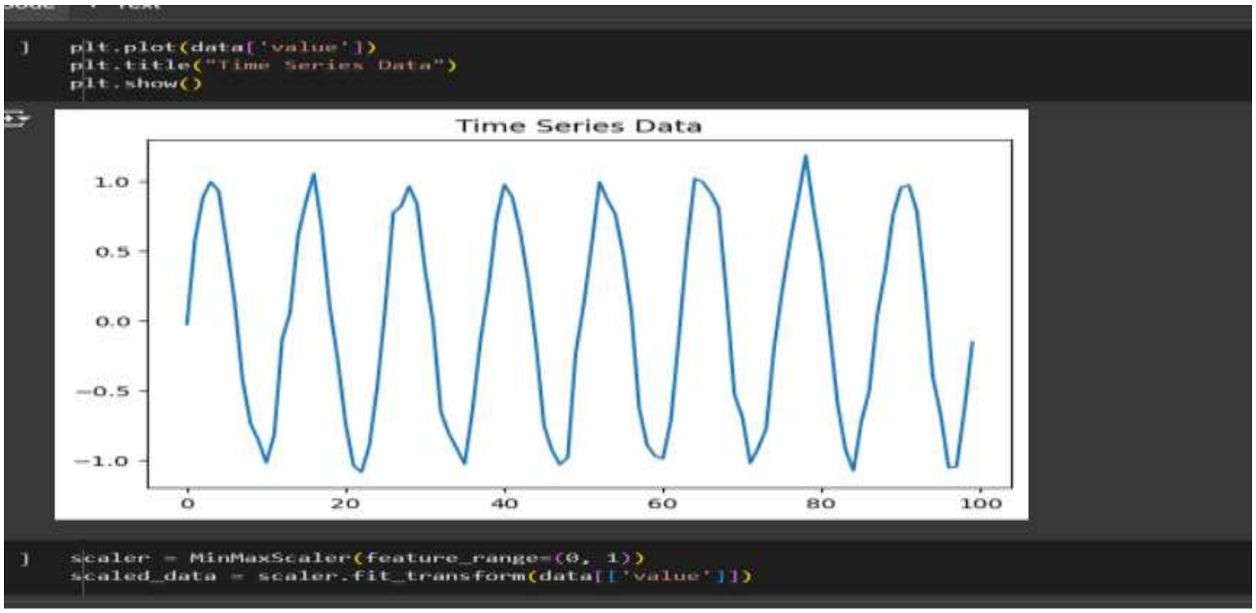


```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
```

```
data = pd.DataFrame({'value': np.sin(np.linspace(0, 50, 100)) + np.random.normal(0, 0.1, 100)})
```

Step 2: Data Normalization

The time series data is scaled to a range of 0 to 1 using the **MinMaxScaler**. This normalization is crucial for training the LSTM model efficiently, as it helps the network converge faster and perform better.



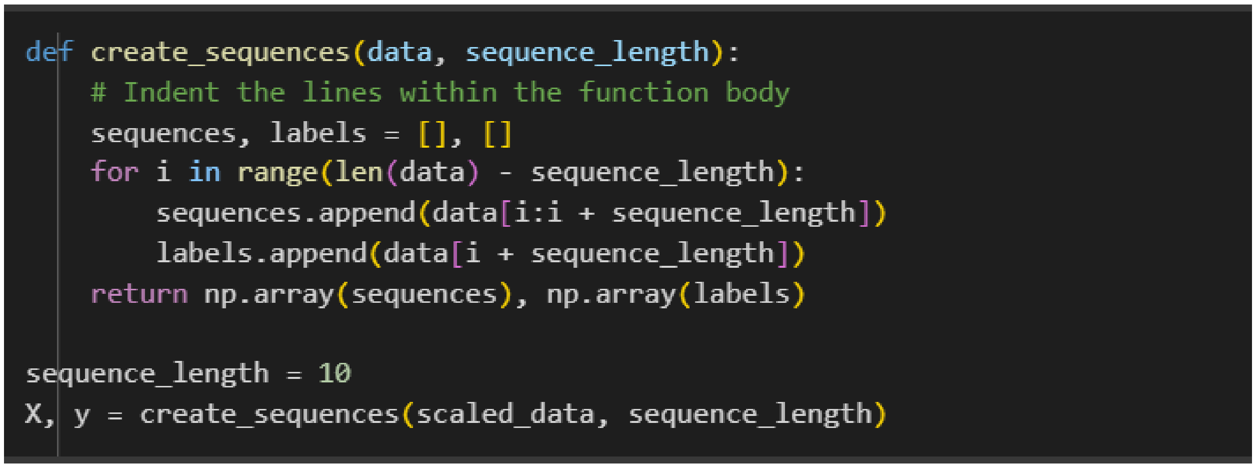
```
plt.plot(data['value'])
plt.title("Time Series Data")
plt.show()
```

```
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data[['value']])
```

Step 3: Sequence Creation

To prepare the data for the LSTM, sequences of fixed length are created:

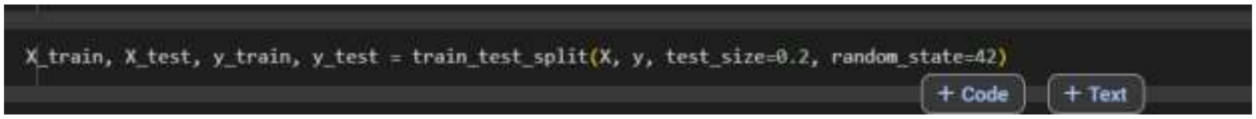
- Each sequence contains a fixed number of time steps (sequence_length) as input.
- The target value is the next value in the sequence.



```python
def create_sequences(data, sequence_length):
    # Indent the lines within the function body
    sequences, labels = [], []
    for i in range(len(data) - sequence_length):
        sequences.append(data[i:i + sequence_length])
        labels.append(data[i + sequence_length])
    return np.array(sequences), np.array(labels)

sequence_length = 10
X, y = create_sequences(scaled_data, sequence_length)
```

Step 4: Splitting Data

The dataset is split into training and testing sets using an 80-20 split. This ensures the model is evaluated on unseen data for reliable performance metrics.

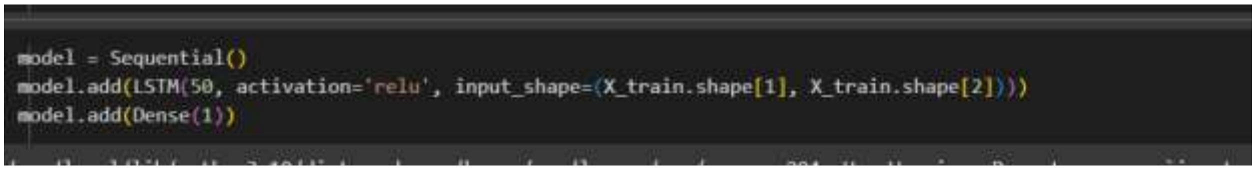


```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 5: Building the LSTM Model

The LSTM model is designed using Keras:

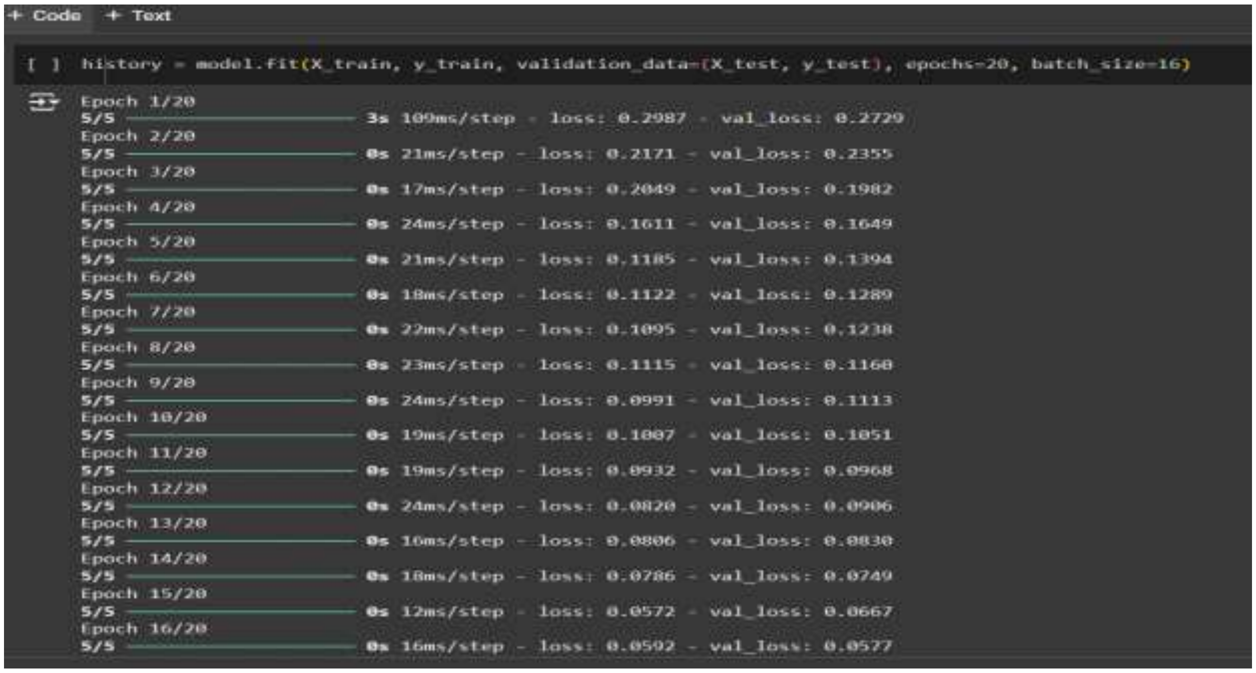
- **LSTM Layer**: Extracts temporal patterns from the sequence data.
- **Dense Layer**: Outputs a single value, representing the next time step in the sequence.



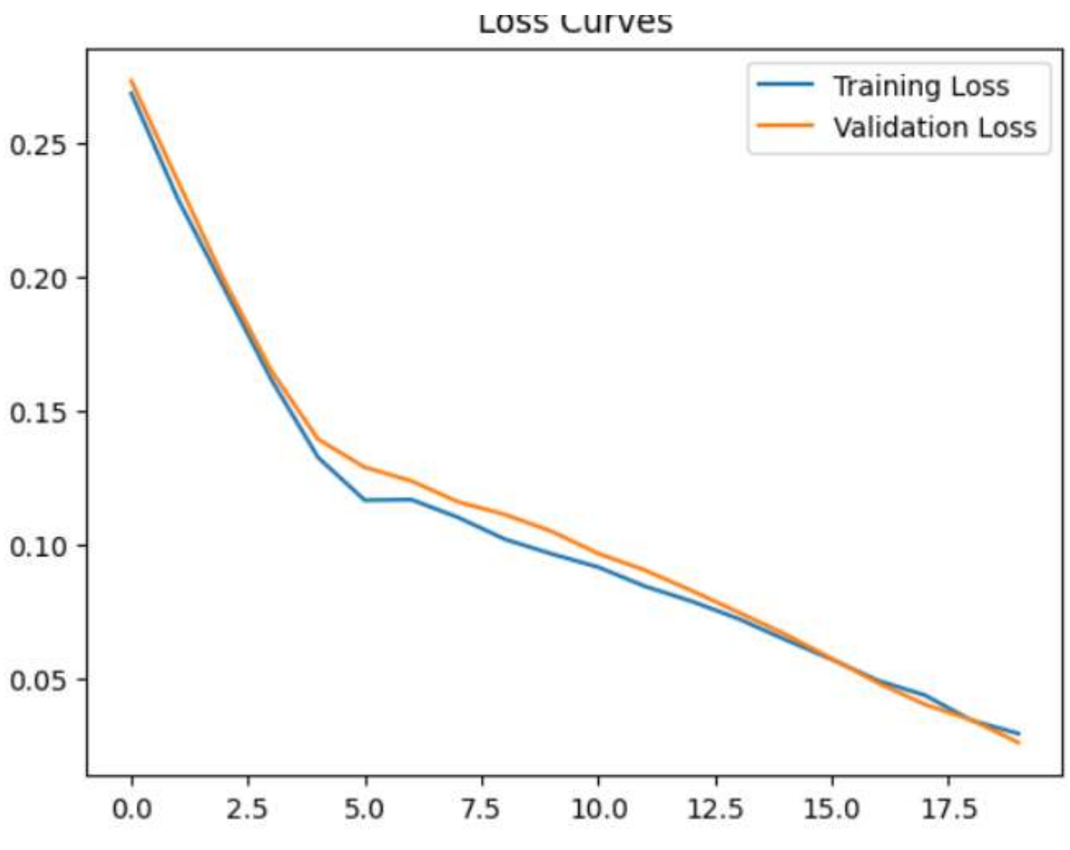
```python
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(X_train.shape[1], X_train.shape[2])))
model.add(Dense(1))
```

Step 6: Training the Model

The model is trained for 20 epochs with a batch size of 16. The training and validation loss are recorded to monitor the model's performance.



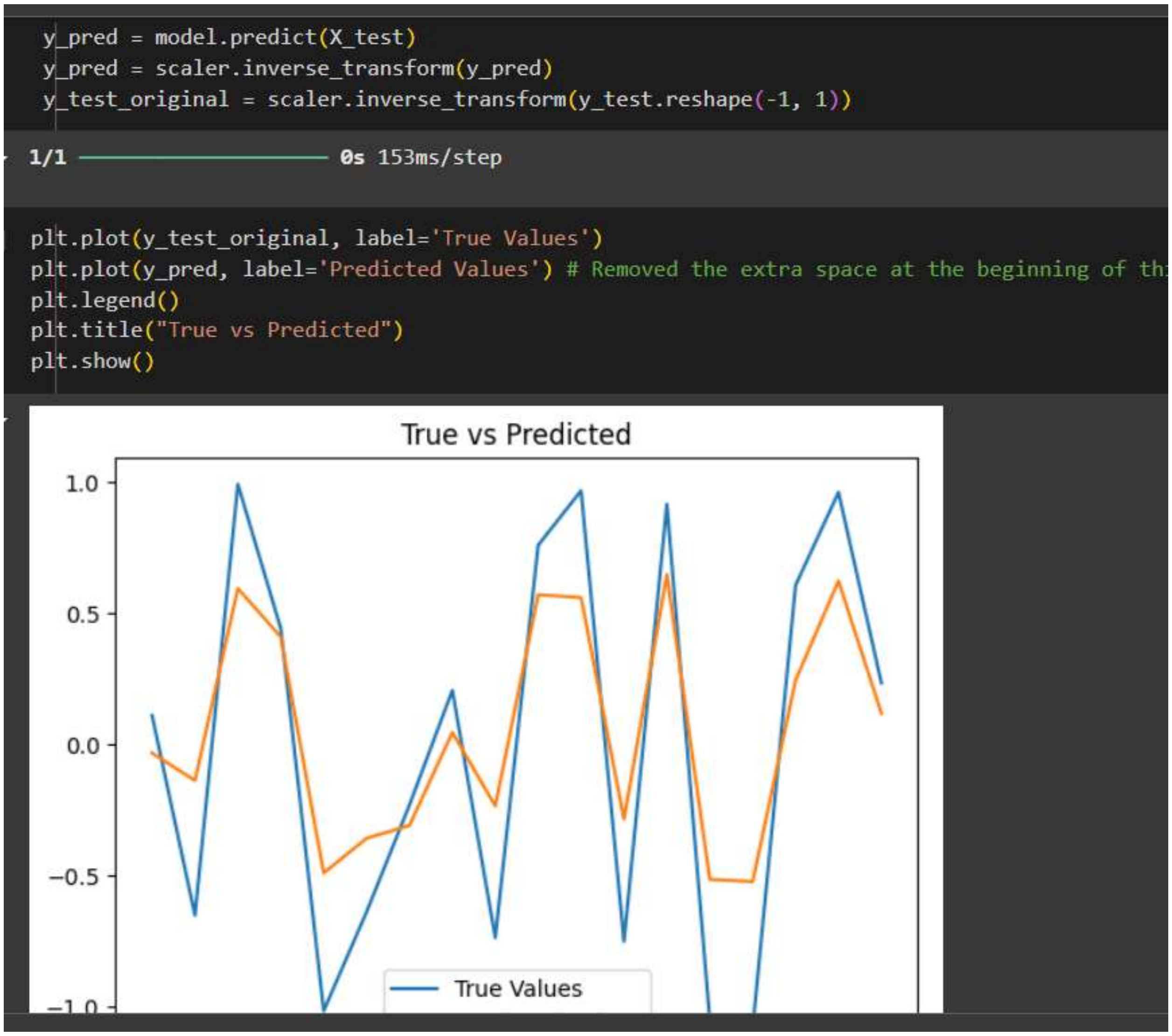
```
[ ] history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=20, batch_size=16)

Epoch 1/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 3s 109ms/step - loss: 0.2987 - val_loss: 0.2729
Epoch 2/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - loss: 0.2171 - val_loss: 0.2355
Epoch 3/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - loss: 0.2049 - val_loss: 0.1982
Epoch 4/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - loss: 0.1611 - val_loss: 0.1649
Epoch 5/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - loss: 0.1185 - val_loss: 0.1394
Epoch 6/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - loss: 0.1122 - val_loss: 0.1289
Epoch 7/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - loss: 0.1095 - val_loss: 0.1238
Epoch 8/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 0.1115 - val_loss: 0.1160
Epoch 9/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - loss: 0.0991 - val_loss: 0.1113
Epoch 10/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - loss: 0.1007 - val_loss: 0.1051
Epoch 11/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - loss: 0.0932 - val_loss: 0.0968
Epoch 12/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - loss: 0.0820 - val_loss: 0.0906
Epoch 13/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - loss: 0.0806 - val_loss: 0.0830
Epoch 14/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - loss: 0.0786 - val_loss: 0.0749
Epoch 15/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.0572 - val_loss: 0.0667
Epoch 16/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - loss: 0.0592 - val_loss: 0.0577
```



Step 8: Making Predictions

The trained model is used to predict the values for the test set:

- Predictions are scaled back to the original range using the inverse transform of the scaler.
- Predicted values are compared with true values to evaluate model performance.



```
y_pred = model.predict(X_test)
y_pred = scaler.inverse_transform(y_pred)
y_test_original = scaler.inverse_transform(y_test.reshape(-1, 1))
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 153ms/step
```

```
plt.plot(y_test_original, label='True Values')
plt.plot(y_pred, label='Predicted Values') # Removed the extra space at the beginning of thi
plt.legend()
plt.title("True vs Predicted")
plt.show()
```