Develop a neural network using backpropagation to classify images from the CIFAR-10

dataset. The dataset contains 60,000 32x32 color images divided into 10 classes (airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks). Your objective is to build a neural network model, train it using backpropagation, and evaluate its performance.

1. Data Preprocessing:

o Load the CIFAR-10 dataset. o Perform necessary data preprocessing steps: ▪ Normalize pixel values to range between 0 and 1. ▪ Convert class labels into one-hot encoded format. ▪ Split the dataset into training and test sets (e.g., 50,000 images for training and 10,000 for testing). ▪ Optionally, apply data augmentation techniques (such as random flips, rotations, or shifts) to improve the generalization of the model.

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


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


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


y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Data Augmentation
datagen = ImageDataGenerator(
    horizontal_flip=True,
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1
)
datagen.fit(x_train)
```

Network Architecture Design:

o Design a feedforward neural network to classify the images. ▪ Input Layer: The input shape should match the 32x32x3 dimensions of the CIFAR-10 images. ▪ Hidden Layers: Use appropriate layers. ▪ Output Layer: The final layer should have 10 output neurons (one for each class) with a softmax activation function for multi-class classification.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten


model = Sequential()


model.add(Flatten(input_shape=(32, 32, 3)))

# Hidden Layers: Two dense layers with ReLU and tanh activation functions
model.add(Dense(512, activation='relu'))  # First hidden layer
model.add(Dense(256, activation='tanh'))  # Second hidden layer

# Output Layer: 10 neurons with softmax activation for multi-class classification
model.add(Dense(10, activation='softmax'))


model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_c
  super().__init__(**kwargs)
Model: "sequential"
```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 3072)	0
dense (Dense)	(None, 512)	1,573,376
dense_1 (Dense)	(None, 256)	131,328
dense_2 (Dense)	(None, 10)	2,570

```
Total params: 1,707,274 (6.51 MB)
Trainable params: 1,707,274 (6.51 MB)
Non-trainable params: 0 (0.00 B)
```

Question:

o Justify your choice of network architecture, including the number of layers, types of layers, and the number of neurons/filters in each layer.

Input Layer: Flattened 32x32x3 images into a 1D vector. Hidden Layers: First hidden layer with 512 neurons to capture higher-level patterns. Second hidden layer with 256 neurons. The choice of neurons provides a balance between capacity and computational efficiency.

Activation Functions ReLU is used in the first hidden layer for fast convergence and to avoid the vanishing gradient problem.Helps speed up learning and is computationally efficient. Tanh is used in the second layer to introduce non-linearity and smooth gradient updates.it is Good for handling hidden layers where we want to balance positive and negative activations.

Loss Function and Optimizer

o Use any two loss functions and compare with the categorical cross entropy since this is a multi-class classification problem. o Select an appropriate optimizer (e.g., SGD, Adam, RMSprop) and explain how the learning rate affects the backpropagation process.

```
# Compile the model with categorical cross-entropy loss and Adam optimizer
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
from tensorflow.keras.callbacks import ReduceLROnPlateau

reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.001)

# Train the model with the learning rate scheduler
model.fit(
    datagen.flow(x_train, y_train, batch_size=64),
    epochs=10,
    validation_data=(x_test, y_test),
    callbacks=[reduce_lr]
)
```

```
Epoch 1/10
/usr/local/lib/python3.10/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` cl
  self._warn_if_super_not_called()
782/782 ━━━━━━━━━━━━━━━━━━━━ 58s 71ms/step - accuracy: 0.1903 - loss: 2.2249 - val_accuracy: 0.3379 - val_loss: 1.8399 - learning_ra
Epoch 2/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 83s 73ms/step - accuracy: 0.3300 - loss: 1.8530 - val_accuracy: 0.3708 - val_loss: 1.7261 - learning_ra
Epoch 3/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 82s 73ms/step - accuracy: 0.3745 - loss: 1.7381 - val_accuracy: 0.4277 - val_loss: 1.5994 - learning_ra
Epoch 4/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 57s 72ms/step - accuracy: 0.4024 - loss: 1.6627 - val_accuracy: 0.4308 - val_loss: 1.5885 - learning_ra
Epoch 5/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 85s 76ms/step - accuracy: 0.4170 - loss: 1.6239 - val_accuracy: 0.4496 - val_loss: 1.5298 - learning_ra
Epoch 6/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 79s 73ms/step - accuracy: 0.4233 - loss: 1.6028 - val_accuracy: 0.4718 - val_loss: 1.4933 - learning_ra
Epoch 7/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 58s 74ms/step - accuracy: 0.4307 - loss: 1.5779 - val_accuracy: 0.4653 - val_loss: 1.4942 - learning_ra
Epoch 8/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 80s 72ms/step - accuracy: 0.4405 - loss: 1.5565 - val_accuracy: 0.4704 - val_loss: 1.4908 - learning_ra
Epoch 9/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 84s 74ms/step - accuracy: 0.4470 - loss: 1.5401 - val_accuracy: 0.4805 - val_loss: 1.4490 - learning_ra
Epoch 10/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 56s 71ms/step - accuracy: 0.4539 - loss: 1.5253 - val_accuracy: 0.4699 - val_loss: 1.4645 - learning_ra
<keras.src.callbacks.history.History at 0x7a69409b94b0>
```

Loss Function:Categorical cross-entropy is ideal for multi-class classification.

Optimizer: Adam is chosen for its adaptive learning rates and efficient convergence.

Training the Model

```
history = model.fit(datagen.flow(x_train, y_train, batch_size=64),
                    validation_data=(x_test, y_test),
                    epochs=50,
                    verbose=1)
```

```
Epoch 1/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 58s 73ms/step - accuracy: 0.4546 - loss: 1.5221 - val_accuracy: 0.4827 - val_loss: 1.4443
Epoch 2/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 56s 72ms/step - accuracy: 0.4581 - loss: 1.5059 - val_accuracy: 0.4781 - val_loss: 1.4714
Epoch 3/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 56s 72ms/step - accuracy: 0.4644 - loss: 1.4901 - val_accuracy: 0.4941 - val_loss: 1.4314
Epoch 4/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 83s 73ms/step - accuracy: 0.4694 - loss: 1.4709 - val_accuracy: 0.4927 - val_loss: 1.4153
Epoch 5/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 81s 72ms/step - accuracy: 0.4676 - loss: 1.4807 - val_accuracy: 0.4827 - val_loss: 1.4380
Epoch 6/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 59s 75ms/step - accuracy: 0.4767 - loss: 1.4559 - val_accuracy: 0.4991 - val_loss: 1.3981
Epoch 7/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 81s 74ms/step - accuracy: 0.4794 - loss: 1.4609 - val_accuracy: 0.4946 - val_loss: 1.4024
Epoch 8/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 57s 72ms/step - accuracy: 0.4783 - loss: 1.4549 - val_accuracy: 0.5105 - val_loss: 1.3700
Epoch 9/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 56s 71ms/step - accuracy: 0.4787 - loss: 1.4514 - val_accuracy: 0.5023 - val_loss: 1.3867
Epoch 10/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 58s 74ms/step - accuracy: 0.4817 - loss: 1.4357 - val_accuracy: 0.4894 - val_loss: 1.4182
Epoch 11/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 82s 74ms/step - accuracy: 0.4853 - loss: 1.4386 - val_accuracy: 0.5072 - val_loss: 1.3906
Epoch 12/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 57s 73ms/step - accuracy: 0.4885 - loss: 1.4319 - val_accuracy: 0.4999 - val_loss: 1.4036
Epoch 13/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 57s 72ms/step - accuracy: 0.4890 - loss: 1.4234 - val_accuracy: 0.5028 - val_loss: 1.3694
Epoch 14/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 83s 74ms/step - accuracy: 0.4903 - loss: 1.4267 - val_accuracy: 0.4947 - val_loss: 1.4145
Epoch 15/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 81s 72ms/step - accuracy: 0.4895 - loss: 1.4222 - val_accuracy: 0.5015 - val_loss: 1.3950
Epoch 16/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 84s 74ms/step - accuracy: 0.4886 - loss: 1.4228 - val_accuracy: 0.5126 - val_loss: 1.3641
Epoch 17/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 80s 71ms/step - accuracy: 0.4934 - loss: 1.4078 - val_accuracy: 0.5051 - val_loss: 1.3746
Epoch 18/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 82s 72ms/step - accuracy: 0.4952 - loss: 1.4033 - val_accuracy: 0.5168 - val_loss: 1.3595
Epoch 19/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 82s 73ms/step - accuracy: 0.4936 - loss: 1.4128 - val_accuracy: 0.5011 - val_loss: 1.3814
Epoch 20/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 57s 73ms/step - accuracy: 0.4955 - loss: 1.4092 - val_accuracy: 0.5141 - val_loss: 1.3503
Epoch 21/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 83s 75ms/step - accuracy: 0.4993 - loss: 1.3944 - val_accuracy: 0.5073 - val_loss: 1.3834
Epoch 22/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 57s 73ms/step - accuracy: 0.4993 - loss: 1.3945 - val_accuracy: 0.5075 - val_loss: 1.3732
Epoch 23/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 81s 72ms/step - accuracy: 0.5023 - loss: 1.3840 - val_accuracy: 0.5177 - val_loss: 1.3418
Epoch 24/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 83s 73ms/step - accuracy: 0.4983 - loss: 1.3916 - val_accuracy: 0.5172 - val_loss: 1.3516
Epoch 25/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 81s 71ms/step - accuracy: 0.5013 - loss: 1.3914 - val_accuracy: 0.5213 - val_loss: 1.3431
Epoch 26/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 83s 73ms/step - accuracy: 0.4998 - loss: 1.3937 - val_accuracy: 0.5211 - val_loss: 1.3447
Epoch 27/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 56s 71ms/step - accuracy: 0.5091 - loss: 1.3836 - val_accuracy: 0.5119 - val_loss: 1.3608
Epoch 28/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 57s 73ms/step - accuracy: 0.5004 - loss: 1.3818 - val_accuracy: 0.5270 - val_loss: 1.3335
Epoch 29/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 58s 74ms/step - accuracy: 0.5078 - loss: 1.3745 - val_accuracy: 0.5233 - val_loss: 1.3441
```

NBackpropagation updates weights by calculating gradients of the loss function and adjusting the weights via gradient descent.

Learning Rate: If the model is not converging, reducing the learning rate could allow smaller, more precise updates.

Model Evaluation

```
# Evaluate model on the test set
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_accuracy * 100:.2f}%')

# Generate predictions and calculate classification metrics
from sklearn.metrics import classification_report, confusion_matrix

y_pred = model.predict(x_test)
y_pred_classes = y_pred.argmax(axis=1)
y_true = y_test.argmax(axis=1)

print(classification_report(y_true, y_pred_classes))
conf_matrix = confusion_matrix(y_true, y_pred_classes)
print(conf_matrix)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5231 - loss: 1.3327
Test accuracy: 51.96%
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step
              precision    recall  f1-score   support

           0       0.68      0.48      0.56      1000
           1       0.61      0.69      0.65      1000
           2       0.42      0.41      0.41      1000
           3       0.37      0.37      0.37      1000
           4       0.52      0.35      0.42      1000
           5       0.39      0.47      0.43      1000
           6       0.54      0.61      0.57      1000
           7       0.66      0.49      0.56      1000
           8       0.59      0.69      0.64      1000
           9       0.49      0.63      0.55      1000

    accuracy                           0.52     10000
   macro avg       0.53      0.52      0.52     10000
weighted avg       0.53      0.52      0.52     10000

[[482  51  61  31  21  27  26  23 185  93]
 [ 20 691  11  20   4  18  15   6  50 165]
 [ 60  27 407  92  95 124  97  36  34  28]
 [ 18  23  75 369  23 236 121  30  31  74]
 [ 36  16 162  74 355  82 134  72  44  25]
 [  9   8  76 208  35 468  64  47  33  52]
 [  2  15  86  79  71  80 611  12  13  31]
 [ 15  26  80  62  53 119  36 489  26  94]
 [ 45  80  11  28  16  25  13   5 694  83]
 [ 23 188  10  25   6  12  25  19  62 630]]
```

Optimization Strategies

Early Stopping: Monitor validation loss and stop training if no improvement is seen.

Learning Rate Scheduling: Adjust the learning rate dynamically during training to fine-tune convergence.

Weight Initialization: Proper initialization prevents slow convergence and vanishing/exploding gradients. it is crucial in neural networks because it significantly affects how well and how quickly a model converges during training.

```
#early stopping and learning rate reduction
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

# Early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=5)

# Reduce learning rate when a plateau in validation loss is detected
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3)

# Train the model with callbacks
```

```
history = model.fit(datagen.flow(x_train, y_train, batch_size=64),
                    validation_data=(x_test, y_test),
                    epochs=20,
                    callbacks=[early_stopping, reduce_lr],
                    verbose=1)
```

```
Epoch 1/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 75s 95ms/step - accuracy: 0.5629 - loss: 1.2308 - val_accuracy: 0.5573 - val_loss: 1.2479 - learning_ra
Epoch 2/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 58s 74ms/step - accuracy: 0.5625 - loss: 1.2311 - val_accuracy: 0.5571 - val_loss: 1.2562 - learning_ra
Epoch 3/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 58s 74ms/step - accuracy: 0.5641 - loss: 1.2280 - val_accuracy: 0.5589 - val_loss: 1.2534 - learning_ra
Epoch 4/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 82s 74ms/step - accuracy: 0.5580 - loss: 1.2402 - val_accuracy: 0.5600 - val_loss: 1.2493 - learning_ra
Epoch 5/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 82s 74ms/step - accuracy: 0.5629 - loss: 1.2229 - val_accuracy: 0.5613 - val_loss: 1.2429 - learning_ra
Epoch 6/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 58s 74ms/step - accuracy: 0.5617 - loss: 1.2283 - val_accuracy: 0.5618 - val_loss: 1.2430 - learning_ra
Epoch 7/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 59s 76ms/step - accuracy: 0.5612 - loss: 1.2270 - val_accuracy: 0.5610 - val_loss: 1.2419 - learning_ra
Epoch 8/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 82s 76ms/step - accuracy: 0.5642 - loss: 1.2207 - val_accuracy: 0.5610 - val_loss: 1.2446 - learning_ra
Epoch 9/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 57s 73ms/step - accuracy: 0.5655 - loss: 1.2283 - val_accuracy: 0.5617 - val_loss: 1.2452 - learning_ra
Epoch 10/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 57s 73ms/step - accuracy: 0.5663 - loss: 1.2211 - val_accuracy: 0.5622 - val_loss: 1.2437 - learning_ra
Epoch 11/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 81s 72ms/step - accuracy: 0.5657 - loss: 1.2266 - val_accuracy: 0.5617 - val_loss: 1.2421 - learning_ra
Epoch 12/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 59s 75ms/step - accuracy: 0.5683 - loss: 1.2138 - val_accuracy: 0.5629 - val_loss: 1.2412 - learning_ra
Epoch 13/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 59s 75ms/step - accuracy: 0.5682 - loss: 1.2190 - val_accuracy: 0.5642 - val_loss: 1.2417 - learning_ra
Epoch 14/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 58s 74ms/step - accuracy: 0.5686 - loss: 1.2076 - val_accuracy: 0.5618 - val_loss: 1.2426 - learning_ra
Epoch 15/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 58s 74ms/step - accuracy: 0.5659 - loss: 1.2219 - val_accuracy: 0.5622 - val_loss: 1.2410 - learning_ra
Epoch 16/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 59s 75ms/step - accuracy: 0.5731 - loss: 1.2085 - val_accuracy: 0.5629 - val_loss: 1.2439 - learning_ra
Epoch 17/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 83s 76ms/step - accuracy: 0.5718 - loss: 1.2125 - val_accuracy: 0.5594 - val_loss: 1.2432 - learning_ra
Epoch 18/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 83s 76ms/step - accuracy: 0.5678 - loss: 1.2184 - val_accuracy: 0.5607 - val_loss: 1.2415 - learning_ra
Epoch 19/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 59s 75ms/step - accuracy: 0.5666 - loss: 1.2126 - val_accuracy: 0.5621 - val_loss: 1.2406 - learning_ra
Epoch 20/20
782/782 ━━━━━━━━━━━━━━━━━━━━ 59s 75ms/step - accuracy: 0.5719 - loss: 1.2149 - val_accuracy: 0.5621 - val_loss: 1.2411 - learning_ra
```

```
import numpy as np
import matplotlib.pyplot as plt
from random import randint

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

num_images = 5
random_indices = np.random.choice(x_test.shape[0], num_images, replace=False)
random_images = x_test[random_indices]
random_labels = y_test[random_indices]

predictions = model.predict(random_images)

# the images with predicted and true labels
for i, idx in enumerate(random_indices):
    plt.figure(figsize=(2, 2))
    plt.imshow(random_images[i])

    predicted_label = np.argmax(predictions[i])
    true_label = np.argmax(random_labels[i])

    plt.title(f"True: {class_names[true_label]} | Predicted: {class_names[predicted_label]}")
    plt.axis('off')
    plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step
```

True: ship | Predicted: airplane

True: dog | Predicted: dog

True: airplane | Predicted: airplane

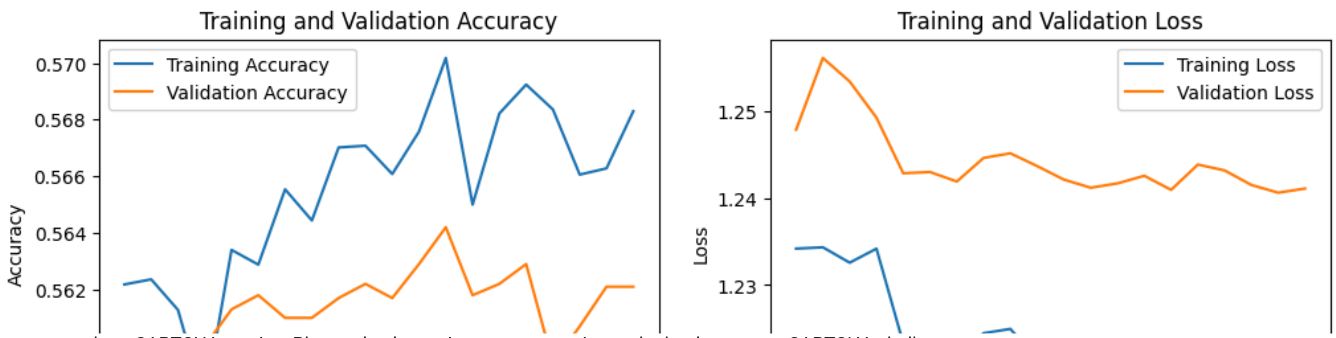
True: cat | Predicted: frog

True: airplane | Predicted: ship

```
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.show()
```


Training and Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy
0.570
0.568
0.566
0.564
0.562
Training and Validation Loss
Training Loss
Validation Loss
Loss
1.25
1.24
1.23