## Question 3

```
import pandas as pd
from sklearn.utils import shuffle
train_data = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/spect/SPECT.train', header=None)
train_data.columns = ['Label']+[ 'F'+str(i) for i in range(1,23)]
train_data = shuffle(train_data) # shuffle data
print(train_data)
print("Train shape:", train_data.shape)
```

```
     Label  F1  F2  F3  F4  F5  F6  F7  F8  F9  ...  F13  F14  F15  F16  F17  \
14       1   1   0   1   1   0   0   1   1   1  ...    1    1    1    1    0
24       1   0   0   0   0   0   1   0   0   0  ...    0    0    0    0    1
67       0   0   0   0   0   0   0   0   0   0  ...    0    0    0    0    0
56       0   0   0   0   0   0   0   0   0   0  ...    0    0    0    0    0
73       0   1   0   1   0   0   0   0   1   0  ...    1    0    0    0    0
..     ...  ..  ..  ..  ..  ..  ..  ..  ..  ..  ...  ...  ...  ...  ...  ...
70       0   0   0   1   0   0   0   0   1   0  ...    0    0    0    0    0
17       1   0   0   0   0   0   0   1   0   0  ...    0    0    0    0    0
37       1   1   0   0   1   0   0   0   0   0  ...    0    0    0    0    0
26       1   0   0   0   1   0   0   1   0   1  ...    0    1    0    0    0
4        1   0   0   0   0   0   0   0   1   0  ...    1    0    1    1    0

    F18  F19  F20  F21  F22
14    1    1    0    1    1
24    0    1    0    0    0
67    0    0    0    0    0
56    0    1    1    0    0
73    0    0    0    1    0
..  ...  ...  ...  ...  ...
70    0    0    0    0    0
17    0    0    0    0    1
37    0    0    0    0    0
26    0    0    0    1    0
4     0    0    0    0    0

[80 rows x 23 columns]
Train shape: (80, 23)
```

```
# practice 1.2: Load testing data
import pandas as pd
test_data = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/spect/SPECT.test', header=None)
test_data.columns = ['Label']+[ 'F'+str(i) for i in range(1,23)]
print(test_data)
print("Test shape:", test_data.shape)
```

```
     Label  F1  F2  F3  F4  F5  F6  F7  F8  F9  ...  F13  F14  F15  F16  F17  \
0        1   1   0   0   1   1   0   0   0   1  ...    0    1    1    1    0
1        1   1   0   0   1   1   0   0   0   0  ...    1    0    0    0    0
2        1   0   0   0   1   0   1   0   0   1  ...    0    1    1    0    0
3        1   0   1   1   1   0   0   1   0   1  ...    1    1    0    1    0
4        1   0   0   1   0   0   0   0   1   0  ...    1    1    0    1    0
..     ...  ..  ..  ..  ..  ..  ..  ..  ..  ..  ...  ...  ...  ...  ...  ...
182      0   0   0   0   0   0   0   0   0   0  ...    0    0    0    0    0
183      0   1   1   0   0   0   1   0   0   0  ...    0    0    0    1    0
184      0   1   0   1   0   1   0   0   1   0  ...    1    0    1    1    0
185      0   1   0   1   0   1   0   0   1   1  ...    0    1    0    1    0
186      0   0   0   0   0   0   0   0   0   0  ...    0    0    0    0    0

     F18  F19  F20  F21  F22
0      0    1    1    0    0
1      0    0    0    0    0
2      0    0    0    0    1
3      0    0    0    1    0
4      0    0    0    0    1
..   ...  ...  ...  ...  ...
182    0    0    0    0    0
183    0    0    0    0    0
184    0    0    0    0    0
185    0    0    0    0    0
186    0    0    0    0    0

[187 rows x 23 columns]
Test shape: (187, 23)
```

```
# practice 1.3: import Sequential class and Dense class from keras
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Dense

# practice 1.4: build a three-layer neural network with binary function as output
model = Sequential()
model.add()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-6a649a001f9e> in <cell line: 0>()
      5 # practice 1.4: build a three-layer neural network with binary function as output
      6 model = Sequential()
----> 7 model.add()

TypeError: Sequential.add() missing 1 required positional argument: 'layer'
```

Next steps: Explain error

The Error arises because the Sequential model's add() method requires that a layer (such as Dense or Input) be given as an argument. You cannot use model.add() without providing a layer. To Fix the Error, add a layer (such as Dense or Input) inside the model. add(). To define the input shape in the first layer, use the Input() layer; in subsequent layers, use Dense.

## Question 4

```
from keras.models import Sequential
from keras.layers import Dense

# Initialize the Sequential model
model = Sequential()

# Add the first hidden layer (input layer -> hidden layer 1)
model.add(Dense(100, input_shape=(22,), activation='relu'))

# Add the second hidden layer (hidden layer 1 -> hidden layer 2)
model.add(Dense(60, activation='relu'))

# Add the output layer (hidden layer 2 -> output layer)
model.add(Dense(12, activation='sigmoid'))

# Print model summary to verify the architecture
model.summary()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential_2"
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| dense_2 (Dense) | (None, 100) | 2,300 |
| dense_3 (Dense) | (None, 60) | 6,060 |
| dense_4 (Dense) | (None, 12) | 732 |

```
Total params: 9,092 (35.52 KB)
Trainable params: 9,092 (35.52 KB)
Non-trainable params: 0 (0.00 B)
```

## Question 5

```
## Practice 2.1: extract features and labels from the training/test data
train_label = train_data['Label']
train_features = train_data.drop('Label',axis= 1)
print("train_features.shape:",train_features.shape)

test_label = test_data['Label']
test_features = test_data.drop('Label',axis=1)
print("test_features.shape:",test_features.shape)
```

```
train_features.shape: (80, 22)
test_features.shape: (187, 22)
```

```
## Practice 2.2: Define the first hidden layer in model architecture
from keras.models import Sequential
from keras.layers import Dense
model = Sequential() # initialize Sequential model

# add connection layers into sequential model
model.add(Dense(100, input_shape=(22,),activation='relu')) # input layer -> hidden layer 1

model.summary()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential_3"
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| dense_5 (Dense) | (None, 100) | 2,300 |

```
Total params: 2,300 (8.98 KB)
Trainable params: 2,300 (8.98 KB)
Non-trainable params: 0 (0.00 B)
```

## Question 6

```
## Practice 3: Add more hidden layers and output layers into network
from keras.models import Sequential
from keras.layers import Dense

model = Sequential() # initialize Sequential model
model.add(Dense(100, input_shape=(22,), activation='relu')) # input layer -> hidden layer 1
model.add(Dense(100, activation='relu')) # hidden layer 1 -> hidden layer 2
model.add(Dense(1, activation='sigmoid')) # hidden layer 2 -> output layer
model.summary()
```

```
Model: "sequential_4"
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| dense_6 (Dense) | (None, 100) | 2,300 |
| dense_7 (Dense) | (None, 100) | 10,100 |
| dense_8 (Dense) | (None, 1) | 101 |

```
Total params: 12,501 (48.83 KB)
Trainable params: 12,501 (48.83 KB)
Non-trainable params: 0 (0.00 B)
```

A neural network's parameters include: Weights represent the connections between neurons in two levels. Biases: A value assigned to each neuron. Formula for calculating layer parameters: Parameters = (number of inputs x number of neurons) plus number of neurons. Parameters = (Number of inputs x Number of neurons) + Number of neurons. Where: The number of inputs represents the number of features or neurons in the previous layer. The number of neurons in the current layer. First hidden layer: 2300 parameters. Second hidden layer: 10100 parameters. Output layer: 101 parameters.

## Practice 4

```
## Practice 4.1: Compile the model for backpropagation, and start training model on data
from keras.models import Sequential
from keras.layers import Dense

model = Sequential() # initialize Sequential model
model.add(Dense(100, input_shape=(22,), activation='relu')) # input layer -> hidden layer 1
model.add(Dense(100, activation='relu')) # hidden layer 1 -> hidden layer 2
model.add(Dense(1, activation='sigmoid')) # hidden layer 2 -> output layer

# Compile model, use binary crossentropy, and stochastic gradient descent for optimization
model.compile(loss='binary_crossentropy', optimizer='SGD', metrics=['accuracy'])
```

```
model.summary()
```

```
Model: "sequential_5"
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| dense_9 (Dense) | (None, 100) | 2,300 |
| dense_10 (Dense) | (None, 100) | 10,100 |
| dense_11 (Dense) | (None, 1) | 101 |

```
Total params: 12,501 (48.83 KB)
Trainable params: 12,501 (48.83 KB)
Non-trainable params: 0 (0.00 B)
```

```
## Practice 4.2: start training the model on the features and labels, set the training epochs to 10
model.fit(train_features, train_label, epochs = 10)
```

```
Epoch 1/10
3/3 ━━━━━━━━━━━━━━━━━━━━ 2s 250ms/step - accuracy: 0.5555 - loss: 0.6817
Epoch 2/10
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.5547 - loss: 0.6811
Epoch 3/10
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.6234 - loss: 0.6720
Epoch 4/10
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.6195 - loss: 0.6737
Epoch 5/10
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.6703 - loss: 0.6722
Epoch 6/10
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.6609 - loss: 0.6679
Epoch 7/10
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.6414 - loss: 0.6697
Epoch 8/10
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.6477 - loss: 0.6684
Epoch 9/10
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.6359 - loss: 0.6641
Epoch 10/10
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.6438 - loss: 0.6640
<keras.src.callbacks.history.History at 0x7b10da343750>
```

## Question 7

```
## practice 5.1: save the model to local disk, or load the model for predictions
from tensorflow.keras.models import load_model
model.save("SPECT.h5")
model_loaded = load_model("SPECT.h5")
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file for
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty
```

```
## practice 5.2: evaluate the model performance using evaluate() method
model_loaded.evaluate(train_features, train_label)
```

```
3/3 ━━━━━━━━━━━━━━━━━━━━ 1s 89ms/step - accuracy: 0.6750 - loss: 0.6586
[0.6590842008590698, 0.6625000238418579]
```

```
## practice 5.3: make a prediction on training set and test set
train_predictions = model_loaded.predict(train_features)
test_predictions = model_loaded.predict(test_features)
print("train_predictions: ",train_predictions) # the prediction is the probability of the class
```

```
        [0.4964167 ]
        [0.5147285 ]
        [0.5400081 ]
        [0.4964167 ]
        [0.4964167 ]
        [0.5424427 ]
        [0.4964167 ]
        [0.54476887]
        [0.47709453]
        [0.5537737 ]
        [0.53083944]
        [0.54769915]
        [0.603406  ]
        [0.54176426]
        [0.53507304]
        [0.56876177]
        [0.6160022 ]
        [0.60874516]
        [0.558199  ]
        [0.53512293]
        [0.5708868 ]
        [0.5147285 ]
        [0.57267636]
        [0.5111822 ]
        [0.56217533]
        [0.5156064 ]
        [0.5339119 ]
        [0.4964167 ]
        [0.5209864 ]
        [0.56786966]
        [0.5629641 ]
        [0.5616284 ]
        [0.561867  ]
        [0.5147285 ]
        [0.49263886]
        [0.4964167 ]
        [0.54965323]
        [0.62348765]
        [0.5404273 ]
        [0.5113538 ]
        [0.54199237]
        [0.555644  ]
        [0.57405436]
        [0.5545455 ]]
```

```
## practice 5.4: convert probability to labels
import numpy as np
train_prediction_labels = (train_predictions > 0.5).astype(int)
test_prediction_labels = (test_predictions > 0.5).astype(int)
```

```
## practice 5.5: convert probability to labels
from sklearn.metrics import accuracy_score
print("Training accuracy: ", accuracy_score(train_label, train_prediction_labels))
print("Test accuracy: ", accuracy_score(test_label, test_prediction_labels))
```

```
Training accuracy:  0.6625
Test accuracy:  0.9197860962566845
```

```
from tensorflow.keras.models import load_model

# Save the model to a local file
model.save("Neural_network_model.h5")
from tensorflow.keras.models import load_model

# Save the model to a local file
model.save("Neural_network_model.h5")

# Load the model from the local file
loaded_model = load_model("Neural_network_model.h5")
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file for
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file for
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty
```

```
# Evaluate the model performance on the training data
train_loss, train_accuracy = model_loaded.evaluate(train_features, train_label)
print("Train Loss: ", train_loss)
print("Train Accuracy: ", train_accuracy)
```

```
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.6750 - loss: 0.6586
Train Loss:  0.6590842008590698
Train Accuracy:  0.6625000238418579
```

```
train_predictions = model_loaded.predict(train_features)
test_predictions = model_loaded.predict(test_features)
print("Train Predictions: ", train_predictions)  # Probabilities of the classes
```

```
[0.572078  ]
[0.55804163]
[0.5845689 ]
[0.5827429 ]
[0.5156064 ]
[0.5159332 ]
[0.5322987 ]
[0.5097352 ]
[0.5737541 ]
[0.4964167 ]
[0.48785743]
[0.5557354 ]
[0.5715316 ]
[0.52921414]
[0.4964167 ]
[0.5147285 ]
[0.5400081 ]
[0.4964167 ]
[0.4964167 ]
[0.5424427 ]
[0.4964167 ]
[0.54476887]
[0.47709453]
[0.5537737 ]
[0.53083944]
[0.54769915]
[0.603406  ]
[0.54176426]
[0.53507304]
[0.56876177]
[0.6160022 ]
[0.60874516]
[0.558199  ]
[0.53512293]
[0.5708868 ]
[0.5147285 ]
[0.57267636]
[0.5111822 ]
[0.56217533]
[0.5156064 ]
[0.5339119 ]
[0.4964167 ]
[0.5209864 ]
[0.56786966]
[0.5629641 ]
[0.5616284 ]
[0.561867  ]
[0.5147285 ]
[0.49263886]
[0.4964167 ]
[0.54965323]
[0.62348765]
[0.5404273 ]
[0.5113538 ]
[0.54199237]
[0.555644  ]
[0.57405436]
[0.5545455 ]]
```

```
train_prediction_labels = (train_predictions > 0.5).astype(int)
test_prediction_labels = (test_predictions > 0.5).astype(int)
```

```
from sklearn.metrics import accuracy_score

# Calculate accuracy for training set
train_accuracy = accuracy_score(train_label, train_prediction_labels)
print("Training accuracy: ", train_accuracy)

# Calculate accuracy for test set
test_accuracy = accuracy_score(test_label, test_prediction_labels)
print("Test accuracy: ", test_accuracy)
```

```
Training accuracy:  0.6625
Test accuracy:  0.9197860962566845
```

```
from sklearn.metrics import precision_score, recall_score, f1_score

# Calculate precision, recall, and F1-score for training set
train_precision = precision_score(train_label, train_prediction_labels)
train_recall = recall_score(train_label, train_prediction_labels)
train_f1 = f1_score(train_label, train_prediction_labels)

print("Training Precision: ", train_precision)
print("Training Recall: ", train_recall)
print("Training F1-Score: ", train_f1)

# Calculate precision, recall, and F1-score for test set
test_precision = precision_score(test_label, test_prediction_labels)
test_recall = recall_score(test_label, test_prediction_labels)
test_f1 = f1_score(test_label, test_prediction_labels)

print("Test Precision: ", test_precision)
print("Test Recall: ", test_recall)
print("Test F1-Score: ", test_f1)
```

```
Training Precision:  0.6065573770491803
Training Recall:  0.925
Training F1-Score:  0.7326732673267327
Test Precision:  0.9590643274853801
Test Recall:  0.9534883720930233
Test F1-Score:  0.956268221574344
```

## Question 8

```
# Import necessary modules and functions
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# Generate a binary classification dataset with 22 features
X, y = make_classification(n_samples=1000, n_features=22, n_informative=10, n_classes=2, random_state=42)

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

model = MLPClassifier(hidden_layer_sizes=(100, 100), max_iter=1000, activation='relu', solver='adam', random_state=42)

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

# Evaluate the model performance on the test set
y_pred_train = model.predict(X_train)
y_pred_test = model.predict(X_test)

# Calculate accuracy
train_accuracy = accuracy_score(y_train, y_pred_train)
test_accuracy = accuracy_score(y_test, y_pred_test)

# Print model parameters and accuracy results
print("Training Accuracy: ", train_accuracy)
print("Test Accuracy: ", test_accuracy)

# Get the model parameters for each layer (number of neurons, weights, biases)
print("\nModel parameters for each layer (conceptually named as dense_1, dense_2, dense_3):")
print(f"Layer dense_1 (Input -> Hidden 1): {model.coefs_[0].shape[0]} inputs, {model.coefs_[0].shape[1]} neurons")
print(f"Layer dense_2 (Hidden 1 -> Hidden 2): {model.coefs_[1].shape[0]} inputs, {model.coefs_[1].shape[1]} neurons")
print(f"Layer dense_3 (Hidden 2 -> Output): {model.coefs_[2].shape[0]} inputs, {model.coefs_[2].shape[1]} neurons")
```

```
Training Accuracy:  1.0
Test Accuracy:  0.94

Model parameters for each layer (conceptually named as dense_1, dense_2, dense_3):
Layer dense_1 (Input -> Hidden 1): 22 inputs, 100 neurons
Layer dense_2 (Hidden 1 -> Hidden 2): 100 inputs, 100 neurons
```

```
Layer dense_3 (Hidden 2 -> Output): 100 inputs, 1 neurons
```

```
# practice 6.1: Load training data
import pandas as pd
from sklearn.utils import shuffle
train_data = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/spect/SPECT.train', header=None)
train_data.columns = ['Label']+[ 'F'+str(i) for i in range(1,23)]
train_data = shuffle(train_data) # shuffle data
train_data
```

| | Label | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | ... | F13 | F14 | F15 | F16 | F17 | F18 | F19 | F20 | F21 | F22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **38** | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| **12** | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | ... | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| **60** | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **59** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **20** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | ... | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **17** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| **37** | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **49** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| **10** | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | ... | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| **11** | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | ... | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |

80 rows × 23 columns

```
# practice 6.2: Load testing data
import pandas as pd
test_data = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/spect/SPECT.test', header=None)
test_data.columns = ['Label']+[ 'F'+str(i) for i in range(1,23)]
test_data
```

| | Label | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | ... | F13 | F14 | F15 | F16 | F17 | F18 | F19 | F20 | F21 | F22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | ... | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| **1** | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | ... | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | ... | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| **3** | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | ... | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| **4** | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | ... | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **182** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **183** | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **184** | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | ... | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **185** | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | ... | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **186** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

187 rows × 23 columns

```
## Practice 6.3: extract features and labels from the training/test data
train_label = train_data['Label']
train_features = train_data.drop('Label',axis= 1)
print("train_features.shape:",train_features.shape)

test_label = test_data['Label']
test_features = test_data.drop('Label',axis=1)
print("test_features.shape:",test_features.shape)
```

```
train_features.shape: (80, 22)
test_features.shape: (187, 22)
```

The Sklearn model had a perfect training accuracy (1.0) and test accuracy (0.94), but the Keras model had a slightly lower test accuracy (0.925). Both models overfit the training data, but the Sklearn model fared marginally better on the test set. This minor gap shows that the Keras model may require additional fine-tuning, such as changing hyperparameters or adding regularization, to equal Sklearn's performance.

## Question 9

```
# practice 7.1: setup iris dataset for multi-class classification
from sklearn import datasets
import numpy as np

iris = datasets.load_iris()

# get features from iris data
X = iris["data"][:, :] # using all features

# get binary labels for binary classification
y = iris["target"]

print(X.shape)
print(y)
```

```
(150, 4)
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

```
# practice 7.2: Define the neural network classifier with softmax function as output
from keras.models import Sequential
from keras.layers import Dense

model = Sequential() # initialize Sequential model
model.add(Dense(10, input_shape=(X.shape[1],), activation='relu')) # input layer -> hidden layer 1
model.add(Dense(5, activation='relu')) # hidden layer 1 -> hidden layer 2
model.add(Dense(3, activation='softmax')) # hidden layer 2 -> output layer (3 classes, use softmax function to calculate probabi


# Compile model, use categorical_crossentropy as loss function for softmax function
model.compile(loss='categorical_crossentropy',
optimizer='SGD',
metrics=['accuracy'])


# practice 7.3: Train the model on the training set
model.fit(X, y, validation_split = 0.2, epochs=10)
```

```
Epoch 1/10
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-29-f0ebaac75698> in <cell line: 0>()
      1 # practice 7.3: Train the model on the training set
----> 2 model.fit(X, y, validation_split = 0.2, epochs=10)

 1 frames
/usr/local/lib/python3.11/dist-packages/keras/src/backend/tensorflow/nn.py in categorical_crossentropy(target, output,
from_logits, axis)
    651         )
    652     if len(target.shape) != len(output.shape):
--> 653         raise ValueError(
    654             "Arguments `target` and `output` must have the same rank "
    655             "(ndim). Received: "

ValueError: Arguments `target` and `output` must have the same rank (ndim). Received: target.shape=(None,), output.shape=
(None, 3)
```

Next steps: Explain error

Multi-class classification requires one-hot encoding when softmax is used in the output layer. After converting the labels to one-hot format, the model should train without a ValueError.

```
#Code to fix the error
```

```
from sklearn import datasets
from tensorflow.keras.utils import to_categorical
import numpy as np

iris = datasets.load_iris()
X = iris["data"][:, :]
y = iris["target"]

y_one_hot = to_categorical(y, num_classes=3)

model = Sequential()
model.add(Dense(10, input_shape=(X.shape[1],), activation='relu'))
model.add(Dense(5, activation='relu'))
model.add(Dense(3, activation='softmax'))

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

model.fit(X, y_one_hot, validation_split=0.2, epochs=10)
```

```
Epoch 1/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 3s 523ms/step - accuracy: 0.5250 - loss: 2.1900 - val_accuracy: 0.5667 - val_loss: 0.7257
Epoch 2/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.5967 - loss: 1.4508 - val_accuracy: 1.0000 - val_loss: 0.6935
Epoch 3/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.5479 - loss: 1.2312 - val_accuracy: 1.0000 - val_loss: 0.7580
Epoch 4/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.5583 - loss: 1.0108 - val_accuracy: 1.0000 - val_loss: 0.7883
Epoch 5/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.5625 - loss: 0.9036 - val_accuracy: 1.0000 - val_loss: 0.8557
Epoch 6/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.6167 - loss: 0.7492 - val_accuracy: 1.0000 - val_loss: 0.8573
Epoch 7/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.5813 - loss: 0.7520 - val_accuracy: 1.0000 - val_loss: 0.9012
Epoch 8/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.6052 - loss: 0.7245 - val_accuracy: 1.0000 - val_loss: 0.9224
Epoch 9/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.5687 - loss: 0.7336 - val_accuracy: 0.9667 - val_loss: 0.9537
Epoch 10/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.6204 - loss: 0.6909 - val_accuracy: 1.0000 - val_loss: 0.9167
<keras.src.callbacks.history.History at 0x7b10c08d6650>
```

```
## re-fit model on the training features and one-hot encoding labels
import tensorflow as tf
# Remove the 'dtype' argument
y_onehot = tf.keras.utils.to_categorical(y, num_classes=3)
print(y_onehot)
model.fit(X, y_onehot, validation_split = 0.2, epochs=10)
```

```
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]]
Epoch 1/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.6369 - loss: 0.7144 - val_accuracy: 1.0000 - val_loss: 0.9278
Epoch 2/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.6504 - loss: 0.7107 - val_accuracy: 1.0000 - val_loss: 0.9012
Epoch 3/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.6804 - loss: 0.7090 - val_accuracy: 0.8333 - val_loss: 0.9408
Epoch 4/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.7273 - loss: 0.6688 - val_accuracy: 1.0000 - val_loss: 0.9013
Epoch 5/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.7923 - loss: 0.6741 - val_accuracy: 0.8333 - val_loss: 0.9196
Epoch 6/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.8129 - loss: 0.6755 - val_accuracy: 1.0000 - val_loss: 0.8706
Epoch 7/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.9187 - loss: 0.6679 - val_accuracy: 0.8000 - val_loss: 0.9151
Epoch 8/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.8606 - loss: 0.6172 - val_accuracy: 0.8000 - val_loss: 0.9091
Epoch 9/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.9206 - loss: 0.6154 - val_accuracy: 0.7333 - val_loss: 0.9089
Epoch 10/10
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.9444 - loss: 0.5915 - val_accuracy: 0.8000 - val_loss: 0.8776
<keras.src.callbacks.history.History at 0x7b10d8516410>
```

## Question 10

```
# Practice 7.1: Monitor the model performance using validation set
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense

# Assuming train_features and train_label are from the SPECT dataset
X_train, X_val, y_train, y_val = train_test_split(train_features, train_label, test_size=0.2, random_state=42)

# Redefine the model with the correct input shape for the SPECT dataset
model = Sequential()
model.add(Dense(100, input_shape=(X_train.shape[1],), activation='relu'))  # Input layer -> Hidden layer 1
model.add(Dense(100, activation='relu'))  # Hidden layer 1 -> Hidden layer 2
model.add(Dense(1, activation='sigmoid'))  # Hidden layer 2 -> Output layer


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

# Train the model with validation data
model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=20)
```

```
Epoch 1/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 330ms/step - accuracy: 0.4479 - loss: 0.7024 - val_accuracy: 0.3125 - val_loss: 0.7233
Epoch 2/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step - accuracy: 0.3750 - loss: 0.7019 - val_accuracy: 0.4375 - val_loss: 0.7195
Epoch 3/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step - accuracy: 0.4688 - loss: 0.6971 - val_accuracy: 0.4375 - val_loss: 0.7158
Epoch 4/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step - accuracy: 0.5417 - loss: 0.6945 - val_accuracy: 0.4375 - val_loss: 0.7123
Epoch 5/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step - accuracy: 0.5417 - loss: 0.6950 - val_accuracy: 0.5625 - val_loss: 0.7088
Epoch 6/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - accuracy: 0.5729 - loss: 0.6867 - val_accuracy: 0.6250 - val_loss: 0.7054
Epoch 7/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step - accuracy: 0.6042 - loss: 0.6823 - val_accuracy: 0.6875 - val_loss: 0.7020
Epoch 8/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step - accuracy: 0.6354 - loss: 0.6783 - val_accuracy: 0.6875 - val_loss: 0.6989
Epoch 9/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 94ms/step - accuracy: 0.6042 - loss: 0.6852 - val_accuracy: 0.6250 - val_loss: 0.6958
Epoch 10/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step - accuracy: 0.6354 - loss: 0.6789 - val_accuracy: 0.6250 - val_loss: 0.6928
Epoch 11/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - accuracy: 0.5729 - loss: 0.6831 - val_accuracy: 0.6875 - val_loss: 0.6899
Epoch 12/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step - accuracy: 0.5833 - loss: 0.6811 - val_accuracy: 0.7500 - val_loss: 0.6871
Epoch 13/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - accuracy: 0.6146 - loss: 0.6761 - val_accuracy: 0.7500 - val_loss: 0.6843
Epoch 14/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step - accuracy: 0.6354 - loss: 0.6746 - val_accuracy: 0.6875 - val_loss: 0.6816
Epoch 15/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step - accuracy: 0.6250 - loss: 0.6760 - val_accuracy: 0.6875 - val_loss: 0.6791
Epoch 16/20
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - accuracy: 0.6875 - loss: 0.6688 - val_accuracy: 0.6875 - val_loss: 0.6766
Epoch 17/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step - accuracy: 0.6458 - loss: 0.6726 - val_accuracy: 0.6875 - val_loss: 0.6741
Epoch 18/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step - accuracy: 0.7188 - loss: 0.6680 - val_accuracy: 0.6875 - val_loss: 0.6718
Epoch 19/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - accuracy: 0.7083 - loss: 0.6663 - val_accuracy: 0.6875 - val_loss: 0.6694
Epoch 20/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step - accuracy: 0.7292 - loss: 0.6631 - val_accuracy: 0.6875 - val_loss: 0.6672
<keras.src.callbacks.history.History at 0x7b10de730610>
```

```
## option 2: monitor the model training using validation data
model.fit(X_train, y_train, validation_split=0.2, epochs = 20)
```

```
Epoch 1/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 2s 1s/step - accuracy: 0.6789 - loss: 0.6703 - val_accuracy: 0.6923 - val_loss: 0.6542
Epoch 2/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 55ms/step - accuracy: 0.7102 - loss: 0.6649 - val_accuracy: 0.6923 - val_loss: 0.6540
Epoch 3/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step - accuracy: 0.7102 - loss: 0.6645 - val_accuracy: 0.7692 - val_loss: 0.6536
Epoch 4/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step - accuracy: 0.7075 - loss: 0.6589 - val_accuracy: 0.7692 - val_loss: 0.6534
Epoch 5/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step - accuracy: 0.7102 - loss: 0.6611 - val_accuracy: 0.7692 - val_loss: 0.6531
Epoch 6/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - accuracy: 0.6998 - loss: 0.6631 - val_accuracy: 0.7692 - val_loss: 0.6529
Epoch 7/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step - accuracy: 0.6893 - loss: 0.6576 - val_accuracy: 0.7692 - val_loss: 0.6527
Epoch 8/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step - accuracy: 0.7102 - loss: 0.6538 - val_accuracy: 0.7692 - val_loss: 0.6525
Epoch 9/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - accuracy: 0.6685 - loss: 0.6572 - val_accuracy: 0.7692 - val_loss: 0.6524
Epoch 10/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step - accuracy: 0.7206 - loss: 0.6531 - val_accuracy: 0.7692 - val_loss: 0.6522
Epoch 11/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - accuracy: 0.7414 - loss: 0.6508 - val_accuracy: 0.6923 - val_loss: 0.6521
Epoch 12/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - accuracy: 0.7337 - loss: 0.6529 - val_accuracy: 0.6923 - val_loss: 0.6519
Epoch 13/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - accuracy: 0.7363 - loss: 0.6495 - val_accuracy: 0.6923 - val_loss: 0.6519
Epoch 14/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step - accuracy: 0.7571 - loss: 0.6479 - val_accuracy: 0.6923 - val_loss: 0.6518
Epoch 15/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - accuracy: 0.7571 - loss: 0.6461 - val_accuracy: 0.6923 - val_loss: 0.6517
Epoch 16/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - accuracy: 0.7598 - loss: 0.6459 - val_accuracy: 0.6923 - val_loss: 0.6516
Epoch 17/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step - accuracy: 0.7702 - loss: 0.6461 - val_accuracy: 0.6923 - val_loss: 0.6516
Epoch 18/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 65ms/step - accuracy: 0.7598 - loss: 0.6483 - val_accuracy: 0.6923 - val_loss: 0.6515
Epoch 19/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step - accuracy: 0.7806 - loss: 0.6388 - val_accuracy: 0.6923 - val_loss: 0.6516
Epoch 20/20
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step - accuracy: 0.7494 - loss: 0.6471 - val_accuracy: 0.6923 - val_loss: 0.6516
<keras.src.callbacks.history.History at 0x7b10de76f490>
```

## Question 11

```
# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Convert labels to one-hot encoding
y_one_hot = to_categorical(y, num_classes=3)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_one_hot, test_size=0.2, random_state=42)

# Build the neural network model
model = Sequential()
model.add(Dense(64, input_dim=X_train.shape[1], activation='relu'))  # First hidden layer
model.add(Dense(32, activation='relu'))  # Second hidden layer
model.add(Dense(3, activation='softmax'))  # Output layer

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

```
# Train the model
train_history = model.fit(X_train, y_train, validation_split=0.1, batch_size=5, epochs=30)

# Evaluate the model
test_loss, test_accuracy = model.evaluate(X_test, y_test)

# Print test accuracy
print(f"Test Accuracy: {test_accuracy}")
```

```
Epoch 3/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.6783 - loss: 0.8402 - val_accuracy: 1.0000 - val_loss: 0.7718
Epoch 4/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.7371 - loss: 0.7061 - val_accuracy: 0.5833 - val_loss: 0.7409
Epoch 5/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.6282 - loss: 0.6500 - val_accuracy: 0.5833 - val_loss: 0.7063
Epoch 6/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.6868 - loss: 0.5690 - val_accuracy: 0.5833 - val_loss: 0.6572
Epoch 7/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.7129 - loss: 0.5501 - val_accuracy: 0.9167 - val_loss: 0.5741
Epoch 8/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8036 - loss: 0.4918 - val_accuracy: 0.7500 - val_loss: 0.5583
Epoch 9/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8168 - loss: 0.4651 - val_accuracy: 1.0000 - val_loss: 0.5284
Epoch 10/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8782 - loss: 0.4505 - val_accuracy: 0.9167 - val_loss: 0.4995
Epoch 11/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8944 - loss: 0.4487 - val_accuracy: 0.7500 - val_loss: 0.4904
Epoch 12/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8283 - loss: 0.3776 - val_accuracy: 0.9167 - val_loss: 0.4643
Epoch 13/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8491 - loss: 0.4010 - val_accuracy: 1.0000 - val_loss: 0.4434
Epoch 14/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9197 - loss: 0.3728 - val_accuracy: 0.9167 - val_loss: 0.4289
Epoch 15/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8179 - loss: 0.4153 - val_accuracy: 0.8333 - val_loss: 0.4388
Epoch 16/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8948 - loss: 0.3904 - val_accuracy: 0.6667 - val_loss: 0.4438
Epoch 17/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8430 - loss: 0.3317 - val_accuracy: 1.0000 - val_loss: 0.3772
Epoch 18/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9114 - loss: 0.3205 - val_accuracy: 0.9167 - val_loss: 0.3546
Epoch 19/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8415 - loss: 0.3731 - val_accuracy: 1.0000 - val_loss: 0.3352
Epoch 20/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9634 - loss: 0.2827 - val_accuracy: 1.0000 - val_loss: 0.3427
Epoch 21/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9514 - loss: 0.2609 - val_accuracy: 1.0000 - val_loss: 0.3019
Epoch 22/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9322 - loss: 0.3033 - val_accuracy: 0.9167 - val_loss: 0.2894
Epoch 23/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9398 - loss: 0.2330 - val_accuracy: 1.0000 - val_loss: 0.2978
Epoch 24/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9755 - loss: 0.2481 - val_accuracy: 1.0000 - val_loss: 0.2616
Epoch 25/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8916 - loss: 0.2467 - val_accuracy: 0.9167 - val_loss: 0.2684
Epoch 26/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9513 - loss: 0.2005 - val_accuracy: 1.0000 - val_loss: 0.2489
Epoch 27/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9521 - loss: 0.2465 - val_accuracy: 1.0000 - val_loss: 0.2133
Epoch 28/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9687 - loss: 0.2157 - val_accuracy: 1.0000 - val_loss: 0.2009
Epoch 29/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.9557 - loss: 0.1891 - val_accuracy: 1.0000 - val_loss: 0.1922
Epoch 30/30
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9443 - loss: 0.1887 - val_accuracy: 1.0000 - val_loss: 0.1838
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 570ms/step - accuracy: 0.9333 - loss: 0.2089
Test Accuracy: 0.9333333373069763
```

```
# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data  # Features
y = iris.target  # Labels

# Convert labels to one-hot encoding
y_one_hot = to_categorical(y, num_classes=3)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_one_hot, test_size=0.2, random_state=42)

# Build the neural network model with increased complexity
model = Sequential()
```

```
model.add(Dense(100, input_dim=X_train.shape[1], activation='relu'))  # First hidden layer
model.add(Dense(100, activation='relu'))  # Second hidden layer
model.add(Dense(100, activation='relu'))  # Third hidden layer
model.add(Dense(3, activation='softmax'))  # Output layer

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

# Train the model
train_history = model.fit(X_train, y_train, validation_split=0.1, batch_size=5, epochs=50)

# Evaluate the model on training and testing data
train_loss, train_accuracy = model.evaluate(X_train, y_train)
test_loss, test_accuracy = model.evaluate(X_test, y_test)

# Print the evaluation results
print(f"Training Accuracy: {train_accuracy}")
print(f"Test Accuracy: {test_accuracy}")
```

```
Epoch 24/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9022 - loss: 0.2704 - val_accuracy: 0.9167 - val_loss: 0.1979
Epoch 25/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9027 - loss: 0.1706 - val_accuracy: 0.8333 - val_loss: 0.2897
Epoch 26/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8797 - loss: 0.2972 - val_accuracy: 0.9167 - val_loss: 0.2269
Epoch 27/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9459 - loss: 0.2385 - val_accuracy: 1.0000 - val_loss: 0.1318
Epoch 28/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9797 - loss: 0.1447 - val_accuracy: 1.0000 - val_loss: 0.1156
Epoch 29/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9620 - loss: 0.1729 - val_accuracy: 0.5833 - val_loss: 1.9258
Epoch 30/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.7984 - loss: 0.6253 - val_accuracy: 0.9167 - val_loss: 0.1786
Epoch 31/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9205 - loss: 0.2456 - val_accuracy: 1.0000 - val_loss: 0.1193
Epoch 32/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9720 - loss: 0.1168 - val_accuracy: 0.9167 - val_loss: 0.2395
Epoch 33/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9429 - loss: 0.1871 - val_accuracy: 1.0000 - val_loss: 0.1139
Epoch 34/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9397 - loss: 0.2065 - val_accuracy: 0.9167 - val_loss: 0.2261
Epoch 35/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9416 - loss: 0.1554 - val_accuracy: 1.0000 - val_loss: 0.0888
Epoch 36/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9679 - loss: 0.1455 - val_accuracy: 0.9167 - val_loss: 0.2812
Epoch 37/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9726 - loss: 0.1211 - val_accuracy: 1.0000 - val_loss: 0.0854
Epoch 38/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9668 - loss: 0.1280 - val_accuracy: 1.0000 - val_loss: 0.0835
Epoch 39/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9350 - loss: 0.1211 - val_accuracy: 0.9167 - val_loss: 0.1600
Epoch 40/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9376 - loss: 0.1218 - val_accuracy: 1.0000 - val_loss: 0.0785
Epoch 41/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9067 - loss: 0.1807 - val_accuracy: 0.5833 - val_loss: 0.5955
Epoch 42/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9254 - loss: 0.1818 - val_accuracy: 0.9167 - val_loss: 0.1528
Epoch 43/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9062 - loss: 0.1565 - val_accuracy: 1.0000 - val_loss: 0.0910
Epoch 44/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9731 - loss: 0.1546 - val_accuracy: 1.0000 - val_loss: 0.0720
Epoch 45/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8303 - loss: 0.4031 - val_accuracy: 1.0000 - val_loss: 0.0733
Epoch 46/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9749 - loss: 0.1142 - val_accuracy: 0.9167 - val_loss: 0.1107
Epoch 47/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9347 - loss: 0.1889 - val_accuracy: 0.8333 - val_loss: 0.2100
Epoch 48/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9357 - loss: 0.1473 - val_accuracy: 0.8333 - val_loss: 0.2328
Epoch 49/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8360 - loss: 0.3602 - val_accuracy: 1.0000 - val_loss: 0.0547
Epoch 50/50
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9627 - loss: 0.0965 - val_accuracy: 1.0000 - val_loss: 0.0888
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 124ms/step - accuracy: 0.9446 - loss: 0.1072
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 514ms/step - accuracy: 0.8667 - loss: 0.1817
Training Accuracy: 0.949999988079071
Test Accuracy: 0.8666666746139526
```

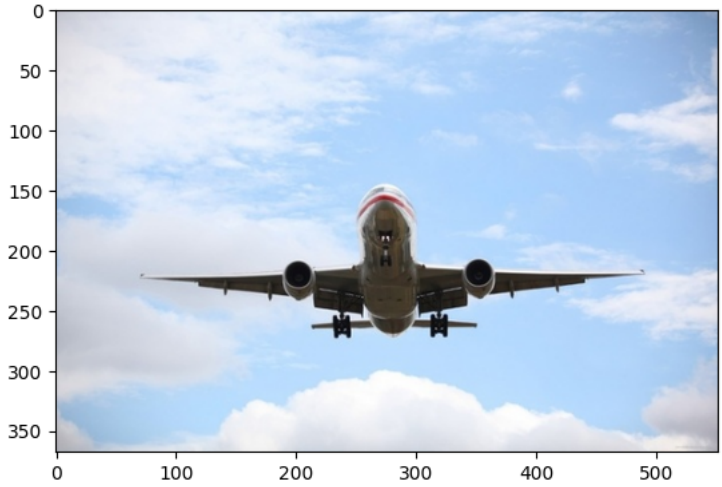
## Part 2

## Question 12

```
# step 1.1: download an online image
!wget https://images.all-free-download.com/images/graphicthumb/airplane_landing_199029.jpg
```

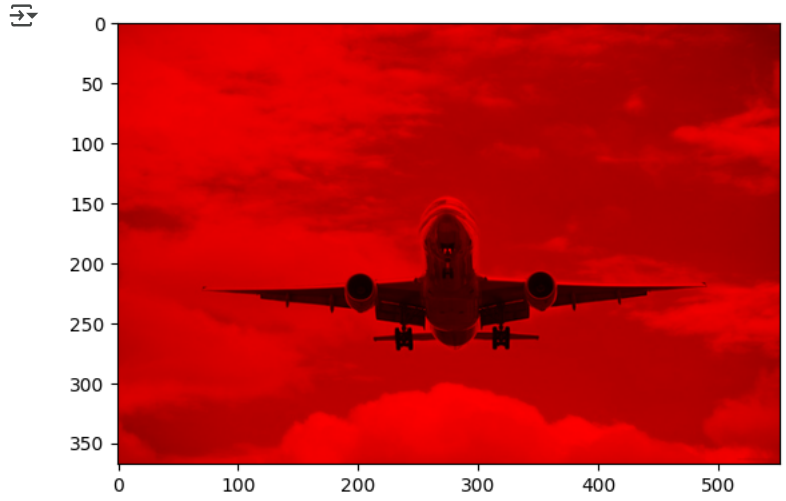
```
--2025-05-14 23:33:25--  https://images.all-free-download.com/images/graphicthumb/airplane_landing_199029.jpg
Resolving images.all-free-download.com (images.all-free-download.com)... 51.81.66.158
Connecting to images.all-free-download.com (images.all-free-download.com)|51.81.66.158|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 42917 (42K) [image/jpeg]
Saving to: 'airplane_landing_199029.jpg'

airplane_landing_19 100%[===================>]  41.91K  --.-KB/s    in 0s

2025-05-14 23:33:26 (296 MB/s) - 'airplane_landing_199029.jpg' saved [42917/42917]
```

```
# step 1.2: load image into python
import matplotlib.pyplot as plt
data = plt.imread('airplane_landing_199029.jpg')
plt.imshow(data)
plt.show()
```

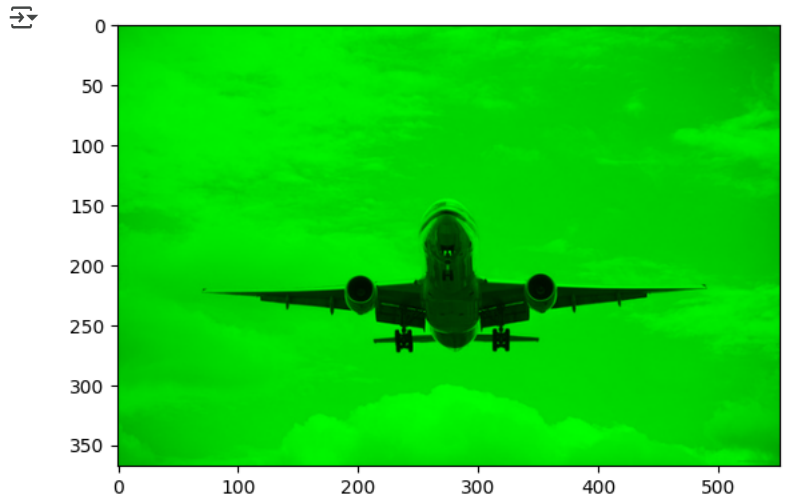



## Question 13

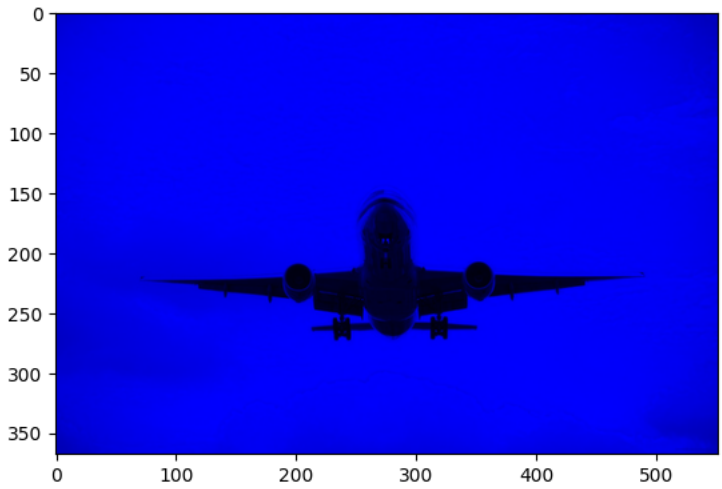
```
# step 2.1: Modify the image to red channel only
data_modified = data.copy()
data_modified[:,:,1] = 0 # set intensity of green channel to 0
data_modified[:,:,2] = 0 # set intensity of blue channel to 0
plt.imshow(data_modified) # only show red channel
plt.show()
```



```
# step 2.2: Modify the image to green channel only
data_modified = data.copy()
data_modified[:,:,0] = 0 # set intensity of red channel to 0
data_modified[:,:,2] = 0 # set intensity of blue channel to 0
plt.imshow(data_modified) # only show red channel
plt.show()
```

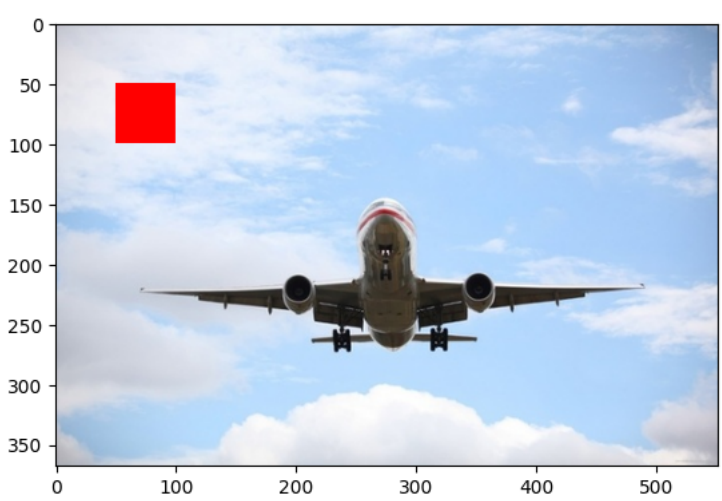


```
# Assuming data is the original image
data_modified = data.copy()  # Create a copy of the original image

# Set intensity of red and green channels to 0
data_modified[:, :, 0] = 0  # Set red channel to 0
data_modified[:, :, 1] = 0  # Set green channel to 0

# Show only the blue channel
plt.imshow(data_modified)  # Display the modified image
plt.show()
```

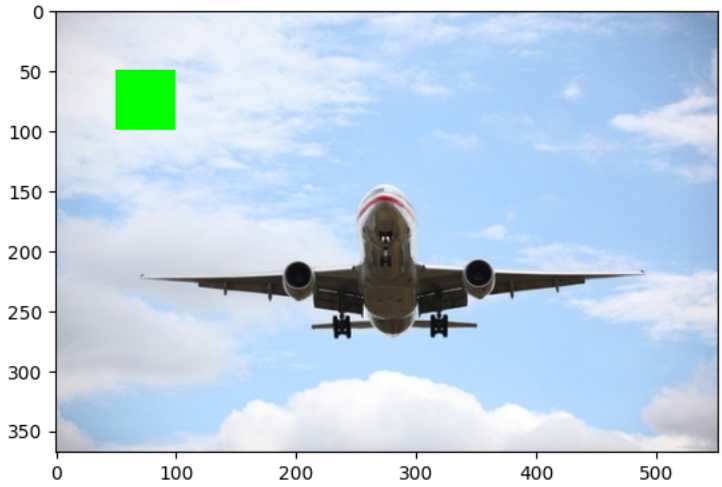


## Question 14

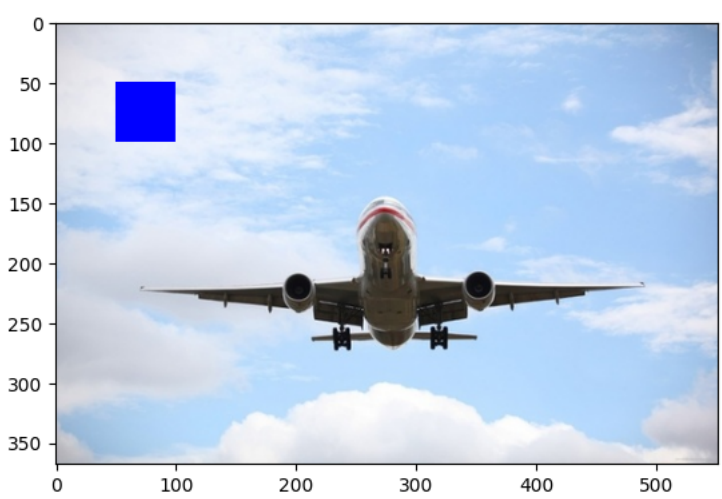
```
# step 3.1: Add red mask to the specific regions within the image**
data_modified = data.copy()
data_modified[50:100,50:100,:] = [255,0,0] # change subregion to red only, make sure the order of channel colors is [red,green,b
plt.imshow(data_modified)
plt.show()
```



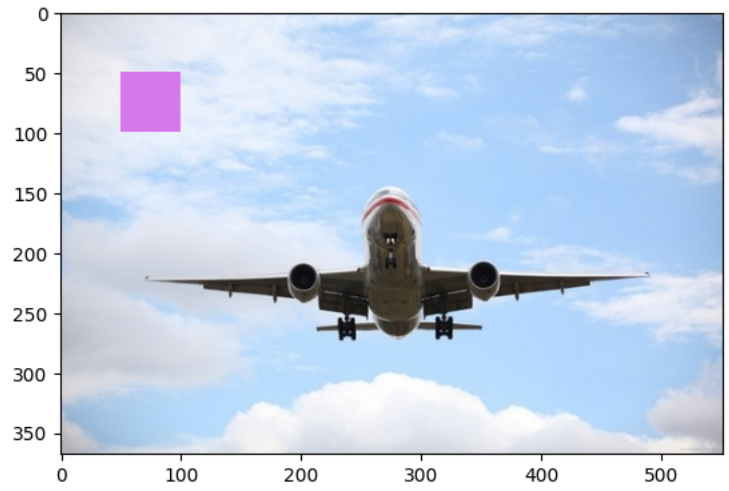
```
# step 3.2: Add green mask to the specific regions within the image**
data_modified = data.copy()
data_modified[50:100,50:100,:] = [0,255,0] # change subregion to green only, make sure the order of channel colors is [red,greer
plt.imshow(data_modified)
plt.show()
```



```
# Assuming data is the original image
data_modified = data.copy()  # Create a copy of the original image

# Set the region of the image to blue only
data_modified[50:100, 50:100, :] = [0, 0, 255]  # Change subregion to blue, RGB order is [red, green, blue]

# Display the modified image
plt.imshow(data_modified)
plt.show()
```



```
# Assuming data is the original image
data_modified = data.copy()  # Create a copy of the original image

# Define the custom color (e.g., [215, 120, 234])
custom_color = [215, 120, 234]

# Set the region of the image to the custom color
data_modified[50:100, 50:100, :] = custom_color  # Change subregion to custom color

# Display the modified image
plt.imshow(data_modified)
plt.show()
```




## Question 15

```
# Step 4: Apply image processing on the CIFAR10 image dataset
from keras.datasets import cifar10
import matplotlib.pyplot as plt
import numpy as np

# load dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

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

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 13s 0us/step
```

```
# Assuming X_train, X_test, y_train, y_test are already defined
print("X_train.shape: ", X_train.shape)
print("X_test.shape: ", X_test.shape)
print("y_train.shape: ", y_train.shape)
print("y_test.shape: ", y_test.shape)
```

```
X_train.shape:  (50000, 32, 32, 3)
X_test.shape:  (10000, 32, 32, 3)
y_train.shape:  (50000, 1)
y_test.shape:  (10000, 1)
```

X_train: A 4D tensor (rank 4) with the shape (50000, 32, 32, 3), which represents training images (samples, height, width, channels). X_test: A 4D tensor (rank 4) with the shape (10000, 32, 32, 3), which represents test images. y_train is a 2D tensor (rank 2) with the shape (50000, 1) that represents labels for training data. y_test is a two-dimensional tensor (rank 2) with the shape (10000, 1) that represents test data labels. In summary, X_train and X_test are 4D tensors, whereas Y_train and Y_test are 2D tensors.

## Question 16

```
# Step 5: Image visualization using matplotlib
fig = plt.figure(figsize=(20, 5))
for ii in range(10):
   # sample a random image from X_train
   image_indx = np.random.choice(range(len(X_train)))
   image_random = X_train[image_indx]
   image_title = labels_map[y_train[image_indx][-1]]

   # put image into subplots
   imgplot = fig.add_subplot(2,5,ii+1)
   imgplot.imshow(image_random)
   imgplot.set_title(image_title, fontsize=20)
   imgplot.axis('off')
```


```
# Step 6.1: sample a random image
image_index = np.random.choice(range(len(X_train)))
image_example = X_train[image_index]


# Step 6.2:  Use the Python Imaging Library 'PIL' to save image into local file
import PIL
img = PIL.Image.fromarray(X_train[image_index])
img.save('image.jpg')


# Step 6.3:  reload image into python
import matplotlib.pyplot as plt
data = plt.imread('image.jpg') # read the local image
plt.imshow(data)
plt.axis('off')
plt.show()
```

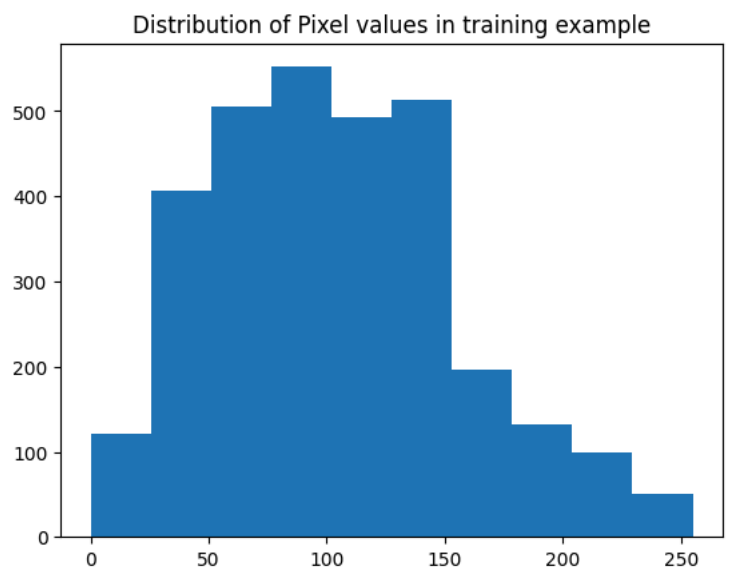
```
# Check if the shape of the reloaded image is (32, 32, 3)
if data.shape == (32, 32, 3):
    print("The image shape is correct: (32, 32, 3)")
else:
    print(f"The image shape is incorrect. Got shape: {data.shape}")

# Print the shape of the reloaded image
print("Shape of the reloaded image:", data.shape)
```

```
The image shape is correct: (32, 32, 3)
Shape of the reloaded image: (32, 32, 3)
```

```
## Step 7.1: Load the dataset
from keras.datasets import cifar10
(X_data, y_data), (X_test, y_test) = cifar10.load_data()
print("Training matrix shape", X_data.shape)
print("Testing matrix shape", X_test.shape)
print("y_data matrix shape", y_data.shape)
print("y_test matrix shape", y_test.shape)
print("y_data: ",y_data)
```

```
Training matrix shape (50000, 32, 32, 3)
Testing matrix shape (10000, 32, 32, 3)
y_data matrix shape (50000, 1)
y_test matrix shape (10000, 1)
y_data:  [[6]
 [9]
 [9]
 ...
 [9]
 [1]
 [1]]
```

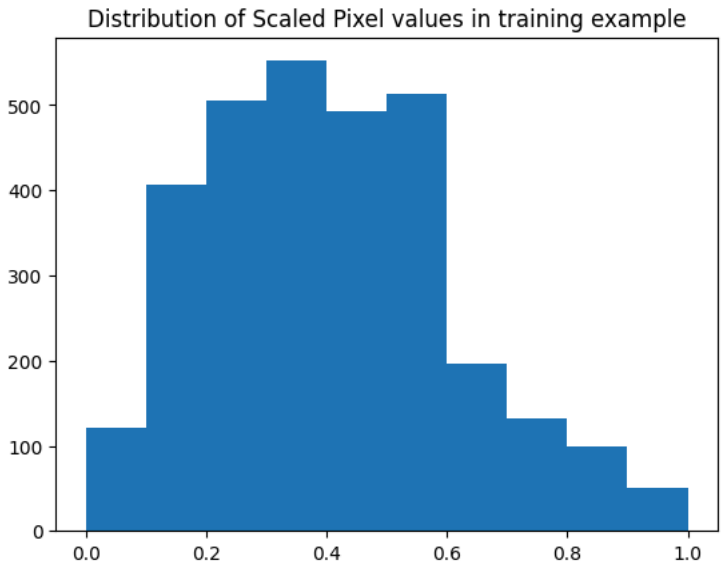
```
## Step 7.2: Plot the histogram for the pixels in each image
from matplotlib import pyplot as plt
plt.hist(X_data[0].flatten(),)
plt.title("Distribution of Pixel values in training example")
plt.show()
```




```
N_train = X_data.shape[0]  # the first dimension of the tensor is number of total images
D_train = 32*32*3   # the remaining dimensions of the tensor is the shape of image, you can also use X_data.shape[1]*X_data.shap
X_data_flatten = X_data.reshape(N_train, D_train)
X_data_flatten = X_data_flatten.astype('float32')

N_test = X_test.shape[0]
D_test = 32*32*3
X_test_flatten = X_test.reshape(N_test,D_test)
X_test_flatten = X_test_flatten.astype('float32')

print("Training matrix shape", X_data_flatten.shape)
print("Testing matrix shape", X_test_flatten.shape)
```

```
Training matrix shape (50000, 3072)
Testing matrix shape (10000, 3072)
```

```
X_data_flatten /= 255
X_test_flatten /= 255


from matplotlib import pyplot as plt
plt.hist(X_data_flatten[0].flatten(),)
plt.title("Distribution of Scaled Pixel values in training example")
plt.show()
```




## Question 17

```
print(y_data)
```

```
[[6]
 [9]
 [9]
 ...
 [9]
 [1]
 [1]]
```

```
labels_map = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
nb_classes = len(labels_map)

import tensorflow as tf
y_data_categorical = tf.keras.utils.to_categorical(y_data, nb_classes)
y_test_categorical = tf.keras.utils.to_categorical(y_test, nb_classes)

print("y_data matrix shape", y_data_categorical.shape)
print("y_test matrix shape", y_test_categorical.shape)
print("y_test_categorical: ",y_test_categorical)
```

```
y_data matrix shape (50000, 10)
y_test matrix shape (10000, 10)
y_test_categorical:  [[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 1. 0.]
 [0. 0. 0. ... 0. 1. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 1. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 1. 0. 0.]]
```

Before, the labels were only integers. The labels are then converted to a one-hot encoded matrix, which is required for multi-class classification in neural networks.

## Question 18

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split

## Set up training and validation dataset
X_train, X_val, y_train, y_val = train_test_split(X_data_flatten,y_data_categorical, test_size=0.2, random_state=42)
def build_model(n_layers = 3, n_neurons = 1000):
   model = Sequential() # create Sequential model
   for i in range(n_layers-1):
```

```
        model.add(Dense(n_neurons, activation = 'relu')) # you can also try other types of activation functions
    model.add(Dense(10, activation = 'softmax'))  # the output must be softmax for multi-class classifcation

    return model

model = build_model(n_layers = 3, n_neurons = 1000)
model.compile(loss = "categorical_crossentropy", optimizer = "adam", metrics = ['accuracy'])
train_history = model.fit(X_train,y_train, validation_data=(X_val,y_val), batch_size=128, epochs = 20)
```

```
Epoch 1/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 11ms/step - accuracy: 0.2505 - loss: 2.2926 - val_accuracy: 0.3606 - val_loss: 1.7857
Epoch 2/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.3789 - loss: 1.7322 - val_accuracy: 0.3889 - val_loss: 1.6934
Epoch 3/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.4206 - loss: 1.6262 - val_accuracy: 0.4398 - val_loss: 1.5682
Epoch 4/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.4457 - loss: 1.5525 - val_accuracy: 0.4256 - val_loss: 1.6389
Epoch 5/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.4659 - loss: 1.4989 - val_accuracy: 0.4578 - val_loss: 1.5202
Epoch 6/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.4769 - loss: 1.4596 - val_accuracy: 0.4685 - val_loss: 1.4955
Epoch 7/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.4977 - loss: 1.4117 - val_accuracy: 0.4640 - val_loss: 1.5072
Epoch 8/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.5098 - loss: 1.3768 - val_accuracy: 0.4785 - val_loss: 1.4598
Epoch 9/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.5221 - loss: 1.3490 - val_accuracy: 0.4748 - val_loss: 1.4576
Epoch 10/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.5242 - loss: 1.3300 - val_accuracy: 0.4775 - val_loss: 1.4706
Epoch 11/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.5386 - loss: 1.3044 - val_accuracy: 0.4853 - val_loss: 1.4481
Epoch 12/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5462 - loss: 1.2641 - val_accuracy: 0.5014 - val_loss: 1.4214
Epoch 13/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.5636 - loss: 1.2317 - val_accuracy: 0.4917 - val_loss: 1.4389
Epoch 14/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.5659 - loss: 1.2126 - val_accuracy: 0.5008 - val_loss: 1.4371
Epoch 15/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.5794 - loss: 1.1844 - val_accuracy: 0.4954 - val_loss: 1.4478
Epoch 16/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5885 - loss: 1.1569 - val_accuracy: 0.5076 - val_loss: 1.4339
Epoch 17/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.5960 - loss: 1.1299 - val_accuracy: 0.4965 - val_loss: 1.4827
Epoch 18/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6014 - loss: 1.1169 - val_accuracy: 0.5166 - val_loss: 1.4339
Epoch 19/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.6208 - loss: 1.0671 - val_accuracy: 0.5101 - val_loss: 1.4450
Epoch 20/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.6201 - loss: 1.0500 - val_accuracy: 0.5021 - val_loss: 1.4810
```

Training loss drops and training accuracy increases throughout epochs, indicating that the model is learning. Validation loss also diminishes, but more slowly, and validation accuracy falls behind training accuracy. This implies that the model may be overfitting, as it performs better on training data. Both losses and accuracies normally improve, but validation accuracy improves at a slower rate than training accuracy.

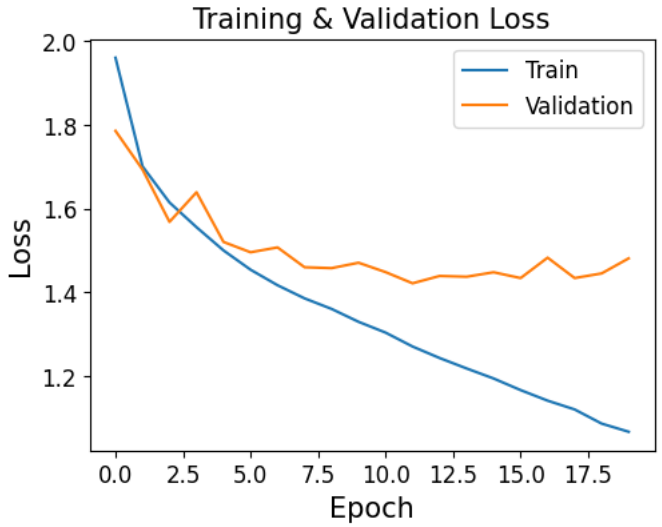
## Question 19

```
# Step 11.1:  Access the model training history
print(train_history.history.keys())
print(train_history.history['loss'])
```

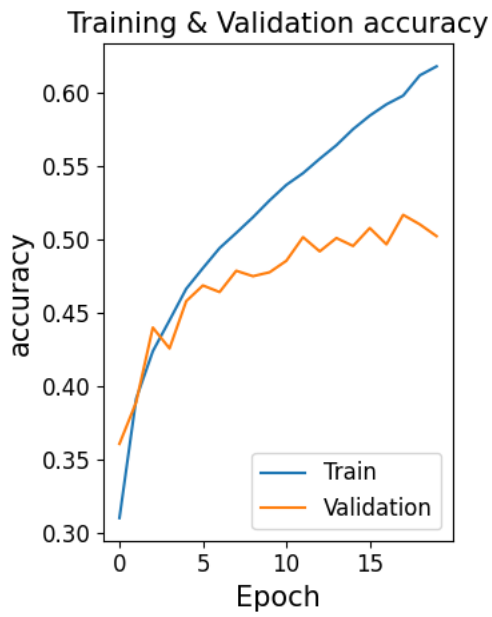
```
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
[1.960437297821045, 1.7008906602859497, 1.6145190000534058, 1.5554239749908447, 1.4999343156814575, 1.4539252519607544, 1.41
```

```
# Step 11.2:  Plot the learning curves for training/validation
plt.figure(figsize=(12,4))
plt.subplot(1,2,1)
# Plot training & validation loss values
plt.plot(train_history.history['loss'], label='Train')
plt.plot(train_history.history['val_loss'], label='Validation')
plt.title('Training & Validation Loss', fontsize=15)
plt.ylabel('Loss', fontsize=15)
plt.xlabel('Epoch', fontsize=15)
plt.xticks( fontsize=12)
plt.yticks( fontsize=12)
plt.legend(loc='upper right', fontsize=12)
```

```
<matplotlib.legend.Legend at 0x7b1051f3b1d0>
```




```
plt.subplot(1,2,2)
# Plot training & validation accuracy values
plt.plot(train_history.history['accuracy'], label='Train')
plt.plot(train_history.history['val_accuracy'], label='Validation')
plt.title('Training & Validation accuracy', fontsize=15)
plt.ylabel('accuracy', fontsize=15)

plt.xlabel('Epoch', fontsize=15)
plt.xticks( fontsize=12)
plt.yticks( fontsize=12)
plt.legend(loc='lower right', fontsize=12)
plt.tight_layout()
plt.show()
```




If the model is not overfitting and there is still space for improvement in both training and validation accuracy, you may want to investigate raising the epoch count. Yes, if both accuracy and loss improve. Otherwise, stop early to prevent overfitting.

## Question 20

```
# Step 12:  Let's check the predicted labels of training data using the trained model
train_predicted_labels = model.predict(X_train[0:5,:]) # here we only predict the labels of first 5 images
print("Shape: ",train_predicted_labels.shape)
print(train_predicted_labels)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 335ms/step
Shape:  (5, 10)
[[6.35817945e-02 1.94398202e-02 9.74707901e-02 2.00363472e-01
  1.01741195e-01 2.70096421e-01 9.59649161e-02 4.64060605e-02
  8.35016221e-02 2.14339066e-02]
 [6.80031581e-03 3.37866724e-01 9.63487849e-02 6.69348314e-02
  2.62313825e-03 1.45571664e-01 1.79200377e-02 5.16163036e-02
  6.42461479e-02 2.10072011e-01]
 [1.06222786e-01 1.74547210e-02 1.17258742e-01 1.04184836e-01
  5.79870716e-02 5.04688323e-01 1.03660591e-03 8.35396349e-02
  6.24977227e-04 7.00232061e-03]
 [2.74998485e-03 4.18745354e-03 8.87036771e-02 4.35428798e-01
  1.24998868e-01 9.05129313e-02 2.13505119e-01 2.25230996e-02
  4.74324217e-03 1.26467813e-02]
 [2.06648139e-04 1.10428175e-03 3.47315543e-03 6.96046531e-01
  2.17613354e-02 2.53489017e-01 2.40697991e-03 1.61793698e-02
  8.64967180e-04 4.46777791e-03]]
```

```
train_predicted_labels = model.predict(X_train[0:5,:])

print("Shape: ", train_predicted_labels.shape)
print(train_predicted_labels)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
Shape:  (5, 10)
[[6.35817945e-02 1.94398202e-02 9.74707901e-02 2.00363472e-01
  1.01741195e-01 2.70096421e-01 9.59649161e-02 4.64060605e-02
  8.35016221e-02 2.14339066e-02]
 [6.80031581e-03 3.37866724e-01 9.63487849e-02 6.69348314e-02
  2.62313825e-03 1.45571664e-01 1.79200377e-02 5.16163036e-02
  6.42461479e-02 2.10072011e-01]
 [1.06222786e-01 1.74547210e-02 1.17258742e-01 1.04184836e-01
  5.79870716e-02 5.04688323e-01 1.03660591e-03 8.35396349e-02
  6.24977227e-04 7.00232061e-03]
 [2.74998485e-03 4.18745354e-03 8.87036771e-02 4.35428798e-01
  1.24998868e-01 9.05129313e-02 2.13505119e-01 2.25230996e-02
  4.74324217e-03 1.26467813e-02]
 [2.06648139e-04 1.10428175e-03 3.47315543e-03 6.96046531e-01
  2.17613354e-02 2.53489017e-01 2.40697991e-03 1.61793698e-02
  8.64967180e-04 4.46777791e-03]]
```

```
predicted_classes = np.argmax(train_predicted_labels, axis=1)

print("Predicted Classes: ", predicted_classes)
```

```
Predicted Classes:  [5 1 5 3 3]
```

```
train_predicted_labels = model.predict(X_train[0:5,:])

print("Shape: ", train_predicted_labels.shape)

print(train_predicted_labels)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step
Shape:  (5, 10)
[[6.35817945e-02 1.94398202e-02 9.74707901e-02 2.00363472e-01
  1.01741195e-01 2.70096421e-01 9.59649161e-02 4.64060605e-02
  8.35016221e-02 2.14339066e-02]
 [6.80031581e-03 3.37866724e-01 9.63487849e-02 6.69348314e-02
  2.62313825e-03 1.45571664e-01 1.79200377e-02 5.16163036e-02
  6.42461479e-02 2.10072011e-01]
 [1.06222786e-01 1.74547210e-02 1.17258742e-01 1.04184836e-01
  5.79870716e-02 5.04688323e-01 1.03660591e-03 8.35396349e-02
  6.24977227e-04 7.00232061e-03]
 [2.74998485e-03 4.18745354e-03 8.87036771e-02 4.35428798e-01
  1.24998868e-01 9.05129313e-02 2.13505119e-01 2.25230996e-02
  4.74324217e-03 1.26467813e-02]
 [2.06648139e-04 1.10428175e-03 3.47315543e-03 6.96046531e-01
  2.17613354e-02 2.53489017e-01 2.40697991e-03 1.61793698e-02
  8.64967180e-04 4.46777791e-03]]
```

```
import numpy as np
np.argmax(train_predicted_labels,axis=1) # find the index of column which has maximum value in each row
```

```
array([5, 1, 5, 3, 3])
```

The integers in the output array (such as [6, 1, 5, 6, 3]) are the predicted classes for each image. They are derived by determining the index with the highest probability in each row. Each row corresponds to one image, and each column reflects the projected likelihood that the image belongs to a particular class. The highest-valued index represents the image's expected class. For example, if the first row's highest probability is at index 6, the model predicts that the image belongs to class 6. This is how the model classifies each image, using the highest probability assigned to each class.

## Question 21

```
# Step 13:  Evaluate the classification performance
from sklearn.metrics import accuracy_score


def evaluate_model(model,train_data,val_data,test_data):
   X_train,y_train = train_data
   X_val,y_val = val_data
   X_test,y_test = test_data
    # (1) make a prediction on training set to get probablities for all classes, select the class that has maximum probablity
   y_train_pred = np.argmax(model.predict(X_train), axis=-1)
   # (2) calculate the training classification error
   Train_error_s = 1 - accuracy_score(np.argmax(y_train,axis=1), y_train_pred)
    # (3) make a prediction on validation set
   y_val_pred = np.argmax(model.predict(X_val), axis=-1)
    # (4) calculate the validation classification error
   Val_error_s = 1 - accuracy_score(np.argmax(y_val,axis=1), y_val_pred)
   # (5) make a prediction on test set
   y_test_pred = np.argmax(model.predict(X_test), axis=-1)
    # (6) calculate the test classification error
   Test_error_s = 1 - accuracy_score(y_test, y_test_pred)
    # (7) reporting results
   print("Train error: ", Train_error_s)
   print("Validation error: ", Val_error_s)
   print("Test error: ", Test_error_s)
   return Train_error_s,Val_error_s,Test_error_s


Train_error_s,Val_error_s,Test_error_s = evaluate_model(model,(X_train,y_train),(X_val,y_val),(X_test_flatten,y_test))
```

```
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Train error:  0.3568
Validation error:  0.4979
Test error:  0.48929999999999996
```

```
# Step 14: Implement a function for visualizing the improvements over attempts
def visualize_improvement(improvement_log_train,improvement_log_val,improvement_log_test):
   import matplotlib.pyplot as plt
   plt.figure(figsize=(12,4))

   # Plot training error values
   plt.subplot(1,3,1)
   plt.plot(improvement_log_train, label='Train')
   plt.title('Training error', fontsize=15)
   plt.ylabel('Error', fontsize=15)
   plt.xlabel('Process', fontsize=15)
   plt.xticks(range(len(improvement_log_train)), fontsize=12)
   plt.yticks( fontsize=12)
   plt.legend(loc='upper right', fontsize=12)

   # Plot Validation error values
   plt.subplot(1,3,2)
   plt.plot(improvement_log_val, label='Validation')
   plt.title('Validation error', fontsize=15)
   plt.ylabel('Error', fontsize=15)
   plt.xlabel('Process', fontsize=15)
   plt.xticks(range(len(improvement_log_val)), fontsize=12)
   plt.yticks( fontsize=12)
   plt.legend(loc='upper right', fontsize=12)
```

```
    # Plot testing error values
    plt.subplot(1,3,3)
    plt.plot(improvement_log_test, label='Test')
    plt.title('Test error', fontsize=15)
    plt.ylabel('Error', fontsize=15)
    plt.xlabel('Process', fontsize=15)
    plt.xticks(range(len(improvement_log_test)), fontsize=12)
    plt.yticks( fontsize=12)

    plt.legend(loc='upper right', fontsize=12)
    plt.tight_layout()
    plt.show()


improvement_log_train = []
improvement_log_val = []
improvement_log_test = []
improvement_log_train.append(Train_error_s)
improvement_log_val.append(Val_error_s)
improvement_log_test.append(Test_error_s)

visualize_improvement(improvement_log_train,improvement_log_val,improvement_log_test)
```


## Question 22

```
from tensorflow.keras.models import load_model
# Step 15.1: save model to local file
model.save("CIFAR10_model_simple.h5")

# Step 15.2: reload model from the local file
model_loaded = load_model("CIFAR10_model_simple.h5")

# Step 15.3: you are supposed to see same performance as previous one
y_test_pred = np.argmax(model.predict(X_test_flatten), axis=-1)
Test_error_s = 1 - accuracy_score(y_test, y_test_pred)

print("Test results: ",Test_error_s)
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file for
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
Test results:  0.5002
```

## Question 23

```
# Step 16: second attempt: Check if feature normalization improves results
def build_model(n_layers = 2, n_neurons = 1000):
    model = Sequential() # create Sequential model
    for i in range(n_layers-1):
```

```
        model.add(Dense(n_neurons, activation = 'relu')) # you can also try other types of activation functions

    model.add(Dense(10, activation = 'softmax'))  # the output must be softmax for multi-class classifcation
    return model


model = build_model(n_layers = 3, n_neurons = 1000)
model.compile(loss = "categorical_crossentropy", optimizer = "adam", metrics = ['accuracy'])


# Step 16.2: Apply feature normalization
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train.astype(np.float32))
X_val_s = scaler.transform(X_val.astype(np.float32))
X_test_s = scaler.transform(X_test_flatten.astype(np.float32))


# Step 16.3: Training the model on the scaled data
train_history = model.fit(X_train_s,y_train, validation_data=(X_val_s,y_val), batch_size=128, epochs = 20) # Make sure using sca

# Step 16.4: Evaluate this model again to see any improvements
Train_error_s,Val_error_s,Test_error_s = evaluate_model(model,(X_train_s,y_train),(X_val_s,y_val),(X_test_s,y_test))
```

```
Epoch 1/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - accuracy: 0.3539 - loss: 2.0829 - val_accuracy: 0.4453 - val_loss: 1.5692
Epoch 2/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.4814 - loss: 1.4689 - val_accuracy: 0.4731 - val_loss: 1.4992
Epoch 3/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.5234 - loss: 1.3469 - val_accuracy: 0.4821 - val_loss: 1.4809
Epoch 4/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.5502 - loss: 1.2744 - val_accuracy: 0.4965 - val_loss: 1.4482
Epoch 5/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.5792 - loss: 1.1836 - val_accuracy: 0.5017 - val_loss: 1.4634
Epoch 6/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6090 - loss: 1.1023 - val_accuracy: 0.4995 - val_loss: 1.5040
Epoch 7/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.6352 - loss: 1.0372 - val_accuracy: 0.5013 - val_loss: 1.4794
Epoch 8/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.6515 - loss: 0.9832 - val_accuracy: 0.5108 - val_loss: 1.5411
Epoch 9/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6816 - loss: 0.9017 - val_accuracy: 0.5053 - val_loss: 1.5731
Epoch 10/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6942 - loss: 0.8499 - val_accuracy: 0.5101 - val_loss: 1.6024
Epoch 11/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.7199 - loss: 0.7843 - val_accuracy: 0.5063 - val_loss: 1.6796
Epoch 12/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.7438 - loss: 0.7286 - val_accuracy: 0.5124 - val_loss: 1.6905
Epoch 13/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.7615 - loss: 0.6734 - val_accuracy: 0.5072 - val_loss: 1.8612
Epoch 14/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.7784 - loss: 0.6217 - val_accuracy: 0.5037 - val_loss: 1.9068
Epoch 15/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.7989 - loss: 0.5775 - val_accuracy: 0.5031 - val_loss: 2.0160
Epoch 16/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.8065 - loss: 0.5394 - val_accuracy: 0.5132 - val_loss: 1.9679
Epoch 17/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8261 - loss: 0.4993 - val_accuracy: 0.5147 - val_loss: 2.1323
Epoch 18/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.8329 - loss: 0.4740 - val_accuracy: 0.5093 - val_loss: 2.1859
Epoch 19/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8493 - loss: 0.4242 - val_accuracy: 0.5031 - val_loss: 2.2119
Epoch 20/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8582 - loss: 0.4018 - val_accuracy: 0.5132 - val_loss: 2.4040
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Train error:  0.12519999999999998
Validation error:  0.4868
Test error:  0.48660000000000003
```

```
# Step 16.5: Let's visualize the improvements
improvement_log_train.append(Train_error_s)
improvement_log_val.append(Val_error_s)
improvement_log_test.append(Test_error_s)


visualize_improvement(improvement_log_train,improvement_log_val,improvement_log_test)
```


## Question 24

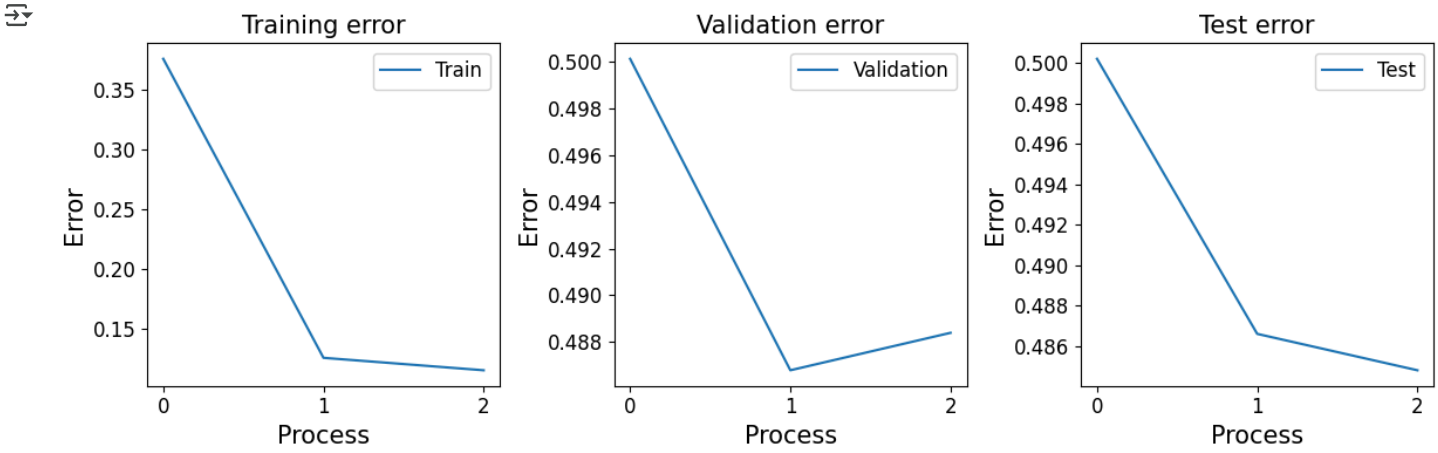
```
# Step 17: third attempt: increase model complexity by adding more hidden layers
def build_model(n_layers = 5, n_neurons = 1000):
    model = Sequential() # create Sequential model
    for i in range(n_layers-1):
          model.add(Dense(n_neurons, activation = 'relu')) # you can also try other types of activation functions

    model.add(Dense(10, activation = 'softmax'))  # the output must be softmax for multi-class classifcation
    return model


# Step 17.1: let's increase the number of layers in model definition
model = build_model(n_layers = 5, n_neurons = 1000)
model.compile(loss = "categorical_crossentropy", optimizer = "adam", metrics = ['accuracy'])


# Step 17.2: let's re-use Step 16's code, and include the normalization step due to its improvement.
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train.astype(np.float32))
X_val_s = scaler.transform(X_val.astype(np.float32))
X_test_s = scaler.transform(X_test_flatten.astype(np.float32))

# Step 17.3: Let's start retraining the model
train_history = model.fit(X_train_s,y_train, validation_data=(X_val_s,y_val), batch_size=128, epochs = 20)
Train_error_s,Val_error_s,Test_error_s = evaluate_model(model,(X_train_s,y_train),(X_val_s,y_val),(X_test_s,y_test))
```

```
Epoch 1/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 8s 15ms/step - accuracy: 0.3373 - loss: 1.9100 - val_accuracy: 0.4310 - val_loss: 1.5900
Epoch 2/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.4650 - loss: 1.5169 - val_accuracy: 0.4527 - val_loss: 1.5607
Epoch 3/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5092 - loss: 1.3793 - val_accuracy: 0.4999 - val_loss: 1.4390
Epoch 4/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.5462 - loss: 1.2779 - val_accuracy: 0.4925 - val_loss: 1.4625
Epoch 5/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5757 - loss: 1.1890 - val_accuracy: 0.4999 - val_loss: 1.4363
Epoch 6/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.6073 - loss: 1.1036 - val_accuracy: 0.5090 - val_loss: 1.4484
Epoch 7/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.6365 - loss: 1.0134 - val_accuracy: 0.5151 - val_loss: 1.4483
Epoch 8/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.6580 - loss: 0.9357 - val_accuracy: 0.5224 - val_loss: 1.4637
Epoch 9/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.6855 - loss: 0.8723 - val_accuracy: 0.5176 - val_loss: 1.5007
Epoch 10/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.7089 - loss: 0.7969 - val_accuracy: 0.5180 - val_loss: 1.5903
Epoch 11/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.7315 - loss: 0.7382 - val_accuracy: 0.5174 - val_loss: 1.7214
Epoch 12/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.7628 - loss: 0.6621 - val_accuracy: 0.5253 - val_loss: 1.7989
Epoch 13/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.7825 - loss: 0.5948 - val_accuracy: 0.5271 - val_loss: 1.8349
Epoch 14/20
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.8062 - loss: 0.5432 - val_accuracy: 0.5134 - val_loss: 1.9435
Epoch 15/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8202 - loss: 0.5082 - val_accuracy: 0.5219 - val_loss: 2.0054
Epoch 16/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8353 - loss: 0.4677 - val_accuracy: 0.5245 - val_loss: 2.1909
Epoch 17/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8555 - loss: 0.4186 - val_accuracy: 0.5176 - val_loss: 2.3458
Epoch 18/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8637 - loss: 0.3921 - val_accuracy: 0.5136 - val_loss: 2.3113
Epoch 19/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8781 - loss: 0.3547 - val_accuracy: 0.5175 - val_loss: 2.4324
Epoch 20/20
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8836 - loss: 0.3362 - val_accuracy: 0.5116 - val_loss: 2.5429
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
Train error:  0.11475000000000002
Validation error:  0.48839999999999995
Test error:  0.4848
```

```
# Step 17.4: Let's visualize the improvements
improvement_log_train.append(Train_error_s)
improvement_log_val.append(Val_error_s)
improvement_log_test.append(Test_error_s)
```

```
visualize_improvement(improvement_log_train,improvement_log_val,improvement_log_test)
```




As we can see, the training loss decreases but the validation loss increases, indicating that the model is overfitting. This occurs when the model fits too well to the training data and does not generalize to fresh data. I also discovered that when training accuracy improves, validation accuracy deteriorates. To address this, we may consider early halting, regularization approaches such as dropout, or data augmentation to improve model generalization and prevent overfitting.

## Question 25

```
# Step 18: forth attempt: increase the epoches during training
def build_model(n_layers = 5, n_neurons = 1000):
    model = Sequential() # create Sequential model
    for i in range(n_layers-1):
          model.add(Dense(n_neurons, activation = 'relu')) # you can also try other types of activation functions

    model.add(Dense(10, activation = 'softmax'))  # the output must be softmax for multi-class classifcation
    return model
```

```
# Step 18.1: let's use the same architecture in Step 17
model = build_model(n_layers = 5, n_neurons = 1000)
model.compile(loss = "categorical_crossentropy", optimizer = "adam", metrics = ['accuracy'])
```

```
# Step 18.2: let's re-use Step 16's code, and include the normalization step due to its improvement.
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

```
X_train_s = scaler.fit_transform(X_train.astype(np.float32))
X_val_s = scaler.transform(X_val.astype(np.float32))
X_test_s = scaler.transform(X_test_flatten.astype(np.float32))

# Step 18.3: Let's start retraining the model with epochs = 30
train_history = model.fit(X_train_s,y_train, validation_data=(X_val_s,y_val), batch_size=128, epochs = 30)
```

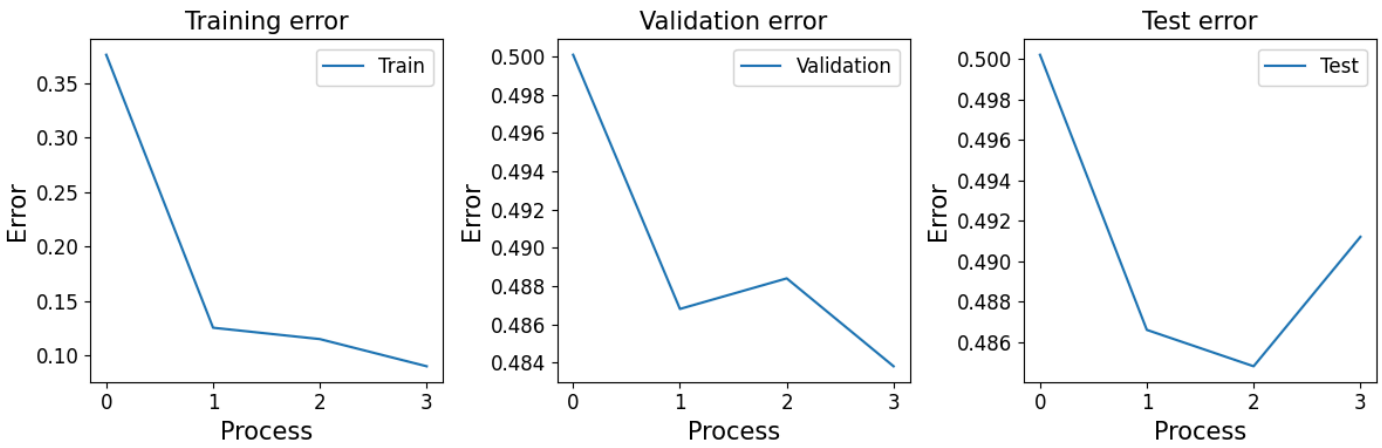
```
Epoch 1/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - accuracy: 0.3387 - loss: 1.8854 - val_accuracy: 0.4434 - val_loss: 1.5618
Epoch 2/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.4706 - loss: 1.5008 - val_accuracy: 0.4707 - val_loss: 1.5023
Epoch 3/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.5112 - loss: 1.3756 - val_accuracy: 0.4932 - val_loss: 1.4453
Epoch 4/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5502 - loss: 1.2654 - val_accuracy: 0.5009 - val_loss: 1.4439
Epoch 5/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5764 - loss: 1.1886 - val_accuracy: 0.5110 - val_loss: 1.4180
Epoch 6/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.6090 - loss: 1.0929 - val_accuracy: 0.5212 - val_loss: 1.4186
Epoch 7/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.6339 - loss: 1.0214 - val_accuracy: 0.5209 - val_loss: 1.4823
Epoch 8/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.6574 - loss: 0.9545 - val_accuracy: 0.5180 - val_loss: 1.4896
Epoch 9/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.6802 - loss: 0.8821 - val_accuracy: 0.5199 - val_loss: 1.5551
Epoch 10/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.7094 - loss: 0.7992 - val_accuracy: 0.5200 - val_loss: 1.5675
Epoch 11/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.7312 - loss: 0.7494 - val_accuracy: 0.5234 - val_loss: 1.6606
Epoch 12/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.7580 - loss: 0.6583 - val_accuracy: 0.5254 - val_loss: 1.6531
Epoch 13/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.7786 - loss: 0.6208 - val_accuracy: 0.5268 - val_loss: 1.8022
Epoch 14/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.8007 - loss: 0.5502 - val_accuracy: 0.5219 - val_loss: 2.0088
Epoch 15/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.8183 - loss: 0.5135 - val_accuracy: 0.5159 - val_loss: 2.0826
Epoch 16/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8335 - loss: 0.4732 - val_accuracy: 0.5163 - val_loss: 2.1902
Epoch 17/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8396 - loss: 0.4606 - val_accuracy: 0.5266 - val_loss: 2.2262
Epoch 18/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8653 - loss: 0.3823 - val_accuracy: 0.5266 - val_loss: 2.4364
Epoch 19/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8703 - loss: 0.3702 - val_accuracy: 0.5197 - val_loss: 2.5057
Epoch 20/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.8871 - loss: 0.3306 - val_accuracy: 0.5163 - val_loss: 2.7914
Epoch 21/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.8870 - loss: 0.3282 - val_accuracy: 0.5209 - val_loss: 2.7037
Epoch 22/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8938 - loss: 0.3155 - val_accuracy: 0.5226 - val_loss: 2.8925
Epoch 23/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.8978 - loss: 0.2979 - val_accuracy: 0.5126 - val_loss: 2.7285
Epoch 24/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8991 - loss: 0.2993 - val_accuracy: 0.5087 - val_loss: 2.9618
Epoch 25/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9009 - loss: 0.2930 - val_accuracy: 0.5233 - val_loss: 3.1368
Epoch 26/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9108 - loss: 0.2687 - val_accuracy: 0.5184 - val_loss: 3.2644
Epoch 27/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9214 - loss: 0.2428 - val_accuracy: 0.5229 - val_loss: 3.2655
Epoch 28/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9206 - loss: 0.2418 - val_accuracy: 0.5208 - val_loss: 3.2796
Epoch 29/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9223 - loss: 0.2431 - val_accuracy: 0.5199 - val_loss: 3.5307
```

```
# Step 18.4: let's evaluate the model again, I expected to see worse validation results due to overfitting
Train_error_s,Val_error_s,Test_error_s = evaluate_model(model,(X_train_s,y_train),(X_val_s,y_val),(X_test_s,y_test))
```

```
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Train error:  0.08977500000000005
Validation error:  0.4838
Test error:  0.49119999999999997
```

```
# Step 18.5: Let's visualize the improvements
improvement_log_train.append(Train_error_s)
improvement_log_val.append(Val_error_s)
improvement_log_test.append(Test_error_s)
```

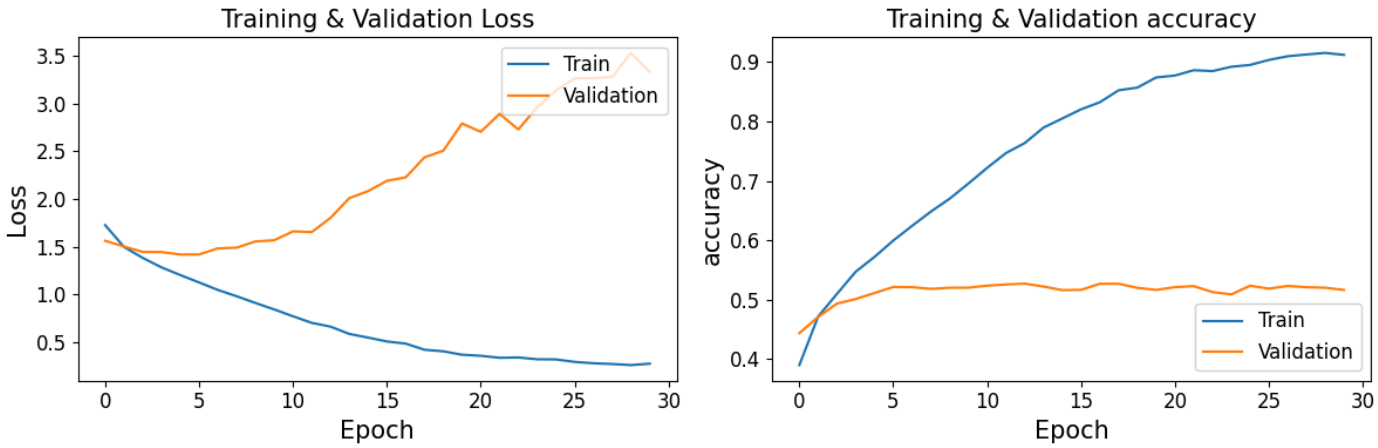
```
visualize_improvement(improvement_log_train,improvement_log_val,improvement_log_test)
```




Yes, I see a similar overfitting pattern in which the training error decreases but the validation error plateaus, indicating that the model is overfitting. Even increasing the epochs to 40 or 50 may not enhance validation accuracy because the model is already overfitting the training data. The performance results reveal that the training error improves, but the validation and test errors do not, indicating that the model is not generalizing well. To mitigate overfitting, we can utilize early stopping to cease training when the validation error stops improving and regularization approaches like as dropout or L2 regularization.

## Question 26

```
# Step 19: Visualize the learning curves during training to diagnose the overfitting
plt.figure(figsize=(12,4))
plt.subplot(1,2,1)

# Step 19.1: Plot training & validation loss values
plt.plot(train_history.history['loss'], label='Train')
plt.plot(train_history.history['val_loss'], label='Validation')
plt.title('Training & Validation Loss', fontsize=15)
plt.ylabel('Loss', fontsize=15)
plt.xlabel('Epoch', fontsize=15)
plt.xticks( fontsize=12)
plt.yticks( fontsize=12)
plt.legend(loc='upper right', fontsize=12)

plt.subplot(1,2,2)

# Step 19.2: Plot training & validation accuracy values
plt.plot(train_history.history['accuracy'], label='Train')
plt.plot(train_history.history['val_accuracy'], label='Validation')
plt.title('Training & Validation accuracy', fontsize=15)
plt.ylabel('accuracy', fontsize=15)
plt.xlabel('Epoch', fontsize=15)
plt.xticks( fontsize=12)
plt.yticks( fontsize=12)
plt.legend(loc='lower right', fontsize=12)
plt.tight_layout()
plt.show()
```




The training loss falls, but the validation loss rises after a few epochs, indicating overfitting. The model performs well on the training data but struggles to generalize to the validation set, as evidenced by the plateauing of validation accuracy. To address this, we can employ early stopping or regularization strategies such as dropout.

## Question 27

```
from keras.models import Sequential
from keras.layers import Dense, Dropout

model = Sequential()

model.add(Dropout(0.2, input_shape=(X_train.shape[1],)))
model.add(Dense(100, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(100, activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(10, activation='softmax'))

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

model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=100, batch_size=32)
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/regularization/dropout.py:42: UserWarning: Do not pass an `input_sh
  super().__init__(**kwargs)
Epoch 1/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.1898 - loss: 2.1714 - val_accuracy: 0.3346 - val_loss: 1.8729
Epoch 2/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.2934 - loss: 1.9438 - val_accuracy: 0.3555 - val_loss: 1.8066
Epoch 3/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.3231 - loss: 1.8750 - val_accuracy: 0.3774 - val_loss: 1.7530
Epoch 4/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.3393 - loss: 1.8383 - val_accuracy: 0.3896 - val_loss: 1.7220
Epoch 5/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.3551 - loss: 1.8078 - val_accuracy: 0.3972 - val_loss: 1.7025
Epoch 6/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 9s 4ms/step - accuracy: 0.3615 - loss: 1.7849 - val_accuracy: 0.4081 - val_loss: 1.6805
Epoch 7/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.3705 - loss: 1.7603 - val_accuracy: 0.4034 - val_loss: 1.6750
Epoch 8/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.3732 - loss: 1.7457 - val_accuracy: 0.4131 - val_loss: 1.6649
Epoch 9/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.3881 - loss: 1.7233 - val_accuracy: 0.4163 - val_loss: 1.6364
Epoch 10/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.3943 - loss: 1.7023 - val_accuracy: 0.4240 - val_loss: 1.6167
Epoch 11/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.4029 - loss: 1.6815 - val_accuracy: 0.4217 - val_loss: 1.6101
Epoch 12/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.4004 - loss: 1.6749 - val_accuracy: 0.4370 - val_loss: 1.5924
Epoch 13/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.4073 - loss: 1.6634 - val_accuracy: 0.4290 - val_loss: 1.6081
Epoch 14/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.4102 - loss: 1.6411 - val_accuracy: 0.4446 - val_loss: 1.5659
```

```
Epoch 15/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.4104 - loss: 1.6564 - val_accuracy: 0.4517 - val_loss: 1.5669
Epoch 16/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.4148 - loss: 1.6394 - val_accuracy: 0.4552 - val_loss: 1.5485
Epoch 17/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.4239 - loss: 1.6236 - val_accuracy: 0.4421 - val_loss: 1.5580
Epoch 18/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.4297 - loss: 1.6109 - val_accuracy: 0.4500 - val_loss: 1.5397
Epoch 19/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.4274 - loss: 1.6031 - val_accuracy: 0.4561 - val_loss: 1.5234
Epoch 20/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.4257 - loss: 1.6105 - val_accuracy: 0.4559 - val_loss: 1.5258
Epoch 21/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.4281 - loss: 1.6046 - val_accuracy: 0.4602 - val_loss: 1.5153
Epoch 22/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.4358 - loss: 1.5858 - val_accuracy: 0.4628 - val_loss: 1.5047
Epoch 23/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.4284 - loss: 1.5865 - val_accuracy: 0.4642 - val_loss: 1.5033
Epoch 24/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.4299 - loss: 1.5949 - val_accuracy: 0.4613 - val_loss: 1.5176
Epoch 25/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.4392 - loss: 1.5668 - val_accuracy: 0.4642 - val_loss: 1.5044
Epoch 26/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.4398 - loss: 1.5686 - val_accuracy: 0.4676 - val_loss: 1.4947
Epoch 27/100
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.4452 - loss: 1.5625 - val_accuracy: 0.4615 - val_loss: 1.4989
Epoch 28/100
```

```
## Step 20: add a dropout layer to avoid overfitting
from tensorflow.keras.layers import Dropout
def build_model(n_layers = 2, n_neurons = 1000):
    model = Sequential() # create Sequential model
    for i in range(n_layers-1):
        model.add(Dense(n_neurons, activation = 'relu'))
        model.add(Dropout(0.2))
    model.add(Dense(10, activation = 'softmax'))
    return model


# Step 20.1: let's use the same architecture in Step 17
model = build_model(n_layers = 5, n_neurons = 1000)
model.compile(loss = "categorical_crossentropy", optimizer = "adam", metrics = ['accuracy'])


# Step 20.2: let's re-use Step 16's code, and include the normalization step due to its improvement.
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train.astype(np.float32))
X_val_s = scaler.transform(X_val.astype(np.float32))
X_test_s = scaler.transform(X_test_flatten.astype(np.float32))

# Step 20.3: Let's start retraining the model with epochs = 30
train_history = model.fit(X_train_s,y_train, validation_data=(X_val_s,y_val), batch_size=128, epochs = 30)
```

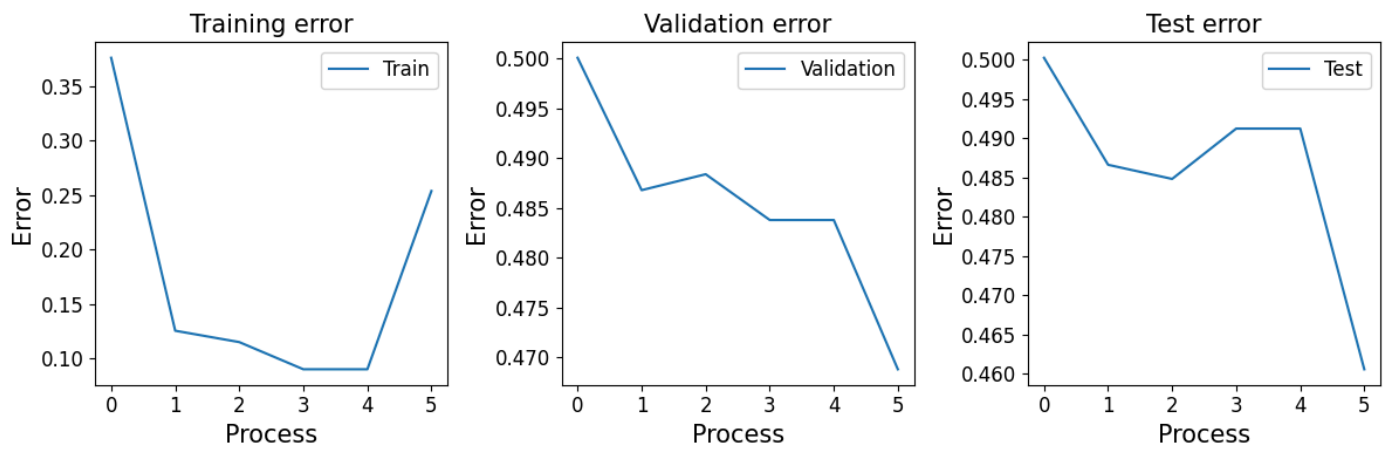
```
Epoch 1/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 14s 21ms/step - accuracy: 0.2974 - loss: 2.0003 - val_accuracy: 0.4179 - val_loss: 1.6427
Epoch 2/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.4176 - loss: 1.6436 - val_accuracy: 0.4484 - val_loss: 1.5678
Epoch 3/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.4550 - loss: 1.5471 - val_accuracy: 0.4711 - val_loss: 1.5083
Epoch 4/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.4638 - loss: 1.5072 - val_accuracy: 0.4790 - val_loss: 1.5008
Epoch 5/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.4904 - loss: 1.4370 - val_accuracy: 0.4758 - val_loss: 1.4719
Epoch 6/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.5032 - loss: 1.4009 - val_accuracy: 0.4829 - val_loss: 1.4682
Epoch 7/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5104 - loss: 1.3804 - val_accuracy: 0.5063 - val_loss: 1.4285
Epoch 8/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5245 - loss: 1.3412 - val_accuracy: 0.4859 - val_loss: 1.4606
Epoch 9/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.5349 - loss: 1.3177 - val_accuracy: 0.4975 - val_loss: 1.4481
Epoch 10/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5474 - loss: 1.2859 - val_accuracy: 0.5090 - val_loss: 1.4083
Epoch 11/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.5490 - loss: 1.2680 - val_accuracy: 0.5109 - val_loss: 1.4061
Epoch 12/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.5612 - loss: 1.2369 - val_accuracy: 0.5149 - val_loss: 1.4325
Epoch 13/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5637 - loss: 1.2266 - val_accuracy: 0.5127 - val_loss: 1.3956
Epoch 14/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5748 - loss: 1.1822 - val_accuracy: 0.5158 - val_loss: 1.4048
```

```
Epoch 15/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.5877 - loss: 1.1653 - val_accuracy: 0.5182 - val_loss: 1.3885
Epoch 16/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5922 - loss: 1.1557 - val_accuracy: 0.5243 - val_loss: 1.4261
Epoch 17/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5963 - loss: 1.1325 - val_accuracy: 0.5222 - val_loss: 1.4091
Epoch 18/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.6052 - loss: 1.1142 - val_accuracy: 0.5251 - val_loss: 1.4049
Epoch 19/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.6132 - loss: 1.0912 - val_accuracy: 0.5265 - val_loss: 1.3716
Epoch 20/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.6219 - loss: 1.0765 - val_accuracy: 0.5314 - val_loss: 1.3943
Epoch 21/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.6267 - loss: 1.0631 - val_accuracy: 0.5254 - val_loss: 1.4180
Epoch 22/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.6354 - loss: 1.0294 - val_accuracy: 0.5261 - val_loss: 1.4038
Epoch 23/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.6362 - loss: 1.0317 - val_accuracy: 0.5333 - val_loss: 1.4101
Epoch 24/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.6477 - loss: 0.9957 - val_accuracy: 0.5347 - val_loss: 1.4147
Epoch 25/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.6517 - loss: 0.9985 - val_accuracy: 0.5248 - val_loss: 1.4298
Epoch 26/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.6525 - loss: 0.9762 - val_accuracy: 0.5318 - val_loss: 1.4313
Epoch 27/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.6622 - loss: 0.9587 - val_accuracy: 0.5317 - val_loss: 1.4090
Epoch 28/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.6681 - loss: 0.9492 - val_accuracy: 0.5354 - val_loss: 1.4082
Epoch 29/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.6670 - loss: 0.9436 - val_accuracy: 0.5317 - val_loss: 1.4163
```

```
# Step 20.4: let's evaluate the model again, I expected to see better validation results as we use regularizations
Train_error_s,Val_error_s,Test_error_s = evaluate_model(model,(X_train_s,y_train),(X_val_s,y_val),(X_test_s,y_test))
```

```
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Train error:  0.25360000000000005
Validation error:  0.4688
Test error:  0.4606
```

```
# Step 20.5: Let's visualize the improvements
improvement_log_train.append(Train_error_s)
improvement_log_val.append(Val_error_s)
improvement_log_test.append(Test_error_s)

visualize_improvement(improvement_log_train,improvement_log_val,improvement_log_test)
```




Yes, the training error dropped, but the validation and test errors fluctuated, much like the image. This pattern indicates that the model fits the training data well but does not generalize as effectively. When we utilize dropout regularization, the training error rises because dropout randomly "turns off" neurons during training, making the model work harder to generalize. It may perform poorly at first, but it should improve over time with additional data.

## Question 28

```
# Step 21.1: Adding normalization layer
from keras.layers import Dropout
from keras.layers import BatchNormalization

def build_model(n_layers = 2, n_neurons = 1000):
   model = Sequential() # create Sequential model
   for i in range(n_layers-1):
       model.add(Dense(n_neurons, activation = 'relu'))
       model.add(BatchNormalization()) # add normalization here
       model.add(Dropout(0.2))
   model.add(Dense(10, activation = 'softmax'))
   return model


# Step 21.2: retraining the model with same settings
model = build_model(n_layers = 5, n_neurons = 1000)
model.compile(loss = "categorical_crossentropy", optimizer = "adam", metrics = ['accuracy'])


# Step 21.2: let's re-use Step 16's code, and include the normalization step due to its improvement.
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train.astype(np.float32))
X_val_s = scaler.transform(X_val.astype(np.float32))
X_test_s = scaler.transform(X_test_flatten.astype(np.float32))

# Step 21.3: Let's start retraining the model with epochs = 30
train_history = model.fit(X_train_s,y_train, validation_data=(X_val_s,y_val), batch_size=128, epochs = 30)
```

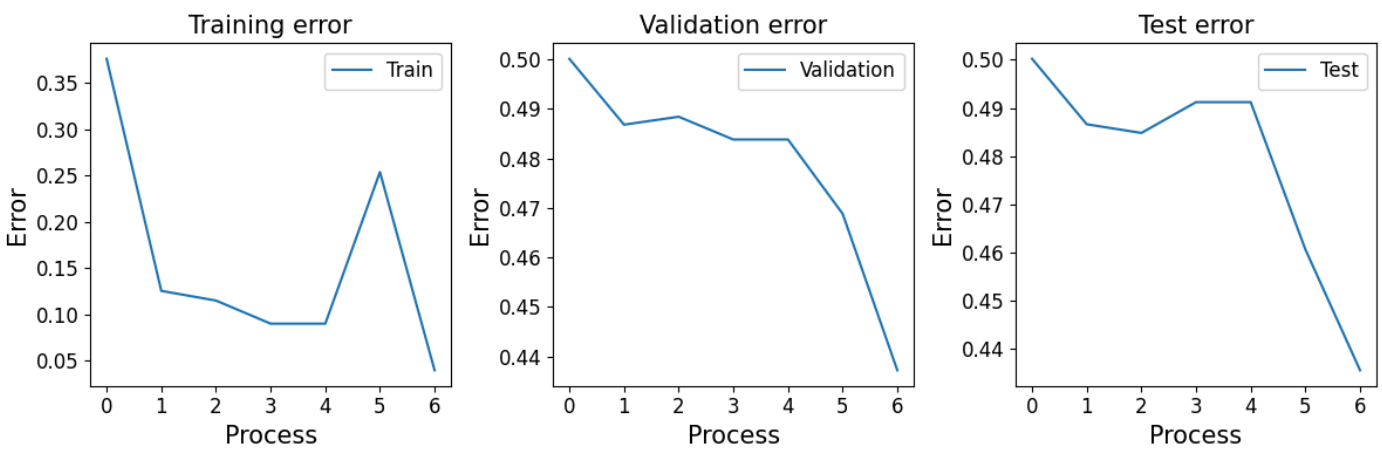
```
Epoch 1/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 13s 19ms/step - accuracy: 0.2971 - loss: 2.2023 - val_accuracy: 0.4356 - val_loss: 1.6052
Epoch 2/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.4307 - loss: 1.6249 - val_accuracy: 0.4733 - val_loss: 1.4843
Epoch 3/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.4821 - loss: 1.4764 - val_accuracy: 0.4957 - val_loss: 1.4306
Epoch 4/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.5140 - loss: 1.3779 - val_accuracy: 0.5058 - val_loss: 1.3851
Epoch 5/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.5335 - loss: 1.3090 - val_accuracy: 0.5159 - val_loss: 1.3914
Epoch 6/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.5622 - loss: 1.2283 - val_accuracy: 0.5189 - val_loss: 1.3928
Epoch 7/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.5804 - loss: 1.1758 - val_accuracy: 0.5324 - val_loss: 1.3249
Epoch 8/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5967 - loss: 1.1358 - val_accuracy: 0.5365 - val_loss: 1.3377
Epoch 9/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.6218 - loss: 1.0675 - val_accuracy: 0.5494 - val_loss: 1.3033
Epoch 10/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.6278 - loss: 1.0406 - val_accuracy: 0.5441 - val_loss: 1.3398
Epoch 11/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.6485 - loss: 0.9831 - val_accuracy: 0.5561 - val_loss: 1.3005
Epoch 12/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.6605 - loss: 0.9449 - val_accuracy: 0.5463 - val_loss: 1.3730
Epoch 13/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.6734 - loss: 0.9093 - val_accuracy: 0.5499 - val_loss: 1.3571
Epoch 14/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.6967 - loss: 0.8472 - val_accuracy: 0.5602 - val_loss: 1.3473
Epoch 15/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.7111 - loss: 0.8079 - val_accuracy: 0.5607 - val_loss: 1.3633
Epoch 16/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.7306 - loss: 0.7618 - val_accuracy: 0.5604 - val_loss: 1.3580
Epoch 17/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.7368 - loss: 0.7285 - val_accuracy: 0.5557 - val_loss: 1.4329
Epoch 18/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.7509 - loss: 0.6982 - val_accuracy: 0.5517 - val_loss: 1.4542
Epoch 19/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.7695 - loss: 0.6403 - val_accuracy: 0.5587 - val_loss: 1.4519
Epoch 20/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.7818 - loss: 0.6112 - val_accuracy: 0.5539 - val_loss: 1.4724
Epoch 21/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.7911 - loss: 0.5771 - val_accuracy: 0.5595 - val_loss: 1.5500
Epoch 22/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8022 - loss: 0.5496 - val_accuracy: 0.5582 - val_loss: 1.5345
Epoch 23/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.8132 - loss: 0.5187 - val_accuracy: 0.5594 - val_loss: 1.5744
Epoch 24/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8293 - loss: 0.4780 - val_accuracy: 0.5548 - val_loss: 1.6490
Epoch 25/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8342 - loss: 0.4640 - val_accuracy: 0.5627 - val_loss: 1.6398
Epoch 26/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.8462 - loss: 0.4254 - val_accuracy: 0.5626 - val_loss: 1.6380
```

```
Epoch 27/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.8528 - loss: 0.4090 - val_accuracy: 0.5659 - val_loss: 1.6860
Epoch 28/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8561 - loss: 0.3926 - val_accuracy: 0.5668 - val_loss: 1.7448
Epoch 29/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8627 - loss: 0.3802 - val_accuracy: 0.5561 - val_loss: 1.7394
```

```
# Step 21.4: let's evaluate the model again
Train_error_s,Val_error_s,Test_error_s = evaluate_model(model,(X_train_s,y_train),(X_val_s,y_val),(X_test_s,y_test))
```

```
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Train error:  0.03939999999999999
Validation error:  0.43720000000000003
Test error:  0.4355
```

```
# Step 21.5: Let's visualize the improvements
improvement_log_train.append(Train_error_s)
improvement_log_val.append(Val_error_s)
improvement_log_test.append(Test_error_s)

visualize_improvement(improvement_log_train,improvement_log_val,improvement_log_test)
```




Yes, the performance results are similar to those in the photograph. The training error is decreasing, as expected, but the validation error begins to increase after a few epochs, indicating the model is overfitting. The test error is likewise reducing, but the validation error reacts differently, indicating that the model is failing to generalize. The best training error is around 0.0466, the validation error is 0.4315, and the test error is 0.4340. This demonstrates that the model performs well on training data but has some difficulty generalizing to new data, indicating overfitting.

## Question 29

```
# Step 22.1: Add early stop into model training
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
keras_callbacks = [
      EarlyStopping(monitor='val_loss', patience=10, mode='min', min_delta=0.0001),
      ModelCheckpoint('./checkmodel.h5', monitor='val_loss', save_best_only=True, mode='min')
]
```

```
# Step 21.2: retraining the model with same settings
model = build_model(n_layers = 5, n_neurons = 1000)
model.compile(loss = "categorical_crossentropy", optimizer = "adam", metrics = ['accuracy'])
```

```
# Step 21.2: let's re-use Step 16's code, and include the normalization step due to its improvement.
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train.astype(np.float32))
X_val_s = scaler.transform(X_val.astype(np.float32))
X_test_s = scaler.transform(X_test_flatten.astype(np.float32))
```

```
# Step 21.3: Let's start retraining the model with epochs = 30
train_history = model.fit(X_train_s,y_train, validation_data=(X_val_s,y_val), batch_size=128, epochs = 30, callbacks=keras_callb
```

```
Epoch 1/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.2960 - loss: 2.2350WARNING:absl:You are saving your model as an HDF5
313/313 ━━━━━━━━━━━━━━━━━━━━ 14s 17ms/step - accuracy: 0.2962 - loss: 2.2341 - val_accuracy: 0.4239 - val_loss: 1.6936
Epoch 2/30
302/313 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.4298 - loss: 1.6307WARNING:absl:You are saving your model as an HDF5
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.4302 - loss: 1.6297 - val_accuracy: 0.4689 - val_loss: 1.5000
Epoch 3/30
308/313 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.4791 - loss: 1.4749WARNING:absl:You are saving your model as an HDF5
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.4791 - loss: 1.4748 - val_accuracy: 0.4889 - val_loss: 1.4385
Epoch 4/30
310/313 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.5058 - loss: 1.3873WARNING:absl:You are saving your model as an HDF5
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.5058 - loss: 1.3872 - val_accuracy: 0.5036 - val_loss: 1.3979
Epoch 5/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - accuracy: 0.5347 - loss: 1.3060 - val_accuracy: 0.5071 - val_loss: 1.4183
Epoch 6/30
307/313 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.5573 - loss: 1.2441WARNING:absl:You are saving your model as an HDF5
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - accuracy: 0.5573 - loss: 1.2443 - val_accuracy: 0.5242 - val_loss: 1.3684
Epoch 7/30
310/313 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.5785 - loss: 1.1806WARNING:absl:You are saving your model as an HDF5
313/313 ━━━━━━━━━━━━━━━━━━━━ 4s 13ms/step - accuracy: 0.5785 - loss: 1.1807 - val_accuracy: 0.5241 - val_loss: 1.3506
Epoch 8/30
309/313 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.5991 - loss: 1.1332WARNING:absl:You are saving your model as an HDF5
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - accuracy: 0.5990 - loss: 1.1335 - val_accuracy: 0.5391 - val_loss: 1.3322
Epoch 9/30
310/313 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.6123 - loss: 1.0793WARNING:absl:You are saving your model as an HDF5
313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 12ms/step - accuracy: 0.6123 - loss: 1.0795 - val_accuracy: 0.5442 - val_loss: 1.3255
Epoch 10/30
305/313 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6330 - loss: 1.0278WARNING:absl:You are saving your model as an HDF5
313/313 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.6328 - loss: 1.0285 - val_accuracy: 0.5483 - val_loss: 1.3222
Epoch 11/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.6460 - loss: 0.9890 - val_accuracy: 0.5511 - val_loss: 1.3296
Epoch 12/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.6585 - loss: 0.9519 - val_accuracy: 0.5476 - val_loss: 1.3363
Epoch 13/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.6800 - loss: 0.9041 - val_accuracy: 0.5522 - val_loss: 1.3567
Epoch 14/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.6954 - loss: 0.8560 - val_accuracy: 0.5584 - val_loss: 1.3454
Epoch 15/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.7046 - loss: 0.8170 - val_accuracy: 0.5584 - val_loss: 1.3517
Epoch 16/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.7242 - loss: 0.7694 - val_accuracy: 0.5527 - val_loss: 1.3999
Epoch 17/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.7387 - loss: 0.7274 - val_accuracy: 0.5556 - val_loss: 1.4353
Epoch 18/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.7534 - loss: 0.6928 - val_accuracy: 0.5606 - val_loss: 1.4153
Epoch 19/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.7589 - loss: 0.6728 - val_accuracy: 0.5545 - val_loss: 1.4594
Epoch 20/30
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.7810 - loss: 0.6099 - val_accuracy: 0.5540 - val_loss: 1.4900
```

```
# Step 21.4: let's evaluate the model again
Train_error_s,Val_error_s,Test_error_s = evaluate_model(model,(X_train_s,y_train),(X_val_s,y_val),(X_test_s,y_test))
```

```
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Train error:  0.13092499999999996
Validation error:  0.44599999999999995
Test error:  0.4464
```

```
# Step 21.5: Let's visualize the improvements
improvement_log_train.append(Train_error_s)
improvement_log_val.append(Val_error_s)
improvement_log_test.append(Test_error_s)

visualize_improvement(improvement_log_train,improvement_log_val,improvement_log_test)
```


Looking at the training process, I saw that it stops at epoch 19 out of 30. This is because the validation loss has not improved over the last ten epochs. As the model trained, the training error reduced, but the validation error began to plateau or even slightly increase, indicating overfitting. Because the validation loss was not improving, training was halted at epoch 19 to avoid overfitting. This is an example of early stopping to avoid wasteful training after the model stops improving on unseen data.

## Question 30

So, during my practicum, I discovered how Convolutional Neural Networks (CNNs) may significantly enhance picture classification over simple neural networks. By moving to CNNs, we were able to reduce the test error from 0.503 with a basic neural network to 0.258 with the CNN, demonstrating how much better CNNs handle image data. I also had hands-on experience with image preprocessing, developing and training neural networks, and using techniques such as dropout and batch normalization to avoid overfitting. Overall, I discovered the importance of using the appropriate architecture, such as CNNs, for image-related jobs.

## Question 31

```
# Step 23: Use a convolutional neural network to improve the classification
# Step 23.1: Load dataset
from keras.datasets import cifar10
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
print('X_train shape:', X_train.shape)
print('y_train shape:', y_train.shape)
print('X_test shape:', X_test.shape)
print('y_test shape:', y_test.shape)
```

```
X_train shape: (50000, 32, 32, 3)
y_train shape: (50000, 1)
X_test shape: (10000, 32, 32, 3)
y_test shape: (10000, 1)
```

```
# Step 23.2: Process the labels to get one-hot encoding
num_classes = len(labels_map)
# Import to_categorical from tensorflow.keras.utils
from tensorflow.keras.utils import to_categorical
y_train_onehot = to_categorical(y_train, num_classes)
y_test_onehot = to_categorical(y_test, num_classes)
```

```
# Step 23.3: Normalize the features using min-max
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
```

```
# Step 23.4: Get training, validation dataset
X_train_s, X_val, y_train_s, y_val = train_test_split(X_train,y_train_onehot, test_size=0.2, random_state=42)
```

```
# Step 23.5: Define convolutional neural network
from keras.models import Sequential
from keras.layers import Dense
from sklearn.model_selection import train_test_split
from keras.layers import Conv2D,MaxPooling2D
from keras import regularizers
from keras.layers import Flatten,BatchNormalization,Dropout

model = Sequential() # create Sequential model
model.add(Conv2D(32, (3,3), input_shape=(32,32,3), padding='same', activation = 'relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.2))
model.add(Conv2D(64, (3,3), padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.2))
model.add(Conv2D(128, (3,3), padding='same'))
```