AML_HW4_Questions_3

April 14, 2024

0.0.1 Name: Apurva Patel

0.0.2 UNI: amp2365

0.1 Setup

[11]:
```
import numpy as np
import matplotlib.pyplot as plt
import pprint
pp = pprint.PrettyPrinter(indent=4)
import warnings
warnings.filterwarnings("ignore")
```

1 Part 1: Neural Network from scratch

For this part, you are not allowed to use any library other than numpy.

In this part, you will implement the forward pass and backward pass (i.e. the derivates of each parameter wrt to the loss) with the network image uploaded.

- The weight matrix for the hidden layer is W1 and has bias b1.
- The weight matrix for the output layer is W2 and has bias b2.
- Activation function is sigmoid for both hidden and output layer
- Loss function is the Mean Squared Error (MSE) loss

Refer to the below dictionary for dimensions for each matrix

[ ]:
```
np.random.seed(0) # don't change this
weights = {
'W1': np.random.randn(3, 2),
'b1': np.zeros(3),
'W2': np.random.randn(3),
'b2': 0,
}
X = np.random.rand(1000,2)
Y = np.random.randint(low=0, high=2, size=(1000,))
```

[ ]:
```
#Sigmoid Function
def sigmoid(z):
    return 1/(1 + np.exp(-z))
```

[ ]:
```
#Implement the forward pass - Z2 and Y
def forward_propagation(X, weights):
    # Z1 -> output of the hidden layer before applying activation
    # H -> output of the  hidden layer after applying activation
    # Z2 -> output of the final layer before applying activation
    # Y -> output of the final layer after applying activation

    Z1 = np.dot(X, weights['W1'].T)  + weights['b1']
    H = sigmoid(Z1)
    # Your code here
    Z2 = np.dot(H, weights['W2']) + weights['b2']
    Y = sigmoid(Z2)

    return Y, Z2, H, Z1
```

[ ]:
```
# Implement the backward pass - dLdZ1, dLdW1, dLdb1
# Y_T are the ground truth labels
def back_propagation(X, Y_T, weights):
    N_points = X.shape[0]

    # forward propagation
    Y, Z2, H, Z1 = forward_propagation(X, weights)
    L = (1/(2*N_points)) * np.sum(np.square(Y - Y_T))

    # back propagation
    dLdY = 1/N_points * (Y - Y_T)
    dLdZ2 = np.multiply(dLdY, (sigmoid(Z2)*(1-sigmoid(Z2))))
    dLdW2 = np.dot(H.T, dLdZ2)

    ones = np.ones((1000))
    dLdb2 = np.dot(ones.T, dLdZ2)
    dLdH = np.dot(dLdZ2.reshape(-1,1), weights['W2'].reshape(-1,1).T)

    # Your code here

    dLdZ1 = np.multiply(dLdH, (sigmoid(Z1) * (1 - sigmoid(Z1))))
    dLdW1 = np.dot(dLdZ1.T, X)
    dLdb1 = np.dot(ones, dLdZ1)

    gradients = {
        'W1': dLdW1,
        'b1': dLdb1,
        'W2': dLdW2,
        'b2': dLdb2,
    }

    return gradients, L
```

```
[ ]: gradients, L = back_propagation(X, Y, weights)
     print(L)
```

```
0.1332476222330792
```

```
[ ]: pp.pprint(gradients)
```

```
{   'W1': array([[ 0.00244596,  0.00262019],
       [-0.00030765, -0.00024188],
       [-0.00034768, -0.000372  ]]),
    'W2': array([0.02216011, 0.02433097, 0.01797174]),
    'b1': array([ 0.00492577, -0.00058023, -0.00065977]),
    'b2': 0.029249230265318688}
```

Your answers should be close to L = 0.133 and 'b1': array([0.00492, -0.000581, -0.00066]).

You will be graded based on your implementation and outputs for L, W1, W2 b1, and b2

2 Part 2: Neural network to classify images: CIFAR-10

CIFAR-10 is a dataset of 60,000 color images (32 by 32 resolution) across 10 classes - airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

The train/test split is 50k/10k.

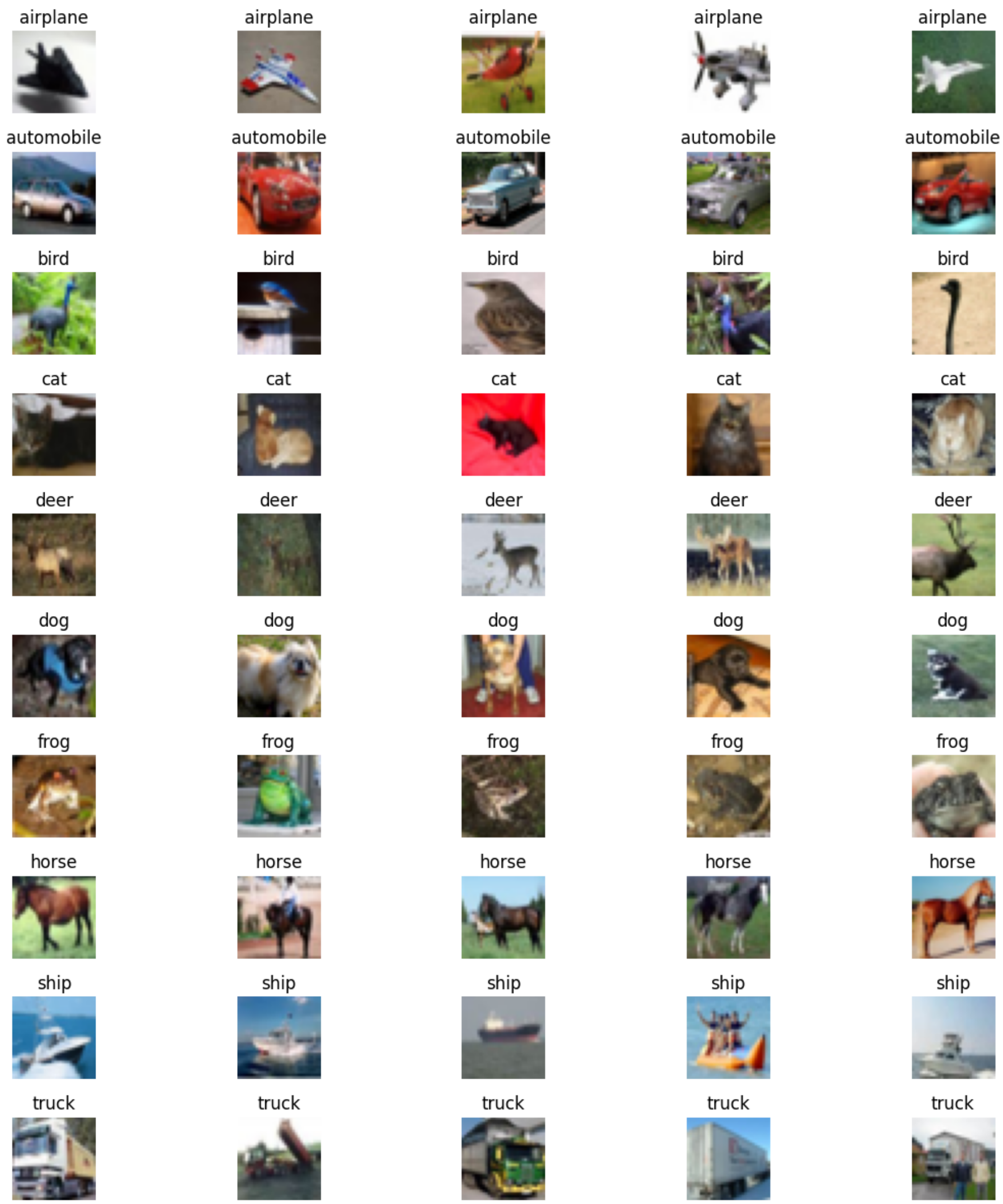
```
[ ]: from tensorflow.keras.datasets import cifar10 #Code to load data, do not change
     (x_dev, y_dev), (x_test, y_test) = cifar10.load_data()

     LABELS =␣
      ↪['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
```

2.0.1 2.1 Plot 50 samples from each class/label from train set on a 10*5 subplot

```
[ ]: #Your code here

     class_indices = {i: [] for i in range(10)}
     for i, label in enumerate(y_dev):
         class_indices[label[0]].append(i)

     fig, axes = plt.subplots(10, 5, figsize=(12, 12))
     for i in range(10):
         for j in range(5):
             idx = class_indices[i][j]
             axes[i, j].imshow(x_dev[idx])
             axes[i, j].set_title(LABELS[i])
             axes[i, j].axis('off')
```

```
plt.tight_layout()
plt.show()
```



2.0.2 2.2 Preparing the dataset for NN

1) Print the shapes - , , ,

2) Flatten the images into one-dimensional vectors and again print the shapes of ,
3) Standardize the development and test sets.
4) One hot encode your labels
5) Train-test split your development set into train and validation sets (80:20 ratio).

[ ]:
```python
#Your code here
print("Shapes before preprocessing:")
print("x_dev shape:", x_dev.shape)
print("y_dev shape:", y_dev.shape)
print("x_test shape:", x_test.shape)
print("y_test shape:", y_test.shape)
```

```
Shapes before preprocessing:
x_dev shape: (50000, 32, 32, 3)
y_dev shape: (50000, 1)
x_test shape: (10000, 32, 32, 3)
y_test shape: (10000, 1)
```

[ ]:
```python
#Your code here
x_dev_flat = x_dev.reshape(x_dev.shape[0], -1)
x_test_flat = x_test.reshape(x_test.shape[0], -1)
```

[ ]:
```python
print("\nShapes after flattening:")
print("x_dev shape (flattened):", x_dev_flat.shape)
print("x_test shape (flattened):", x_test_flat.shape)
```

```

Shapes after flattening:
x_dev shape (flattened): (50000, 3072)
x_test shape (flattened): (10000, 3072)
```

[ ]:
```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

[ ]:
```python
#Your code here
scaler = StandardScaler()
x_dev_scaled = scaler.fit_transform(x_dev_flat)
x_test_scaled = scaler.transform(x_test_flat)
```

[ ]:
```python
#Your code here
encoder = OneHotEncoder(sparse=False)
y_dev_encoded = encoder.fit_transform(y_dev)
y_test_encoded = encoder.transform(y_test)
```

[ ]:
```python
#Your code here
```

```
x_train, x_val, y_train, y_val = train_test_split(x_dev_scaled, y_dev_encoded,␣
↪test_size=0.2, random_state=42)

print("\nShapes after train-test split:")
print("x_train shape:", x_train.shape)
print("y_train shape:", y_train.shape)
print("x_val shape:", x_val.shape)
print("y_val shape:", y_val.shape)
```

```
Shapes after train-test split:
x_train shape: (40000, 3072)
y_train shape: (40000, 10)
x_val shape: (10000, 3072)
y_val shape: (10000, 10)
```

2.0.3 2.3 Build the feed forward network with the below specifications

First layer size = 128

hidden layer size = 64

last layer size = Figure this out from the data!

The last layer size is 10 as there are 10 classes in the dataset

[ ]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten

#Your code here
input_size = x_train.shape[1]
first_layer_size = 128
hidden_layer_size = 64
output_size = y_train.shape[1]

# the model
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(first_layer_size, activation='relu',␣
↪input_shape=(input_size,)),
    tf.keras.layers.Dense(hidden_layer_size, activation='relu'),
    tf.keras.layers.Dense(output_size, activation='softmax')
])

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

2.0.4 2.4 Print out the model summary. Mention the number of parameters for each layer.

[ ]:
```
#Your code here
model.summary()
```

```
Model: "sequential"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               393344

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 10)                650

=================================================================
Total params: 402250 (1.53 MB)
Trainable params: 402250 (1.53 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

2.0.5 2.5 Do you think the number of parameters is dependent on the image height and width?

- Yes, the number of parameters in a neural network is indirectly influenced (sort opf) by the image height and width.
- In a fully connected layer, each neuron connects to every neuron in the previous layer, resulting in a large number of parameters. Therefore, larger images with more pixels will lead to more parameters in the network.
- However, the direct dependency is on the number of neurons in each layer and the connections between them, rather than the specific dimensions of the image.

[ ]:
```
#Your comments here: Wrote in markdown above
```

Printing out your model's output on first train sample. This will confirm if your dimensions are correctly set up. The sum of this output should equal to 1.

[ ]:
```
#modify name of X_train based on your requirement

model.compile()
output = model.predict(x_train[0].reshape(1,-1))

print("Output: {:.2f}".format(sum(output[0])))
```

```
1/1 [==============================] - 1s 786ms/step
Output: 1.00
```

2.0.6 2.6 Using the right metric and the right loss function, with Adam as the optimizer, train your model for 20 epochs.

[ ]:
```
#Your code here
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(x_train, y_train, epochs=20, validation_data=(x_val, y_val))

test_loss, test_accuracy = model.evaluate(x_test_scaled, y_test_encoded)

print("\nTest Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

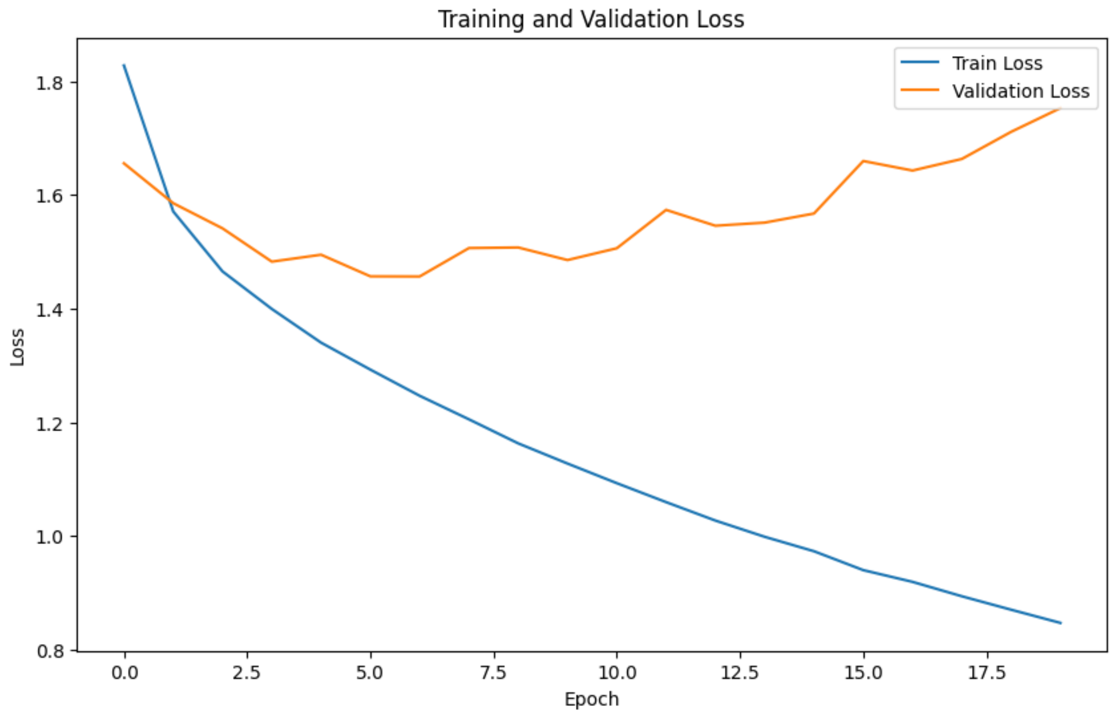
```
Epoch 1/20
1250/1250 [==============================] - 8s 5ms/step - loss: 1.8277 -
accuracy: 0.3670 - val_loss: 1.6556 - val_accuracy: 0.4169
Epoch 2/20
1250/1250 [==============================] - 5s 4ms/step - loss: 1.5710 -
accuracy: 0.4466 - val_loss: 1.5855 - val_accuracy: 0.4391
Epoch 3/20
1250/1250 [==============================] - 5s 4ms/step - loss: 1.4656 -
accuracy: 0.4819 - val_loss: 1.5413 - val_accuracy: 0.4611
Epoch 4/20
1250/1250 [==============================] - 5s 4ms/step - loss: 1.3996 -
accuracy: 0.5053 - val_loss: 1.4827 - val_accuracy: 0.4824
Epoch 5/20
1250/1250 [==============================] - 5s 4ms/step - loss: 1.3403 -
accuracy: 0.5274 - val_loss: 1.4949 - val_accuracy: 0.4868
Epoch 6/20
1250/1250 [==============================] - 5s 4ms/step - loss: 1.2928 -
accuracy: 0.5427 - val_loss: 1.4569 - val_accuracy: 0.4885
Epoch 7/20
1250/1250 [==============================] - 5s 4ms/step - loss: 1.2469 -
accuracy: 0.5594 - val_loss: 1.4566 - val_accuracy: 0.4946
Epoch 8/20
1250/1250 [==============================] - 5s 4ms/step - loss: 1.2054 -
accuracy: 0.5758 - val_loss: 1.5066 - val_accuracy: 0.4882
Epoch 9/20
1250/1250 [==============================] - 5s 4ms/step - loss: 1.1631 -
accuracy: 0.5897 - val_loss: 1.5076 - val_accuracy: 0.4899
Epoch 10/20
1250/1250 [==============================] - 5s 4ms/step - loss: 1.1277 -
accuracy: 0.6027 - val_loss: 1.4855 - val_accuracy: 0.4925
Epoch 11/20
1250/1250 [==============================] - 5s 4ms/step - loss: 1.0932 -
accuracy: 0.6122 - val_loss: 1.5063 - val_accuracy: 0.4946
```

```
Epoch 12/20
1250/1250 [==============================] - 5s 4ms/step - loss: 1.0599 -
accuracy: 0.6261 - val_loss: 1.5737 - val_accuracy: 0.4999
Epoch 13/20
1250/1250 [==============================] - 5s 4ms/step - loss: 1.0274 -
accuracy: 0.6387 - val_loss: 1.5459 - val_accuracy: 0.5010
Epoch 14/20
1250/1250 [==============================] - 6s 5ms/step - loss: 0.9989 -
accuracy: 0.6471 - val_loss: 1.5513 - val_accuracy: 0.5032
Epoch 15/20
1250/1250 [==============================] - 4s 4ms/step - loss: 0.9735 -
accuracy: 0.6535 - val_loss: 1.5673 - val_accuracy: 0.5007
Epoch 16/20
1250/1250 [==============================] - 5s 4ms/step - loss: 0.9403 -
accuracy: 0.6658 - val_loss: 1.6597 - val_accuracy: 0.4894
Epoch 17/20
1250/1250 [==============================] - 5s 4ms/step - loss: 0.9196 -
accuracy: 0.6756 - val_loss: 1.6429 - val_accuracy: 0.4854
Epoch 18/20
1250/1250 [==============================] - 6s 5ms/step - loss: 0.8943 -
accuracy: 0.6824 - val_loss: 1.6633 - val_accuracy: 0.4903
Epoch 19/20
1250/1250 [==============================] - 5s 4ms/step - loss: 0.8707 -
accuracy: 0.6888 - val_loss: 1.7110 - val_accuracy: 0.4946
Epoch 20/20
1250/1250 [==============================] - 5s 4ms/step - loss: 0.8475 -
accuracy: 0.6997 - val_loss: 1.7527 - val_accuracy: 0.4838
313/313 [==============================] - 1s 3ms/step - loss: 1.7657 -
accuracy: 0.4821

Test Loss: 1.7657122611999512
Test Accuracy: 0.4821000099182129
```

2.0.7 2.7 Plot the training curves as described below

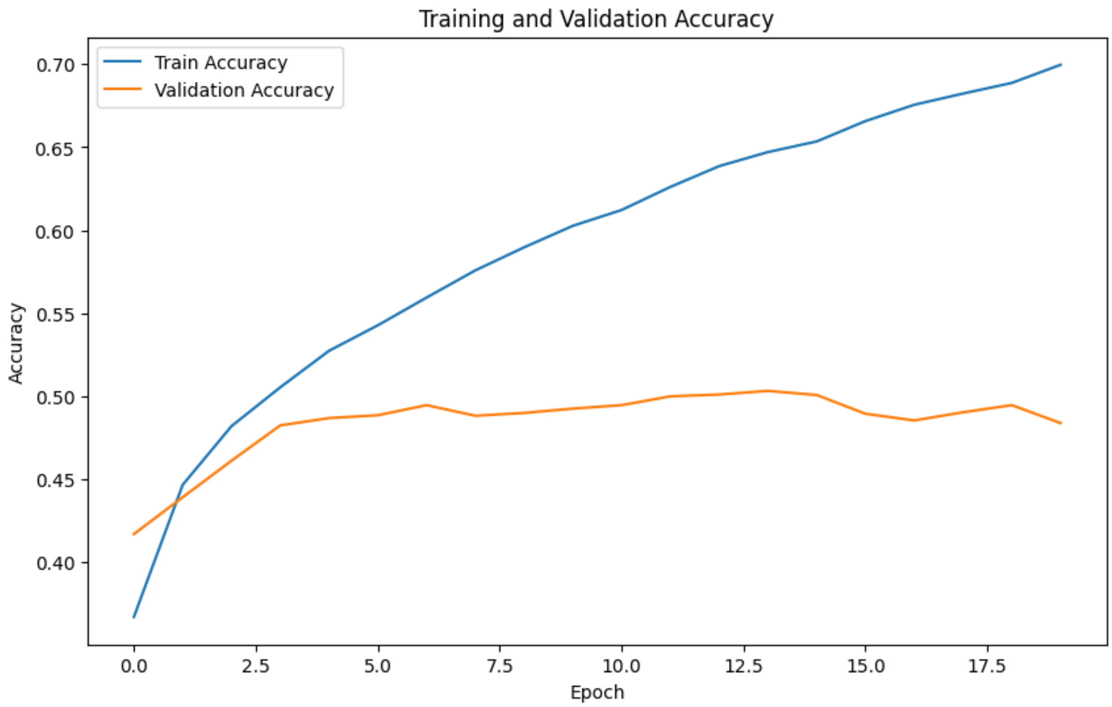
2.7.1 Display the train vs validation loss over each epoch

[ ]:
```python
#Your code here
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



2.7.2 Display the train vs validation accuracy over each epoch

[ ]:
```
#Your code here
plt.figure(figsize=(10, 6))
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



2.0.8 2.8 Finally, report the metric chosen on test set

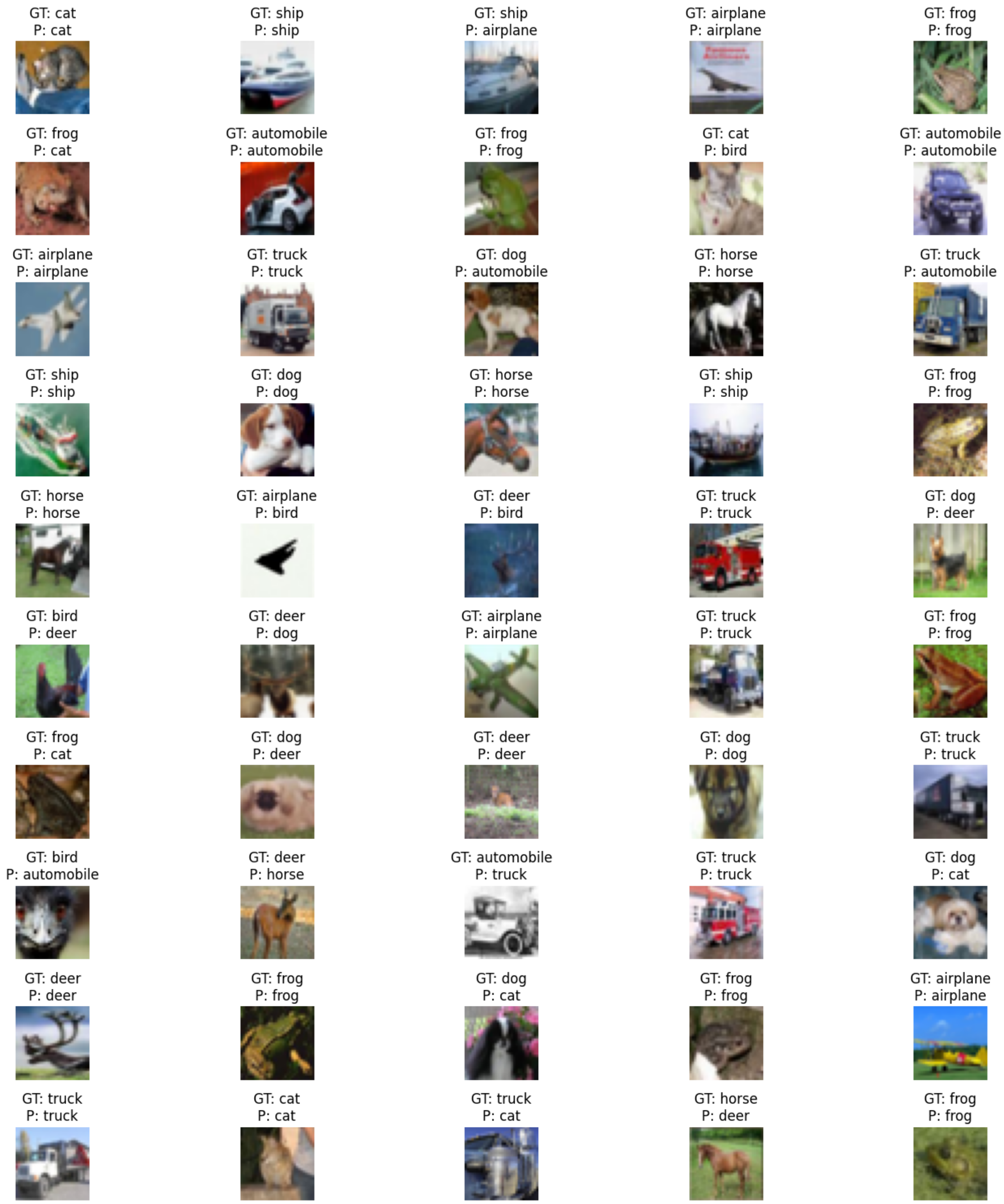
[ ]:
```
#Your code here
print("Test Accuracy:", test_accuracy)
```

```
Test Accuracy: 0.4821000099182129
```

2.0.9 2.9 Plot the first 50 samples of test dataset on a 10*5 subplot and this time label the images with both the ground truth (GT) and predicted class (P). (Make sure you predict the class with the improved model)

[ ]:
```
#Your code here
predictions = model.predict(x_test_scaled[:50])
predicted_classes = np.argmax(predictions, axis=1)

plt.figure(figsize=(15, 15))
for i in range(50):
    plt.subplot(10, 5, i + 1)
    plt.imshow(x_test[i])
    plt.title(f"GT: {LABELS[y_test[i][0]]}\nP: {LABELS[predicted_classes[i]]}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

```
2/2 [==============================] - 0s 8ms/step
```



3 Part 3 - Convolutional Neural Networks

In this part of the homework, we will build and train a classical convolutional neural network on the CIFAR Dataset

[1]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPool2D, Flatten
import pandas as pd
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import cifar10
```

[2]:
```
#Code to load the dataset - Do not change
(x_dev, y_dev), (x_test, y_test) = cifar10.load_data()
print("x_dev: {},y_dev: {},x_test: {},y_test: {}".format(x_dev.shape, y_dev.
 ↪shape, x_test.shape, y_test.shape))

x_dev, x_test = x_dev.astype('float32'), x_test.astype('float32')
x_dev = x_dev/255.0
x_test = x_test/255.0


from sklearn.model_selection import train_test_split

X_train, X_val, y_train, y_val = train_test_split(x_dev, y_dev,test_size = 0.2,␣
 ↪random_state = 42)
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 7s 0us/step
x_dev: (50000, 32, 32, 3),y_dev: (50000, 1),x_test: (10000, 32, 32, 3),y_test:
(10000, 1)
```

[3]:
```
# OHE the labels for y
y_train = to_categorical(y_train, num_classes=10)
y_val = to_categorical(y_val, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
```

[4]:
```
print("Shape of X_train:", X_train.shape)
print("Shape of X_val:", X_val.shape)

print("Shape of x_test:", x_test.shape)
```

```
Shape of X_train: (40000, 32, 32, 3)
Shape of X_val: (10000, 32, 32, 3)
Shape of x_test: (10000, 32, 32, 3)
```

[5]:
```
print("Shape of y_train:", y_train.shape)
print("Shape of y_val:", y_val.shape)

print("Shape of y_test:", y_test.shape)
```

```
Shape of y_train: (40000, 10)
Shape of y_val: (10000, 10)
Shape of y_test: (10000, 10)
```

3.0.1 3.1 We will be implementing one of the first CNN models put forward by Yann LeCunn, which is commonly referred to as LeNet-5. The network has the following layers:

1) 2D convolutional layer with 6 filters, 5x5 kernel, stride of 1, 0 padding, ReLU activation
2) Maxpooling layer of 2x2
3) 2D convolutional layer with 16 filters, 5x5 kernel, stride of 1, 0 padding, ReLU activation
4) Maxpooling layer of 2x2
5) Flatten the convolution output to feed it into fully connected layers
6) A fully connected layer with 120 units, ReLU activation
7) A fully connected layer with 84 units, ReLU activation
8) The output layer where each unit respresents the probability of image being in that category. What activation function should you use in this layer? (You should know this)

[6]:
```
#Your code here
num_classes=10

cnn = Sequential()

#1
cnn.add(Conv2D(6, kernel_size=(5, 5), strides=(1, 1), padding='valid',␣
↪activation='relu', input_shape=(32, 32, 3)))

# 2
cnn.add(MaxPool2D(pool_size=(2, 2)))

# 3
cnn.add(Conv2D(16, kernel_size=(5, 5), strides=(1, 1), padding='valid',␣
↪activation='relu'))

# 4
cnn.add(MaxPool2D(pool_size=(2, 2)))

# 5
cnn.add(Flatten())

# 6
cnn.add(Dense(120, activation='relu'))

# 7
cnn.add(Dense(84, activation='relu'))

# 8
cnn.add(Dense(10, activation='softmax'))
```

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

3.0.2 3.2 Report the model summary

[7]:
```
#Your code here
cnn.summary()
```

```
Model: "sequential"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 6)         456

 max_pooling2d (MaxPooling2  (None, 14, 14, 6)         0
 D)

 conv2d_1 (Conv2D)           (None, 10, 10, 16)        2416

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 16)          0
 g2D)

 flatten (Flatten)           (None, 400)               0

 dense (Dense)               (None, 120)               48120

 dense_1 (Dense)             (None, 84)                10164

 dense_2 (Dense)             (None, 10)                850

=================================================================
Total params: 62006 (242.21 KB)
Trainable params: 62006 (242.21 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

3.0.3 3.3 Model Training

1) Train the model for 20 epochs. In each epoch, record the loss and metric (chosen in part 3) scores for both train and validation sets.

2) Plot separate plots for:

- displaying train vs validation loss over each epoch
- displaying train vs validation accuracy over each epoch

3) Report the model performance on the test set. Feel free to tune the hyperparameters such as batch size and optimizers to achieve better performance.

[8]:
```python
#Your code here

history = cnn.fit(X_train, y_train, epochs=20, validation_data=(X_val, y_val))
```

```
Epoch 1/20
1250/1250 [==============================] - 17s 8ms/step - loss: 1.6520 -
accuracy: 0.3947 - val_loss: 1.4595 - val_accuracy: 0.4588
Epoch 2/20
1250/1250 [==============================] - 11s 9ms/step - loss: 1.3653 -
accuracy: 0.5061 - val_loss: 1.3515 - val_accuracy: 0.5101
Epoch 3/20
1250/1250 [==============================] - 13s 11ms/step - loss: 1.2607 -
accuracy: 0.5462 - val_loss: 1.2648 - val_accuracy: 0.5535
Epoch 4/20
1250/1250 [==============================] - 14s 11ms/step - loss: 1.1890 -
accuracy: 0.5770 - val_loss: 1.2442 - val_accuracy: 0.5554
Epoch 5/20
1250/1250 [==============================] - 13s 10ms/step - loss: 1.1174 -
accuracy: 0.6039 - val_loss: 1.2069 - val_accuracy: 0.5725
Epoch 6/20
1250/1250 [==============================] - 20s 16ms/step - loss: 1.0641 -
accuracy: 0.6223 - val_loss: 1.2499 - val_accuracy: 0.5701
Epoch 7/20
1250/1250 [==============================] - 12s 10ms/step - loss: 1.0162 -
accuracy: 0.6383 - val_loss: 1.1208 - val_accuracy: 0.6090
Epoch 8/20
1250/1250 [==============================] - 10s 8ms/step - loss: 0.9709 -
accuracy: 0.6552 - val_loss: 1.1342 - val_accuracy: 0.6030
Epoch 9/20
1250/1250 [==============================] - 11s 8ms/step - loss: 0.9256 -
accuracy: 0.6714 - val_loss: 1.1517 - val_accuracy: 0.6099
Epoch 10/20
1250/1250 [==============================] - 12s 9ms/step - loss: 0.8941 -
accuracy: 0.6832 - val_loss: 1.1419 - val_accuracy: 0.6163
Epoch 11/20
1250/1250 [==============================] - 11s 9ms/step - loss: 0.8574 -
accuracy: 0.6977 - val_loss: 1.1363 - val_accuracy: 0.6141
Epoch 12/20
1250/1250 [==============================] - 9s 7ms/step - loss: 0.8240 -
accuracy: 0.7082 - val_loss: 1.1504 - val_accuracy: 0.6144
Epoch 13/20
1250/1250 [==============================] - 10s 8ms/step - loss: 0.7904 -
accuracy: 0.7196 - val_loss: 1.1537 - val_accuracy: 0.6196
Epoch 14/20
1250/1250 [==============================] - 12s 10ms/step - loss: 0.7590 -
accuracy: 0.7306 - val_loss: 1.2035 - val_accuracy: 0.6138
Epoch 15/20
```

```
1250/1250 [==============================] - 12s 9ms/step - loss: 0.7358 -
accuracy: 0.7389 - val_loss: 1.2005 - val_accuracy: 0.6164
Epoch 16/20
1250/1250 [==============================] - 10s 8ms/step - loss: 0.7094 -
accuracy: 0.7494 - val_loss: 1.2299 - val_accuracy: 0.6132
Epoch 17/20
1250/1250 [==============================] - 10s 8ms/step - loss: 0.6804 -
accuracy: 0.7567 - val_loss: 1.2469 - val_accuracy: 0.6104
Epoch 18/20
1250/1250 [==============================] - 12s 10ms/step - loss: 0.6566 -
accuracy: 0.7667 - val_loss: 1.3132 - val_accuracy: 0.6078
Epoch 19/20
1250/1250 [==============================] - 9s 7ms/step - loss: 0.6327 -
accuracy: 0.7735 - val_loss: 1.2992 - val_accuracy: 0.6099
Epoch 20/20
1250/1250 [==============================] - 8s 6ms/step - loss: 0.6151 -
accuracy: 0.7803 - val_loss: 1.3588 - val_accuracy: 0.6026
```

[12]:

```python
#Your code here

# train vs. valid loss
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# train vs. valid accuracy
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
```

[ ]:
```
# y_test_encoded = to_categorical(y_test, num_classes=10)
```

[ ]:
```
# y_test_encoded.shape
```

[13]:
```
#Your code here
test_loss, test_accuracy = cnn.evaluate(x_test, y_test)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
313/313 [==============================] - 2s 5ms/step - loss: 1.4129 -
accuracy: 0.5915
Test Loss: 1.4128663539886475
Test Accuracy: 0.5914999842643738
```

3.0.4 3.4 Overfitting

1) To overcome overfitting, we will train the network again with dropout this time. For hidden layers use dropout probability of 0.3. Train the model again for 20 epochs. Report model performance on test set.

Plot separate plots for:

- displaying train vs validation loss over each epoch
- displaying train vs validation accuracy over each epoch

2) This time, let's apply a batch normalization after every hidden layer, train the model for 20 epochs, report model performance on test set as above.

Plot separate plots for:

- displaying train vs validation loss over each epoch

- displaying train vs validation accuracy over each epoch

3) Compare batch normalization technique with the original model and with dropout, which technique do you think helps with overfitting better?

[14]:
```
from tensorflow.keras.layers import Dropout, BatchNormalization, MaxPooling2D
```

3.4.1 Dropout

[15]:
```
#Your code here
cnn_dropout = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    Flatten(),
    Dropout(0.3),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

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

cnn_dropout.summary()
```

```
Model: "sequential_1"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 30, 30, 32)        896

 max_pooling2d_2 (MaxPoolin  (None, 15, 15, 32)        0
 g2D)

 conv2d_3 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_3 (MaxPoolin  (None, 6, 6, 64)          0
 g2D)

 conv2d_4 (Conv2D)           (None, 4, 4, 64)          36928

 flatten_1 (Flatten)         (None, 1024)              0

 dropout (Dropout)           (None, 1024)              0

 dense_3 (Dense)             (None, 64)                65600
```

```
 dense_4 (Dense)             (None, 10)                650

=================================================================
Total params: 122570 (478.79 KB)
Trainable params: 122570 (478.79 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

[16]:
```
dropout_history = cnn_dropout.fit(X_train, y_train, epochs=20,␣
 ↪validation_data=(X_val, y_val))
```

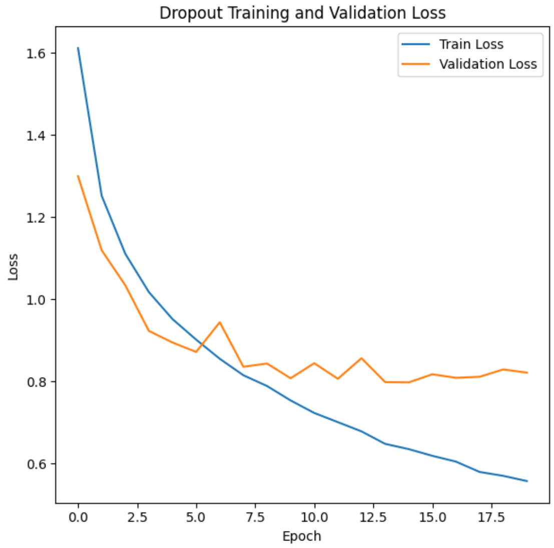
```
Epoch 1/20
1250/1250 [==============================] - 16s 9ms/step - loss: 1.6119 -
accuracy: 0.4065 - val_loss: 1.2999 - val_accuracy: 0.5349
Epoch 2/20
1250/1250 [==============================] - 10s 8ms/step - loss: 1.2525 -
accuracy: 0.5529 - val_loss: 1.1198 - val_accuracy: 0.6108
Epoch 3/20
1250/1250 [==============================] - 12s 10ms/step - loss: 1.1111 -
accuracy: 0.6087 - val_loss: 1.0343 - val_accuracy: 0.6196
Epoch 4/20
1250/1250 [==============================] - 12s 10ms/step - loss: 1.0175 -
accuracy: 0.6402 - val_loss: 0.9230 - val_accuracy: 0.6742
Epoch 5/20
1250/1250 [==============================] - 12s 10ms/step - loss: 0.9517 -
accuracy: 0.6675 - val_loss: 0.8948 - val_accuracy: 0.6859
Epoch 6/20
1250/1250 [==============================] - 10s 8ms/step - loss: 0.9018 -
accuracy: 0.6851 - val_loss: 0.8718 - val_accuracy: 0.6935
Epoch 7/20
1250/1250 [==============================] - 10s 8ms/step - loss: 0.8551 -
accuracy: 0.6995 - val_loss: 0.9440 - val_accuracy: 0.6690
Epoch 8/20
1250/1250 [==============================] - 9s 7ms/step - loss: 0.8150 -
accuracy: 0.7149 - val_loss: 0.8357 - val_accuracy: 0.7105
Epoch 9/20
1250/1250 [==============================] - 13s 10ms/step - loss: 0.7888 -
accuracy: 0.7215 - val_loss: 0.8435 - val_accuracy: 0.7049
Epoch 10/20
1250/1250 [==============================] - 11s 9ms/step - loss: 0.7536 -
accuracy: 0.7335 - val_loss: 0.8077 - val_accuracy: 0.7220
Epoch 11/20
1250/1250 [==============================] - 9s 7ms/step - loss: 0.7231 -
accuracy: 0.7437 - val_loss: 0.8444 - val_accuracy: 0.7141
Epoch 12/20
1250/1250 [==============================] - 9s 7ms/step - loss: 0.7007 -
accuracy: 0.7506 - val_loss: 0.8064 - val_accuracy: 0.7230
```

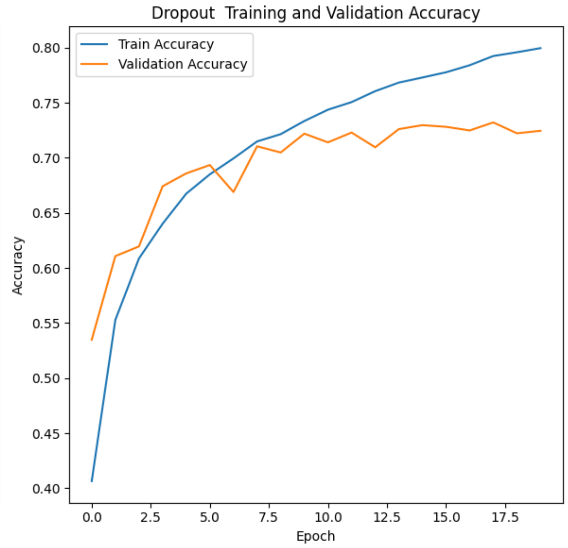
```
Epoch 13/20
1250/1250 [==============================] - 10s 8ms/step - loss: 0.6782 -
accuracy: 0.7606 - val_loss: 0.8566 - val_accuracy: 0.7096
Epoch 14/20
1250/1250 [==============================] - 8s 6ms/step - loss: 0.6478 -
accuracy: 0.7684 - val_loss: 0.7984 - val_accuracy: 0.7261
Epoch 15/20
1250/1250 [==============================] - 10s 8ms/step - loss: 0.6350 -
accuracy: 0.7730 - val_loss: 0.7976 - val_accuracy: 0.7297
Epoch 16/20
1250/1250 [==============================] - 11s 9ms/step - loss: 0.6187 -
accuracy: 0.7778 - val_loss: 0.8173 - val_accuracy: 0.7282
Epoch 17/20
1250/1250 [==============================] - 9s 7ms/step - loss: 0.6045 -
accuracy: 0.7841 - val_loss: 0.8087 - val_accuracy: 0.7249
Epoch 18/20
1250/1250 [==============================] - 9s 7ms/step - loss: 0.5794 -
accuracy: 0.7925 - val_loss: 0.8113 - val_accuracy: 0.7322
Epoch 19/20
1250/1250 [==============================] - 13s 10ms/step - loss: 0.5699 -
accuracy: 0.7959 - val_loss: 0.8292 - val_accuracy: 0.7223
Epoch 20/20
1250/1250 [==============================] - 11s 9ms/step - loss: 0.5572 -
accuracy: 0.7996 - val_loss: 0.8214 - val_accuracy: 0.7246
```

[17]:

```
#Your code here

# train vs val loss
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(dropout_history.history['loss'], label='Train Loss')
plt.plot(dropout_history.history['val_loss'], label='Validation Loss')
plt.title('Dropout Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# train vs val accuracy
plt.subplot(1, 2, 2)
plt.plot(dropout_history.history['accuracy'], label='Train Accuracy')
plt.plot(dropout_history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Dropout  Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
```

```
plt.show()
```





[21]:
```
#Your code here
test_loss, test_accuracy = cnn_dropout.evaluate(x_test, y_test)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
313/313 [==============================] - 5s 15ms/step - loss: 0.8237 -
accuracy: 0.7254
Test Loss: 0.823741614818573
Test Accuracy: 0.7253999710083008
```

3.4.2 Batch Normalization

[18]:
```
#Your code here

cnn_bn = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    BatchNormalization(),
    Flatten(),
    Dense(64, activation='relu'),
    BatchNormalization(),
```

```
    Dense(10, activation='softmax')
])

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

cnn_bn.summary()
```

```
Model: "sequential_2"

_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d_5 (Conv2D)           (None, 30, 30, 32)        896       
                                                                 
 batch_normalization (Batch  (None, 30, 30, 32)        128       
 Normalization)                                                  
                                                                 
 max_pooling2d_4 (MaxPoolin  (None, 15, 15, 32)        0         
 g2D)                                                            
                                                                 
 conv2d_6 (Conv2D)           (None, 13, 13, 64)        18496     
                                                                 
 batch_normalization_1 (Bat  (None, 13, 13, 64)        256       
 chNormalization)                                                
                                                                 
 max_pooling2d_5 (MaxPoolin  (None, 6, 6, 64)          0         
 g2D)                                                            
                                                                 
 conv2d_7 (Conv2D)           (None, 4, 4, 64)          36928     
                                                                 
 batch_normalization_2 (Bat  (None, 4, 4, 64)          256       
 chNormalization)                                                
                                                                 
 flatten_2 (Flatten)         (None, 1024)              0         
                                                                 
 dense_5 (Dense)             (None, 64)                65600     
                                                                 
 batch_normalization_3 (Bat  (None, 64)                256       
 chNormalization)                                                
                                                                 
 dense_6 (Dense)             (None, 10)                650       
                                                                 
=================================================================
Total params: 123466 (482.29 KB)
Trainable params: 123018 (480.54 KB)
Non-trainable params: 448 (1.75 KB)
```

---

[19]:
```
#Your code here
bn_history = cnn_bn.fit(X_train, y_train, epochs=20, validation_data=(X_val,␣
↪y_val))
```

```
Epoch 1/20
1250/1250 [==============================] - 21s 14ms/step - loss: 1.3839 -
accuracy: 0.5096 - val_loss: 1.7182 - val_accuracy: 0.4506
Epoch 2/20
1250/1250 [==============================] - 12s 10ms/step - loss: 1.0174 -
accuracy: 0.6414 - val_loss: 1.1658 - val_accuracy: 0.5826
Epoch 3/20
1250/1250 [==============================] - 14s 11ms/step - loss: 0.8622 -
accuracy: 0.6986 - val_loss: 1.1748 - val_accuracy: 0.5976
Epoch 4/20
1250/1250 [==============================] - 13s 11ms/step - loss: 0.7618 -
accuracy: 0.7347 - val_loss: 0.9241 - val_accuracy: 0.6810
Epoch 5/20
1250/1250 [==============================] - 14s 11ms/step - loss: 0.6756 -
accuracy: 0.7643 - val_loss: 0.9059 - val_accuracy: 0.6902
Epoch 6/20
1250/1250 [==============================] - 12s 10ms/step - loss: 0.6098 -
accuracy: 0.7848 - val_loss: 1.2568 - val_accuracy: 0.6013
Epoch 7/20
1250/1250 [==============================] - 15s 12ms/step - loss: 0.5458 -
accuracy: 0.8081 - val_loss: 1.0013 - val_accuracy: 0.6878
Epoch 8/20
1250/1250 [==============================] - 15s 12ms/step - loss: 0.4917 -
accuracy: 0.8262 - val_loss: 0.9570 - val_accuracy: 0.7023
Epoch 9/20
1250/1250 [==============================] - 15s 12ms/step - loss: 0.4454 -
accuracy: 0.8415 - val_loss: 0.9753 - val_accuracy: 0.6976
Epoch 10/20
1250/1250 [==============================] - 14s 11ms/step - loss: 0.4020 -
accuracy: 0.8575 - val_loss: 0.9361 - val_accuracy: 0.7110
Epoch 11/20
1250/1250 [==============================] - 12s 10ms/step - loss: 0.3633 -
accuracy: 0.8718 - val_loss: 1.0653 - val_accuracy: 0.6949
Epoch 12/20
1250/1250 [==============================] - 15s 12ms/step - loss: 0.3351 -
accuracy: 0.8802 - val_loss: 1.1558 - val_accuracy: 0.6803
Epoch 13/20
1250/1250 [==============================] - 14s 11ms/step - loss: 0.3062 -
accuracy: 0.8896 - val_loss: 1.0328 - val_accuracy: 0.7127
Epoch 14/20
1250/1250 [==============================] - 14s 11ms/step - loss: 0.2772 -
accuracy: 0.9019 - val_loss: 1.2350 - val_accuracy: 0.6937
```

```
Epoch 15/20
1250/1250 [==============================] - 13s 10ms/step - loss: 0.2580 -
accuracy: 0.9081 - val_loss: 1.2298 - val_accuracy: 0.6899
Epoch 16/20
1250/1250 [==============================] - 15s 12ms/step - loss: 0.2369 -
accuracy: 0.9158 - val_loss: 1.2827 - val_accuracy: 0.6875
Epoch 17/20
1250/1250 [==============================] - 15s 12ms/step - loss: 0.2266 -
accuracy: 0.9190 - val_loss: 1.1549 - val_accuracy: 0.7093
Epoch 18/20
1250/1250 [==============================] - 13s 11ms/step - loss: 0.2141 -
accuracy: 0.9230 - val_loss: 1.4285 - val_accuracy: 0.6658
Epoch 19/20
1250/1250 [==============================] - 14s 11ms/step - loss: 0.1952 -
accuracy: 0.9302 - val_loss: 1.3459 - val_accuracy: 0.6849
Epoch 20/20
1250/1250 [==============================] - 14s 11ms/step - loss: 0.1840 -
accuracy: 0.9352 - val_loss: 1.3568 - val_accuracy: 0.6940
```

[20]:

```
#Your code here

# train vs val loss
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(bn_history.history['loss'], label='Train Loss')
plt.plot(bn_history.history['val_loss'], label='Validation Loss')
plt.title('Batch Norm Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# train vs val accuracy
plt.subplot(1, 2, 2)
plt.plot(bn_history.history['accuracy'], label='Train Accuracy')
plt.plot(bn_history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Batch Norm  Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
```

[22]:
```
test_loss, test_accuracy = cnn_bn.evaluate(x_test, y_test)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

```
313/313 [==============================] - 5s 14ms/step - loss: 1.3582 -
accuracy: 0.6933
Test Loss: 1.358229637145996
Test Accuracy: 0.6933000087738037
```

Dropout Model: - Dropout regularization seems to be the most effective in reducing overfitting, as indicated by the higher test accuracy and lower test loss compared to the original model.

Batch Normalization Model: - While batch normalization helps to some extent, it does not perform as well as dropout in mitigating overfitting, as the test accuracy is lower compared to the dropout model.

SO for me Droput performed better than Batch Normalization.

[ ]:
```
```