AML_HW4_Questions_th3061

November 17, 2023

0.0.1 Instructions

1) Please push the .ipynb and .pdf to Github Classroom prior to the deadline, .py file is optional (not needed).
2) Please include your Name and UNI below.

0.0.2 Name: Ta-Wei Huang

0.0.3 UNI: th3061

##Setup

```
[ ]: 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 the scratch

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

In this part, you will 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 ouput layer is W2 and has bias b2.

Activatation function is sigmoid for both hidden and output layer

Loss function is the 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,))
```

[ ]:
```
weights
```

[ ]:
```
{'W1': array([[ 1.76405235,  0.40015721],
        [ 0.97873798,  2.2408932 ],
        [ 1.86755799, -0.97727788]]),
 'b1': array([0., 0., 0.]),
 'W2': array([ 0.95008842, -0.15135721, -0.10321885]),
 'b2': 0}
```

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

[ ]:
```
#Implement the forward pass
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)

    Z2 = np.dot(H, weights['W2'].T) + weights['b2']
    Y = sigmoid(Z2)

    return Y, Z2, H, Z1
```

[ ]:
```
# Implement the backward pass
# 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)
    # dLdZ1 =
    dLdZ1 = np.multiply(dLdH, (sigmoid(Z1)*(1-sigmoid(Z1))))
    # dLdW1 =
    dLdW1 = np.dot(X.T, dLdZ1)
    # dLdb1 =
    dLdb1 = np.dot(ones.T, 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, -0.00030765, -0.00034768],
       [ 0.00262019, -0.00024188, -0.000372  ]]),
    'W2': array([0.02216011, 0.02433097, 0.01797174]),
    'b1': array([ 0.00492577, -0.00058023, -0.00065977]),
    'b2': 0.029249230265318685}
```

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, truck). The train/test split is 50k/10k.

[ ]:
```
from tensorflow.keras.datasets import cifar10
(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 [==============================] - 5s 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
def plot_samples(x, y, labels, num_samples=5, num_classes=10):
    fig, axes = plt.subplots(num_classes, num_samples+1, figsize=(20, 20))
    #fig.subplots_adjust(hspace=0.5, wspace=0.1)

    for i, label in enumerate(labels):
        idx = np.where(y == i)[0][:num_samples+1 ]

        for j, img_idx in enumerate(idx):
            ax = axes[i, j]
            if j == 0:
                ax.set_title(label, size=12, y=0.5)
                ax.axis('off')
            else:

                ax.imshow(x[img_idx])
                ax.axis('off')

plot_samples(x_dev, y_dev, LABELS)
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) Train-test split your development set into train and validation sets (8:2 ratio).

```
[ ]: # 1
     print("x_dev shape:", x_dev.shape) # number, height, width, color
     print("y_dev shape:", y_dev.shape)
     print("x_test shape:", x_test.shape)
     print("y_test shape:", y_test.shape)
     # 2
     x_dev_flat = x_dev.reshape(x_dev.shape[0], -1) # 50000*1
     x_test_flat = x_test.reshape(x_test.shape[0], -1)
     print("x_dev_flat shape:", x_dev_flat.shape)
     print("x_test_flat shape:", x_test_flat.shape)
     # 3
     x_dev_standardized = x_dev_flat / 255.0
     x_test_standardized = x_test_flat / 255.0
     # 4
     from sklearn.model_selection import train_test_split
     x_train, x_val, y_train, y_val = train_test_split(x_dev_standardized, y_dev,␣
      ↪test_size=0.2, random_state=42)

     print("x_train shape:", x_train.shape)
     print("x_val shape:", x_val.shape)
     print("y_train shape:", y_train.shape)
     print("y_val shape:", y_val.shape)
```

```
x_dev shape: (50000, 32, 32, 3)
y_dev shape: (50000, 1)
x_test shape: (10000, 32, 32, 3)
y_test shape: (10000, 1)
x_dev_flat shape: (50000, 3072)
x_test_flat shape: (10000, 3072)
x_train shape: (40000, 3072)
x_val shape: (10000, 3072)
y_train shape: (40000, 1)
y_val shape: (10000, 1)
```

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!

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


     model = Sequential([
         Dense(128, activation='relu', input_shape=(3072,)),  # First dense layer␣
      ↪with 128 units
         Dense(64, activation='relu'),                        # Second (hidden)␣
      ↪dense layer with 64 units
```

```
    Dense(10, activation='softmax')                    # Output layer with 10␣
 ↪units for 10 classes
])
```

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

[ ]:
```
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)
_________________________________________________________________
```

```
_________________________________________________________________
 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 NN is dependent on the image height and width. For example, in the inital layer, the number of parameters is `input_size * number_of_neuron + number_of_neuron(bias)`, as for each neuron, there is one weight per input unit.

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 equal to 1 upto two decimal

places?

```
[ ]: #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 [==============================] - 0s 24ms/step
Output: 1.00
1/1 [==============================] - 0s 24ms/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.

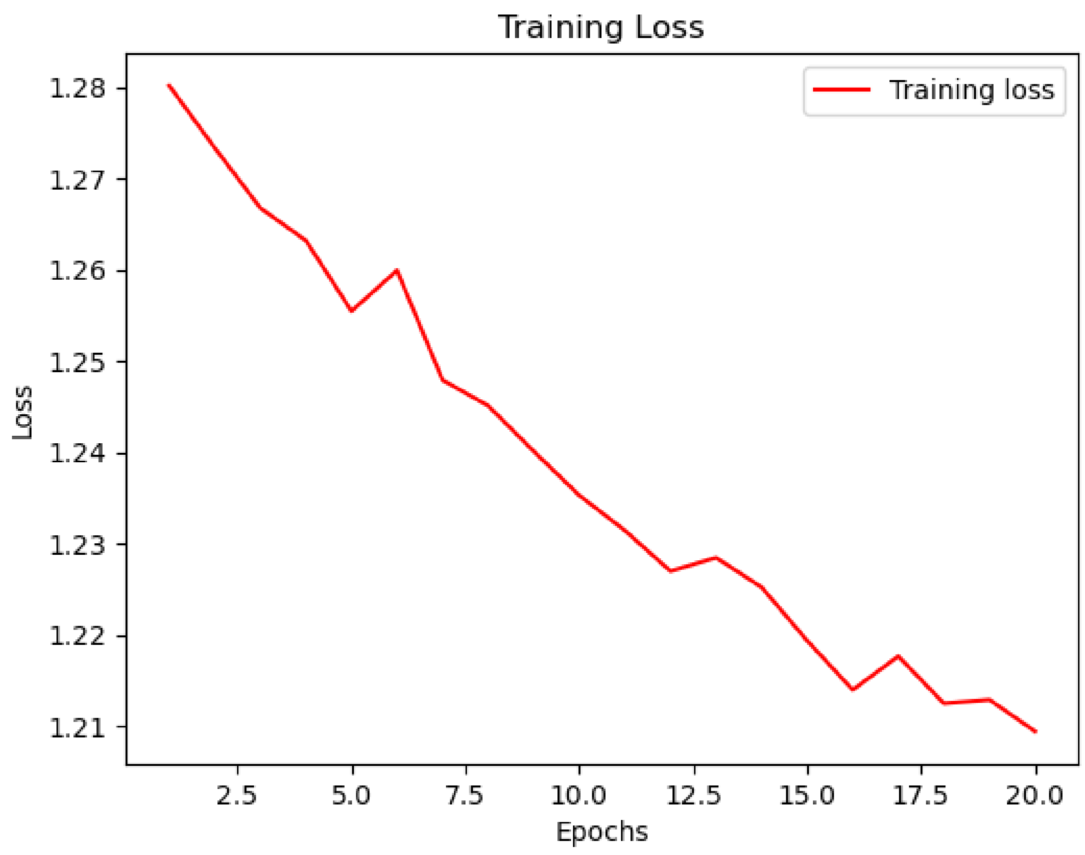
```
[ ]: model.compile(optimizer='adam',
                   loss='sparse_categorical_crossentropy',
                   metrics=['accuracy'])
     history = model.fit(x_train, y_train, epochs=20, validation_split=0.2)
```

```
Epoch 1/20
1000/1000 [==============================] - 1s 1ms/step - loss: 1.2801 -
accuracy: 0.5429 - val_loss: 1.3402 - val_accuracy: 0.5232
Epoch 2/20
1000/1000 [==============================] - 1s 895us/step - loss: 1.2733 -
accuracy: 0.5417 - val_loss: 1.3265 - val_accuracy: 0.5321
Epoch 3/20
1000/1000 [==============================] - 1s 935us/step - loss: 1.2667 -
accuracy: 0.5471 - val_loss: 1.3578 - val_accuracy: 0.5217
Epoch 4/20
1000/1000 [==============================] - 1s 894us/step - loss: 1.2632 -
accuracy: 0.5493 - val_loss: 1.3722 - val_accuracy: 0.5176
Epoch 5/20
1000/1000 [==============================] - 1s 894us/step - loss: 1.2554 -
accuracy: 0.5513 - val_loss: 1.4578 - val_accuracy: 0.4846
Epoch 6/20
1000/1000 [==============================] - 1s 923us/step - loss: 1.2599 -
accuracy: 0.5469 - val_loss: 1.3771 - val_accuracy: 0.5141
Epoch 7/20
1000/1000 [==============================] - 1s 955us/step - loss: 1.2479 -
accuracy: 0.5532 - val_loss: 1.3708 - val_accuracy: 0.5214
Epoch 8/20
1000/1000 [==============================] - 1s 899us/step - loss: 1.2451 -
accuracy: 0.5542 - val_loss: 1.3787 - val_accuracy: 0.5131
Epoch 9/20
1000/1000 [==============================] - 1s 917us/step - loss: 1.2401 -
accuracy: 0.5536 - val_loss: 1.3948 - val_accuracy: 0.5176
```

```
Epoch 10/20
1000/1000 [==============================] - 1s 906us/step - loss: 1.2353 -
accuracy: 0.5562 - val_loss: 1.4000 - val_accuracy: 0.5121
Epoch 11/20
1000/1000 [==============================] - 1s 952us/step - loss: 1.2314 -
accuracy: 0.5574 - val_loss: 1.4313 - val_accuracy: 0.4985
Epoch 12/20
1000/1000 [==============================] - 1s 922us/step - loss: 1.2270 -
accuracy: 0.5610 - val_loss: 1.3922 - val_accuracy: 0.5113
Epoch 13/20
1000/1000 [==============================] - 1s 899us/step - loss: 1.2284 -
accuracy: 0.5595 - val_loss: 1.4279 - val_accuracy: 0.5024
Epoch 14/20
1000/1000 [==============================] - 1s 950us/step - loss: 1.2252 -
accuracy: 0.5578 - val_loss: 1.4062 - val_accuracy: 0.5107
Epoch 15/20
1000/1000 [==============================] - 1s 971us/step - loss: 1.2193 -
accuracy: 0.5623 - val_loss: 1.4261 - val_accuracy: 0.5023
Epoch 16/20
1000/1000 [==============================] - 1s 906us/step - loss: 1.2140 -
accuracy: 0.5652 - val_loss: 1.4303 - val_accuracy: 0.5107
Epoch 17/20
1000/1000 [==============================] - 1s 978us/step - loss: 1.2176 -
accuracy: 0.5628 - val_loss: 1.4091 - val_accuracy: 0.5109
Epoch 18/20
1000/1000 [==============================] - 1s 905us/step - loss: 1.2125 -
accuracy: 0.5661 - val_loss: 1.4694 - val_accuracy: 0.4935
Epoch 19/20
1000/1000 [==============================] - 1s 903us/step - loss: 1.2129 -
accuracy: 0.5643 - val_loss: 1.4343 - val_accuracy: 0.5044
Epoch 20/20
1000/1000 [==============================] - 1s 1ms/step - loss: 1.2094 -
accuracy: 0.5676 - val_loss: 1.4229 - val_accuracy: 0.5092
```

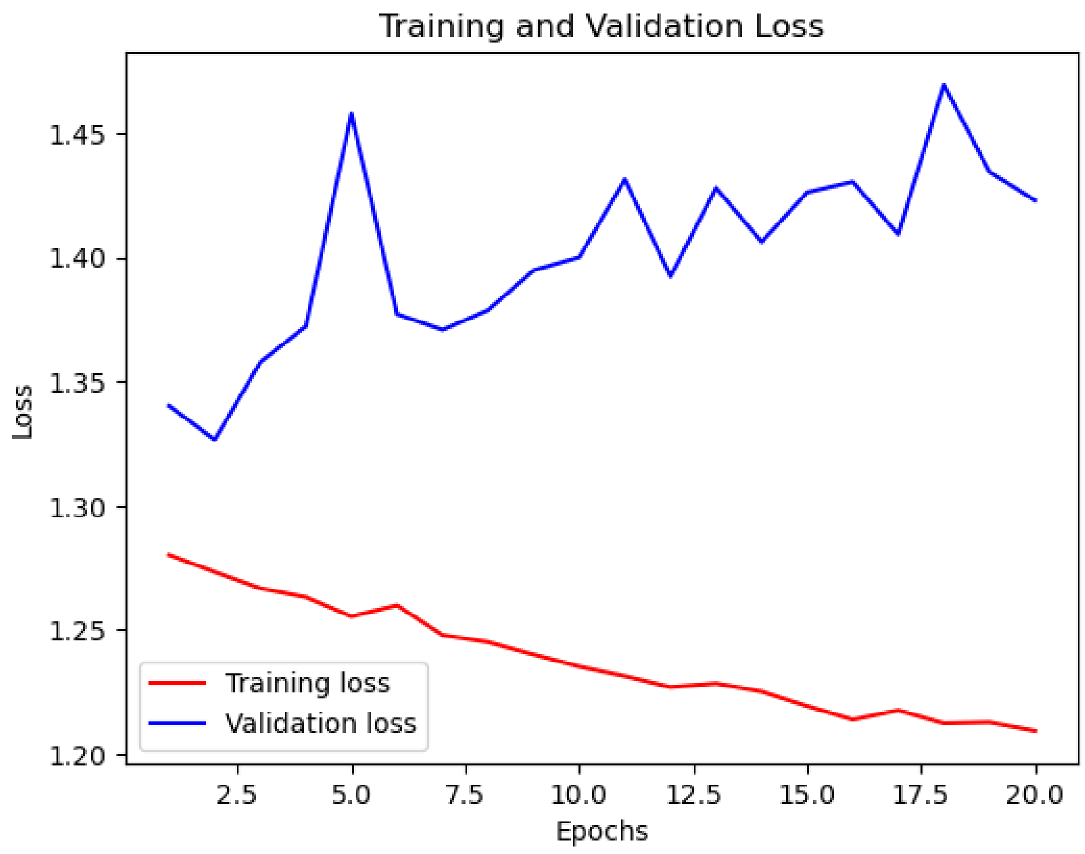
2.0.7 2.7 Plot the training curves described below

[ ]:
```
import matplotlib.pyplot as plt

train_loss = history.history['loss']
epochs = range(1, len(train_loss) + 1)

plt.figure()
plt.plot(epochs, train_loss, 'r', label='Training loss')
plt.title('Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



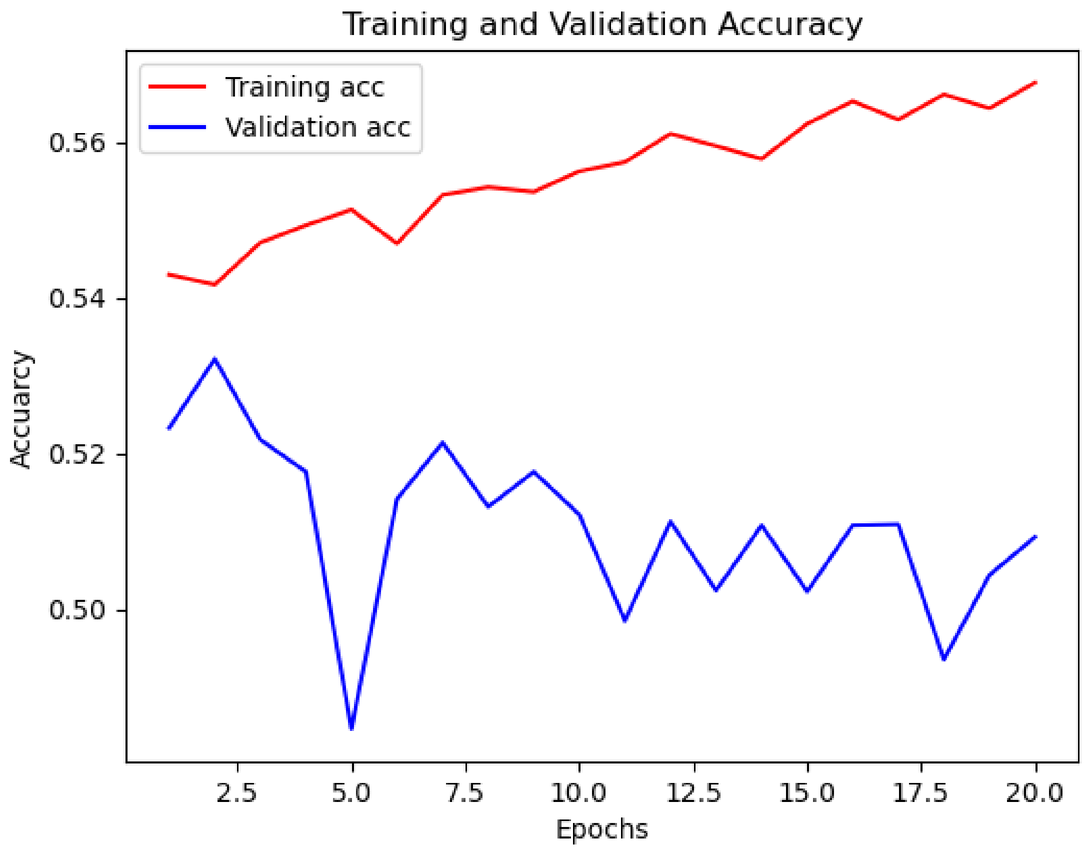
2.0.8 2.7.1 Display the train vs validation loss over each epoch

[ ]:
```
import matplotlib.pyplot as plt

train_loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(train_loss) + 1)

plt.figure()
plt.plot(epochs, train_loss, 'r', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



2.0.9 2.7.2 Display the train vs validation accuracy over each epoch

[ ]:
```python
train_acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
epochs = range(1, len(train_acc) + 1)

plt.figure()
plt.plot(epochs, train_acc, 'r', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuarcy')
plt.legend()
plt.show()
```



2.0.10 2.8 Finally, report the metric chosen on test set

```
[ ]: test_loss, test_acc = model.evaluate(x_test_standardized, y_test, verbose=2)
     print('Test accuracy:', test_acc)
```

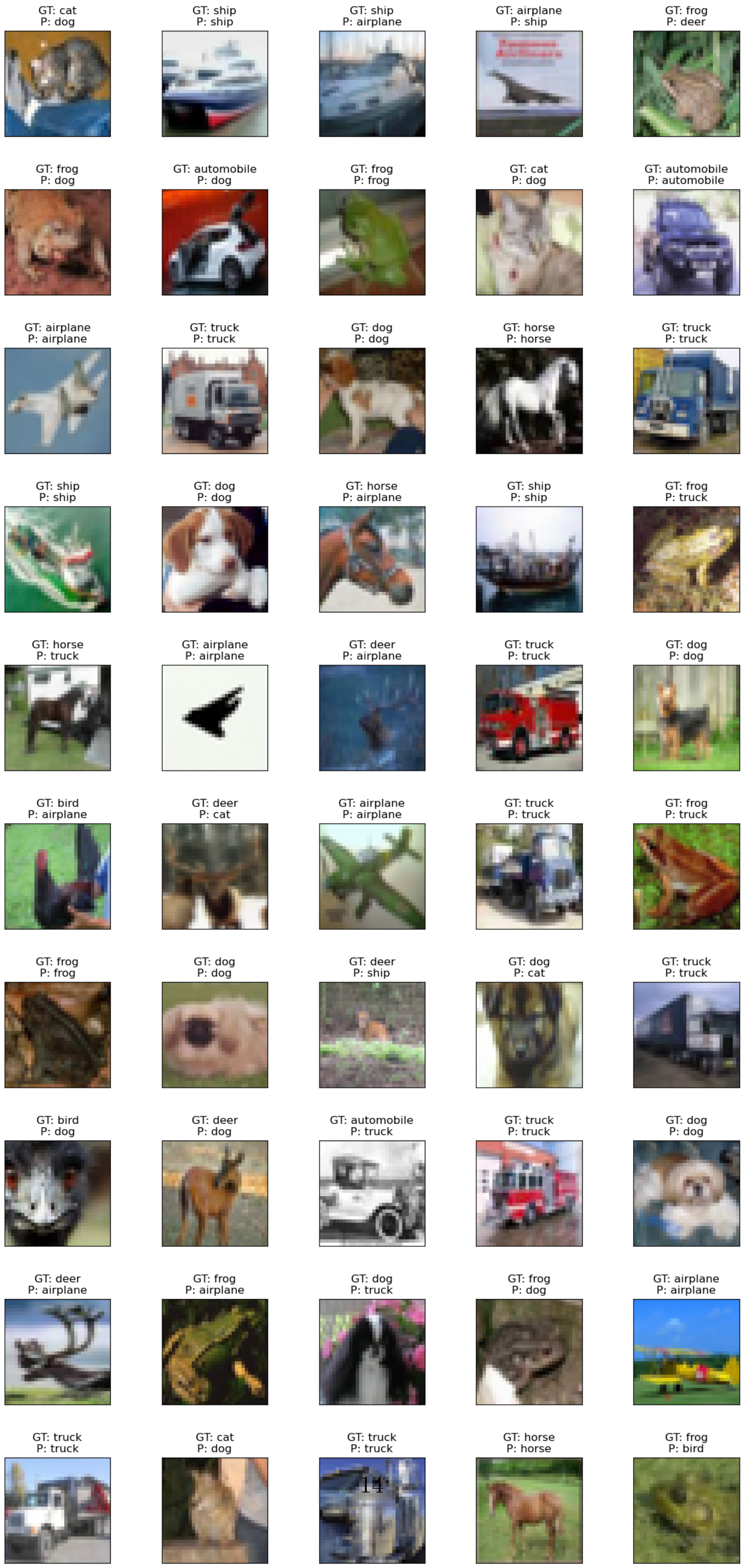
```
313/313 - 0s - loss: 1.4972 - accuracy: 0.4846 - 172ms/epoch - 548us/step
Test accuracy: 0.4846000075340271
Test accuracy: 0.4846000075340271
```

2.0.11 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)

```
[ ]: predictions = model.predict(x_test_flat[:50])
     #print(predictions)
     predicted_classes = np.argmax(predictions, axis=1)
     #print(predicted_classes)
     fig, axes = plt.subplots(10, 5, figsize=(15, 30))
     fig.subplots_adjust(hspace=0.5, wspace=0.1)
     for i, ax in enumerate(axes.flat):
```

```
    # Plot image
    ax.imshow(x_test[i])

    # Add label with ground truth and prediction
    true_label = LABELS[y_test[i][0]]
    predicted_label = LABELS[predicted_classes[i]]
    ax.set_title(f"GT: {true_label}\nP: {predicted_label}")

    # Remove axis ticks
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
```

```
2/2 [==============================] - 0s 1ms/step
```


GT: cat P: dog
GT: ship P: ship
GT: ship P: airplane
GT: airplane P: ship
GT: frog P: deer
GT: frog P: dog
GT: automobile P: dog
GT: frog P: frog
GT: cat P: dog
GT: automobile P: automobile
GT: airplane P: airplane
GT: truck P: truck
GT: dog P: dog
GT: horse P: horse
GT: truck P: truck
GT: ship P: ship
GT: dog P: dog
GT: horse P: airplane
GT: ship P: ship
GT: frog P: truck
GT: horse P: truck
GT: airplane P: airplane
GT: deer P: airplane
GT: truck P: truck
GT: dog P: dog
GT: bird P: airplane
GT: deer P: cat
GT: airplane P: airplane
GT: truck P: truck
GT: frog P: truck
GT: frog P: frog
GT: dog P: dog
GT: deer P: ship
GT: dog P: cat
GT: truck P: truck
GT: bird P: dog
GT: deer P: dog
GT: automobile P: truck
GT: truck P: truck
GT: dog P: dog
GT: deer P: airplane
GT: frog P: airplane
GT: dog P: truck
GT: frog P: dog
GT: airplane P: airplane
GT: truck P: truck
GT: cat P: dog
GT: truck P: truck
GT: horse P: horse
GT: frog P: bird

[ ]:

```
```