UNIVERSITY OF TEHRAN

COLLEGE OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

INTELLIGENT SYSTEM

ASSIGNMENT#3

MOHAMMAD HEYDARI

810197494

UNDER SUPERVISION OF:

DR. RESHAD HOSSINI

ASSISTANT PROFESSOR

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

UNIVERSITY OF TEHRAN

Nov. 2021

1 CONTENTS

2 QUESTION #1

2.1 CONVOLUTIONAL NEURAL NETWORK IN CLASSIFICATION(CNN)

In this part we intend to design CNN using keras library, we have used Cifar10 dataset to analyse data.

This dataset includes 60000 pictures which classified in 10 classes.

In first part of this analyse we are supposed to report the error and accuracy in each epoch and also plot cost function of evaluation and test data based on epoch in time domain.

Note that the minimum value for epoch must be around 10 cycles.

Also Note that I make several functions for each part of investigation to have dipper insight on functionality of each architecture.

First of all, we load Cifar10 dataset and consider 20 percent of data for evaluation and also 80 percent of that for training model.

```
import sys
from matplotlib import pyplot
from keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers import Dropout


# load train and test dataset


def load_dataset():
  # load dataset
  (trainX, trainY), (testX, testY) = cifar10.load_data()
  # one hot encode target values
  trainY = to_categorical(trainY)
  testY = to_categorical(testY)
  return trainX, trainY, testX, testY
```

```
trainX, trainy, testX, testy = load_dataset()
train_dataX = trainX[0 : round(trainX.shape[0]  * 0.8)]
train_dataY = trainy[0 : round(trainX.shape[0]  * 0.8)]
eval_dataX = trainX[round(trainX.shape[0] * 0.8):-1]
eval_dataY = trainy[round(trainX.shape[0] * 0.8):-1]
```

In the next step we want to show a picture from each classes to get more familiar with this dataset:

```
kinds = []
indexes = []
i = 0
while  len(kinds) < 10:
    current_cat = np.where(train_dataY[i] == 1)[0][0]
    if kinds.count(current_cat) == 0:
        kinds.append(current_cat)
        indexes.append(i)
    i = i+1

for i in range(len(indexes)):
    pyplot.imshow(train_dataX[indexes[i]])
    pyplot.title(kinds[i])
    pyplot.show()
```

And as result of above code we have:

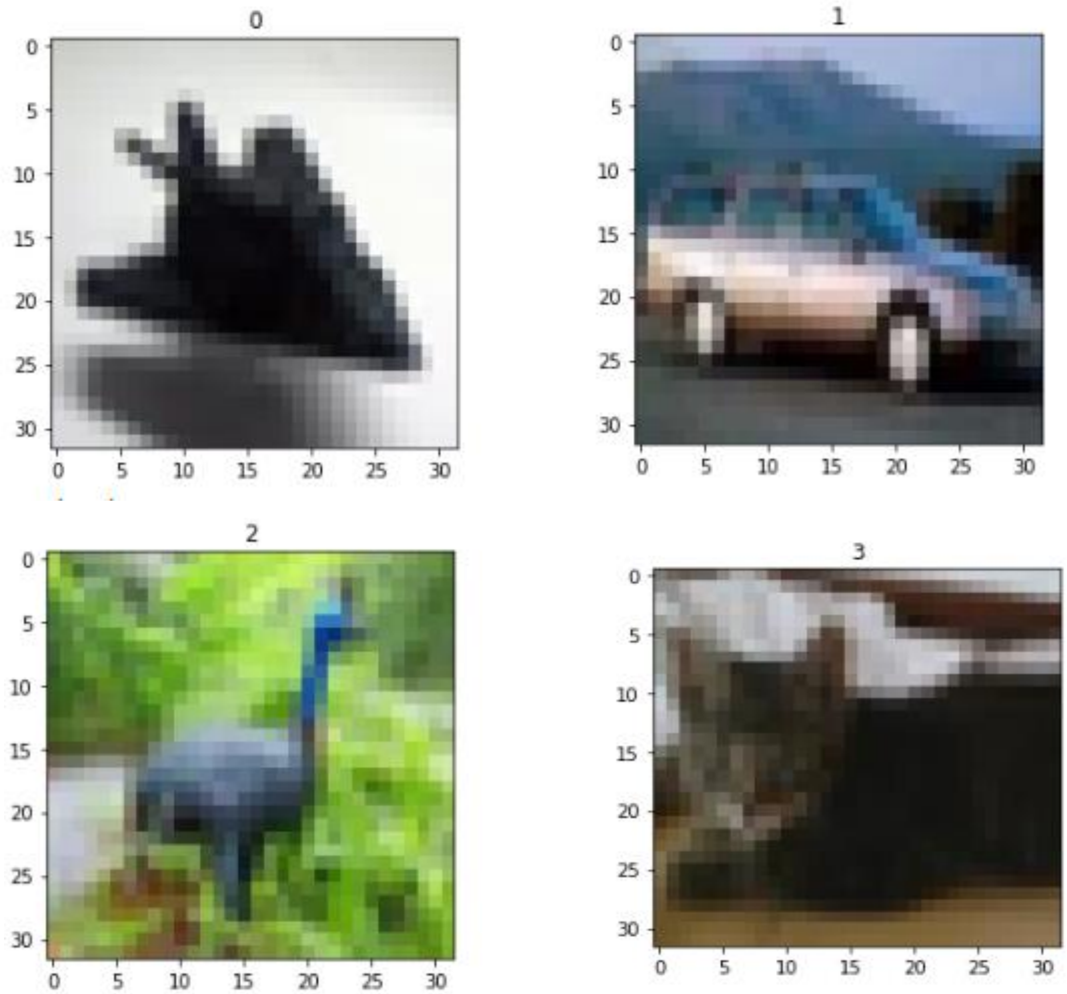


Figure 1: Representation of Dataset

Figure 2: Representation of Dataset

Then, we are going to use appropriate prepressing methods on our data and describe why those type of methods are used.

1.

you can see further I have written a function called prep-pixels which normalize RGB pictures and change those data in interval of 0 to 1.

Note that with normalize data with dividing by 255 which is the maximum of each RGB channel.

```
def prep_pixels(train, test):
  # convert from integers to floats
  train_norm = train.astype('float32')
  test_norm = test.astype('float32')
  # normalize to range 0-1
  train_norm = train_norm / 255.0
  test_norm = test_norm / 255.0
  # return normalized images
  return train_norm, test_norm
```

Furthermore, we also function to

2.

we have used reshape() as of the second pre-processing tool as you can see below:

```
# return normalized images
train_norm = np.reshape(train_norm,(len(train),32,32,3))
validation_norm = np.reshape(validation_norm,(len(validation),32,32,3))
test_norm = np.reshape(test_norm,(len(test),32,32,3))
```

3.

And finally using to_categorical() function which implement one hot encode target values considered as of the third pre-processing which we do on data.

```
# one hot encode target values
trainY = to_categorical(trainY)
testY = to_categorical(testY)
return trainX, trainY, testX, testY
```

Afterward, we are going to design model according to HW description.

We have used **softmax** as of activation function in the final layer because in this problem we face with a classification task and also we have coded each output to one-hot form so in these cases softmax helps us to normalize output of final layer and also showing probability of classification to a specific group in one-hot implementation can give us a better insight.

Note that in this implementation the batch size value is 64 and also I have used 20 epochs in my analyses.

2.2 Increasing Hidden Layer

In this part we intend to analyse the effect of increasing number of hidden layer on accuracy.

So in this part I have written different function with different number of hidden layer and analyse accuracy of each of them.

You can see my implementation below:

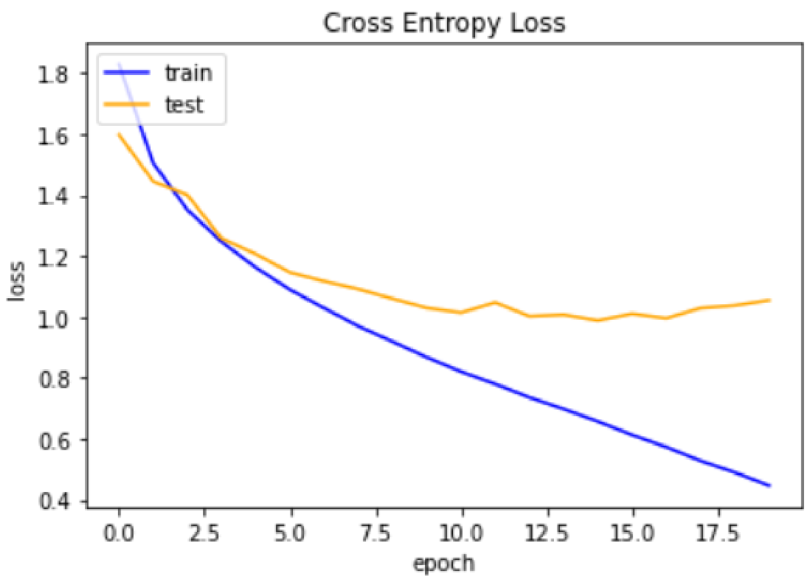
```
def define_model_baseline1():
  model = Sequential()
  model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=(32, 32, 3)))
  model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Flatten())
  model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
  model.add(Dense(10, activation='softmax'))
  # compile model
  opt = SGD(learning_rate=0.001, momentum=0.9)
  model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
  return model

def define_model_baseline2():
  model = Sequential()
  model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=(32, 32, 3)))
  model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
  model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Flatten())
  model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
  model.add(Dense(10, activation='softmax'))
  # compile model
  opt = SGD(lr=0.001, momentum=0.9)
  model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
  return model
```

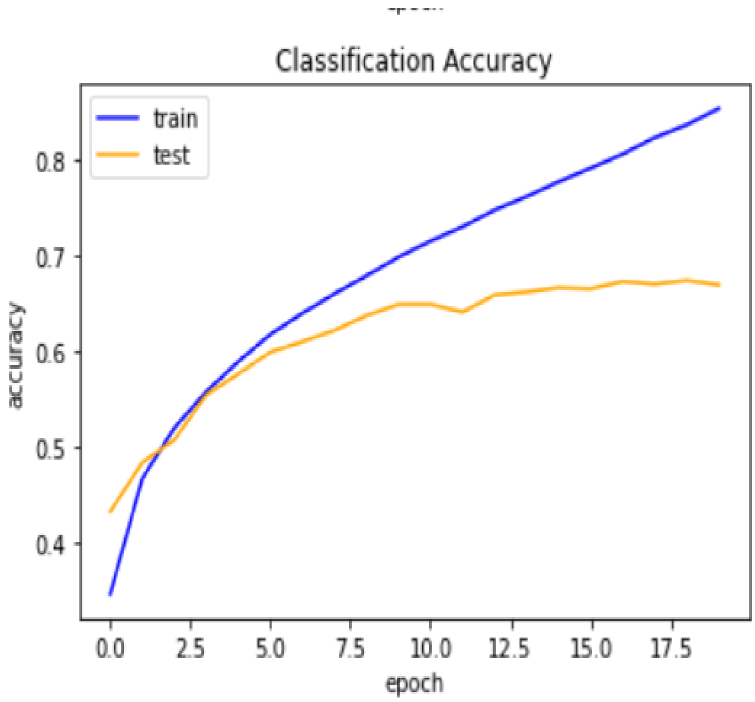
```
def define_model_baseline3():
  model = Sequential()
  model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=(32, 32, 3)))
  model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
  model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
  model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Flatten())
  model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
  model.add(Dense(10, activation='softmax'))
  # compile model
  opt = SGD(lr=0.001, momentum=0.9)
  model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
  return model
```

<u>In this part I have tested three state of adding hidden layer and found that the best Accuracy and F1score is for CNN with 2 hidden layers so we choose 2 for number of hidden layers.</u>

Further you can see the result of 1 hidden layer CNN for 20 epochs!

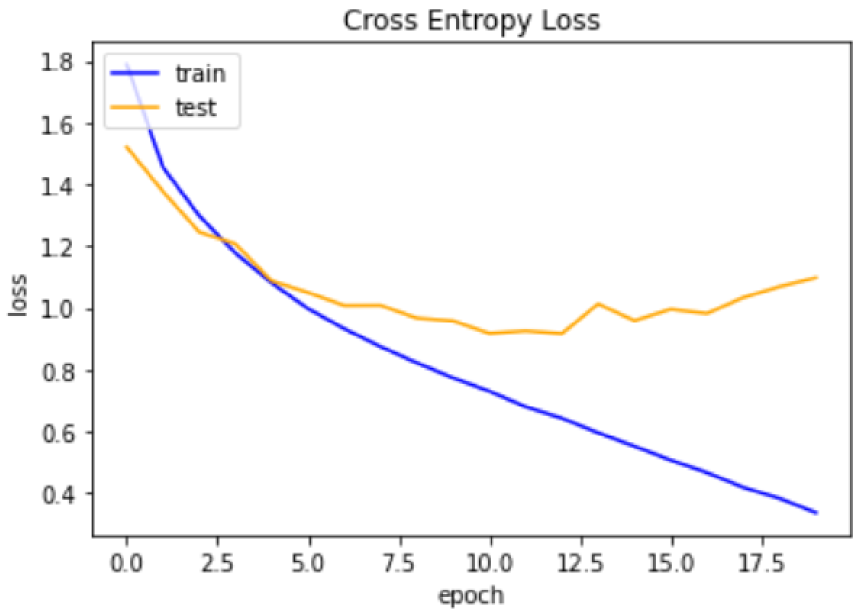
```
Epoch 1/20
625/625 [==============================] - 10s 16ms/step - loss: 1.8284 - accuracy: 0.3467 - val_loss: 1.5973 - val_accuracy: 0.4325
Epoch 2/20
625/625 [==============================] - 10s 15ms/step - loss: 1.5044 - accuracy: 0.4670 - val_loss: 1.4443 - val_accuracy: 0.4837
Epoch 3/20
625/625 [==============================] - 10s 15ms/step - loss: 1.3511 - accuracy: 0.5199 - val_loss: 1.3998 - val_accuracy: 0.5075
Epoch 4/20
625/625 [==============================] - 10s 15ms/step - loss: 1.2459 - accuracy: 0.5578 - val_loss: 1.2571 - val_accuracy: 0.5545
Epoch 5/20
625/625 [==============================] - 10s 16ms/step - loss: 1.1616 - accuracy: 0.5892 - val_loss: 1.2059 - val_accuracy: 0.5764
Epoch 6/20
625/625 [==============================] - 9s 15ms/step - loss: 1.0890 - accuracy: 0.6176 - val_loss: 1.1455 - val_accuracy: 0.5991
Epoch 7/20
625/625 [==============================] - 9s 15ms/step - loss: 1.0283 - accuracy: 0.6395 - val_loss: 1.1166 - val_accuracy: 0.6099
Epoch 8/20
625/625 [==============================] - 9s 15ms/step - loss: 0.9689 - accuracy: 0.6599 - val_loss: 1.0908 - val_accuracy: 0.6218
Epoch 9/20
625/625 [==============================] - 9s 15ms/step - loss: 0.9184 - accuracy: 0.6787 - val_loss: 1.0592 - val_accuracy: 0.6371
Epoch 10/20
625/625 [==============================] - 9s 14ms/step - loss: 0.8669 - accuracy: 0.6982 - val_loss: 1.0292 - val_accuracy: 0.6489
Epoch 11/20
625/625 [==============================] - 9s 14ms/step - loss: 0.8199 - accuracy: 0.7150 - val_loss: 1.0136 - val_accuracy: 0.6490
Epoch 12/20
625/625 [==============================] - 10s 15ms/step - loss: 0.7794 - accuracy: 0.7299 - val_loss: 1.0471 - val_accuracy: 0.6408
Epoch 13/20
625/625 [==============================] - 9s 14ms/step - loss: 0.7354 - accuracy: 0.7473 - val_loss: 1.0017 - val_accuracy: 0.6587
Epoch 14/20
625/625 [==============================] - 8s 13ms/step - loss: 0.6975 - accuracy: 0.7615 - val_loss: 1.0065 - val_accuracy: 0.6618
Epoch 15/20
625/625 [==============================] - 9s 14ms/step - loss: 0.6566 - accuracy: 0.7769 - val_loss: 0.9877 - val_accuracy: 0.6663
Epoch 16/20
625/625 [==============================] - 9s 14ms/step - loss: 0.6125 - accuracy: 0.7911 - val_loss: 1.0097 - val_accuracy: 0.6651
Epoch 17/20
625/625 [==============================] - 8s 13ms/step - loss: 0.5725 - accuracy: 0.8058 - val_loss: 0.9949 - val_accuracy: 0.6728
Epoch 18/20
625/625 [==============================] - 8s 13ms/step - loss: 0.5276 - accuracy: 0.8230 - val_loss: 1.0287 - val_accuracy: 0.6701
Epoch 19/20
625/625 [==============================] - 9s 14ms/step - loss: 0.4896 - accuracy: 0.8363 - val_loss: 1.0371 - val_accuracy: 0.6740
Epoch 20/20
625/625 [==============================] - 8s 13ms/step - loss: 0.4461 - accuracy: 0.8533 - val_loss: 1.0541 - val_accuracy: 0.6693
313/313 [==============================] - 2s 6ms/step - loss: 238.5952 - accuracy: 0.5945
> 59.450
```

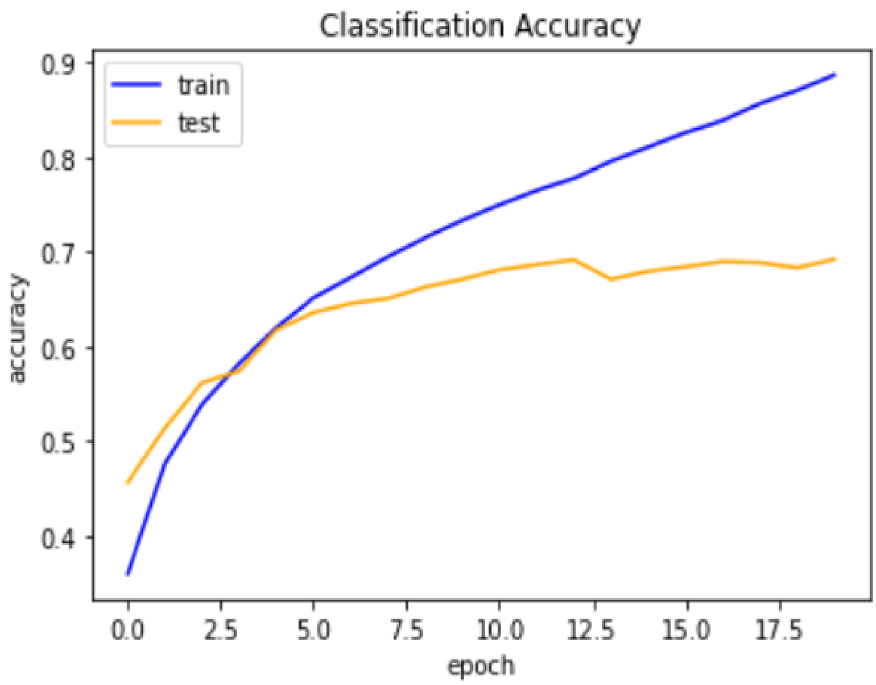




Further you can see the result of 2 hidden layer CNN for 20 epochs!

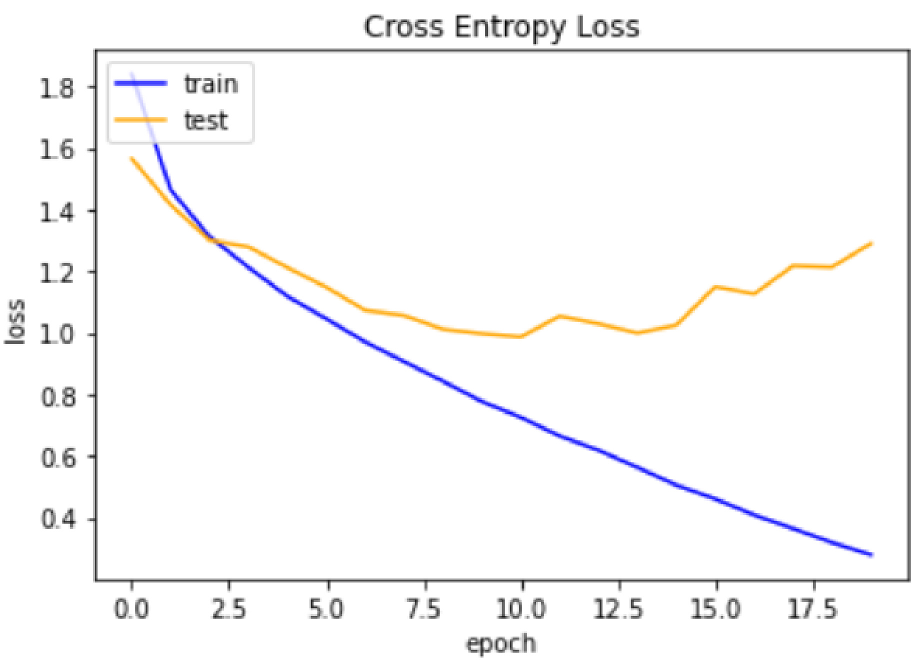
```
Epoch 1/20
625/625 [==============================] - 14s 20ms/step - loss: 1.7880 - accuracy: 0.3601 - val_loss: 1.5207 - val_accuracy: 0.4565
Epoch 2/20
625/625 [==============================] - 11s 17ms/step - loss: 1.4547 - accuracy: 0.4762 - val_loss: 1.3763 - val_accuracy: 0.5140
Epoch 3/20
625/625 [==============================] - 11s 17ms/step - loss: 1.2981 - accuracy: 0.5388 - val_loss: 1.2453 - val_accuracy: 0.5617
Epoch 4/20
625/625 [==============================] - 11s 17ms/step - loss: 1.1765 - accuracy: 0.5823 - val_loss: 1.2066 - val_accuracy: 0.5742
Epoch 5/20
625/625 [==============================] - 11s 17ms/step - loss: 1.0799 - accuracy: 0.6199 - val_loss: 1.0866 - val_accuracy: 0.6177
Epoch 6/20
625/625 [==============================] - 10s 17ms/step - loss: 0.9971 - accuracy: 0.6514 - val_loss: 1.0493 - val_accuracy: 0.6359
Epoch 7/20
625/625 [==============================] - 11s 17ms/step - loss: 0.9320 - accuracy: 0.6730 - val_loss: 1.0072 - val_accuracy: 0.6455
Epoch 8/20
625/625 [==============================] - 11s 17ms/step - loss: 0.8740 - accuracy: 0.6947 - val_loss: 1.0077 - val_accuracy: 0.6507
Epoch 9/20
625/625 [==============================] - 11s 17ms/step - loss: 0.8215 - accuracy: 0.7149 - val_loss: 0.9665 - val_accuracy: 0.6628
Epoch 10/20
625/625 [==============================] - 10s 17ms/step - loss: 0.7728 - accuracy: 0.7333 - val_loss: 0.9575 - val_accuracy: 0.6710
Epoch 11/20
625/625 [==============================] - 11s 17ms/step - loss: 0.7294 - accuracy: 0.7495 - val_loss: 0.9170 - val_accuracy: 0.6809
Epoch 12/20
625/625 [==============================] - 11s 17ms/step - loss: 0.6792 - accuracy: 0.7648 - val_loss: 0.9252 - val_accuracy: 0.6865
Epoch 13/20
625/625 [==============================] - 11s 17ms/step - loss: 0.6421 - accuracy: 0.7775 - val_loss: 0.9163 - val_accuracy: 0.6914
Epoch 14/20
625/625 [==============================] - 11s 17ms/step - loss: 0.5950 - accuracy: 0.7956 - val_loss: 1.0125 - val_accuracy: 0.6709
Epoch 15/20
625/625 [==============================] - 11s 17ms/step - loss: 0.5513 - accuracy: 0.8107 - val_loss: 0.9582 - val_accuracy: 0.6795
Epoch 16/20
625/625 [==============================] - 11s 17ms/step - loss: 0.5063 - accuracy: 0.8257 - val_loss: 0.9953 - val_accuracy: 0.6843
Epoch 17/20
625/625 [==============================] - 11s 18ms/step - loss: 0.4660 - accuracy: 0.8386 - val_loss: 0.9815 - val_accuracy: 0.6897
Epoch 18/20
625/625 [==============================] - 11s 17ms/step - loss: 0.4181 - accuracy: 0.8565 - val_loss: 1.0331 - val_accuracy: 0.6886
Epoch 19/20
625/625 [==============================] - 11s 17ms/step - loss: 0.3825 - accuracy: 0.8704 - val_loss: 1.0682 - val_accuracy: 0.6832
Epoch 20/20
625/625 [==============================] - 11s 17ms/step - loss: 0.3363 - accuracy: 0.8866 - val_loss: 1.0971 - val_accuracy: 0.6921
313/313 [==============================] - 2s 7ms/step - loss: 275.7925 - accuracy: 0.6066
> 60.660
```

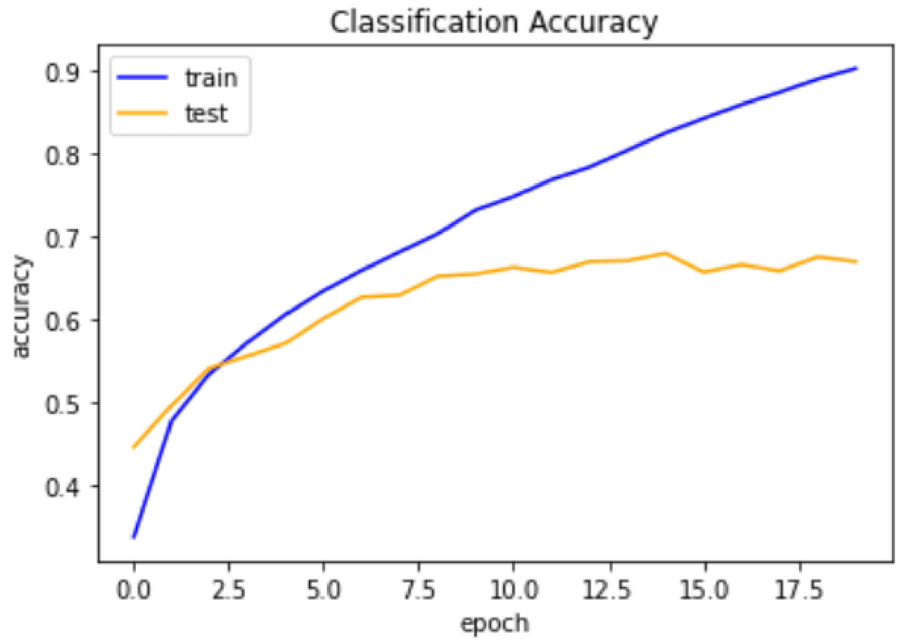




Further you can see the result of 3 hidden layer CNN for 20 epochs!

```
Epoch 1/20
625/625 [==============================] - 16s 23ms/step - loss: 1.8393 - accuracy: 0.3383 - val_loss: 1.5652 - val_accuracy: 0.4465
Epoch 2/20
625/625 [==============================] - 14s 22ms/step - loss: 1.4637 - accuracy: 0.4780 - val_loss: 1.4151 - val_accuracy: 0.4964
Epoch 3/20
625/625 [==============================] - 14s 22ms/step - loss: 1.3120 - accuracy: 0.5344 - val_loss: 1.3005 - val_accuracy: 0.5414
Epoch 4/20
625/625 [==============================] - 14s 22ms/step - loss: 1.2122 - accuracy: 0.5720 - val_loss: 1.2787 - val_accuracy: 0.5559
Epoch 5/20
625/625 [==============================] - 14s 22ms/step - loss: 1.1182 - accuracy: 0.6057 - val_loss: 1.2116 - val_accuracy: 0.5711
Epoch 6/20
625/625 [==============================] - 14s 22ms/step - loss: 1.0453 - accuracy: 0.6340 - val_loss: 1.1477 - val_accuracy: 0.6009
Epoch 7/20
625/625 [==============================] - 14s 22ms/step - loss: 0.9701 - accuracy: 0.6586 - val_loss: 1.0728 - val_accuracy: 0.6264
Epoch 8/20
625/625 [==============================] - 14s 22ms/step - loss: 0.9062 - accuracy: 0.6806 - val_loss: 1.0558 - val_accuracy: 0.6289
Epoch 9/20
625/625 [==============================] - 13s 21ms/step - loss: 0.8426 - accuracy: 0.7024 - val_loss: 1.0114 - val_accuracy: 0.6516
Epoch 10/20
625/625 [==============================] - 13s 22ms/step - loss: 0.7767 - accuracy: 0.7309 - val_loss: 0.9970 - val_accuracy: 0.6543
Epoch 11/20
625/625 [==============================] - 14s 22ms/step - loss: 0.7255 - accuracy: 0.7473 - val_loss: 0.9862 - val_accuracy: 0.6622
625/625 [==============================] - 14s 22ms/step - loss: 0.7255 - accuracy: 0.7473 - val_loss: 0.9862 - val_accuracy: 0.6622
Epoch 12/20
625/625 [==============================] - 14s 22ms/step - loss: 0.6647 - accuracy: 0.7675 - val_loss: 1.0541 - val_accuracy: 0.6561
Epoch 13/20
625/625 [==============================] - 14s 22ms/step - loss: 0.6184 - accuracy: 0.7826 - val_loss: 1.0291 - val_accuracy: 0.6691
Epoch 14/20
625/625 [==============================] - 14s 22ms/step - loss: 0.5627 - accuracy: 0.8030 - val_loss: 0.9991 - val_accuracy: 0.6701
Epoch 15/20
625/625 [==============================] - 14s 22ms/step - loss: 0.5053 - accuracy: 0.8237 - val_loss: 1.0249 - val_accuracy: 0.6790
Epoch 16/20
625/625 [==============================] - 14s 22ms/step - loss: 0.4603 - accuracy: 0.8411 - val_loss: 1.1496 - val_accuracy: 0.6565
Epoch 17/20
625/625 [==============================] - 13s 22ms/step - loss: 0.4084 - accuracy: 0.8578 - val_loss: 1.1257 - val_accuracy: 0.6655
Epoch 18/20
625/625 [==============================] - 13s 22ms/step - loss: 0.3643 - accuracy: 0.8724 - val_loss: 1.2173 - val_accuracy: 0.6580
Epoch 19/20
625/625 [==============================] - 14s 22ms/step - loss: 0.3188 - accuracy: 0.8884 - val_loss: 1.2131 - val_accuracy: 0.6748
Epoch 20/20
625/625 [==============================] - 14s 22ms/step - loss: 0.2791 - accuracy: 0.9013 - val_loss: 1.2896 - val_accuracy: 0.6691
313/313 [==============================] - 3s 8ms/step - loss: 327.4387 - accuracy: 0.5875
> 58.750
```




Classification Accuracy
train
test
accuracy
epoch
0.9
0.8
0.7
0.6
0.5
0.4
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

2.3 RELU ACTIVATION FUNCTION VS TANH ACTIVATION FUNCTION

In this part we intend to analyse the effect of replacing Tanh instead of ReLU activation function on accuracy.

You can see my implementation below:

```
def define_model_baseline1_tanh():
  model = Sequential()
  model.add(Conv2D(32, (3, 3), activation='tanh', kernel_initializer='he_uniform', padding='same', input_shape=(32, 32, 3)))
  model.add(Conv2D(32, (3, 3), activation='tanh', kernel_initializer='he_uniform', padding='same'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Flatten())
  model.add(Dense(128, activation='tanh', kernel_initializer='he_uniform'))
  model.add(Dense(10, activation='softmax'))
  # compile model
  opt = SGD(learning_rate=0.001, momentum=0.9)
  model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
  return model
```

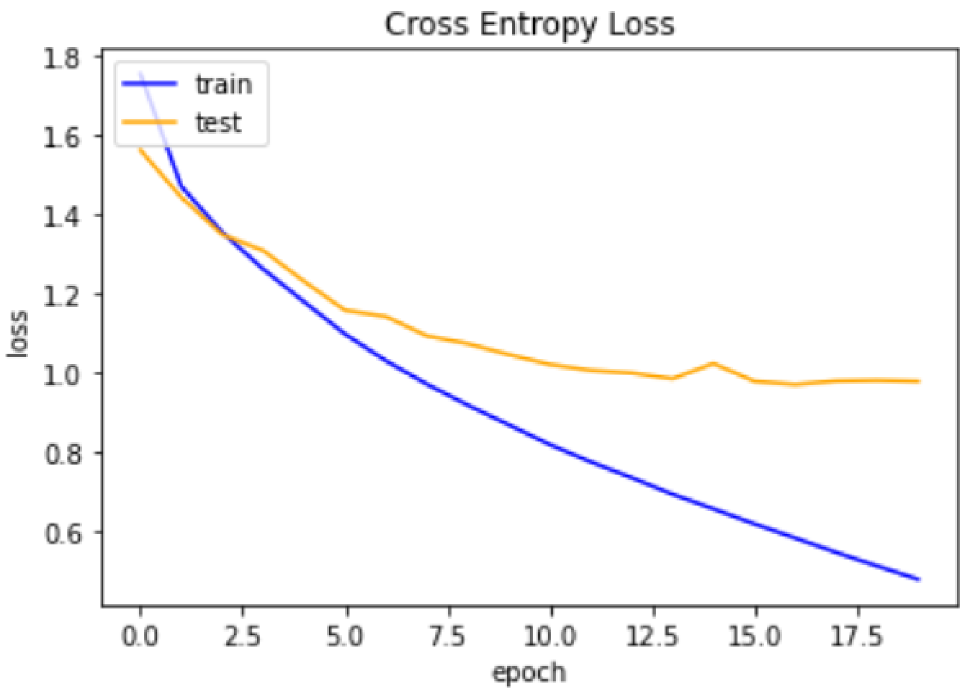
<u>Results have been showed that the ReLU activation function acts pretty much better in comparison with tanh activation function!</u>

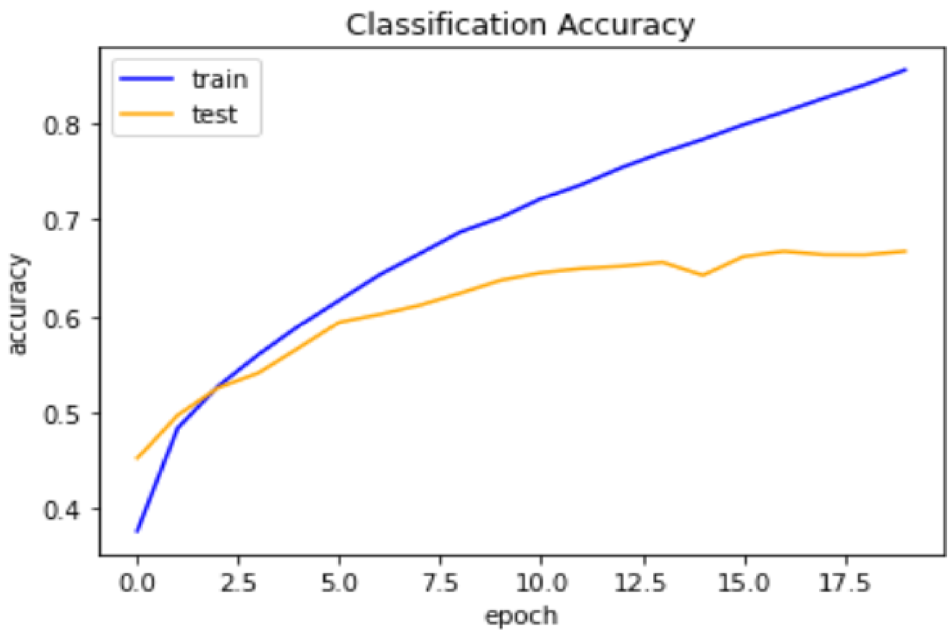
Further you can see the result of 1 hidden layer CNN for 20 epochs with Tanh!

Note that the ReLU baseline has been showed in previous part so I have ignored to show the results of that in this part!

As you can see further the accuracy for tanh is 37% which is very low in comparison to ReLU!

```
Epoch 1/20
625/625 [==============================] - 10s 16ms/step - loss: 1.7518 - accuracy: 0.3765 - val_loss: 1.5598 - val_accuracy: 0.4520
Epoch 2/20
625/625 [==============================] - 9s 14ms/step - loss: 1.4683 - accuracy: 0.4836 - val_loss: 1.4407 - val_accuracy: 0.4969
Epoch 3/20
625/625 [==============================] - 9s 14ms/step - loss: 1.3525 - accuracy: 0.5269 - val_loss: 1.3466 - val_accuracy: 0.5254
Epoch 4/20
625/625 [==============================] - 8s 14ms/step - loss: 1.2598 - accuracy: 0.5598 - val_loss: 1.3067 - val_accuracy: 0.5408
Epoch 5/20
625/625 [==============================] - 8s 13ms/step - loss: 1.1773 - accuracy: 0.5897 - val_loss: 1.2289 - val_accuracy: 0.5669
Epoch 6/20
625/625 [==============================] - 9s 14ms/step - loss: 1.0958 - accuracy: 0.6165 - val_loss: 1.1560 - val_accuracy: 0.5936
Epoch 7/20
625/625 [==============================] - 9s 14ms/step - loss: 1.0285 - accuracy: 0.6430 - val_loss: 1.1399 - val_accuracy: 0.6019
Epoch 8/20
625/625 [==============================] - 9s 14ms/step - loss: 0.9696 - accuracy: 0.6651 - val_loss: 1.0912 - val_accuracy: 0.6115
Epoch 9/20
625/625 [==============================] - 8s 14ms/step - loss: 0.9168 - accuracy: 0.6874 - val_loss: 1.0715 - val_accuracy: 0.6240
Epoch 10/20
625/625 [==============================] - 9s 14ms/step - loss: 0.8670 - accuracy: 0.7024 - val_loss: 1.0442 - val_accuracy: 0.6374
Epoch 11/20
625/625 [==============================] - 9s 14ms/step - loss: 0.8174 - accuracy: 0.7221 - val_loss: 1.0193 - val_accuracy: 0.6452
Epoch 12/20
625/625 [==============================] - 9s 14ms/step - loss: 0.7738 - accuracy: 0.7370 - val_loss: 1.0044 - val_accuracy: 0.6496
Epoch 13/20
625/625 [==============================] - 9s 14ms/step - loss: 0.7339 - accuracy: 0.7548 - val_loss: 0.9977 - val_accuracy: 0.6520
Epoch 14/20
625/625 [==============================] - 9s 14ms/step - loss: 0.6922 - accuracy: 0.7702 - val_loss: 0.9833 - val_accuracy: 0.6560
Epoch 15/20
625/625 [==============================] - 9s 14ms/step - loss: 0.6551 - accuracy: 0.7839 - val_loss: 1.0217 - val_accuracy: 0.6425
Epoch 16/20
625/625 [==============================] - 9s 14ms/step - loss: 0.6174 - accuracy: 0.7990 - val_loss: 0.9769 - val_accuracy: 0.6619
Epoch 17/20
625/625 [==============================] - 8s 13ms/step - loss: 0.5817 - accuracy: 0.8122 - val_loss: 0.9687 - val_accuracy: 0.6674
Epoch 18/20
625/625 [==============================] - 9s 14ms/step - loss: 0.5454 - accuracy: 0.8265 - val_loss: 0.9778 - val_accuracy: 0.6638
Epoch 19/20
625/625 [==============================] - 9s 14ms/step - loss: 0.5111 - accuracy: 0.8403 - val_loss: 0.9793 - val_accuracy: 0.6636
Epoch 20/20
625/625 [==============================] - 8s 13ms/step - loss: 0.4778 - accuracy: 0.8558 - val_loss: 0.9768 - val_accuracy: 0.6674
313/313 [==============================] - 2s 5ms/step - loss: 2.3889 - accuracy: 0.3754
> 37.540
```


Cross Entropy Loss
train
test
loss
epoch
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Classification Accuracy
train
test
accuracy
epoch
0.8
0.7
0.6
0.5
0.4
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

2.4 ADAM VS SGD

In this part we intend to analyse the effect of replacing ADAM solver instead of SGD activation function in minimizing cost function on accuracy.

You can see my implementation below:

```
def define_model_baseline1_dropout(dropout_percent):
  model = Sequential()
  model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=(32, 32, 3)))
  model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Dropout(dropout_percent))
  model.add(Flatten())
  model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
  model.add(Dropout(dropout_percent))
  model.add(Dense(10, activation='softmax'))
  # compile model
  opt = SGD(learning_rate=0.001, momentum=0.9)
  model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
  return model
```

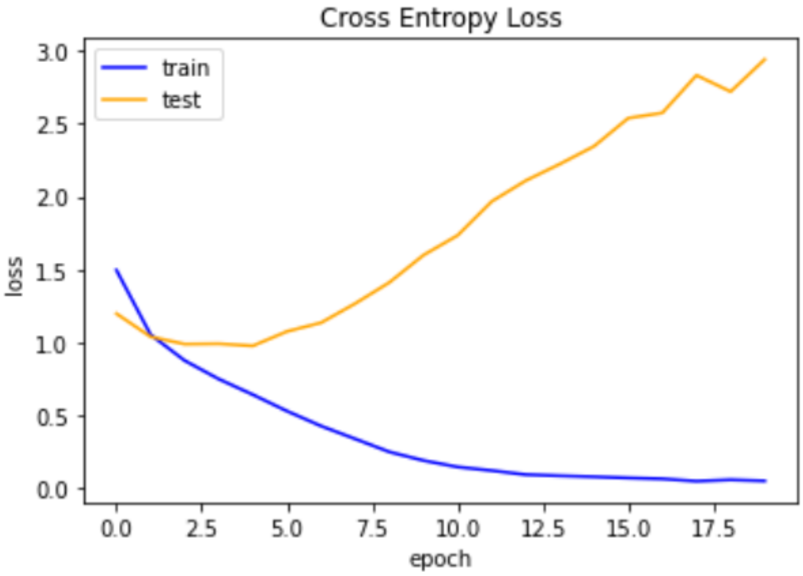
<u>Results have been showed that the ADAM solver acts pretty much better in comparison with SGD activation function!</u>

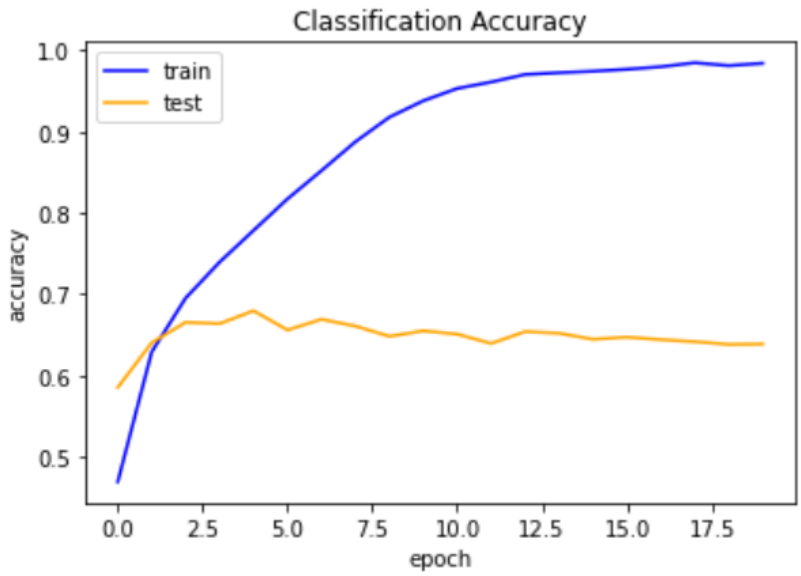
Further you can see the result of 1 hidden layer CNN for 20 epochs with ADAM!

Note that the SGD baseline has been showed in previous part so I have ignored to show the results of that in this part!

As you can see further the accuracy for ADAM is **56%** which is very low in comparison to SGD!

```
Epoch 1/20
625/625 [==============================] - 10s 16ms/step - loss: 1.4960 - accuracy: 0.4695 - val_loss: 1.1975 - val_accuracy: 0.5855
Epoch 2/20
625/625 [==============================] - 10s 16ms/step - loss: 1.0534 - accuracy: 0.6291 - val_loss: 1.0390 - val_accuracy: 0.6404
Epoch 3/20
625/625 [==============================] - 9s 14ms/step - loss: 0.8754 - accuracy: 0.6953 - val_loss: 0.9884 - val_accuracy: 0.6656
Epoch 4/20
625/625 [==============================] - 9s 14ms/step - loss: 0.7487 - accuracy: 0.7395 - val_loss: 0.9908 - val_accuracy: 0.6638
Epoch 5/20
625/625 [==============================] - 8s 13ms/step - loss: 0.6415 - accuracy: 0.7785 - val_loss: 0.9766 - val_accuracy: 0.6797
Epoch 6/20
625/625 [==============================] - 8s 13ms/step - loss: 0.5288 - accuracy: 0.8172 - val_loss: 1.0753 - val_accuracy: 0.6561
Epoch 7/20
625/625 [==============================] - 9s 14ms/step - loss: 0.4263 - accuracy: 0.8519 - val_loss: 1.1352 - val_accuracy: 0.6692
Epoch 8/20
625/625 [==============================] - 8s 13ms/step - loss: 0.3379 - accuracy: 0.8874 - val_loss: 1.2669 - val_accuracy: 0.6609
Epoch 9/20
625/625 [==============================] - 8s 13ms/step - loss: 0.2489 - accuracy: 0.9176 - val_loss: 1.4114 - val_accuracy: 0.6483
Epoch 10/20
625/625 [==============================] - 9s 14ms/step - loss: 0.1898 - accuracy: 0.9378 - val_loss: 1.5988 - val_accuracy: 0.6550
Epoch 11/20
625/625 [==============================] - 9s 14ms/step - loss: 0.1455 - accuracy: 0.9529 - val_loss: 1.7341 - val_accuracy: 0.6510
Epoch 12/20
625/625 [==============================] - 9s 14ms/step - loss: 0.1207 - accuracy: 0.9608 - val_loss: 1.9671 - val_accuracy: 0.6395
Epoch 13/20
625/625 [==============================] - 8s 13ms/step - loss: 0.0929 - accuracy: 0.9701 - val_loss: 2.1102 - val_accuracy: 0.6541
Epoch 14/20
625/625 [==============================] - 9s 14ms/step - loss: 0.0850 - accuracy: 0.9723 - val_loss: 2.2240 - val_accuracy: 0.6521
Epoch 15/20
625/625 [==============================] - 9s 14ms/step - loss: 0.0784 - accuracy: 0.9740 - val_loss: 2.3458 - val_accuracy: 0.6447
Epoch 16/20
625/625 [==============================] - 9s 14ms/step - loss: 0.0704 - accuracy: 0.9765 - val_loss: 2.5384 - val_accuracy: 0.6473
Epoch 17/20
625/625 [==============================] - 9s 14ms/step - loss: 0.0638 - accuracy: 0.9796 - val_loss: 2.5743 - val_accuracy: 0.6442
Epoch 18/20
625/625 [==============================] - 8s 13ms/step - loss: 0.0484 - accuracy: 0.9845 - val_loss: 2.8326 - val_accuracy: 0.6417
Epoch 19/20
625/625 [==============================] - 8s 13ms/step - loss: 0.0578 - accuracy: 0.9810 - val_loss: 2.7204 - val_accuracy: 0.6384
Epoch 20/20
625/625 [==============================] - 8s 13ms/step - loss: 0.0503 - accuracy: 0.9838 - val_loss: 2.9412 - val_accuracy: 0.6388
313/313 [==============================] - 2s 6ms/step - loss: 779.2141 - accuracy: 0.5658
> 56.580
```


Cross Entropy Loss
train
test
loss
epoch


Classification Accuracy
train
test
accuracy
epoch

2.5 Effect of adding Dropout layer

In this part we intend to analyse the effect of adding Dropout layer in on accuracy.

You can see my implementation below:

```
def define_model_baseline1_dropout(dropout_percent):
  model = Sequential()
  model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=(32, 32, 3)))
  model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
  model.add(MaxPooling2D((2, 2)))
  model.add(Dropout(dropout_percent))
  model.add(Flatten())
  model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
  model.add(Dropout(dropout_percent))
  model.add(Dense(10, activation='softmax'))
  # compile model
  opt = SGD(learning_rate=0.001, momentum=0.9)
  model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
  return model
```

I have used four values as of input argument which abbreviated below:

[0.1 0.2 0.25 0.3]

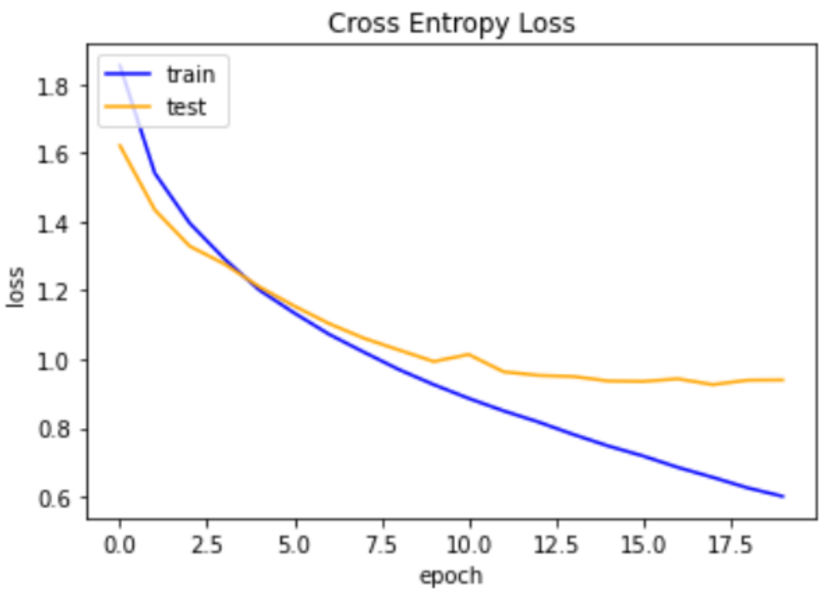
Results have been showed that the best values for dropout_percent is **0.1** and with testing based on this value we have obtained best accuracy.

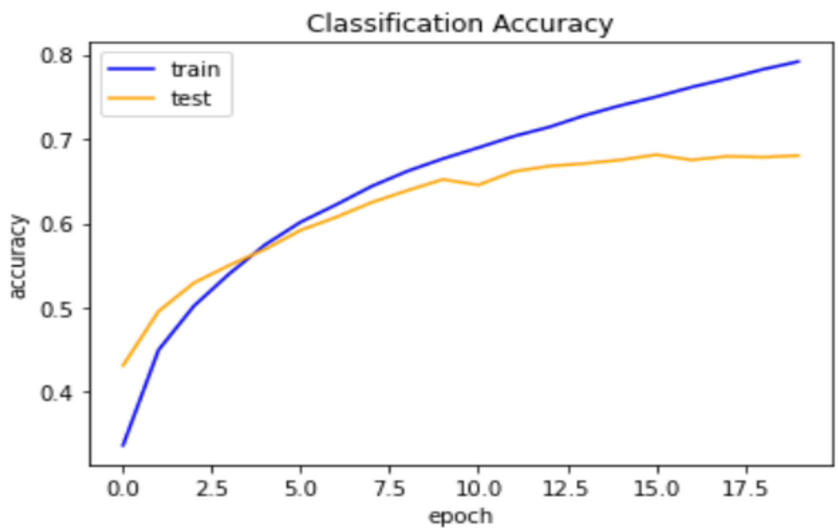
Further you can see the result of 1 hidden layer CNN for 20 epochs with different dropout percent parameters based on above values.

As you can see further the accuracy for **0.1** dropout percent is **54%** which is very considerable in comparison with each other!

```
Epoch 1/20
625/625 [==============================] - 11s 17ms/step - loss: 1.8543 - accuracy: 0.3368 - val_loss: 1.6215 - val_accuracy: 0.4315
Epoch 2/20
625/625 [==============================] - 10s 15ms/step - loss: 1.5412 - accuracy: 0.4499 - val_loss: 1.4344 - val_accuracy: 0.4958
Epoch 3/20
625/625 [==============================] - 9s 14ms/step - loss: 1.3959 - accuracy: 0.5023 - val_loss: 1.3289 - val_accuracy: 0.5296
Epoch 4/20
625/625 [==============================] - 9s 14ms/step - loss: 1.2916 - accuracy: 0.5405 - val_loss: 1.2769 - val_accuracy: 0.5506
Epoch 5/20
625/625 [==============================] - 10s 16ms/step - loss: 1.2009 - accuracy: 0.5746 - val_loss: 1.2104 - val_accuracy: 0.5695
Epoch 6/20
625/625 [==============================] - 9s 15ms/step - loss: 1.1342 - accuracy: 0.6015 - val_loss: 1.1545 - val_accuracy: 0.5920
Epoch 7/20
625/625 [==============================] - 9s 14ms/step - loss: 1.0722 - accuracy: 0.6221 - val_loss: 1.1033 - val_accuracy: 0.6074
Epoch 8/20
625/625 [==============================] - 10s 15ms/step - loss: 1.0202 - accuracy: 0.6440 - val_loss: 1.0608 - val_accuracy: 0.6249
Epoch 9/20
625/625 [==============================] - 9s 14ms/step - loss: 0.9698 - accuracy: 0.6618 - val_loss: 1.0267 - val_accuracy: 0.6390
Epoch 10/20
625/625 [==============================] - 9s 14ms/step - loss: 0.9258 - accuracy: 0.6765 - val_loss: 0.9932 - val_accuracy: 0.6521
Epoch 11/20
625/625 [==============================] - 9s 15ms/step - loss: 0.8858 - accuracy: 0.6896 - val_loss: 1.0137 - val_accuracy: 0.6455
```

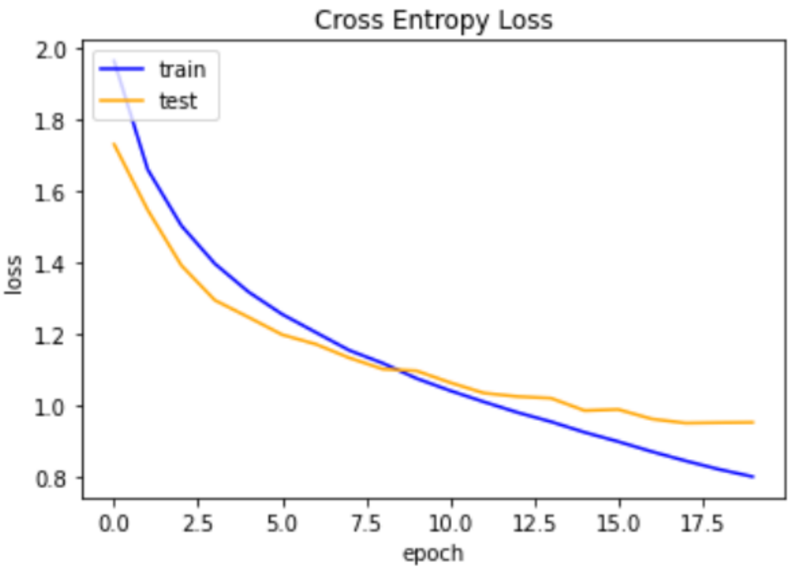
```
Epoch 11/20
625/625 [==============================] - 9s 15ms/step - loss: 0.8858 - accuracy: 0.6896 - val_loss: 1.0137 - val_accuracy: 0.6455
Epoch 12/20
625/625 [==============================] - 9s 14ms/step - loss: 0.8496 - accuracy: 0.7032 - val_loss: 0.9634 - val_accuracy: 0.6616
Epoch 13/20
625/625 [==============================] - 9s 15ms/step - loss: 0.8172 - accuracy: 0.7142 - val_loss: 0.9528 - val_accuracy: 0.6679
Epoch 14/20
625/625 [==============================] - 10s 15ms/step - loss: 0.7809 - accuracy: 0.7280 - val_loss: 0.9496 - val_accuracy: 0.6710
Epoch 15/20
625/625 [==============================] - 9s 14ms/step - loss: 0.7474 - accuracy: 0.7397 - val_loss: 0.9366 - val_accuracy: 0.6752
Epoch 16/20
625/625 [==============================] - 9s 14ms/step - loss: 0.7183 - accuracy: 0.7499 - val_loss: 0.9353 - val_accuracy: 0.6814
Epoch 17/20
625/625 [==============================] - 9s 14ms/step - loss: 0.6849 - accuracy: 0.7614 - val_loss: 0.9428 - val_accuracy: 0.6752
Epoch 18/20
625/625 [==============================] - 9s 15ms/step - loss: 0.6560 - accuracy: 0.7714 - val_loss: 0.9259 - val_accuracy: 0.6795
Epoch 19/20
625/625 [==============================] - 8s 13ms/step - loss: 0.6253 - accuracy: 0.7824 - val_loss: 0.9388 - val_accuracy: 0.6785
Epoch 20/20
625/625 [==============================] - 9s 14ms/step - loss: 0.6009 - accuracy: 0.7917 - val_loss: 0.9396 - val_accuracy: 0.6803
313/313 [==============================] - 2s 5ms/step - loss: 216.1297 - accuracy: 0.5441
> 54.410
```

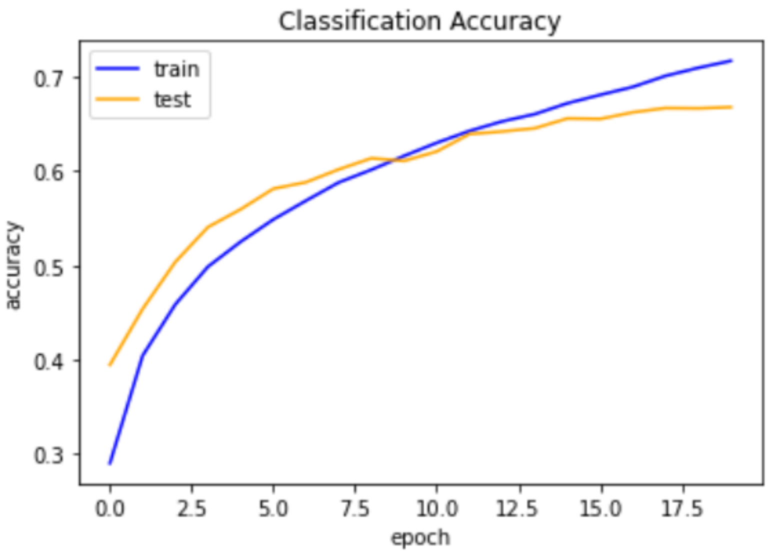




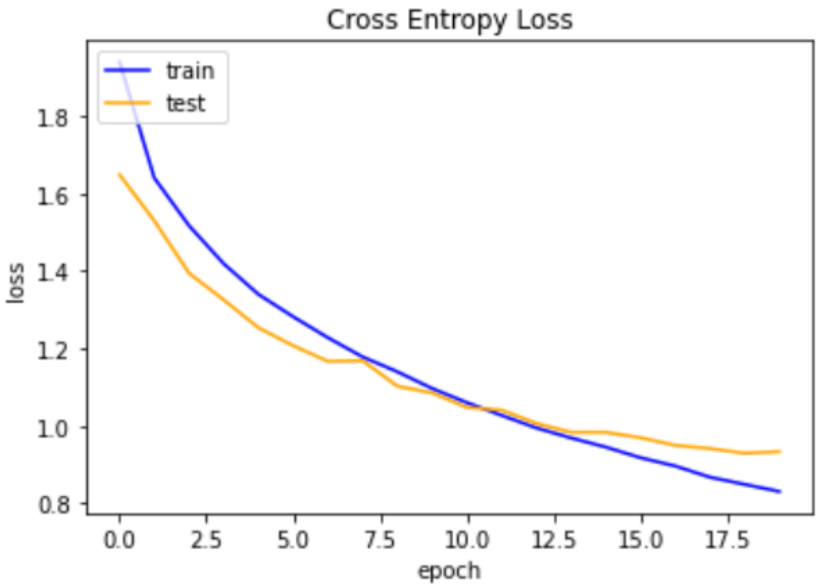
```
Epoch 1/20
625/625 [==============================] - 10s 15ms/step - loss: 1.9636 - accuracy: 0.2904 - val_loss: 1.7307 - val_accuracy: 0.3947
Epoch 2/20
625/625 [==============================] - 10s 15ms/step - loss: 1.6603 - accuracy: 0.4044 - val_loss: 1.5474 - val_accuracy: 0.4540
Epoch 3/20
625/625 [==============================] - 9s 15ms/step - loss: 1.5039 - accuracy: 0.4588 - val_loss: 1.3928 - val_accuracy: 0.5039
Epoch 4/20
625/625 [==============================] - 9s 14ms/step - loss: 1.3971 - accuracy: 0.4988 - val_loss: 1.2952 - val_accuracy: 0.5408
Epoch 5/20
625/625 [==============================] - 9s 15ms/step - loss: 1.3185 - accuracy: 0.5253 - val_loss: 1.2478 - val_accuracy: 0.5597
Epoch 6/20
625/625 [==============================] - 9s 14ms/step - loss: 1.2562 - accuracy: 0.5489 - val_loss: 1.1990 - val_accuracy: 0.5815
Epoch 7/20
625/625 [==============================] - 9s 14ms/step - loss: 1.2055 - accuracy: 0.5688 - val_loss: 1.1726 - val_accuracy: 0.5884
Epoch 8/20
625/625 [==============================] - 9s 14ms/step - loss: 1.1552 - accuracy: 0.5881 - val_loss: 1.1343 - val_accuracy: 0.6020
Epoch 9/20
625/625 [==============================] - 10s 15ms/step - loss: 1.1193 - accuracy: 0.6016 - val_loss: 1.1029 - val_accuracy: 0.6138
Epoch 10/20
625/625 [==============================] - 10s 15ms/step - loss: 1.0768 - accuracy: 0.6162 - val_loss: 1.0980 - val_accuracy: 0.6110
Epoch 11/20
625/625 [==============================] - 9s 15ms/step - loss: 1.0427 - accuracy: 0.6302 - val_loss: 1.0652 - val_accuracy: 0.6210
Epoch 12/20
625/625 [==============================] - 10s 15ms/step - loss: 1.0115 - accuracy: 0.6427 - val_loss: 1.0360 - val_accuracy: 0.6394
Epoch 13/20
625/625 [==============================] - 10s 15ms/step - loss: 0.9819 - accuracy: 0.6528 - val_loss: 1.0263 - val_accuracy: 0.6422
Epoch 14/20
625/625 [==============================] - 10s 15ms/step - loss: 0.9560 - accuracy: 0.6605 - val_loss: 1.0216 - val_accuracy: 0.6456
```

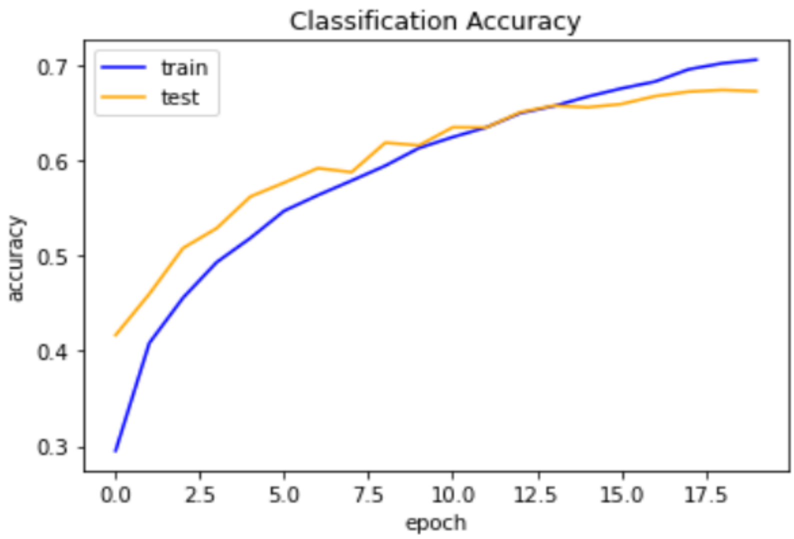
```
Epoch 15/20
625/625 [==============================] - 10s 15ms/step - loss: 0.9263 - accuracy: 0.6721 - val_loss: 0.9871 - val_accuracy: 0.6560
Epoch 16/20
625/625 [==============================] - 9s 15ms/step - loss: 0.9005 - accuracy: 0.6809 - val_loss: 0.9902 - val_accuracy: 0.6555
Epoch 17/20
625/625 [==============================] - 9s 15ms/step - loss: 0.8725 - accuracy: 0.6894 - val_loss: 0.9637 - val_accuracy: 0.6625
Epoch 18/20
625/625 [==============================] - 10s 16ms/step - loss: 0.8468 - accuracy: 0.7011 - val_loss: 0.9521 - val_accuracy: 0.6670
Epoch 19/20
625/625 [==============================] - 10s 16ms/step - loss: 0.8229 - accuracy: 0.7096 - val_loss: 0.9532 - val_accuracy: 0.6667
Epoch 20/20
625/625 [==============================] - 10s 15ms/step - loss: 0.8026 - accuracy: 0.7171 - val_loss: 0.9539 - val_accuracy: 0.6680
313/313 [==============================] - 2s 5ms/step - loss: 159.8818 - accuracy: 0.5049
> 50.490
```





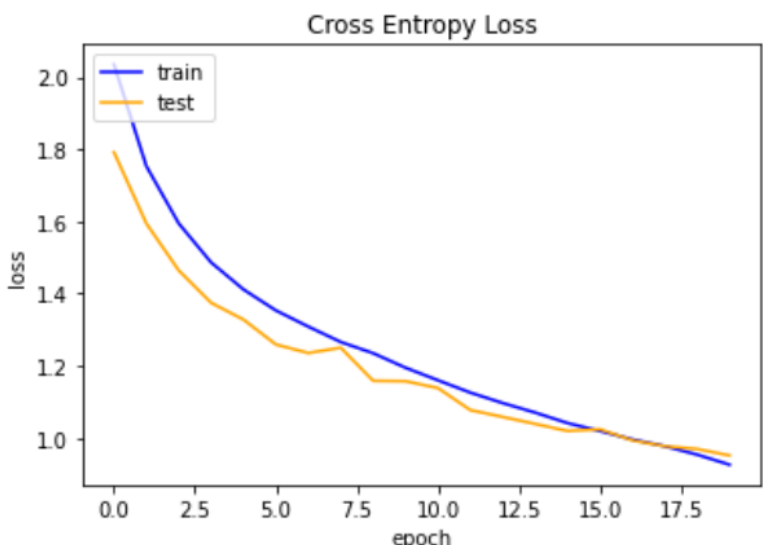

```
Epoch 1/20
625/625 [==============================] - 15s 22ms/step - loss: 1.9400 - accuracy: 0.2952 - val_loss: 1.6490 - val_accuracy: 0.4165
Epoch 2/20
625/625 [==============================] - 10s 16ms/step - loss: 1.6406 - accuracy: 0.4082 - val_loss: 1.5299 - val_accuracy: 0.4597
Epoch 3/20
625/625 [==============================] - 9s 15ms/step - loss: 1.5173 - accuracy: 0.4557 - val_loss: 1.3947 - val_accuracy: 0.5079
Epoch 4/20
625/625 [==============================] - 9s 14ms/step - loss: 1.4190 - accuracy: 0.4931 - val_loss: 1.3256 - val_accuracy: 0.5288
Epoch 5/20
625/625 [==============================] - 9s 15ms/step - loss: 1.3399 - accuracy: 0.5185 - val_loss: 1.2537 - val_accuracy: 0.5620
Epoch 6/20
625/625 [==============================] - 9s 14ms/step - loss: 1.2821 - accuracy: 0.5470 - val_loss: 1.2074 - val_accuracy: 0.5766
Epoch 7/20
625/625 [==============================] - 9s 14ms/step - loss: 1.2281 - accuracy: 0.5634 - val_loss: 1.1669 - val_accuracy: 0.5918
Epoch 8/20
625/625 [==============================] - 9s 14ms/step - loss: 1.1783 - accuracy: 0.5787 - val_loss: 1.1688 - val_accuracy: 0.5876
Epoch 9/20
625/625 [==============================] - 9s 14ms/step - loss: 1.1401 - accuracy: 0.5943 - val_loss: 1.1028 - val_accuracy: 0.6186
Epoch 10/20
625/625 [==============================] - 9s 15ms/step - loss: 1.0970 - accuracy: 0.6129 - val_loss: 1.0854 - val_accuracy: 0.6157
Epoch 11/20
625/625 [==============================] - 9s 14ms/step - loss: 1.0604 - accuracy: 0.6244 - val_loss: 1.0479 - val_accuracy: 0.6347
Epoch 12/20
625/625 [==============================] - 10s 16ms/step - loss: 1.0274 - accuracy: 0.6346 - val_loss: 1.0411 - val_accuracy: 0.6345
Epoch 13/20
Epoch 14/20
625/625 [==============================] - 9s 14ms/step - loss: 0.9691 - accuracy: 0.6569 - val_loss: 0.9838 - val_accuracy: 0.6575
Epoch 15/20
625/625 [==============================] - 9s 14ms/step - loss: 0.9456 - accuracy: 0.6669 - val_loss: 0.9838 - val_accuracy: 0.6557
Epoch 16/20
625/625 [==============================] - 9s 15ms/step - loss: 0.9180 - accuracy: 0.6754 - val_loss: 0.9698 - val_accuracy: 0.6591
Epoch 17/20
625/625 [==============================] - 9s 15ms/step - loss: 0.8967 - accuracy: 0.6827 - val_loss: 0.9501 - val_accuracy: 0.6674
Epoch 18/20
625/625 [==============================] - 9s 15ms/step - loss: 0.8678 - accuracy: 0.6955 - val_loss: 0.9418 - val_accuracy: 0.6721
Epoch 19/20
625/625 [==============================] - 8s 14ms/step - loss: 0.8492 - accuracy: 0.7019 - val_loss: 0.9297 - val_accuracy: 0.6739
Epoch 20/20
625/625 [==============================] - 8s 14ms/step - loss: 0.8309 - accuracy: 0.7055 - val_loss: 0.9339 - val_accuracy: 0.6726
313/313 [==============================] - 2s 5ms/step - loss: 164.5964 - accuracy: 0.5191
> 51.910
```

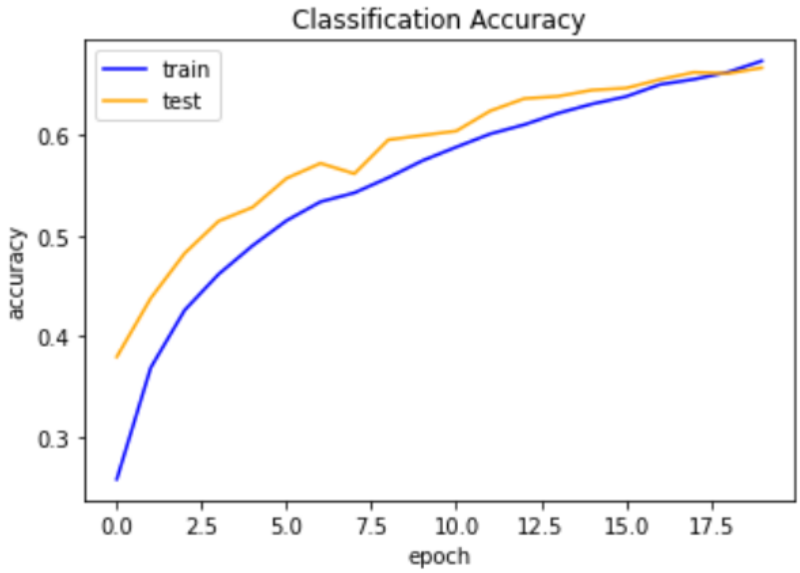

Cross Entropy Loss
train
test
loss
1.8
1.6
1.4
1.2
1.0
0.8
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
epoch


Classification Accuracy
train
test
accuracy
0.7
0.6
0.5
0.4
0.3
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
epoch

```
Epoch 1/20
625/625 [==============================] - 10s 16ms/step - loss: 2.0351 - accuracy: 0.2580 - val_loss: 1.7918 - val_accuracy: 0.3796
Epoch 2/20
625/625 [==============================] - 10s 16ms/step - loss: 1.7531 - accuracy: 0.3688 - val_loss: 1.5942 - val_accuracy: 0.4378
Epoch 3/20
625/625 [==============================] - 9s 15ms/step - loss: 1.5944 - accuracy: 0.4261 - val_loss: 1.4640 - val_accuracy: 0.4825
Epoch 4/20
625/625 [==============================] - 9s 15ms/step - loss: 1.4862 - accuracy: 0.4618 - val_loss: 1.3743 - val_accuracy: 0.5146
Epoch 5/20
625/625 [==============================] - 9s 15ms/step - loss: 1.4112 - accuracy: 0.4902 - val_loss: 1.3277 - val_accuracy: 0.5283
Epoch 6/20
625/625 [==============================] - 10s 15ms/step - loss: 1.3530 - accuracy: 0.5150 - val_loss: 1.2588 - val_accuracy: 0.5571
Epoch 7/20
625/625 [==============================] - 9s 14ms/step - loss: 1.3083 - accuracy: 0.5337 - val_loss: 1.2351 - val_accuracy: 0.5720
Epoch 8/20
625/625 [==============================] - 9s 15ms/step - loss: 1.2657 - accuracy: 0.5426 - val_loss: 1.2500 - val_accuracy: 0.5617
Epoch 9/20
625/625 [==============================] - 9s 15ms/step - loss: 1.2347 - accuracy: 0.5577 - val_loss: 1.1582 - val_accuracy: 0.5954
Epoch 10/20
625/625 [==============================] - 9s 15ms/step - loss: 1.1944 - accuracy: 0.5745 - val_loss: 1.1572 - val_accuracy: 0.5998
Epoch 11/20
625/625 [==============================] - 9s 15ms/step - loss: 1.1597 - accuracy: 0.5881 - val_loss: 1.1382 - val_accuracy: 0.6041
Epoch 12/20
625/625 [==============================] - 10s 15ms/step - loss: 1.1254 - accuracy: 0.6012 - val_loss: 1.0771 - val_accuracy: 0.6241
Epoch 13/20
625/625 [==============================] - 9s 15ms/step - loss: 1.0965 - accuracy: 0.6103 - val_loss: 1.0581 - val_accuracy: 0.6363
Epoch 14/20
625/625 [==============================] - 10s 15ms/step - loss: 1.0701 - accuracy: 0.6217 - val_loss: 1.0388 - val_accuracy: 0.6386
```

```
Epoch 14/20
625/625 [==============================] - 10s 15ms/step - loss: 1.0701 - accuracy: 0.6217 - val_loss: 1.0388 - val_accuracy: 0.6386
Epoch 15/20
625/625 [==============================] - 9s 15ms/step - loss: 1.0410 - accuracy: 0.6309 - val_loss: 1.0192 - val_accuracy: 0.6448
Epoch 16/20
625/625 [==============================] - 9s 15ms/step - loss: 1.0191 - accuracy: 0.6383 - val_loss: 1.0239 - val_accuracy: 0.6467
Epoch 17/20
625/625 [==============================] - 10s 15ms/step - loss: 0.9958 - accuracy: 0.6503 - val_loss: 0.9931 - val_accuracy: 0.6553
Epoch 18/20
625/625 [==============================] - 10s 15ms/step - loss: 0.9770 - accuracy: 0.6555 - val_loss: 0.9769 - val_accuracy: 0.6624
Epoch 19/20
625/625 [==============================] - 9s 14ms/step - loss: 0.9536 - accuracy: 0.6625 - val_loss: 0.9692 - val_accuracy: 0.6613
Epoch 20/20
625/625 [==============================] - 9s 14ms/step - loss: 0.9258 - accuracy: 0.6737 - val_loss: 0.9512 - val_accuracy: 0.6669
313/313 [==============================] - 2s 6ms/step - loss: 125.5179 - accuracy: 0.4941
> 49.410
```





For running this part please have an accurate look on related directory and There you can easily find all of you need and also I would appreciate it if you would consider them ☺

Here you can find all of my code implementation using Python if there is any issue with that please let me know to provide more detailed information. (be in touch **here!**)

3 QUESTION #2

3.1 FULLY CONNECTED NEURAL NETWORK(THEORETICAL)

In this part we intend to repeat algorithm for two separated cycles for both forwarding and back-propagating, you can see more detailed information below where I have done all of step for a theoretical approach.

پاسخ قسمت تحلیلی

4 9 4

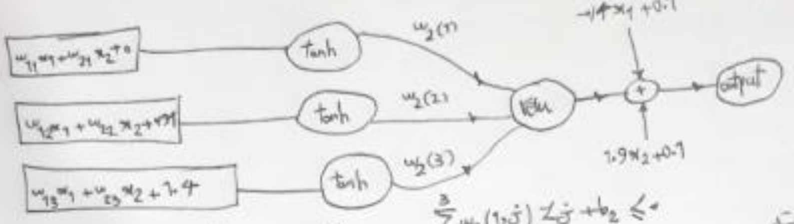
y b a

$$w_1 = \begin{bmatrix} 1.49 & -0.74 & 0.9 \\ 0.29 & -1.4 & 0.93 \end{bmatrix}$$

$$w_2 = \begin{bmatrix} 1.45 & -0.59 & 0.94 \end{bmatrix} \quad , \; b_2 = 0.1$$

$$x = \begin{bmatrix} In1 \\ In2 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \qquad w_3 = \begin{bmatrix} -0.4 \\ 1.9 \end{bmatrix} \qquad b_1 = \begin{pmatrix} 0 \\ 0.91 \\ 1.4 \end{pmatrix}$$

$$b_3 = 0.1$$



به صورت کلی روابط زیر را خواهیم داشت

$$\frac{\partial h}{\partial \hat{y}} = (\hat{y} - y) \quad , \quad \frac{\partial \hat{y}}{\partial w_2(1,j)} = \begin{cases} 0 & \sum_{j=1}^{3} w_2(1,j) Z_j + b_2 \leq 0 \\ Z_j & \sum_{j=1}^{3} w_2(1,j) Z_j + b_2 > 0 \end{cases}$$

$$\frac{\partial \hat{y}}{\partial b_2} = \begin{cases} 0 & \sum_{j=1}^{3} w_2(1,j) Z_j + b_2 \leq 0 \\ 1 & \sum_{j=1}^{3} w_2(1,j) Z_j + b_2 > 0 \end{cases} \qquad \frac{\partial \hat{y}}{\partial w_3(j,1)} = x_i \qquad \frac{\partial \hat{y}}{\partial b_3} = 1$$

$$\frac{\partial Z_i}{\partial w_1(i,j)} = x_i \left[1 - \tanh^2\left[\sum_{i=1}^{2} w_1(i,j) x_i + b_1(j) \right] \right] \qquad j = 1, 2, 3$$

$$\frac{\partial Z_i}{\partial b_1(j)} = 1 - \tanh^2\left[\sum_{i=1}^{2} w_1(i,j) x_i + b_1(j) \right] , \; j = 1, 2, 3$$

حال برای مرحله ۱ خواهیم داشت

$$Z_1 = \tanh\left[(1.49 \times 2 + 0.29 \times 3 + 0) \right] = 0.99$$

$$Z_2 = \tanh\left[-0.74 \times 2 - 1.4 \times 3 + 0.91 \right] = -0.99$$

$$Z_3 = \tanh\left[0.9 \times 2 + 3 \times 0.93 + 1.4 \right] = 0.99$$

$$Relu \rightarrow \max\left[1.45 Z_1 + 0.94 Z_3 - 0.59 Z_2 + 0.1 , 0 \right]$$

$$= 2.9509 + 0.1 = 3.0509$$

$$\rightarrow Relu_{out} = 3.0509$$

$$\hat{y} = 3.0509 - 0.4 \times 2 + 0.1 + 1.9 \times 3 + 0.1 = \boxed{8.1509}$$

$$\rightarrow \begin{cases} y = 4 \\ \hat{y} = 8.1509 \end{cases} \rightarrow \hat{y} - y = 4.1509$$

حال خواهیم داشت

$$\frac{\partial \hat{y}}{\partial w_2(1,1)} = 0.99 \qquad \frac{\partial \hat{y}}{\partial b_2} = 1 \; , \; \frac{\partial \hat{y}}{\partial w_3(j,1)} = x_1 \qquad \frac{\partial \hat{y}}{\partial w_3(j,2)} = x_2 \qquad \frac{\partial \hat{y}}{\partial b_3} = 1$$

$$\frac{\partial \hat{y}}{\partial w_2(1,2)} = -0.99$$

$$\frac{\partial \hat{y}}{\partial w_2(1,3)} = 0.99$$

حال طبق روابط بدست آمده خواهیم داشت ←

$$\frac{\partial \hat{L}}{\partial w_{11}} = (\hat{y}-y) \times 1 \times 1.45 \times x_1 \times \left[1 - z_1^2\right] = 0/2395$$

$$\frac{\partial \hat{L}}{\partial w_{21}} = (\hat{y}-y) \times 1 \times 1.45 \times x_2 \times \left[1 - z_1^2\right] = 0/3593$$

$$\frac{\partial \hat{L}}{\partial w_{12}} = 4.1509 \times 1 \times [-0/59] \times 2 \times \left[1-(0/99)^2\right] = -0.09747$$

$$\frac{\partial \hat{L}}{\partial w_{22}} = 4.1509 \times 1 \times [-0/59] \times 3 \times \left[1-(0/99)^2\right] = -0.1462$$

$$\frac{\partial \hat{L}}{\partial w_{13}} = 4.1509 \times 1 \times [0/94] \times 2 \times \left[1-(0/99)^2\right] = 0/1553$$

$$\frac{\partial \hat{L}}{\partial w_{23}} = 4.1509 \times 1 \times [0/94] \times 3 \times \left[1-[0/99]^2\right] = 0/2329$$

$$\begin{cases} \dfrac{\partial L}{\partial b_1(1)} = 0/11975 \\ \dfrac{\partial L}{\partial b_1(2)} = -0/048735 \\ \dfrac{\partial L}{\partial b_1(3)} = 0/07765 \end{cases}$$

$$\frac{\partial L}{\partial w_3(1)} = 4.1509 \times 2 = 8.3018 \quad , \quad \frac{\partial L}{\partial w_3(2)} = 4.1509 \times 3 = 12.4527 \quad , \quad \begin{cases} \dfrac{\partial L}{\partial b_3} = 4.1509 \\ \dfrac{\partial L}{\partial b_2} = 4.1509 \end{cases}$$

$$\frac{\partial L}{\partial w_2(1)} = 4.1509 \times 0/99 = \boxed{4.109} \quad * \quad \frac{\partial L}{\partial w_2(2)} = 4.1509 \times -0/99 = \boxed{-4.109391}$$

$$\frac{\partial L}{\partial w_2(3)} = 4.1509 \times 0/99 = \boxed{4.109391}$$

$$\Rightarrow w_3 = \begin{bmatrix} -0/4 \\ 1.9 \end{bmatrix} - 0/1 \begin{bmatrix} 8.3018 \\ 12.4527 \end{bmatrix} = \begin{bmatrix} -1.23018 \\ +0/65473 \end{bmatrix}$$

$$b_2 = b_3 = 0/1 - 0/1 \times [4.1509] = -0/31509$$

$$w_2 = \begin{bmatrix} 1.45 & -0/59 & 0/94 \end{bmatrix} - 0/1 \begin{bmatrix} 4.109 & -4.109391 & 4.109391 \end{bmatrix} = \begin{bmatrix} 1.0391 & -0.1791 & 0/5291 \end{bmatrix}$$

$$b_1 = \begin{bmatrix} 0 \\ 0/91 \\ 1.4 \end{bmatrix} - 0/1 \begin{bmatrix} 0/11975 \\ -0/048735 \\ 0/07765 \end{bmatrix} = \begin{bmatrix} -0/01197 \\ 0/9051 \\ 1.3922 \end{bmatrix}$$

$$w_1 = \begin{bmatrix} 1.49 & -0/14 & 0.9 \\ 0/29 & -1.4 & 0.93 \end{bmatrix} - 0/1 \begin{bmatrix} 0/2395 & -0/09747 & 0/1553 \\ 0/3593 & -0/1462 & 0/2329 \end{bmatrix} = \begin{bmatrix} 1.46605 & -0/13026 & 0/88447 \\ 0/25407 & -1.38538 & 0/90671 \end{bmatrix}$$

حال برای حالت دوم خواهیم داشت ←

$$z_{1\text{-new}} = \tanh\left[1.46605 \times 2 + 3 \times 0/25407 - 0/01197\right] = 0/99$$

$$z_{2\text{-new}} = \tanh\left[-0/13026 \times 2 - 3 \times 1.38538 + 0/9051\right] = -0/99$$

$$z_{3\text{-new}} = \tanh\left[0/88447 \times 2 + 3 \times 0/90671 + 1.3922\right] = 0/99$$

$$\text{Relu} \rightarrow \max\left[1.0391 \times 0/99 + 0.1791 \times 0/99 + 0/99 \times 0/5291 - 0/31509\right] = 1.4147$$

$$\Rightarrow \hat{y} = 1.4147 - 2 \times [-1.23018] + 0/1 + 3 \times 0/65473 + 0/1 = 6.03925 \quad \Rightarrow \hat{y} - y = 6.03926 - 4 = \boxed{2.03926}$$

$$\frac{\partial L}{\partial w_{11}} = 2.03925 \times 1 \times 1.0391 \times 2\left[1-(0.99)^2\right] = 0.08433 \qquad \frac{\partial L}{\partial b_{11}} = 0.04058$$

$$\frac{\partial L}{\partial w_{21}} = 2.03925 \times 1 \times 1.0391 \times 3 \times \left[1-(0.99)^2\right] = 0.1217 \qquad \frac{\partial L}{\partial b_{12}} = 0.00726$$

$$\frac{\partial L}{\partial w_{12}} = 2.03925 \times 1 \times [-0.1791] \times 2 \times \left[1-0.99^2\right] = -0.01453 \qquad \frac{\partial L}{\partial b_{1(3)}} = 0.02147$$

$$\frac{\partial L}{\partial w_{22}} = 2.03925 \times 1 \times [-0.1791] \times 3 \times \left[1-0.99^2\right] = -0.02181$$

$$\frac{\partial L}{\partial w_{13}} = 2.03925 \times 1 \times [0.5291] \times 2 \times \left[1-0.99^2\right] = 0.04294 \qquad \frac{\partial L}{\partial w_{3(1)}} = 2 \times 2.03925 = 4.0785$$

$$\frac{\partial L}{\partial w_{23}} = 2.5147 \times 1 \times [0.5291] \times 3 \times \left[1-0.99^2\right] = 0.0644 \qquad \frac{\partial L}{\partial w_{3(2)}} = 3 \times 2.03925 = 6.11775$$

$$\frac{\partial L}{\partial b_3} = 2.03925 = \frac{\partial L}{\partial b_2} \qquad \frac{\partial L}{\partial w_{2(1)}} = 2.03925 \times 0.99 = 2.01885$$

$$\frac{\partial L}{\partial w_{2(2)}} = 2.03925 \times 0.99 = -2.01885$$

$$\frac{\partial L}{\partial w_{2(3)}} = 2.03925 \times 0.99 = +2.01885$$

حال برای update کردن متغیرها خواهیم داشت ⟵

$$w_3 = \begin{bmatrix} -1.23078 \\ +0.65473 \end{bmatrix} - 0.1 \begin{bmatrix} 4.0785 \\ 6.11775 \end{bmatrix} = \begin{bmatrix} -1.63803 \\ 0.042955 \end{bmatrix}$$

$$b_2 = b_3 = -0.31509 - 0.1\,(2.03925) = -0.519015$$

$$b_1 = \begin{bmatrix} -0.01197 \\ 0.9057 \\ 0.3922 \end{bmatrix} - 0.1 \begin{bmatrix} 0.04058 \\ 0.00726 \\ 0.02147 \end{bmatrix} = \begin{bmatrix} -0.016028 \\ 0.904374 \\ 1.390053 \end{bmatrix}$$

$$w_2 = \begin{bmatrix} 1.0391 & -0.1791 & 0.5291 \end{bmatrix} - 0.1 \begin{bmatrix} 2.01885 & -2.01885 & 2.01885 \end{bmatrix} = \begin{bmatrix} 0.837215 & 0.022785 & 0.327215 \end{bmatrix}$$

$$w_1 = \begin{bmatrix} 1.4665 & -0.13026 & 0.88447 \\ 0.25487 & -1.38538 & 0.90671 \end{bmatrix} - 0.1 \begin{bmatrix} 0.08433 & -0.01453 & 0.04294 \\ 0.1217 & -0.02181 & 0.0644 \end{bmatrix}$$

$$= \begin{bmatrix} 1.4576 & -0.1288 & 0.8802 \\ 0.2419 & -1.3832 & 0.9003 \end{bmatrix}$$

3.2 RESEARCH

In this part we intend to find solutions for several questions which let us to have a dipper insight on concepts!

3.2.1 Different cost function

L1 and L2 are two loss functions in machine learning which are used to minimize the error.

L1 Loss function stands for Least Absolute Deviations. Also known as LAD.

L2 Loss function stands for Least Square Errors. Also known as LS.

L1 Loss Function is used to minimize the error which is the sum of the all the absolute differences between the true value and the predicted value.

$$L1LossFunction = \sum_{i=1}^{n} |y_{true} - y_{predicted}|$$

L2 Loss Function is used to minimize the error which is the sum of the all the squared differences between the true value and the predicted value.

$$L2LossFunction = \sum_{i=1}^{n} (y_{true} - y_{predicted})^2$$

Generally, L2 Loss Function is preferred in most of the cases. But when the outliers are present in the dataset, then the L2 Loss Function does not perform well. The reason behind this bad performance is that if the dataset is having outliers, then because of the consideration of the squared differences, it leads to the much larger error. Hence, L2 Loss Function is not useful here. Prefer L1 Loss Function as it is not affected by the outliers or remove the outliers and then use L2 Loss Function.

Huber_Loss:

Huber Loss is loss function that is used in robust regression. It is the solution to problems faced by L1 and L2 loss functions. It is less sensitive to outliers in data than the squared error loss. It’s also differentiable at 0.

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for} |y - f(x)| \le \delta, \\ \delta\,|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

The above function becomes quadratic when error value a is small and linear when a is large.

Reason for using Huber Loss?

One big problem with using MAE is its constantly large gradient when using gradient decent for training. This can lead to missing minima at the end of training using gradient descent. While with MSE, gradient decreases as the loss gets close to its minima, making it more precise.

Huber loss can be here, as it curves around the minima which decreases the gradient.

Compared with L2 loss, Huber Loss is less sensitive to outliers (because if the residual is too large, it is a piecewise function, loss is a linear function of the residual). Among them, δ is a set parameter, y represents the real value and f(x) represent the predicted value.

The Huber function is less sensitive to small errors than the $\ell 1$ norm, but becomes linear in the error for large errors.

3.2.2 Using evaluation data

This happening may occur when our model trains as well as it could be and means that we face with a perfect model which caused the error for validation data and trained data become as much as close it could be!

And also it caused when the complication of validation data is less that trained data and in these case we also face with this happening.

3.2.3 Gradient descent using Momentum

Gradient descent is an optimization algorithm that follows the negative gradient of an objective function in order to locate the minimum of the function.

A problem with gradient descent is that it can bounce around the search space on optimization problems that have large amounts of curvature or noisy gradients, and it can get stuck in flat spots in the search space that have no gradient.

Momentum is an extension to the gradient descent optimization algorithm that allows the search to build inertia in a direction in the search space and overcome the oscillations of noisy gradients and coast across flat spots of the search space.

The answer is Gradient descent using momentum as we know Gradient descent can be accelerated by using momentum from past updates to the search position.

It is designed to accelerate the optimization process, e.g. decrease the number of function evaluations required to reach the optima, or to improve the capability of the optimization algorithm, e.g. result in a better final result.

The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

Momentum has the effect of dampening down the change in the gradient and, in turn, the step size with each new point in the search space.

Momentum can increase speed when the cost surface is highly nonspherical because it damps the size of the steps along directions of high curvature thus yielding a larger effective learning rate along the directions of low curvature.

Momentum is most useful in optimization problems where the objective function has a large amount of curvature (e.g. changes a lot), meaning that the gradient may change a lot over relatively small regions of the search space.

It is also helpful when the gradient is estimated, such as from a simulation, and may be noisy, e.g. when the gradient has a high variance.

Finally, momentum is helpful when the search space is flat or nearly flat, e.g. zero gradient. The momentum allows the search to progress in the same direction as before the flat spot and helpfully cross the flat region.

3.3 APPROXIMATE SINE FUNCTION (NO PYTHON LIBRARIES)

Implementation in Python:

In this section we intend to implement a fully connected neural network to approximate a sine function in interval of [0,2pi].

$$f(x, y) = \sin(x + y)\,; (x, y) \in [0, 2\pi] \times [0, 2\pi];$$

At the first step we are going to sampling from the [0,2pi] interval using random.uniform() function in python, afterward we make x1 and x2 vector as of our input neurons.

Afterward we use np.sin() function to obtain corresponding labels and finally we make dataset which have three rows:

First one normalize data values for x1-input

second one normalize data values for x2-input

and third one the corresponding labels for normalize data.

Note that in this question I have used 80 percent of data for training and also I have used 20 percent of that for evaluation and rest of that for testing!

And also Note that my ReLU function gets a vector as of its input and give us a vector.

I implement ReLU in this way cuase of my implementation in sine_train() function in next part.

And also I sample from data in [0,2pi] interval because I asked from correspond TA and she said me that for reaching best accuracy go through this way!

Some extra notes:

I have used one layer neural network with ReLU activation function in each node and also I have used stochastic gradient descent approach according to HW description!

Further you can see all of my codes using forwarding and back-propagation to reach a reasonable y value!

Note that at the end code block I have used a matrix called data_plot which has two rows first one is x2 real values without normalizing which used for ploting because the interval must be 0,2pi and after normalization it would be something between 0,1 and it’s not suitable for plotting!

And the normalize x2-value used for prediction because we train model with normalize data!

And unfortunately I have faced with a big problem!

I have checked whole algorithm more than once and everything was going true!

But I want to claim that the final result is just Mirror of correct answer!!

I talked with my friends and found that they were also struggling with same problem so I add a minus manually in my code which made result graphically correct but it's not still true 😅😅

But I used my best of efforts to correct that by double checking algorithm several times but I failed in all of my tries :((

Here you can find all of my code implementation using Python if there is any issue with that please let me know to provide more detailed information. (be in touch **here!**)

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random
np.random.seed(1000)
n=10000
y=np.zeros(n)

x1=np.random.uniform(0,2*(np.pi),n)
x2=np.random.uniform(0,2*(np.pi),n)


for i in range(n):
    y[i] = np.sin(x1[i]+x2[i])

x1_mean = sum(x1)/len(x1)
x2_mean = sum(x2)/len(x2)
x1_variance = np.std(x1)
x2_variance = np.std(x2)

x1_normalize = (x1-x1_mean)/x1_variance
x2_normalize = (x2-x2_mean)/x2_variance

data = np.array([x1_normalize,x2_normalize,y])
train_data = data[:,:int(0.6*n)]
validation_data = data[:,int(0.6*n):int(0.8*n)]
test_data = data[:,int(0.8*n):n]
```

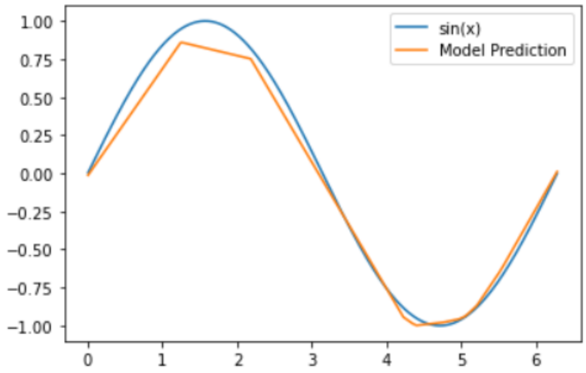
```
def ReLU(x):
    data=np.zeros(np.size(x))
    for i in range(np.size(x)):
        data[i]=max(0,x[i])

    return data

def der_ReLU(x):
    data=np.zeros(np.size(x))
    for i in range(np.size(x)):
        data[i] = 1 if x[i]>0 else 0

    return data

def Sine_train(train_data):

    epoch=30
    hidden_layer=150
    input_neurons=2
    output_neurons=1
    learning_rate=0.08
```

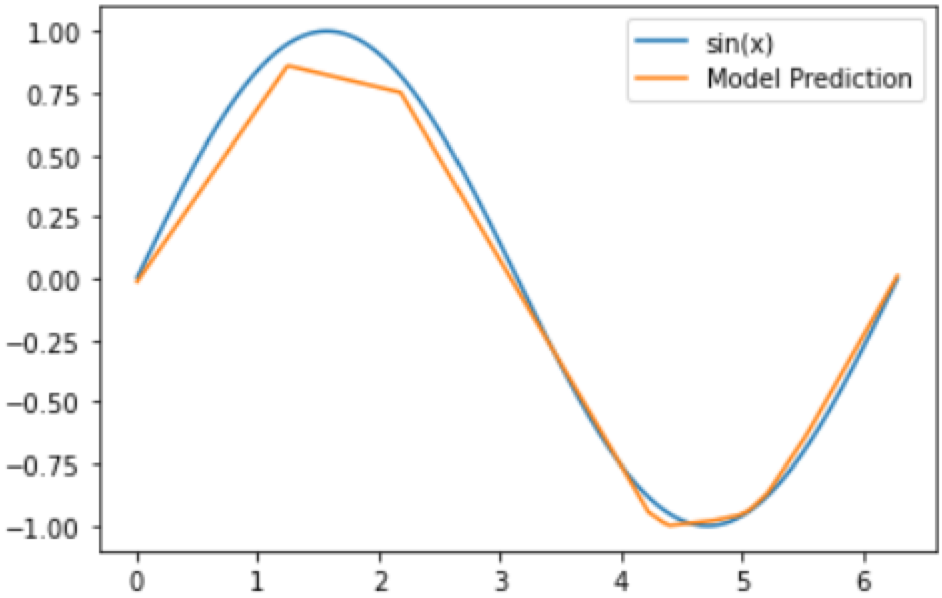
```python
def Sine_train(train_data):

    epoch=30
    hidden_layer=150
    input_neurons=2
    output_neurons=1
    learning_rate=0.08


    weights_in=  0.001 * np.random.randn(input_neurons,hidden_layer)
    bias_in=np.zeros(hidden_layer)

    weights_out= 0.001 * np.random.randn(hidden_layer)
    for i in range(epoch):
        for j in range(np.size(train_data[0,:])):

            x1_train=train_data[0,j]
            x2_train=train_data[1,j]
            y_train= train_data[2,j]
            #forward propogation

            hidden_layer_input= weights_in[0]*x1_train+weights_in[1]*x2_train+bias_in

            hidden_layer_output=ReLU(hidden_layer_input)

            predicted_output=np.dot(hidden_layer_output,weights_out)

            # #backward propogation

            Error=(predicted_output-y_train)


            delta_w2=hidden_layer_output*Error
            delta_w1_1=weights_out*Error * der_ReLU(hidden_layer_input)*x1_train
            delta_w1_2=weights_out*Error * der_ReLU(hidden_layer_input)*x2_train

            delta_b1=weights_out*Error * der_ReLU(hidden_layer_input)

            delta_w1=np.zeros((2,hidden_layer))
            delta_w1[0]=delta_w1_1
            delta_w1[1]=delta_w1_2

            weights_out-=delta_w2*learning_rate
            weights_in-=delta_w1*learning_rate
            bias_in-=delta_b1*learning_rate


    return weights_out,weights_in,bias_in
```

```python
weights_out,weights_in,bias_in=Sine_train(train_data)
```

```python
data_plot = np.array([x2[int(0.8*np.size(x1)):np.size(x1)],x2_normalize[int(0.8*np.s
data_plot_sort = data_plot[:, data_plot[0].argsort()]

test_data=np.array([data_plot_sort[1,:],np.zeros(2000)])

x1_test=test_data[0,:]
x2_test=test_data[1,:]
p=np.size(x1_test)
sin_value_vector=np.zeros(p)
f_test=np.zeros(p)
for i in range(p):

    x1_test=test_data[0,i]
    x2_test=test_data[1,i]

    hidden_layer_input= weights_in[0]*x1_test+weights_in[1]*x2_test+bias_in

    hidden_layer_output=ReLU(hidden_layer_input)

    sin_value_vector[i]=np.dot(hidden_layer_output,weights_out)
    f_test[i] = np.sin(data_plot_sort[0,i])

sin_value_vector
```

```
plt.plot(data_plot_sort[0,:],f_test)
plt.plot(data_plot_sort[0,:],-sin_value_vector)
plt.legend(["sin(x)", "Model Prediction"])
plt.show()
```





For running this part please have an accurate look on related directory and There you can easily find all of you need and also I would appreciate it if you would consider them ☺

4 ACKNOWLEDGEMENT

I am really grateful for Mr MohammadReza Tavakoli (810197477) because in some part of this project we had a nice and effective discussion which highly helped us to have a better analyse and also have more adequate results! (Note that we only talked about ideas behind different problems.)

Afterwards, I am thankful to all of course teaching assistants: MahsaMassoud(mahsamassoud@gmail.com) and Fateme Jafarian (jafarian.fateme7899@gmail.com) and Amirhossein Rokni (a.rokni@ut.ac.ir) who designed this project with high quality.

5 REFERENCES

[1]https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-cifar-10-photo-classification/

[2] https://afteracademy.com/blog/what-are-l1-and-l2-loss-functions

[3] https://machinelearningmastery.com/gradient-descent-with-momentum-from-scratch/