

CSE : 343 Machine Learning

Assignment 3

Arka Sarkar , 2018222

Question 3

1. Hidden Units

- a. Hidden units = [5 ,20, 50 , 100 , 200]

Model for n = 5

Progress : 100% | 100/100 [12:44<00:00, 7.64s/it]

Average Training Loss : 0.6642457783553336
Average Validation Loss : 0.7875496702061758
Model for n = 20

Progress : 100% | 100/100 [14:35<00:00, 8.75s/it]

Average Training Loss : 0.29244775255521127
Average Validation Loss : 0.5615422285927667
Model for n = 50

Progress : 100% | 100/100 [12:05<00:00, 7.26s/it]

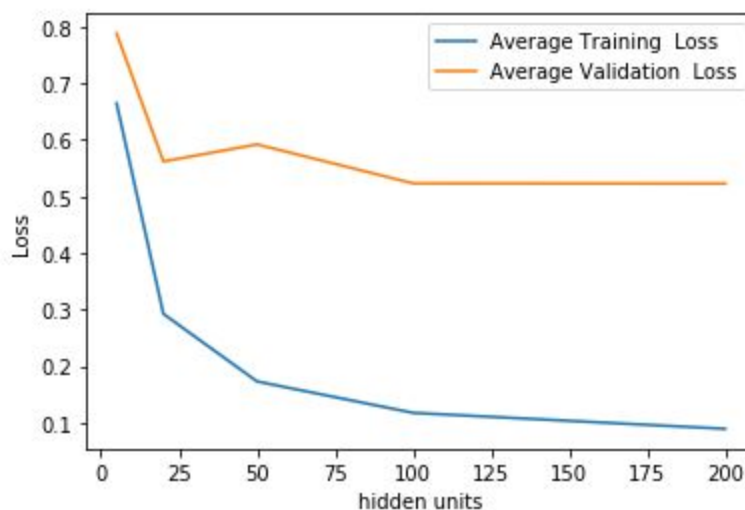
Average Training Loss : 0.17313576152341223
Average Validation Loss : 0.591699907928705
Model for n = 100

Progress : 100% | 100/100 [14:01<00:00, 8.42s/it]

Average Training Loss : 0.11769026785881985
Average Validation Loss : 0.522913781205813
Model for n = 200

Progress : 100% | 100/100 [12:47<00:00, 7.67s/it]

Average Training Loss : 0.08948867799921167
Average Validation Loss : 0.5225038929449187



- b. Observation :

- i. It is clearly evident that as the number of hidden units increased the training loss decreased substantially consequently overfitting the data.

- ii. The validation loss however kept a steady flow (more or less), introducing a high variance to our model → overfitting.

2. Learning Rates

Model for $l = 0.1$

Progress : 100% | 100/100 [10:26<00:00, 6.27s/it]

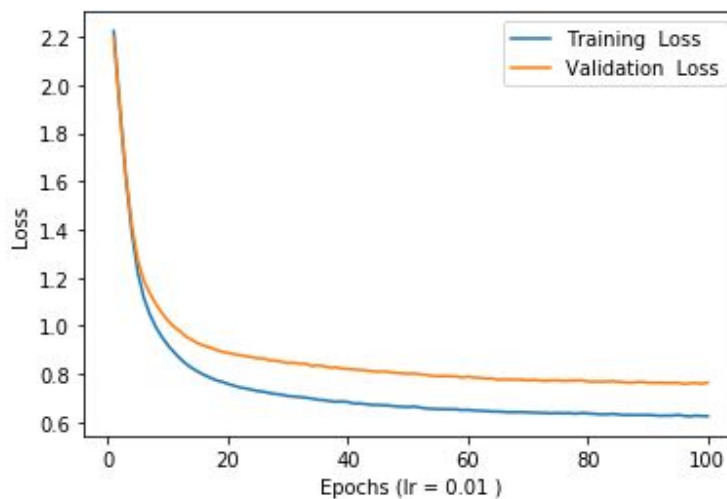
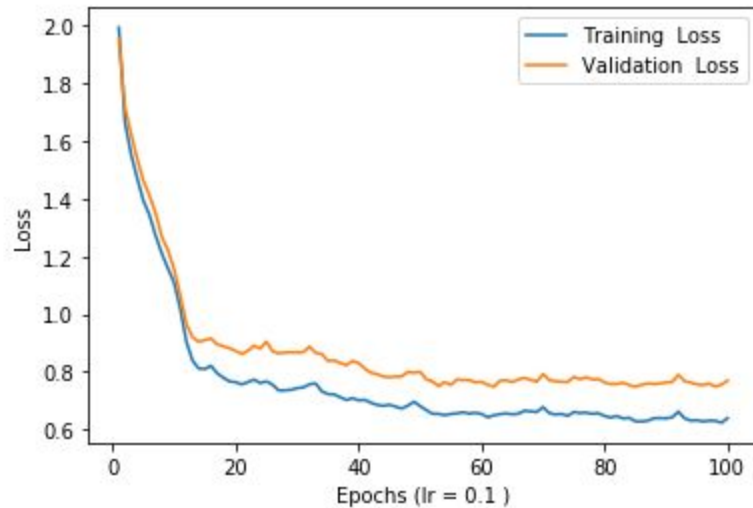
Average Training Loss : 0.7658797559473253
Average Validation Loss : 0.873876634173923
Model for $l = 0.01$

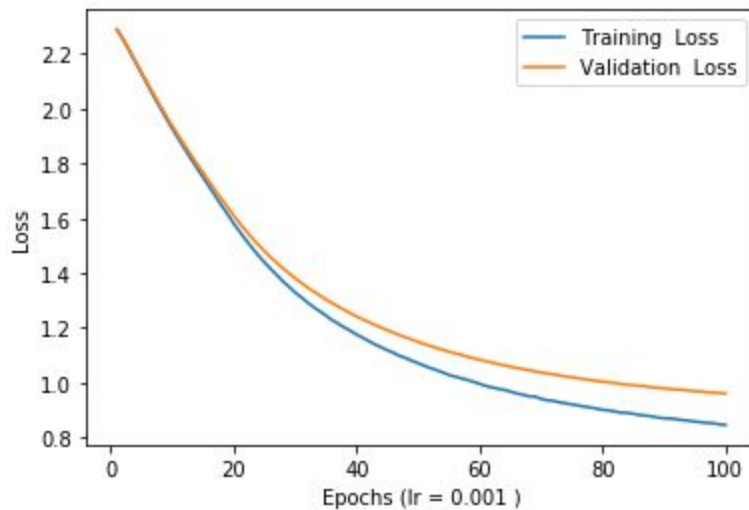
Progress : 100% | 100/100 [10:20<00:00, 6.21s/it]

Average Training Loss : 0.7438545272747673
Average Validation Loss : 0.8700967090990807
Model for $l = 0.001$

Progress : 100% | 100/100 [09:51<00:00, 5.92s/it]

Average Training Loss : 1.2285192669100238
Average Validation Loss : 1.2975009656283591





Learning Rate = 0.001

Observations :

1. $Lr = 0.1$
 - a. The model trained good enough but the losses were a little unstable which indicates that the gradient was fluctuating around the minimum.
 - b. The learning was therefore not the best of the bunch.
2. $Lr = 0.01$
 - a. This model trained perfectly showing a smooth loss vs epoch curve for both training and validation sets.
 - b. The learning curve was best of the bunch.
3. $Lr = 0.001$
 - a. The training wasn't sufficient for the 100 epochs as it can be clearly seen that the loss didn't converge.
 - b. Since the loss didn't converge, it would be confident to say that the learning rate was the inferior of the bunch.

The effect of changing learning rates showed a vast difference in convergence of the models, $Lr = 0.01$ showed greater convergence whereas $Lr = 0.1$ showed unstable convergence and $Lr = 0.001$ didn't converge at all to our liking.

Question 2

1. architecture [#input, 256, 128, 64, #output], learning rate=0.1, and number of epochs=100.

The test accuracies obtained on Relu, Sigmoid, Linear and tanh are :

```
.....  
  
In [43]: relu_nn = pickle.load(open("models/relu", "rb"))  
sigmoid_nn = pickle.load(open("models/sigmoid", "rb"))  
linear_nn = pickle.load(open("models/linear2", "rb"))  
tanh_nn = pickle.load(open("models/tanh", "rb"))
```

```
In [44]: relu_nn.score(X_test, y_test)|
```

```
Out[44]: 0.9388
```

```
In [45]: sigmoid_nn.score(X_test, y_test)
```

```
Out[45]: 0.9036
```

```
In [46]: linear_nn.score(X_test, y_test)
```

```
Out[46]: 0.8163
```

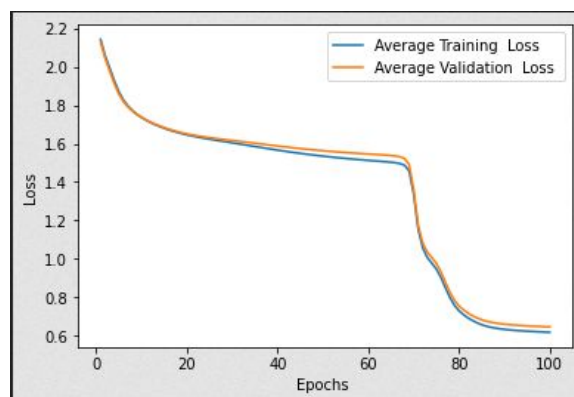
```
In [47]: tanh_nn.score(X_test, y_test)
```

```
Out[47]: 0.91
```

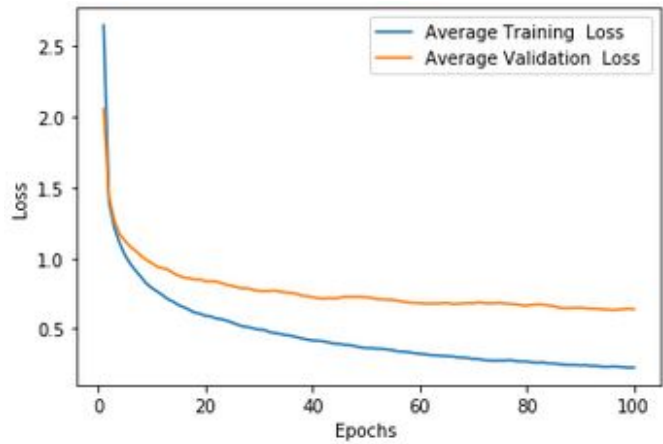
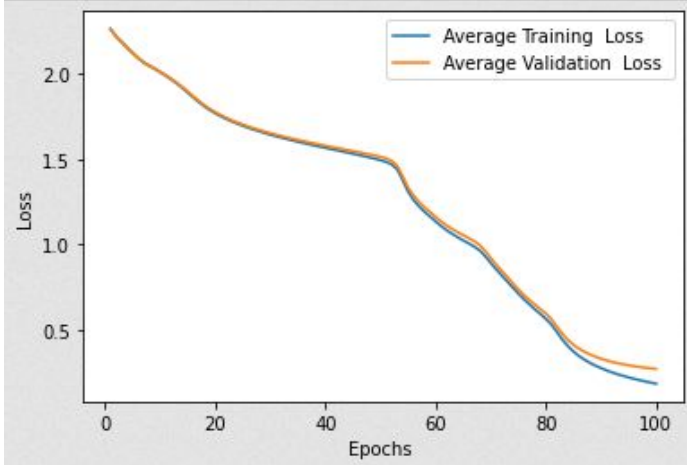
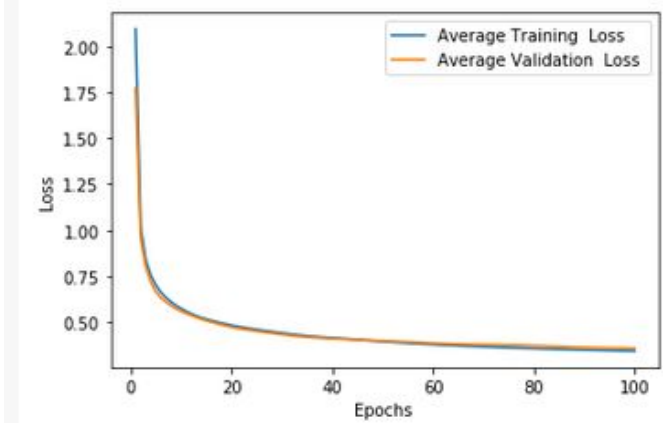
The models are saved using pickle then imported to get the above results.

The best model was using the ReLU activation function.

2. Training error vs epoch curve and testing error vs epoch curve for ReLU, sigmoid, linear and tanh activation function.



Linear Activation

	Tanh Activation
	ReLU Activation
	Sigmoid Activation

3. In every Case the activation function of the output layer would be softmax, the reason behind this is we are dealing with a multi classification problem which requires probabilities for each class at the output layer. Since, the softmax function gives us probabilities for each class we use that only for the output layer activation.

```
exp = np.exp(X)
return exp/(np.sum(exp,axis = 1, keepdims = True))
```

4. Total number of layers = 4
Number of hidden layers = 3

Hidden_layer1 → Activation1 → Hidden_layer2 → Activation2 → Hidden_layer3 → Activation3 →
Output_Layer → Softmax → Loss.

5. Visualise the features of the final hidden layer by creating tSNE plots for the model with highest test accuracy.

```
In [18]: W4 = relu_nn.parameters["W4"]

In [48]: A_1, act, _ = relu_nn.forward_prop(X_train, relu_nn.parameters)

In [50]: A3 = act["A3"]

In [51]: A3.shape

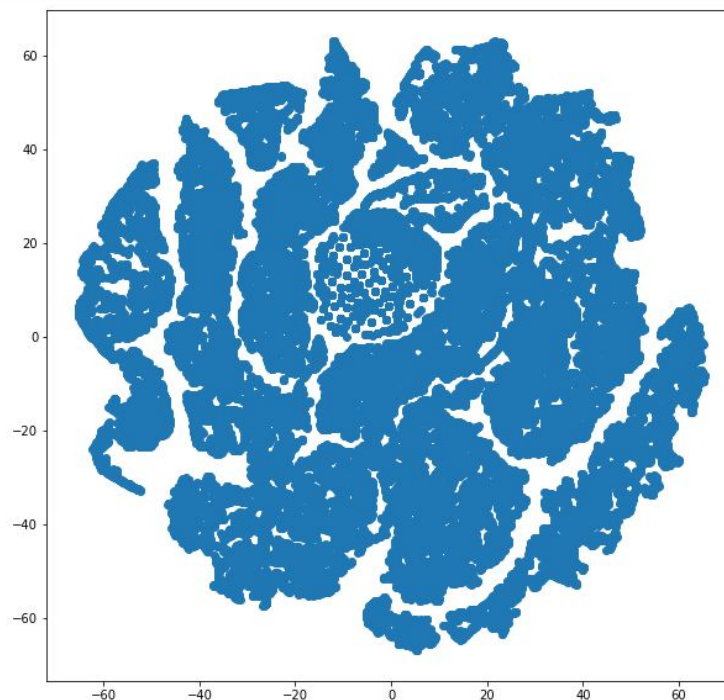
Out[51]: (60000, 64)

In [*]: from sklearn.manifold import TSNE
tsne_em = TSNE(n_components=2, perplexity=30.0, n_iter=1000, verbose=1).fit_transform(A3)

[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 60000 samples in 2.628s...
[t-SNE] Computed neighbors for 60000 samples in 24.351s...
[t-SNE] Computed conditional probabilities for sample 1000 / 60000
```

Relu provided the best accuracy (i.e. **93.88**)

The tSNE plot that achieved was :



6. Running Sklearn on the same dataset with the same parameters fetched the following results :

```
In [39]: s1 = clf.score(X_test,y_test)|
print("For Relu Activation : ", s1)

For Relu Activation : 0.9767

In [40]: s2 = clf2.score(X_test,y_test)
print("For Sigmoid/Logistic Activation : ", s2)

For Sigmoid/Logistic Activation : 0.9644

In [41]: s3 = clf3.score(X_test,y_test)
print("For tanh Activation : ", s3)

For tanh Activation : 0.9667

In [42]: s4 = clf.score(X_test,y_test)
print("For Linear/Identity Activation : ", s4)

For Linear/Identity Activation : 0.9767
```

We observed that sklearn gave better results for every case.
For ReLU, Sigmoid and tanh we got similar results.
But Linear for the scratch implementation gave much poorer results as compared to sklearn.

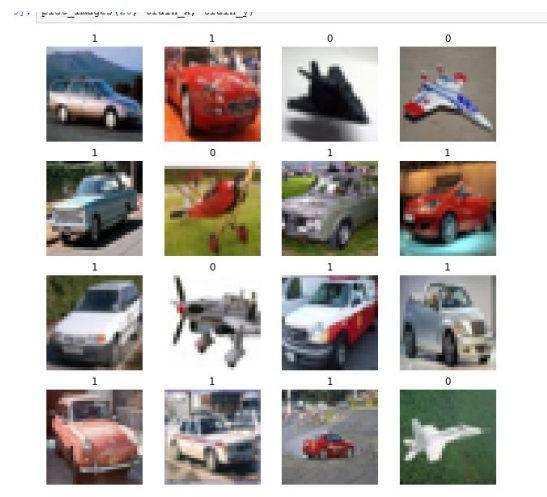
Question 4

1. EDA on CIFAR - 10

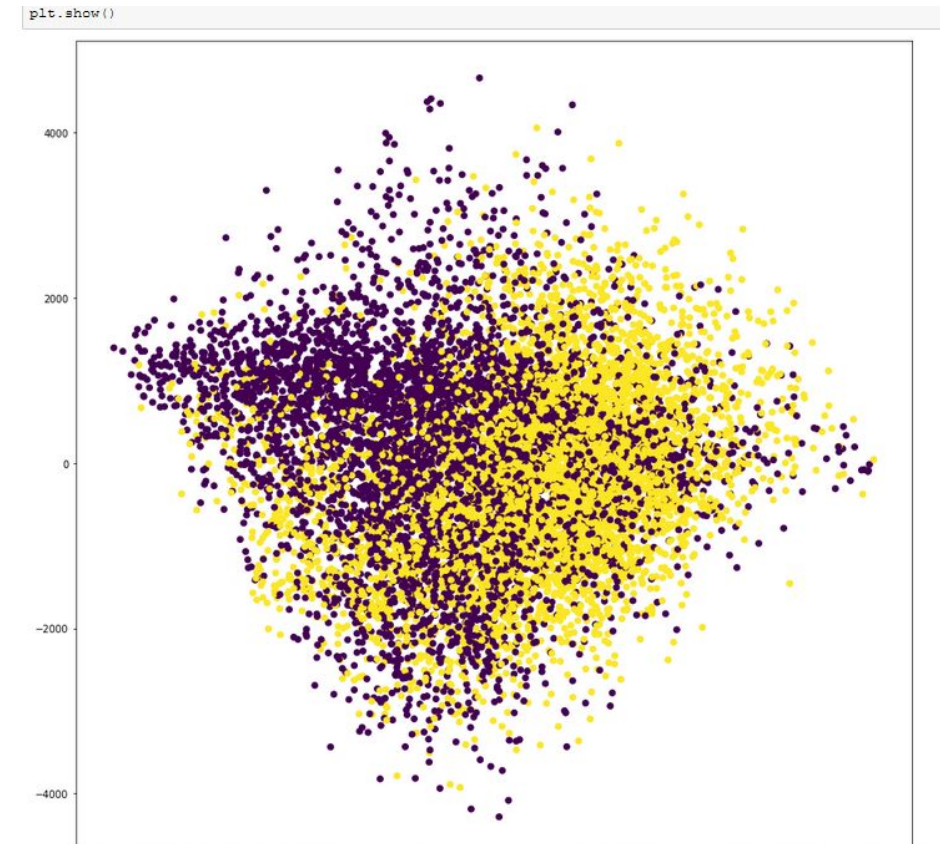
a. Visualizing the dataset

The dataset size is (n,3072), the red, green and blue channels are flattened out to a single array, on resizing the array to (32,32,3) we see the actual image.

The first 16 samples are :



- b. To look at the distribution of the dataset, I reduced the dimension of the array using PCA and visualised in 2d, fetching the following plot:



This tells us that the data is very well distributed.

- c. Calculating the class distributions we get :

```
In [60]: def calculate_class_distribution(labels):  
         n = len(labels)  
         classes = np.unique(labels)  
         for i in classes:  
             curr = np.count_nonzero(labels == i)  
             print("Class frequency of ", i , " is ", str(curr), "/", str(n))
```

```
In [61]: calculate_class_distribution(np.asarray(train_y))
```

```
Class frequency of  0  is  5000 / 10000  
Class frequency of  1  is  5000 / 10000
```

Class 1 = 5000/10000 → 50%

Class 0 = 5000/10000 → 50%

2. Using the existing AlexNet Model from PyTorch (pretrained on ImageNet) as a feature extractor for the images in the CIFAR subset

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

alexnet.to(device)

AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

```
train_extracted_x = []
train_extracted_y = []
for (x, y) in tqdm(trainloader, desc = "Progress : ", leave = True, position = 0 , total = len(trainloader)) :

    x = x.to(device)
    y = y.to(device)
    y_pred= alexnet(x.float())

    train_extracted_x.append(y_pred)
    train_extracted_y.append(y)
    del x
    del y

Progress : 100%|██████████| 10000/10000 [00:23<00:00, 432.79it/s]

# train_extracted_x = np.asarray(train_extracted_x)
# print(train_extracted_x.shape)

test_extracted_x = []
test_extracted_y = []
for (x, y) in tqdm(testloader, desc = "Progress : ", leave = True, position = 0 , total = len(testloader)) :

    x = x.to(device)
    y = y.to(device)
    y_pred= alexnet(x.float())

    test_extracted_x.append(y_pred)
    test_extracted_y.append(y)
    del x
    del y

Progress : 100%|██████████| 10000/10000 [00:23<00:00, 431.06it/s]
```

3. Since extracting features from alexnet takes up a lot of RAM, the code was ran on google colab and then the extracted features were saved to a pickle file.
Afterwards clearing the RAM cache, the network was further trained on the given inputs.

```
Epoch 100/100
313/313 [=====] - ETA: 0s - loss: 5.2154e-08 - accuracy: 0.59 - ETA: 1s - loss: 6.0846e-08 - acc
uracy: 0.64 - ETA: 1s - loss: 6.1095e-08 - accuracy: 0.63 - ETA: 1s - loss: 6.0536e-08 - accuracy: 0.63 - ETA: 1s - loss:
5.8957e-08 - accuracy: 0.63 - ETA: 1s - loss: 6.0054e-08 - accuracy: 0.63 - ETA: 1s - loss: 5.9764e-08 - accuracy: 0.63 -
ETA: 1s - loss: 5.9066e-08 - accuracy: 0.63 - ETA: 0s - loss: 5.9134e-08 - accuracy: 0.64 - ETA: 0s - loss: 5.8824e-08 -
accuracy: 0.64 - ETA: 0s - loss: 5.8840e-08 - accuracy: 0.64 - ETA: 0s - loss: 5.8507e-08 - accuracy: 0.63 - ETA: 0s - l
oss: 5.9092e-08 - accuracy: 0.64 - ETA: 0s - loss: 5.9456e-08 - accuracy: 0.63 - ETA: 0s - loss: 5.9352e-08 - accuracy: 0.
63 - ETA: 0s - loss: 5.9498e-08 - accuracy: 0.64 - ETA: 0s - loss: 5.9645e-08 - accuracy: 0.64 - ETA: 0s - loss: 5.9773e-
08 - accuracy: 0.63 - ETA: 0s - loss: 5.9940e-08 - accuracy: 0.63 - ETA: 0s - loss: 5.9804e-08 - accuracy: 0.63 - ETA: 0s
- loss: 5.9699e-08 - accuracy: 0.63 - ETA: 0s - loss: 5.9635e-08 - accuracy: 0.63 - ETA: 0s - loss: 5.9633e-08 - accurac
y: 0.63 - ETA: 0s - loss: 5.9550e-08 - accuracy: 0.64 - ETA: 0s - loss: 5.9592e-08 - accuracy: 0.64 - ETA: 0s - loss: 5.9
454e-08 - accuracy: 0.63 - ETA: 0s - loss: 5.9593e-08 - accuracy: 0.63 - 1s 4ms/step - loss: 5.9605e-08 - accuracy: 0.63f
0
```

The accuracy received after 100 epochs is 63%. On the training set.

4. The results on the test set are:

```
In [148]: y_pred = model.predict(new_test_X)
```

```
In [156]: from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
```

```
In [153]: y_pred[y_pred>=0.5] = 1
y_pred[y_pred<0.5] = 0
print(y_pred)
```

```
[[0.]
 [0.]
 [1.]
 ...
 [0.]
 [1.]
 [1.]]
```

```
In [154]: cm = confusion_matrix(y_pred, new_test_y)
print(cm)
```

```
[[2419 1071]
 [2581 3929]]
```

```
In [158]: acc = accuracy_score(y_pred, new_test_y)
print(acc)
```

```
0.6348
```

