Q1. 3) Scaling factor 0.01 had a negative impact on running sigmoid on the given parameter but gave a better result on tanh when the output layer activation was sigmoid. I chose scaling factor 0.1 because it had a positive impact on all activations.

4) fit() - also accepts test/validation data.

Predict_proba()- uses softmax on the ouput activations.

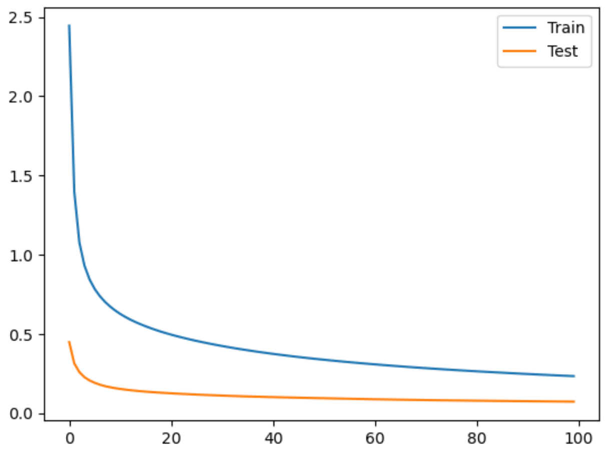
Q2. 1) Test accuracy - Sigmoid - 88% (No scaling)

Tanh - 96% (0.1 scaling)

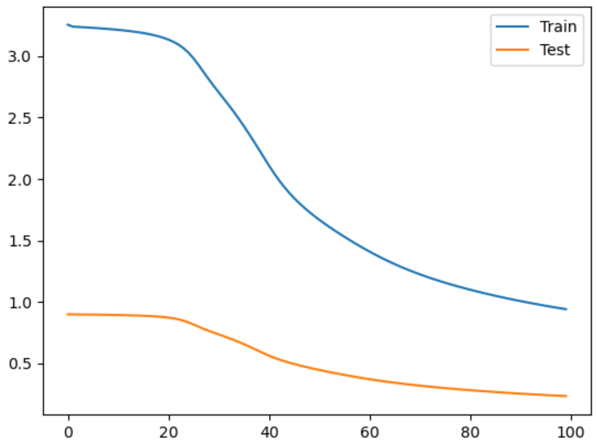
Relu - 96 (0.1 scaling)

Linear - 91% (0.1 scaling)

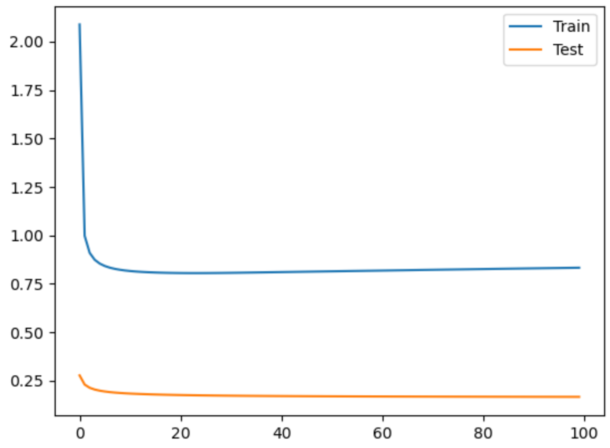
2) Relu



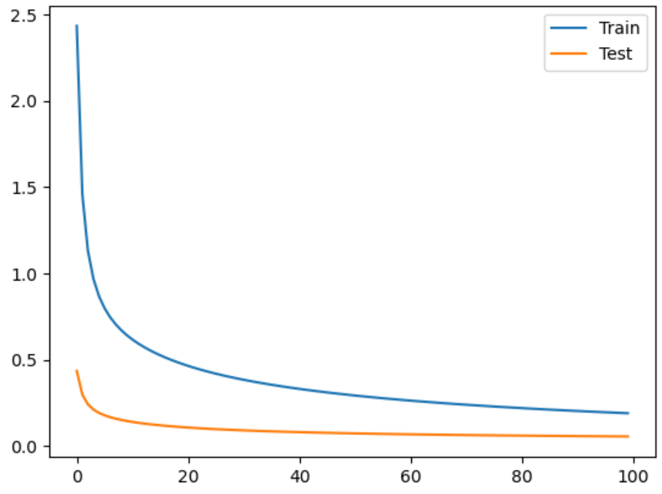
Sigmoid



Linear


Train
Test
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0
20
40
60
80
100

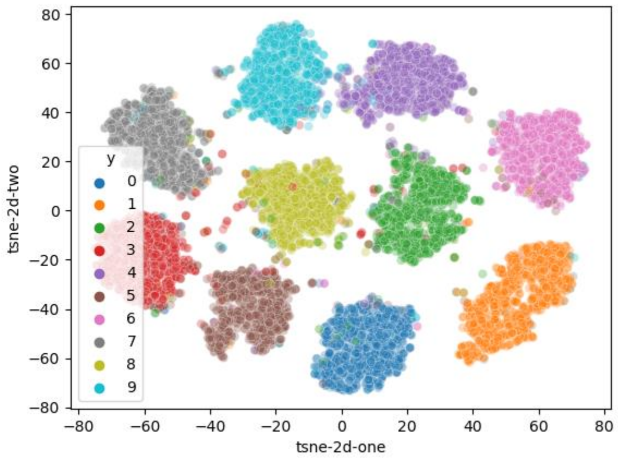
Tanh

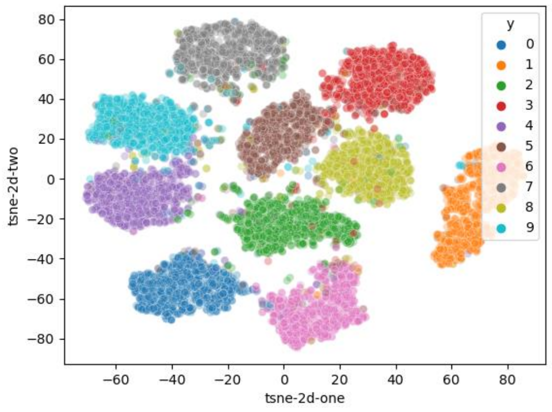

Train
Test
2.5
2.0
1.5
1.0
0.5
0.0
0
20
40
60
80
100

3) Softmax is a better activation function for all because we can also get classwise distribution. For relu and linear, sigmoid was better capable of handling the larger values and gave better accuracy. Sigmoid can be used with itself when we have once hot encoded labels but like in pytorch does not use one hot encoded because on memory inefficiency. Same reasoning for tanh being used with itself but when sigmoid was used as the output layer activation function, it performed than tanh being used.

4) Number of layers = 5 -> 1 input, 3 hidden and 1 output.

5) tanh and relu both had close to 96% accuracy so the plots for both of them respectively.





Both are able to make clear clusters of the labels which is why they are getting such good accuracy.

6) Sklearn - Sigmoid - 0.9707

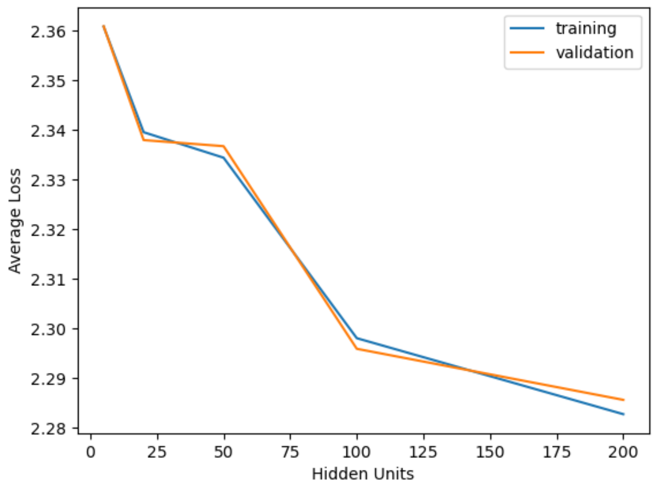
Relu= 0.9804

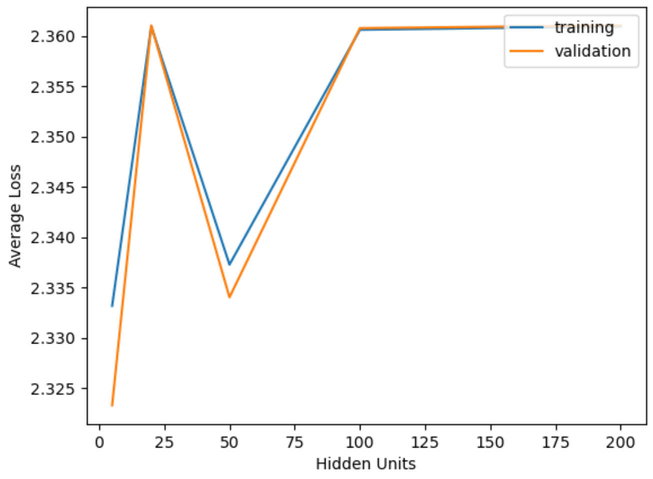
Tanh = 0.9804

Linear = 0.9181

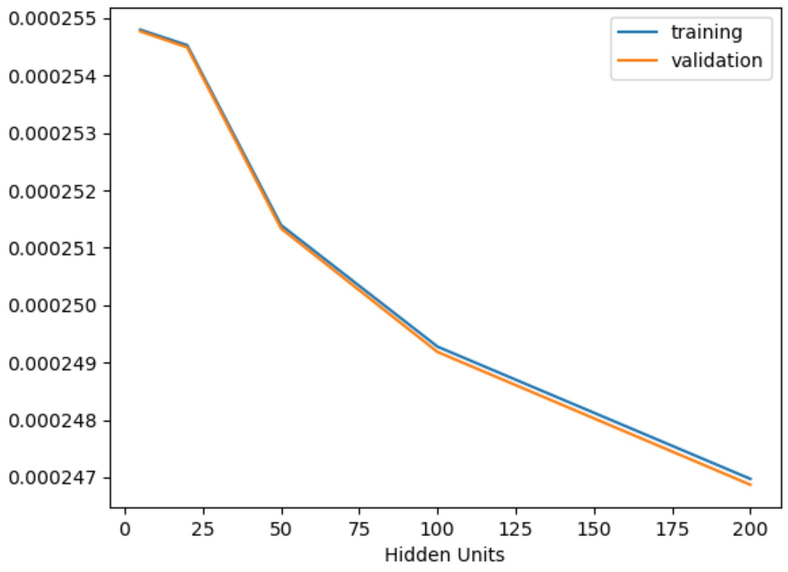
Results are almost similar to that of sklearn. Sklearn uses momentum which helps converge faster. In the sigmoid graph we can see the model has not converged yet and hence sk learn is able to perform better on the same parameters.

Q3. A) Error increases for validation because the weights are randomly initialized using a uniform distribution and with increasing hidden units and constant learning rate, epochs taken to converge will increase which is why there is such a graph on the right. The graph changes quite a bit with every run because of the random initialization which sometimes leads to weights taking a longer time to converge which in this case was for when number of units were 100.



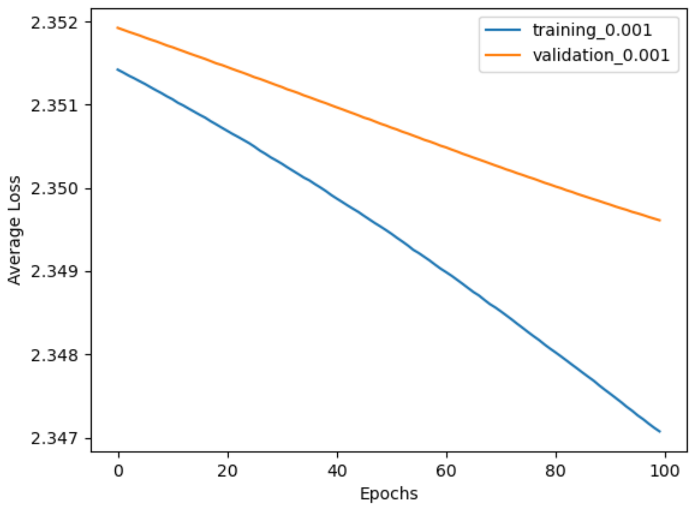


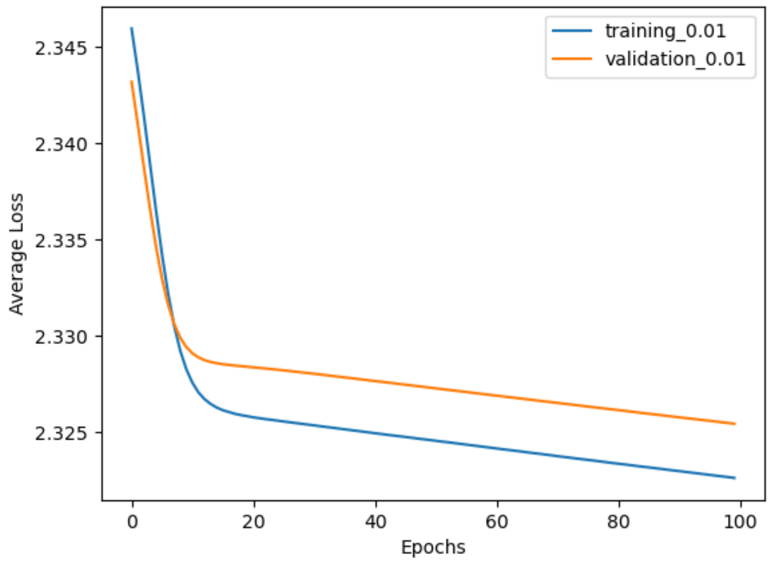
When weights were not initialized explicitly, the graph is much more stable and as expected i.e error goes down with increase in number of units.

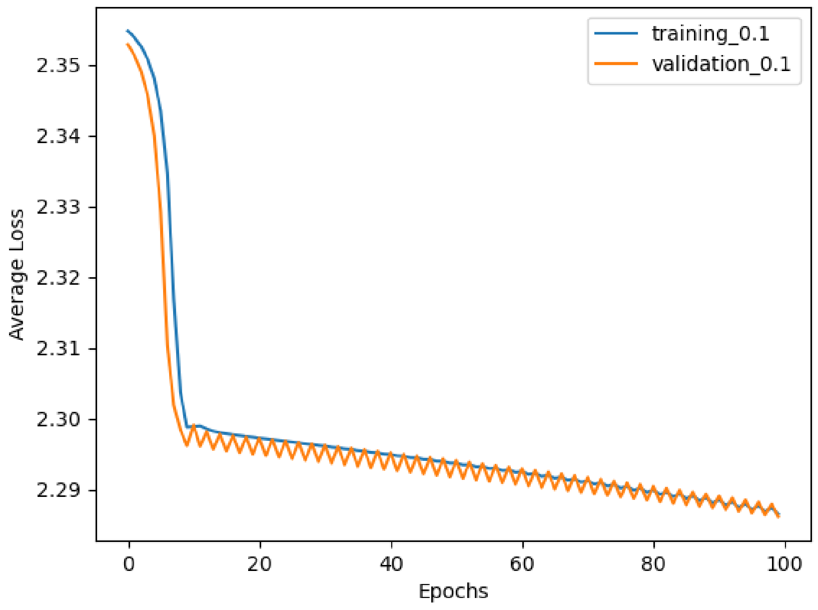


b)

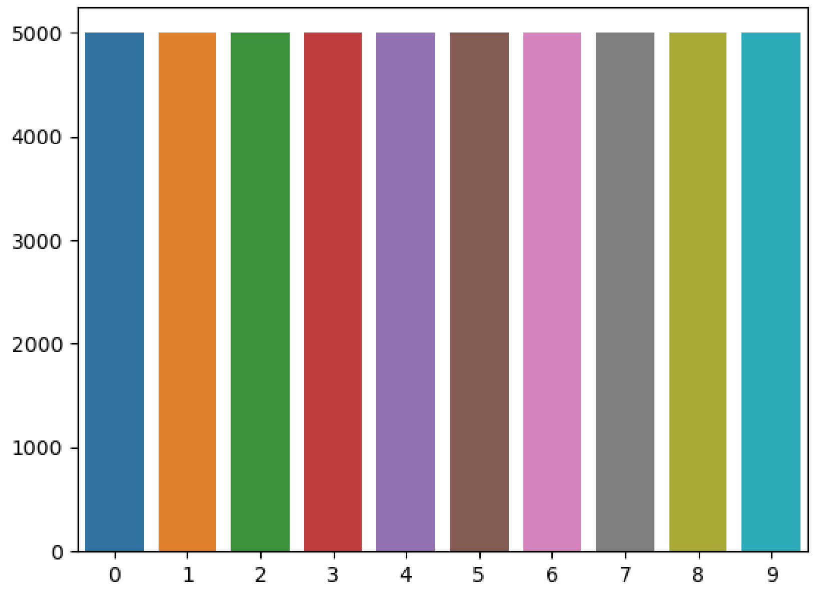
For lower learning rate, we do not converge in 100 epochs while for lr=0.1, we converge very fast but we can see the zig zag behavior of gradient descent. Lr = 0.01 is stable and almost converges within 100 epochs. All 3 graphs show behavior which is expected as we vary learning rates.

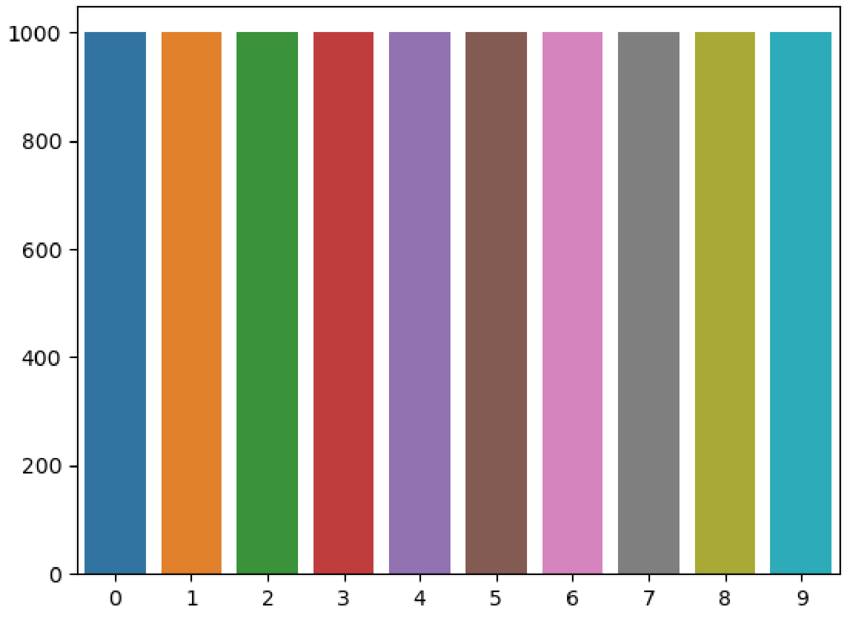

training_0.001
validation_0.001
Average Loss
Epochs
2.352
2.351
2.350
2.349
2.348
2.347
0
20
40
60
80
100


training_0.01
validation_0.01
Average Loss
Epochs
2.345
2.340
2.335
2.330
2.325
0
20
40
60
80
100


training_0.1
validation_0.1
Average Loss
Epochs
2.35
2.34
2.33
2.32
2.31
2.30
2.29
0
20
40
60
80
100

Q4. Plots below are of train and test set respectively and they both have 10 classes with equal amounts of samples in them.





The train set has 50000 images and each image is of 32X32 size. The images were normalized. Original Alexnet was not trained on 32X32 images so the images had to be resized.

The pretrained alexnet model was loaded and output features were set to 1000 as the question instructed.

Model was defined as below

```
model = torch.nn.Sequential(
    torch.nn.Linear(1000, 512),
    torch.nn.ReLU(),
    torch.nn.Linear(512, 256),
    torch.nn.ReLU(),
    torch.nn.Linear(256, 10),
    torch.nn.Softmax(0)
)
```

This model gave the following results.

Confusion Matrix accuracy = 0.6527%

```
[[648.,  52.,  63.,  15.,  28.,   2.,   4.,  40.,  85.,  60.],

 [ 51., 656.,  24.,  25.,   5.,  31.,  24.,  17.,  45.,  94.],
 [ 48.,   2., 500.,  45.,  45.,  54.,  40.,  36.,   2.,   0.],
 [ 31.,  25.,  90., 627.,  59., 186.,  99.,  58.,  33.,  11.],
 [ 18.,   1., 126.,  65., 712.,  59.,  62., 145.,   8.,   7.],
 [  0.,   5.,  27.,  86.,  16., 555.,  20.,  54.,   3.,   3.],
 [ 11.,  34., 103.,  66.,  54.,  29., 741.,   8.,  11.,   4.],
 [ 14.,  13.,  38.,  26.,  70.,  60.,   3., 574.,   1.,  10.],
 [143.,  53.,  26.,  19.,   7.,  15.,   6.,   4., 764.,  61.],
 [ 36., 159.,   3.,  26.,   4.,   9.,   1.,  64.,  48., 750.]]
```

I tried to use sigmoid but it gave poor results. Also, I tried leaky relu and the model was defined like

```
model = torch.nn.Sequential(
    torch.nn.Linear(1000, 512),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(512, 256),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(256, 10),
    torch.nn.Softmax(0)
)
```

The accuracy for this model was 0.6474%

```
[[574.,  33.,  69.,  12.,  23.,   0.,   3.,  30.,  64.,  38.],
 [ 75., 699.,  32.,  32.,   5.,  31.,  30.,  14.,  67., 102.],
 [ 43.,   0., 478.,  52.,  46.,  43.,  48.,  30.,   1.,   0.],
 [ 31.,  29.,  96., 645.,  64., 200.,  92.,  65.,  29.,  13.],
 [ 13.,   2., 134.,  56., 676.,  58.,  73., 117.,   4.,   5.],
 [  3.,   6.,  38.,  79.,  22., 568.,  15.,  66.,   4.,   7.],
 [  9.,  25.,  85.,  63.,  51.,  26., 728.,   6.,  16.,   3.],
 [ 28.,  20.,  42.,  27.,  90.,  55.,   3., 610.,   4.,  25.],
 [192.,  51.,  25.,  20.,  20.,  10.,   7.,  10., 778.,  89.],
 [ 32., 135.,   1.,  14.,   3.,   9.,   1.,  52.,  33., 718.]]
```

Hyperparameter optimizing might lead to better results in the case of leakyrelu which I believe should generally perform better than standard relu but I did not try it because of time constraints.