INTRODUCTION TO MACHINE LEARNING

Assignment 2-2021AMA2090(Rahman Alam)

Contents

`

NON-AIP701-PART 1_A

1. Pytorch library

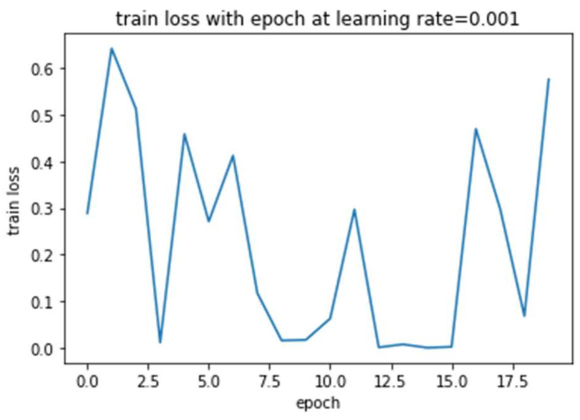
I will be using Pytorch library as I am familiar with this library work working with the neural network.

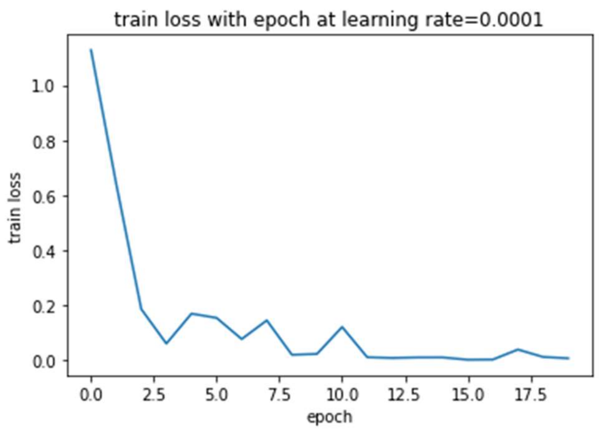
```
import torch
import torch.nn as nn
from torch.nn import ReLU
import torch.utils.data as td
from torch.utils.data import DataLoader
import torch.nn.functional as F
```

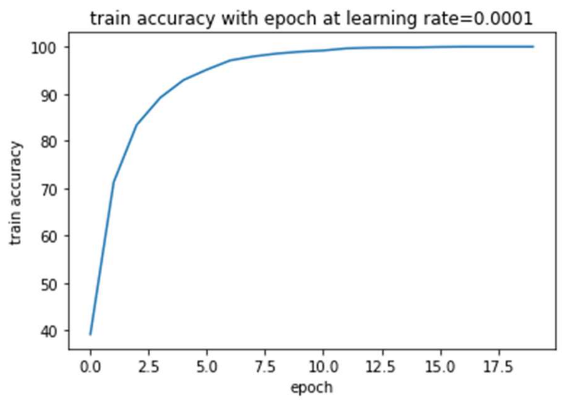
The module used here will be as shown above.The initial basic neural network I have shown below.

```
class MLP(nn.Module):
        def __init__(self,input_size=784,output_size=10,layers=[64,32]):
                super().__init__()
                self.x1=nn.Linear(input_size,layers[0])
                self.x2=nn.Linear(layers[0],layers[1])
                self.x3=nn.Linear(layers[1],output_size)

        def forward(self,X):
                X=F.relu(self.x1(X))
                X=F.relu(self.x2(X))
                X= self.x3(X)
                return F.log_softmax(X,dim=1)

model=MLP()
print(model)
```

In basic neural network just varying the learning rate is also effecting the test accuracy.







```
Test loss: 1.0687636137008667 Test Accuracy: 86.0 (at LR=0.0001)

Test loss: 1.8031316995620728 Test Accuracy: 89.60000610351562 (at LR=0.001)
```

So we can say that at a lower learning rate the model is learning smoothly but slower if we give more epochs with a lower learning rate it may give a better result. The higher learning rate is jumping here and there. So in next section we will see hyperparameter tuning as now we are familiar with the library.

2. Cross Validation and Hyperparameter Tuning

Hyperparameter are tuned as learning rate, Number of neurons and batch size.

Epoch=15 for all the cases

Learning rate chosen as [0.001,0.0001]

Number of neurons in 2 layers chosen as [128,64] ,[64,32]

Batch size taken as [10,20]

After training for 20 epochs, I have used K fold stratified cross validation as we need to balance the class also in the dataset. Random state is also chosen to reproduce the same results.

```
kf = StratifiedKFold(5, shuffle=True, random_state=42)
```

The data set I have divided in 2000 and 1000 samples. The model trained on 2000 samples and K fold is run at 1000 samples. And results were same for both 2000 and 1000 samples(unseen).

Fold #1

Fold score (accuracy): 99.5

Fold #2

Fold score (accuracy): 99.75

Fold #3

Fold score (accuracy): 99.25

Fold #4

Fold score (accuracy): 99.5

Fold #5

Fold score (accuracy): 99.0

Final score (accuracy): 99.4

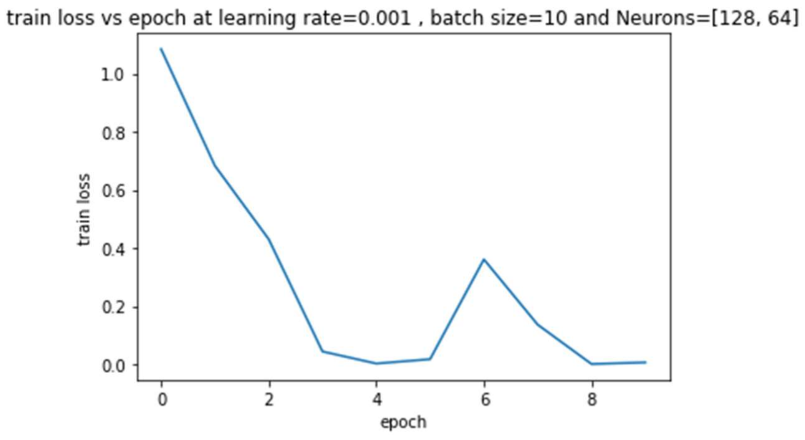
Predicted

Actual	0	1	2	3	4	5	6	7	8	9
0	41	0	0	0	0	0	0	0	0	0
1	0	41	0	0	0	0	0	0	0	0
2	0	0	37	0	0	0	0	0	0	0
3	0	0	0	39	0	0	0	0	1	0
4	0	0	0	0	41	0	0	0	0	0
5	0	0	0	0	0	43	0	0	0	0
6	0	0	0	0	0	0	41	0	0	0
7	0	0	0	0	0	0	0	37	0	1
8	0	1	0	0	0	0	0	0	39	0
9	0	0	0	0	0	0	0	1	0	37

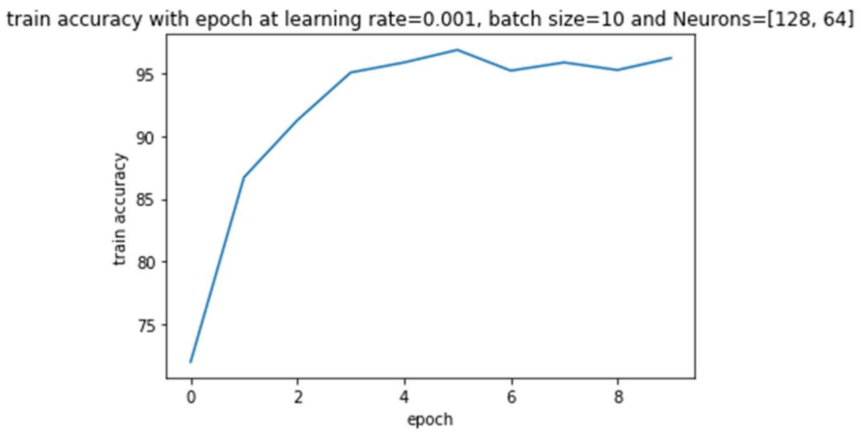
3. Batch size =10

Results for [LR, Number of neurons, Batch size] at [0.001,[128,64],10]

- Graph for Train loss vs epoch



- Graph for Train accuracy vs epoch



- Train, Test accuracy and Time taken

  epoch: 9 batch: 200 Train loss: 0.006110057234764099 Train accuracy: 96.20000457763672

  Test loss: 0.7356 Test accuracy: 89.2 Time taken for training=8.89 sec

- **Confusion Matrix for train and test**
  For Training

	0	1	2	3	4	5	6	7	8	9
0	196	0	2	1	0	4	2	0	0	0
1	0	192	1	1	0	0	0	0	1	0
2	1	0	186	1	0	1	0	1	3	4
3	0	0	0	177	0	6	1	0	7	3
4	0	2	1	0	188	0	3	1	3	13
5	0	0	0	9	3	164	3	0	3	2
6	2	1	1	1	0	2	176	0	1	0
7	1	3	1	1	1	1	0	207	1	9
8	0	1	4	5	1	5	4	1	197	2
9	0	1	0	3	9	1	1	3	2	165

For Testing

	0	1	2	3	4	5	6	7	8	9
0	94	0	2	5	0	1	2	0	0	0
1	0	110	0	1	1	0	1	1	1	3
2	0	0	78	3	0	0	0	5	0	0
3	0	0	4	101	0	3	0	3	1	1
4	1	0	3	0	71	0	1	2	1	2
5	0	0	0	2	0	75	1	0	1	0
6	1	1	1	2	1	2	98	0	0	0
7	0	0	1	0	3	1	0	96	0	4
8	1	2	6	2	4	2	1	0	80	0
9	0	0	0	1	16	1	0	2	1	89

- CV Results

Fold #1

Fold score (accuracy): 97.0

Fold #2

Fold score (accuracy): 98.75

Fold #3

Fold score (accuracy): 96.5

Fold #4

Fold score (accuracy): 98.5

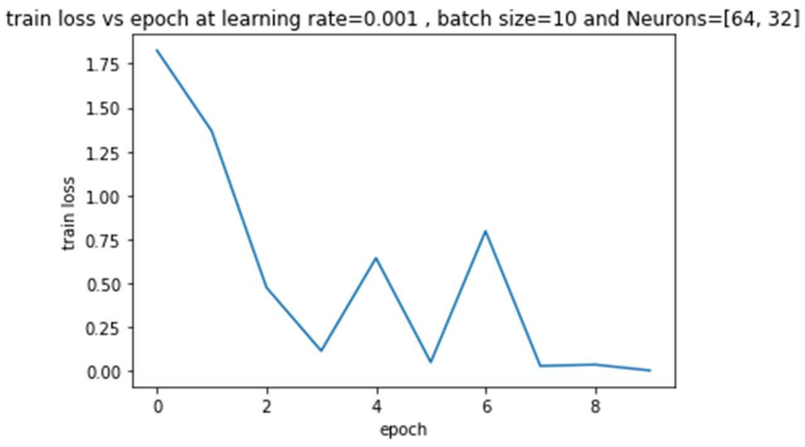
Fold #5

Fold score (accuracy): 97.5

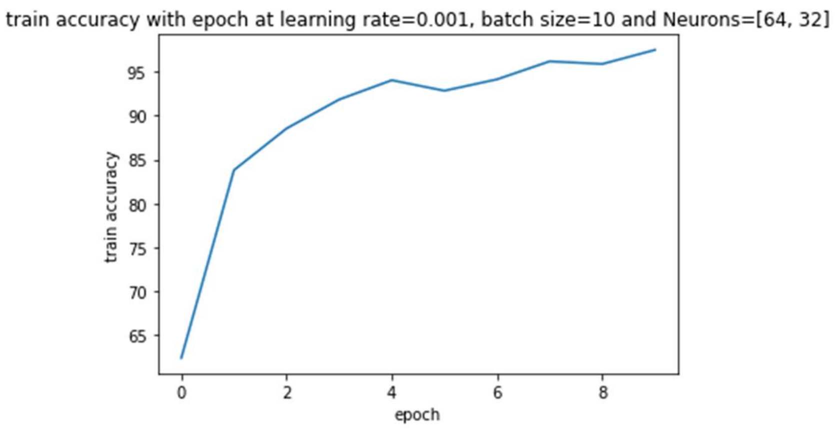
Final score (accuracy): 97.64

Results for [LR , Number of neurons,Batch size] at [0.001,[64,32],10]

- Graph for Train loss vs epoch



- Graph for Train accuracy vs epoch



- Test loss accuracy and Time taken

  epoch: 9 batch: 200 Train loss: 0.00446 Train accuracy: 97.5

  Test loss: 0.636 Test accuracy: 89.0 Time taken for training(sec)=7.82

- **Confusion Matrix for train and test**
  For Training

	0	1	2	3	4	5	6	7	8	9
0	179	1	3	0	0	4	0	1	0	1
1	0	185	1	0	0	2	2	0	8	0
2	2	1	190	4	0	0	6	2	3	2
3	3	1	5	134	0	2	0	0	6	2
4	3	0	0	0	181	1	0	2	1	6
5	3	1	0	5	2	191	2	0	4	3
6	0	1	3	1	2	3	193	1	5	1
7	0	0	3	0	6	1	1	197	0	10
8	2	3	3	6	0	11	4	0	193	2
9	1	0	0	4	9	0	1	6	4	168

For Testing

	0	1	2	3	4	5	6	7	8	9
0	91	0	1	0	0	1	5	0	0	0
1	0	100	0	0	0	0	1	1	1	0
2	1	0	86	3	1	0	0	4	1	0
3	2	2	2	94	0	1	0	1	1	0
4	0	0	1	1	84	0	0	4	0	2
5	1	2	1	10	0	76	1	0	0	1
6	1	0	2	0	2	0	95	0	1	0
7	0	1	1	0	1	0	0	94	0	5
8	1	8	1	6	0	5	2	1	79	0
9	0	0	0	3	8	2	0	4	2	91

- CV Results
  Fold #1

  Fold score (accuracy): 97.25

  Fold #2

  Fold score (accuracy): 99.25

  Fold #3

  Fold score (accuracy): 99.75

  Fold #4

  Fold score (accuracy): 99.25

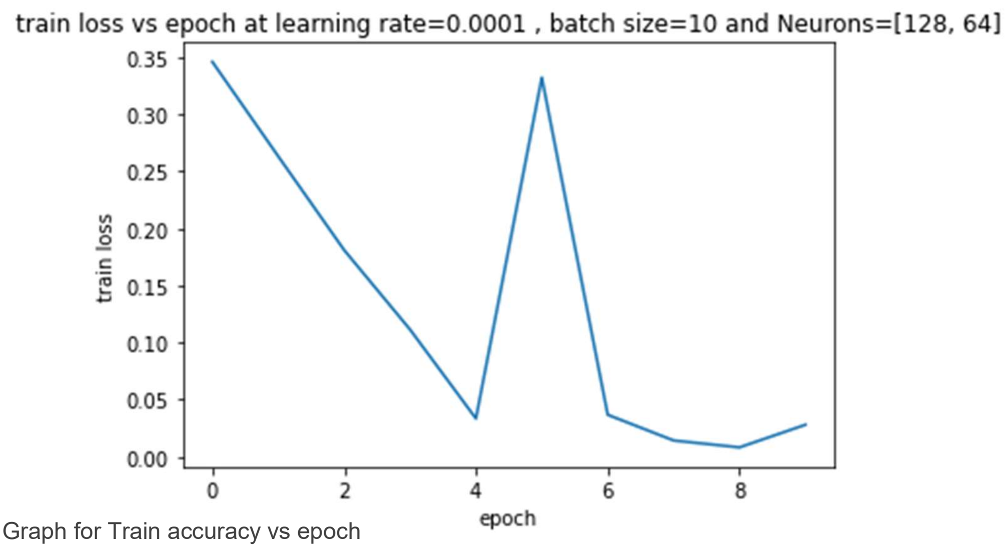
  Fold #5

  Fold score (accuracy): 99.0

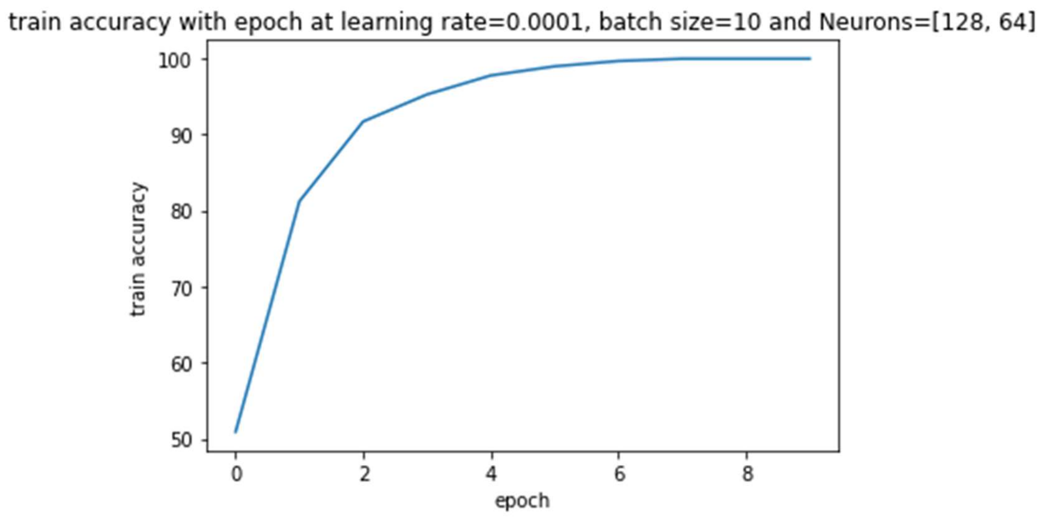
  Final score (accuracy): 98.9

Results for [LR, Number of neurons, Batch size] at [0.0001,[128,64],10]

- Graph for Train loss vs epoch



- Graph for Train accuracy vs epoch



- Test loss accuracy and Time taken

```
epoch: 9 batch:  200 Train loss: 0.007541 Train accuracy: 100.0

Test loss: 0.43914 Test accuracy: 88.8 Time taken for training(sec)=7.76
```

- **Confusion Matrix for train and test**

For Training

	0	1	2	3	4	5	6	7	8	9
0	203	2	0	0	1	2	2	0	2	3
1	0	194	4	0	0	0	0	1	0	1
2	3	2	169	3	2	0	5	3	2	1
3	1	3	0	154	2	4	1	2	5	5
4	3	0	3	1	205	0	1	4	2	4
5	2	3	1	4	0	198	1	0	3	3
6	3	1	0	0	2	1	176	0	1	1
7	1	1	0	1	0	0	0	184	1	5
8	0	5	4	5	0	3	3	2	180	2
9	1	2	1	0	2	4	1	6	3	178

For Testing

	0	1	2	3	4	5	6	7	8	9
0	91	0	1	2	0	3	1	1	0	0
1	0	106	1	1	1	0	0	2	0	3
2	0	1	76	2	3	0	2	2	1	0
3	0	0	0	96	0	1	0	4	2	0
4	0	1	3	0	85	0	3	1	1	3
5	3	1	2	8	0	75	1	0	1	0
6	2	4	2	0	0	2	95	0	0	0
7	0	0	1	1	0	0	0	96	0	6
8	1	0	7	4	1	3	2	0	76	1
9	0	0	2	3	6	1	0	3	4	86

- CV Results

  Fold #1

  Fold score (accuracy): 100.0

  Fold #2

  Fold score (accuracy): 100.0

  Fold #3

  Fold score (accuracy): 100.0

  Fold #4

  Fold score (accuracy): 100.0

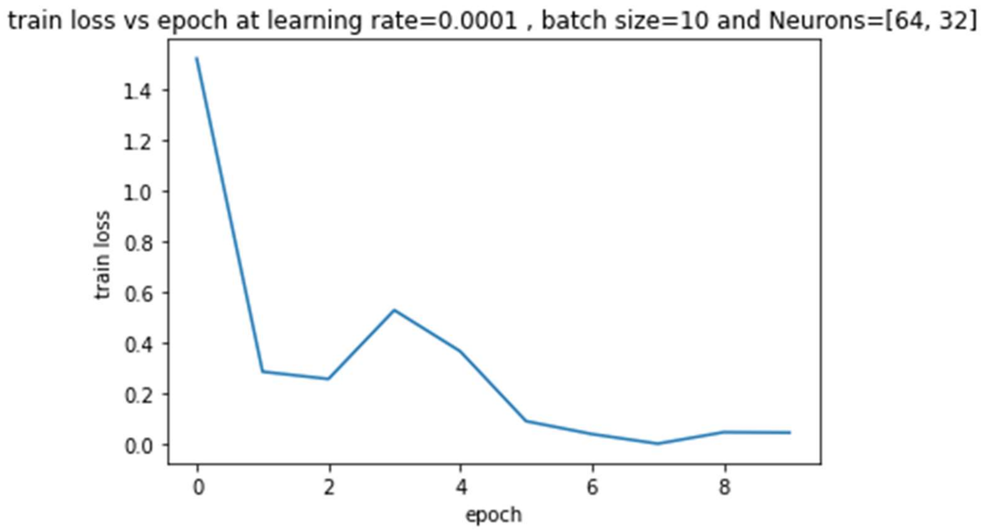
  Fold #5

  Fold score (accuracy): 100.0

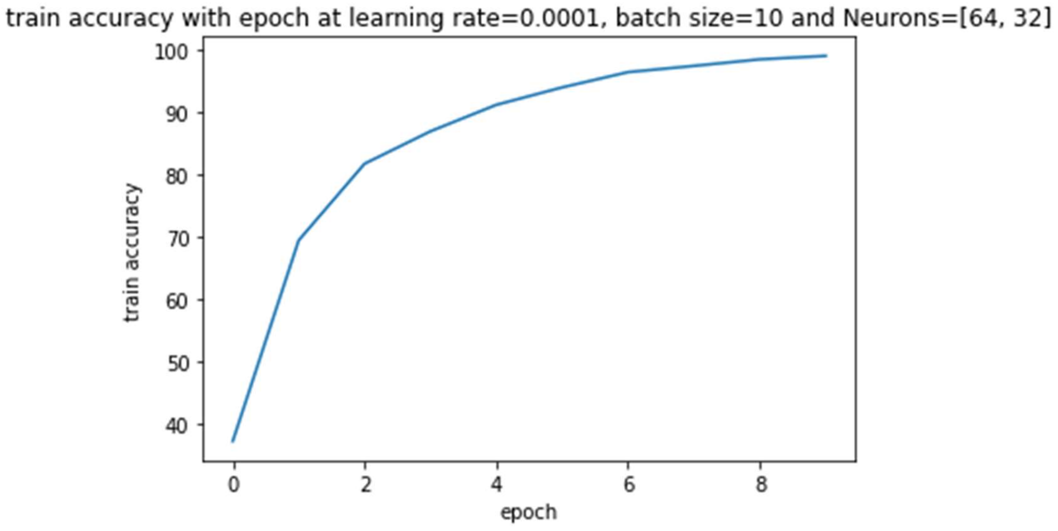
  Final score (accuracy): 100.0

Results for [LR,Number of neurons,Batch size] at [0.0001,[64,32],10]

- Graph for Train loss vs epoch



- Graph for Train accuracy vs epoch



- Test loss accuracy and Time taken

  epoch: 9 batch: 200 Train loss: 0.04702 Train accuracy: 98.949

  Test loss: 0.6613 Test accuracy: 85.3 Time taken for training(sec)=6.63
- **Confusion Matrix for train and test**
  For Training

0	1	2	3	4	5	6	7	8	9	
0	180	0	1	0	6	7	6	0	4	0
1	1	149	1	0	1	2	0	1	3	0
2	2	1	188	5	5	3	3	4	5	1
3	1	2	13	143	0	16	1	1	9	2
4	3	0	3	0	213	2	0	1	3	10
5	5	2	2	5	4	185	1	0	16	3
6	8	1	2	0	4	0	177	3	3	1
7	3	0	4	0	4	1	2	133	2	8
8	3	3	8	7	5	17	3	4	167	1
9	1	2	4	2	14	1	1	6	7	163

For Testing

0	1	2	3	4	5	6	7	8	9	
0	87	0	4	4	1	1	0	1	2	1
1	0	105	1	2	0	1	1	1	3	1
2	2	4	79	4	2	1	2	1	3	1
3	0	1	1	94	0	4	0	2	2	0
4	1	2	2	0	78	1	2	2	1	3
5	3	0	1	10	1	68	1	0	2	1
6	3	0	2	0	1	2	97	1	2	0
7	1	0	2	0	0	1	0	95	1	8
8	0	1	3	3	1	6	1	2	67	1
9	0	0	0	0	12	0	0	4	2	83

- CV Results

  Fold #1

  Fold score (accuracy): 99.5

  Fold #2

  Fold score (accuracy): 99.75

  Fold #3

  Fold score (accuracy): 99.25

  Fold #4

  Fold score (accuracy): 99.25

  Fold #5

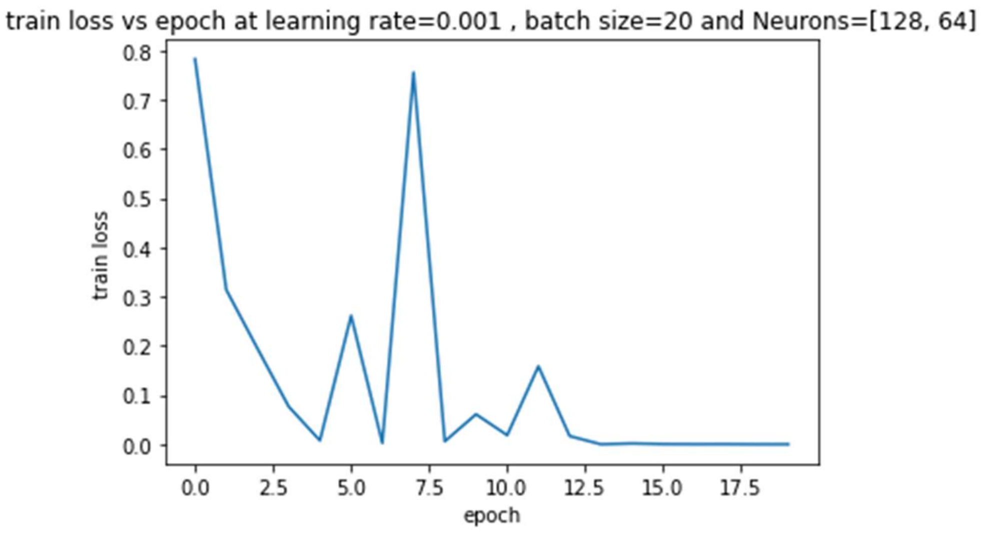
  Fold score (accuracy): 99.5

  Final score (accuracy): 99.45

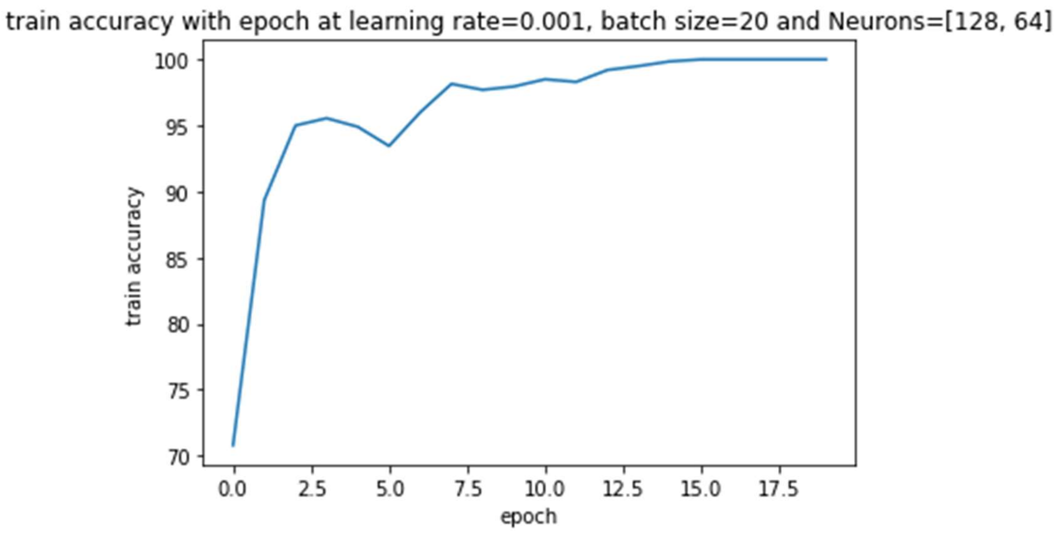
4. Batch Size =20

Results for [LR,Number of neurons,Batch size] at [0.001,[128,64],20]

- Graph for Train loss vs epoch



- Graph for Train accuracy vs epoch



- Test loss accuracy and Time taken

epoch:19 batch: 100 Train loss: 6.584e-05 Train accuracy: 100.0

Test loss: 0.5687 Test accuracy: 91.8 Time taken for training(sec)=9.821

- **Confusion Matrix for train and test**
  For Training

0	1	2	3	4	5	6	7	8	9	
0	210	0	2	0	0	0	1	0	0	0
1	0	170	0	1	1	0	0	0	2	0
2	0	0	209	1	2	0	0	2	1	2
3	0	1	1	179	2	3	0	1	4	1
4	0	0	0	1	199	1	1	1	0	5
5	2	0	0	3	0	197	1	1	1	0
6	1	0	0	0	1	1	213	0	0	0
7	1	0	0	0	0	0	0	179	0	2
8	1	3	4	2	0	3	0	0	192	1
9	1	0	1	2	2	1	0	4	0	176

For Testing

0	1	2	3	4	5	6	7	8	9	
0	92	0	1	1	0	0	1	0	0	0
1	0	108	1	1	0	0	0	2	0	1
2	1	0	83	3	0	0	0	2	1	0
3	0	0	0	105	1	1	0	4	4	0
4	0	0	3	0	87	0	1	1	1	2
5	0	2	1	4	0	76	0	0	0	1
6	2	1	3	1	2	1	100	0	1	0
7	0	1	0	0	0	1	0	98	1	3
8	2	1	2	0	1	4	2	0	77	0
9	0	0	1	2	5	2	0	2	0	92

- CV Results

  Fold #1

  Fold score (accuracy): 100.0

  Fold #2

  Fold score (accuracy): 100.0

  Fold #3

  Fold score (accuracy): 100.0

Fold #4

Fold score (accuracy): 100.0

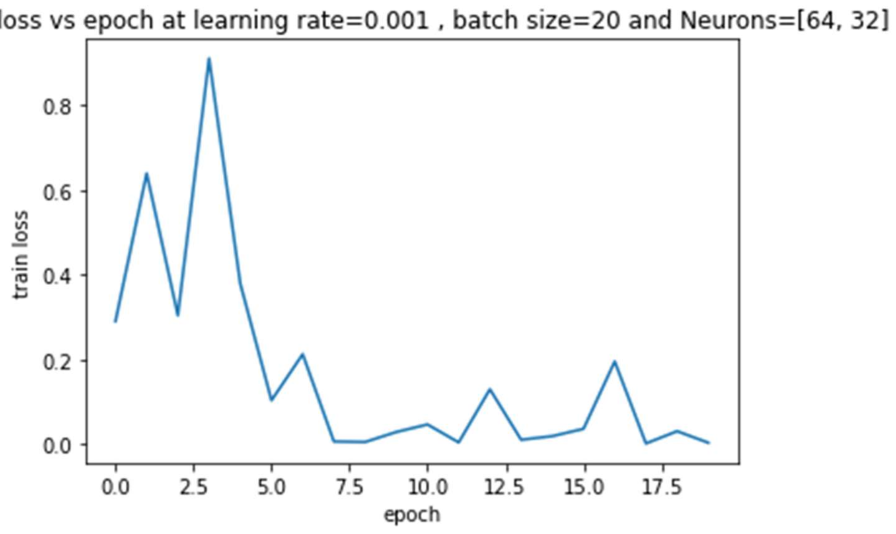
Fold #5

Fold score (accuracy): 100.0

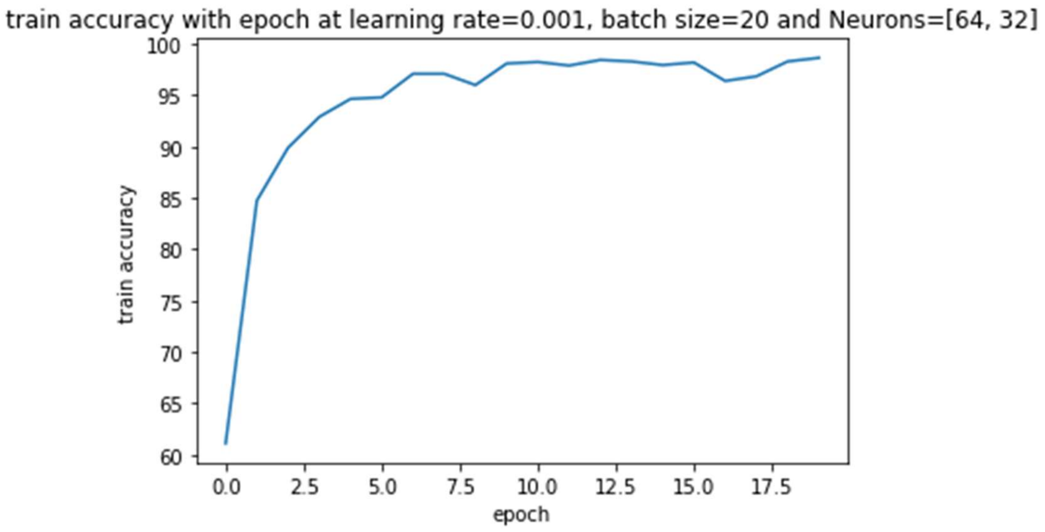
Final score (accuracy): 100.

Results for [LR,Number of neurons,Batch size] at [0.001,[64,32],20]

- Graph for Train loss vs epoch



- Graph for Train accuracy vs epoch



- Test loss accuracy and Time taken

epoch:19 batch: 100 Train loss: 0.00324 Train accuracy: 98.6

Test loss: 0.7779 Test accuracy: 89.5 Time taken for training(sec)=8.06

- **Confusion Matrix for train and test**
  For Training

0	1	2	3	4	5	6	7	8	9	
0	196	0	1	1	2	4	0	1	0	0
1	1	174	0	0	0	1	0	1	2	0
2	0	0	187	0	0	1	1	1	9	0
3	2	2	0	168	0	6	0	2	6	1
4	0	1	0	0	210	0	1	2	2	7
5	2	4	1	2	0	214	0	0	3	2
6	4	0	2	0	1	1	168	0	0	1
7	1	0	0	1	1	0	0	193	0	4
8	1	1	3	4	0	1	0	0	191	4
9	0	1	0	2	5	1	0	1	0	187

  For Testing

	0	1	2	3	4	5	6	7	8	9
0	91	0	2	2	1	1	1	0	0	0
1	0	103	1	0	0	0	0	0	0	0
2	2	2	83	0	1	0	1	2	1	0
3	1	3	0	102	1	3	1	2	2	2
4	0	0	2	0	83	0	1	1	0	3
5	1	1	1	5	0	72	2	0	0	0
6	1	1	3	1	1	2	98	0	1	0
7	0	0	1	2	1	1	0	100	2	9
8	1	3	2	5	3	5	0	1	78	0
9	0	0	0	0	5	1	0	3	1	85

- CV Results

  Fold #1

  Fold score (accuracy): 97.5

  Fold #2

  Fold score (accuracy): 99.0

  Fold #3

Fold score (accuracy): 99.25

Fold #4

Fold score (accuracy): 99.25

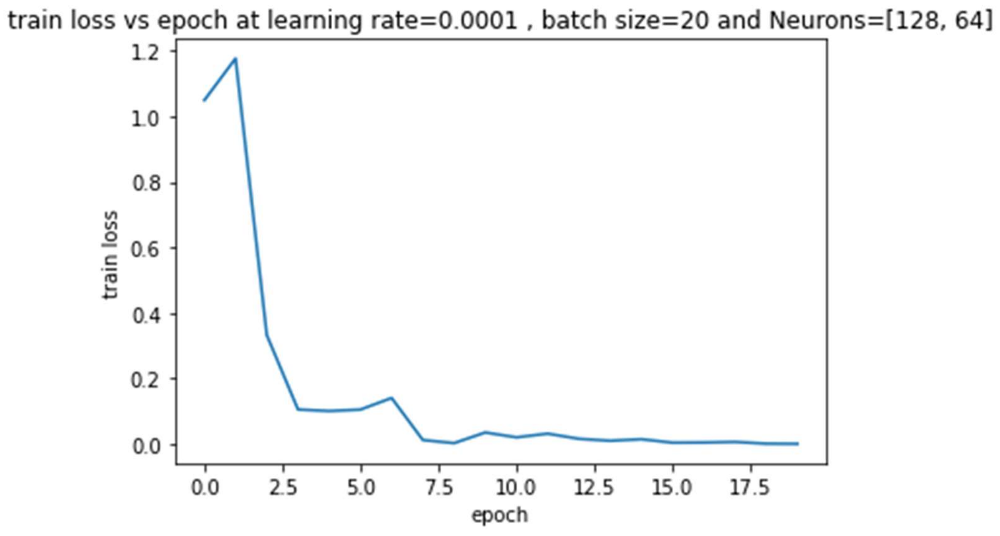
Fold #5

Fold score (accuracy): 99.0

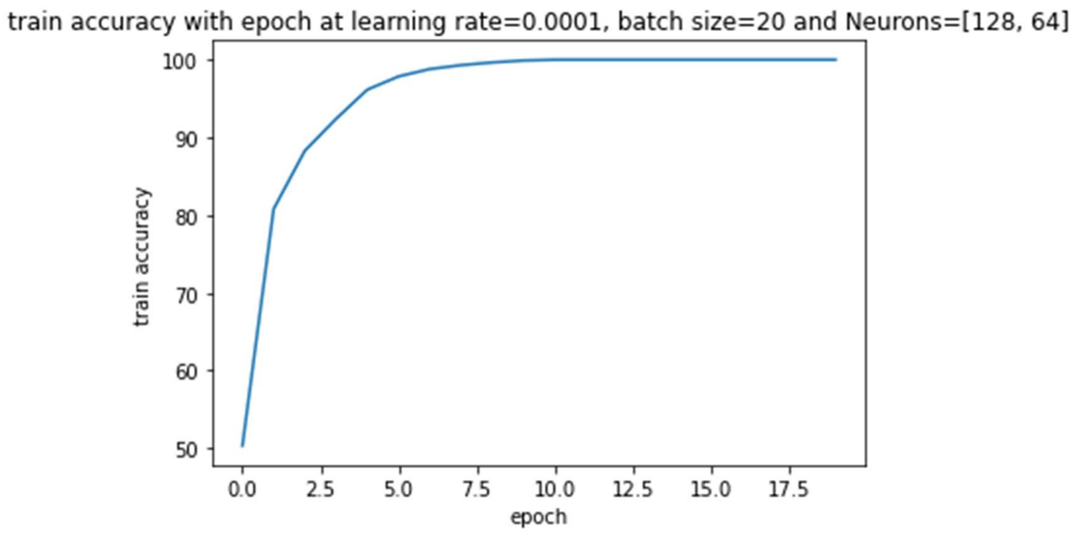
Final score (accuracy): 98.80

Results for [LR,Number of neurons,Batch size] at [0.0001,[128,64],20]

- Graph for Train loss vs epoch



- Graph for Train accuracy vs epoch



- Test loss accuracy and Time taken

  epoch:19 batch: 100 Train loss: 0.0013Train accuracy: 100.0

  Test loss: 0.5177 Test accuracy: 87.3 Time taken for training(sec)=8.49

- **Confusion Matrix for train and test**
  For Training

0	1	2	3	4	5	6	7	8	9	
0	207	0	4	1	0	1	2	0	2	0
1	1	187	0	0	0	0	0	0	1	1
2	0	1	192	2	1	2	2	1	0	1
3	1	0	0	152	0	4	1	1	2	1
4	0	2	0	1	194	2	2	0	0	5
5	0	1	0	5	0	188	0	1	3	3
6	0	0	2	0	0	2	197	0	2	1
7	0	1	1	1	2	0	0	190	0	2
8	1	5	1	2	0	1	1	0	191	1
9	0	2	3	0	3	1	0	3	1	205

  For Testing

	0	1	2	3	4	5	6	7	8	9
0	90	0	2	4	0	2	1	0	0	1
1	0	106	3	2	0	1	1	2	1	1
2	1	3	79	2	1	0	3	2	0	2
3	2	0	3	95	0	2	0	5	1	0
4	0	1	2	0	81	0	1	2	2	3
5	1	1	1	7	0	70	3	0	0	1
6	1	1	2	1	3	2	94	0	1	0
7	0	0	0	1	0	1	0	97	1	4
8	2	1	1	5	2	3	1	0	76	2
9	0	0	2	0	9	4	0	1	3	85

- CV Results

  Fold #1

  Fold score (accuracy): 100.0

  Fold #2

Fold score (accuracy): 100.0

Fold #3

Fold score (accuracy): 100.0

Fold #4

Fold score (accuracy): 100.0

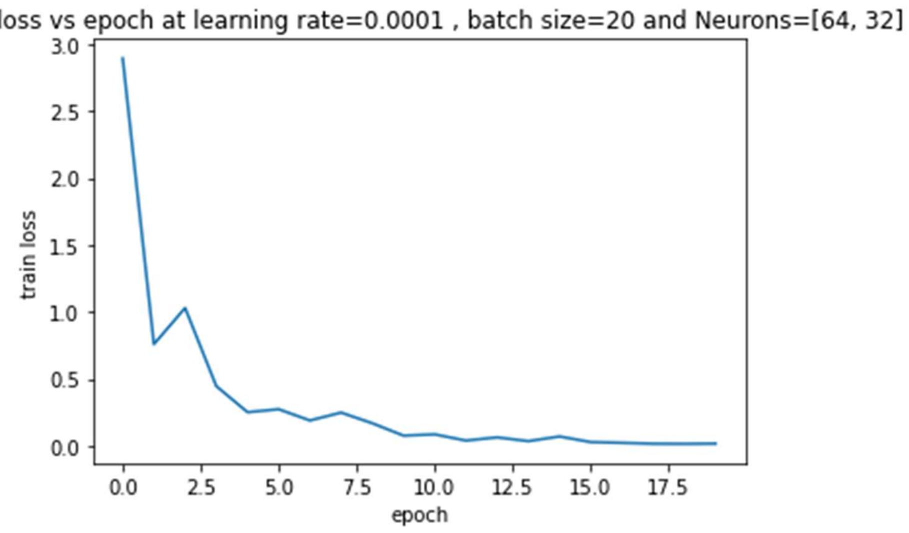
Fold #5

Fold score (accuracy): 100.0

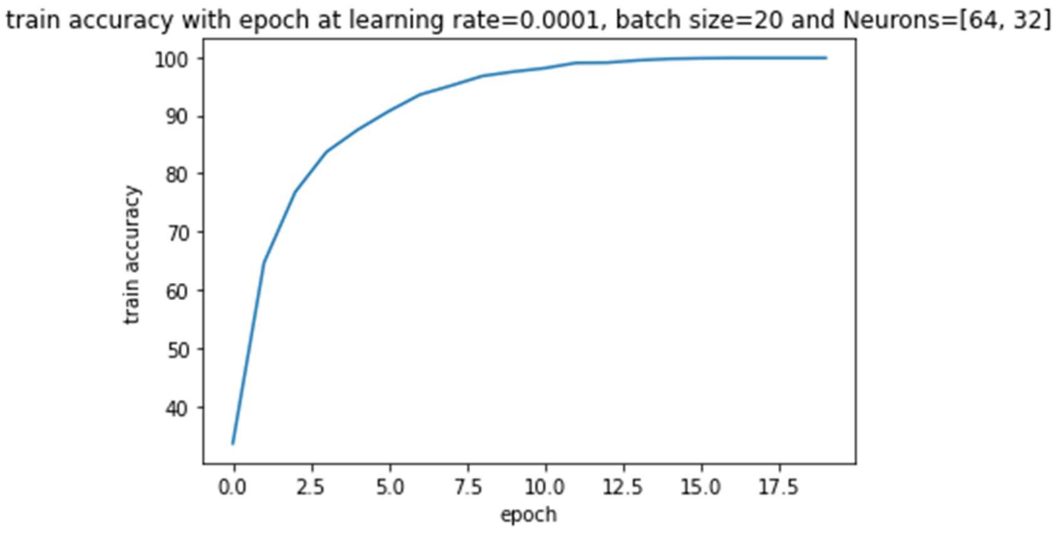
Final score (accuracy): 100.0

Results for [LR,Number of neurons,Batch size] at [0.0001,[64,32],20]

- Graph for Train loss vs epoch



- Graph for Train accuracy vs epoch



- Test loss accuracy and Time taken

```
epoch:19 batch:  100 Train loss: 0.0151795 Train accuracy: 99.84

Test loss: 0.6592 Test accuracy: 85.4 Time taken for training(sec)=7.523
```

- **Confusion Matrix for train and test**
  For Training

0	1	2	3	4	5	6	7	8	9	
0	173	1	1	4	1	0	0	0	1	1
1	0	195	3	0	0	1	0	1	2	2
2	1	0	178	7	3	3	3	1	0	0
3	1	2	4	163	0	9	1	0	7	7
4	1	2	3	0	182	2	0	1	6	3
5	1	0	1	13	5	196	3	0	6	1
6	1	3	5	1	1	0	181	0	3	0
7	0	0	1	1	2	1	1	177	0	6
8	1	5	6	3	2	7	4	2	191	3
9	1	2	2	2	6	1	0	4	0	172

For Testing

0	1	2	3	4	5	6	7	8	9	
0	90	0	0	2	0	2	0	0	1	0
1	0	102	1	1	1	0	1	0	2	1
2	3	8	78	4	2	0	1	4	3	1
3	0	0	1	94	0	4	0	4	2	0
4	0	0	4	1	78	0	2	4	3	4
5	1	0	1	6	1	69	2	1	1	0
6	1	0	3	3	2	3	97	0	1	0
7	0	0	1	1	1	2	1	92	0	7
8	2	3	5	3	2	3	0	0	68	0
9	0	0	1	2	9	2	0	4	4	86

- CV Results

  Fold #1

  Fold score (accuracy): 100.0

  Fold #2

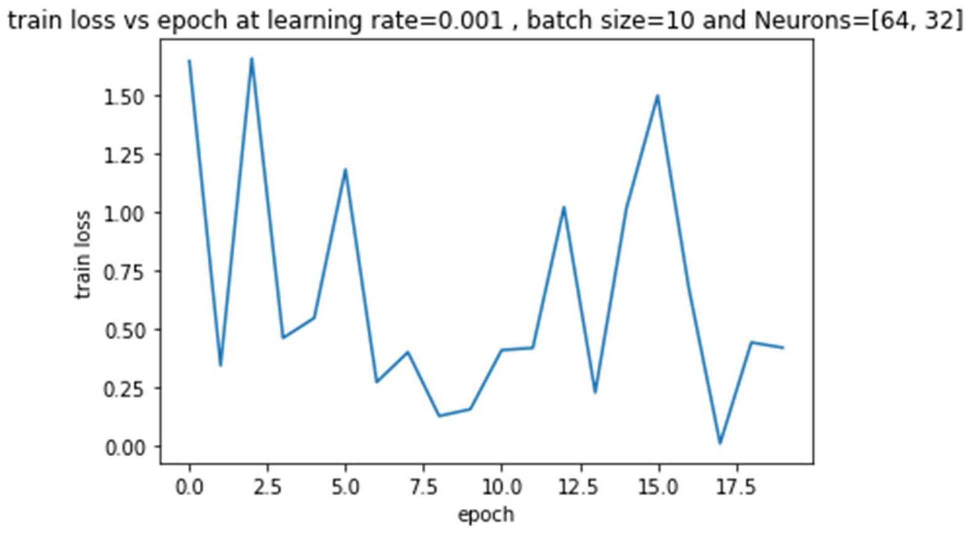
  Fold score (accuracy): 100.0

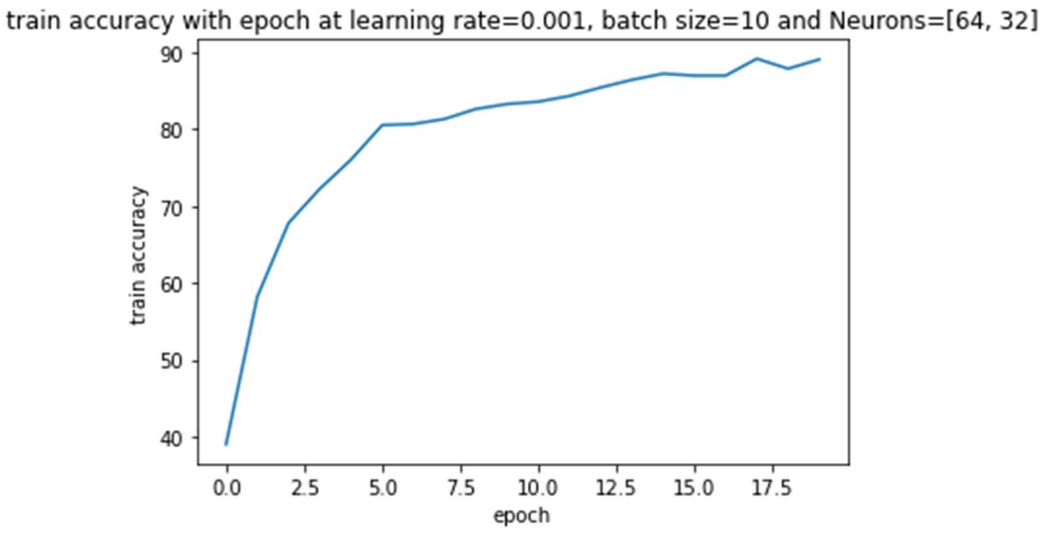
```
Fold #3

Fold score (accuracy): 100.0

Fold #4

Fold score (accuracy): 100.0

Fold #5

Fold score (accuracy): 99.75

Final score (accuracy): 99.95
```

5. Regularization with dropout

I have tried different dropout probability but 0.25 works very well. Time for training have increased. It is saving from the overfit but nut working a good as simple NN.

```
epoch:19 batch:  200 Train loss: 0.4211Train accuracy: 89.09

Test loss: 0.84 Test accuracy: 82.3 Time taken for training(sec)=18.28

Fold #1

Fold score (accuracy): 88.5

Fold #2

Fold score (accuracy): 90.75

Fold #3

Fold score (accuracy): 87.75

Fold #4

Fold score (accuracy): 90.5

Fold #5

Fold score (accuracy): 88.75

Final score (accuracy): 89.25
```





Testing Confusion matrix for regularization

0	1	2	3	4	5	6	7	8	9	
0	83	0	0	0	0	4	1	0	0	0
1	0	95	1	0	0	1	0	0	2	0
2	2	1	86	1	0	2	4	0	0	0
3	0	0	2	81	0	8	0	2	1	0
4	0	0	0	0	69	0	0	0	2	6

5	1	1	0	10	0	73	4	0	1	1
6	4	0	2	1	2	0	86	0	0	0
7	0	0	3	0	2	3	0	87	0	7
8	5	2	16	4	11	7	6	4	84	11
9	1	0	1	0	14	1	0	6	6	79

6. Hyperparameter and Cross validation summary

Model	Epoch=20,K=5	Train accuracy(%)	Test accuracy(%)	K-Fold accuracy(%)	Time to train(Sec)	Underfit/Overfit/ Generalized
Mod1	**[0.001, [128,64],10]**	96	89.2	97.64	8.89	Generalized
Mod2	**[0.001, [64,32],10]**	97.5	89	98.9	7.82	Generalized(Pick)
Mod3	**[0.0001, [128,64],10]**	100	88.8	100	7.76	Overfit
Mod4	**[0.0001, [64,32],10]**	98.94	85.3	99.45	6.63	Overfit
Mod5	**[0.001, [128,64],20]**	100	91.8	100	9.82	Overfit
Mod6	**[0.001, [64,32],20]**	98.6	89.5	98.8	8.06	Overfit
Mod7	**[0.0001, [128,64],20]**	100	87.3	100	8.49	Overfit
Mod8	**[0.0001, [64,32],20]**	99.84	85.4	99.95	7.52	Overfit
Regularization	**[0.001, [64,32],10],0.25**	89.09	82.3	89.25	18.28	Generalized

So, we can see more number of neuron are giving overfitting results as it is memorizing the data due to more capacity.

So will be picking model 2 as the final model.

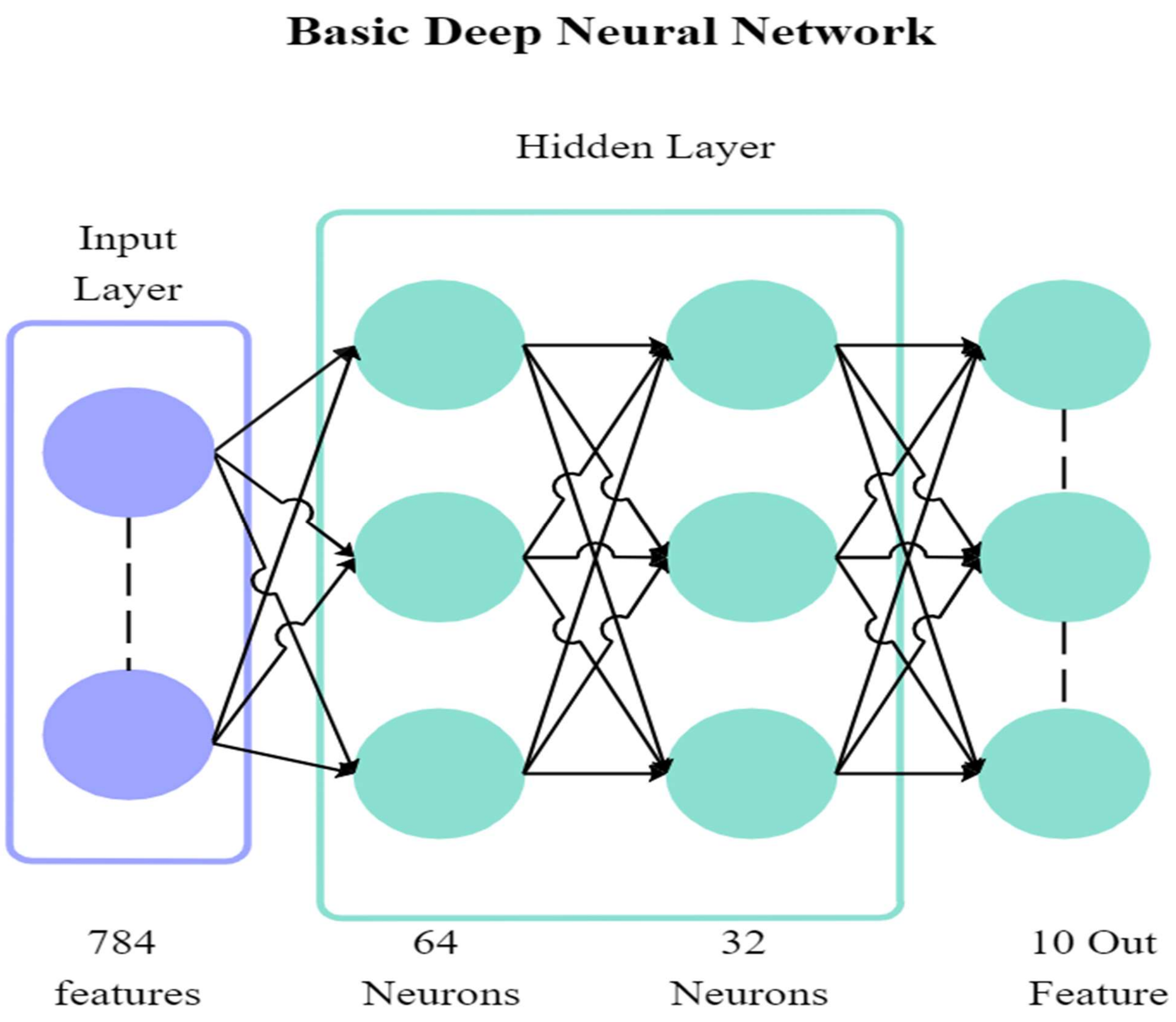
7. Hidden Neuron Representation foe model 2

Mod2	**[0.001, [64,32],10]**	**97.5**	**89**	**98.9**	**7.82**	**Generalized(Pick)**

(x1): Linear(in_features=784, out_features=64, bias=True)

(x2): Linear(in_features=64, out_features=32, bias=True)

(x3): Linear(in_features=32, out_features=10, bias=True)



```
Ordered Dict([('x1.weight',
              tensor([[ 0.0073,  0.0177, -0.0275,  ..., -0.0165, -0.0311, -0.0129],
                      [-0.0182,  0.0049,  0.0175,  ..., -0.0237,  0.0176,  0.0137],
                      [-0.0178,  0.0078,  0.0322,  ..., -0.0078, -0.0280, -0.0156],
                      ...,
                      [-0.0295,  0.0303, -0.0345,  ..., -0.0187, -0.0199,  0.0176],
                      [-0.0030, -0.0128,  0.0065,  ..., -0.0077,  0.0117,  0.0083],
                      [ 0.0278,  0.0010, -0.0075,  ..., -0.0338, -0.0135, -0.0194]])),
             ('x1.bias',
              tensor([-0.0432, -0.0308, -0.0450,  0.0034, -0.0343,  0.0029,  0.0416, -0.0395,
                      -0.0339,  0.0354, -0.0349, -0.0299, -0.0194, -0.0129, -0.0148, -0.0024,
         -0.0265, -0.0351, -0.0146,  0.0130,  0.0552,  0.0196, -0.0042,  0.0223,
         -0.0096, -0.0122,  0.0313,  0.0147, -0.0366, -0.0050,  0.0174,  0.0060,
          0.0114, -0.0115, -0.0351,  0.0023, -0.0057, -0.0382, -0.0276,  0.0154,
          0.0112, -0.0409,  0.0430,  0.0210, -0.0341, -0.0114,  0.0081,  0.0063,
         -0.0215, -0.0305, -0.0191, -0.0060,  0.0140, -0.0379, -0.0272,  0.0091,
          0.0136,  0.0121,  0.0137,  0.0188,  0.0028, -0.0110,  0.0065,  0.0142])),
('x2.weight',
 tensor([[-0.0253, -0.0499,  0.0794,  ...,  0.1055, -0.0139,  0.0320],
         [ 0.0926, -0.0752, -0.0917,  ...,  0.0749, -0.0947,  0.0392],
         [-0.0886,  0.0666, -0.0225,  ..., -0.0633, -0.0679, -0.0710],
         ...,
         [-0.1094,  0.1086, -0.0623,  ..., -0.0422, -0.0973,  0.0621],
         [ 0.1066,  0.0450, -0.0158,  ..., -0.0175, -0.0328, -0.0836],
         [ 0.0924,  0.0019, -0.0706,  ...,  0.0750,  0.0771, -0.0157]])),
('x2.bias',
 tensor([-0.0274,  0.0461, -0.0812, -0.0284, -0.0409, -0.1304,  0.0376,  0.0258,
          0.0712,  0.0908, -0.0980,  0.0376, -0.0740, -0.1171, -0.0755, -0.0026,
          0.0419, -0.0161,  0.1109, -0.0384,  0.0407, -0.0695,  0.0405, -0.0555,
         -0.0094, -0.0237, -0.1074,  0.0446, -0.0067, -0.0048,  0.0649, -0.0518])),
('x3.weight',
 tensor([[ 3.3355e-02,  5.5480e-02, -5.7151e-02, -9.6644e-02,  4.5154e-02,
          -1.0102e-01,  1.0741e-01,  8.6900e-02, -3.7598e-02,  2.7011e-02,
           4.2815e-02,  1.4810e-01, -4.1110e-02, -4.8840e-02, -1.0413e-01,
           1.0109e-01, -1.6579e-01, -8.3479e-03,  7.7700e-02,  9.3501e-02,
          -1.4313e-01, -3.9077e-03,  2.3961e-02, -1.1172e-01,  1.1879e-01,
           2.9632e-03, -2.0927e-02, -4.0597e-02, -1.2255e-01, -5.4601e-03,
          -3.6578e-02,  1.0137e-02],
         [-4.8840e-02, -2.6603e-02, -5.5412e-02,  4.8693e-02,  9.5399e-02,
          -2.2156e-01, -1.1026e-01,  4.5900e-02,  1.0444e-01,  1.6075e-01,
          -1.3325e-01, -1.2226e-01, -7.6031e-02, -1.7523e-01, -1.0030e-01,
 -1.2330e-01,  5.8201e-02, -1.3994e-01, -1.6089e-01,  6.0373e-02,
 -8.1995e-02,  1.6566e-01, -1.3054e-01, -9.7372e-03, -1.8753e-02,
 -1.4560e-01, -1.3676e-01,  8.1771e-02, -9.6702e-02,  3.2206e-02,
  1.1886e-01,  1.7511e-03],
[ 6.0334e-02,  4.5038e-02, -6.3806e-02, -1.1567e-01, -9.9152e-02,
 -6.8509e-02, -7.5749e-02,  6.1960e-02,  2.5344e-02, -5.7138e-02,
  1.3378e-01, -8.2126e-02,  9.6532e-02, -8.7117e-02, -1.1187e-02,
  1.2243e-01, -8.2770e-02,  5.0941e-02, -1.6027e-02, -2.6526e-02,
  1.1805e-01,  1.7200e-01, -4.9525e-02,  1.5332e-01, -7.0996e-03,
  1.3930e-01,  6.0560e-02,  4.8883e-02, -6.1377e-02,  2.2337e-02,
  1.5161e-01,  1.0933e-01],
[-8.0114e-02, -9.1585e-02, -4.4414e-02,  6.8370e-02,  6.1105e-02,
  2.6359e-02, -8.4449e-02, -6.1779e-02, -8.3400e-02,  2.6999e-03,
 -3.8579e-02,  2.8690e-02, -1.3206e-01,  6.9449e-02, -2.6100e-02,
 -1.8600e-02, -3.4291e-03, -7.3613e-03,  1.1294e-01, -1.1274e-02,
 -3.4828e-02,  1.0364e-01, -1.4566e-01,  4.6389e-02,  1.4416e-02,
 -2.0040e-01,  1.0423e-01,  3.7226e-02, -1.0594e-01,  1.1380e-01,
  8.5249e-02,  6.0869e-02],
[ 8.1009e-02,  8.3126e-05,  1.5632e-01, -9.6806e-03, -9.9499e-02,
 -5.5172e-02, -3.5486e-02, -6.1963e-02,  1.4741e-01, -1.4217e-01,
  5.6017e-02, -1.8460e-02,  5.3713e-02, -6.8186e-02, -1.6312e-01,
  1.9453e-02,  6.4753e-02,  1.7479e-01,  5.1509e-02, -2.1996e-02,
  1.8872e-01, -6.5754e-02,  8.0750e-02,  5.0177e-02,  2.6270e-02,
  1.5109e-01,  9.7376e-02, -7.3803e-02,  2.5396e-02,  1.1147e-01,
  3.5267e-02, -1.2142e-01],
[ 9.4772e-02, -1.0202e-02, -1.3483e-01,  1.2543e-01, -7.8470e-02,
  6.0216e-02,  4.0154e-02,  4.3020e-02,  1.2187e-01,  6.8926e-02,
  1.1451e-01, -6.8061e-02, -1.3130e-01,  2.4906e-02,  9.5567e-02,
 -1.1941e-01,  8.6044e-02,  1.2319e-01,  7.2234e-02, -8.4918e-02,
 -6.3487e-02, -7.3154e-02,  1.6152e-01,  3.6315e-02,  2.8876e-02,
 -3.0196e-02,  4.9188e-02, -2.6658e-02, -8.9969e-02,  5.3803e-03,
  5.8363e-02, -9.4775e-02],
[-1.1249e-01,  1.7290e-01,  7.0442e-02,  6.4141e-03,  6.9088e-02,
  1.4866e-02,  6.8993e-02,  8.8571e-02,  7.0331e-02, -1.5796e-01,
 -9.3930e-02, -5.1283e-02,  2.1028e-02, -1.0968e-01, -1.3596e-01,
 -1.0427e-02, -1.1124e-01,  3.0444e-02, -1.6305e-01, -1.3894e-01,
  1.0043e-01,  8.9516e-02, -6.2762e-02,  7.0916e-03,  1.2116e-01,
 -5.4392e-02, -1.2624e-01, -9.8831e-02, -4.3707e-02, -5.2229e-02,
 -1.6186e-01, -1.0501e-01],
[ 4.8405e-02, -1.8258e-01, -5.2244e-02, -2.6332e-02,  7.1344e-02,
  5.6615e-02,  3.3945e-02, -1.5221e-01, -6.0872e-02,  5.8505e-02,
  1.0089e-01, -1.7694e-02, -1.3344e-01, -7.7550e-02, -1.1935e-01,
  6.2919e-02,  8.3131e-02, -1.6812e-01,  8.9642e-02,  6.9698e-02,
 -1.2910e-01, -1.0141e-02, -4.9259e-02, -4.3266e-02,  1.7131e-01,
  1.0361e-01, -1.2320e-01, -1.3592e-01, -6.0839e-02,  2.9333e-02,
  5.0480e-02,  4.3113e-02],
[-2.1198e-01,  5.7253e-03, -3.2904e-03,  8.1360e-02, -4.5288e-02,
 -9.8439e-02,  4.3937e-02, -4.3809e-02,  6.5607e-02, -1.0882e-01,
  4.0409e-02, -1.6550e-01, -7.9645e-02, -1.1302e-01, -9.4206e-02,
  9.7668e-02, -5.1222e-02, -2.0018e-02,  2.0533e-02,  1.2288e-01,
  9.7123e-02,  4.1729e-02,  1.1714e-01,  6.7392e-03,  3.3370e-02,
 -7.8736e-03,  3.7278e-02,  9.8870e-03,  4.2215e-02,  6.9745e-02,
 -7.4494e-02,  3.5127e-02],
[ 1.1442e-01, -1.4481e-02, -3.0073e-02, -6.2372e-02, -5.1139e-02,
  1.4206e-01, -4.3966e-02, -7.9624e-02,  1.5201e-01, -1.8645e-01,
  6.7773e-02, -9.8610e-02, -7.2492e-02, -6.5140e-02, -1.0296e-01,
 -4.1853e-02, -9.4876e-02, -1.0621e-01,  9.2451e-02,  1.5826e-01,
  7.0520e-02, -3.9322e-02,  1.1515e-01,  6.3688e-02, -2.5712e-02,
  1.5636e-01, -5.3550e-02,  4.9586e-02,  1.2992e-01,  9.2514e-02,
 -1.8180e-02, -3.1030e-02]])),
('x3.bias',
 tensor([-0.0583, -0.1253, -0.1388,  0.0904,  0.1574,  0.1670, -0.0028, -0.1929,
        -0.1519, -0.0780]))])
```

So, we can see output weight matrix is of 10X32 size. As we have 10 features and 32 neurons also, we have 10 biases for every output. All the classes are well balanced, so it is a balanced data.

8. Miss classified Class Enquiry

Model 2 Train and test Confusion matrix shows that during train class 7,8,and 9 are highly miss classified.

During testing 5,8 and 9 classes are misclassified more. So, we can say class 8 and 9 are having comparatively high misclassification.

For Training(2000 Samples)

	0	1	2	3	4	5	6	7	8	9
0	179	1	3	0	0	4	0	1	0	1
1	0	185	1	0	0	2	2	0	8	0
2	2	1	190	4	0	0	6	2	3	2
3	3	1	5	134	0	2	0	0	6	2
4	3	0	0	0	181	1	0	2	1	6
5	3	1	0	5	2	191	2	0	4	3
6	0	1	3	1	2	3	193	1	5	1
7	0	0	3	0	6	1	1	197	0	10
8	2	3	3	6	0	11	4	0	193	2
9	1	0	0	4	9	0	1	6	4	168

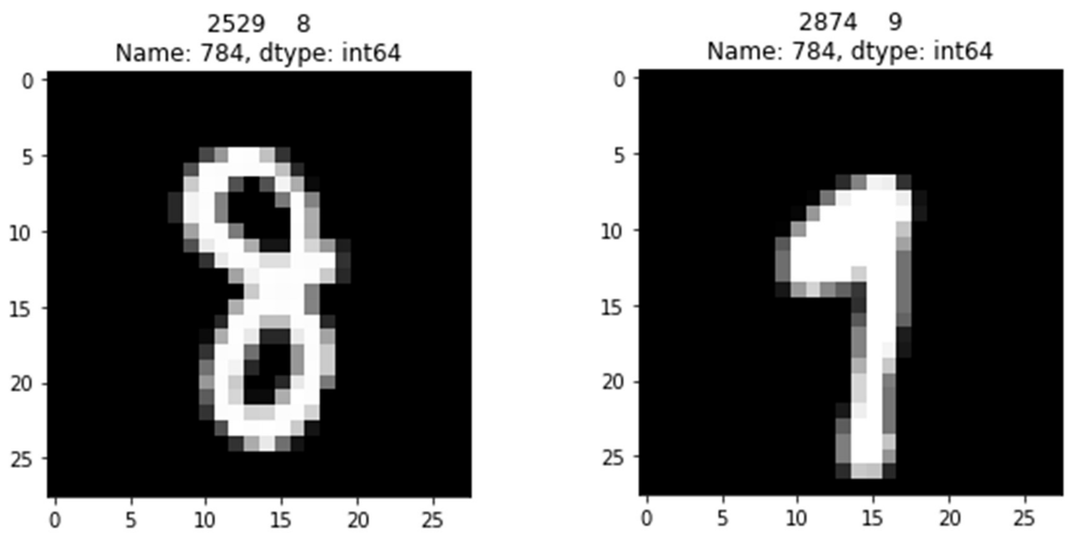
For Testing(1000 Samples)

	0	1	2	3	4	5	6	7	8	9
0	91	0	1	0	0	1	5	0	0	0
1	0	100	0	0	0	0	1	1	1	0
2	1	0	86	3	1	0	0	4	1	0
3	2	2	2	94	0	1	0	1	1	0
4	0	0	1	1	84	0	0	4	0	2
5	1	2	1	10	0	76	1	0	0	1
6	1	0	2	0	2	0	95	0	1	0
7	0	1	1	0	1	0	0	94	0	5

8	1	8	1	6	0	5	2	1	79	0
9	0	0	0	3	8	2	0	4	2	91

After printing these two classes we observe as there are a lot similar in these two class that's why they are getting misclassified.

```
pixels = np.array(feature[2:3], dtype = 'int64')
print(pixels.shape)
# Reshape the array into 28 x 28 array (2-dimensional array)
pixels = pixels.reshape((28, 28))
print(pixels.shape)
# Plot
plt.title(f'{target[2:3]}')
plt.imshow(pixels, cmap='gray')
plt.show()
```



NON-AIP701-PART 1_B

9. Comparing with PCA features

They are very different representation as PCA is converting higher dimension to lower dimensions sample wise but neural nets wts are common for all the data points. If we see the representation of PCA and neural nets wts then we found that same digit can be represented by many ways in PCA(300X25), but neural net is more compressed for it is representing just by 32 wts if the last hidden layer is of 32 neurons. It means PCA has compressed from 300X784 to 300X25 and neural net has compressed from 300X784 to 1X32. So we can say **YES** Neural network is able to learn better representation than PCA.

10. Training model with PCA output(NN with no hidden layer)

Training scheme-

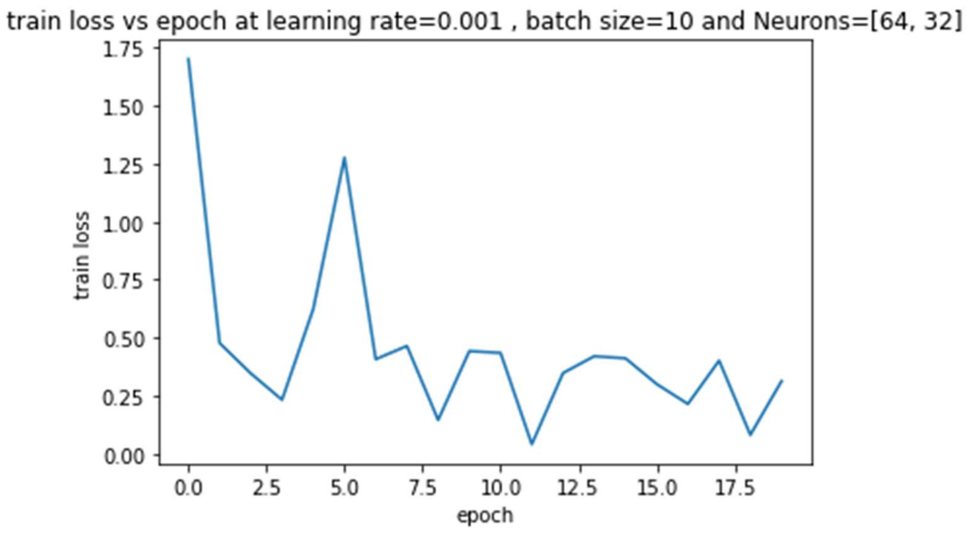
Epoch=20

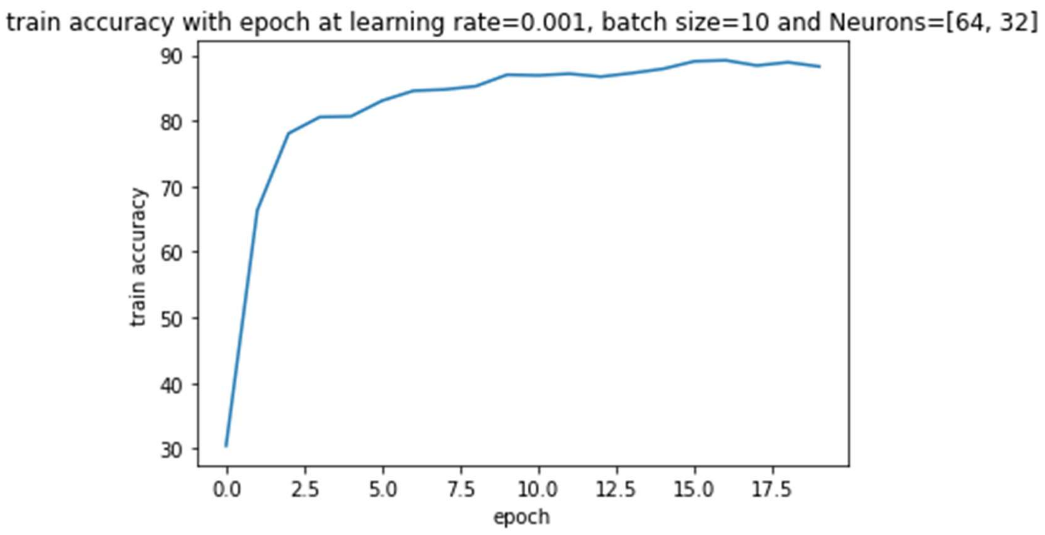
Batch size=10

LR=0.001

epoch:19 batch: 200 Train loss: 0.356 Train accuracy: 86.75

Test loss: 0.5318 Test accuracy: 85.1 Time taken for training(sec)=5.7





Fold #1

Fold score (accuracy): 90.75

Fold #2

Fold score (accuracy): 91.0

Fold #3

Fold score (accuracy): 89.5

Fold #4

Fold score (accuracy): 90.0

Fold #5

Fold score (accuracy): 90.25

Final score (accuracy): 90.3

	0	1	2	3	4	5	6	7	8	9
0	95	0	2	2	1	1	3	0	0	0
1	0	97	3	1	0	0	0	0	3	0
2	1	0	78	0	1	0	3	2	3	2
3	2	1	1	89	0	8	1	0	4	1
4	0	1	4	0	83	0	2	1	4	10
5	4	0	0	8	3	83	5	0	3	3
6	1	0	1	1	1	2	90	0	3	0
7	0	1	7	0	0	2	0	87	0	11
8	0	4	1	1	0	1	1	1	75	3
9	0	0	1	1	3	0	0	6	6	70

11. Training model with PCA output(NN with 2 hidden layer)

Training scheme-(Same as above with regularization)

Epoch=20

Dropout Prob.=0.25

Input feature=25

LR=0.001

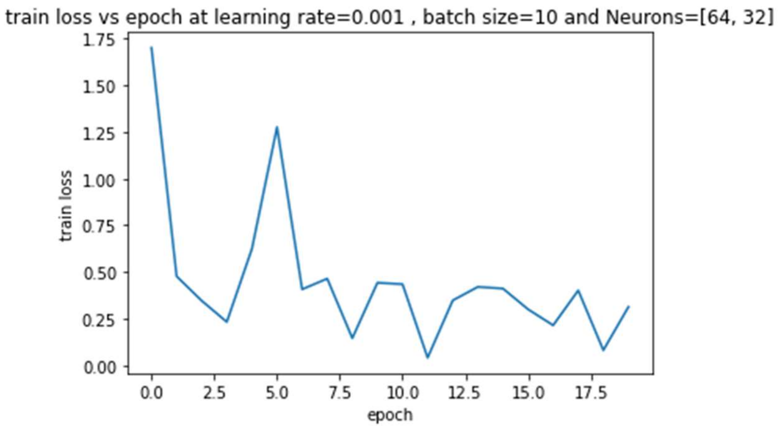
Batch size=10

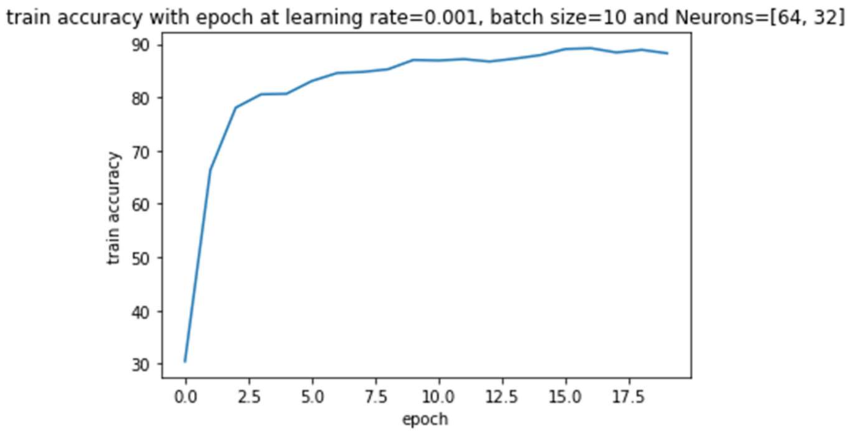
Neurons=[64,32]

Hidden layers=2

Activation function =Relu

```
epoch:19 batch:  200 Train loss: 0.313Train accuracy: 88.25

Test loss: 0.470 Test accuracy: 84.7 Time taken for training(sec)=10.59
```





```
Fold #1

Fold score (accuracy): 90.75

Fold #2

Fold score (accuracy): 91.0

Fold #3

Fold score (accuracy): 89.5

Fold #4

Fold score (accuracy): 90.0

Fold #5

Fold score (accuracy): 90.25

Final score (accuracy): 90.3
```

Model	Epoch=20,K=5	Train accuracy(%)	Test accuracy(%)	K-Fold accuracy(%)	Time to train(Sec)	Underfit/Overfit/ Generalized
Regularization	**[0.001, [64,32],10],0.25**	89.09	82.3	89.25	18.28	Generalized
PCA Model(0 hidden layers)	**[0.001]**	86.75	85.1	90.3	5.7	Generalized(best)
PCA Model(2 hidden layers)	**[0.001, [64,32],10],0.25**	88.25	84.7	90.3	10.59	Generalized

12. Conclusion

Model with no hidden layer which is trained on PCA outputs shows the best results as the data is very simple.

We get very intuitive results as input feature where just 25 so model training was faster by 13 sec which is very good.

PCA feature helped Neural net learn effectively as test accuracy is increase by 3 percent. So neural net trained on

PCA feature gives more generalized model with no hidden layers.

So, we can say adding hidden layer does not help the training improvement this is because of PCA output is very simple for model. 64 ,32 neurons having very high capacity, so they are memorizing the data giving better train accuracy and poor test accuracy. But with no hidden layer it is generalizing better as it will work simple classification problem and it is learning true features.

Output compared to raw pixel is not good which is very intuitive as we are losing the information in PCA.

NON-AIP701-PART 2

13. Convolutional Neural net for MNIST classification

Here Pytorch module is used to build a Neural network using convolution layers along with pooling layer.

Also stride and padding used to make model faster and conserver the information at corners as well. Kernel size use 5X5 and the channel is chosen 1 as the data is grey scale images.

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2,
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        # fully connected layer, output 10 classes
        self.out = nn.Linear(32 * 7 * 7, 10)
    def forward(self, x):

        x = self.conv1(x)
        x = self.conv2(x)
        # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output, x    # return x for visualization
```

```
CNN(
  (conv1): Sequential(
    (0): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (out): Linear(in_features=1568, out_features=10, bias=True)
)
```

```
Epoch [10/10], Step [100/600], Loss: 0.0184 Train Accuracy : =100.0
Epoch [10/10], Step [200/600], Loss: 0.0800 Train Accuracy : =98.0
Epoch [10/10], Step [300/600], Loss: 0.0395 Train Accuracy : =98.0
Epoch [10/10], Step [400/600], Loss: 0.0461 Train Accuracy : =98.0
Epoch [10/10], Step [500/600], Loss: 0.0474 Train Accuracy : =98.0
Epoch [10/10], Step [600/600], Loss: 0.0610 Train Accuracy : =98.0
```

Test Accuracy of the model on the 10000 test images: =95.0   and Time taken is =250 seconds

So, the error is around 5% which is better than many models available officially. If we compare with deep learning model available, it is not performing better as more epochs need to train.

CLASSIFIER	PREPROCESSING	TEST ERROR RATE (%)	Reference
Linear Classifiers			
linear classifier (1-layer NN)	none	12.0	LeCun et al. 1998
linear classifier (1-layer NN)	deskewing	8.4	LeCun et al. 1998
pairwise linear classifier	deskewing	7.6	LeCun et al. 1998
K-Nearest Neighbors			
K-nearest-neighbors, Euclidean (L2)	none	5.0	LeCun et al. 1998
K-nearest-neighbors, Euclidean (L2)	none	3.09	Kenneth Wilder, U. Chicago
K-nearest-neighbors, L3	none	2.83	Kenneth Wilder, U. Chicago
K-nearest-neighbors, Euclidean (L2)	deskewing	2.4	LeCun et al. 1998

14. Autoencoder with Conv2D for MNIST classification

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder,self).__init__()
        self.encoder = nn.Sequential(
            # 28 x 28
            nn.Conv2d(1, 4, kernel_size=5),
            # 4 x 24 x 24
            nn.ReLU(True),
            nn.Conv2d(4, 8, kernel_size=5),
            nn.ReLU(True),
            # 8 x 20 x 20 = 3200
            nn.Flatten(),
            nn.Linear(3200, 10),

            # 10
            nn.Softmax(),
            )
        self.decoder = nn.Sequential(
            # 10
            nn.Linear(10, 400),
            # 400
            nn.ReLU(True),
            nn.Linear(400, 4000),
            # 4000
            nn.ReLU(True),
            nn.Unflatten(1, (10, 20, 20)),
            # 10 x 20 x 20
            nn.ConvTranspose2d(10, 10, kernel_size=5),
            # 24 x 24
            nn.ConvTranspose2d(10, 1, kernel_size=5),
            # 28 x 28
            nn.Sigmoid(),
            )
    def forward(self, x):
        enc = self.encoder(x)
        dec = self.decoder(enc)
        return dec

model_ae=Autoencoder()
print(model_ae)
```

```
Autoencoder(
  (encoder): Sequential(
    (0): Conv2d(1, 4, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(4, 8, kernel_size=(5, 5), stride=(1, 1))
    (3): ReLU(inplace=True)
    (4): Flatten(start_dim=1, end_dim=-1)
    (5): Linear(in_features=3200, out_features=10, bias=True)
    (6): Softmax(dim=None)
  )
  (decoder): Sequential(
    (0): Linear(in_features=10, out_features=400, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=400, out_features=4000, bias=True)
    (3): ReLU(inplace=True)
    (4): Unflatten(dim=1, unflattened_size=(10, 20, 20))
    (5): ConvTranspose2d(10, 10, kernel_size=(5, 5), stride=(1, 1))
    (6): ConvTranspose2d(10, 1, kernel_size=(5, 5), stride=(1, 1))
    (7): Sigmoid()
  )
)
```

The accuracy is very low as it is an unsupervised form of training. And just with 20 epochs it is not enough to train this type of network. Either reduce the dataset or increase the number of epoch for training, The below confusion matrix is for 50 epochs , while only 20 epochs were giving very huge errors.

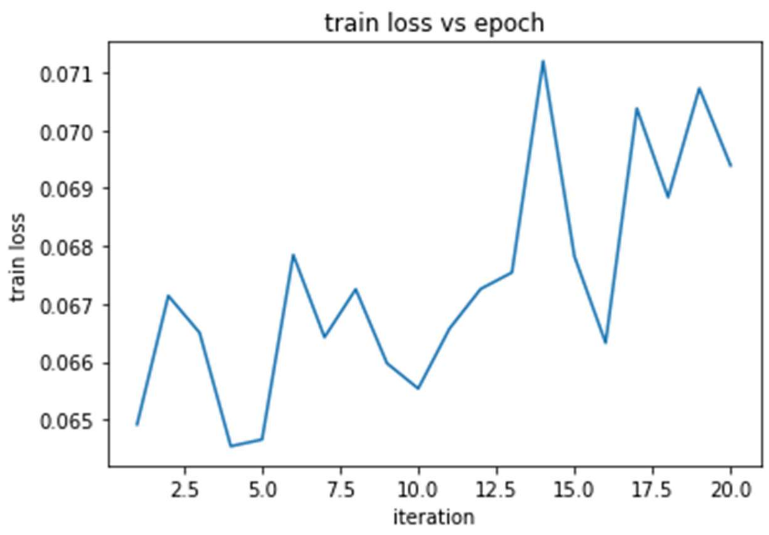
Confusion matrix on 10000 test data

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	959	0	0	21	0
1	0	0	170	0	0	1	0	0	959	5
2	0	0	11	0	0	335	0	0	686	0
3	0	0	65	0	0	492	0	0	453	0
4	0	0	208	0	0	473	0	0	301	0
5	0	0	9	0	0	546	0	0	336	1
6	0	0	11	0	0	658	0	0	289	0
7	0	0	190	0	0	317	0	0	521	0
8	0	0	11	0	0	190	0	0	773	0
9	0	0	224	0	0	406	0	0	379	0

15. Autoencoder with FCNN for MNIST classification

```
class AE(torch.nn.Module):
    def __init__(self):
        super().__init__()

        # Building an linear encoder with Linear
        # layer followed by Relu activation function
        # 784 ==> 9
        self.encoder = torch.nn.Sequential(
            torch.nn.Linear(28 * 28, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, 36),
            torch.nn.ReLU(),
            torch.nn.Linear(36, 18),
            torch.nn.ReLU(),
            torch.nn.Linear(18, 9)
        )

        # Building an linear decoder with Linear
        # layer followed by Relu activation function
        # The Sigmoid activation function
        # outputs the value between 0 and 1
        # 9 ==> 784
        self.decoder = torch.nn.Sequential(
            torch.nn.Linear(9, 18),
            torch.nn.ReLU(),
            torch.nn.Linear(18, 36),
            torch.nn.ReLU(),
            torch.nn.Linear(36, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 28 * 28),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

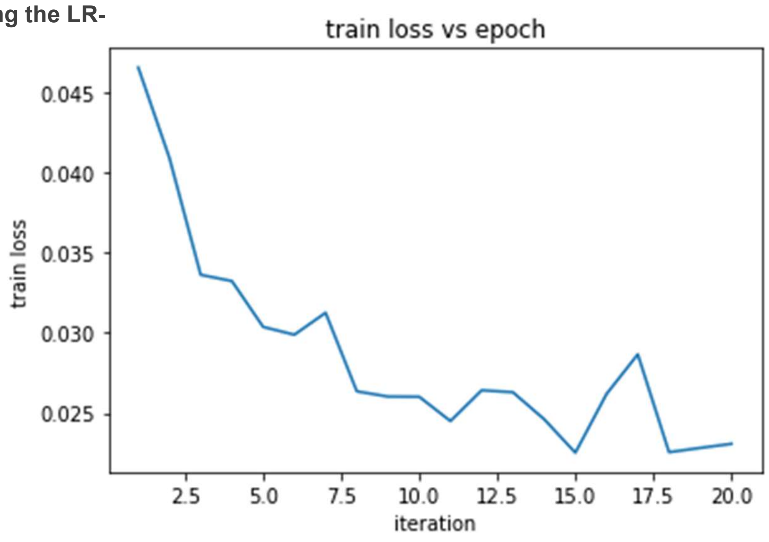
Here the layers are all fully connected. And training them for just 20 epochs are giving a huge error, more epoch training will improve the results as above. Train loss is increasing as with the epoch, so we need hyperparameter tuning for this. Currently the learning rate used is 0.1 we may need to decrease the LR.



Testing of FCNN autoencoder on 10000 test data(LR=0.1)

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	980	0	0	0
1	0	0	0	0	0	0	1135	0	0	0
2	0	0	0	0	0	0	1032	0	0	0
3	0	0	0	0	0	0	1010	0	0	0
4	0	0	0	0	0	0	982	0	0	0
5	0	0	0	0	0	0	892	0	0	0
6	0	0	0	0	0	0	958	0	0	0
7	0	0	0	0	0	0	1028	0	0	0
8	0	0	0	0	0	0	974	0	0	0
9	0	0	0	0	0	0	1009	0	0	0

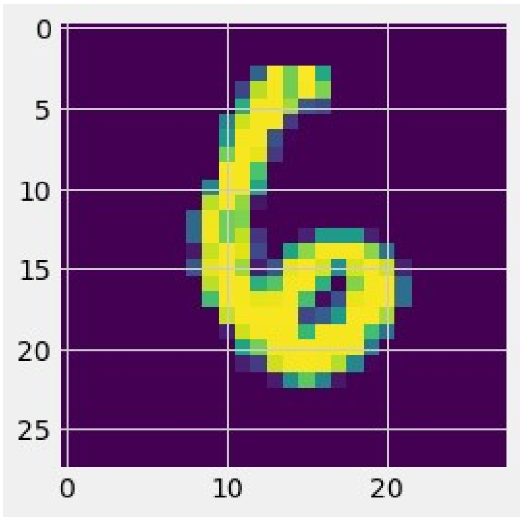
After Tuning the LR-

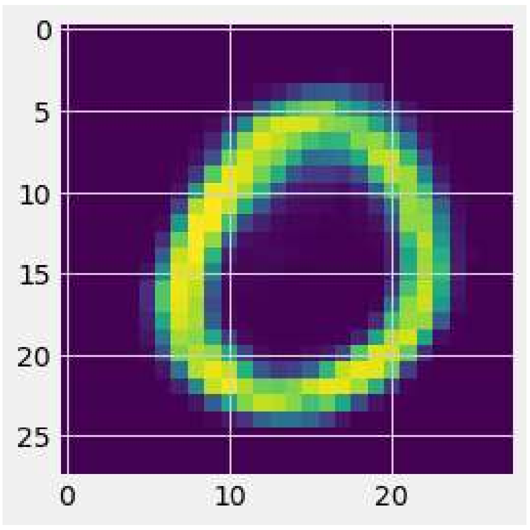


Testing of FCNN autoencoder on 10000 test data(LR=0.001)

	0	1	2	3	4	5	6	7	8	9
0	0	259	0	0	1	710	10	0	0	0
1	0	1127	0	0	8	0	0	0	0	0
2	0	1003	0	0	1	27	0	1	0	0
3	0	949	0	0	23	38	0	0	0	0
4	0	848	0	0	34	2	98	0	0	0
5	0	858	0	0	7	22	5	0	0	0
6	53	449	0	2	183	233	31	7	0	0
7	0	1007	0	0	2	0	19	0	0	0
8	0	961	0	0	9	3	1	0	0	0
9	0	986	0	0	5	1	17	0	0	0

16. Reconstructed Images-





17. Conclusion

CNN works well both accuracy wise and time complexity wise.

Autoencoder are not working well for low epochs they need higher epochs to work well. But they both are learning different features .The results form the convolution are comparable to results available on the site with respect to the linear models and K mean models but not as good compared to DL model present on site as we have trained only for few epochs but the official sites DL model are got trained for higher epochs. Model hyperparameter are equally applicable as compared to normal deep learning fully connected models. The use of stride and max pool layer after the relu activation makes the CNN cost effective.

We can see the sparse autoencoder has learn in unsupervised way but due to less training it is not able to reconstruct the input again. So the representation learnt by autoencoder is very different than the CNN and FCNN.

18. Reference

- training set images (9912422 bytes)--->http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
- training set labels (28881 bytes)----->http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
- test set images (1648877 bytes)------->http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
- test set labels (4542 bytes)---------->http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
- https://www.youtube.com/watch?v=xp-WhryKhas
- https://github.com/SimpleAI2022/MNIST_FC1/blob/main/MNIST_FC1_Complete.ipynb
- https://bytepawn.com/building-a-pytorch-autoencoder-for-mnist-digits.html
- https://medium.com/analytics-vidhya/training-mnist-handwritten-digit-data-using-pytorch-5513bf4614fb
- https://stackoverflow.com/questions/69975400/pytorch-cnn-expected-input-to-have-1-channel-but-got-60000-channels-instead
- https://www.youtube.com/watch?v=Y_hSBwucDjg