Assignment 3

MSBD 5002 (Data Mining)

Gardas Nuthan
20643274

Introduction:

In this assignment, I've done binary and multi-label classification with the help of **Pytorch**, and some additional help with **skorch.**

Binary classification:Binary or binomial classification is the task of classifying the elements of a given set into two groups or labels (Wiki)

Multi-label classification: multi-label classification are variants of the classification problem where multiple labels may be assigned to each instance.

So by defining these two classification, pytorch helps us define, train a Neural network model of one hidden layer (binary classification), two hidden-layers (multi-label)

In pytorch, it's easy to define the network model for both of these multi-layer perceptron models

For example, one hidden-layer:

```
class One_hidden(nn.Module):
    def __init__(self,n_hidden,n_output=2,n_feature=13):
        super(One_hidden,self).__init__()
        self.hidden=nn.Linear(n_feature,n_hidden)
        self.output=nn.Linear(n_hidden,n_output)


    def forward(self,x):
        x=F.relu(self.hidden(x))
        x=self.output(x)

        return x
```

Where the activation layer is ReLu and input as features of the dataset, output as a binary classification label.

For two hidden-layers:

```
'''Neural Network Model'''

#Define the model of your Neural Network:
# two hidden layers with input as features of the data and the labels as output

#input=784 (28*28)
#hidden1,hidden2= Number of hidden nodes
#Output=10 classes

#Activation used: ReLu function
class two_hidden(nn.Module):
    def __init__(self,n_hidden1,n_hidden2,n_output=10,n_feature=784):
        super(two_hidden,self).__init__()
        self.hidden_1=nn.Linear(n_feature,n_hidden1)
        self.hidden_2=nn.Linear(n_hidden1,n_hidden2)
        self.output=nn.Linear(n_hidden2,n_output)

    def forward(self,x):
        x1=F.relu(self.hidden_1(x))
        x2=F.relu(self.hidden_2(x1))
        x=self.output(x2)

        return x
```

Input is a 1D array 789 features, output as 10 classes, whereas two hidden layers are defined with the nodes respectively.

These two models form the basis or foundation of different datasets.

Skorch:
Skorch is basically a sklearn wrapper of pytorch, where we can use sklearn.metrics on pytorch. Pytorch doesn't offer the functionality of these metrics such as auc_score,confusion_matrix, cross validation,etc. Skorch helps us by enabling these metrics to facilitate the training process of the neural network.

```
iris=dataset('iris.npz')

train_X=iris[0]
train_y=iris[1]
test_X=iris[2]
test_y=iris[3]
# We initiate the classifier:
net = NeuralNetClassifier(
        One_hidden(n_feature=4,n_hidden=10, n_output=2),
        max_epochs=30,
        lr=0.1,
        optimizer=optim.SGD,
        criterion=nn.CrossEntropyLoss,
        iterator_train__shuffle=True,
        )


net.fit(train_X,train_y)
```

A skorch implementation, NeuralNetClassifier class is initiated where the parameters along with the neural network model can be defined. Parameters such as optimizer, learning_rate, loss function can be initiated inside the class.

I tried doing both of these implementations to know how cross validation works with a pytorch model.

Task-1:

We get 5 datasets with features given:

| Data Set | #features | # train | # test |
|---|---|---|---|
| Breast cancer | 10 | 547 | 136 |
| Diabetes | 8 | 615 | 153 |
| Digit | 64 | 800 | 200 |
| Iris | 4 | 120 | 30 |
| Wine | 13 | 142 | 36 |

Parameters:
Criterion: CrossEntropyLoss()
Optimizer: SGD,Adam
Learning_rate:0.001
Epochs:20-30, 30 being the common for every dataset.

Criterion:
nn.CrossEntropyLoss()-
Measures the cross-entropy between the predicted and the actual value. Cross-entropy as a loss function is used to learn the probability distribution of the data. While other loss functions like squared loss penalize wrong predictions, cross entropy gives a greater penalty when incorrect predictions are predicted with high confidence. What differentiates it with negative log loss is that cross entropy also penalizes wrong but confident predictions and correct but less confident predictions, while negative log loss does not penalize according to the confidence of predictions.

This loss function is implemented in classification tasks like this assignment, and it yields out good results.

Optimizer:
Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum. It uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of gradient itself like SGD with momentum. It's an adaptive learning method.

SGD can be regarded as a stochastic approximation of gradient descent optimization, since it replaces the actual gradient (calculated from the entire data set) by an estimate thereof (calculated from a randomly selected subset of the data).

These two definitions sums up the entire range of optimizers we use for these datasets. Adam is used preferably than SGD.

Learning rate: the learning rate is a tuning parameter in an optimization algorithm that determines the step size at each iteration while moving toward a minimum of a loss function.

Epochs indicate the number of passes through the entire training dataset or batches of the dataset.

Now that we have defined what these parameters are about, let's look at Task-1 specifically to check how the model is working.

Task-1:

**Diabetes:**

Parameters used:
Criterion: CrossEntropyLoss()
Optimizer: Adam Optimizer
Learning_rate:0.001
Training batch_size=1 i,e: one training sample in the trainloader class
Validation batch_size=2, 2 validation samples in the loader class
Best Hidden parameter:8
Final val_accuracy:78.42%
Final test_accuracy:77.77%
Training time: 9.152s
Auc_score:0.5

**Digit:**

Parameters used:

Criterion: CrossEntropyLoss()
Optimizer: Adam Optimizer
Learning_rate:0.001
Training batch_size=2 i,e: two training sample in the trainloader class
Validation batch_size=1,one validation samples in the loader class
Best Hidden parameter:10
Final val_accuracy:78.42%
Final test_accuracy:77.77%
Training time: 12.35s
Auc_score:0.921

**Iris:**

Parameters used:
Criterion: CrossEntropyLoss()
Optimizer: Adam Optimizer
Learning_rate:0.001
Training batch_size=1,i,e: one training sample in the trainloader class
Validation batch_size=1,one validation samples in the loader class
Best Hidden parameter:10
Final val_accuracy:100%
Final test_accuracy:100%
Training time: 2s
Auc_score:1

**Wine:**

Parameters used:
Criterion: CrossEntropyLoss()
Optimizer: Adam Optimizer
Learning_rate:0.001
Training batch_size=2 i,e: two training sample in the trainloader class
Validation batch_size=1,one validation samples in the loader class
Best Hidden parameter:7
Final val_accuracy:79.31%
Final test_accuracy:77.77%
Training time: 2.2s
Auc_score:0.5000

**Breast cancer:**

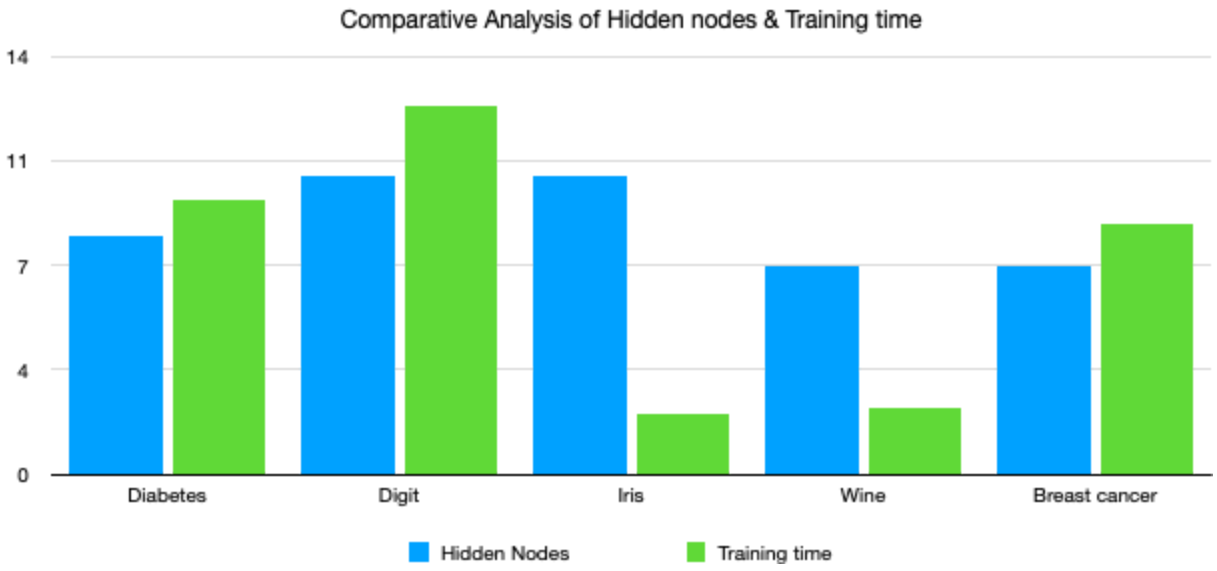
Parameters used:
Criterion: CrossEntropyLoss()

Optimizer: Adam Optimizer
Learning_rate:0.001
Training batch_size=1 i,e: two training sample in the trainloader class
Validation batch_size=2,two validation samples in the loader class
Best Hidden parameter:7
Final val_accuracy:96.36%
Final test_accuracy:96.32%
Training time: 8.36s
Auc_score:0.9470

Comparison of Datasets

| | Hidden Nodes | Training time | AUC Score | Val_accuracy | Final accuracy |
|---|---|---|---|---|---|
| Diabetes | 8 | 9.152 | 0.50000 | 78.42% | 77.77% |
| Digit | 10 | 12.35 | 0.9210 | 97.50% | 95.5% |
| Iris | 10 | 2 | 1.0000 | 100% | 100% |
| Wine | 7 | 2.2 | 0.5000 | 79.31% | 77.77% |
| Breast cancer | 7 | 8.36 | 0.9470 | 96.36% | 96.32% |

Tabular information of final accuracies along with some parameters



Discussion:

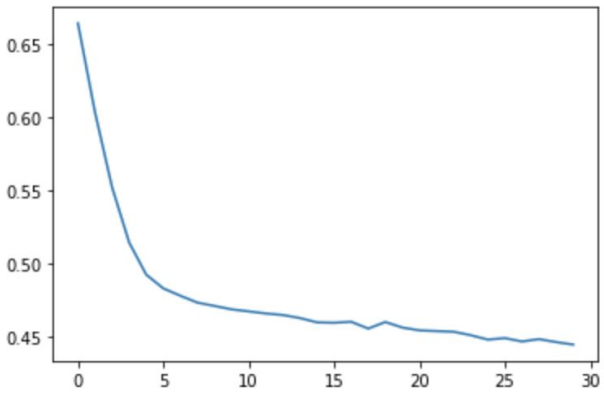
- If you see the second figure, we can draw a conclusion that more the hidden nodes defined, the more training time it computes, except for iris and wine, which relates to smaller dataset (120,142), which allows the network to get a better accuracy in less time
- For the rest, samples are adequate enough to carry on more computation, hence more hidden nodes are suited.

Workflow of the Model:
All of these networks follow one basic pipeline:
Dataloader class is used to load the train,val and testing dataset. These classes are used to input the data in batches if specified and also does the shuffle operation to maintain some randomness even after splitting.

**model_train** function is defined to start the training process with a number of epochs defined, and loss function, optimizer and the learning rate is also defined.
The model forwards the data, computes the loss and computes the backward propagation, finally completing a parameter update(.step()).
Loss values are appended to a list for easier plotting.

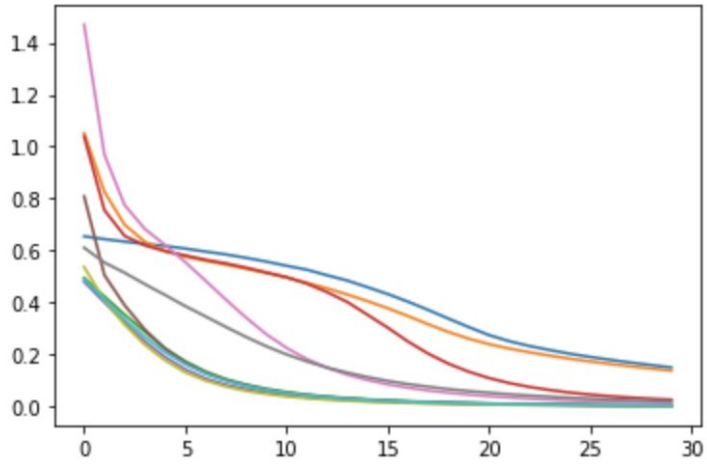
Finally, with a given list of best hidden parameters, I have iterated the training process over these hidden nodes and calculated the validation accuracy for every best possible node. We store them in a dictionary and check the best accuracy among them and use that node to train the model from scratch using all the training instances(mainloader class).

After training the model, we finally test them with the testloader class and report the final accuracy.




Example of a loss curve of Diabetes network model

When we iterate over a range of hidden parameters, we get a bunch of loss curves:




Example of loss curves over the range of hidden parameters

It's difficult to understand each curve over there, but it gives out a more regulated way of understanding how the loss is converging.

**Task-2:**

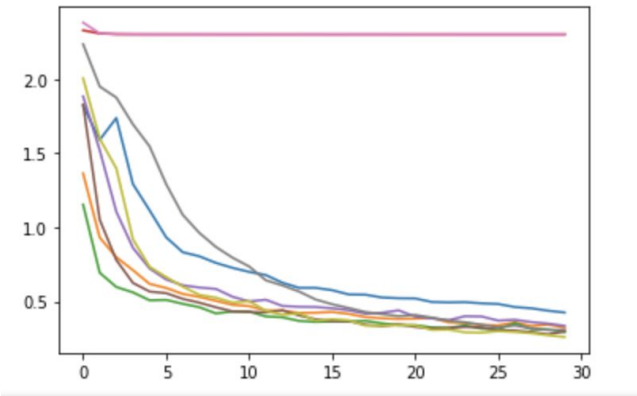
In this task, training samples of 10,000 images(flattened) are given to train a particular 2-hidden layer model over a range of hidden parameters for both hidden layer-1 and layer-2.

The workflow is same as the task-1, but here we are initiating 2 hidden layers, therefore to iterate we have to use the possible values for both of the hidden layers to find the best accuracy among the best hidden parameters.

```
'''Finding Hidden layer nodes'''
#Using a for loop to iterate over the possible values for the best hidden nodes for the two layers, training the moc
# And finding the accuracy of the validation dataset and comparing to find the best one.
h1=[50,75,100]
h2=[10,15,20]
storage=dict()
for i in h1:
    for x in h2:
        net=two_hidden(i,x)
        storage[i,x]=model_train(net,trainloader)

print(storage)
```

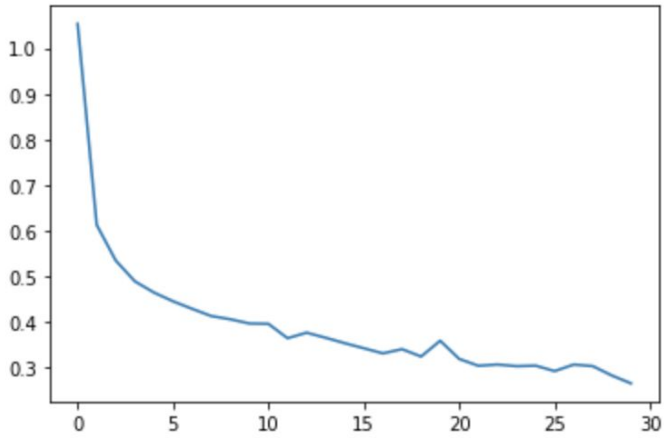
In this code snippet, I've iterated over the hidden parameters, trained the model with those parameters and stored the hidden parameters with the accuracy. The accumulating loss curves of these models is shown




As seen, except for two models, most of the models are trying to converge. But it's still hard to understand, and therefore we have to look at the dictionary of the best accuracies.

From the dictionary, these hidden parameters(75,20) gives the best accuracy over the range:84.25 %

The following loss curve is:




It's converging to the point, and thereby checking the final accuracy of the model gives:84.2%

After this, a confusion matrix was generated with the classes specified, after that calculated the accuracy per class of the confusion matrix.

```
print(confusion_matrix)
```

```
tensor([[ 93.,   1.,   3.,   4.,   0.,   0.,   5.,   0.,   1.,   0.],
        [  0., 100.,   0.,   5.,   0.,   0.,   0.,   0.,   0.,   0.],
        [  4.,   0.,  82.,   1.,  12.,   0.,  12.,   0.,   0.,   0.],
        [  6.,   2.,   4.,  75.,   2.,   0.,   4.,   0.,   0.,   0.],
        [  0.,   0.,  12.,   3.,  93.,   0.,   7.,   0.,   0.,   0.],
        [  0.,   0.,   0.,   0.,   0.,  79.,   0.,   1.,   0.,   7.],
        [ 21.,   0.,  12.,   4.,   9.,   0.,  51.,   0.,   0.,   0.],
        [  0.,   0.,   0.,   0.,   0.,   2.,   0.,  92.,   0.,   1.],
        [  1.,   0.,   0.,   1.,   0.,   0.,   1.,   5.,  87.,   0.],
        [  0.,   0.,   0.,   0.,   0.,   1.,   0.,   4.,   0.,  90.]])
```

And the accuracy per class:

```
tensor([0.8692, 0.9524, 0.7387, 0.8065, 0.8087, 0.9080, 0.5258, 0.9684, 0.9158,
        0.9474])
```

Total classes=10
From the tensor, the 6th class has the lowest accuracy of 52.58%

Therefore to summarise, the parameters are:

**Multi-label Classification:**

Parameters used:
Criterion: CrossEntropyLoss()
Optimizer: Adam Optimizer
Learning_rate:0.001
Training batch_size=25 i,e: 25 training sample in one batch the trainloader class
Validation batch_size=25,25 validation samples in one batch of the loader class
Best Hidden parameter:75,20
Final val_accuracy:84.25%
Final test_accuracy:84.2%
Training time: 27.585s

Skorch Implementation:

In Skorch, the parameters are:

**Diabetes:**

Parameters used:
Criterion: CrossEntropyLoss()
Optimizer: SGD Optimizer
Learning_rate:0.1
epochs=20
Best Hidden parameter:3
Final val_accuracy:65.6%
Final test_accuracy:64.7%
auc_score:0.5
Training time: 9.12

**Digit:**

Parameters used:
Criterion: CrossEntropyLoss()
Optimizer: Adam Optimizer
Learning_rate:0.001
epochs=20
Best Hidden parameter:7
Final val_accuracy:91.25%
Final test_accuracy:91.5%
auc_score:.0.921
Training time: 12

**Iris:**

Parameters used:
Criterion: CrossEntropyLoss()
Optimizer: Adam Optimizer
Learning_rate:0.001
epochs=20
Best Hidden parameter:10
Final val_accuracy:99.1%
Final test_accuracy:100%
auc_score:1.0
Training time: 2

**Wine:**

Parameters used:
Criterion: CrossEntropyLoss()
Optimizer: SGD Optimizer
Learning_rate:0.1
Best Hidden parameter:2
Final val_accuracy:59.8%
Final test_accuracy:61.1%
Auc_score:0.5
Training time:2.2

**Breast cancer:**

Parameters used:
Criterion: CrossEntropyLoss()
Optimizer: SGD Optimizer
epochs=20
Learning_rate:0.1
Best Hidden parameter:10
Final val_accuracy:91.22%
Final test_accuracy:96.32%
Auc_score:0.947
Training time:8.36

These are the parameters obtained from skorch, most of the parameters are similar except for the learning rate and optimizer used.

Task-2:

**Multi-label Classification:**

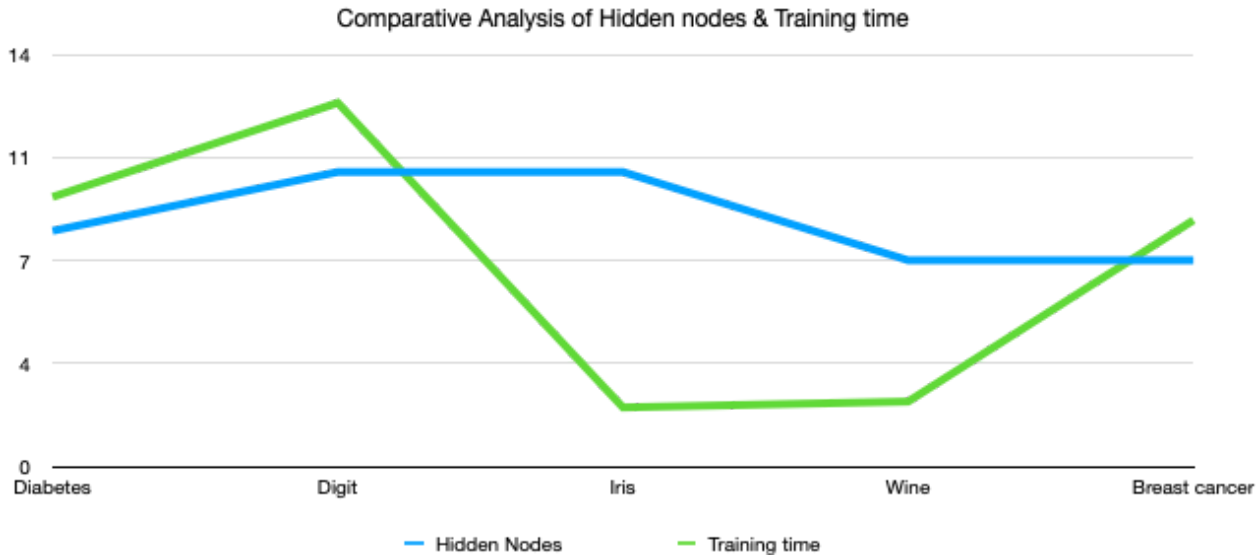
Parameters used:
Criterion: CrossEntropyLoss()
Optimizer: Adam Optimizer
Learning_rate:0.001
Best Hidden parameter:50,20
Final val_accuracy:82.46%
Final test_accuracy:82.5%
Training time:192.345s

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.81 | 0.69 | 0.75 | 107 |
| 1 | 0.96 | 0.95 | 0.96 | 105 |
| 2 | 0.74 | 0.73 | 0.74 | 111 |
| 3 | 0.81 | 0.75 | 0.78 | 93 |
| 4 | 0.79 | 0.70 | 0.74 | 115 |
| 5 | 0.93 | 0.97 | 0.95 | 87 |
| 6 | 0.49 | 0.70 | 0.58 | 97 |
| 7 | 0.93 | 0.97 | 0.95 | 95 |
| 8 | 0.98 | 0.94 | 0.96 | 95 |
| 9 | 0.98 | 0.91 | 0.94 | 95 |
| accuracy | | | 0.82 | 1000 |
| macro avg | 0.84 | 0.83 | 0.83 | 1000 |
| weighted avg | 0.84 | 0.82 | 0.83 | 1000 |

Same like from the normal pytorch implementation, the class 6 gets the lowest accuracy of 58%
This is the classification report generated from the skorch implementation.

In Skorch, sklearn methodology(.fit) method is implemented which is different from pytorch, but can be used if you prefer sklearn much more than pytorch.

More Comparative Analysis:




This 2D line diagram shows how the trends of hidden nodes and training time are related.

Appendix:

1. For the binary classification we can choose the output=2 or 1 depending upon the loss function you define,
   For output=2, Crossentropyloss is fine, but for output=1, BCE(Binary cross entropy) loss function should be defined. Sigmoid is suitable for output=1, whereas for output=2, relu is better.
   Source:
   https://stackoverflow.com/questions/53628622/loss-function-its-inputs-for-binary-classification-pytorch

2. Both implementations are different and all of it I've done on my own with occasional discussion with my peers, rest all is of my implementation without copying from anyone, adhering to the University's policy.