1. **Consider the following training set. If we are using a linear regression model. What would the final model be after fitting it on this data?**

x	y
1	0.5
2	1
4	2
0	0

Answer: Since we are using linear regression, we will be having the model in the form of a line with the following equation: Y = aX + b. Here if we carefully look at the points, we would result in the following values for a and b.

$$\mathbf{a} = \frac{1}{2}\ \textit{and}\ \mathbf{b} = 0 \tag{1}$$

Hence the final model after fitting on this data would be $\mathbf{y} = \frac{\mathbf{x}}{\mathbf{2}}$.

2. **What is K-Means. How is it different from Meanshift?**
Answer: K-means is an algorithm that tries to partition data into K (already defined) non-overlapping subgroups. Each data point belongs to exactly one subgroup. The goal is to make the intra subgroup data points as similar as possible and keep the subgroups as far apart as possible. The algorithm is as follows:

   (a) Fix a value of K
   (b) Shuffle the data and Initialize K centroids by randomly choosing K data points from this shuffled data.
   (c) Compute the squared distance between data point from all the centroids and assign the data point to the cluster/centroid having the minimum distance.
   (d) Compute the updated centroids by taking the mean of all the data points that belong to a particular cluster.
   (e) Keep repeating steps (c)-(e) until there is no change to centroids (that is all the data points to clusters don't change)

Meanshift partitions is based on the concept of Kernal Density function and hence is knows to be a mode seeking algorithm. Initially consider a kernal at center C and having radius R. Now this kernal keeps on shifting to the high density region of the points in its viscinity. Hence, it groups the datapoints into clusters such that each point is within a certain radius from the centroid of the group.

Hence the primary differences between mean shift and K-means is that unlike K means, mean shift doesn't require the necessity to predfine the number of clusters, K. Besides that, meanshift returns multiple levels of centroids as it primarily uses the radius of the centroid to group datapoints. Higher the radius, lower would be the number of clusters.

3. **What is softmax? Under what cases does softmax computation become unstable?**
Answer: Softmax is an activation function that converts a given sequence of numbers (also known as logits in ML) to probability distributions that sum up to 1. Softmax is given by the following equation:

$$S(y_j) = \frac{e^{y_j}}{\sum_j e^{y_j}} \tag{2}$$

For example: If $x = \{1, 3, 2\} => S(x) = \{0.09, 0.67, 0.24\}$ here we can see that sum of S(x) is 1.

Softmax computations can become unstable for very large and very small valued elements in x. Overflow occurs when atleast one value in x is approximated as ∞. While underflow occurs when all values in x are near $-\infty$, thus resulting in an almost 0 denominator. A common technique to combat these issues is to subtract the maximum element of the array x (max(x)) from all the elements, which leaves a vector that has only non-positive entries (entries in range $(-\infty, 0)$, ruling out overflow and besides that atleast 1 element (the maximum element) will be zero which rules out a vanishing denominator (underflow only occurs when all entries tend towands negative infinity).

4. **What is regularization in CNNs? Explain 1 common technique used for regularization.**
   Answer: Regularization in CNNs is a method to control the model complexity due to the large number of parameters that are present in the model. Sometimes, when we train a CNN say we train it to detect cars using a particular dataset, what happens is when we try to test this model in a completely different setting compared to our dataset, we will see that the model isn't able to detect a car with good accuracy. Hence, we say that the model is overfitting on the training set but is not generalizing on the test set. Hence, we use regularization to solve this problem in CNNs.

   A few common techniques for regularization in CNN are:

   (a) **Dropout:** Dropout is a technique that is widely used for regularization in CNNs. When a dropout is applied to a layer, the neurons present in the layer are randomly dropped out with a probability p. The dropout layer in pytorch takes as parameter probability p, which is the probability of an element to be zeroed.
   (b) **L1 Regularization:** In L1 regularization we take the absolute value of all the parameters and sum them up. We have a constant (λ) which controls the amount of the regularization. Equation: $\lambda \sum_i \theta_i$ here λ is a constant which is initialized beforehand and θ_i represents the parameters.
   (c) **L2 Regularization:** L2 regularization is similar to L1 regularization, however, here, we take the square of the parameters and sum them up. Again, we have a constant λ which controls the amount of the regularization. Equation: $\lambda \sum_i \theta_i^2$ here λ is a constant which is initialized beforehand and θ_i represents the parameters.

   Here, L2 regularization can be used in Pytorch by using the parameter "weight_decay" in the optimizer.

5. **What is a GAN? What does it mean for a GAN to 'mode collapse'?**
   Answer: GAN stands for Generative Adversarial Network. GAN is a deep learning, unsupervised Machine Learning technique. It consists of a generator, whose objective is to generate data and a discriminator whose objective is to identify if the generated data is real or fake. Thus, the generator tries to maximize the probability of a fake image while the discriminator tries to maximize its accuracy in predicting whether an image is real or fake.

   Sometimes, instead of producing a wide range of outputs, a generator might produce a certain output (or a small set of outputs) that seems plausible to the discriminator. Now, if the generator starts producing that particular output (or set of outputs) again and again, then the discriminator should learn to, ideally, reject it. However, if the next generation of the discriminator doesn't find this best strategy the next generation of the generator will find the most plausible output for the current discriminator.Here, the disciminator gets stuck in a local minima and the discriminator weights improve only slightly. As a result, the generator rotates through a small set of output images thus finding an optimization shortcut. This type of failure in GAN is called “mode collapse”.

6. **Describe how you would compute PCA for a set of feature data points?**
   Answer: PCA stands for Principal Component Analysis. To compute the PCA for a set of feature points, we would perform the following procedure. Lets say we have a dataset D which consists of n samples having d number of features. Now to perform the PCA on this dataset D, we follow the following steps.
   (a) Compute the mean of every dimension in the dataset. This would give us a vector, $\vec{X} = \{\vec{\mu_1}, \vec{\mu_2}, ..., \vec{\mu_d}\}$ representing the means of each dimention.
   (b) Next, compute the covariance matrix of the entire dataset. Lets say the covariance matrix is C, then each element C_{ij} represents the measure of cavariance between the features (x_i, x_j). This is given by the following equation:
   $$C_{ij} = Covariance(x_i, x_j) = \sum_{i=1}^{n} \frac{(x_i - \mu_i)(x_j - \mu_j)}{n-1} \tag{3}$$
   (c) Next, we compute the eigenvectors and the corresponding eigenvalues by solving the equation: $C\vec{v} = \lambda\vec{v}$
   (d) In the above equation, the vectors, $\vec{v}$, represent the eigenvectors and λ represent the eigenvalues. Here eigenvectors form the principal components while eigenvalues represent the magnitude of variance in the dataset corresponding to that particular eigenvector.
   (e) Finally, we select the number of principal components (eigenvectors) we need, say c, correponding to the c largest eigenvalues present in the dataset.

7. **What are some common initialization methods for Convolution layers and Fully Connected layers in a CNN?**
   Answer: One of the primary reasons for initilizing weights for a Deep Neural Network is to prevent the layer activation outputs from vanishing or overflowing druing the forward pass throught the network. Few of the common weight initiliazation methods for a Deep neural network are:
   - **Xavier Initialization:** Xavier Initialization sets the weights of a layer to values that are chosen randomly from a uniform distribution thats bounder in the range $[-\frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}, +\frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}]$, where n_i are the number of incoming network connections (also know as fan-in) and n_{i+1} are the number of outgoing network connections (also known as fan-out). Xavier initialization mantains the variance of activations and back-propagated gradients all the way up or down the layers of a network.
     Reference: "Understanding the difficulty of training deep feedforward neural networks." by Glorot, Xavier, and Yoshua Bengio.
   - **Kaiming Initialization:** Kaiming initialization was motivated by exploration into how to best initialize weights in networks with ReLU-like activations. Kaiming He, in his paper, demonstrated that deep networks would converge much earlier if the following weight initialization techniques was used:

(a) Create a tensor with the dimensions appropriate for a weight matrix for a layer, and populate it with numbers randomly chosen from a standard normal distribution.

(b) Multiply each randomly chosen number by $\frac{\sqrt{2}}{\sqrt{n}}$ where n is the number of incoming connections coming into a given layer from the previous layer's output.

(c) Bias tensors are initialized to 0.

The paper also verifies that the above stratergy prevents the activation outputs to vanish or explode even if RELU is used at all the layers.
Reference: "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." by He, Kaiming, et al.

8. **One interesting application of machine learning is in the area of medical diagnoses. Implement Neural Network in python to classify the data into Benign or Malignant for the Wisconsin Diagnostic Breast Cancer (WDBC) dataset. Attach the code and the output. Things to keep in mind:**

   - **How will you choose the features?**
     Answer: Here, in the task we are supposed to try out various techniques to classify whether a tumor is benign or malignant. In our case, we are using a Neural Network model to accomplish this task. Now, a neural network will itself determine what features it should give more weight to and which it shouldn't while figuring out the activation value for the neurons in the first layer. And subsequently, it will use these weights of the first layer to determine the activations for the further layers. Hence, we do not need feature selection.
     Also, if we were to do feature selection we can use Random Forest classifier to know which features to give the most weight to. However, as we can see form our experiments, we can easily gain a decent accuracy on the test set by just using a simple 2 layer neural network.

   - **How to overcome overfitting?**
     Answer: Overfitting was something that was an issue in this task as the dataset is too small. Without using any regularization techniques, I was achieving a very low accuracy of (66%). However, using an L2 regularization techniques improved the accuracy (and reduced the losses) many folds! The various loss and accuracy plots (attached further in the answer for Q8) show us how accuracies and losses tend to change given the different weight decay values.

   - **How many layers should be there?**
     Answer: Here, I used a baseline model of Logistic Regression which gives an accuracy of 95.67%. I then started creating shallow neural networks with varying layers. Here, I tested neural networks having 2 and 3 hidden layers. The below table shows the accuray and losses on the test set for the best model among the 2 and 3 layer neural network. As we can see that there is hardly any difference among the values for either of them, hence I concluded that a 2 layer neural network is sufficient to achieve decent results on this dataset. Also, we can see that the neural network performs just slightly better than the Logistic Regression model hence we can conclude that the dataset is quite simple and hence doesn't require the use of complex models.

Parameters	2 layer	3 layer
Test Accuracy	98.21	98.28
Test Loss	0.0943	0.0772

Please attach training curves, final losses and visualizations of errors made by the trained model, along with a description of the methods/tricks you tried to improve performance.

- ***Data Exploration (PCA)***

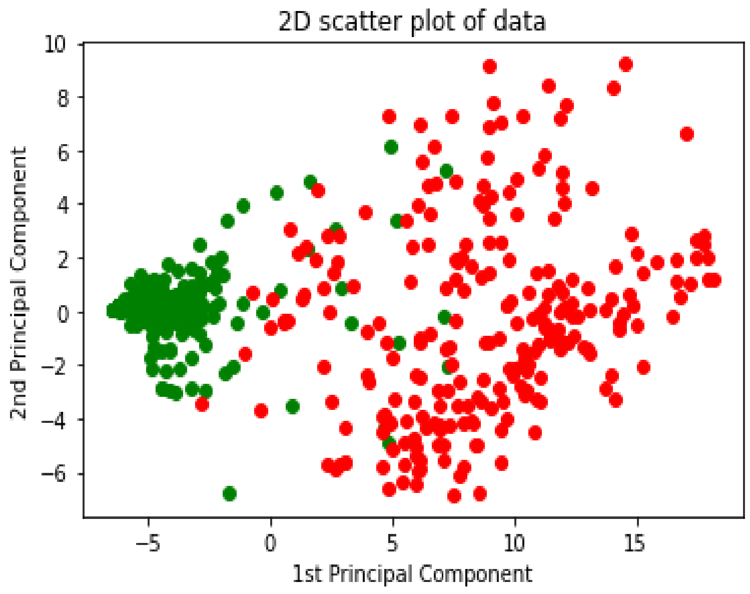


Figure 1: PCA on the dataset with number of principal components = 2

- ***Training Losses***

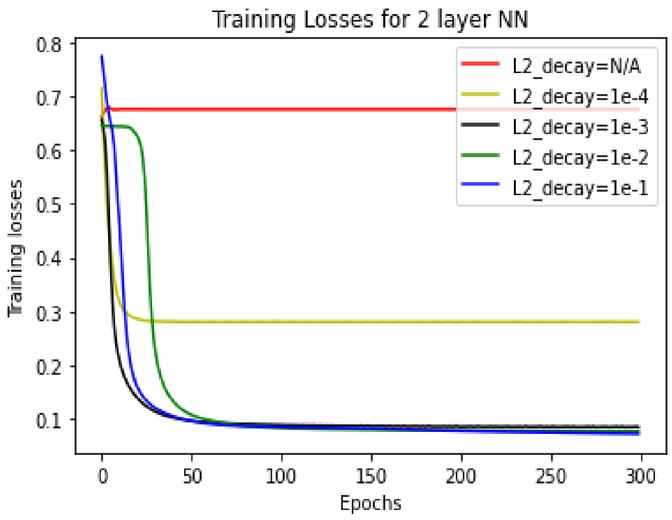


Figure 2: Training Loss for 2 Layer Neural Network

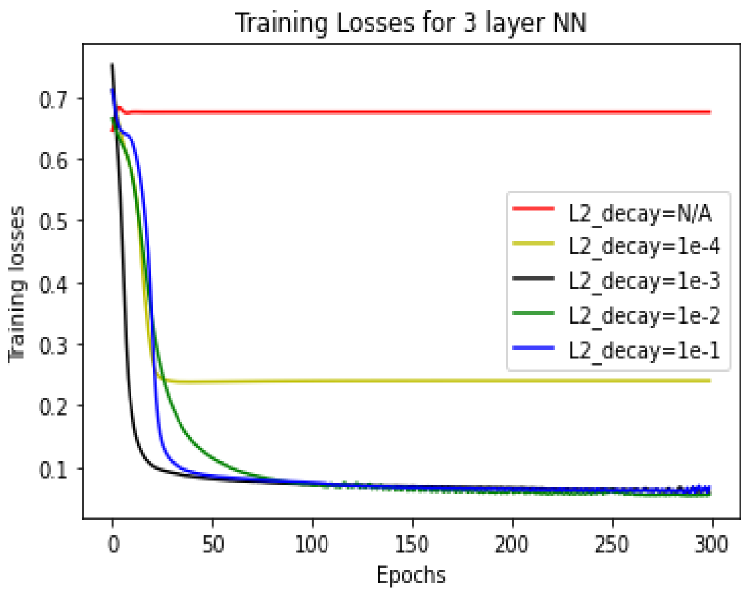


Figure 3: Training Loss for 3 Layer Neural Network

- ***Validation Losses***

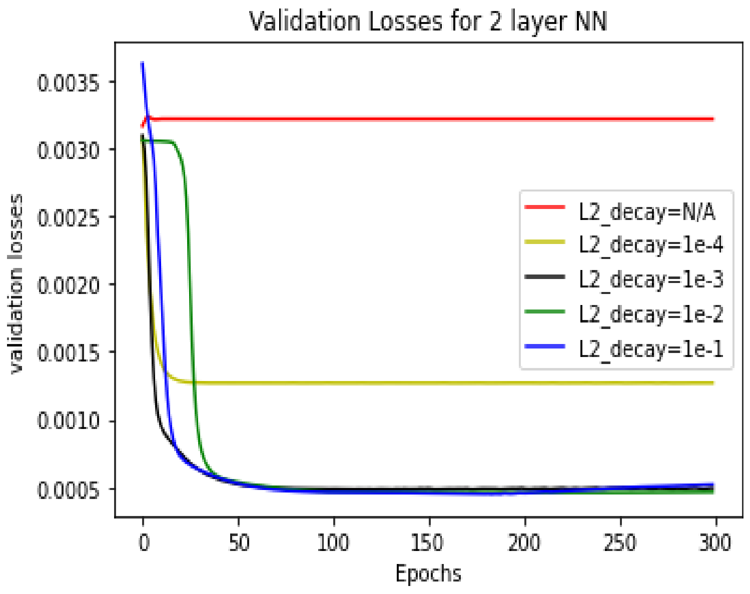


Figure 4: Validation Loss for 2 Layer Neural Network

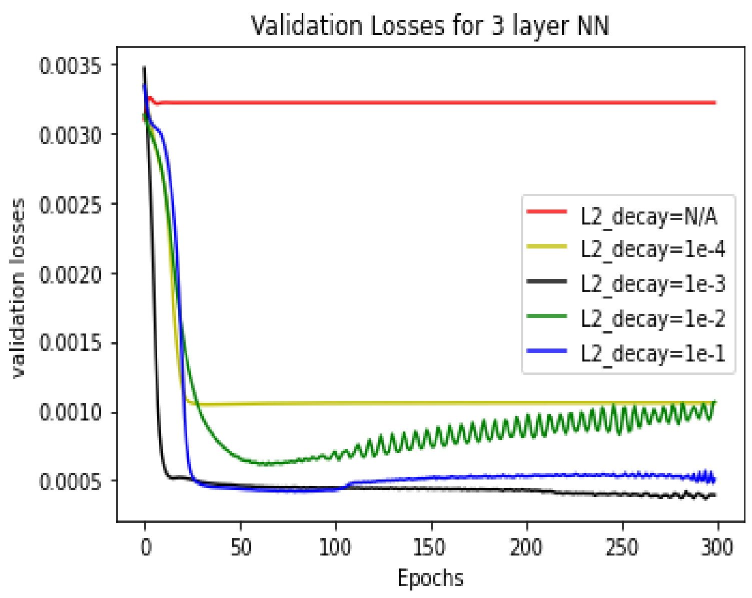


Figure 5: Validation Loss for 3 Layer Neural Network

- ***Validation Accuracy***

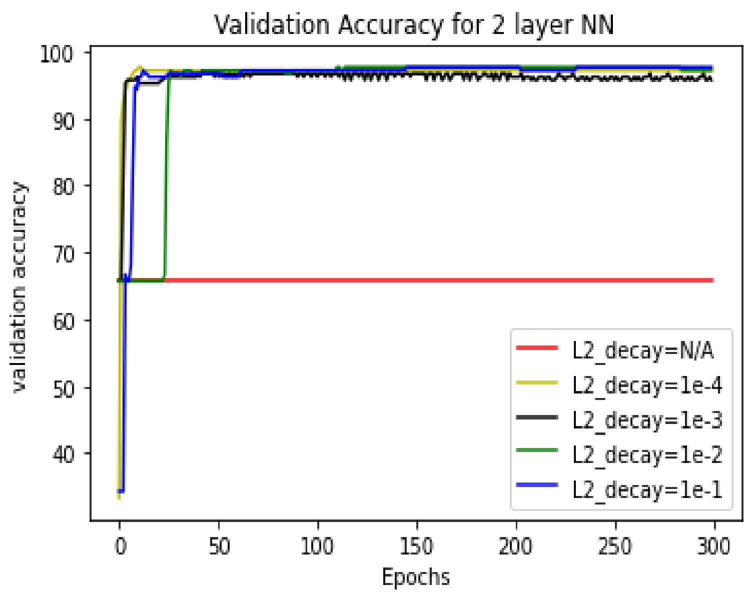


Figure 6: Validation Accuracy for 2 Layer Neural Network

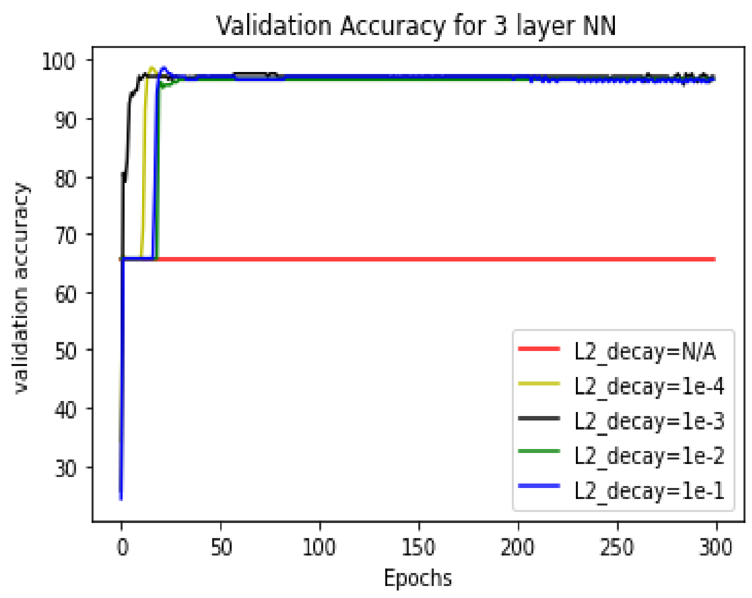
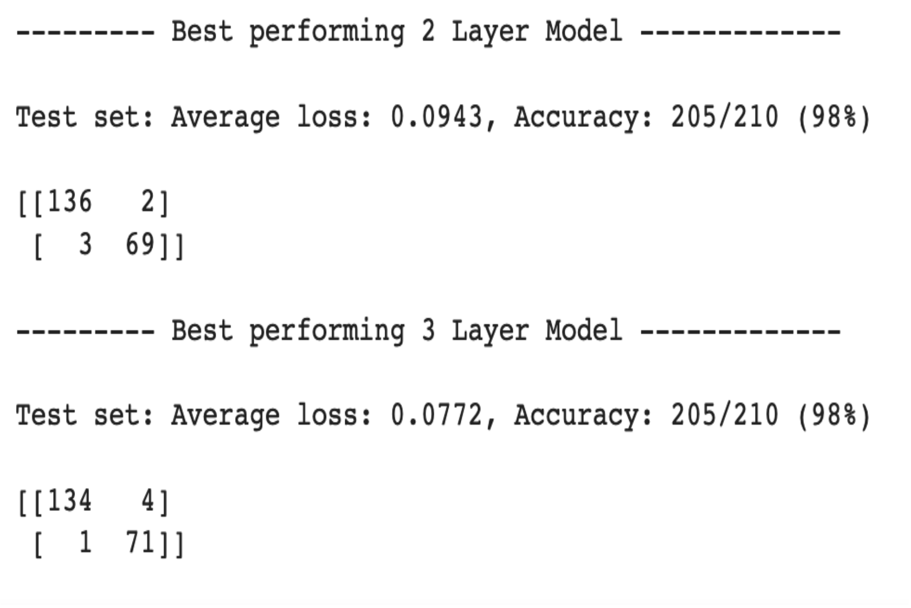


Figure 7: Validation Accuracy for 3 Layer Neural Network

- ***Confusion Matrices***



```
--------- Best performing 2 Layer Model -------------

Test set: Average loss: 0.0943, Accuracy: 205/210 (98%)

[[136   2]
 [  3  69]]

--------- Best performing 3 Layer Model -------------

Test set: Average loss: 0.0772, Accuracy: 205/210 (98%)

[[134   4]
 [  1  71]]
```

Figure 8: Confusion matrices for best performing models

The first task was to handle the missing values in the dataset. I filled in the missing values by taking the median of the values for that particular feature. Next step was to try out several models on this updated dataset. As one can see, several techniques were tried out to improve the model performance. Initially, a baseline model was created which is a Logistic Regression model that achieved an accuracy of around 95%. Once I had my baseline model, I started training a simple 2 layer neural network. I observed that even with the simple neural network, there was an issue of overfitting. Hence, I started using weight decays. Each of the above training and validation losses and accuracy plots are with respect to all the used weight decays. Besides that, I also experimented with a 3 layer Neural Network to see how increasing the number of layers affect the performance. There was just a slight improvement so I didn't try out more number of layers. The experiments ran were compatible with my intuition that since this is a small and fairly simple dataset even the simplest of models would work well and hence we won't have to try deep models!

Q8_RL

September 17, 2020

[167]:
```
import pandas as pd
import torch
import numpy as np

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
import cohttp://localhost:8888/notebooks/Desktop/Semester3/DeepRL/Assignment0/
 ↪Q8_RL.ipynb#py

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA
from torch.utils.data import TensorDataset, DataLoader
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
```

[168]:
```
df = pd.read_csv('/content/drive/My Drive/Semester3/DRL/Assignment0/
 ↪breast-cancer-wisconsin.csv', header = None)
```

0.0.1 The original dataframe after reading the csv file.

[169]:
```
df.head()
```

[169]:
```
         0   1   2   3   4   5   6   7   8   9   10
0  1000025   5   1   1   1   2   1   3   1   1   2
1  1002945   5   4   4   5   7  10   3   2   1   2
2  1015425   3   1   1   1   2   2   3   1   1   2
3  1016277   6   8   8   1   3   4   3   7   1   2
4  1017023   4   1   1   3   2   1   3   1   1   2
```

###Dropping the ID column as it is not a feature and shuffling the data randomly.

[170]:
```
df = df.drop(columns = [0])
#shuffling the rows randomly
```

```
df = df.sample(frac = 1)
```

0.0.2 Taking the class labels and storing them in a numpy array. Converting the class labels to 0,1 for clarity.

```
[171]: Y = np.array(df[10], dtype = 'float32')
       Y[Y == 4] = 1.0
       Y[Y == 2] = 0.0
```

###Comparing the number of samples corresponding to each class.

```
[172]: print(f"Number of datarows corresponding to benign tumor: {np.count_nonzero(Y␣
        ↪== 0)}")
       print(f"Number of datarows corresponding to malignant tumor: {np.
        ↪count_nonzero(Y == 1)}")
```

```
Number of datarows corresponding to benign tumor: 458
Number of datarows corresponding to malignant tumor: 241
```

As we can see here, there is enough data for each category to perform a proper classification task.

###Now, as we can notice, there are some data points which have some missing features (denoted by '?') so the best approach to replace those '?' is by putting the median of the features.

```
[173]: df = df.drop(columns = [10])
       data_columns = [i for i in range(1,10)]
       print(data_columns)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[174]: # Swap out missing data with median for that column
       for feature_name in data_columns:
           df[feature_name][df[feature_name] == '?'] = None
           feature_median_val = df[feature_name].median(skipna=True)
           df[feature_name] = df[feature_name].fillna(feature_median_val)
```

```
/usr/local/lib/python3.6/dist-packages/pandas/core/ops/array_ops.py:253:
FutureWarning: elementwise comparison failed; returning scalar instead, but in
the future will perform elementwise comparison
  res_values = method(rvalues)
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  This is separate from the ipykernel package so we can avoid doing imports
until
```

###Now, lets convert the dataframe containing the datapoints to a Numpy array.

[175]:
```
X = np.array(df[data_columns],dtype='float32')
```

Q) How do we choose the features to train the model on?

A) Here, in the task we are supposed to try out various techniques to classify whether a tumor is benign or malignant. In our case, we are using a Neural Network model to accomplish this task. Now, a neural network will itself determine what features it should give more weight to and which it shouldn't while figuring out the activation value for the neurons in the first layer. And subsequently, it will use these weights of the first layer to determine the activations for the further layers. Hence, we do not need feature selection.

Also, if we were to do feature selection we can use Random Forest classifier to know which features to give the most weight to. However, as we can see form our experiments, we can easily gain a decent accuracy on the test set by just using a simple 2 layer neural network.

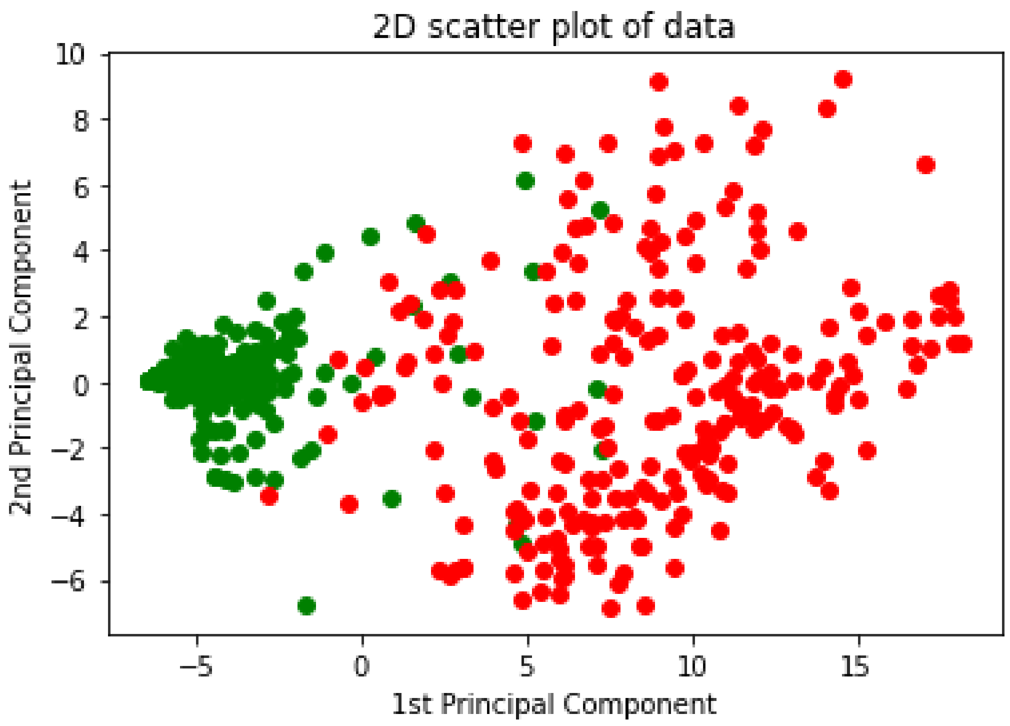
[176]:
```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,␣
 ↪stratify=Y,test_size=0.3,random_state=1)
```

[177]:
```
print(X.shape, Y.shape)
```

```
(699, 9) (699,)
```

###Lets try to visualize the data in 2D place by performing PCA and choosing the eigenvectors corresponding to the largest 2 eigenvalues.

[178]:
```
pca_obj = PCA(n_components=2)
pca_obj.fit(X)
vector_dim_2 = pca_obj.transform(X)

plt.title("2D scatter plot of data")
x = vector_dim_2[Y == 0]
plt.scatter(x[:, 0], x[:, 1], label="Benign", color = 'g')
x = vector_dim_2[Y == 1]
plt.scatter(x[:, 0], x[:, 1], label="Malignant", color = 'r')

plt.xlabel("1st Principal Component")
plt.ylabel("2nd Principal Component")
plt.show()
```



###Lets run a Logistic Regression model as our baseline model.

```
[179]: clf = LogisticRegression(penalty='l2', random_state=1, max_iter = 3000)
       clf.fit(X_train, Y_train)
```

```
[179]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                          intercept_scaling=1, l1_ratio=None, max_iter=3000,
                          multi_class='auto', n_jobs=None, penalty='l2',
                          random_state=1, solver='lbfgs', tol=0.0001, verbose=0,
                          warm_start=False)
```

```
[180]: Y_predicted = clf.predict(X_test)
       accuracy_score(Y_predicted, Y_test)
```

```
[180]: 0.9714285714285714
```

###Lets convert the data into Pytorch tensors.

```
[181]: X_train_tensors = torch.Tensor(X_train)
       Y_train_tensors = torch.Tensor(Y_train)
       X_test_tensors = torch.Tensor(X_test)
       Y_test_tensors = torch.Tensor(Y_test)
```

[182]:
```
training_data = TensorDataset(X_train_tensors, Y_train_tensors)
testing_data = TensorDataset(X_test_tensors, Y_test_tensors)
```

[183]:
```
train_data_loader = DataLoader(training_data, batch_size=64, num_workers=2)
test_data_loader = DataLoader(testing_data, batch_size=64, num_workers=2)
```

###Lets define a NN having 2 hidden layers.

[184]:
```
class TwoLayerNeuralNetwork(nn.Module):
    def __init__(self, input_size, n_hidden, output_size):
        super(TwoLayerNeuralNetwork, self).__init__()
        self.input_size = input_size
        self.network = nn.Sequential(
            nn.Linear(input_size, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, output_size),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = x.view(-1, self.input_size)
        return self.network(x)
```

[185]:
```
class ThreeLayerNeuralNetwork(nn.Module):
    def __init__(self, input_size, n_hidden, output_size):
        super(ThreeLayerNeuralNetwork, self).__init__()
        self.input_size = input_size
        self.network = nn.Sequential(
            nn.Linear(input_size, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, output_size),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = x.view(-1, self.input_size)
        return self.network(x)
```

[186]:
```
device = "cpu"#torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

[187]:
```python
def train(epoch, model):
    model.train()
    train_loss = 0.0
    for batch_idx, (data, target) in enumerate(train_data_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        output = output.squeeze(1)
        loss = F.binary_cross_entropy(output, target)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()*len(data)
    return train_loss/len(train_data_loader.dataset)

def validate(model):
    model.eval()
    val_loss = 0.0
    correct = 0
    for data, target in test_data_loader:
        # send to device
        data, target = data.to(device), target.to(device)
        output = model(data)
        output = output.squeeze(1)
        val_loss += F.binary_cross_entropy(output, target, reduction='sum').
 ↪item()
        pred = torch.round(output)
        correct += pred.eq(target.data.view_as(pred)).cpu().sum().item()
    val_loss /= len(test_data_loader.dataset)
    accuracy = 100. * correct / len(test_data_loader.dataset)
    return accuracy, val_loss/len(test_data_loader.dataset)

def test(model):
    predlist=torch.zeros(0,dtype=torch.long, device='cpu')
    lbllist=torch.zeros(0,dtype=torch.long, device='cpu')
    test_loss = 0
    correct = 0
    for data, target in test_data_loader:
        # send to device
        #print(data)
        data, target = data.to(device), target.to(device)
        output = model(data)
        output = output.squeeze(1)
        test_loss += F.binary_cross_entropy(output, target, reduction='sum').
 ↪item() # sum up batch loss                                              ␣
 ↪
        pred = torch.round(output)
        #print(pred, target)                                                    ␣
↪
        correct += pred.eq(target.data.view_as(pred)).cpu().sum().item()
        predlist=torch.cat([predlist,pred.view(-1).cpu()])
        lbllist=torch.cat([lbllist,target.view(-1).cpu()])
        #print(len(predlist), len(lbllist))
    test_loss /= len(test_data_loader.dataset)
    accuracy = 100. * float(correct) / len(test_data_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.
↪format(
        test_loss, correct, len(test_data_loader.dataset),
        accuracy))
    return predlist, lbllist
```

[188]:
```
def plot_losses(train, val):
  N = len(train)
  X = np.arange(N)

  plt.title("Train/Val Loss over epochs")
  plt.xlabel("Epoch")
  plt.ylabel("Loss")

  plt.plot(X, train, label = "Training Loss")
  plt.plot(X, val, label = "Validation Loss")

  plt.legend()

  plt.show()
```

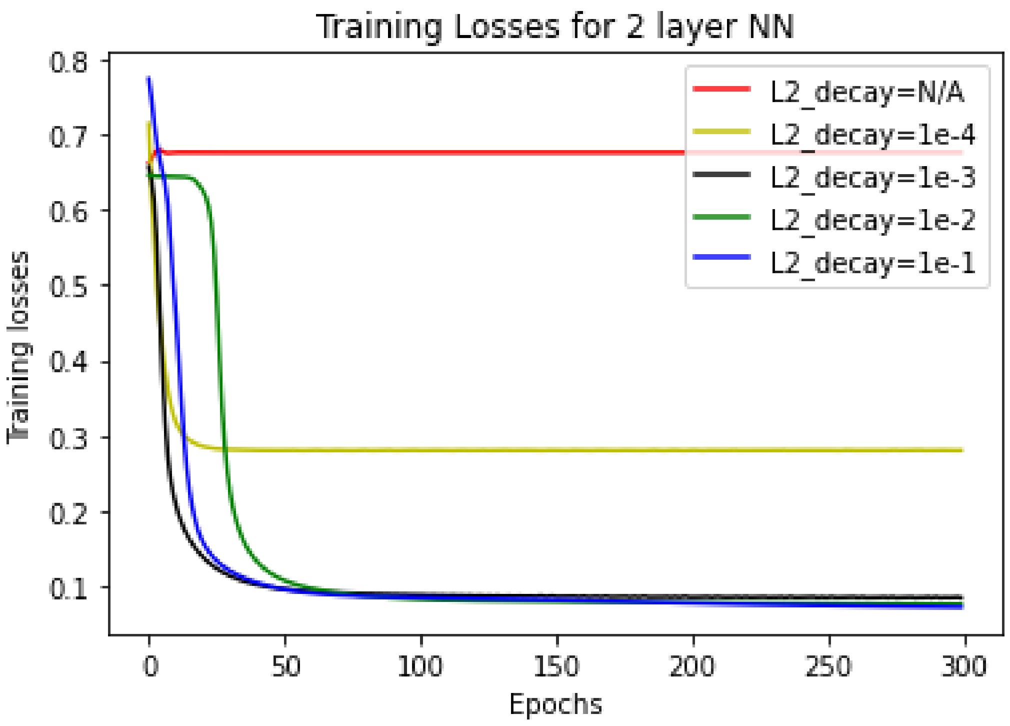
[189]:
```
def plot(no_epochs, dicto, plot_title, ylabel):
    epochs = range(no_epochs)
    plt.plot(epochs, dicto[0], 'r', label='L2_decay=N/A')
    plt.plot(epochs, dicto[1], 'y', label='L2_decay=1e-4')
    plt.plot(epochs, dicto[2], 'k', label='L2_decay=1e-3')
    plt.plot(epochs, dicto[3], 'g', label='L2_decay=1e-2')
    plt.plot(epochs, dicto[4], 'b', label='L2_decay=1e-1')
    plt.title(plot_title)
    plt.ylabel(ylabel)
    plt.xlabel('Epochs')

    plt.legend(loc='best')
    plt.show()
```

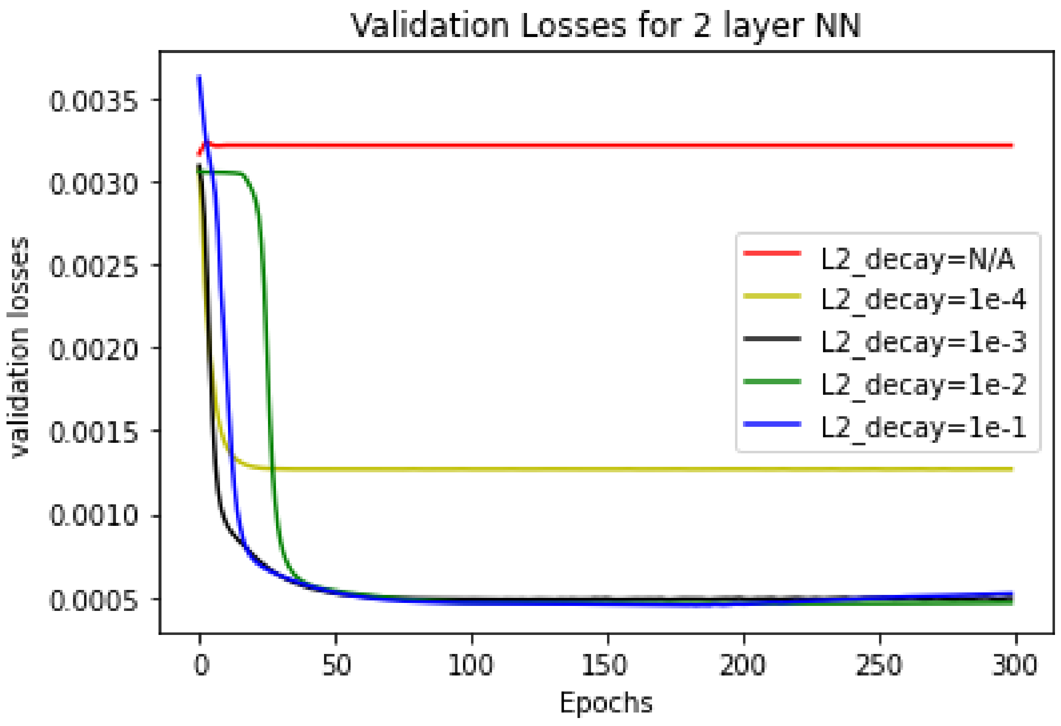
[190]:
```
print(device)
```

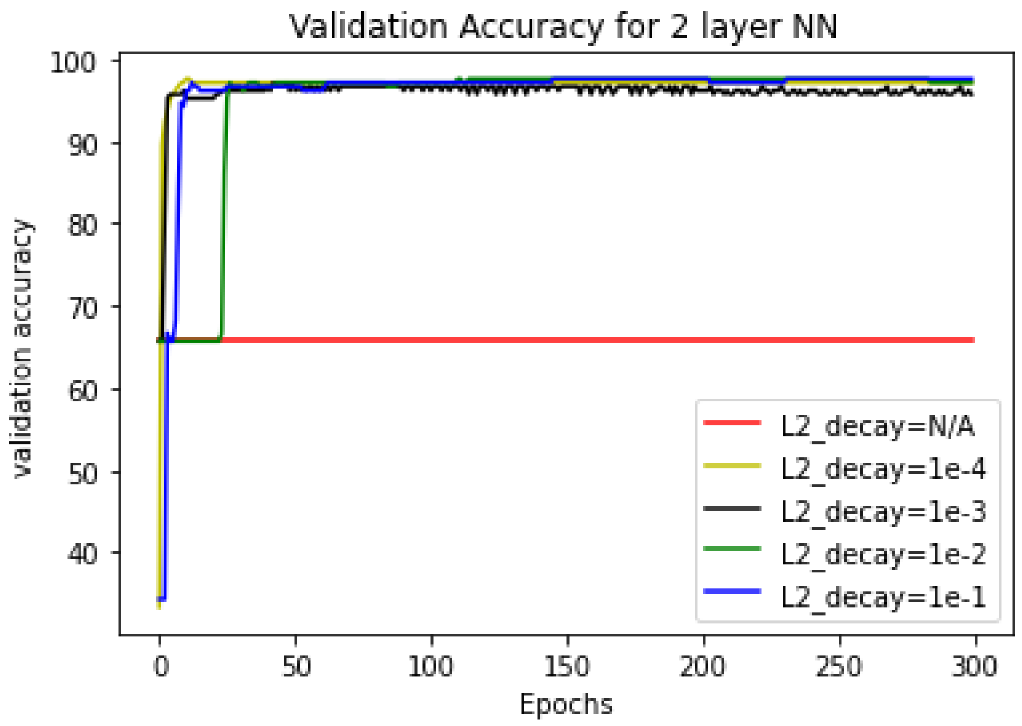
```
cpu
```

```
[191]: n_hidden = 5
input_size = 9
output_size = 1

best_2_layer_model_state = None
best_3_layer_model_state = None
best_2_layer_model_val_loss = float(10.)
best_3_layer_model_val_loss = float(10.)

print("-----------Plots for 2 Layer NN------------")

losses_dict = {}
training_dict = {}
acc_dict = {}
for i in range(0,5):
    model_fnn = TwoLayerNeuralNetwork(input_size, n_hidden, output_size)
    model_fnn.to(device)
    optimizer = optim.SGD(model_fnn.parameters(), lr=0.01, momentum=0.9,␣
 ↪weight_decay=1/pow(10,i))
    train_losses = []
    val_losses = []
    val_accuracy = []
    for epoch in range(0, 300):
        train_losses.append(train(epoch, model_fnn))
        val_acc, val_loss = validate(model_fnn)
        val_losses.append(val_loss)
        val_accuracy.append(val_acc)
        if val_loss < best_2_layer_model_val_loss:
            best_2_layer_model_val_loss = val_loss
            best_2_layer_model_state = copy.deepcopy(model_fnn.state_dict())
    losses_dict[i] = val_losses
    training_dict[i] = train_losses
    acc_dict[i] = val_accuracy
    print(f"Plot for weight decay : 1e-{i}")
    test(model_fnn)
plot(300,training_dict,"Training Losses for 2 layer NN", "Training losses")
plot(300,losses_dict,"Validation Losses for 2 layer NN", "validation losses")
plot(300,acc_dict,"Validation Accuracy for 2 layer NN", "validation accuracy")


print("-----------Plots for 3 Layer NN------------")

losses_dict = {}
training_dict = {}
acc_dict = {}
for i in range(0,5):
    model_fnn = ThreeLayerNeuralNetwork(input_size, n_hidden, output_size)
    model_fnn.to(device)
    optimizer = optim.SGD(model_fnn.parameters(), lr=0.01, momentum=0.9,␣
 ↪weight_decay=1/pow(10,i))
    train_losses = []
    val_losses = []
    val_accuracy = []
    for epoch in range(0, 300):
        train_losses.append(train(epoch, model_fnn))
        val_acc, val_loss = validate(model_fnn)
        val_losses.append(val_loss)
        val_accuracy.append(val_acc)
        if val_loss < best_3_layer_model_val_loss:
            best_3_layer_model_val_loss = val_loss
            best_3_layer_model_state = copy.deepcopy(model_fnn.state_dict())
    losses_dict[i] = val_losses
    training_dict[i] = train_losses
    acc_dict[i] = val_accuracy
    print(f"Plot for weight decay : 1e-{i}")
    test(model_fnn)
    #plot(train_losses, val_losses)
plot(300,training_dict,"Training Losses for 3 layer NN", "Training losses")
plot(300,losses_dict,"Validation Losses for 3 layer NN", "validation losses")
plot(300,acc_dict,"Validation Accuracy for 3 layer NN", "validation accuracy")
```

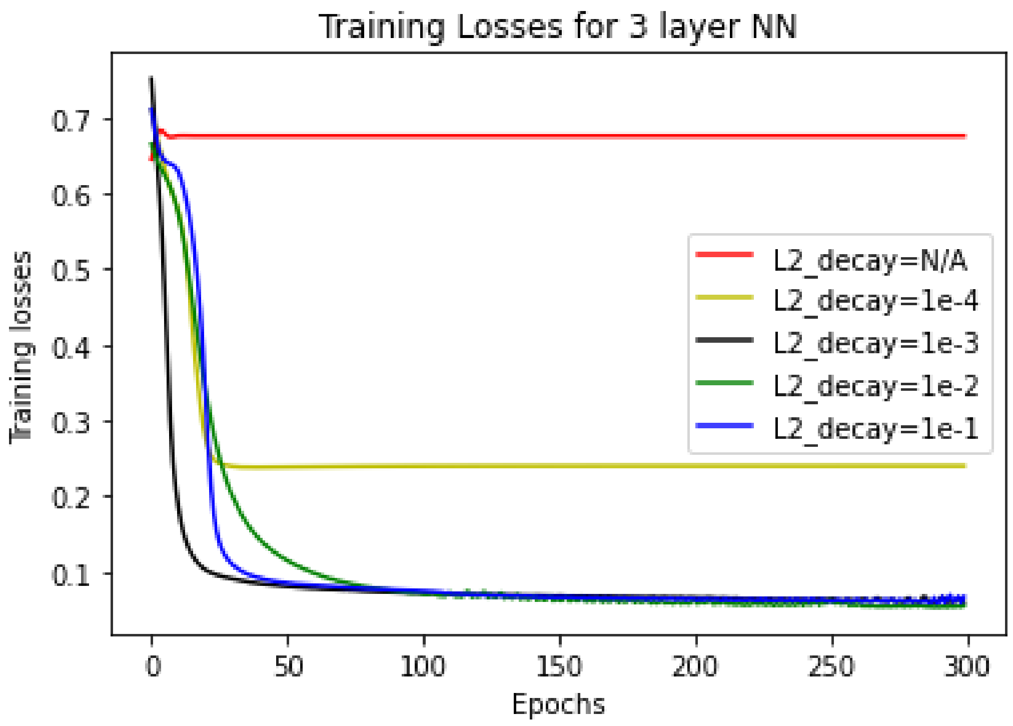
```
-----------Plots for 2 Layer NN------------
Plot for weight decay : 1e-0

Test set: Average loss: 0.6755, Accuracy: 138/210 (66%)

Plot for weight decay : 1e-1

Test set: Average loss: 0.2666, Accuracy: 204/210 (97%)

Plot for weight decay : 1e-2

Test set: Average loss: 0.1037, Accuracy: 201/210 (96%)

Plot for weight decay : 1e-3

Test set: Average loss: 0.0978, Accuracy: 204/210 (97%)

Plot for weight decay : 1e-4

Test set: Average loss: 0.1094, Accuracy: 205/210 (98%)
```

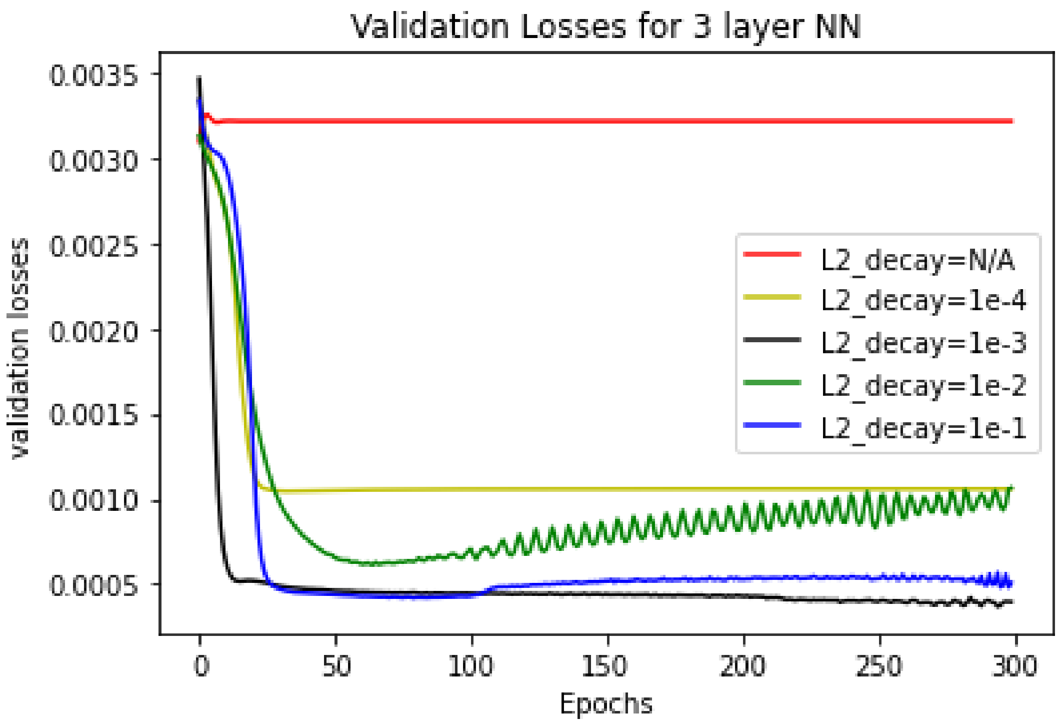

Training Losses for 2 layer NN
Training losses
Epochs
L2_decay=N/A
L2_decay=1e-4
L2_decay=1e-3
L2_decay=1e-2
L2_decay=1e-1
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
50
100
150
200
250
300

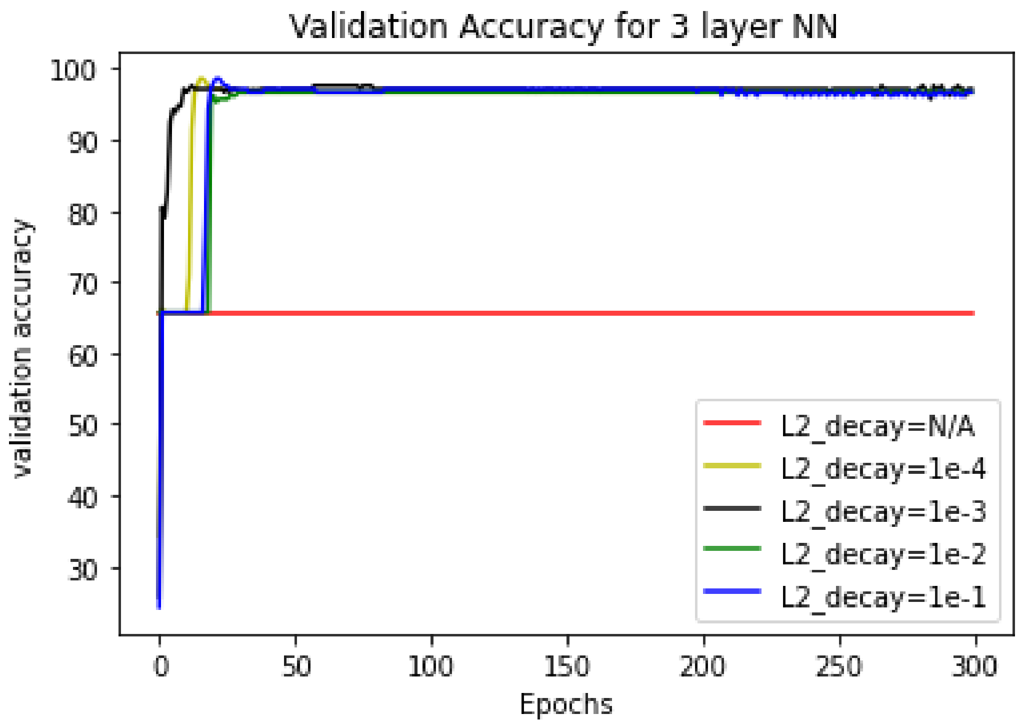

Validation Losses for 2 layer NN
validation losses
Epochs
L2_decay=N/A
L2_decay=1e-4
L2_decay=1e-3
L2_decay=1e-2
L2_decay=1e-1
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0
50
100
150
200
250
300



```
-----------Plots for 3 Layer NN------------
Plot for weight decay : 1e-0

Test set: Average loss: 0.6755, Accuracy: 138/210 (66%)

Plot for weight decay : 1e-1

Test set: Average loss: 0.2223, Accuracy: 203/210 (97%)

Plot for weight decay : 1e-2

Test set: Average loss: 0.0831, Accuracy: 204/210 (97%)

Plot for weight decay : 1e-3

Test set: Average loss: 0.2236, Accuracy: 203/210 (97%)

Plot for weight decay : 1e-4

Test set: Average loss: 0.1065, Accuracy: 203/210 (97%)
```


Training Losses for 3 layer NN
Training losses
Epochs
L2_decay=N/A
L2_decay=1e-4
L2_decay=1e-3
L2_decay=1e-2
L2_decay=1e-1


Validation Losses for 3 layer NN
validation losses
Epochs
L2_decay=N/A
L2_decay=1e-4
L2_decay=1e-3
L2_decay=1e-2
L2_decay=1e-1



Q)How to overcome overfitting?

A) Overfitting was something that was an issue in this task as the dataset is too small. Without using any regularization techniques, I was achieving a very low accuracy of (66%). However, using an L2 regularization techniques improved the accuracy (and reduced the losses) many folds! The various loss and accuracy plots show us how accuracies and losses tend to change given the different weight decay values.

[193]:
```
torch.save(best_2_layer_model_state, "/content/drive/My Drive/Semester3/DRL/
↪Assignment0/best_2_layer_model.pth")
torch.save(best_3_layer_model_state, "/content/drive/My Drive/Semester3/DRL/
↪Assignment0/best_3_layer_model.pth")
```

[195]:
```
print("--------- Best performing 2 Layer Model -------------")
state = torch.load("/content/drive/My Drive/Semester3/DRL/Assignment0/
↪best_2_layer_model.pth")
model_best_2_layer = TwoLayerNeuralNetwork(input_size, n_hidden, output_size)
model_best_2_layer.load_state_dict(state)
predlist,lbllist = test(model_best_2_layer)
conf_mat=confusion_matrix(lbllist.detach().numpy(), predlist.detach().numpy())
print(conf_mat)
print()
print("--------- Best performing 3 Layer Model -------------")
```

```
state = torch.load("/content/drive/My Drive/Semester3/DRL/Assignment0/
 ↪best_3_layer_model.pth")
model_best_3_layer = ThreeLayerNeuralNetwork(input_size, n_hidden, output_size)
model_best_3_layer.load_state_dict(state)
predlist,lbllist = test(model_best_3_layer)
conf_mat=confusion_matrix(lbllist.detach().numpy(), predlist.detach().numpy())
print(conf_mat)
```

```
--------- Best performing 2 Layer Model -------------

Test set: Average loss: 0.0943, Accuracy: 205/210 (98%)

[[136   2]
 [  3  69]]

--------- Best performing 3 Layer Model -------------

Test set: Average loss: 0.0772, Accuracy: 205/210 (98%)

[[134   4]
 [  1  71]]
```

How many layers should be there?

A) Here, I used a baseline model of Logistic Regression which gives an accuracy of 97.14%. I then started creating shallow neural networks with varying layers. Here, I tested neural networks having 2 and 3 hidden layers. The below table shows the accuray and losses on the test set for the best model among the 2 and 3 layer neural network. As we can see that there is hardly any difference among the values for either of them, hence I concluded that a 2 layer neural network is sufficient to achieve decent results on this dataset. Also, we can see that the neural network performs just slightly better than the Logistic Regression model hence we can conclude that the dataset is quite simple and hence doesn't require the use of complex models.