PRNN Assignment - 2

1. Question 1

Here we have to use SVM on the classification problems using different kernels.

1.1. Implementation

- At first datasets for P1 is loaded into colab and datas are separated into train, validation and test sets.
- Then a grid search is done for each kernels respective hyperparameters to select the best hyperparameters(one giving highest accuracy).
- The best hyperparameter value is plugged in and one vs all SVM models are trained for each class.
- The test dataset is then tested on these models and accuracy noted.
- Then a multiclass svm is also built and its accuracy compared.
- The above steps are repeated for 3 kernels and for 2 classification questions.
- Here extra contextlib library is called to suppress the print statement printed by libsvm library.

1.2. Results

The accuracy of various models and questions are summarised in the below table Here Question refers to the classification question 1 and 2.Each question has 2 implementation type, one is multi class svm where all the classes are passed into libsvm and second type is one vs all where as many svms are built as there are class types. The last 3 columns of the following tables are different kernels which are tested. and each value represents percentage accuracy. The different kernels that we tested are gaussian, polynomial and linear kernels. Here the output or testing is done on 3 things:

Question	Gaussian	Polynomial	Linear
Multiclass(1)	58.94	57.85	56.50
One vs all(1)	56.7	57.4	56.88
Multiclass(2)	94.64	94.16	92.73
One vs all(2)	91.15	92.46	91.34

SVM is also built for ”without slack case” for gaussian kernel, and accuracy was found to be 44.87 in this case.

1.3. Observations

Libsvm library is used to train a svm classifier, each kernel of libsvm has variety of parameter that need to be choosen wisely. For instance in gaussian kernel one has to choose gamma and ’c’ where as in polynomial kernel one has to choose degree of polynomial, gamma and ’c’. ’c’ is the upper limit of lagrange multiplier.

For different combinations of hyperparameters it is observed that the accuracy in Q1(problem 3 of assignment 1) doesnot crosses 60 percent while accuracy of Q2 (problem 5 of assignment) is well above 90 percent This is probably because of the type of datas presented (in question 1) are overlapping and even the kernels are not able to separate them. Problem 4 and 5 of assignment 1 are the same classification problems so only problem 5 is choosen for Q2.

1.4. Problems Faced

Here one of the problem faced is the time to train the SVM, each svm training models takes around 30 minutes and we have to do a grid search on all the hyperparameters, then train the models for each class (one vs all). To eliminate this problem we set the ’-e’ parameter to 1 (increased the tolerance criterion for termination) as well as decreased the train and validation test data sets to a size of 20000 each. This makes the algorithm comparatively faster.

2. Question 2

Here our aim is to implement FLDA on classification problems of Assignment 1.

2.1. Implementation

We have implemented Fischer Linear Discriminant (FLD) in this question. We have used FLDA here for multiclass classification. First, we calculated the within and between class variance matrix, and then we calculate the eigenvalues of $S_w^{-1}S_B$. Then we project our data into the K-1 eigenvectors corresponding to the top K-1 eigenvalues, where K is the number of classes. Then after projecting the data onto a smaller space we used logistic regression as the classifier. This is applied for the two classification questions of assignment 1.

2.2. Results and Observations

In this question, we got an accuracy of 46.1 for the dataset given in question-3 in assignment-1. F1 scores for different classes are: [0.39, 0.43, 0.59, 0.56, 0.22] In this question, we got an accuracy of 89.81 for the dataset given in question-5 in assignment-1. F1 scores for different classes are: [0.89, 0.92, 0.96, 0.865, 0.92, 0.92, 0.86, 0.77, 0.92, 0.90]

We got low accuracy for the first case, which can be possible if the data is not linearly separable.

2.3. Problems faced

If the data is coming from overlapping distributions then FLDA may not be able to project the data such that it is linearly separable.

3. Question 3

Here we have to overfitt the given data and then apply different regularizers and visualise the change.

At first datasets for P3 is loaded into colab and datas are separated into train and test datasets Various models like bayes, logistic regressor are created and different metrices are computed.

3.1. Implementation

- First functions are written to calculate the optimum weights for L1 regularisation and regularisation with both L1 and L2.

- For only L2 regularisation and without any regularisation case we can directly use the following formulae to find the optimum weights. For unregularised case:

$$W = (A^T A)^{-1} A^T Y \tag{1}$$

  For L2 regularised case:

$$W = (A^T A + \lambda I)^{-1} A^T Y \tag{2}$$

  Where I is identity matrix and A is data matrix.

- Next functions are made to increase the dimensions of data by concatenating new features into dataset (to overfit). 3 functions are written one for polynomial model, one for exponential model and one for sin/cos model.

- For each of the above model first a high degree polynomial is overfitted (high variance low bias) into the data.

- Then with different regularised term (L1, L2, both L1 and L2) the bias is tried to brought down (by increasing the regularised term coefficient) so that the test error decreases and graphs are plotted.

3.2. Results and Observations

The complexity of the model (number of features) is increased and graph is plotted between complexity and train and test error. As in fig 1 the train error decreases and test error first decreases and then increases with increase in complexity. The reason why the decrease in test error is not that that prominent because the data given is linear data and hence best fit is obtained for linear models, and as we cannot go below linear features, hence with increase in complexity test error increases. The following is the bias - variance curve for polynomial model Similar graph is also

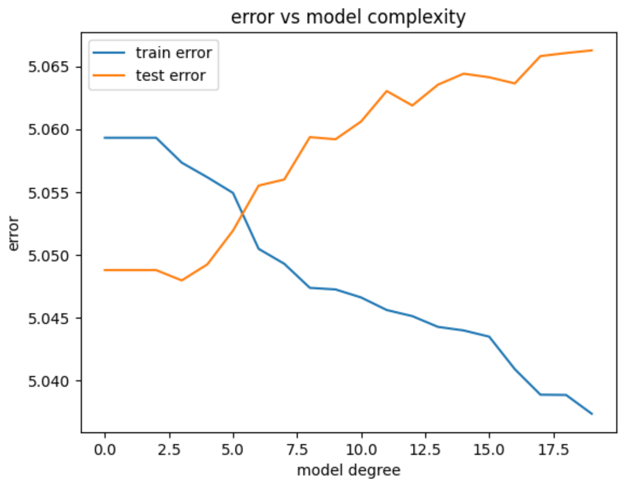

Figure 1

obtained for rest 2 models.

Now with fixing model complexity (fixed order of polynomial to be 20) the regularisation coefficient is increased in a step wise manner and for each step test error and train error is stored and later plotted. For sinusoid model we got the following curve(Fig 2 and 3) for L2 regularisation curve:

As expected the train error increases with increase in regularisation term and test error first reduces and then increases with regularisation term. The x axis is inverted so that to make the graph more clear. Clearly from graph the bias is increasing as we increase the regulariser coefficient and hence the variance is decreasing.

The above type of bias-variance curves are obtained for all the cases of L1, L2 and both L1-L2 regularisation for all the 3 type of models but only one is shown here for space constraints, do refer to colab code for more graphs.

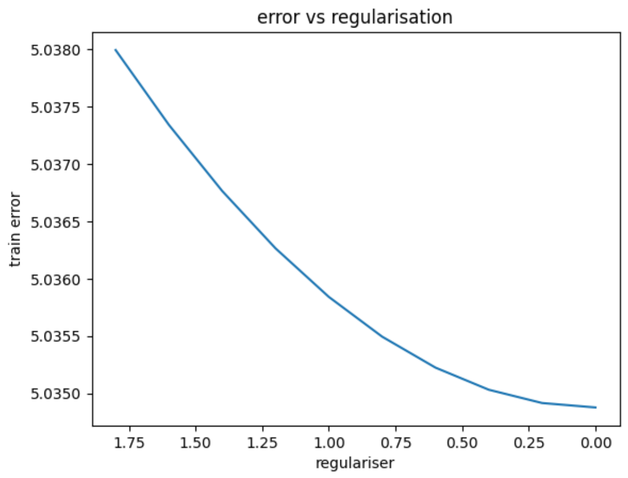

Figure 2. L2 reg(Sinusoid model)

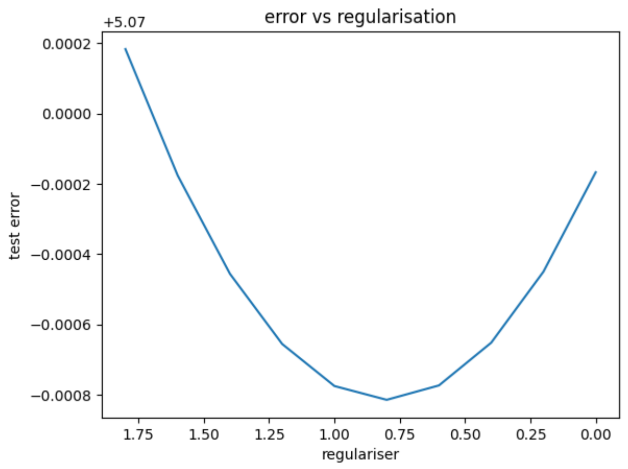

Figure 3. L2 reg (Sinusoid model)

4. Question 4

Here we have to built a neural network from scratch and use it on K-MNIST dataset and note accuracies for different models.

4.1. Implementation

- At first the zip file containing all the images are uploaded to google colab and unzipped, then each image is converted to a feature vector by flattening it and then stored in a dataset. Further the dataset is divided into train and test dataset in a 50:50 ratio. For importing each image to colab 'os' library is loaded into colab and to read images into numpy array cv2 image library is loaded ('cv2' and 'os'library only used for loading image and converting into numpy array). Each feature is normalised by dividing with 255 (as image value ranges between 0-255).
- Now various classes needed for implementing multi-layer perceptron is built.Below are the important classes:
- Neural-net class - This class has the number of neurons contained in each layer. It has two functions in it forward and backward. the forward function calculates the output of the current neuron layer and the backward function calculates the Δw and the derivatives of current layer to be passed to previous layer.
- Activation-classes - We have built 3 activation classes sigmoid, softmax and relu. Each of them has two functions - forward and backward. Forward calculates the output of activation function and backward calculates the derivative of the activation function, multiplies it with the given derivatives and passes it to the previous neural-net layer.
- Optimizer class - We have built two optimizer classes - grad_des optimizer and adam optimizer. grad_des (it is sgd optimizer) simply updates the weight while adam remembers the previous updates and uses a combination of previous updates and current updates to update the weights.
- Now 3 different models are built with different architectures each for squared loss and cross entropy loss and their accuracy noted.
- For each model we are doing online update (i.e batch size is one), because of its speed. In each update in each epoch, first we do forward pass where we calculate outputs of all layers, next we do backprop where we update weights depending on type of optimizer used.
- Each model is trained for 20 epochs and then tested on test set.

4.2. Results

Model 1 has 3 layer, sigmoid activation and sgd optimizer, Model 2 has 3 layer with adam optimiser, Model 3 has 5 layers, adam optimizer and different neurons in each layer.

For cross entropy a near simmilar pattern is followed for models except softmax layer is added in the end.

The below table shows accuracy for each of the models.

Model no.	Squared error loss	CE loss
1	88.39	90.58
2	90.79	92.51
3	90.95	89.19

The F1 score for all classes and all models are good (between 0.85 - 0.95). F1 scores for all models are not mentioned here because of space constraint but they are printed out in the colab notebook.

4.3. Observations

One thing we observed is accuracy value changes as the algorithm is ran again. This is because we have randomly initialised our weight vector so every time we run the algorithm a new weight is being assigned. Besides this we are only running the algorithm for 20 epochs and hence it is not enough for it to converge to minima. Nonetheless the accuracy we get each time we run the code is around 90 to 92 percent. Another thing to note is when we are taking batch size of graeter than one (4 or 8 lets say) the decrease in loss is lesser than when we are doing online updates. Hence to save time we have used online update(i.e batch size as 1). Another thing we observed is the deeper the layer we are taking the more accuracy we are getting (on an average)

4.4. Problems Faced and Failures

One problem we faced was regarding the usage of relu layer, Whenever we are including relu layer as activation function the loss was decreasing at first but after few epochs the gradient was becoming zero and hence loss remained constant after that. We even tried to cap the maximum value of relu layer but without any progress. Whatever be the type of optimizer we are using the loss was not decreasing after a point. probably because most of the inputs are negative and hence there gradients are becoming zero.

5. Question 5

In this question, we have implemented Convolutional Neural Network (CNN) from scratch. We used the Kannada MNIST dataset (KMNIST).

5.1. Implementation

We tried three different architectures. Following are the layers that are implemented and then each is trained for 5 epochs and accuracy noted.

- Convolution layer: In this layer, we learn a filter of fixed size. The size of the filter and the number of filters at each layer are hyperparameters. These two will have an effect on the network performance. In convolution layer weight sharing is done so it reduces the computation as compared to the fully connected network. The convolution layer also helps us in getting a local receptive field.

- Max Pooling layer: In this layer, we decide on a window of fixed size just like we set the filter of the fixed size in the convolution layer. We iterate the input using this window and extract the maximum of numbers lying within the window. This acts as a regularizer.

- Fully connected layer: This layer is just like we implemented in question-4. In this layer, every node in the input layer is connected to every node in the output layer.

- Sigmoid: It is an activation function. This layer does not have any weights to be learned.

- ReLu: It is also an activation function just like the Sigmoid. It also does not have any weight to be learned. Activation functions help us in creating non-linear separating boundaries between the data points, without the activation function we may not be able to get good accuracy.

5.2. Results

For three different architectures, we got these accuracies:

- For the first architecture we got an accuracy of 92.13.
- For the second architecture we got an accuracy of 92.05.
- For the third architecture we got an accuracy of 95.03.

5.3. Observations

- Normalization of the dataset gives better results and we need fewer epochs.
- If the network is too deep and the dataset is small then the network may overfit the training dataset.
- Training for a large number of epochs can also overfit the training data.
- Different loss functions will yield different accuracy for the same training and testing data.
- Type of activation function also has an effect on the accuracy.

5.4. Problems faced

- If the network is too deep, training will take a sufficient amount of time. Because we are supposed to use only Numpy, we can't use GPU for the training because that is optimized for tensors, not for the Numpy array.
- We overfit the data many times, for CNNs it is very easy to overfit because of the large depth of the CNN.

6. Question 6

In this question we have to overfit the image data and then use regularisation to plot bias-variance curve.

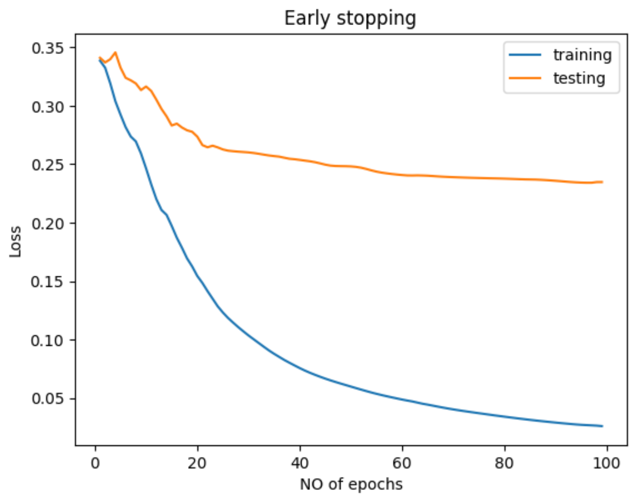

Figure 4. Bias variance curve for early stopping case

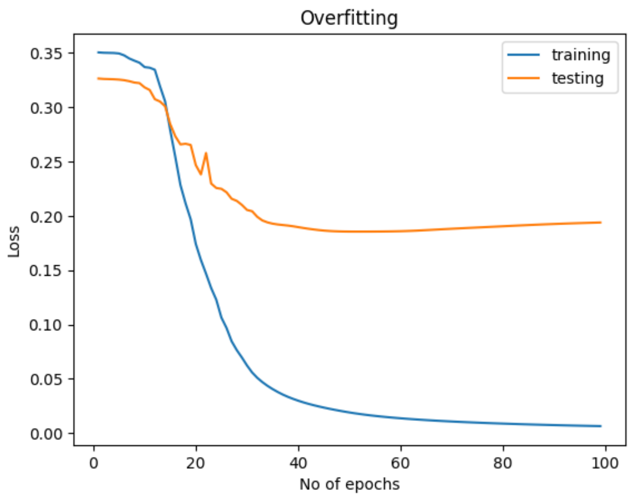

Figure 5. Bias variance curve for overfitting case

6.1. Implementation

In this question, we had all the layers implemented as in question-5 and additionally, we had these layers:

- Dropout layer: In the dropout layer, we drop some of the nodes (or inputs) randomly based on the dropping rate. This layer acts as a regularizer and helps us not to overfit the training data.
- Then train and test error are plotted with epochs in x axis and with various regularisations.

6.2. Results

In this question, we first tried overfitting the training data. AS number of epochs increases train error decreases, test error first decreases but then slowly starts increasing at a slow rate.

6.3. Observations

We have observed that the L2 regularizer reduces the chance of overfitting the data. We tried the L2 regularizer on the given data and then we mixed the noise in the data and then used the L2 regularizer, we observed that due to the L2 regularizer effect of the noise is reduced drastically and we got the almost same result as with the given original data.

6.4. Problems faced

In order to overfit the data we tried several deep architectures for a large number of epochs. The problem with a very deep network is that it takes too much time to train, as we are not using any tensor so we can't use the GPU. In some cases to overfit the complete dataset it may take up to 5-6 hours or even more just to overfit the data in using Numpy.

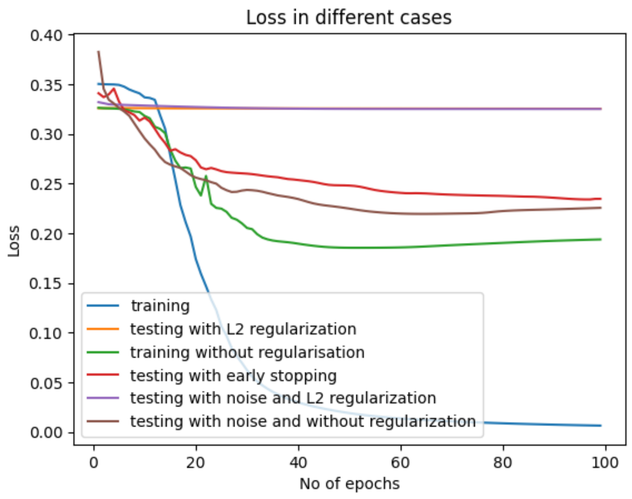

Figure 6. Bias variance curve different cases

7. Question 7

7.1. Implementation

Here the implementation steps are same as Question 4, the same classes that are built in P4 are used here also. The only difference is in the datasets, earlier we had images, hence a 784 dimension data, but here we only have 10 dimensions data (thanks to PCA). The datas are loaded into colab and then divided into train and test dataset in 40:60 ratio.

Same models for squared error and cross entropy are applied as in P4 and accuracy noted. Each model is trained for 20 epochs.

7.2. Results and Observations

Model 1 has 3 layer, sigmoid activation and sgd optimizer, Model 2 has 4 layer with adam optimiser, Model 3 has 5 layers, adam optimizer and different neurons in each layer.

For cross entropy a near simmilar pattern is followed for models except softmax layer is added in the end.

The below table shows accuracy for each of the models.

Model no.	Squared error loss	CE loss
1	94.27	92.85
2	93.88	92.77
3	90.93	94.05

Here F1 scores for each class and each model are mostly between 0.90 to 0.95. Here also we observed changes in accuracy as many times we run, but all of the times accuracy is around 92 - 94 percent. We only have trained this for 20 epochs, accuracy will further increase if trained for more number of epochs.

Similar kinds of problems are also faced in this question as to that of question 4.