COMP24112 Lab2 Report

Ayesha Tabrez

3.2) Explain briefly the knowledge supporting your implementation and your design step by step. Explicitly comment on the role of any arguments you have added to your functions.

The first function I implemented is **l2_rls_train**. This function trains a linear model by minimizing the L2-regularized sum of squares loss through zeroing the loss gradient. The function takes in a set of training samples **data** and their corresponding labels **labels**, as well as a user-specified regularization parameter **p** (lambda). The regularization parameter **p** is used to control the amount of regularization applied to the model. When **p = 0**, there is no regularization applied, and a pseudo-inverse is used to implement the solution. Otherwise, the model is regularized using the L2-norm penalty, which is proportional to the square of the magnitude of the weights.

Here's a step-by-step breakdown of the function:

1. The function first expands the input data **X** with a column of ones, to account for the bias term in the linear model. The expanded data is stored in **X_tilde**.
2. The function computes the coefficient vector **w**. If the regularization parameter **p** is 0, the function computes the pseudo-inverse of **X_tilde** and multiplies it by **y** to get **w**. Otherwise, the function computes the inverse of the matrix **(X_tilde^T * X_tilde + p * I)** and multiplies it by **X_tilde^T * y**, where **I** is the identity matrix.
3. The function returns the learned weights **w**.

The second function I implemented is **l2_rls_predict**. This function takes in the trained weights **w** obtained from **l2_rls_train** and the query data **data** and returns the corresponding prediction. Here's a step-by-step breakdown of the function:

1. The function first expands the input query data **X** with a column of ones, to account for the bias term in the linear model. The expanded data is stored in **X_tilde**.
2. The function computes the predicted labels by multiplying the expanded query data **X_tilde** by the learned weights **w**. Therefore, the multiplication performed is **(X_tilde @ w)**, which is equivalent to **X_tilde.T.dot(w)**
3. The function returns the predicted labels.

In both functions, **the role of the added arguments** is to allow the user to specify the regularization parameter **p** and the input data **data** and labels **labels** for training and query data **data** for prediction. The regularization parameter **p** is used to control the amount of regularization applied to the model, while the input data and labels are used to train the model. The goal of L2 regularization is to encourage the model to learn smaller weights, which can help to prevent overfitting by reducing the complexity of the model. The query data is used to make predictions based on the learned weights **w**.

4.2) Explain the classification steps and report your chosen hyper-parameter and results on the test set. Did you notice any common features among the easiest and most difficult subjects to classify? Describe your observations and analyse your results.

The code performs a classification task on a dataset using the SoftMax regression algorithm with L2 regularization. The dataset is partitioned into training and testing sets. Random subsampling is used to select the best hyperparameter (lambda) for the model. The classification steps are -:

1. Hyperparameter Selection: The regularization parameter lambda is selected using random subsampling on the training set. The function **hyperparameter_selection()** implements this step by evaluating the l2-rls algorithm on a range of lambda values which range from 10**-6 to 10**9, and selecting the lambda that gives the lowest classification error.
2. Training: The l2-rls algorithm is trained on the entire training set using the selected lambda value. The training labels are generated using the **train_label()** function.
3. Classification: The trained model is then used to predict the class labels of the images in the testing set. The classify_err() function implements this step. It first calls the 'train_label' function to get the training label array. Then it calls the 'l2_rls_train' function to learn the weight vector 'w' using L2

regularization. Next, it calls the 'l2_rls_predict' function to predict the output label for each test sample. Finally, it computes the SoftMax probability for each predicted class label using the 'softMax' function and assigns the class label with the highest probability as the predicted label. It then computes the classification error rate as the fraction of test samples whose predicted label is different from their true label. If the 'flag' variable is true, it prints the confusion matrix and error rates for each class label.

We can see that minimum classification error is obtained for lambda=10. Hence best lambda is chosen as 10. It is important to note that since the "partition_data" function randomly splits the images into training and testing sets per subject, the lambda values and resulting accuracies may vary each time the code is executed. The **image_err** list is a list of error for each test sample and **lamda_errors** contains error for each lambda value. We can see that most samples have an error of 0, but a few have errors>0, indicating that they were misclassified. To classify easy and hard images I have used the confusion matrix returned by the function. For each row of the confusion matrix, it finds the index of the column with the highest value (i.e., the class with the highest confidence score) and checks if the value of the highest score is greater than 3. If it is, then it adds the index of that column to a list of "good indices" for that row.

Finally, we see that the overall error mean on the test set was around 0.045, which means that the model has an accuracy of 95.5%. As for the observations, we noticed that the model's performance in classifying images is influenced by the complexity and distinctness of the shapes and features within the images. The model performs better with simpler and more distinct shapes, while struggling with more complex and ambiguous shapes. Another factor that affects classification difficulty is the similarity between images within a class. Classes with distinct features and shapes are easier to classify, while classes with similar features and shapes are more challenging.

5.2) Report the MAPE and make some observations regarding the results of the face completion model. How well has your model performed? Offer one suggestion for how it can be improved.

The MAPE obtained is around 0.2155. It can be concluded that the face completion model performs moderately well, but there is still room for improvement. The result of the face completion model on left face testing set looks quite similar to the right face of the testing set, however there are some minor differences in details. The model is off by approximately 21.55% on average. Therefore, reducing the mean error further could improve the model's performance. One possible reason for the model's poor performance could be the lack of a suitable hyperparameter. In the given code, the hyperparameter is set to 0, which means that no regularization is being applied. Regularization can help prevent overfitting and improve the model's generalization performance. By tuning the hyperparameter, we can potentially improve the model's accuracy and make it more effective in completing faces. Therefore, one suggestion for improving the model's performance is to find a better hyperparameter. By trying different values of the hyperparameter and measuring the model's performance on a validation set, you can select the value that gives the best performance. This can help improve the model's accuracy and make it more effective in completing faces.

6.3) Analyse the impact that changing the learning rate has on the cost function and obtained testing accuracies over each iteration in experiment 6.2. Drawing from what you observed in your experiments, what are the consequences of setting the learning rate and iteration number too high or too low?

In my implementation, two learning rates were tested: 10^-3 and 10^-2, and both were run for 200 iterations. For 10**-3 the cost function decreases whereas for 10**-2 it increases and reaches infinity. For the learning rate of 10^-3, the training and testing accuracies increased and then stabilised to 1 after around 15 iterations for training set and around 20 iterations for testing set, indicating that the model was converging towards a good solution. This indicates that the model is gradually improving in accuracy and is converging towards the optimal weights and achieves perfect classification accuracy after some iterations. It can be noted that the training accuracy stabilises to 1 after few number of iterations compared to testing accuracy because the model overfits to the training data. However, since the testing data is unseen, it stabilises after more number of iterations.

On the other hand, for the learning rate of 10^-2, the training and testing accuracy remained stable at 0.5 for all iterations. This suggests that the learning rate was too large, causing the weights to become too large and affecting the accuracy of the model. Large learning rates can cause the model to diverge, overshoot the optimal solution and become unstable, which results in poor performance and convergence issues. As a result, the model is not improving in accuracy and is stuck at a suboptimal solution. Moreover, setting iteration number too high may cause the model to overfit to the training data while setting it too low might cause the model to underfit to the training data leading to poor performance.

In conclusion, the choice of learning rate and iteration number should be carefully considered in order to achieve the best performance of a model. A learning rate that is too small may lead to slow convergence, while a learning rate that is too large may lead to unstable results. The number of iterations should be chosen such that the model has enough time to converge, but not so much that it overfits the data.

7.3) Explain in the report your experiment design, comparative result analysis and interpretation of obtained results. Try to be thorough in your analysis.

In this experiment, I used linear least squares (LLS) with stochastic gradient descent (SGD) to classify data for two subjects (subjects 1 and 30). The data for each subject is partitioned into training and testing sets, and the training set is used to train the LLS model using SGD. I have partitioned data into 3 images per training and 7 images per testing The model is then used to predict the class labels for both the training and testing sets, and the accuracy of the predictions is calculated using the TestingAccuracy() function.

The experiment design involves the following steps:

1. Load the data and corresponding class labels.
2. Partition the data for subject 1 and 30 into training and testing sets using the partition_data() function.
3. Concatenate the training and testing sets for both subjects.
4. Get the corresponding data and labels for the concatenated sets.
5. Train the LLS model using the train_data and train_label.
6. Evaluate the performance of the model using the test_data and test_label.
7. Plot the cost over iterations for the training process.
8. Plot the classification accuracy for the training and testing sets over iterations.

To evaluate the performance of the model, we calculate the accuracy of the predicted labels for both the training and testing sets using the TestingAccuracy() function. The function first converts the predicted labels to 1 or -1 based on a threshold of 0, and then calculates the number of correct predictions and the total number of samples. The accuracy is then calculated as the ratio of correct predictions to total samples.

It can be seen that the change of sum-of-squares error loss decreases for both gradient descent and stochastic gradient descent after a certain number of iterations. However, the SGD graph seems to be more jagged and wiggly. The training and testing accuracies increase for both GD and SGD and then stabilise at 1 after around only 20 iterations for GD and around 200 iterations for SGD. I also observed that for SGD the testing accuracy sometimes stabilises to 1 and sometime close to 1, around 95%.

GD is better than SGD since it achieves higher accuracy in fewer iterations compared to SGD. The advantage of SGD is that it helps prevent overfitting compared to GD. This is because it randomly samples one training data point in each iteration, which introduces some noise and randomness to the optimization process, making it less likely for the model to memorize the training data and overfit. In contrast, GD considers all the training data in each iteration, which can lead to overfitting, especially if the model is complex and the dataset is small.

Thus, we can see that GD and SGD differ in the number of iterations needed to minimise the loss to a stable level. This is because GD updates model parameters by computing the gradients of the loss function with respect to all parameters using the entire training dataset in each iteration whereas SGD updates parameters using only one training examples at a time, which introduces randomness and results in a more stochastic optimization process.GD typically takes longer to converge because it processes the entire dataset in each iteration, whereas SGD updates the parameters more frequently and results in a more wiggly and jagged loss curve. Thus, in this case GD performs better but takes more time compared to SGD. To improve SGD, we could use mini-batches of 2-3 training samples.