Middle East Technical University Department of Computer Engineering

CENG 499

Intro to ML

Fall 2024-2025

Assignment 1

Name SURNAME: Aly Asad GILANI
Student ID: 2547875

This is my report for Part 3 of the first homework of CENG499: ML. Let's first discuss the training settings I used, for starting out, I used very basic settings recommended by the PDF and the settings that I saw used in Part 1 and Part 2 of the same HW. They were:

1) Number of hidden layers: [1, 2]
2) Number of nodes in each hidden layer: [64, 128]
3) Learning rate: [0.01, 0.001]
4) Number of epochs: [1000, 2000]
5) Activation functions: [ReLU, Sigmoid, Tanh]

While the first three hyper-parameters were fine, I changed the number of epochs as it was just too large for the code to run in a moderate amount of time, it would have taken hours or even days to run. I tried epochs 500-1000, but that was too high too. In the end, I settled with [50, 100]. If your computer is really powerful, you can try increasing the number of epochs for better accuracy. I also changed my activation functions, since the amount of permutations would have been 48, and that would've taken a huge amount of time to run. I tried all three activation functions, and found Tanh to work much better than the other two, give accuracies 1.5-2x better than the others, so I decided to only use Tanh for the activation function, and it brought the total permutations down to 16. So, in the end, my hyper-parameter settings are (also mentioned in the code itself):

1) Number of hidden layers: [1, 2]
2) Number of nodes in each hidden layer: [64, 128]
3) Learning rate: [0.01, 0.001]
4) Number of epochs: [50, 100]
5) Activation functions: [Tanh]

Now, I will talk about my code and its structure and the functions I used, starting with the MLP class. I made the class since in torch library's official website, it stated that we have to create the class inheriting the nn.Module subclass. In it, I defined the hidden layers and the output layer so the torch functions can propagate through the layers and calculate the output automatically. Next, I defined the "train_model" function that trained a model once, by iterating over its epochs and updating the weights. I first used full batch gradient descent, but it was taking a lot of time to run so I decided to use the Adam optimizer mentioned in the PDF. This made the training faster by optimizing the updates (hopefully).

Next, I defined the ”grid_search” function, which contains my hyper-parameter values and iterates over each combination of them 10 times, training them and finding the accuracy of each model. To find the accuracies, I have defined a ”evaluate_accuracy” function that tests the model with the given input and expected output data, it also applies softmax to the model's output to normalize it. The grid search function then prints the average accuracy and confidence interval of each combination and updates the best model if the average accuracy of the current model is the highest.

The last function I defined was the ”final_training_and_evaluation” function, which used the best model from the 16 combinations I tried and trained it using the training and validation data, then tested the model using the test data and printed the test mean accuracy and confidence interval. Lastly, I called the grid search function and then the final training and evaluation function to run the algorithm and find the best model and its output.

The results I obtained are very expected, from my runs the best accuracies I got were around 85-90%, which matched the accuracy ballpark given in the ODTUClass forums. The confidence interval was pretty low, around 0.3-0.4%, which was reassuring as it meant that the models are consistent on average even with randomized inputs.

This concludes my report for Part 3, now I will answer the additional questions asked at the end of the PDF:

Question 1: What type of measure or measures have you considered to prevent overfitting?

We can use a lower number of epochs, so the model doesn't over-fit itself on the training data. We can also shrink the weights towards zero slightly after each update so they don't grow too large. We can also stop the iterations over the dataset earlier than our specified number of epochs if we see the signs of overfitting (which are explained in the next question).

Question 2: How could one understand that a model being trained starts to overfit?

A very well-known way is to compare the model's training accuracy with its validation accuracy. If the training accuracy remains high or is increasing but the validation accuracy is decreasing/validation loss is increasing, then the model is being overfitted.

Question 3: Could we get rid of the search over the number of iterations (epochs) hyper-parameter by setting it to a relatively high value and doing some additional work? What may this additional work be?

If we just set the number of epochs really high without any restraints, the model will most probably over-fit. We can avoid this by constantly checking for the signs of overfitting described in the above question, and stop the loop when the validation accuracy has peaked. This will effectively get rid of the search over the number of epochs we should use.

Question 4: Is there a "best" learning rate value that outperforms the other tested learning values in all hyperparameter configurations?

From the limited configurations I tested, the learning rate of "0.01" had an increased 5-10% accuracy when compared to "0.001". This might be because my number of epochs were too low, so the low learning rate did not have enough time and iterations to converge towards a minimum.

Question 5: Is there a "best" activation function that outperforms the other tested activation functions in all hyperparameter configurations?

From the initial tests I performed, the "Tanh" function definitely performed better than both the "Sigmoid" and "ReLU" functions by a good margin, for example initially I was getting 60-70% accuracy with the "Tanh" function while I was only getting around 30-40% accuracy with the other two. This might be because of my other hyper-parameter choices, and the other two functions might perform better with different configurations, or the "Tanh" function might just be generally better as an activation function than the other two for this task.

Question 6: What are the advantages and disadvantages of using a small learning rate?

A small learning rate gives more stability for convergence, so there is less chance to miss a local minimum, as well as a less chance for it to oscillate around the minimum. It could also lead to better accuracy as the model learns slowly.

However, since the model learns slowly, it takes much more time to train and will require more overall cost. Additionally, it is very difficult for the model to escape a local minimum and it can be detrimental because it doesn't get to explore other possible solutions.

Question 7: What are the advantages and disadvantages of using a big learning rate?

Because of a big learning rate, the model can converge faster, which saves both time and money. The model also has more chances to jump out of local minimas, allowing it to explore other solutions.

However, because of the large jumps it makes, the model could accidentally leave the optimal minimum behind, causing it to go further away from the optimal solution. It could also oscillate around the minimum.

Question 8: Is it a good idea to use stochastic gradient descent learning with a very large dataset? What kind of problem or problems do you think could emerge?

It could be a good idea to use Stochastic Gradient Descent (SGD) with very large datasets since it is much more efficient than the full batch gradient descent because it uses small clumps of the data at a time rather than all of the data at once, which allows it to run faster and use less time and resources. However, we could run into some problems, like the weight updates could be inaccurate as its only using a small portion of the data. Additionally, even though SGD is faster than batch gradient descent, it might not converge as quickly because of the inaccuracies and might take even more time to converge than batch gradient descent.

Question 9: In the given source code, the instance features are divided by 255 (Please recall that in a gray scale-image pixel values range between 0 and 255). Why may such an operation be necessary? What would happen if we did not perform this operation?

We need to divide the features by 255 to scale them between 0 and 1, as the range of the inputs is between 0 and 255. We need a normalized range for the input to the model as a wide range of values can cause instability in the calculated gradient. Additionally, for very large values, the tanh, sigmoid, and softmax functions saturate and their gradient approaches zero, and since most of the values in the range [0, 255] would saturate the functions, it would be heavily biased, which would not allow the model to learn properly.