CENG499 Homework 1: Part 3

Ozan Kamalı

Model Architecture and Hyperparameter Search

For this task, several hyperparameters are tested to determine their effects on the performance of the model. The key hyperparameters that I have chosen are as follows:

- **Number of Neurons:** Two configurations are considered: 64 and 128 neurons in the hidden layer.
- **Iterations:** The model is trained for either 75 or 150 iterations.
- **Activation Function:** Three activation functions are evaluated: ReLU, Tanh, and Sigmoid.
- **Learning Rate:** Two learning rates are used: 0.001 and 0.01.

Model Description

The model utilized in this task is a Multi-Layer Perceptron (MLP) implemented using PyTorch. The architecture consists of an input layer, a single hidden layer, and an output layer. The details of the model architecture are as follows:

- **Input Layer:** The input consists of 784 features, which is for dataset MNIST, representing 28x28 pixel images.
- **Hidden Layer:** The number of neurons in the hidden layer is a hyperparameter that can be adjusted (64 or 128). The activation function used in the hidden layer is also configurable (ReLU, Tanh, or Sigmoid).
- **Output Layer:** The output layer consists of 10 neurons, corresponding to the 10 classes for classification tasks.
- **Loss Function:** Cross Entropy Loss function of the torch library is used.
- **Optimizer:** Adam optimizer of the torch library is used.

Implementation

The implementation of the model consists of a simple MLP with a single input layer, single hidden layer and one output layer. The hyperparameter search is conducted by the itertools library to get the cartesian product of the hyperparameter values and from there it is passed to several for loops to accomplish the task. Then there is another file for the testing of the best hyperparameter configuration that we get from the previous mentioned method. The file configuration for 'part3.zip' is as follows:

- **part3_hp_tester.py:** This code creates the cartesian product of the hyperparameter values, and then dynamically creates an MLP model from its arguments. For every configuration, the model is created from the beginning 10 times, run on training and validation dataset, and then finally on the test dataset, and for each 10 iterations the mean accuracy and the confidence interval scores on the test dataset is calculated and kept in $general - accuracies$ array. When we have finished iterating through all configurations, the best hyperparameter configuration (the one with the best mean accuracy value and confidence interval score) is written to a pickle file format to be parsed from the part3.py file. Since this code takes a lot of time to run, we have utilized the cuda capabilities of PyTorch by moving the datasets and the MLP model to pass it to cuda enabled device. The code uses CPU in cases the cuda is not enabled.
- **part3.py:** This code starts by parsing the $best - hyperparameters.pkl$ file to create our final model. Then, our dataset is again loaded into the code and as stated in the task the training and validation dataset are combined to get a bigger training dataset. The only loop in this code is for the one hyperparameter configuration to run 10 times. After we get the test accuracy scores, we then calculate the final mean and confidence interval scores for this best hyperparameter configuration, and print it.
- **README.md:** Contains information on how to run these two files.

Results

Table Description:

This table summarizes the results of various hyperparameter configurations evaluated during the model training process. The columns are defined as follows:

- **#N.:** The number of neurons in the hidden layer of the neural network.
- **Epochs:** The number of complete passes through the training dataset.
- **A.F.:** The activation function used in the hidden layers of the network, which determines the output of each neuron.
- **L.R.:** The learning rate.

#N.	Epochs	A.F.	L.R.	M.A	C.I.
64	75	ReLU	0.001	0.79666	(0.79467, 0.79865)
64	75	ReLU	0.01	0.84567	(0.84395, 0.84739)
64	75	Tanh	0.001	0.78941	(0.78764, 0.79118)
64	75	Tanh	0.01	0.85371	(0.85206, 0.85536)
64	75	Sigmoid	0.001	0.69953	(0.69270, 0.70636)
64	75	Sigmoid	0.01	0.84133	(0.84024, 0.84242)
64	150	ReLU	0.001	0.83109	(0.82986, 0.83232)
64	150	ReLU	0.01	0.86197	(0.86108, 0.86286)
64	150	Tanh	0.001	0.82935	(0.82842, 0.83028)
64	150	Tanh	0.01	0.87039	(0.86981, 0.87097)
64	150	Sigmoid	0.001	0.77164	(0.76987, 0.77341)
64	150	Sigmoid	0.01	0.86424	(0.86369, 0.86479)
128	75	ReLU	0.001	0.81524	(0.81443, 0.81605)
128	75	ReLU	0.01	0.85223	(0.85011, 0.85435)
128	75	Tanh	0.001	0.81459	(0.81343, 0.81575)
128	75	Tanh	0.01	0.85763	(0.85494, 0.86032)
128	75	Sigmoid	0.001	0.74004	(0.73735, 0.74273)
128	75	Sigmoid	0.01	0.84783	(0.84734, 0.84832)
128	150	ReLU	0.001	0.84163	(0.84089, 0.84237)
128	150	ReLU	0.01	0.86996	(0.86912, 0.87080)
128	150	Tanh	0.001	0.84168	(0.84093, 0.84243)
128	150	Tanh	0.01	0.87335	(0.87183, 0.87487)
128	150	Sigmoid	0.001	0.80354	(0.80156, 0.80552)
128	150	Sigmoid	0.01	0.86854	(0.86775, 0.86933)

Table 1: Results of Hyperparameter Configurations

- **M.A:** The mean accuracy achieved by the model during evaluation.
- **C.I.:** The confidence interval for the mean accuracy.

Best Hyperparameter Configuration

In my model and implementation, the best performing hyperparameter configuration was achieved with the following settings:

- #N.: 128 (Number of neurons in the hidden layer)
- Epochs: 150 (Number of training epochs)
- A.F.: TanH (Activation function)
- L.R.: 0.01 (Learning rate)
- M.A: 0.87335 (Mean accuracy)
- C.I.: (0.87183,0.87487) (Confidence interval for the mean accuracy)

Final Results

In my final calculations I have found the following scores for the best performing hyperparameter configuration:

- **Final Mean Accuracy:** 0.8749
- **Final Confidence Interval** (0.8709474035267941, 0.8788525964732059)

Additional Details

- To prevent overfitting I did not train the model for too many epochs, reducing the risk of the model memorizing the training data instead of learning to generalize from it. Also the training data is shuffled before each epoch. This ensures that the model sees the data in a different order every time it trains and prevents the model from learning patterns.
- If the training accuracy is high, but we perform bad on the unseen test dataset, we can understand that we are overfitting.
- We can get rid of the epoch hyperparameter by setting it to a relatively high value, however as mentioned in the first bulletpoint, this may lead to overfitting. Thus the additional work would be to eliminate this overfitting. We can utilize cross validation or feature selection (so the feature count is not too high to overfit) to improve the productivity of our model.
- In my model, the higher learning rate which is 0.01 compared to the lower one 0.001 performed better in every hyperparameter configuration, as can be seen in Table 1.
- In my model, the Tanh activation function performed better on every configuration with the learning rate is at: 0.01. However, ReLu activation function performed generally better when the learning rate is at: 0.001. Noting that in each of these configurations, I could not observe a significant difference between Tanh and ReLu activation functions but most of the time they outperformed the Sigmoid function.
- A high learning rate can be good because it leads to higher convergence of a loss function, however there is a risk of overshooting circulating around the intended minimum value.
- A low learning rate allows for more stable updates and convergence and a better chance at avoiding overshooting, however since we would need more steps to update it can lead to longer training times and higher computations. Also there is a risk of getting stuck at non optimal minimal values.

- Stochastic gradient descent divides the dataset into smaller batches and calculates the weight updates from the sum of the losses of this batches instead of updating them after the whole iteration over the complete dataset. So in a very large dataset, it is a good idea to use stochastic gradient descent since it will lower the computational load. However since we are updating the weights based on approximations over these batches, the descent could be noisier and might converge slower to the optimal minimum.
- The division by 255 is done so that we normalize the pixel values between (0-1). It is always better for numbers to be minimized to keep the numerical stability, also the activation functions used such as Tanh, ReLu, Sigmoid all saturate or disrupt higher values, which leads the gradient and the computation to be lost, creating unreliability in our computation.

Conclusion

In this report, we have explored the effects of various hyperparameters on the performance of a Multi-Layer Perceptron model for classifying the MNIST dataset. The results of these configurations are stated within Table 1, and also the best performing hyperparameter with its performance score and confidence interval is given in this report with the explanation of various concepts in the additionals section.