ROB313 Assignment 4: Neural Networks

Andrew Jairam
1006941972
andrew.jairam@mail.utoronto.ca

Introduction and Objectives

Neural networks work consistently when they are used in conjunction with good initialization, overfitting prevention, and making use of structures in the problem. This assignment aims to walk through an implementation of a neural network for classification on the MNIST dataset using various built-in modules and design practices that gratify the aforementioned consistency criteria.

This implementation will examine the use of Xavier initialization, popularly used to preserve zero mean and unit variance across the outputs. Additionally, the multi-class log-likelihood will be derived for use, which uses the LogSumExp trick for computational stability. This trick is abundant in the field of machine learning, especially when analyzing quantities in the log scale, and as such, warrants a review of how it works, and how it is used to improve results.

With the neural network in place, the final objective is to employ tuning and evaluation practices to optimize performance: which helps with preventing overfitting and allows for problem structures to be observed. Specifically, a general method to tune the model hyperparameters to optimize the accuracy of the model will be determined to uncover good practices for quickly optimizing neural networks that align with expected theoretical results. With these hyperparameters, model evaluation using the convergence of the loss will be explored to evaluate the effective time it takes for the model to reach a good prediction. Additionally, the model will also be analyzed using a confusion matrix to inspect the aforementioned problem structures to practice employing problem-specific strategies in neural network problems.

Derivation of Variance for Xavier Initialization with Linear Activation

In this section, the variance for Xavier Initialization with linear, identity activation to be used in the neural network will be derived. This means that it is required to assert a variance ε^2 on the weights taken from a normal distribution $w_{ij} \sim N(0,\ \varepsilon^2)$ such that the variance of the input $x_j \sim N(0,\ \eta^2)$ is equivalent to the variance on any output layer z_i: that is, $Var(z_i) = \eta^2$.

The hidden unit output expression for z_i is given as:

$$z_i = \sigma\left(\sum_{j=1}^{D} w_{ij}x_j\right) = c\sum_{j=1}^{D} w_{ij}x_j + b$$

Where c and b are arbitrary constants by virtue that σ is a linear activation function. Furthermore, we are given that σ is also identity: and thus it can be assumed that $c = b = 1$, allowing the simplification:

$$z_i = \sum_{j=1}^{D} w_{ij} x_j.$$

Taking the variance of this expression, it can be noted that the variance of w_{ij} and x_j are constant for all j as $Var(w_{ij}) = \varepsilon^2$ and $Var(x_j) = \eta^2$, so the variance of the summation can be written in terms of one variance term without the sum as:

$$Var(z_i) = Var(\sum_{j=1}^{D} w_{ij} x_j) = D[Var(w_{ij} x_j)] \textbf{ (1)}.$$

The derivation carries on with the formation of an expression for $Var(w_{ij} x_j)$, where the variance in terms of expectation formula is used, which takes the general form

$$Var(X) = E[X^2] - E[X]^2 \textbf{ (2)}$$

for a random variable X. Using that the random variables w_{ij} and x_j are independent:

$$Var(w_{ij} x_j) = E[(w_{ij} x_j)^2] - (E[w_{ij}]E[x_j])^2 = E[w_{ij}^2]\, E[x_j^2] - (E[w_{ij}]E[x_j])^2$$

Using **(2)** again to expand $E[w_{ij}^2]\, E[x_j^2]$, we arrive at the final expression for $Var(w_{ij} x_j)$ by fully simplifying, noting that the expected values for w_{ij} and x_j are zero due to being sampled from normal distributions with zero mean:

$$Var(w_{ij} x_j) = [Var(w_{ij}) + E[w_{ij}]^2][Var(x_j) + E[x_j]^2] - (E[w_{ij}]E[x_j])^2$$

$$Var(w_{ij} x_j) = Var(w_{ij})Var(x_j) + E[x_j]^2 Var(w_{ij}) + E[w_{ij}]^2 Var(x_j) + (E[w_{ij}]E[x_j])^2 - (E[w_{ij}]E[x_j])^2$$

$$Var(w_{ij} x_j) = Var(w_{ij})Var(x_j) = \varepsilon^2 \eta^2 \textbf{ (3)}$$

Plugging **(3)** into **(1),** we arrive at the final expression for the variance of the weights:

$$Var(z_i) = D[Var(w_{ij} x_j)] = \eta^2 = D\varepsilon^2 \eta^2$$

$$\varepsilon = 1/\sqrt{D}$$

The result, $\varepsilon = 1/\sqrt{D}$, was used to initialize the weights in the init_xavier function. Initialization is key to the function of a neural network, as initializing too large leads to exploding gradients, initializing too small leads to vanishing gradients, and not initializing at all leads to static, non-changing gradients. Xavier initialization additionally preserves the zero mean and unit variance of inputs, which can be beneficial when analyzing conditioning.

The LogSumExp Trick in Classification Problems

In multi-class classification, it is standard to use dataset log-likelihood to perform classifications. This process involves the use of logarithm and exponential computations on the outputs, taking the 'naive' form as given in the assignment handout:

$$outputs = outputs - np.log(np.exp(outputs).sum(-1, keepdims = True)).$$

This form is considered 'naive' because it poses computational challenges. When calculating the sum of the exponential of the outputs, there exists a risk of overflow or underflow if outputs are very large and positive or very large and negative respectively. This could cause the program to crash, as overflow/underflow values would be interpreted as positive or negative infinity. This is a problem because we want the neural network to be able to deal with different kinds of outputs, and it is common in practice that this overflow issue can occur.

Instead of the naive computation form, the LogSumExp trick is employed to combat this issue while doing the same math. The premise of the trick is to center the exponential computation at the largest output value to minimize the risk of overflow. In turn, the computations are made more stable as it guarantees that the largest possible value the exponential term can be is $exp(0) = 1$. The use of this ensures that the model likelihood does not blow up: which is important in producing both a flexible and stable model.

Construction of log-likelihood for Multi-Class Classification

This section outlines the steps to implement the mean_log_like function in the starter code that returns the averaged log-likelihood classification metric. To implement the classifier, the categorical distribution, a generalized Bernoulli distribution for *K* categories was written as derived in lecture, shown below:

$$Pr(\mathbf{y}^{(i)} | \boldsymbol{x}^{(i)}, \boldsymbol{w}) = \prod_{i=1}^{N} \prod_{j=1}^{K} \widehat{f}_j(\boldsymbol{x}^{(i)}; \boldsymbol{w})^{y_j^{(i)}}$$

In logarithmic terms, the log-likelihood for the dataset is:

$$log\ Pr(\mathbf{y}^{(i)} | \boldsymbol{x}^{(i)}, \boldsymbol{w}) = \sum_{i=1}^{N} \sum_{j=1}^{K} y_j^{(i)} log\ \widehat{f}_j(\boldsymbol{x}^{(i)}; \boldsymbol{w})$$

The $log\ \widehat{f}_j(\boldsymbol{x}^{(i)}; \boldsymbol{w})$ is given in the starter code by the neural_net_predict function which initializes the neural network, and $y_j^{(i)}$ are the targets passed into the mean_log_like function. The above expression can be implemented using a few simple line of vectorized Python code:

```
def mean_log_like(params, inputs, targets):
    """ TODO: return the log-likelihood / the number of inputs
    """
    log_probabilities = neural_net_predict(params, inputs)
    num_inputs = len(inputs)
    return np.sum(targets * log_probabilities) / num_inputs
```

Figure 1: Snippet of the implemented mean_log_like function

Parameter Tuning on a Neural Network for Classification

With the model now implemented, this section will focus on procedures to tune the parameters of the resulting neural network to optimize performance. It is desired that the neural network achieves 95% accuracy as a design constraint, so the hyperparameters had to be tuned until this was satisfied.

There are four hyperparameters that can be tuned in the neural network: the number of layers, individual layer size, the number of epochs, and learning rate.

The design of an algorithm that seeks a relatively 'optimal' neural network in terms of maximizing performance and minimizing time warrants special considerations. As per theory, increasing the number of hidden units, increasing the number of layers and decreasing learning rate increase accuracy at the cost of computation time. Increasing the number of epochs increases convergence only: it was confirmed by observation that there is a maximum value of the number of epochs given a fixed learning rate where increasing the number of epochs has no effect on performance. In the implemented tuning algorithm, the accuracy was considered 'converged' if the accuracy fluctuates less than 1% between epochs. Additionally, tuning the learning rate has the highest sensitivity to improving accuracy and increasing runtime, so the tuning of the learning rate should come after the other hyperparameters are played with. In order to achieve the desired 95% accuracy in good time, an incremental approach was designed.

The tuning process executed took these theoretical factors into account. The parameters were first initialized to one 200-size layer with a learning rate of 0.1 and 10 epochs. The process to tune the gains is described below, and the process was stopped when an accuracy of 95% on the validation set was observed.

For each learning rate, the number of epochs was increased until the 1% convergence metric was met. The layer size was increased in increments of ten up to a layer size of 750 was tested. For layer sizes 250, 500, and 750, two layers both of the sizes given above were tested. If 95% validation accuracy still was not observed, the hyperparameters were reset to their initialized values, and the process was repeated with a learning rate decreased by a factor of ten.

The final selected hyperparameters and training, validation, and test accuracies are shown in Figure 2 below. As it can be seen, the test accuracy is above 95%, and the neural network thus performs as desired: confirming that the tuning method is functional. To

briefly note some takeaways from the tuning process, the method used did work as expected with theory. In particular, however, it was found that increasing the number of layers did not significantly improve performance while significantly decreasing program runtime: it was much better to increase the layer size as marginally better improvements were witnessed over marginally increased runtime. This result would likely depend on the data, but the key factor in this tuning analysis is the learning rate: which should be tuned in small steps and gradually. In future, different learning rates should be tried first in the tuning process to find a learning rate that yields results closer to the desired accuracy, and then fine-tuning by tweaking the other hyperparameters should be executed after this to reduce the number of steps in the optimization procedure.

Number of Epochs	Learning Rate	Number of Layers	Number of Units (Layer Size)	Final Training Accuracy	Final Validation Accuracy	Final Test Accuracy	Runtime
15	0.01	1	750	99.76%	95.2%	96.1%	49.9 s

Figure 2: Model Specifications and Final Reported Performances

Figure 3 shows the negative log-likelihood, which we know is analogous to loss from the third assignment, plotted against the epoch number for the training and validation sets. As can be observed, convergence happens within about one epoch while the loss slightly diverges.

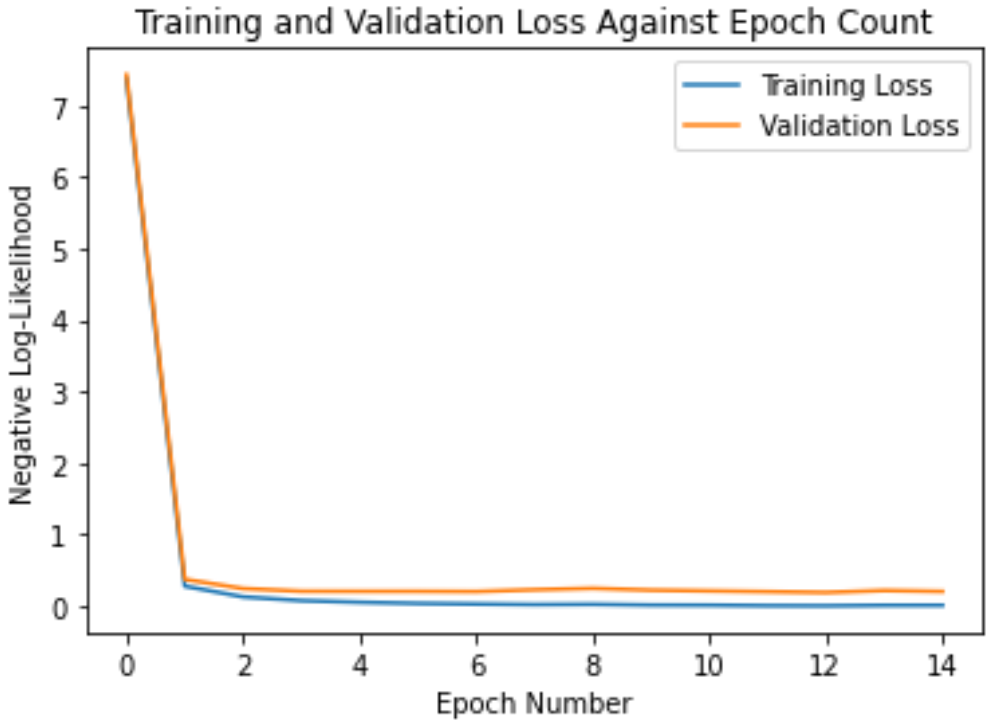

Figure 3: Training and Validation Loss against Epoch Count

Evaluation of Neural Networks using Confusion Matrices

To evaluate how the algorithm performs with respect to outliers, a confusion matrix using the built-in sklearn method is plotted in Figure 3 below, which plots the number of times the model guessed a label *i* when the correct label was *j*. The benefit of this structure is that it can be clearly seen which labels are most likely to be misclassified as another specific label. The confusion matrix was plotted using the test data and predictions.

The results of the confusion matrix indicate that the model performed quite well across the board: there are not any significant points where outliers exist. Most of the data lies along the diagonal of the confusion matrix, which indicates correct labelling. The biggest outliers in the model are three misclassifications of label '7' as '2' and label '3' as '8'.

One improvement that can be made is to add more label '5' samples to the test set to better evaluate performance. The accuracy "heatmap" shows that the accuracy in predicting label '5' correctly is the lowest only because there are the least amount of label '5' samples, and if there are less samples, an incorrect prediction affects the accuracy with greater impact. This results in the classifier having low confidence in predicting the correct label despite making correct predictions. The label' 5' classifier looks like it performs well in comparison to other labels relatively, but the lack of data leads to some uncertainty if this trend would continue. More data on labels 6 and 8 would also be helpful, as it looks like the classification of these labels is falling into the same pitfall as what is happening in the label '5' classification.

A second improvement can be discussed by noting that the classifier incorrectly predicts labels as the label '8' much more than it misclassifies any other label. To solve this, more label '8' samples should be added to increase the classifier's precision in predicting label '8'. An improvement that might come with this is introducing hyperparameters and tuning all model hyperparameters to intentionally skew misclassifications to label '8' by reducing the variance of predicting other labels. As more label '8' samples are added, the classifier would be more confident in not misclassifying digits as label '8': which would increase performance as we skewed model variance so that it most likely incorrectly mispredicts labels as label '8'.

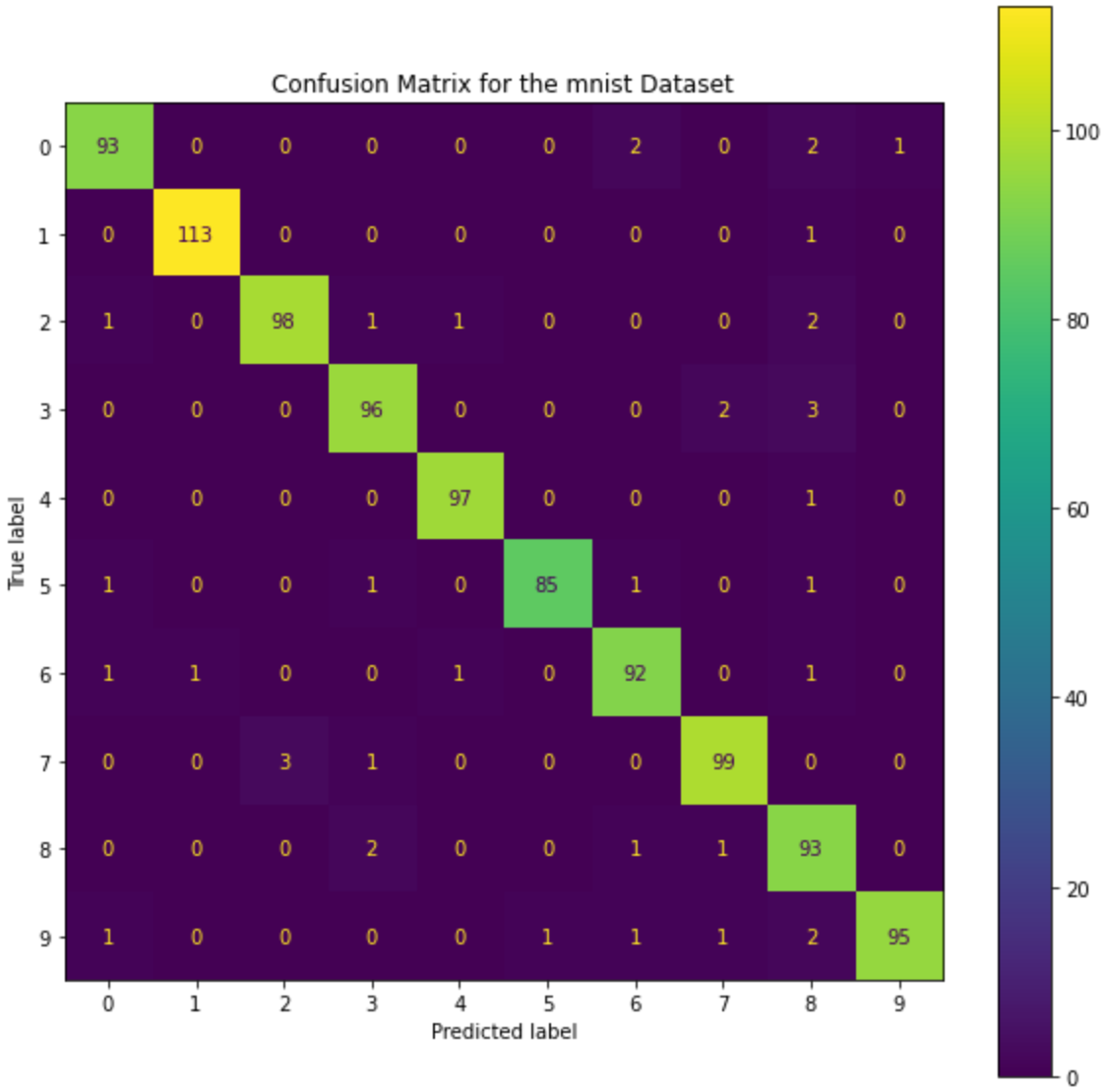

Figure 4: Confusion Matrix for the Implemented Neural Network

- Suggest two strategies to improve the model
    - Add more label '5' samples to the data

Conclusion

In implementing neural networks, it is crucial to perform setup steps to increase performance efficacy. Xavier initialization has proven to be a robust initialization algorithm based on the results, as has the treatment of classification with the use of a log-likelihood metric. Furthermore, the LogSumExp method used while initializing was found to keep model likelihood from blowing up: creating computation stability.

An incremental tuning process results in a quick and efficient method to optimize performance. From the conclusions of the implemented procedure, it is optimal to first converge on desired accuracy by setting a learning rate that almost satisfies the accuracy, and tune the layer size and number of layers until fine accuracy is accrued. This should be

done while keeping the number of epochs to converge to the optimal accuracy low for optimality in speed.

The two improvements from evaluation methods were to add more label ‘5’ and label ‘8’ samples, and to decrease the variance of labels other than label ‘8’ to skew incorrect predictions towards label ‘8’. This results in the classifier having more confidence in predictions that it is correctly guessing, or if the classifier performs poorly over the added data, serves as a point of discussion for future model improvements.