Samuel Looper
1003044181

2019-03-26

ROB313 Assignment #4

1. Assignment Objectives

The purpose of this assignment is to assess the performance of the autograd Python library for neural network machine learning applications. Specifically, the main objective of the assignment is to create a 3 layer, 100 neuron neural network for multi-class classification of the MNIST letter image data set. Some initial skeleton code was provided and modified for the multi-class classification application. First, a softmax output function was added to ensure outputs were valid probability densities. Next, the negative log likelihood function was modified for a Bernoulli probability distribution rather than a Gaussian function. Then, the full neural net was built and initialized using the Xavier method. The performance of the system was evaluated for the training and validation data sets. Finally, the number of hidden layers was varied, the performance re evaluated, and the full configuration visualized using Python library tools.

2. Algorithm Overview

This neural network algorithm is built on existing skeleton code, with functions to perform a forward pass, calculate the loss function, and update parameters, as well as the Autograd Python library, which includes utility functions to calculate gradients for each of the parameters in the neural net. The main code loop includes the necessary functions to load the data sets, initialize the neural network, and then a for loop which performs iterations on the neural network, computing estimated labels, and updating weights and biases at every iteration. The final portion of the code is for the visualization, which was mostly comprised of simple calls to the data_util's plot_digits() function. An additional helper function was created for part 6 to identify problematic input values.

.

3. Results

3.1 Softmax

For the first question of the assignment, the existing skeleton code was modified to include a SoftMax output layer. This function recalculates the final output values according to the formula:

$$y_j = \text{softmax}(z_1, z_2, \ldots, z_K)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

This formula creates a set of valid probability distributions (i.e. all probability inputs sum to 1), and represents a continuous analog to setting the maximum probability to 1 and all others to 0. In its original representation, it is not vectorized, and can be numerically unstable due to overflow/underflow. Underflow means that some large negative value of z can lead to the exponential being such a small number that it is evaluated to be 0. Overflow occurs for large positive values of z where the exponent being evaluated is larger than the maximum integer size (typically 2^{32}). To solve both of these issues we use the Log-Exponent-Sum method of calculating Softmax. First, a variable *a* is set to the maximum value of all the outputs, and the following expression is calculated:

$$log\ SoftMax\ =\ p_i - a - log\ \sum_{j=0}^{k} e^{p_j - a}$$

This formulation allows us to vectorize the calculations using standard numpy functions such as np.exp, np.log, np.sum(a,b, axis=1) and np.amax(M, axis=1). In addition, this ensures numerical stability. Since we subtract a from the exponent in our sum term, we avoid overflow, and underflow simply rounds to zero in the sum. As long as there is at least one value that does not underflow in the sum, the log will evaluate some bounded value. Assuming the difference between the inputs are bounded, this function will output a bounded output, and thus it is numerically stable.

3.2 Bernoulli Log Likelihood Function

The next step in modifying the skeleton code for classification was to change the way the log likelihood function is calculated. Previously, the log likelihood was calculated assuming a Gaussian distribution, and calculated through a sum of the log of a series of Gaussian probabilities. We replaced this calculation with a new log likelihood assuming a Bernoulli Probability Distribution:

$$\Pr(y|\mathbf{w},\mathbf{x}) = \left[\hat{f}(\mathbf{x};\mathbf{w})\right]^{y}\left[1-\hat{f}(\mathbf{x};\mathbf{w})\right]^{1-y},$$

The sum of the log of these probabilities, which is how we compute our final log likelihood, is thus the following:

$$log\ P(Y|w,X) = \sum_{i=1}^{N} y^{(i)}\ log[\widehat{f}(x^{(i)})\ ] + (1-y^{(i)})\ log[\ 1-\widehat{f}(x^{(i)})\ ]$$

3.3 3-Layer Neural Network

Finally, a full 3 layer neural network can be assembled to perform multi-class classification on the MNIST data set. The neural net includes 100 neurons at each layer with a ReL activation function, and the softmax at the output of the net. The weights are initialized by Xavier Initialization, whereby each are sampled for a normalized Gaussian distribution. Next, the Autograd library was used to create a gradient function used in the optimization. With a fixed learning rate and iteration number, the Neural Network is ran and optimized iteratively. As a performance metric, the Bernoulli Log Likelihood is calculated at each iteration for the final output. The following plots show this performance metric at every iteration number for the training and validation sets.

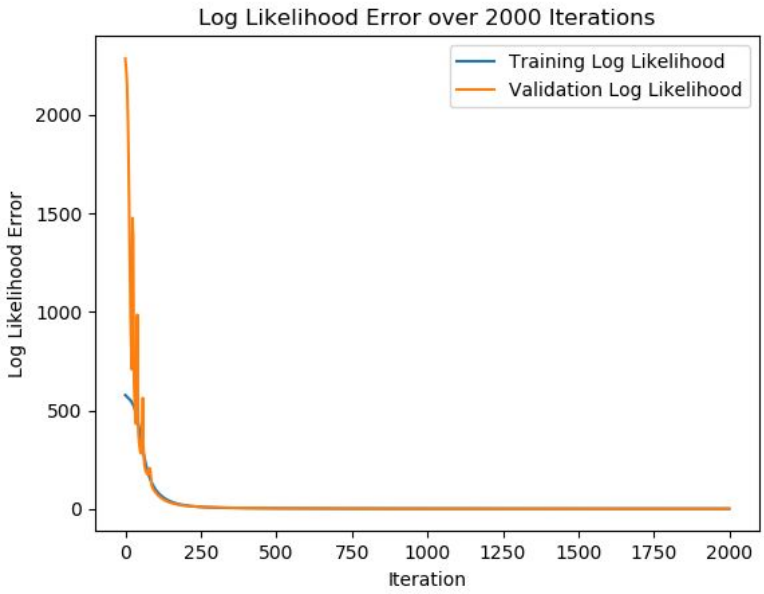

Fig: Resulting plot of log likelihood for training and validation sets

We can see that both exhibit very similar converging behavior, as both log likelihoods quickly converge to zero. However, the Validation set's likelihood starts at a much greater value and exhibits some stochastic behavior as it converges.

3.4 Varying Number of Neurons in 3-Layer Neural Net

Next, the set accuracy and log likelihood were analyzed for a series of neural networks with a different number of hidden neurons per layer (n = 50, 100, and 150). The data in the following table results in an optimization run similar to what was run in section 3.3, with a learning rate of 0.0002 and 2000 iterations.

# of Neurons per Layer	**Validation Set Log Likelihood**	**Validation Set Accuracy**
50	662.444962	0.797
100	579.524013	0.812
150	559.054525	0.822

Fig: Validation Set Log Likelihood and Accuracy for various number of neurons

Testing Set Accuracy for 100 Neurons per layer: 0.85
Testing Set Log Likelihood for 100 Neurons per layer: 498.729

Looking at the progress of the training error for each of the 3 different optimization runs, all three neurons exhibit very similar convergence behavior, especially for accuracy, where all 3 sets reach 1.00 not too long into the 2000 iterations. The steady state log likelihood slightly decreases for the optimizations with larger numbers of neurons. The most significant different between the different neural nets was in how the reductions in training error translated into low validation error. The neural nets with a higher number of neurons were better generalized for the different data sets. Nonetheless, by this metric, the difference between 50 and 100 was more pronounced than between 150 and 100, suggesting diminishing returns with respect to this metric as your grow the size of your model. While the runtime was not noticeably affected, this could become an issue for a large, over-parameterized neural network.

3.5 Weight Visualization

Once trained, the neural net neurons can be thought of as a series of filters looking for correlation for specific patterns in the input data image for the MNIST digits. They will be filtering for some defining feature of the number that allows it to have a high degree of accuracy even when given new data it has never seen. To gleam insight from the values at those weights, he data_utils plotDigit function was used to plot the weights as images themselves. This image would be a visual representation of the image filter it applies to data inputs to classify into digits. Below are the 16 neurons that were plotted, taken at random from W1.

Looking at these weights, they initially seem quite abstract. With 100 neurons per layer, it would make sense that the 100 different features the neural network is filtering for may not have straightforward explanations. However, we can see some emerging patterns in the weight filters. Note: You can see the visualized weights in Appendix A. Some, like the third weight, seem to be filtering for specific parts of the image (in this case looking for high correlation in the center of the image). In other cases, you see faint slanted lines or edges that are slightly darker, such as the 4th or the last filter, which could be reminiscent of the edges in a seven or one. Finally, we also see some filters with distinct curved shapes, which we know differentiate a 9 from a 4, for example (see weights 11 and 14).

3.6 Problematic Input Visualization

The MNIST data set used as an input for our digital classifying neural net has thousands of images of digits. These pictures are handwritten and include many variations of the digits to classify to improve the performance of the neural net when it is generalized to any handwriting. As a result, several of these input digits will be purposefully confusing even to the human eye.

While the neural network created in this assignment will not have the same idea of confusing numbers as us, it nonetheless had some inputs for which it had a very low confidence (i.e. more distributed probabilities between digits), and some for which it did not correctly select. For the last part of this assignment, these problematic values were visualized using the same tool used in part 5, the plot_digits() function provided in data_utils.

To do this the helper function used to calculated testing accuracy was modified to output the input values that the neural net guessed incorrectly and had a maximum probability below a certain threshold. Those were the values plotted for inspection. In total there were 13 inputs from the testing set incorrectly guessed with probabilities below 0.37. By inspection we can tell why these values may be confusing. Some could be easily confused between two digits, leading the probability to be split (see middle image below). Others had some extra dot or feature that would likely confuse the neural network, like the dot on top of the rightmost image below. Finally, some can be discerned to be a digit by the human eye, but lack the definitive features the neural network likely uses to identify the digit (see leftmost image below)

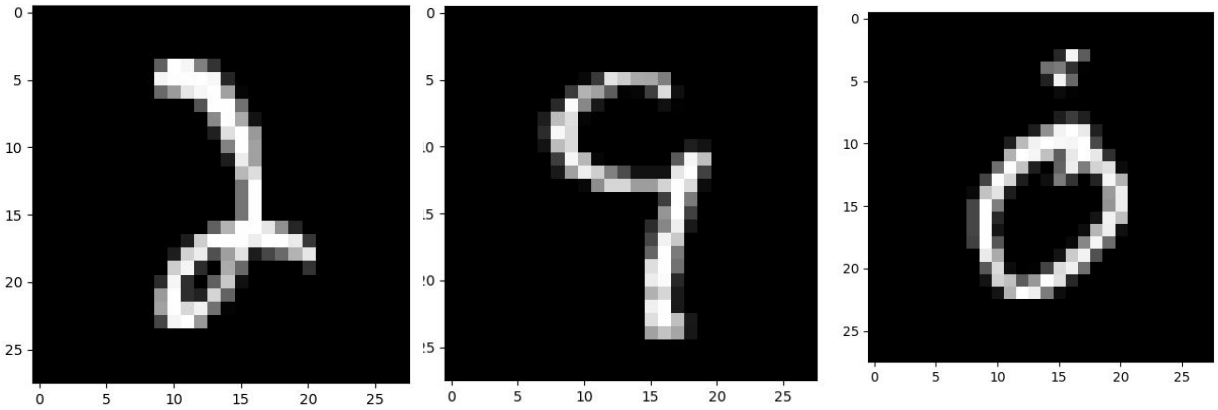

Fig: 3 sample problematic input digits plotted for visualization purposes

However, in some cases, the digits look perfectly normal and easily discernible, giving every benefit of the doubt to the neural net. Nonetheless, just like there were some weights with indiscernible features, some of these input digits will be problematic for reasons beyond our comprehension. This is a great case study in one of the key difficulties of the study of neural nets, which is the traceability of the decisions it makes. It is hard for us to understand why the classifier neural net chooses certain labels instead of others, even with the benefit of visualization.

Appendix A: Weight Visualization

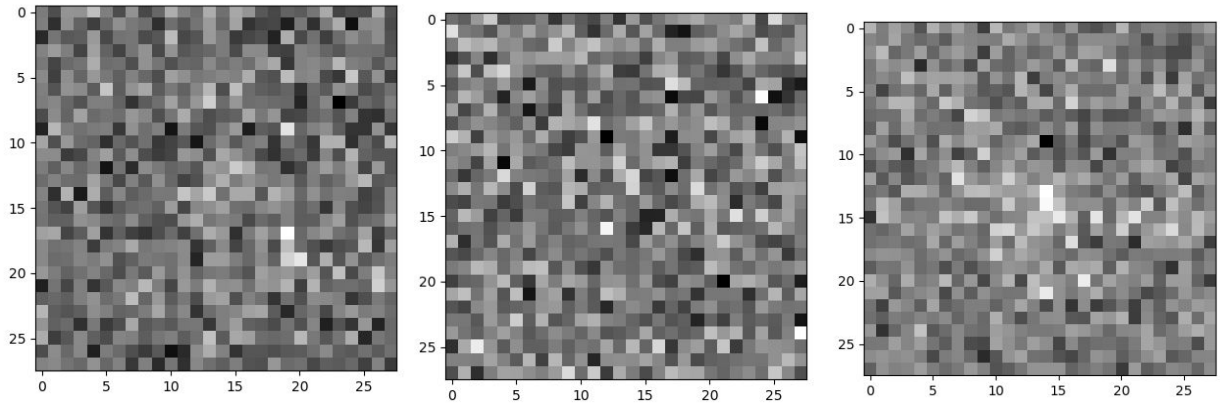

Fig: Weights 1, 2 & 3

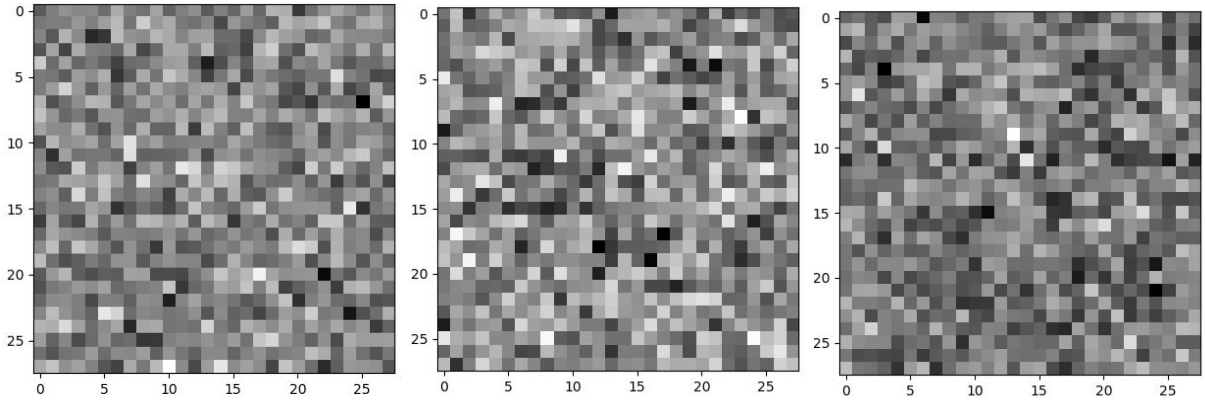

Fig: Weights 4, 5 & 6

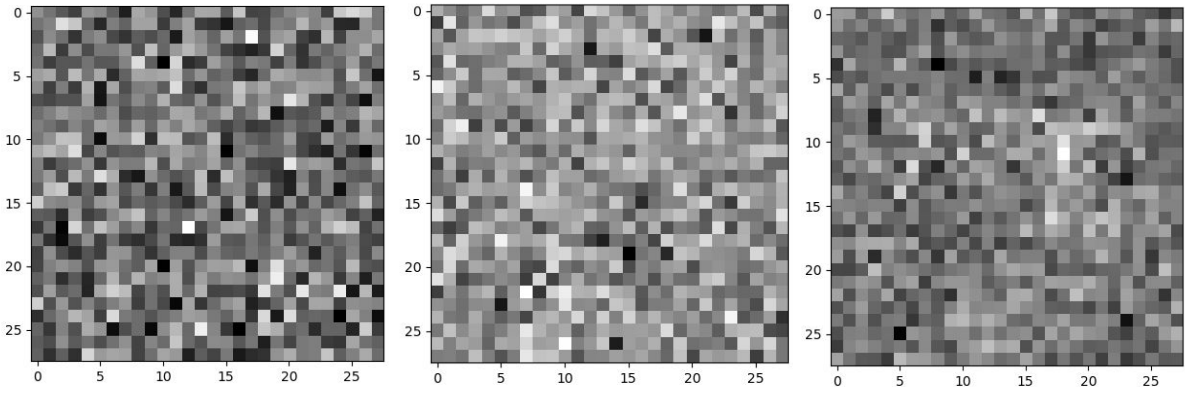

Fig: Weights 7, 8 & 9

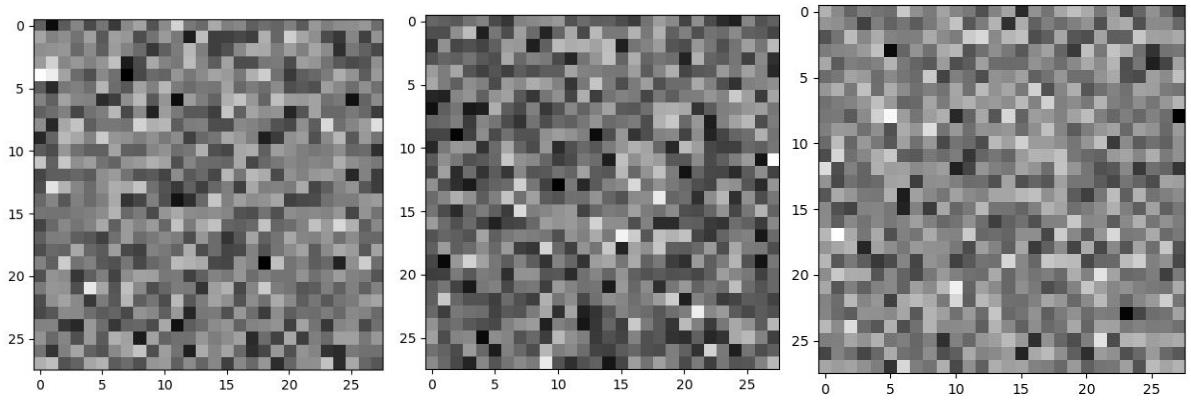

Fig: Weights 10, 11 & 12

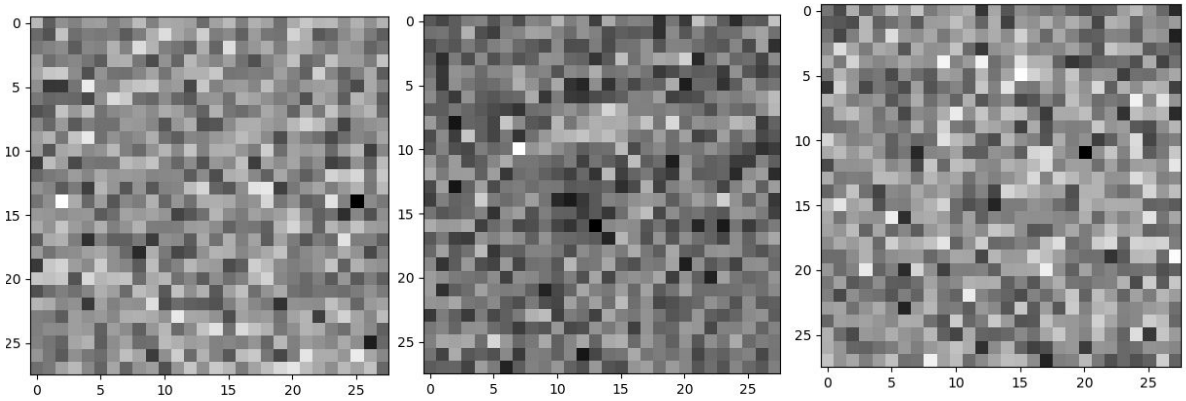

Fig: Weights 13, 14 & 15

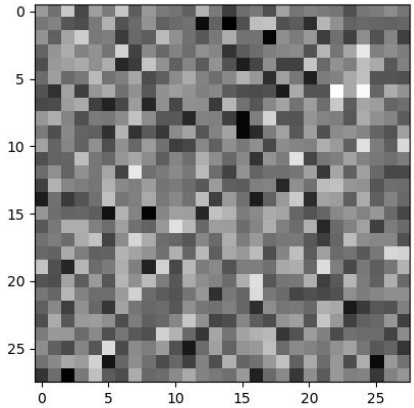

Fig: Weight 16