Optimizing Deep Neural Networks for Handwritten Digit Recognition

Chase Wiederstein

March 2025

Abstract

This project explores the application of various deep learning architectures, including Multilayer Perceptrons (MLP), Convolutional Neural Networks (CNN), and Locally Connected Neural Networks (LCNN), for the classification of handwritten digits from the U.S. Postal Service. Through extensive experimentation with hyperparameters, the study aims to optimize model performance to achieve high accuracy on digit recognition tasks. Results indicate significant differences in model efficiencies, underscoring the critical role of hyperparameter tuning in enhancing predictive capabilities.

1 The Dataset

The dataset consists of handwritten digits scanned from U.S. Postal Service envelopes. It has been preprocessed and normalized, containing 16×16 grayscale images with pixel values scaled to the range (-1,1). This is a supervised classification task, where the goal is to correctly assign each image to one of 10-digit classes (0-9). The training set includes 7,291 samples, while the test set contains 2,007 samples. Additionally, 20% of the training set was set aside as a validation set for hyperparameter optimization. Each entry in the dataset consists of a digit label (0-9) followed by 256-pixel values representing the flattened 16×16 image. A special thanks to Yann LeCun and the neural network group at AT&T Research Labs for making this dataset publicly available. Figure 1 below shows an example entry of a handwritten digit from the training set.

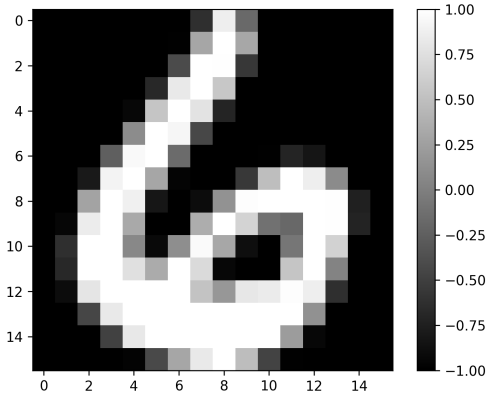

Figure 1: A Sample Number 6 from the Training Set

2 Model Architectures

The criteria for the models require at least four layers, one usage of a rectified activation function, and one usage of a sigmoid or tanh function. I designed three distinct neural network architectures: a fully connected multilayer perceptron (MLP), a locally connected network (LCNN) without shared weights, and a convolutional neural network (CNN) with shared weights.

2.1 Fully Connected Neural Network

For this network, I designed a simple five-layer architecture. One downside of using an MLP is that it requires the input data to be flattened from its original 16×16 grayscale matrix into a one-dimensional vector. This means the input layer must have 256 neurons, corresponding to the total number of pixels in the image. The description of its five layers are listed below.

1. The input layer consists of 256 neurons, representing the flattened 16×16 grayscale images.
2. The first hidden layer consists of 128 neurons and applies the ReLU activation function allowing the network to detect important patterns and edges while zeroing out negative values.
3. The second hidden layer consists of 64 neurons and applies the sigmoid activation function to enable the model to learn complex representations by mapping values between 0 and 1.
4. The third hidden layer consists of 32 neurons and again applies ReLU to further refine learned features.
5. The output layer consists of 10 neurons where the softmax function is used to produce a probability distribution over the 10 classes for classification.

2.2 Convolutional Neural Network

Unlike the MLP, the CNN keeps the original matrices from our dataset and utilizes them to observe features of higher quality than that of the other models. Due to this, I expect to see a better performance when using the CNN. The CNN used consists of 7 layers mentioned below.

1. A convolutional layer applies 32 filters with kernel size 3×3 with a stride of 1 preserving the input dimensions at 16×16. ReLU is applied here to enhance edge detection.
2. A pooling layer is used with a 2×2 kernel and a stride of 2 reducing the output to 8×8.
3. A second convolutional layer applies 64 filters with a kernel size of 3×3 and a stride of 1 keeping the dimensions at 8×8. The ReLU activation function is applied again to enhance feature extraction.
4. A pooling layer applies a 2×2 kernel again with a stride of 2 reducing the features to 4×4.
5. A fully connected layer flattens the 4×4 matrix and then maps it to 128 neurons for feature extraction. The tanh activation function is used to regulate feature values before classification.
6. A dropout layer is applied with an optimized probability to reduce overfitting.
7. The output layer then maps these 128 neurons to 10 output neurons corresponding to the 10 classifications. Softmax is applied here to convert raw scores into probabilities for classification.

2.3 Locally Connected Neural Network

The LCNN is quite similar to the CNN mentioned above. However, the LCNN does not share weights. For an LCNN, instead of a single CNN filter scanning across the entire image, each location has its own independent set of learned weights. This allows the network to learn location-specific features. My implementation consists of six layers, which are listed below.

1. A locally connected layer with eight independent 3×3 kernels, each operating on local regions of the 16×16 input matrix. ReLU activation is used here to ensure positive features are preserved.

2. Another locally connected layer with 16 independent kernels and a stride of 2 produces a 6×6 output per channel. ReLU activation is applied again to enhance feature extraction.

3. A pooling layer that applies a 2×2 kernel with a stride of 2 reducing the dimensions to 3×3 per channel.

4. A fully connected layer that takes the flattened $16 \times 3 \times 3$ output and maps it to 128 neurons for higher feature extraction. The tanh activation function is used here to ensure features stay within the range of (-1, 1).

5. The last layer is fully connected and maps the 128 neurons to 10 output neurons corresponding to the digit classifications. Softmax is applied here to convert raw scores into probabilities for classification.

3 Training

For each model, the training method was implemented as stochastic gradient descent with momentum. Given that this is a multi-class classification problem, the loss function chosen was cross-entropy loss. To prevent overfitting during training, dropout layers were included in all models, as stated in the architecture descriptions above. I implemented minibatch gradient descent to optimize our weights and biases with respect to the cross-entropy function. During training, all weights were initialized from the Xavier uniform distribution, and biases were initialized to zero for all models.

4 Hyperparameter Optimization

The process of hyperparameter optimization is crucial in fine-tuning the models to achieve optimal performance. For this process, we chose to perform a coarse-to-fine optimization strategy, beginning with a random search to explore a wide range of values, followed by Bayesian optimization for fine-tuning. The hyperparameters targeted for optimization include learning rate, momentum, batch size, and dropout rate.

4.1 Coarse Random Search

The random search is a straightforward approach. Random combinations of the four hyperparameters are tried, and the three models are trained using these combinations, then their performance is tested on a validation set. The distributions from which the hyperparameters are randomly sampled are as follows:

- Learning Rate: $(10^{-7}, 10^{-1})$. This range helps to explore very slow to moderately fast learning speeds, balancing the risks of slow convergence and overshooting minima.
- Batch Size: {16, 32, 64, 128}. This set was chosen to adhere to standard practices of using powers of two for computational efficiency and batch normalization stability.
- Momentum: $(0.5, 0.99)$. This range was chosen to investigate the effect of different degrees of acceleration in the convergence process.
- Dropout: $(0.1, 0.7)$. This range was chosen to test a spectrum of regularization strengths to mitigate overfitting without under-utilizing the network capacity.

A total of 30 trials were conducted for each model during this search phase. Each trial involved training the models with a set of randomly chosen hyperparameters, with performance evaluated based on validation accuracy. The primary goal of this phase is to identify a broad set of promising hyperparameter values that can be refined further.

4.2 Kernel Density Estimation

Following the coarse random search, Bayesian optimization is then used to fine-tune the hyperparameters within more narrowly defined intervals. This process begins with a statistical analysis of the top 20% performing trials from the random search phase. We perform kernel density estimations on the distributions of hyperparameters from all trials and focus specifically on the best-performing samples to identify regions of high potential. For each hyperparameter, the mean μ_{hp} and standard deviation σ_{hp} are calculated for the top performers. The new search intervals are then defined as $(\mu_{hp} - \sigma_{hp}, \mu_{hp} + \sigma_{hp})$ which will be used in Bayesian optimization. Since batch size is a discrete random variable, in order to determine the best batch size used in fine optimization, the most occurring batch size out of the top 20% is chosen. Figure 2 below shows the KDE distributions of the top 20% of statistics alongside the KDE for the entire sample of 30.

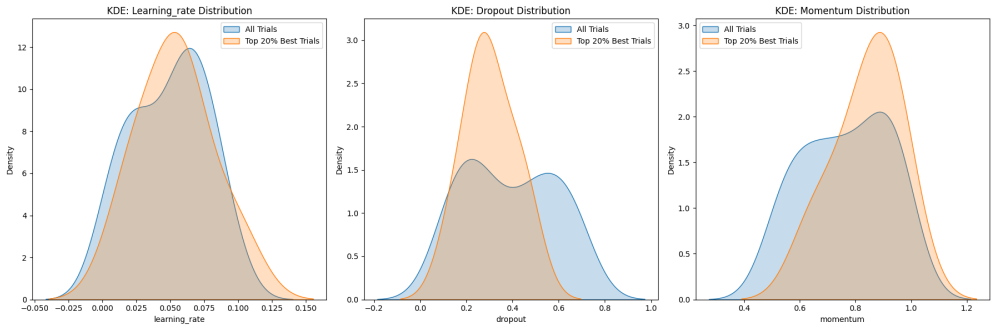

Figure 2: KDE Plots for the MLP Random Sampling

4.3 Bayesian Optimzation

Now that the new bounds are found for the hyperparameters on all models, Bayesian optimization is then conducted. The package used is Optuna which will handle most of the workload in this section. Bayesian hyperparameter optimization considers past evaluation results before selecting the next hyperparameter combination. It uses an objective function to base its result. In our case, the objective function is the validation accuracy on the validation set. The number of trials is set to 20. As the number of trials increases, a gradual increase in validation accuracy is expected. In Table 1 below, I have included the best validation accuracies from the random search and then the improved validation accuracy from Bayesian optimization.

	Coarse Validation Acc.	Coarse Test Acc.	Fine Validation Acc.	Fine Test Acc.
MLP	93.21%	91.08%	94.45%	98.18%
LCNN	96.57%	94.42%	96.34%	93.17%
CNN	98.77%	95.86%	98.70%	96.31%

Table 1: Validation and Test Accuracy Comparison between Fine and Coarse Optimization

As seen from above, all percentages are successfully increased moving from coarse accuracies to fine accuracies. The only exception is that the LCNN failed to increase its performance during Bayesian optimization. In this scenario, the original hyperparameters found in coarse optimization will be used in the final model for the LCNN.

5 Ensemble

To improve classification accuracy, the predictions from three different models are combined using a weighted voting approach. Each model generates its own probability distribution over the classes, and different weights are assigned to their outputs before making a final decision. The weights are optimized through a random

search to find the best combination. The intervals at which the weights are uniformly drawn are (0,.2) and (0,.8) for the MLP and LCNN. The weight for the CNN is then the difference between 1 and the sum of the randomly drawn weights for the MLP and LCNN. This was determined due to the test performance being the worst in MLP and best in CNN. The number of trials is 100 for weight optimization. Ultimately, the ensemble resulted in a final test accuracy of 96.31%, but the improvement was negligible. The optimal weights found through the search were nearly 0 for both the MLP and LCNN, with the CNN receiving a weight of 0.98. This suggests that the ensemble did not contribute meaningful performance gains beyond the CNN alone, as it naturally dominated the decision-making process.

6 Hyperparameter Influence on Model Performance

For this assignment, the dynamic impacts of various hyperparameters on model performance are explored, focusing on parameter initialization, learning rates, batch sizes, momentum, and dropout effects. This section delves into how these hyperparameters influence training dynamics and learning outcomes, using experiments conducted across multiple model architectures.

6.1 Impacts of Weight Initialization on Gradient Dynamics

To ensure stable training, all models utilize Xavier uniform initialization, a widely used method that helps maintain a balanced gradient flow. When weights are initialized with very small values, the gradients during SGD become too small, slowing down learning. This can result in whats known as vanishing gradients preventing the model from updating its parameters effectively. To simulate slow learning, weights were drawn uniformly from the distribution (-0.001, 0.001) for all models. To demonstrate fast but ineffective learning, the weights were drawn uniformly from (10,000, 20,000) for the LCNN and CNN, and the interval was (10, 20) for the MLP. Initializing weights with excessively large values can lead to results in exploding gradients, where weight updates grow uncontrollably, making training unstable. Table 2 below shows the average gradients for the different weight initialization strategies.

	Slow	Effective	Fast
MLP	2.02×10^{-4}	0.1456	0.0
LCNN	0.0314	0.1175	1.0799
CNN	0.0309	0.7607	1.0008

Table 2: Average Gradient Magnitudes for Different Initialization Strategies

The optimal range for effective learning is gradient sizes between .1 and .99. Any gradient smaller than .1 is considered slow learning and any gradient faster than .99 is considered too fast. As seen from above, the MLP falls between these ranges except for the fast region having an average gradient of 0 indicating vanishing gradients have occurred. The values for both the LCNN and CNN are in appropriate regions.

6.2 Impact of Learning Rate Scaling on Model Training Stability

For this task, the optimal learning rates are scaled for each model by factors of 0.01, 1, and 10 while keeping all other hyperparameters the same as those found during hyperparameter optimization. The training loss is then logged over 20 epochs as each model is trained using these different learning rates. Figure 3 depicts the training loss of the MLP model. For clarity, the plots for the CNN and LCNN have been omitted, as they follow the same trends as the MLP. A scaling factor of 0.01 results in a very slow decrease or even a slight rise in training loss in all models, while a scaling factor of 100 leads to unstable training and unpredictable loss spikes.

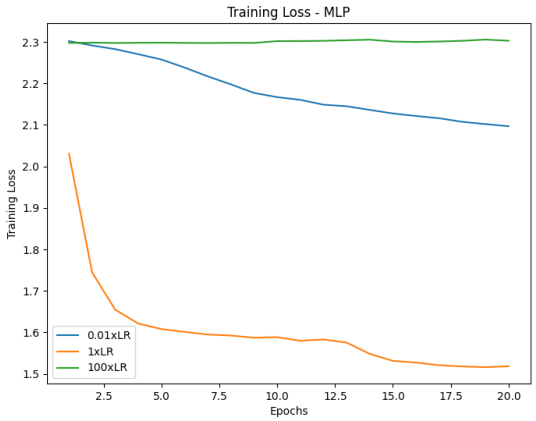

Figure 3: Effects of Learning Rate Scaling on MLP Training Loss

6.3 Impact of Batch Size Variations on CNN Training

Batch size impacts batch normalization in a CNN by determining how accurately the mean and variance of activations are estimated during training. Smaller batch sizes lead to unstable statistics and noisy updates, while larger batch sizes provide more stable normalization, improving training consistency and generalization. During hyperparameter optimization, an optimal batch size of 32 was identified. To analyze its effects, a batch size of 8 was compared against 32 by tracking gradient magnitudes across training. After only 9 epochs, the gradient magnitude for batch size 8 drops below 0.01, leading to very slow learning. In contrast, with batch size 32, the gradient remains stable throughout all 20 epochs, enabling more effective learning. Figure 4 below shows the results of the gradient magnitudes of the two batch sizes throughout training.

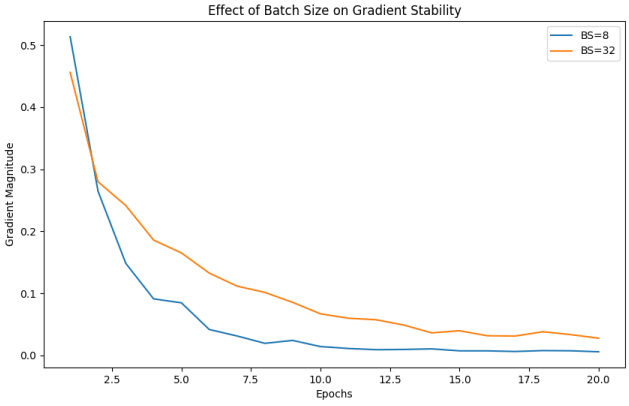

Figure 4: Batch size of 8 and 32 Training Magnitudes

6.4 Impacts of Momentum on CNN Training

During coarse optimization, 30 random trials revealed a correlation coefficient of -0.559410 between momentum and validation accuracy. This suggests that higher momentum values generally lead to lower accuracy. Figure 5 illustrates that momentum values of 0.99 result in significantly larger gradient magnitudes, indicating unstable learning. In contrast, lower momentum values yielded more stable training and a consistent validation accuracy of approximately 98%.

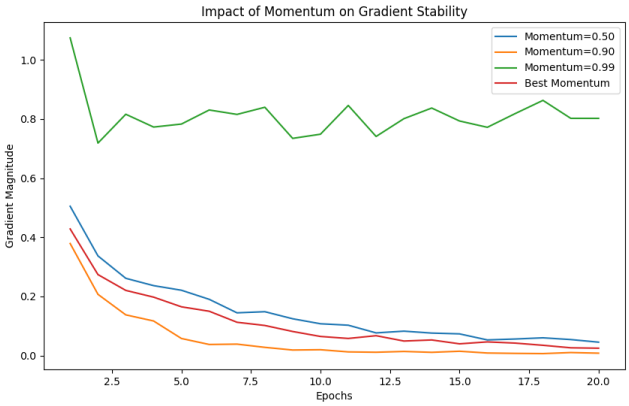

Figure 5: Momentum Effects on Gradient Stability

6.5 Impacts of Dropout on MLP Training

From the 30 random trials conducted during the coarse optimization phase, a correlation coefficient of -0.133966 was observed between dropout and validation accuracy, suggesting a minimal impact of dropout on accuracy. A significant effect is only anticipated at extreme values, such as a dropout rate of 0.99. Figure 6 illustrates the effects of dropout on training loss across epochs, highlighting that a dropout value of 0.99 notably impedes training.

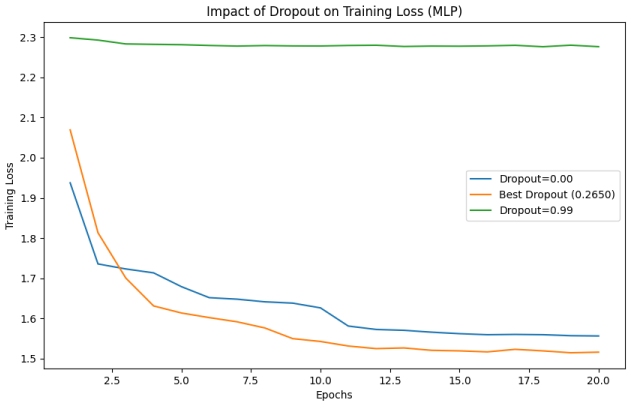

Figure 6: Dropout Effects on Training Loss