Handwriting recognition with neural network

Michail Niotis

January 2026

1 Data preprocessing and training methods

For this classification problem, I used the MNIST dataset and trained various models (MLPs) on PyTorch to compare their performance based on their depth and hidden layer width as well as the effect of L2 regularization. Throughout training and evaluation, I processed data in mini batches using Pytorch DataLoader, as this method makes it easier to work with large datasets, is memory-efficient and faster. It also allows for more efficient training since we update the model parameters based on the gradients of the loss in each batch, which acts a bit as regularization as it's a noisy estimate of the true gradient. Since the training dataset consists of 60000 images and the test dataset consists of 10000 images, I used a minibatch size of 200. For every iteration over the DataLoader, we get a batch of inputs and a batch of the corresponding targets. Since the MNIST dataset contains images, inputs are tensors of shape $[200, 1, 28, 28]$ and targets have a shape of [200] containing the true class of each image. In order to pass the input through MLPs and perform matrix multiplication, we must flatten each image to a vector of length $28 * 28 = 784$, so the input is reshaped to $[200, 784]$. The chosen loss function for this problem is Cross Entropy, since it's a multi-class classification problem and we want to compare the model's prediction with true classes. In PyTorch, Cross Entropy expects the logits scores for each class and not the probabilities and that's why each model outputs a 10-th dimensional vector for each image (the final layer of the neural network has a size of 10), containing logit scores over all possible classes. Regarding optimization, I chose Adam optimizer and a learning rate of 5×10^{-4}.

2 Model Comparison

To compare how depth and hidden layer size of MLPs affect the performance of the model to predict correctly the true classes of the digits, without applying L2 regularization, I trained the following models for 100 epochs and tracked the Cross Entropy Test Loss per epoch:

- A simple Linear Model without hidden layers:

$z(x) = Wx + b$, where for every image $x \in R^{784}$ I apply a linear transformation with a weight matrix $W \in R^{10\times784}$ and add a bias $b \in R^{10}$, getting 10 logit scores ($z \in R^{10}$), one for each class to predict.

- 4 Shallow MLPs with different hidden layer sizes $(512, 256, 128, 64)$

A shallow MLP is described as: $z(x) = W_2\sigma(W_1x + b_1) + b_2$, where $x \in R^{784}$, $W_1 \in R^{hid\_dim\times784}$, $b_1 \in R^{hid\_dim}$, $W_2 \in R^{10\times hid\_dim}$, $b_2 \in R^{10}$, $z \in R^{10}$ and σ is the ReLU activation function where $\sigma(x) = \max(0, x)$

- 4 Deep MLPs with two hidden layers

I chose the size of the second hidden layer to be half that of the first. Hidden layer sizes are $\{(512, 256), (256, 128), (128, 64), (64, 32)\}$ and the model is described as: $z(x) = W_3\sigma(W_2\sigma(W_1x + b_1) + b_2) + b_3$, where $x \in R^{784}$, $W_1 \in R^{hid\_dim1\times784}$, $b_1 \in R^{hid\_dim1}$, $W_2 \in R^{hid\_dim2\times hid\_dim1}$, $b_2 \in R^{hid\_dim2}$, $W_3 \in R^{10\times hid\_dim2}$, $b_3 \in R^{10}$, $z \in R^{10}$ and σ is the ReLU activation function.

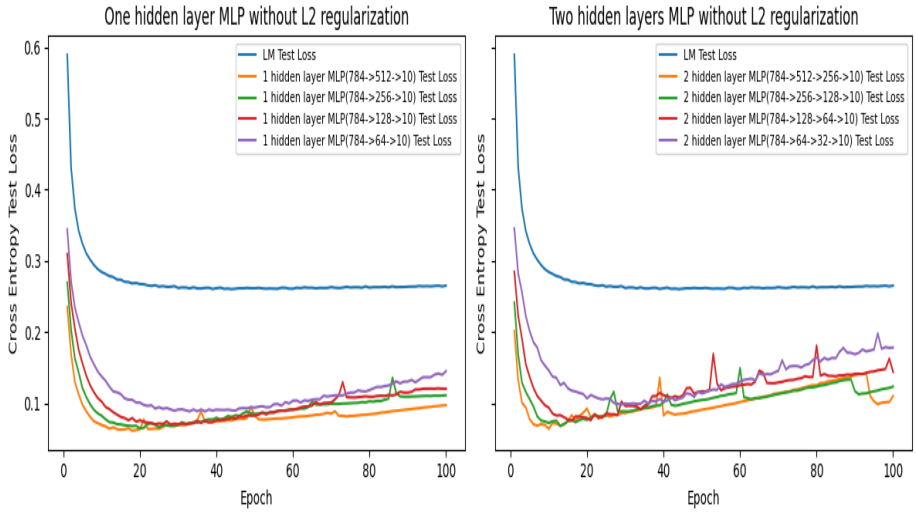

Figure 1: Effect of layer size on test loss

Although the best practice is to split the training set into training and validation set and track the loss on the validation set to evaluate performance, I followed the task's instructions and computed the loss on the test dataset. Based on the graph, it can be clearly seen that the Linear model underfits and it doesn't have enough capacity to learn more complex patterns, with the CE loss converging slightly under 0.3. Training for longer doesn't change the loss much because the model cannot learn to distinguish the nonlinear features and it can only represent linear decision boundaries, so it cannot memorize the nonlinear features of the training dataset and overfit. Among MLPs of the

same depth, those with wider layers seem to perform better, reaching a lower minimum test loss and with fewer epochs of training needed. All MLP curves drop fast early, reach their minimum test loss in a certain region (this is where the best generalization happens) and then the loss slowly starts to rise, showing overfitting, as the model learns to adapt too much on the training dataset and lose generalization power. Regarding depth, as can be seen from the graphs below, deeper networks reach their minimum test loss with fewer epochs of training but overfit harder and faster compared to the shallow MLPs which reach their minimum loss after more epochs and then start to overfit smoother than the deeper ones. For wider layers, the minimum test loss between shallow and deep MLPs is almost identical (the shallow MLPs achieve a slightly lower minimum loss) but shallow MLPs need more epochs to achieve that. Models with less wide layers achieve better performance on the shallow MLPs. Deep MLPs overfit harder as they have more capacity and they can easily mimic the training dataset and lose generalization power. There are also more spikes in the loss of Deep MLPs as epochs rise and overfitting happens, because they have more capacity and they become more confidently wrong and CE penalizes that. It can also happen due to stochasticity of the train loader, as each epoch used to train is a bit different from the previous one (because shuffle = True).

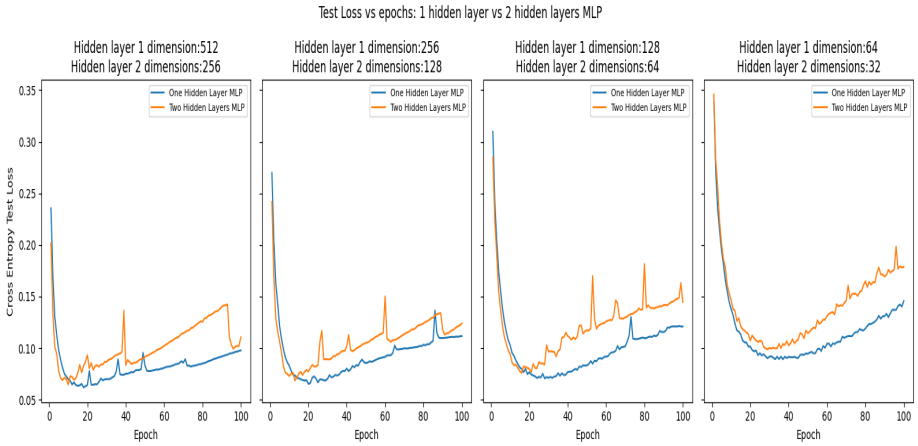

Figure 2: Effect of depth on test loss

3 Effect of L2 Regularization

To observe the effect of L2 regularization, I trained three different linear models: one with no regularization, one with low regularization ($\lambda = 1 \times 10^{-4}$) and one with high regularization ($\lambda = 1 \times 10^{-1}$) with a learning rate of 5×10^{-4} over 30 epochs. L2 regularization penalizes weights and shrinks them, depending on the value of λ and the value of the weights. After L2 regularization the loss

function becomes:

$$L(W, b) = L_{CE}(W, b) + \lambda \sum_{i=1}^{10} \sum_{j=1}^{784} w_{i,j}^2$$

In the linear model, the weight matrix $W \in R^{10\times784}$, where each row corresponds to the weight vector used to compute the logit score for class k:

$$z_k(x) = w_k^\top x + b_k,$$

where $w_k \in R^{784}$, $x \in R^{784}$ and $b_k \in R$. Positive values of w_k increase the logit score for class k, providing evidence that the input belongs to class k, whereas negative weight values provide evidence against it. To show this more clearly, we reshape each row of the learned weight matrix corresponding to each class into a 28×28 image, where red pixels indicate positive weights and blue pixels indicate negative weights. White pixels indicate weights that are 0 or close to 0. Each row of the following graph represents the learned weights with no regularization, low regularization and high regularization respectively.

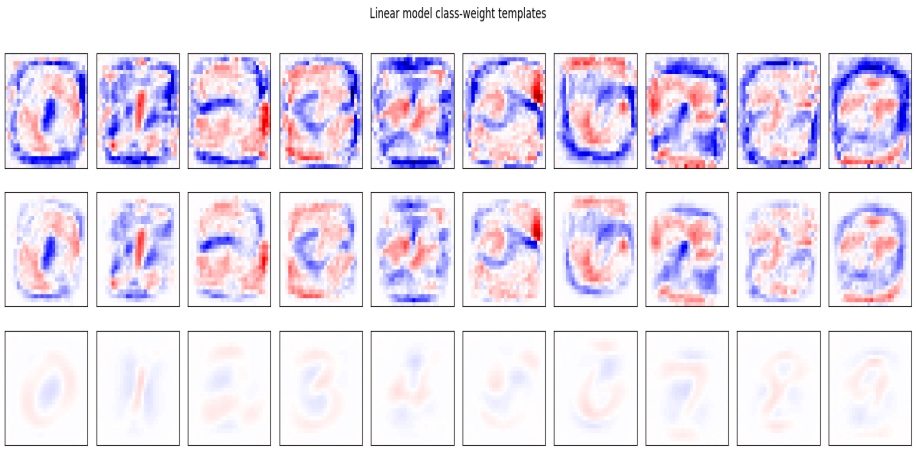

Figure 3: Visualization of learned weights

It is clear that on the model without regularization, weight values are greater in absolute value and that's why red and blue pixels are brighter and the contrast is higher compared to the models with regularization. Without regularization, templates also contain noise (as can be seen in the left corner of class 0 for example), as overfitting happens and learned patterns rely too much on training data which results in worse generalization. Due to extreme weight values, the model can fit too confidently the training data and generalize worse. When we apply low regularization (2nd row) we can see that the brightness of red/blue pixels drops as the weights shrink, making the model use the same patterns to detect classes but not rely too heavily on them and avoiding extreme weights resulting in better generalization. With high regularization(3rd row), the learned

class-weight templates become smoother and more structured, as the model suppresses noisy pixel weights and relies only on consistent patterns. However, it shrinks all weights too much and this can result in less confident predictions, decreasing the model's accuracy.

4 Evaluation and Confusion Matrix

As it was seen from previous graphs, the two best performing MLPs were the one hidden layer MLP with a hidden layer dimension of 512 ($784 \to 512 \to 10$) and the two hidden layers MLP with hidden layer dimensions 512 and 256 ($784 \to 512 \to 256 \to 10$). To decide between those two, I trained each model 5 times and compared the minimum test loss achieved as well as the epoch where it did so. I used a learning rate of 5×10^{-4} and trained for 20 epochs as the previous graphs suggested that minimum test loss is achieved for both models prior to 20 epochs. Four out of five times, the one hidden layer MLP performed better with a mean test loss of 0.0618 and a standard deviation of 0.001 achieved at epochs $13, 15, 15, 15, 19$ respectively, while the Deep MLP had a mean test loss of 0.0641 and a standard deviation of 0.0028, reaching its minimum loss at epochs $10, 10, 7, 9, 7$ respectively. It is clear that the deep MLP reaches its minimum faster compared to the shallow one. However, since the shallow one outperformed the deep MLP, I chose this as the best performing model and stopped the training at epoch 15. I then computed the total accuracy of the model which was 98.05% as well as the per class accuracy to plot the confusion matrix seen below. To compute the accuracy of the model, on the test dataset, I selected the argmax of each row of the output of the model, as this corresponds to the class which the model assigns the greatest logit score for each input and therefore the class it gives the greatest probability, since softmax function returns for each logit z_k in a row,

$$softmax(z_k) = \frac{e^{z_k}}{\sum e^{z_i}} \ for \ i = 0, \ldots, 9$$

In the confusion matrix below, each row corresponds to the true class while each column corresponds to the predicted class of the model. The diagonal represents the per class accuracy. It is clear that the model's accuracy is overall very high, with the lowest accuracy observed for 9, which may be explained due to the fact that it resembles $3, 4$ and 7 and we can see that the percentage of classifying 9 as $3, 4$ or 7 is higher compared to other misclassifications ($0.008, 0.013$ and 0.007 respectively). It is also notable that 5 is sometimes classified as 3 (0.011) which makes sense as they share some edges.

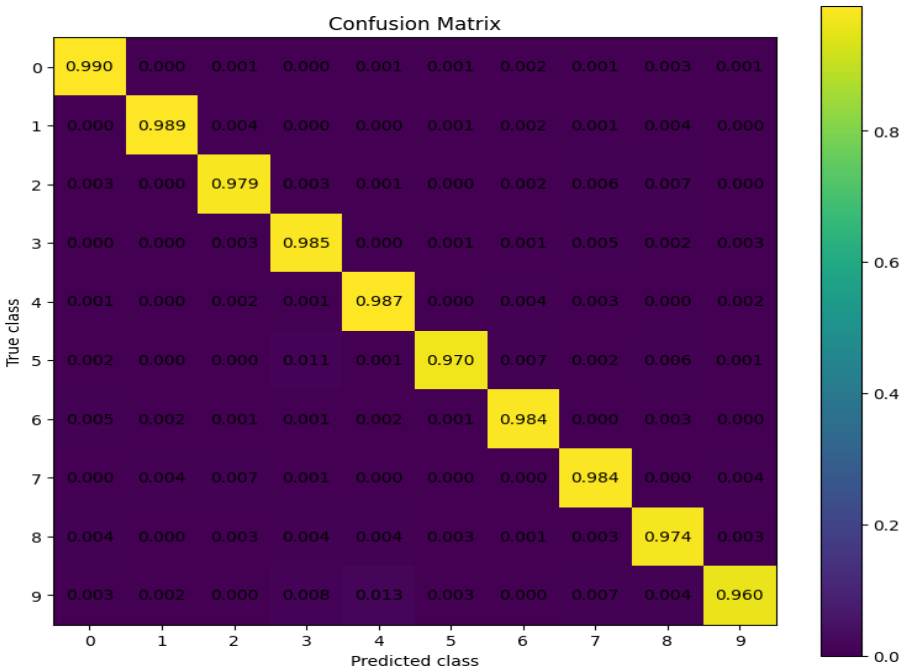

Figure 4: Confusion Matrix

5 Regularization and Overfitting

To select the optimal regularization on the one hidden layer MLP with a hidden layer dimension of 512, I trained the model with different λ values and plotted train vs test loss and train vs accuracy over epochs. The loss function of this model after regularization becomes:

$$L(W_1, W_2, b_1, b_2) = L_{CE}(W_1, W_2, b_1, b_2) + \lambda \left(\sum_{i=1}^{512} \sum_{j=1}^{784} w_{i,j}^2 + \sum_{i=1}^{10} \sum_{j=1}^{512} w_{i,j}^2 \right)$$

$W_1 \in R^{512\times784}$,$b_1 \in R^{512}$, $W_2 \in R^{10\times512}$, $b_2 \in R^{10}$, $\lambda \in R$

The λ values I chose were: 0, 3×10^{-3}, 1×10^{-3}, 3×10^{-4}, 3×10^{-5}. I trained for 100 epochs with a learning rate of 5×10^{-4}. The left graph depicts training vs test loss over different λ values, while the one on the right depicts training vs test accuracy. Dotted lines show training loss/accuracy, while solid lines show test loss/accuracy and lines of the same colour correspond to the same λ value. Circular marks show points of minimum loss/maximum accuracy while vertical lines show the epoch where this was achieved.

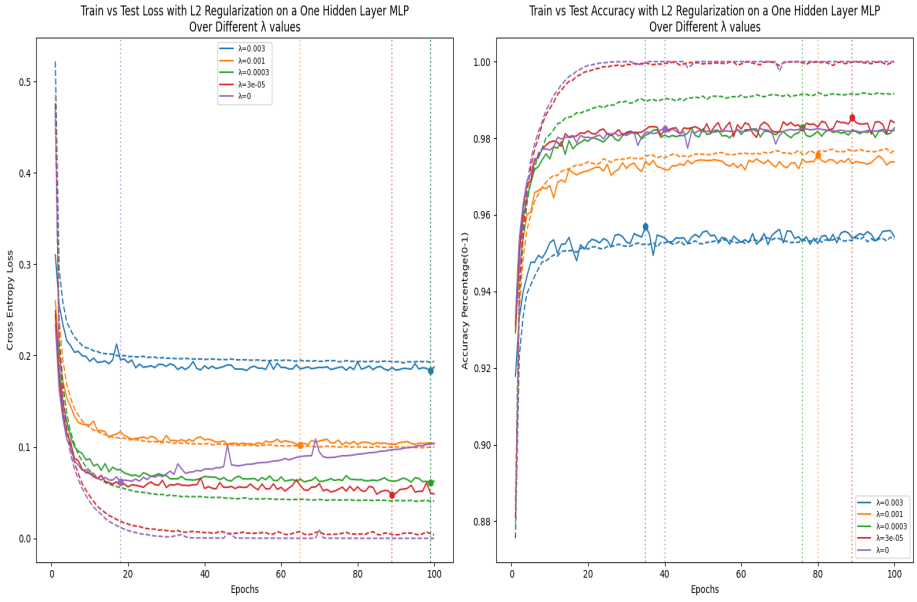

Figure 5: Effect of L2 regularization over different λ values

First of all, it's clear that adding regularization helps avoid overfitting, as the only test loss curve that shows a clear upward trend after hitting its minimum is the one without regularization (purple), while the training loss on that converges close to 0. All other lines show that the test loss plateaus around its minimum with some spikes but do not show a clear upward trend after they hit

their minimum. However, the strength of regularization has a strong effect on the performance of the model. With $\lambda = 3 \times 10^{-5}$, we achieve the lowest test loss(0.048) on epoch 89 and maximum accuracy(0.985) on the same epoch, and it seems to be the best-performing model as it allows for the smallest test loss (and highest accuracy) which doesn't show clear signs of overfitting as epochs increase. That means it has the best generalization power. Moderate regularization(green line) impacts the training loss, making it converge to a higher value compared to the ones with low or no regularization, as the model cannot memorize the training dataset as well due to the shrinking of weights. However, it's clear that we avoid overfitting, reaching a very good minimum test loss on epoch 99 (0.0612) which is close to the minimum test loss of the model without regularization(0.0614). This model also achieves a very good accuracy (0.982). Stronger regularization (orange and blue lines) leads to underfitting as training loss stops improving early and stays relatively high and test loss is much higher compared to other models, while accuracy is also lower. It's also worth noting that the model without regularization reaches its optimal status way earlier during training(epoch 18 for test loss, epoch 40 for accuracy) compared to other models with regularization. This happens because it can fit the training data aggressively and become confident quickly, while regularization causes weights to shrink slowing down this process and delaying overfitting. Therefore, for this task I choose the model with L2 regularization and $\lambda = 3 \times 10^{-5}$ as the best performing.

6 Per class accuracy during training

To visualize how the model learns to recognize different digits throughout training, I chose the one hidden layer MLP with L2 regularization($\lambda = 3 \times 10^{-5}$) and computed the accuracy per digit over epochs. I trained the model with a learning rate of 5×10^{-4} over 100 epochs.

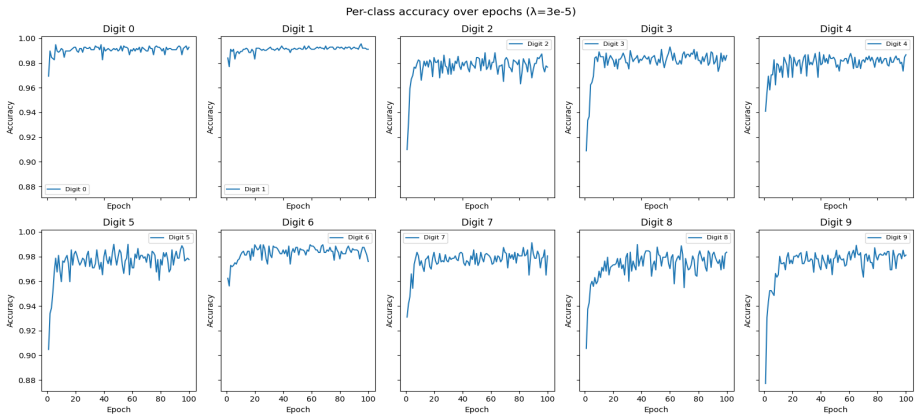

Figure 6: Digit accuracy vs epochs

It's clear that digits 0 and 1 are the easiest to learn, as accuracy fluctuates at 0.99 without huge spikes and it took only a few epochs to reach that. This might happen because 0 has a unique structure compared to other digits, as it forms a closed loop with an empty space in the middle. 1 also in handwritten form is usually just a vertical stroke, so it is easily distinguishable from other digits. On the other hand, digits $5, 8$ and 9 seem to be the hardest to learn as it takes more epochs to reach optimal accuracy and it fluctuates harder from one epoch to another, as the accuracy might change due to small weight updates over epochs. This might happen because these digits can be written in different ways and they may become more indistinguishable. For example, depending on how it's written, 9 might resemble 4 or 7 , while 5 might be confused with 3 and 8 might be confused with 3 or 9.