A Study of Feature Engineering and Neural Networks on the MNIST Data Set

Ahmad Hamzeh
Washington University in St. Louis
Introduction to Electrical and Systems Engineering
St. Louis, MO, USA
a.h.hamzeh@wustl.edu

Raynah Fandozzi
Washington University in St. Louis
Introduction to Electrical and Systems Engineering
St. Louis, MO, USA
f.raynah@wustl.edu

***Abstract*—With the end of the course in sight, we were tasked with the challenge of classification. The focus was on the MNIST dataset, a large dataset containing thousands of handwritten digits, often used as an introduction to the practice of computer classification. Initially, we applied concepts developed in lectures and the textbook such as feature engineering and least squares to train a weight vector to classify the data. Building on this, we sought to enhance our understanding and accuracy by exploring more sophisticated solutions. Our study shows a clear progression emphasizing feature engineering, the impact of preprocessing and the ability for neural networks to optimize weights and biases. Leveraging advanced concepts in calculus and linear algebra, we went beyond the case study requirements by coding a basic neural network, aiming to achieve a more accurate technique and to deepen our understanding of classification methods.**

I. Introduction

Classification is a cornerstone of modern computing. Whether it be a large-scale language model transformers or, like in our case, a small multi-class classifier, computers are tasked with interpreting the world around them. Unlike humans, computers lack the basic intuition to recognize patterns from plain observation. This problem will be addressed in the following study.

Our research highlights the sophistication and abstraction involved in teaching computers to classify data effectively, in this case, utilized the MNIST database of digits for this task[1]. We begin with the foundational methods introduced during lectures such as data preprocessing, feature engineering through convolution, average pixel intensity and frequency and post hoc analysis with a confusion matrix. Then, building on these principals, we will expand our approach to explore more advanced solutions. This will illustrate the progression from fundamental concepts to sophisticated techniques in computer classification.

II. Methods

A. Data Preprocessing

In each of the three parts, we were given 28x28 images that we flattened into 784x1 row vectors for ease of classification. Since, in parts 1 and 3, we are using a least squares classifier, the vector is more easily able to be processed. Flattening the images made it easier to process the data and let each row represent a distinct feature which was helpful in creating the weight vector to classify unknown images. This was also helpful later when constructing the feature matrix specifically in cases where features were reduced to a single value per image (frequency and intensity).

Additionally, in parts 2 and 3, we played around with normalizing the data. We did this through a z-score normalization (1) where we used the mean and standard deviation of all of the training data. This helped make the data more consistent across different images.

$$z = \frac{x - \mu}{\sigma} \tag{1}$$

Next, we then Winsorized the data by setting all extreme values to either -3 or 3, whichever value they were closest to. We did this in order to get rid of outliers that may skew our weight vector during training. We also chose to use the training mean and standard deviation for the testing and digitClassifier function. This would make sure that that everything is consistent, since the weight vector was trained off the training data.

B. By-Hand Model Training

In part two of the case study, we constructed a naive decision boundary for a specific digit, in our case, we chose 0. To create the weight vector, we used a for loop to determine the frequency of different pixels for all the images of digit 0, similar to how we did for the feature engineering in part 3. These frequencies were the averaged by dividing them by the total amount of 0 images. We then created a threshold value to determine which pixels to use in the weight vector. We decided that this threshold would be the median of all the frequency values. This decision needed deliberation since it balanced the relationship between the amount of 0 digit's classified accurately, and the amount of other digits classified incorrectly because the threshold value was too low.

After this, to improve our model, we decided to preprocess the flattened data by using the z-score normalization as described in Section II Part A. This improved our accuracy rate in classifying the 0 digit while not increasing the false classifications for other digits.

C. Least Squares Model Training

The goal of this case study is to be able to create an algorithm that successfully assigns a classification as an output given the input of a handwritten digit. We utilizing equation (2) to do this.

$$X^T w = \hat{Y} \tag{2}$$

Given that X represents the data from the images, Y represents the actual labels of the images, $\hat{Y}$ represents the predicted labels of the images, and w is the weight vector we aim to construct. This is where least squares comes in. Least squares is a method used to minimize the error between predicted and actual outputs (3) and (4).

$$\min_{w} \|X^T w - Y\|_2^2 \tag{3}$$

or

$$\min_{w} \|\hat{Y} - Y\|_2^2 \tag{4}$$

By using least squares to reduce the discrepancy in the actual digit versus predicted digit, it can help in creating an effective algorithm. We can solve for w (6) using the Moore-Penrose pseudo-inverse $X^\dagger$ (5).

$$X^\dagger = (X^T X)^{-1} X^T \tag{5}$$

$$w = X^\dagger Y \tag{6}$$

We used this method in parts 1 and 3 of the case study. In part 1, we used the flattened raw data for X to train a weight vector. But, even after using least squares, when we used the trained weight vector to classify other digits outside of the training set, the accuracy rate was very low. This could be the case for multiple reasons. First, we only used 1000 images to train our weight vector, which is not very much compared to the 24000 images we used to train the matrix in parts two and 3. Another issue is that the data is not altered in any way, meaning it is not normalized or processed outside of flattening it. We found that the algorithm had a hard time in distinguishing between digits that were similar such as 7 and 2, 8 and 4 and 3 and 2. This was determined through post hoc analysis using a confusion matrix [Section II, Part C].

In part 3 of the case study, we implemented more feature engineering to improve the methods used in part 1. The goal was to be able to better utilize the least squares method to classify the digits. We used four features to do so: the raw flattened image, the convolved image, the average frequency and average intensity. We used the raw flattened image as a feature as we did in part 1. For the next feature, we decided to focus on the edges of the image by using convolution. We decided that the best way to convolve the image after trial and error was to use the Sobel operator for the horizontal and vertical edge detection.

$$x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

This was important since the features found through convolution are not as dependent on the position of object as the raw image is. Features are still able to be detected if they are the edges. For the next feature, we used the average frequency of pixels that were non zero on each image. For the final feature, we used the average intensity of the actual value of each pixel for every image. For the last two features, average frequency and average intensity, we did it across the entire image rather than doing them for each pixel. There is something to say between the accuracy increasing if we were to do that, but at the cost of time to code and simplicity of the code. Then, we concatenated all of these features into a matrix so that each image we trained the weight vector with had the four features listed above. When we used least squares, we created a weight vector that was much more accurate than the one trained in part 1.

After this, we then tried to add a bias term b alongside the weight vector to shift the model output independently of the input features (7).

$$Y = X^T w + b \tag{7}$$

This could help account for non-zero offsets that the model cannot account for. We tried doing this and ultimately failed to increase the accuracy rate since every time we attempted to add a bias term, the accuracy rate decreased. This could be due to many things. First, the model could not be over fitting the data for the training data which means it would fail to successfully classify the testing data. This could also be due to the fact that the relationship between the inputs and the outputs are already well modeled and do not need a bias term. For those reasons, we did not include one in our model.

D. Confusion Matrix and Post-Hoc Analysis

In creating a classifier, a good way to determine where the classifier is making mistakes is looking at a confusion matrix [Fig. 1].

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Fig. 1. Confusion matrix showing predictions and actual outcomes.

This confusion matrix has the predicted output as the rows, in our case this is $\hat{Y}$ and the actual output Y as the columns. False positive means that the model predicted the value to be positive while it was actually negative, and the opposite goes for false negative. Our goal is to get the most values in the true positive and true negative elements since those indicate that the classifier did its job correctly. In context to this case study, the confusion matrix would be a 10x10 matrix, where the diagonal would be the correctly identified digits 0-9. Outside the diagonal, we have the number of inaccurately classified digits. For example, the value in (8,3) in the matrix would be the amount of images that were classified as 2 but were actually 7. This is a specific example is a common error that the weight vector makes since the 2 and the 7 digit have similar features.

We used this method to complete a post-hoc analysis of our classifier and backtrack to change it based on where it was going wrong. We used this analysis in part two of the case study, since there was a relationship between the accuracy rate of classifying the 0 digit, and the false positive and false negative values. As we lowered the threshold value for including a pixel in our naive weight vector, the accuracy rate of increasing the 0 digit increased which makes sense since more digits would pass as 0. The issue was that as we decreased the threshold, the false positive and false negative values increased since an image would not have to match the weight vector as much to be classified as 0.

We did a post hoc analysis using the confusion matrix for part 3 of the case study as well. Once we trained our weight vector, we used it to classify the testing data and produce a confusion matrix [Fig. 2].

	0	1	2	3	4	5	6	7	8	9
0	775	0	3	2	1	2	3	0	4	0
1	0	876	4	1	1	2	4	0	5	1
2	15	18	708	21	13	1	9	17	28	3
3	9	5	13	694	1	15	3	19	25	7
4	1	9	9	1	682	2	9	3	8	54
5	14	6	2	29	20	609	20	7	15	6
6	9	3	6	2	11	21	732	0	2	0
7	5	18	14	8	17	2	0	696	1	44
8	18	15	4	27	17	18	10	5	660	11
9	10	5	5	14	33	1	1	30	10	701

Fig. 2. Confusion matrix for the part 3 classifier. Rows represent actual labels, and columns represent predicted labels.

In this matrix, you can see that the number classified with the largest error was the digit 4, where the classifier mistook the digit for 9 54 times. To analyze this further, we took a look at some of the images the weight vector was misclassifying to determine what the similarities were and how we can fix this.

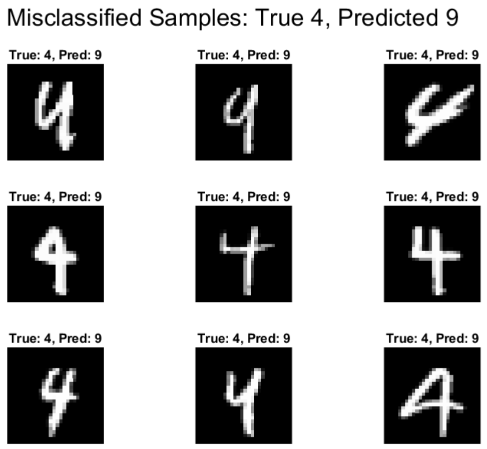


Fig. 3. Misclassified digit 9 as digit 7, showing similarities between misclassified images

Based on these images, we can see that the common mistake stems from the four being nearly connected at the top, similar to a 9. This mistake makes sense and ideally we would want to put more weight on the top of the image when classifying between 4's and 9's.

III. Neural Networks

A. Overview

The process described earlier may be somewhat effective but is it scalable? In today's computing heavy scene, it is in our best advantage to leverage computers to automatically compute and optimize weights and biases in an unsupervised fashion. Luckily for us, we can use calculus and clever linear transformation to achieve optimized weights and biases that boost our classification potential to a large-scale level.

B. Layers, Activation and the Forward Pass

A neural network at its core is just a "fancy" combination of linear transformations from an input layer to an output layer. Our input layer here is of dimensions 784x1, which corresponds to our training image vector x_i. Our first layer weight vector $w^{[1]}$ is of dimensions 16x784. We apply a linear transformation and add a bias vector $b^{[1]} \in R^{16x1}$ to form the hidden layer of our neural network, $z^{[1]}$ as shown in (8):

$$z^{[1]} = w^{[1]}x + b^{[1]} \tag{8}$$

Before we blindly apply another linear transformation to the next layer, it is critical to stop and think about what exactly is happening. If we continue to another layer without changing anything, then we've basically done nothing meaningful to warrant the use of a hidden layer, and the sophistication of our network is lost. So, before we continue, we apply an activation function to our network. In our case study, we Rectified Linear Unit (ReLU) activation function (9).

$$\text{ReLU}(x) = \begin{cases} x & \text{for } x > 0, \\ 0 & \text{for } x \leq 0. \end{cases} \tag{9}$$

This function operates element-wise on $z^{[1]}$, and we define the resulting matrix to be $a^{[1]} = ReLU(z^{[1]})$. Now, we can apply a linear transformation on $a^{[1]}$ using new weight and bias vectors to get $z^{[2]}$ (10).

$$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}, w^{[2]} \in \mathbb{R}^{10\times16}, b^{[2]} \in \mathbb{R}^{10\times1} \tag{10}$$

Finally, to normalize $z^{[2]}$ into probabilities for our classifier, we apply the softmax function on $z^{[2]}$ to get $a^{[2]}$.

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{11}$$

$a^{[2]} \in R^{10x1}$ is now a vector with probabilities in each entry which corresponds to a digit in the base 10 number system. Our weights and biases were randomly initialized, so it goes without saying that this classification here is very poor. This leads us into the discussion of optimization.

C. Minimizing our Cost using Back Propagation

First, we must find a way to compare labels to our output $a^{[2]}$. We did this by first converting our labels into a vector $y_i \in R^{10x1}$ where every index except the label + 1 index is set to 0, and the label + 1 index is set to 1. This process is known as one-hot encoding. For example, if the ith element of the labels is 5, then $y_i = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]$. In our model we are trying to minimize (12).

$$cost = \|a^{[2]} - y\|^2 \tag{12}$$

We did this by finding the gradient of the cost with respect to the weights and biases. This gradient represents how a change in the weights or biases will increase the cost. So, if we subtract away our gradient, we will begin to reduce the cost. Now, we calculate the gradients for the weights and biases in both layers using the chain rule (13, 14).

$$\frac{\partial C}{\partial w^{[2]}} = \frac{\partial z^{[2]}}{\partial w^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial C}{\partial a^{[2]}} \tag{13}$$

$$\frac{\partial C}{\partial b^{[2]}} = \frac{\partial z^{[2]}}{\partial b^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial C}{\partial a^{[2]}} \tag{14}$$

Now, we use calculus on the equations described above. For simplification purpose, we will assign $\frac{\partial a^{[2]}}{\partial z^{[2]}} = 1$ for the output layer because using least squares changes the compatibility of the gradient of softmax with the cost. As shown in our findings, the network will still converge.

$$\frac{\partial z^{[2]}}{\partial w^{[2]}} = a^{[1]}, \frac{\partial C}{\partial a^{[2]}} = 2(a^{[2]} - y), \frac{\partial z^{[2]}}{\partial b^{[2]}} = 1 \tag{15}$$

Next, we follow the process of back propagation using the equations in (16).

$$\frac{\partial C}{\partial a^{[1]}} = \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial C}{\partial a^{[2]}} \text{ and } \frac{\partial z^{[2]}}{\partial a^{[1]}} = w^{[1]} \tag{16}$$

Before we move on the next step, we must note that the derivative of ReLU is

$$\text{ReLU'}(x) = \begin{cases} 1 & \text{for } x > 0, \\ 0 & \text{for } x \leq 0. \end{cases} \tag{17}$$

A similar process is applied to the first layer using the aspects of the gradient from the layer above. This is where the idea of back propagation comes into effect. For our first and second layer gradients, respectively, we get:

$$\frac{\partial C}{\partial w^{[2]}} = a^{[1]} * 2(a^{[2]} - y) \tag{18}$$

$$\frac{\partial C}{\partial b^{[2]}} = 2(a^{[2]} - y) \tag{19}$$

$$\frac{\partial C}{\partial w^{[1]}} = w^{[2]} * 2(a^{[2]} - y) * ReLU'(z) * x_i \tag{20}$$

$$\frac{\partial C}{\partial b^{[1]}} = w^{[2]} * 2(a^{[2]} - y) * ReLU'(z) \tag{21}$$

Our goal with this process is to converge to a local minimum for our cost function. So, after calculating the gradients for our weights and biases, we subtract it from the weights and biases and apply a learning rate, α. This learning rate is applied because it prevents us from overshooting our local minimum. An example is shown in (22).

$$w^{[2]} = w^{[2]} - \alpha \frac{\partial C}{\partial w^{[2]}} \tag{22}$$

Each iteration over this process is known as an epoch. However, the approach listed out above is computationally expensive, and will require an immense amount of time if we only go example by example. So, we leveraged the computational efficiency of matrix operations in MATLAB to process each epoch using the entire batch of data. Furthermore, the model can be trained on a CPU, but we also leveraged the capabilities of an nVIDIA GPU in MATLAB to speed up training time (both, the CPU and GPU files will be turned in). The model will converge after a certain amount of epochs as explained in the next section.

D. Analysis

As stated before, the goal of this algorithm is to adjust weights and biases in order to reach a local minimum of the cost function. This is achieved by iterating over the forward pass and back propagation steps over many epochs. Our group found that the optimal number of epochs is in the range of about 400-500. One could argue that letting the algorithm run for longer would produce better testing results, but it actually ends up over-fitting the training data which makes it suboptimal for classifying the testing data. The figure below shows how the cost is minimized over epochs.

Our set of weights and biases classified digits by following the same forward pass that is described in part B. At the end,

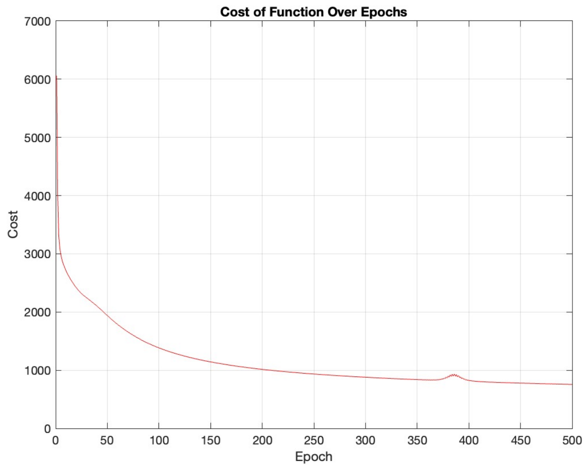
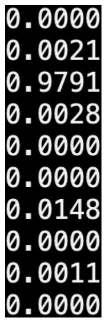


Fig. 4. Cost of Function Over Epochs

we picked the maximum argument of our resulting 10-vector, which is the computers best prediction of what the number is.



```
0.0000
0.0021
0.9791
0.0028
0.0000
0.0000
0.0148
0.0000
0.0011
0.0000
```

Fig. 5. Sample Output Vector of a Classification of a Handwritten Two

We observed a 90% successful classification rate with this model. We also produced another confusion matrix for this model [Fig. 6].

	0	1	2	3	4	5	6	7	8	9
0	760	0	1	3	1	12	7	1	5	0
1	0	870	3	5	0	1	3	1	11	0
2	13	17	712	21	15	5	12	8	27	3
3	2	4	24	688	0	25	2	14	23	9
4	0	1	6	0	699	1	12	2	7	50
5	14	4	7	23	12	604	19	9	29	7
6	8	3	15	4	12	10	724	0	7	3
7	2	7	24	6	7	1	0	722	6	30
8	8	5	11	20	6	27	14	9	674	11
9	8	3	5	7	53	11	0	33	10	680

Fig. 6. Confusion matrix for multi-layer perceptron neural network. Rows represent actual labels, and columns represent predicted labels.

As we can see, our network was very successful on classifying digits, but there does exist some confusion in classification. In particular, we can see some confusion when it comes to the network classifying 4's and 9's which makes sense given their similar shaping.

Another difference in our two approaches is the feature extraction exhibited in the models. In the neural network, the computer is automatically optimizing weights that represent certain pixel values. This can produce chaotic looking images.

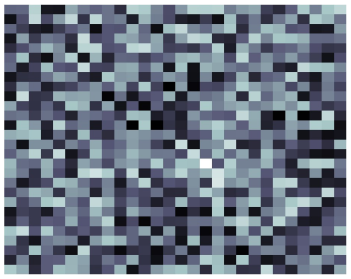


Fig. 7. Example of weights matrix

However, within this some patterns can be observed such as the slight light curve on the middle right which could represent a feature of the digits.

IV. Results and Discussion

The results of this case study provide insights into the performance of both foundational and advanced classifiers to demonstrate the progression from feature engineering to neural networks, highlighting key insights and challenges we faced along the way.

A key metric we used to evaluate progress throughout the study was the confusion matrix. Not only does it show the accuracy of the classifier in determining specific digits, but also shows where the classifier went wrong. Through this, we are able to complete a post hoc analysis to understand the classifier's strengths and weaknesses, identifying where improvements can be made.

However, in part 3 of our case study, we ran into an issue: translating the information from the confusion matrix into tangible changes in the weight vector. This difficulty stemmed from the way the weight vector was constructed. In parts 1 and 3, we used a pseudoinverse and least squares method to let the computer find the most efficient solution based on the input data. In contrast, part two involved manually constructing a naive weight vector by hand just based on the common patterns we saw for the digit zero.

The limitations of our entirely automated approach became apparent when we identified the reoccurring error in part 3: confusion between digits 4 and 9. While the confusion matrix helped us find this issue, actually adjusting the weight vector proved challenging because it was completely determined by the computer's calculations. We realized that this issue came from the rigid nature of the pseudoinverse and least squares models, which has no flexibility when trying to incorporate human intuition into the solution.

To address this, we propose a hybrid approach: starting with a manually constructed naive weight vector, like in part two, to ensure that initial insights and patterns show up then followed by the least squares algorithm in MATLAB to optimize the solution, like in parts 1 and 3. This combined method could make a more flexible and accurate classification algorithm by integrating human intuition with computational efficiency.

Another relationship we addressed in the case study, across both the feature engineering and neural network strategies, was the relationship between complexity and efficiency. In feature engineering, this relationship showed up with the amount of features used versus the resulting accuracy rate.

Initially, we experimented with the average frequency and intensity features for the 784 pixels in each image. Our first approach involved calculating the average frequency and intensity for all individual pixels, but we quickly realized this quickly realized that would be too much complexity on top of the raw data and the convolved data. To simplify, we then went to a more broad approach and only calculated a single average frequency and single average intensity value for the entire image. This resulted in an accuracy rate of 89.14%.

Next, we decided to split the image into four quadrants and calculated the average frequency and intensity values for each quadrant. This increased our accuracy rate to 89.16%. We then further split the image into 16 sections and calculated the average intensity and frequency for each of them. This increased our accuracy rate to 89.19%. While adding more features and complexity led to an increase in accuracy rate, the amount gained was minimal compared to the additional data and computing needed. We ultimately decided to go back to using a single overall average for frequency and intensity per image because of the lack of significance in the accuracy rate improvement.

As for the neural networks, we couldn't run through every single computation as it would take an immense amount of time. We sacrificed precise gradient calculations in order to develop an efficient batch processing algorithm that also ultimately resulted in convergence and the correct classification of 89.53% of the 8000 test examples.

V. Conclusion

In this case study, we explored multiple different approaches to classification each with its own strengths and trade offs. The choice of method ultimately depends on the specific need you have for the application. For example, if you want a scalable large language models, neural networks may provide you with the tools you need. On the other hand, if you need to fine tune your model for mission critical applications, you may opt to explore feature engineering.

All in all, this case study showcases how leveraging linear algebra in the right way has powerful capabilities. Although our focus in this study was classifying handwritten digits, the same models we explored - feature engineering, least squares optimization, and neural network - extend far beyond this example. These same methods are being used to solve real world problems spanning from medical diagnosis to autonomous driving.

For instance, the back propagation algorithms developed in part 3 serves as a commonly used method for modern AI systems like ChatGPT. This case study serves as a foundational step in understanding the potential of these technologies, offering our first look into the technology of the future.

References

[1] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST database of handwritten digits," [Online]. Available: http://yann.lecun.com/exdb/mnist