Building Models for Image Classification

Clare Gartz

Advisor: Mr. Villegas

December 7, 2025

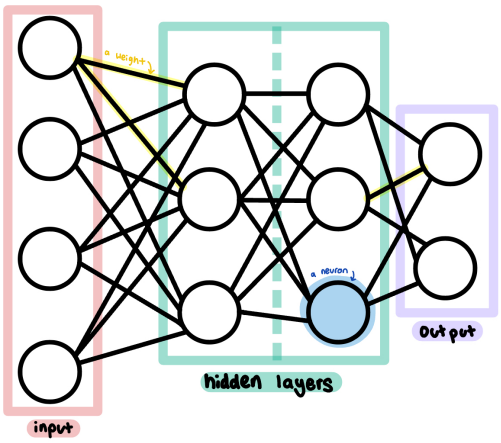
Table of Contents

1 Understanding Neural Networks and the MNIST Dataset

1.1 The Goal

The central objective is to understand how a program can learn to identify the values of different handwritten digits accurately. I am employing a basic neural network (NN) to achieve this with the MNIST dataset. This project examines the contribution of linear algebra and multivariable calculus concepts to the training process. The performance of the model is based on the accuracy and confidence in the classification of the validation data. Here, accuracy refers to images correctly classified, with confidence. Confidence is interpreted as the probability output of the Softmax function.

1.2 Components of Neural Networks



Before diving into the training of NNs, it is essential to define the key components conceptually and mathematically.

A NN has three conceptual layer types, the first being the **input layer**. This layer contains the raw data you are looking to classify. The middle layers are called **hidden layers** (also called inner layers), where most of the math happens. The last layer is the **output layer**. The output layer is where the final "answer" is determined. The structure of the input and output layers generally have a fixed format, but the number and size of hidden layers is adjustable.

The **neuron** (also called the node) is a mathematical representation of a biological neuron. Similar to its biological counterpart, the artificial neuron receives inputs, operates, and then produces an output, which is passed to the neurons in the next layer. If an inner layer has size 6, it means there are 6 neurons.

In a NN, the **weight** (w) is the connection between a data point and a neuron, or between two neurons in adjacent layers. The weight holds a numerical value that represents the strength and influence of that connection.

The **bias** (b) is a constant that is added to the pre-activation, which shifts the output left or right. It acts as an intercept to help adjust the fit of the data.

The **activation function** (σ) is the mathematical function that transforms the output of all neurons. The activation allows for non-linearity, expanding the capability and complexity of a NN. The activation can be applied at a given neuron or across an entire layer.

The term inside the activation function (the weighted sum plus the bias) is called the **pre-activation** (z), in $a = \sigma(z)$.

At a single neuron, the equation for its activation is

$$a = \sigma(w_0 x_0 + w_1 x_1 + \cdots + w_{n-1} x_{n-1} + b)$$

or

$$a = \sigma\left(\sum_{i=0}^{n-1} w_i x_i + b\right)$$

At the first hidden layer, the activations can be represented through matrix multiplication.

$$\begin{bmatrix} a_0 \\ a_1 \\ \dots \\ a_{k-1} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n-1} \\ w_{1,0} & w_{1,1} & \dots & w_{1,n-1} \\ \dots & \dots & \dots & \dots \\ w_{k-1,0} & w_{k-1,1} & \dots & w_{k-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_{n-1} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \dots \\ b_{k-1} \end{bmatrix}\right)$$

In the hidden layers following the first, the activations are dependent on the values at the

previous neurons, rather than the input data. This means that a replaces x.

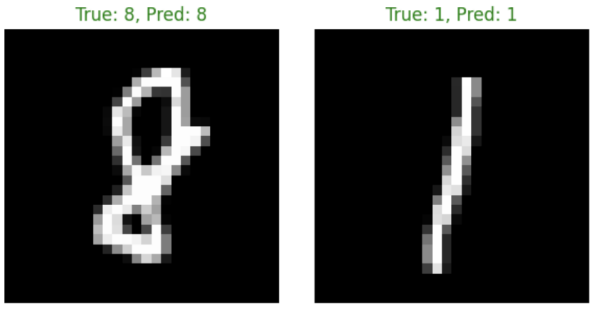
$$= \sigma \left(\begin{bmatrix} w_{0,0} & w_{0,1} & ... & w_{0,n-1} \\ w_{1,0} & w_{1,1} & ... & w_{1,n-1} \\ ... & ... & ... & ... \\ w_{k-1,0} & w_{k-1,1} & ... & w_{k-1,n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ ... \\ a_{n-1} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ ... \\ b_{k-1} \end{bmatrix} \right)$$

The **Sigmoid function**, the **Rectified Linear Unit function** (ReLU), and the **Hyperbolic Tangent function** (tanh) are common choices for σ. They all effectively transform the pre-activation into a non-linear function. The choice of which to use is typically dependent on computational cost, range, and gradient flow. In this project, I use ReLU because it is a simple activation function that computes quickly. There are more factors that differentiate between these, which can be important to consider for other projects. The equation for ReLU is

$$\text{ReLU}(z) = \max(0, z)$$

Besides these core concepts, there are other NN vocabulary terms worth defining. An **epoch** is one cycle through an entire dataset. So, three cycles would be three epochs. The **layer size** refers to the number of neurons in a given layer. The elements above can be categorized as parameters or hyperparameters. The **parameters** refer to components that are learned, like weights and biases. The **hyperparameters** refer to structural components, like the layer size, learning rate, and the number of inner layers. In Part 3, we explore how changing different hyperparameters can impact the accuracy.

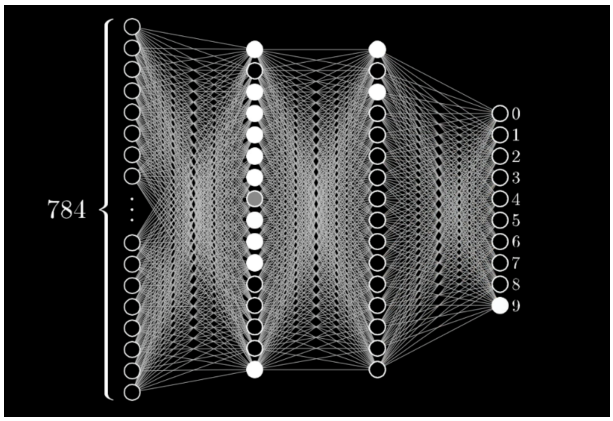
1.3 The MNIST Dataset



The Modified National Institute of Standards and Technology (MNIST) dataset contains 70,000 grayscale images (60,000 for training and 10,000 for testing) of handwritten digits, ranging from 0 to 9. The images are 28-by-28, so the total pixel count is 784. Each pixel contains a grayscale value, ranging from 0 (completely black) to 255 (completely white).

The MNIST dataset is often considered the "hello world" of machine learning and image classification. However, it has concrete value. Handwritten digit classification was initially driven by USPS's zip code identification. I chose this dataset for its simplicity and straightforwardness—perfect for an introduction to machine learning.

1.4 Visualizing the MNIST Dataset as a Neural Network



Combining the terms defined earlier with the MNIST dataset, the basic structure is set. The input layer will have 784 neurons made up of the pixels in a single 28-by-28 image. Each pixel contains a grayscale value, which we will transform to scale from 0 to 1 (rather than 0 to 255). The output layer will have 10 neurons, representing the digits 0 through 9. The output neuron with the highest value is the predicted digit. The image above shows this, with a random choice of two inner layers, each size 16.

2 The Learning

2.1 The Cost Function

From a basic standpoint, the **Cost function** (J), (also called the **Loss function**), quantifies error to measure the performance of a NN. Often the phrases Cost function and Loss function are used interchangeably, as I do here, but in reality Loss refers to error in a single training example while Cost refers to the average loss in a training set.

The Cost function compares the predicted value to the correct value. When the Cost function has a small value, it indicates high accuracy and minimum loss. Minimizing the Cost function is the goal of a NN. The input of the Cost function is essentially all of the weights and biases, so the "learning" in machine learning is finding the best weights and biases to minimize the Cost function.

In the output layer, I implemented **Softmax** as the activation function. This ensures that the values of each output neuron sum to 1, producing a probability distribution. This allows the output to represent the model's confidence. If there is 0.7 at node 1, and 0.3 at node 2, we can interpret this as 70% confidence in node 1 and 30% confidence in node 2. The equation is

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Categorical Cross-Entropy Loss function (Cross-Entropy) is typically used for classification with multiple mutually exclusive classes. The formula for Cross-Entropy is

$$J = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{k} y_{ij} \log(\hat{y}_{ij})$$

Where m is the number of examples, k is the number of classes, y_{ij} is the correct label, and $\hat{y}_{ij}$ is the predicted probability. The predicted probability refers to the values in the output layer, which Softmax allows us to view as probabilities. The ideal situation is to have high confidence in the single, correct output node for the given image.

There are many different types of Loss functions that can be used. I use Cross-Entropy for the MNIST dataset, but in Part 3, I explore different kinds of Loss functions through smaller, binary experiments.

2.2 Gradient

To explain the "learning," it is easier to step outside of NNs entirely. The essential concept to understand is the **gradient vector** (∇f). The gradient is the direction of steepest increase at a given location, which is found through the partial derivatives of the function. It is easiest to visualize this in 3D, using x, y, and z directions. With a function $f_{(x,y,z)}$, the gradient is

$$\nabla f(x,y,z) = \left\langle \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right\rangle$$

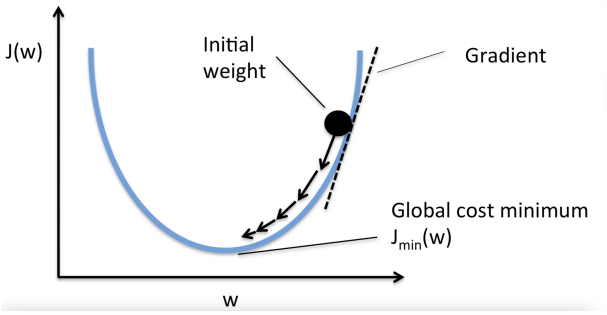
in vector notation. The gradient vector can also be represented with matrix notation

$$\nabla f(x,y,z) = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix}$$

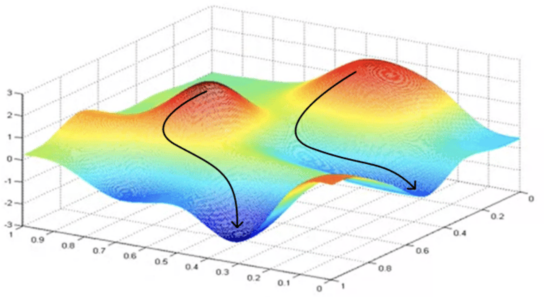
Because we will use the gradient to *minimize*, we want the opposite direction of the steepest increase, the direction of steepest decrease. Therefore, we really use $-\nabla f$.

2.3 Gradient Descent

The gradient is a direction, and **gradient descent** is the act of repeatedly taking steps in that direction. This means finding the gradient, taking a small step in the opposite direction, and then finding the gradient again. The common analogy used to describe the effect of gradient descent is ball rolling downhill, as it stops once it reaches a local minimum.



The image above shows what gradient descent looks like on a simple parabola. It continues to take steps until it reaches the minimum of the graph.

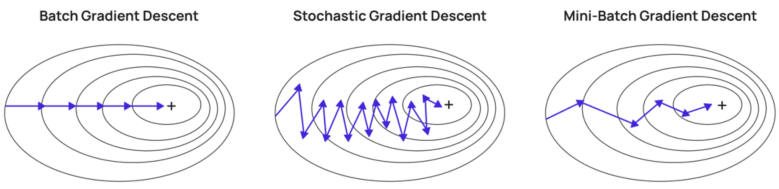


Gradient Descent can also be visualized in 3D. The image above shows a key part of gradient descent—where you start matters. Based on the starting point, there will be different paths and different final locations, as it leads to the local minimum, which is not necessarily the global minimum. While the image shows a solid line, it is really small steps or points that make up the path to the local minimum.

The **learning rate** (η) determines the distance between updates of the gradient, or steps. The learning rate can be described with the analogy of climbing down a mountain. The learning rate represents the size of a stride before reevaluating direction. Let θ represent any parameter (weight or bias). The update equation is

$$\theta_j^{\text{new}} = \theta_j^{\text{old}} - \eta \frac{\partial J}{\partial \theta_j}$$

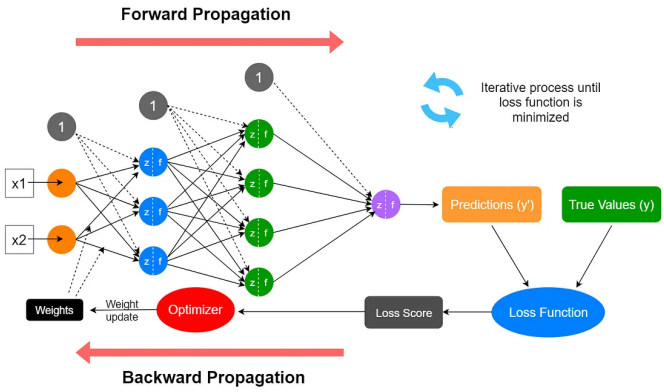
While I described gradient descent with 2D and 3D graphs, the MNIST dataset is in a much greater dimension. When applying gradient descent to the MNIST dataset and our network, the real inputs are all the weights and biases that make up the network, as well as the cost function itself. It is impossible to visualize this, so it is easier to think of it conceptually as 3D.



There are three typical types of gradient descent, depending on the amount of training data used before each update. **Batch Gradient Descent** uses all of the training data to find the gradient, so it is very slow and requires a lot of computing power. **Stochastic**

Gradient Descent (SGD) uses a single training example, making it fast, but possibly random. The most standard type is **Mini-Batch Gradient Descent**, which uses small sets of training examples. It is more stable than SGD and faster than Batch Gradient Descent, which is why I implement it in my project.

2.4 Backpropagation



Backward Propagation of Errors, shortened as **Backpropagation**, is essentially an efficient algorithm for gradient descent, by working backwards through all of the parameters. The image above shows the process of Backpropagation.

The first part of Backpropagation is the forward pass, where the input data is fed through the network, using the initialized weights and biases to produce the initial Cost. The second part is the backward pass, which uses the Cost and works backwards with the chain rule. The calculated partial derivatives are then used for gradient descent, to adjust the weights and biases accordingly. Backpropagation continues for as many epochs as you set. The final result is values of weights and biases for the NN to use on the validation data.

3 Code Implementation and Experimentation

3.1 The Setup

There are a few key steps in setting up the code, before getting into the training. First, it is important to note that my code is in Python. While there are a variety of libraries that can be employed, I just used numpy, pandas (to import the dataset), and matplotlib (to create graphs). There are libraries set up to make creating a NN easier, but I chose to use simpler libraries for the MNIST dataset, so I could show the math steps better. In the binary experiments, I explore other libraries. The setup is relatively simple.

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

data = pd.read_csv('train.csv')

data = np.array(data)
rows, cols = data.shape

labels = data[:, 0]
pixels = data[:, 1:] #this separates the row that contains labels
X = pixels / 255.0 #this transforms the grayscale values to scale 0-1
```

This code pulls the data and separates it nicely. This code accomplishes: uploading the dataset, shaping the data, separating the labels, and transforming the pixels to a 0–1 scale.

The next step is setting up one hot encoding. One hot encoding gets its name from electronics, where only one bit can be "hot," like an on/off switch. Initially, each image has a numerical label, like 0, 4, or 9. One hot encoding converts the label from a magnitude to a location on a matrix, so the value of the number has no weight in operations.

```
def one_hot_encoding(y, num_classes=10):
    m = y.shape[0]
    Y = np.zeros((m, num_classes))
    Y[np.arange(m), y.astype(int)] = 1

    return Y

Y = one_hot_encoding(labels)
```

This converts all labels into a matrices size 10 where every entry is a 0 expect for the location correlating to the label, which holds value 1. For example, the number 3 is represented by $[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]$. (Remember that we start counting at 0!)

Based on where I sourced the dataset, I only have a training file. The dataset needs to be split so some images can be used to train and other data remains unseen as validation data. It is important to have this validation data because it acts as an unbiased measure of performance, and prevents **overfitting**. Overfitting occurs when the model is tuned to the specifics of the training data and noise, rather than general patterns.

```
X_train = X
Y_train = Y

def split_training_data(X_train, Y_train, val_size=0.2, random_seed
    =42):
    #20% for validation and 80% for training
    m = X_train.shape[0]
    val_examples = int(m * val_size)

    #randomize so validation set represents all numbers with a
        relatively even distribution

    np.random.seed(random_seed)
    indices = np.random.permutation(m)
    print(f"Shuffled␣indices␣example:␣{indices[:10]}...")

    val_indices = indices[:val_examples]
    train_indices = indices[val_examples:]
    print(f"Validation␣indices:␣{len(val_indices)}␣examples")
    print(f"Training␣indices:␣{len(train_indices)}␣examples")

    X_val = X_train[val_indices]  # Validation images
    Y_val = Y_train[val_indices]  # Validation labels

    X_train = X_train[train_indices]  # Training images
    Y_train = Y_train[train_indices]  # Training labels

    return X_train, Y_train, X_val, Y_val  #the final groupings after
        splits
```

Next, I randomly initialize the parameters, the weights and biases. This creates a starting point for gradient descent. Additionally, I set up one of the hyperparameters, the layer sizes. Line 19 won't be consistently set to the value written, as I experiment with layer sizes in the next section.

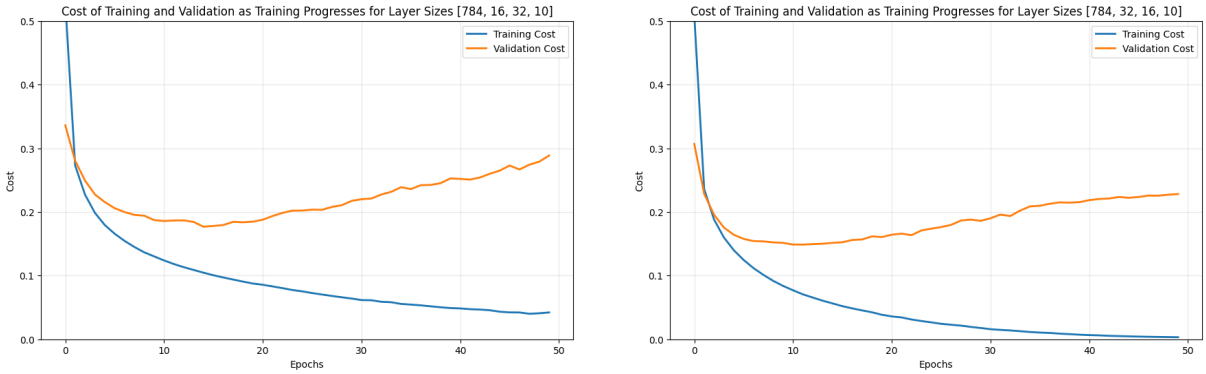
```
1  def initialize_parameters(layer_sizes):
2      parameters = {}
3      L = len(layer_sizes)  # number of layers (4) --> two hidden
4
5      for l in range(1, L):
6          fan_in = layer_sizes[l-1]
7          fan_out = layer_sizes[l]
8
9          #randomizes initial weights
10         parameters[f'W{l}'] = np.random.randn(fan_in, fan_out) * np.
               sqrt(2.0 / fan_in)
11
12         # Initialize biases to zeros
13         parameters[f'b{l}'] = np.zeros((1, fan_out))
14
15         print(f"Layer␣{l}:␣W{l}.shape␣=␣{parameters[f'W{l}'].shape},␣b
               {l}.shape␣=␣{parameters[f'b{l}'].shape}")
16
17     return parameters
18
19 layer_sizes = [784, 128, 64, 10] #We will change this in our
       experiments
20 parameters = initialize_parameters(layer_sizes)
```

3.2 Adjusting Layer Sizes

Choosing the best layer sizes is an important decision. The input layer will always be 784 and the output will always be 10. But, it is the user's choice to decide the number of inner layers and the sizes of inner layers. To keep it simple, in this section only layer sizes will be adjusted, rather than the number of layers (which will be kept as 2). To reiterate, the goal is to have the lowest possible cost. I performed experiments where I changed the

layer sizes to examine the impact on both training cost and validation cost. I kept the learning rate and batch size consistent across all of the experiments, with the intention of controlling the experiment. However, in reality, different layer sizes are more or less effective with different learning rates. For example, often greater layer sizes work better with smaller learning rates.

The ideal graph would show the training cost (blue line) decreasing toward 0, as the cost should be minimized through gradient descent. For the validation cost (orange line), it should also decrease, to indicate accuracy. Two factors that will come into play is the size of the inner layers, and the sizes of the inner layers relative to each other.



(a) Layer Sizes: [784, 16, 32, 10]

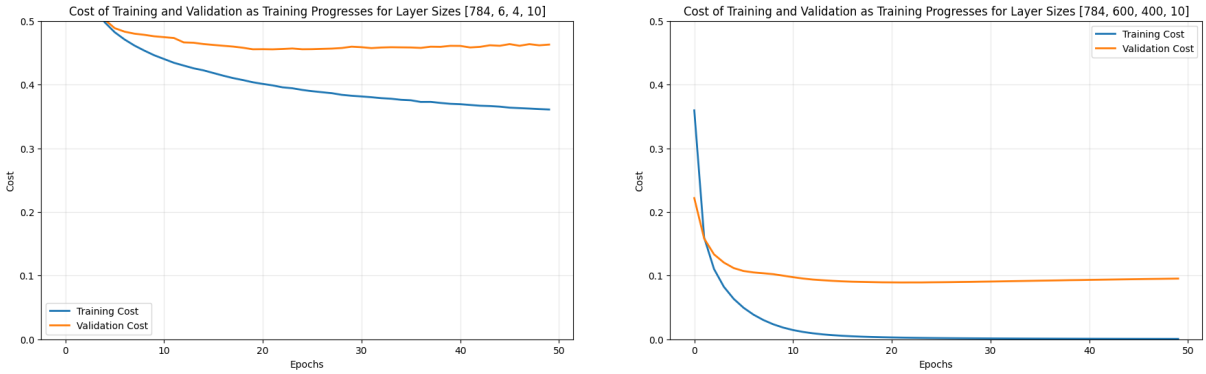
(b) Layer Sizes: [784, 32, 16, 10]

Figure 1: Performance comparison of [784, 16, 32, 10] and [784, 32, 16, 10]

These graphs compare having inner layers [16, 32] and [32, 16], so same numbers but different order. Graph (b) performed better than (a): the training cost reached zero and minimized faster. Additionally, the minimum of the validation cost and the final validation cost in graph (b) were lower, i.e., better. This should be expected. The first inner layer has direct connection to the input layer with weights, but the second inner layer does not (it has weights connected to the first inner layer). With two different sized layers, the first one should be the larger one so it can extract the useful information from the data—it matters more. In fact, for a simple dataset like MNIST, a decent NN can be created with just one inner layer.

Both of these graphs illustrate a common issue worth noting. While the training cost decreases, the validation cost decreases and then increases. This indicates overfitting. The model learned the training data set too well, meaning it is overly conditioned to those specific images. One way to combat this is to decrease the number of epochs, so the

endpoint is at the minimum of the validation cost line (around 10 or 15 here). Another solution is to adjust the hyperparameters, like the layer sizes and learning rate.



(a) Layer Sizes: [784, 6, 4, 10]

(b) Layer Sizes: [784, 600, 400, 10]

Figure 2: Performance comparison of [784, 6, 4, 10] and [784, 600, 400, 10]

Graph (a) illustrates an example of layer sizes being too small. Neither the training or validation costs get close to zero. Graph (b) is an example of a good NN. There is not a lot of overfitting and both lines decrease. Graph (b) may indicate that the overfitting issue is likely not about the layer sizes, but other factors. However, [600, 400] could be a bad layer size choice given a different learning rate. While it is useful to see the impact of hyperparameters through experiments, it is difficult because the hyperparameters are dependent on one another. With the NN code linked through GitHub in the extended listings section, I have played around with switching the hyperparameters, with the goal of finding the best combination.

3.3 Binary Experiments

In the this section, I am going to step away from the large MNIST dataset problem and focus on a more binary one. The question driving this section will be, “Is this a three or not?” Using linear regression, logistic regression, logistic regression with gradient descent, and a single-neuron NN, I created models that determine if a given image is a 3 or not. In the following images, the color scale ranges from dark blue (negative) to yellow (positive).

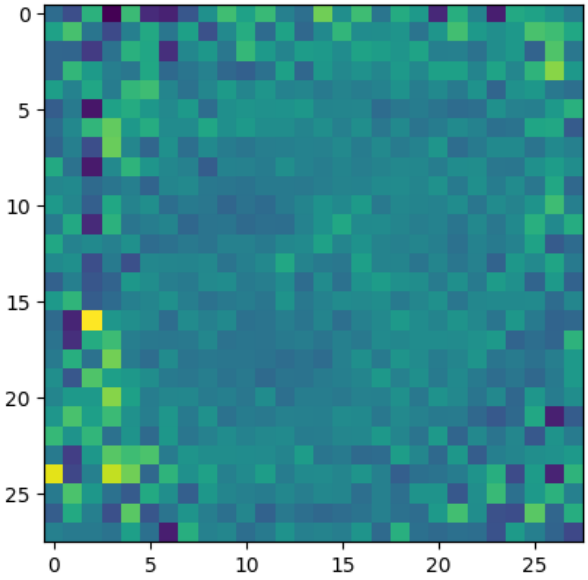


Figure 3: Visualization of Weights Found by Linear Regression

Figure 3 shows the visualization of weights found by linear regression, which is the most basic way to fit the data. Linear regression can be solved with the normal equation, which is $(X^TX)\beta = X^TY$; and β can be isolated so $\beta = (X^TX)^{-1}X^TY$. X is the input data, Y is the output data, T is transpose, and β is the slope, which is what is visualized in Figure 3. To use linear regression, I had to add noise, because in order to take the inverse of a matrix, it needs to be non-singular.

```
Y_all_threes  = (labels == 3)
Y_all_threes_encoded = Y_all_threes.astype(float) # acts like one
    hot encoding but focuses on the 3s

X_train_mod = X_train + np.random.normal(0,0.02, size= 42000*784).
    reshape(42000, 784) #adds noise to make nonsingular

beta =  np.linalg.inv(X_train_mod.T @ X_train_mod) @ X_train_mod.T
     @ Y_all_threes_encoded #the normal equation

plt.imshow(beta.reshape(28,28))  #makes the graph
```

Linear regression is a simple fit, so it doesn't have the best accuracy. The visualization of beta hints at this, as it appears scattered.

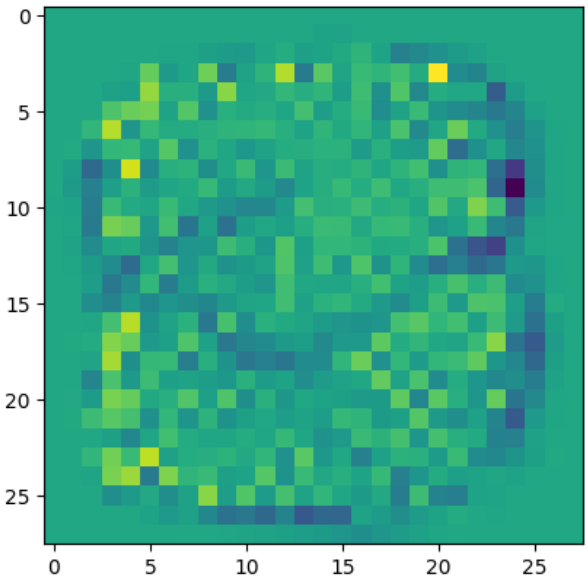


Figure 4: Visualization of Weights Found by Logistic Regression with L-BFGS solver (scikit-learn)

I used the scikit-learn (sklearn) library to implement logistic regression with the L-BFGS solver. The L-BFGS solver is sklearn's default way to optimize, rather than gradient descent. While gradient descent uses the first derivative and small steps, L-BFGS estimates the second derivative as well and takes larger steps. This is the default for sklearn because it typically converges faster.

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(penalty=None, C=1.0, solver='lbfgs',
    random_state=42)
model.fit(X_train, Y_all_threes_encoded)

sk_learn_beta = model.coef_.reshape(28,28) #identifies the var
    beta

plt.imshow(beta.reshape(28,28)) #makes the graph
```

Figure 4 illustrates the weights with a more clear image of a general 3 shape compared to what linear regression produced. Linear regression and logistic regression *can* solve problems—the task doesn't require a complicated NN.

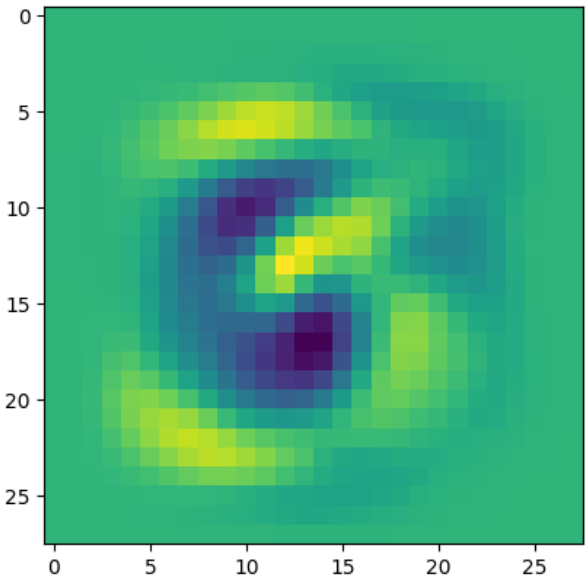


Figure 5: Visualization of Weights Found by Logistic Regression with Gradient Descent

Without a library to simplify the code, it takes several lines to run a logistic regression model with gradient descent. But, it is still manageable. I included the code in the extended listings section for readability and formatting preferances.

I used sigmoid as the activation function. It is important to note that I initialized all of the weights and biases to zero. This starts with the weights holding the value of the background color, complete black. Then, the model will adjust from that point, rather than a random one. This shortens the time fitting takes, as the background is already set. However, initializing to zero is not good practice and can lead to a symmetry breaking problem with standard multi-layer networks.

Figure 5 is clearly a 3. The shape is defined by critical negative and positive weights. The negative weights (dark blue) penalize for having color in those areas, to clearly define the shape of a 3. For example, imagine a 5 overlaid on Figure 5. The bottom curve and top line match the general outline of a 3. For the model to distinguish a 3 from a 5, it assigns negative values to the areas where a 3 wouldn't be in. The pixels making up the vertical line in a 5 would be multiplied by the negative weights in that area, to help clarify that the image is not a 3.

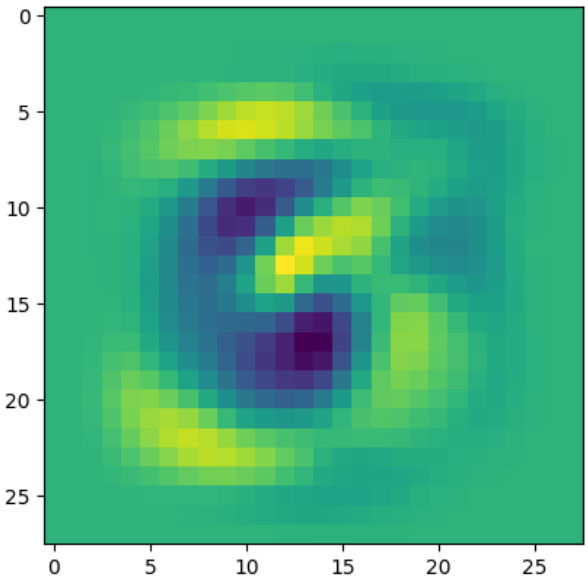


Figure 6: Visualization of Weights Found by Single Neuron NN (PyTorch)

Lastly, I used a single-neuron NN, meaning the network just has layers [784, 1]. The value at the last neuron indicates whether it is a three or not based on probability. A threshold needs to be chosen. For example, with threshold = 0.5, if the output is greater than 0.5, the model predicts a 3. I used the library PyTorch for this experiments. Due to length, I put the code under the extended listings section.

Figure 5 and Figure 6 are identical because the models are mathematically equivalent. This demonstrates how logistic regression is embedded in the neural network framework. A single-neuron NN is the same as logistic regression with gradient descent. Also, the fact of initializing the weights and biases to zero in both experiments ensures they are actually identical.

The purpose of these binary experiments is to see how image classification can be done with mathematically basic models. The roots of machine learning models and NNs are in linear and logistic regression. It is useful to understand this before jumping into complex problems and large models, where the math can get lost.

3.4 Extended Listings

For document readability, I am placing the extended listings below. This covers the logistic regression with gradient descent experiment that correlates to Figure 5 and the single-neuron NN experiment that correlates to Figure 6. Additionally, I am including

the link to my GitHub repository for the NN code that guided this project, and was used in the layer sizes experiments. Link: https://github.com/claregartz/Independent-Study-Image-Classification

Listing 1: Logistic Regression with Gradient Descent Experiment

```
    def sigmoid(z):
        return 1 / (1 + np.exp(-z))

    def compute_loss(y_true, y_pred):
        # Clip predictions to avoid log(0)
        y_pred = np.clip(y_pred, 1e-9, 1 - 1e-9)
        return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.
            log(1 - y_pred))

    def gradient_descent(X, y, learning_rate, n_iterations):
        m, n = X.shape
        y = y.reshape(len(y),1)
        # Initialize weights and bias to zeros (based on our problem
            this helps accuracy)
        w = np.zeros((n, 1))
        b = 0
        loss_history = []

        for i in range(n_iterations):
            if i % 10 == 0 :
                print("on interation ", i)
            z = np.dot(X, w) + b
            y_pred = sigmoid(z) #activation

            # Calculate gradients
            dw = (1 / m) * np.dot(X.T, (y_pred - y)) # really \partial
                CE / \partial w
            db = (1 / m) * np.sum(y_pred - y)
            # Update parameters
            w -= learning_rate * dw
            b -= learning_rate * db

            # Record loss
            loss = compute_loss(y, y_pred)
```

```
            loss_history.append(loss)

        return w, b, loss_history

    weights, bias, losses  = gradient_descent(X_train,
        Y_all_threes_encoded, 0.01, 1000) #.01 is learning rate, 1000
        is iterations

    plt.imshow(weights.reshape(28,28))
```

Listing 2: PyTorch Single-Neuron NN Experiment

```
    import torch
    import torch.nn as nn
    import torch.optim as optim

    X_tor = torch.tensor(X_train, dtype=torch.float32)
    y_tor = torch.tensor(Y_all_threes_encoded.reshape(-1, 1), dtype=
        torch.float32)

    class LogisticRegressionSigmoidTorch(nn.Module):
        def __init__(self, input_dim):
            super(LogisticRegressionSigmoidTorch, self).__init__()
            self.linear = nn.Linear(input_dim, 1)
            self.sigmoid = nn.Sigmoid() # Explicit sigmoid layer
            nn.init.zeros_(self.linear.weight) #starts with zeros


        def forward(self, x):
            z = self.linear(x)
            y_pred = self.sigmoid(z) # Apply sigmoid in forward pass
            return y_pred

    model_sig = LogisticRegressionSigmoidTorch(input_dim=28*28)

    # 2. Define the simple BCELoss and Optimizer
    # Note: This requires the model output to be probabilities (0 to
        1)
    criterion_bce = nn.BCELoss()
    optimizer_sig = optim.SGD(model_sig.parameters(), lr=0.01)
```

```
27
28      # 3. Training Loop with Automatic Differentiation
29      n_iterations = 1000
30      loss_history_sig = []
31
32      for iteration in range(n_iterations):
33          # Forward pass: compute predicted y (probabilities)
34          y_pred_prob = model_sig(X_tor)
35
36          # Compute loss
37          loss = criterion_bce(y_pred_prob, y_tor)
38          loss_history_sig.append(loss.item())
39
40          # Zero gradients, backward pass, and update weights
41          optimizer_sig.zero_grad()
42          loss.backward()
43          optimizer_sig.step()
44
45      plt.imshow(nn_beta.detach().numpy().reshape(28,28)) #makes the
            graph
```

4 Reflection

In the beginning of this project, I struggled with where to begin. Playing around with pre-written codes sparked the questions that drove this project. I learned a lot about NNs, even though I only scratched the surface. I want to thank Mr. Villegas for advising me and my Dad for helping me with the experiments.

I'm looking forward to improving my Python skills and deepening my understanding of NNs—and machine learning as a whole.

References

3Blue1Brown. Backpropagation, intuitively | deep learning chapter 3, 2025a. URL https://youtu.be/Ilg3gGewQ5U?si=tLJMM8AkmuLoYVEw.

3Blue1Brown. Backpropagation calculus | deep learning chapter 4, 2025b. URL https://youtu.be/tIeHLnjs5U8?si=VbrPz59z9D27SrBw.

3Blue1Brown. But what is a neural network? | deep learning chapter 1, 2025c. URL https://youtu.be/aircAruvnKk?si=9ekGB59EFnO9Digj.

3Blue1Brown. Gradient descent, how neural networks learn | deep learning chapter 2, 2025d. URL https://youtu.be/IHZwWFHWa-w?si=tEK9_GPRR8SvjoFN.

Dave Bergmann and Cole Stryker. What is backpropagation? | ibm, 07 2024. URL https://www.ibm.com/think/topics/backpropagation.

Caroline Clabaugh, Dave Myszewski, and Jimmy Pang. Neural networks, 2000. URL https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Neuron/.

GeeksforGeeks. Activation functions in neural networks, 01 2018. URL https://www.geeksforgeeks.org/machine-learning/activation-functions-neural-networks/.

Michael A Nielsen. Neural networks and deep learning, 2018. URL http://neuralnetworksanddeeplearning.com/index.html.

Shivansh Srivastava. Understanding the difference between relu and sigmoid activation functions in deep learning, 04 2024. URL https://medium.com/@srivastavashivansh8922/understanding-the-difference-between-relu-and-sigmoid-activation-functions-in-deep-

Suryansh. Gradient descent: All you need to know, 10 2020. URL https://medium.com/hackernoon/gradient-descent-aynk-7cbe95a778da.

Monika Szumilo. 3d gradient descent, 2025. URL https://aero-learn.imperial.ac.uk/vis/Machine%20Learning/gradient_descent_3d.html.

Turing. Importance of neural network bias and how to add it. URL https://www.turing.com/kb/necessity-of-bias-in-neural-networks.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv:1611.03530 [cs]*, 02 2017. URL https://arxiv.org/abs/1611.03530.

Samson Zhang. Building a neural network from scratch (no tensorflow/pytorch, just numpy and math), 2025. URL https://youtu.be/w8yWXqWQYmU?si=vckHRdyWx2ESDtbW.