# Gradient Descent and a Feed Forward Neural Network

Evan Matel, Martin Jakhelln

November 2023

**Abstract**

In this report, we will cover two different approaches to machine learning, namely gradient descent(GD) and feed-forward neural networks(FFNN). We will apply these two methods to the Wisconsin breast cancer data set and compare how these perform. By doing this we hope to look at differences in performance, efficiency, and underlying mechanics. The gradient descent algorithm, a classic optimization technique, will serve as the benchmark when we compare the efficiency of the FFNN.

## 1 Introduction

Artificial intelligence has exploded in the public eye over the last couple of years. This is mostly due to large deep-learning algorithms like AlphaGo and ChatGPT, both of which are highly complicated neural networks. This report focuses on applying machine learning to classify the Wisconsin Breast Cancer dataset. We aim to evaluate the performance of our two approaches to machine learning in context of the cancer classification. First, gradient descent (GD), a widely used machine learning algorithm, will serve as our baseline in both performance and efficiency. The simplicity of the model makes it ideal when we compare it to the complex feed-forward neural network (FFNN). We will also aim to implement multi-classification into our neural network and train it on the famous MNIST data set. By comparing these two models, we hope to uncover the respective drawbacks and advantages of both.

## 2 Method

### 2.1 Gradient Descent (GD)

Gradient descent is a method where you start at a random position in the space of your cost function; the number of dimensions depends on how many features your data has. You then calculate the gradient, which tells you how the cost function changes for each feature ($\theta$). One then moves in the direction that decreases your cost the most, and how much you move is decided by your learning rate eta $\eta$ which you multiply your gradient by and update $\theta$ accordingly. You then continue to do this process until your algorithm reaches a local minimum, which one hopes is the global minimum (see Figure 1). To improve the performance of our algorithm, we are introducing the regularization parameter lambda $\lambda$ from ridge regression. This parameter scales the gradient with the previous weights in hopes of improving our GD.

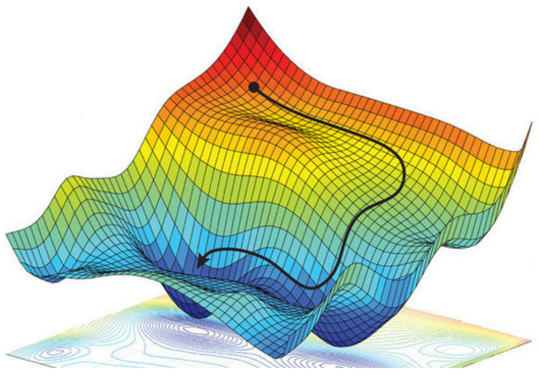


Figure 1: Visualization of a 3D cost function and how gradient descent minimizes said function.[2]

### 2.1.1 Stochastic Gradient Descent (SGD)

Stochastic gradient descent differs from gradient descent by instead of considering the entire dataset for each interaction it chooses small batches of data each iteration. We are doing this by randomly choosing the batches with replacements, hence the name. By doing this our algorithm is more efficient and also better at avoiding local minima because of the randomness introduced. As mentioned previously the cost function is an important step of both GD and SGD, and we are going to use the ordinary least squares and cross-entropy cost function.

$$costOLS = \sum_{i=1}^{m} (\theta(x^{(i)}) - y^{(i)})^2 \tag{1}$$

$$CostCross = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log(\theta x^{(i)} + \epsilon) + (1 - y^{(i)}) \log(1 - \theta x^{(i)} + \epsilon) \tag{2}$$

### 2.1.2 Momentum, RMSprop, and ADAM

Beyond just using GD and SGD we are going to implement algorithms that hopefully improve the results of our classification. First, we implement momentum, introducing inertia, which mimics the behavior of a moving object by accumulating velocity in the direction of the gradient. By implementing this we hope that the algorithm improves its ability to traverse flat surfaces faster and overcome oscillating surfaces. This is done by integrating past gradients into the current gradients, and our theta would then be updated like this:

$$change = \eta * gradient * \Delta_momentum * change$$

$$theta = theta - change$$

Where $\Delta_{momentum}$ is a pre-determined parameter and normally holds a value between 0 and 1.

RMSprop is another algorithm used to improve our gradient descent, what this introduces is a giter parameter that specifically aims to improve the learning rate for each update. By doing this RMSprop aims to mitigate the influence of outliers and should make for a stable convergence. With this, our update for each iteration would look like this:

$$Giter = \rho * Giter + (1 - \rho) * gradient * gradient$$

$$update = \frac{gradient * \eta}{delta} + \sqrt{Giter}$$

$$theta = theta - change$$

Giter is reset to 0 for every epoch and $\rho$ is a parameter that's pre-determined.

Finally, ADAM puts both of the above-mentioned algorithms together, introducing moment and hyperparameters. For each iteration, ADAM computes gradients, updates moment, and applies biases. This makes ADAM an efficient and robust algorithm that we found the most success with ourselves. A update would look like this:

$$firs\,moment = beta1 * first\,moment + (1 - beta1) * gradient$$

$$second\,moment = beta2 * second\,moment + (1 - beta2) * gradient * gradient$$

$$first\,term = \frac{first\,moment}{1 - beta * *iter}$$

$$second\,term = \frac{second\,moment}{1 - beta2 * *iter}$$

$$update = eta * \frac{first\,term}{\sqrt{second\,term + delta}}$$

$$theta = theta - update$$

## 2.2 Feed Forward Neural Network (FFNN)

The neural network is the most common method to create an AI program today. The idea behind a neural network (NN) comes from simulating how a brain works, and how neurons connect to each other. A NN is composed of many weights and biases that are finely tuned over a complicated training process. Another main component is the hidden layer, which is quite a cryptic name for something so simple. A hidden layer consists of a number of neurons that are there to transfer the "information" from the inputs, through the weights and biases, and to the output. We will explain further. As a simple example, we will imagine a neural network with only 1 hidden layer between the inputs and output.

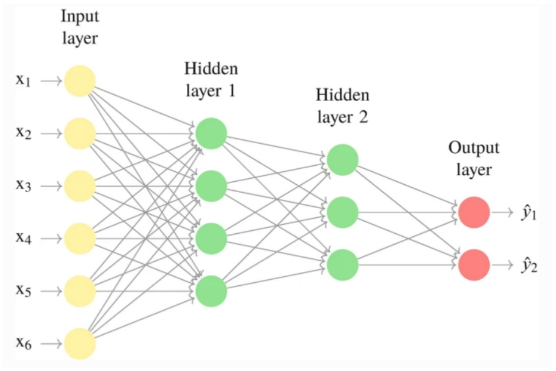


Figure 2: A visual aid for understanding the composition of a feed-forward neural network.[3]

One initializes the network with normally distributed weights and biases which are then adjusted and stabilized over time as the program goes through training. The first part of the training loop is when we move forward through the structure of the network. For each neuron in a hidden layer, one multiplies each input by its corresponding weight, sums them up, and then the bias is added to this sum, this value is commonly called $z$. One then uses an activation function, like the sigmoid function for example. This value is then called the activation layer, or simply $a$, and is then the input for the next hidden layer, or in our example with only one hidden layer, the output (see Equation 3).

$$z = input \cdot W + B, \quad a = act_func(z) \tag{3}$$

Going back to activation functions, there are several commonly used ones, but we only chose to use three. First is the sigmoid function, which is very commonly used for binary classification (see Equation 4). What the sigmoid function does is it "squeezes" all numbers into a range from 0 to 1. One of the most commonly used activation functions today is the Rectified Linear Unit (RELU) function, what RELU does is if the input is positive, it doesn't change it at all, but if it is negative, RELU returns 0. Another activation function, commonly used at the output for multi-classification problems is the softmax function, which takes an array of output values and converts them into probabilities, the program then chooses the value with the greatest probability (see Equation 5).

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{4}$$

$$softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}, \quad y_i^{pred} = max(softmax(\mathbf{z})_i) \tag{5}$$

Backpropagation is a commonly used method in neural networks. One starts at the output and calculates the difference between what the algorithm predicts, *y_pred*, and the true labels, *y_test*. It then alters the weights and biases connecting the hidden layer to the output accordingly so that the activation layer gives the correct output, this is done using gradient descent, as explained earlier. This process then continues backwards through the program. Traditional backpropagation is done by taking the entire dataset into account, this means taking all of those small changes in the weights into account for every input and then taking the mean. This can be very computationally demanding, and therefore it's very common to use batch training, as explained earlier with stochastic gradient descent.

The model is then trained forward and backwards for $N$ iterations or "epochs" and for $M$ minibatches. The number $M$ is calculated by dividing how many inputs your data has by how large you want your minibatches to be. When the program is done training, one then uses the *predict* method on the test data which checks how well the program performs on data it has never seen before, like the test set. One checks this by using a toolbox of performance metrics like accuracy, MSE, R2 score, and many others.

## 3 Results

### 3.1 GD and SGD

Figure 3 shows a confusion matrix which is a valuable insight into what is actually happening when our model classifies the dataset. On the diagonal, we find the true negatives and the true positives, and those marked with negative positive or positive negative are cases where our model predicted positive when it was negative or vice versa. out of 114 cases, we got 95 of the cases correct which equals a 83.3% accuracy, which is decent.

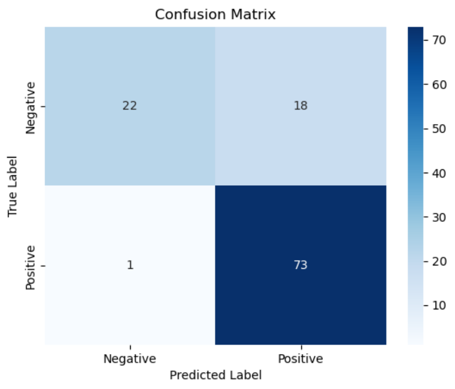


Figure 3: This is a confusion matrix showing what our gradient decent algorithm predicted on The Wisconsin Breats Cancer Dataset

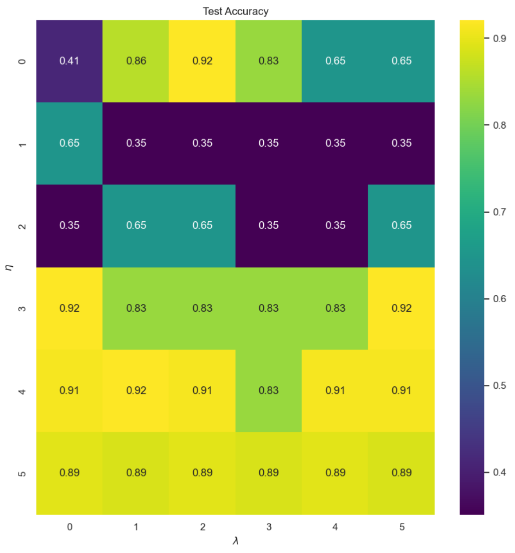


Figure 4: This figure shows how we tested different parameters and how well they performed with our gradient descent algorithm

Figure 4 shows a heatmap of our accuracy while testing the regularization parameters $\lambda$ and $\eta$. This illustrates our model's sensitivity when changing these parameters, it is also ideal when trying to find the best parameters for our model and makes it a lot quicker for us when tuning our hyperparameters.

We tested $\lambda$ between $10^{-1}$ and $10^{-6}$ and tested $\eta$ between $10^{-1}$ and $10^{-4}$ and we found that for gradient decent a eta $1.5 \times 10^{-3}$ and lambda at $10^{-2}$ which gave us the best result. While our heat map shows a possible accuracy of 92% we only achieved an accuracy of $83,3\%$ which we will discuss later.

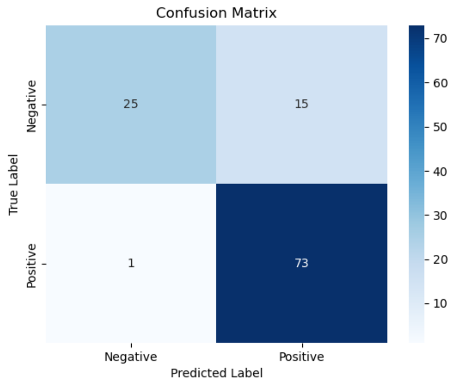


Figure 5: This is a confusion matrix showing what our stochastic gradient descent algorithm predicted on The Wisconsin Breast Cancer dataset

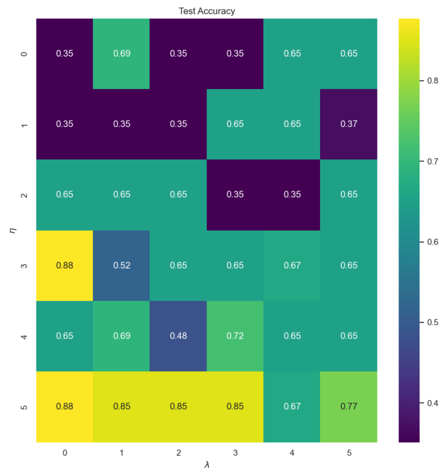
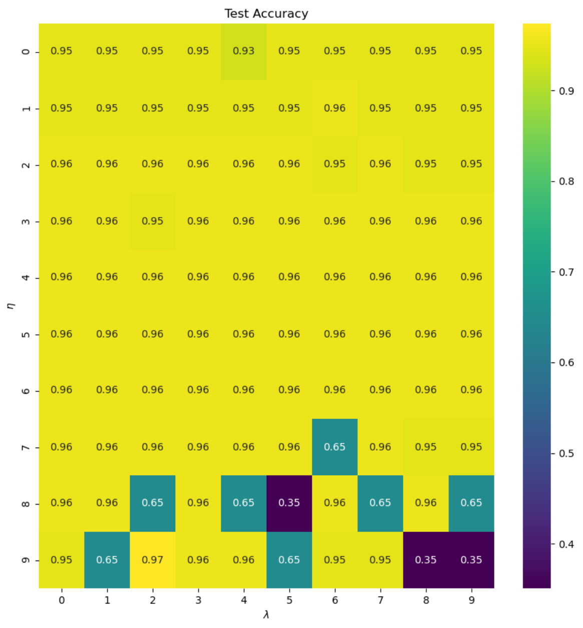


Figure 6: This figure shows how we tested different parameters and how well they performed with our stochastic gradient descent algorithm

Figure 5 shows the confusion matrix for our stochastic gradient descent and here we got 98 out of 114 possible classifications correct which gave us an accuracy of 85.9%. Examining Figure 6, it becomes clear that we generally have fewer parameters that give us a good result, which we will discuss later. We tested for the same range of etas and lambdas and found that the optimal lambda was $10^{-3}$ and the optimal eta was $10^{-4}$, which is a relatively small learning rate.

## 3.2 FFNN

Using our hyperparameter testing function, we found that $\eta = 1.0$ and $\lambda = 4.641 \times 10^{-4}$ gave an accuracy of 97.4% for the binary Wisconsin breast cancer data set. (see Figure 7). When training our model for multi-classification, using MNIST data set, we found that $\eta, \lambda = 2.154 \times 10^{-2}$ gave an accuracy of 98.1% (see Figure 8).



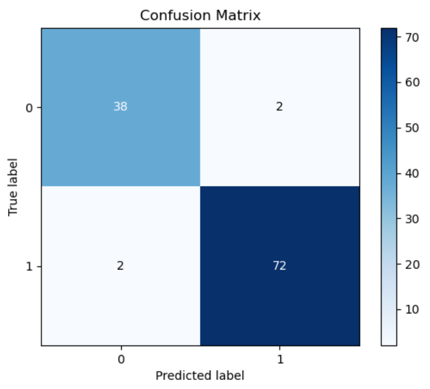
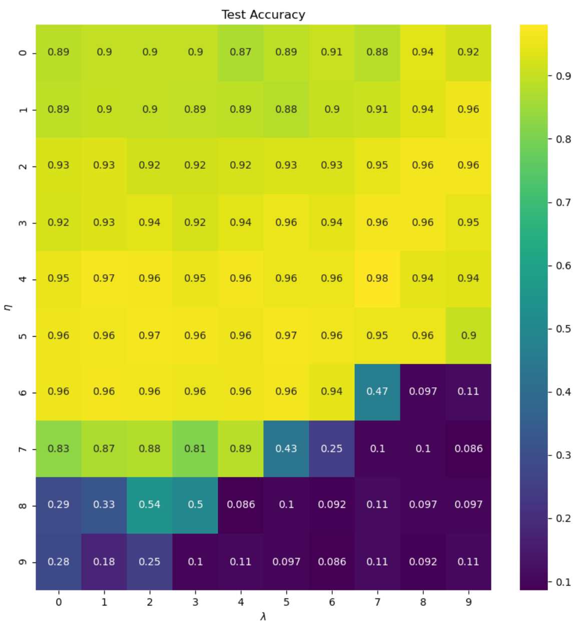


Figure 7: Heat map of accuracies with different etas and lambda, and the confusion matrix from our data being ran on the best of those. This was from our model being trained on the Wisconsin breast cancer data, and with a hidden layer structure of [10, 3].



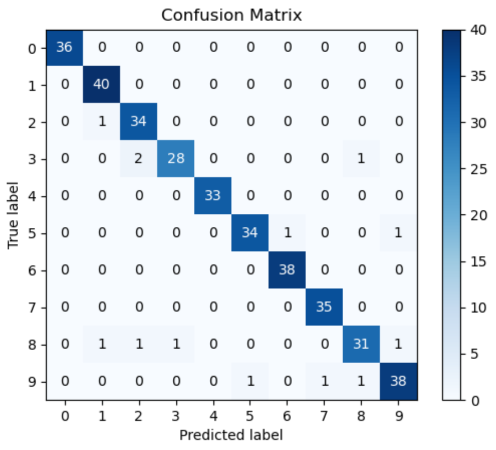


Figure 8: Heat map of accuracies with different etas and lambda, and the confusion matrix from our data being ran on the best of those. This was from our model being trained on the MNIST dataset, and with a hidden layer structure of [300, 300].

## 4 Discussion

We had an interesting bug in our code that we couldn't quite figure out. When testing for hyperparameters, we ran the program many times and trained many different instances to find which hyperparameters give the best accuracy. The function then spits out the hyperparameters that give the best accuracy. However, when we then used those hyperparameters on a new instance afterward, we never got the same accuracy, which we found very puzzling. We knew there had to be some random element involved, but we used the same seed for the whole program.

After performing both GD and SGD models on the dataset we can clearly see the differences between them, where GD is a more stable and slow method and SGD is a faster but unstable method as seen in figures 4 and 6. This trade-off between speed and ability shows us the choice one must make when choosing an algorithm.

We also ran into a lot of problems when using anything but the ADAM method. Normal GD, SGD, and RMSprop were highly unstable and we either had to use a lower learning rate and get bad results or a normal learning rate and run into error codes. while the exact reason for this instability remains uncertain, it is most a problem with the gradient and theta, since having a low learning made the program slightly more reliable.

Exploring various forms of GD alongside FFNN we have gained insight into how these models differ from each other. GD, including its several variants, is relatively always faster and more efficient, and gives us a result that we can use for comparing with our FFNN, which while a rigorous and time-consuming method, always outperformsed the GD methods in our case, showing how it seems to be a superior method, which 99% of people in the field would agree with.

## 5 Conclusion

Using the WBC data and starting with normal gradient descent, we reached an accuracy of 83.3%, then onto stochastic gradient descent where our model was 85.9% accurate, and finally, our neural network, where we reached a satisfactory accuracy of 97.4%. We also experimented with multi-classification using the MNIST data set, where we achieved an even greater accuracy at 98.1%. During our work, we have trained several models with increasing complexity and said models' accuracy has increased proportionally as we had expected. We had several complications along the way but ended up with what we deemed as acceptable results.

## Resources

[1] Lecture notes for FYS-STK3155 by Morten Hjorth-Jensen., https://compphysics.github.io/MachineLearni

[2] Non-Convex Optimization. We Utilize Stochastic Gradient Descent to Find ..., www.researchgate.net/figure/Non-convex-optimization-We-utilize-stochastic-gradient-descent-to-find-a-local-optimum$_fig1_325142728.Accessed19Nov.2023$.

[3] Dixon, Matthew F., et al. "Feedforward Neural Networks." SpringerLink, Springer International Publishing, 1 Jan. 1970, link.springer.com/chapter/10.1007/978-3-030-41068-1$_4$.