Introduction

This project entailed implementing multinomial logistic regression for a multi-classification task. Here, we aim to classify digits in the MNIST dataset. Our labels, $\vec{y}$, are one hot encoded, giving that $\vec{y} \in \mathbb{R}^{10}$, each element is $y_j \in \{0, 1\}$ and there is exactly one $j \in \{1, ..., 10\}$ such that $y_j = 1$. Each example $\vec{x_i}$ being trained on is a 28×28 images with a digit whose center of mass is centered on the image. The image is flattened for processing, giving $\vec{x_i} \in \mathbb{R}^{784}$. For a parameter matrix $W \in \mathbb{R}^{10 \times 784}$, our hypothesis is

$$h_W(\vec{x}) = \text{softmax}(W\vec{x})$$

where

$$\text{softmax}(\vec{u}) : \mathbb{R}^{10} \to \mathbb{R}^{10}$$

is defined by

$$(\text{softmax}(\vec{u}))_j = \frac{e^{u_j}}{\sum_{k=1}^{10} e^{u_k}}.$$

The regularized empirical risk for our dataset $\mathcal{D}$, using regularization parameter $\gamma > 0$ is

$$R(h_W) = \frac{1}{|\mathcal{D}|} \sum_{(x,y) \in \mathcal{D}} L(h_W(x), y) + \frac{\gamma}{2} \|W\|_F^2$$

where

$$L(\hat{y}, \vec{y}) = -\sum_{j=1}^{10} y_j \cdot \log(\hat{y}_j)$$

is the cross entropy loss function.

Part 1: The Gradient of the Loss Function

We begin the implementation by deriving an expression for the gradient of the total loss with regularization over a whole dataset with respect to the weights W. In computing the gradient of this expression, we first note that the labels, $\vec{y}$, are one hot encoded. This allows us to get rid of the summation since only one term will be nonzero. That is, for some $i \in \{1, ..., 10\}$, $\vec{y} = e_i$ where e_i is the unit basis vector in $\mathbb{R}^{10}$. Concretely, this gives

$$L(h_W(x), y) = -\log(\hat{y}_i) = -\log\left(\frac{\exp(e_i^T W x)}{\sum_{k=1}^{c} \exp(e_k^T W x)}\right) = \log\left(\sum_{k=1}^{c} \exp(e_k^T W x)\right) - e_i^T W x$$

We now take the gradient with respect to the weights matrix, W, for a single example and obtain

$$\nabla_W L(h_W(\vec{x}), \vec{y}) = \frac{\sum_{k=1}^{n} \exp(e_k^T W \vec{x}) e_k \vec{x}^T}{\sum_{k=1}^{c} \exp(e_k^T W \vec{x})} - e_i \vec{x}^T = \text{softmax}(W\vec{x})\vec{x}^T - e_i \vec{x}^T = \text{softmax}(W\vec{x} - \vec{y})\vec{x}^T.$$

as the gradient of our loss function without regularization. Computing the gradient of the regularizer with respect to the weight matrix, W, gives

$$\nabla_W \left(\frac{\gamma}{2} \|W\|_F^2 \right) = \gamma W.$$

Summing over all examples, and combining the above two results yields

$$\nabla_W R(h_W) = \frac{1}{|\mathcal{D}|} \sum_{(\vec{x},\vec{y}) \in \mathcal{D}} \text{softmax}(W\vec{x} - \vec{y})\vec{x}^T + \gamma W.$$

With respect to the above, a single update of gradient descent has the update rule

$$W_{t+1} \longleftarrow W_t - \alpha_t \nabla_W R(h_W).$$

Part 2: Gradient Descent

Approximating the Accuracy of the Gradient

Having computed the gradient, we now develop a means for testing its accuracy. Knowing that for a function $f : \mathbb{R}^d \to \mathbb{R}$ and vectors $\vec{x}, \vec{v}$ that

$$\vec{v}^T \nabla f(\vec{x}) \approx \lim_{\eta \to 0} \frac{f(\vec{x} + \eta\vec{v}) - f(x)}{\eta}$$

is an approximation to the gradient, we make a slight modification for the problem at hand.

To test our gradient of the function $f(x, y, W)$ (total risk and regularization), we use the approximation:

$$\lim_{\eta \longrightarrow 0} Tr(V^T \nabla f(x, y, W)) \approx \frac{f(x, y, W + \eta V) - f(x, y, W)}{\eta}$$

The left side of the equation is the inner product of matrices V and $\nabla f(x, y, W)$. The right side is the difference quotient, and as we take the limit as $\eta \to \infty$, we approximate the gradient of ∇f.

In code, we set $\eta = 1\text{e}{-7}$ and decrease it by an order of magnitude until we reach $1\text{e}{-12}$ which is roughly machine precision. We check to see if the right hand side is roughly equal to the left side within some small, ϵ, tolerance. In our tests, we randomly generated weights matrix W, and for each instance, we tested the gradient on each example $\vec{x}_i \in X$ and $\vec{y}_i \in Y$. We tested for about twenty values of W and all $60,000$ samples in the data set. We observed that for $\eta = 1\text{e}{-7}$, the greatest difference from the left side and right side of the above equation was approximately $\epsilon \leq 1\text{e}{-8}$. For $\eta = 1\text{e}{-12}$, we observed that $\epsilon \leq 1\text{e}{-5}$. As we increased the value of η, we saw that numerical precision played a role in the results, and our error increased. Overall, we posit that our implementation of gradient is correct as we observed relatively low error in the approximation.

Timing the Algorithm

After ensuring our gradient function computes the gradient correctly, we now run the gradient descent algorithm with $L2$ regularization parameter $\gamma = 0.0001$ and step size $\alpha = 1.0$ for 10 iterations. The time it took to run these 10 iterations was approximately 68.9 seconds. We thus expect 1000 iterations to take $\approx 69 \cdot 100 = 6900$ seconds, or approximately 2 hours.

Part 3: Computing More Efficiently

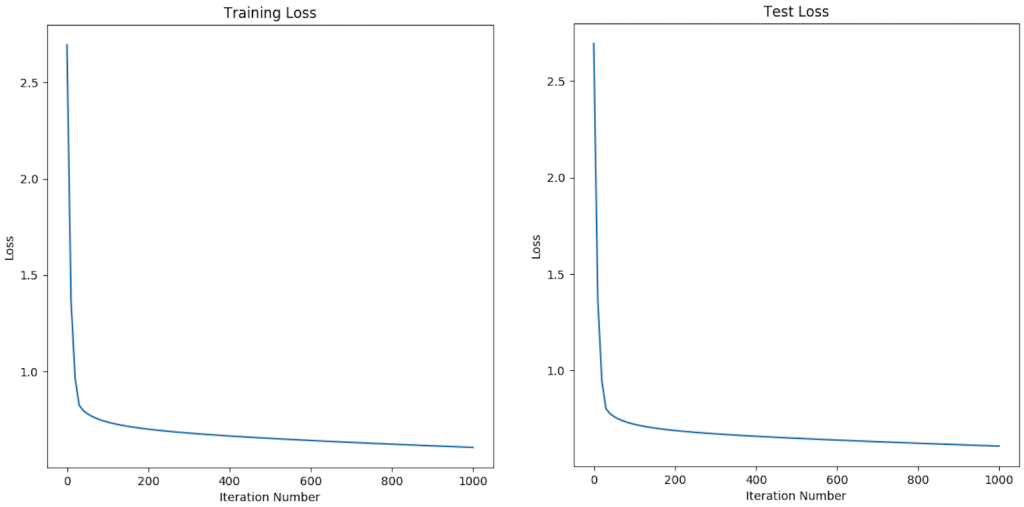
The functions `multinomial_logreg_total_loss` and `multinomial_logreg_total_grad`, which compute the total loss and gradient as specified above respectively, were now changed to use NumPy matrix arithmetic as opposed to Python for-loops. After the changes were implemented, we re-ran the gradient descent algorithm with $L2$ regularization parameter $\gamma = 0.0001$ and step size $\alpha = 1.0$ for 10 iterations as above. However, the time it took this time was about 7.37 seconds. This is a roughly $10\times$ improvement over the prior computation and the result can be attributed to NumPy using parallelism.
Python itself is dynamically typed and thus each time a computation is done the data type has to be rechecked. This causes a lot of overhead and prevents Python from using its C under-typing to do efficient computations that a low level language like C can. NumPy on the other hand bypasses this with it's type implementation of NumPy arrays allowing for quick computations, as is capable in other low level languages.[1]

Part 4: Evaluating Gradient Descent

We now implement a function `multinomial_logreg_error` to compute the error of the classifier given parameters W. We rerun the gradient descent algorithm with $L2$ regularization parameter $\gamma = 0.0001$ and step size $\alpha = 1.0$ for 1000 iterations, and record the value of the parameters every 10 iterations.

Evaluating Error and Loss

Plotting the Training Error, Testing Error, Training Loss and Testing Loss at each recorded iteration, we produce the figures seen in Figures 1 and 2.

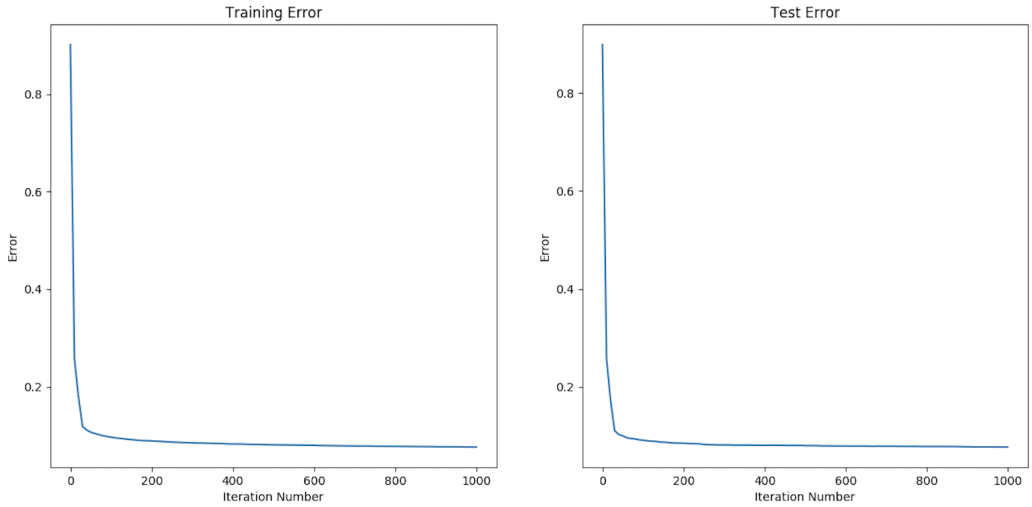


(a) Training Loss

(b) Testing Loss

Figure 1: Losses

[1] https://www.datadiscuss.com/proof-that-numpy-is-much-faster-than-normal-python-array/



(a) Training Error (b) Testing Error

Figure 2: Errors

Estimating Error

Now we analyze the performance of our classifier on the training set and testing set by looking at error rates. The data for Training Data is as follows:

	Training Data	
Data	Error Rate	Time to Calculate
Overall	0.077	0.15
100 samples	0.07	0.0014
1000 samples	0.08	0.011

The data for Test Data:

	Testing Data	
Data	Error Rate	Time to Calculate
Overall	0.0776	0.0255
100 samples	0.09	0.00085
1000 samples	0.08	0.0074

First, examining the training data error, we see that the overall error is around 7.6%. The 100 samples and 1000 samples had similar errors. For the training data, it seems that the 100 samples is sufficient to get the same error rate.

For the testing error, we observed similar results. All samples were similar to the overall error rate which is approximately 8%. The 100 samples was a little bit higher, but within 12% of the actual error rate. Here, it seems that 1000 samples will suffice.

We can make this more concrete with Hoeffding's inequality. For Z_i iid which take on

value $L_{0,1}(\hat{y}_i, y_i)$ with probability $\frac{1}{|\mathcal{D}|}$, we define $S_K := \frac{1}{K}\sum_{i=1}^{K} Z_i$. Then if $z_{min} \leq Z_k \leq z_{max}$,

$$\mathbb{P}\left(|S_k - \mathbb{E}[S_k]| \geq \alpha\right) \leq 2\exp\left(\frac{-2K\alpha^2}{(z_{max} - z_{min})^2}\right)$$

We note that classification error is a $0-1$ loss and thus $z_{min} = 0$ and $z_{max} = 1$. For an error rate of 7.6%, with probability 99% we therefore need

$$\begin{aligned} K &\geq \frac{1}{2\alpha^2}\log\left(\frac{2}{\epsilon}\right) \\ &\geq \frac{1}{2(0.076^2)}\log\left(\frac{2}{0.01}\right) \\ &\geq\approx 199. \end{aligned}$$

This number makes sense with our observed data. Now in observing the time it takes to complete this, taking smaller samples, as expected, took linearly proportional less time than larger samples. This is because we have to iterate through each sample, and apply our weight vector to it. So naturally, a larger sample set will take longer. We do not see any results outside of what we expected in execution time.

Conclusion

This project mainly explored optimizations in the performance of gradient descent for the MNIST data set. We started with a naive implementation of gradient descent where we used a for loop to iterate over training data when calculating gradients. We observed that this took a long time to compute, so we used NumPy operations to vectorize the code, in an attempt to speed it up - which is reflected in our results. We then explored how our trained model performs on never seen before testing data. We saw admirable results here with a X percent classification accuracy. Finally, we analyzed sampling methods and how various sample sizes impacted our overall error. The majority of the effort in this assignment was in testing and verifying the correctness of our gradient calculation and outputs. We also spent a lot of time looking at various NumPy operations to vectorize our code. While there are many possible ways of speeding up the algorithms, we believe that using NumPy operations performed well in this context. Additionally, we observed that we do not need that many samples to accurately estimate the error on training set data. Overall, we found this assignment thought provoking and helpful in bridging the gap between the theoretical ideas behind gradient descent and how they can be applied to a specific learning problem.