Project 2 - Classification and Regression

An exploration of different methods in Machine Learning

FYS-STK3155 at University of Oslo

Simen Løken

November 2020

1 Abstract

In this report, we'll create and maintain multiple different algorithms and methods for solving both classification problems and regression problems. We'll discuss these methods at length, their pros and cons, and weigh them against each other and the data we're working with. We find that for small data sets or when you don't lose to much accuracy in your results, it is preferable to work with smaller algorithms, like logical regression in regards to classification, or just plain stochastic gradient descent for regression when our data sets are sufficiently small.

2 Introduction

Imagine if you will that you it's pitch dark outside and you find yourself on a mountain. You've got no vision to guide you, and you wish to make it back down. How would you do it?
It's likely that you answered the rhetorical question above with going downhill. It's the logical thing to do if you want to get down and we can represent a problem like this in mathematics quite simply actually. If we instead imagine our mountain to be a some arbitrary function like this:

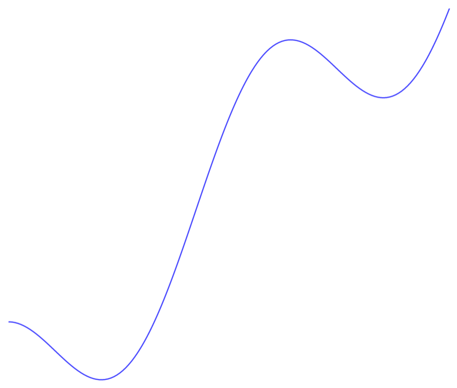


Figure 1: Some arbitrary function with two minimums

The logical approach to finding the bottom in such a scenario would be to use the derivative:

$$f'(x) = \frac{\delta f}{\delta x}$$

which in turn gives us the slope for a given x.
But what if we're working in a three-dimensional space? How would we then go about finding the slope for a given x and y.
If you've ever touched calculus, you might recall a handy thing called the gradient:

$$\nabla f(x,y) = \left(\frac{\partial f}{\partial x}\hat{\imath}, \frac{\partial f}{\partial y}\hat{\jmath}\right)$$

which also happens to return the slope for a given x and y position.
This is absolutely central to the methods we're going to be discussing here.
Imagine instead that our mountain is a cost function. For most cost functions, the goal is of course to get the lowest value possible, that is, the least error and compromises in our model. So what if we could make a algorithm that uses the gradient in it's current position to approach a local minimum on the cost function, and then retain the values that gave us that answer.
In this report, we're going to be examining problems like these, mainly two different types of problems, regression problems and classification problems, and try different methods on them to see their strengths and weaknesses, before finally doing a critical evaluation of what we've shown and learned.

3 Theory and Method

As we hinted at in the introduction, the bulk of this report is going to be using the method called Gradient Descent (and it's derivatives), so let's take a closer look at these methods:

3.1 Gradient Descent

The **Gradient Descent** method is, in it's most general form, a first order iterative optimization algorithm, which we can use to find a local minimum of a function.
A good illustration of Gradient Descent in action can be seen here: This is ideally what we'd wish

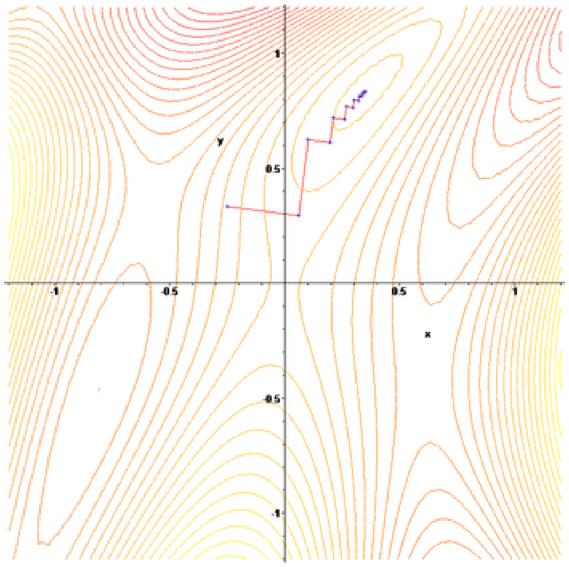


Figure 2: A contour-plot showing the movement given by a Gradient Descent method.
Notice how the function always moves directly "down-stream" from it's position on the surface of the plot. This gives the very characteristic zig-zag of a Gradient Descent method. Note also that it is not a given that we end up at a global minimum
Image Source: [3]

to accomplish with a Gradient Descent method, but instead of moving toward a function minimum, we're instead moving towards the values that give the lowest error, or rather, the least cost.
So what does it look like? Mathematically, that is. Well, in it's most general form, imagine that we have some parameter θ that we wish to minimize. Given what we've already discussed, this θ could be expressed as:

$$\theta_{i+1} = \theta_i - \eta \nabla f(\theta_i) \tag{1}$$

where η is the learning rate (timestep) and $\nabla f(\theta_i)$ is the gradient in for the previous parameter θ_i. Notice how the second term is negative, as we're trying to reach minimize our parameter. Conversely, had we instead used Gradient Ascent, the second term would be positive, and we would've instead been looking for a local maximum.
There are however, other versions of the Gradient Descent method, the **Stochastic Gradient**

Descent method is perhaps more popular, as it better serves to minimize bias and potentially find even better values.
Stochastic, from the Greek word 'stókhos', which is a combination of the words 'aim' and 'guess', is essentially interchangeable with the word random in mathematics. As the name suggests, we instead of going through one point at a time, choose a random piece of data to fit to our model. The main purpose this is to speed up workloads of big data sets. In the case of Gradient Descent, we have to run through all of our samples in our training data to update our parameters once, while in the case of Stochastic Gradient Descent, we only run through one data point.
There is however an expansion to the Stochastic Gradient Descent, the **Stochastic Gradient Descent w/ Mini-batches**.
Here, we instead of choosing only one random point of data, look at a randomly (preferably shuffled as to avoid bias) subset of data. This in turn allows us even faster run times, and allows us to more quickly perform a decent enough fit where our cost function hits a local minimum.
Note that the Equation given in [1] does not change between all these three methods. It is only the data we're examining that changes. There are however some modifications that can be done.
The most common way to modify this algorithm would be to include a momentum, which in turn helps slow run times for when our gradient term gets small. This would be:

$$v_i = \gamma v_{i-1} + \eta \nabla f(\theta_i)$$

where v is the momentum, and γ a momentum parameter. We then get:

$$\theta_{i+1} = \theta_i - v_i \tag{2}$$

Alternatively, an even better method would be to use the Nesterov Momentum, which is given on the form:

$$v_i = \gamma v_{i-1} + \eta \nabla f(\theta_i + \eta v_{i-1})$$

giving

$$\theta_{i+1} = \theta_i - v_i \tag{3}$$

Notice that we are essentially time stepping our gradient term using the previous momentum, which in turn converges even faster, saving precious computing time.

3.2 Adaptive Methods

In addition, there are alternative methods that I have dubbed adaptive methods, namely **Adam** and **Nadam**. Let's take a look at these:
Adam is an adaptive learning rate optimization algorithm. It updates its weights as follows:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}} + \epsilon} \tag{4}$$

where:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$m_t = \beta_1 m_{t_1} + (1 + \beta_1)\nabla f(\theta_i)$$

$$v_t = \beta_1 m_{t_1} + (1 + \beta_2)\nabla f(\theta_i)^2$$

retrieved from [6] and where ϵ is a smoothing parameter, to keep us from dividing by zero and β_1, β_2 are forgetting parameters, typically with with values 0.9 and 0.999 respectively.
We can also combine the Adam algorithm with the Nesterov momentum, giving:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}} + \epsilon}\left(\beta_1 \hat{m}_t + \frac{(1 - \beta_t)\nabla f(\theta_i)}{1 - \beta_1^t}\right) \tag{5}$$

which is commonly referred to as the **Nadam** method.

3.3 What are Neural Networks

So what are Neural Networks? Essentially, neural networks are a way to do machine learning where the computer performs a task after analyzing training samples. This might sound familiar to a normal regression case, which it can be, but quite commonly for, and uniquely, for neural networks is that these examples have been hand-labeled in advance. There are a given set of nodes, or neurons, on each layer. Any given neuron could be connected to many neurons in the previous layer, and it might feed information itself to many neurons in the next layer.
In it's simplest form, a one-hidden-layer Neural Network could look like:
For more advanced applications, much more advance than what we're going to examine in this report, you might instead get something like this:

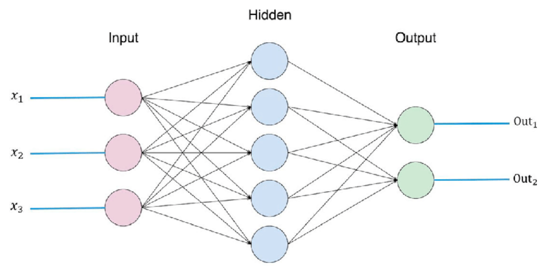


Figure 3: A very simple one-hidden-layer Neural Network with a binary output.
In our case for the MNIST data, we have 10 outputs, the numbers zero through nine.
Image Source: [4]

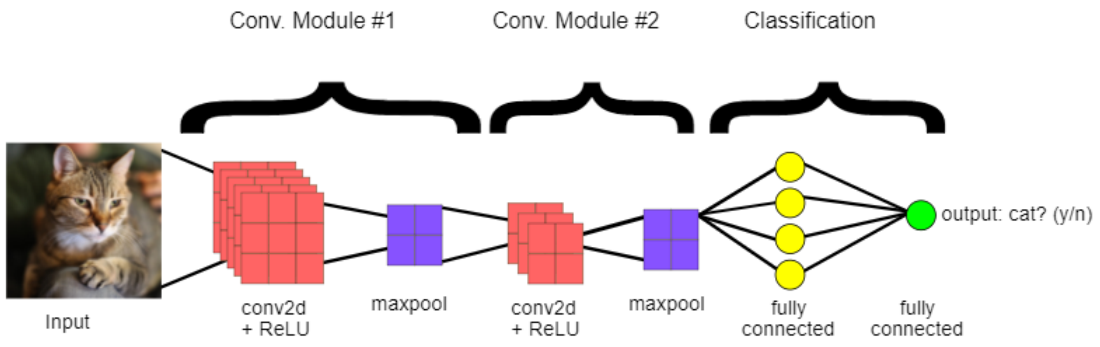


Figure 4: A more advanced model using a method of convolutional Neural Networks (abbreviation: CNN)
Here, data is fed, and then ran through a convolution algorithm and a RELU activation function, before it is subject to max-pooling process. (Max-pooling is reducing the dimensions of the feature map, but still preserving the most important information)
Image Source: [5]

3.4 Regression vs. Classification

So what is the difference between regression and classification?
They are both methods of machine learning, but let's declare them properly and look at the differences.
Regression is simply put mapping a function, let's say f, given from input variables in an array X onto a continuous output variable, z. This allows us to look at distribution movement given the data. We've already talked at length about regression in Project 1, so I'll leave it at that for now, but it is an important distinction.
Classification is however, using the same variables, mapping a function f, given input variables in an array X onto discrete output variables z. That is, we wish to create a model that can read some data, and then try to map it to a set of categories to the best of it's abilities, which means, the main thing separating regression and classification is that when we're working with classification,

we're trying to sort the data into an already defined subset, while in regression we're trying to predict some function f given a correlation between data. You cannot for example let a classification problem keep running "beyond" the scope of the data and it should then come as no surprise that we can't solve these two issues in the exact same way.

3.5 Activation Functions

We'll also need something called activation functions, which are, in short, functions in a given layer that defines it's output given an input.
We'll be using mostly ridge activation functions, some of which are:

$$Sigmoid(x) = \frac{1}{1+e^{-x}} \tag{6}$$

$$Softmax(x) = \frac{e^{-x}}{\sum e^{-x}} \tag{7}$$

$$Softplus(x) = \ln \frac{1+e^{kx}}{k} \tag{8}$$

where k is a sharpness parameter.
Additionally, there are the RELU family of activation functions:

$$RELU(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{9}$$

$$LeakyRELU(x) = \begin{cases} 0.01x & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{10}$$

$$ELU(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{11}$$

where α is a tweakable parameter.
These were just a few examples, and there are many more activation functions, each with their strengths and weaknesses.

3.6 How to evaluate the success of a model

So when all is said and done, how do we know how well we did? There are different ways to do this. Recall from Project 1 that we discussed the mean squared error:

$$MSE(y, y_p) = \frac{\sum_1^N (y - y_p)^2}{N} \tag{12}$$

where y is our data and y_p is the predictory data, both of length $N-1$, retrieved after performing the fit.
You may also recall that we discussed R^2 score, given as:

$$R^2(y, y_p) = 1 - \frac{\sum_1^N (y - y_p)^2}{\sum_1^N (y - \bar{y})^2} \tag{13}$$

There are both a great way to properly gauge how well your model has preformed, with the caveat that that model is of a regression problem. However, in this report we're also going to be looking at the very famous MNIST-data, which is entirely a classification problem, so we're going to have to introduce some new methods for gauging. What if we instead decided to compare our real data to the predictory one, and give it one point if it is correct, then divide the total points by the length of the data?
This is called the accuracy, and is a very common tool for gauging the fit of a classification model. In pseudo code, this would be:

```
for i in predictory:
    if i == real_data:
        corr += 1
acc = corr/len(predictory)
```

Lastly, we also need to define a cost-function for each scenario.
For a regression problem, it is natural to use the mean squared error during the calculation to gauge if we've found a minimum or not, meaning:

$$C_r(y, P) = MSE(y, P) \tag{14}$$

however, it is not so easy for a classification problem. While it is natural for regression to always calculate how far off you were, it is not for classification.
To better illustrate this, say we're working with the MNIST set, and we'd like to sort the numbers. What do we do when the algorithm gets it wrong? Sure, 5 might look like a 6, and it messing up is understandable, but how do we quantify that as an error.
The answer is you can't, or maybe you can, but you shouldn't. Instead, let us use something called the Cross Entropy:

$$C_c(y, P) = -\sum_{i=1}^{N} y_i \log P_i + (1 - y_i) \log(1 - P_i) \tag{15}$$

which, when made as small as possible, should give us the best possible results.

4 Results

We first compare a simple Stochastic Gradient Descent with mini-batches to previous methods we have developed in Project 1.

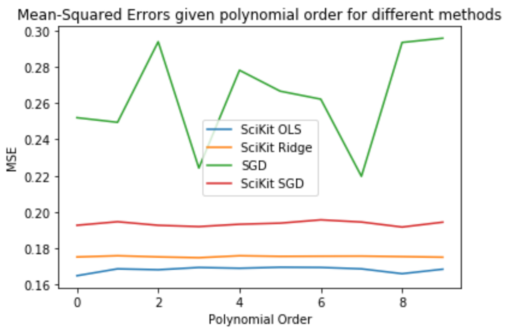
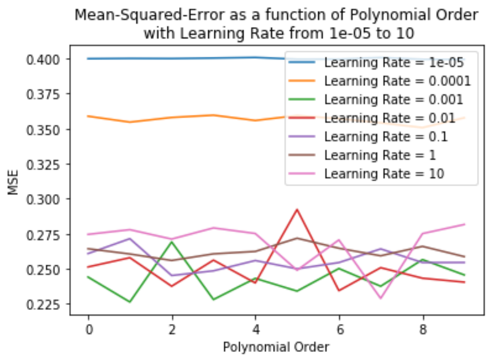
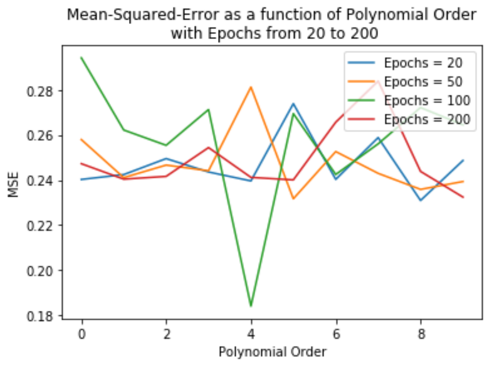


Figure 5: A plot showing the MSE of own SGD, SciKit's SGD, SciKit's OLS and SciKit's Ridge method for varying polynomial orders.

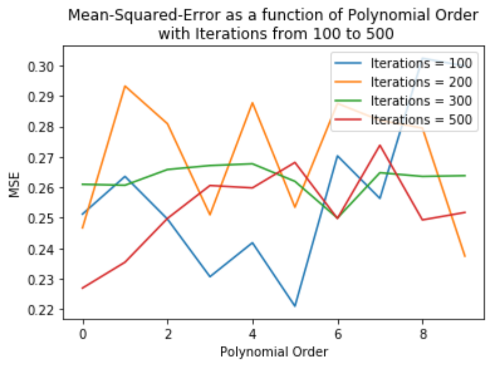
Let us now isolate our model, and see how it changes for different variables:



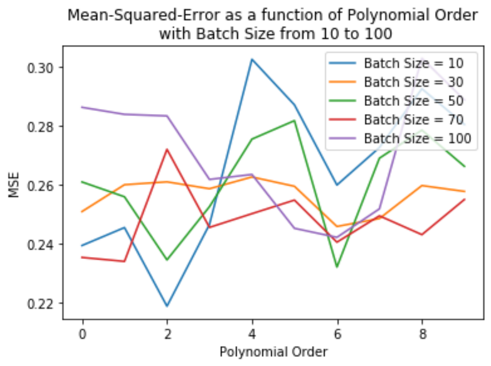
(a) Varying learning rate



(b) Varying epoch count



(c) Varying iteration count



(d) Varying batch size

Figure 6: Our SGD solution by itself for many different variables.

Let us now examine a FFNN case with regression:
We use our design matrix X and frankedata z from Project 1, and get:

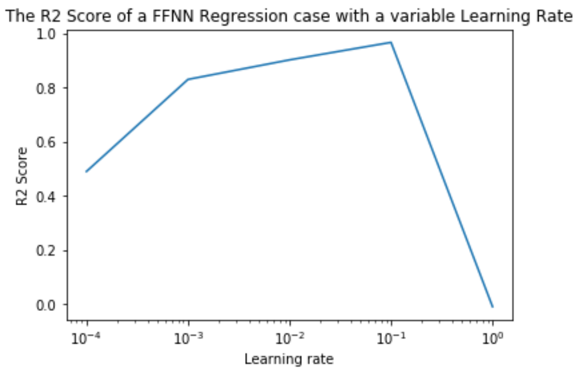


Figure 7: The R2 Score for a FFNN Regression case with variable learning rate
We see clearly that the ideal learning rate for this model is about 0.2, and it gives a very good fit. This is using RELU.

We try again for Sigmoid this time, giving

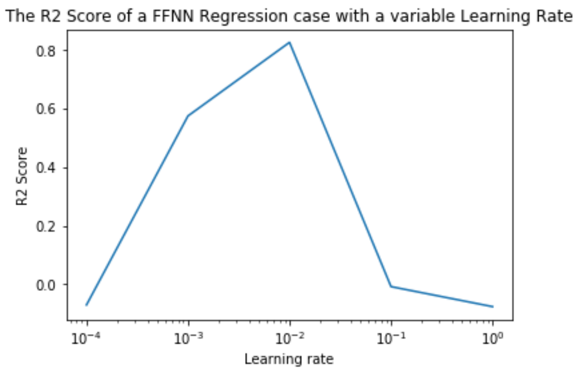
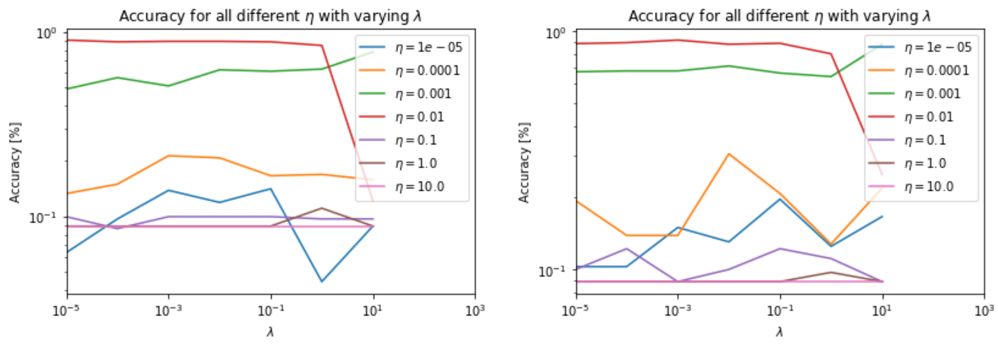


Figure 8: The R2 Score for a FFNN Regression case with variable learning rate
We see clearly that the ideal learning rate for this model is about 0.01, and it gives a very good fit. This is using Sigmoid.

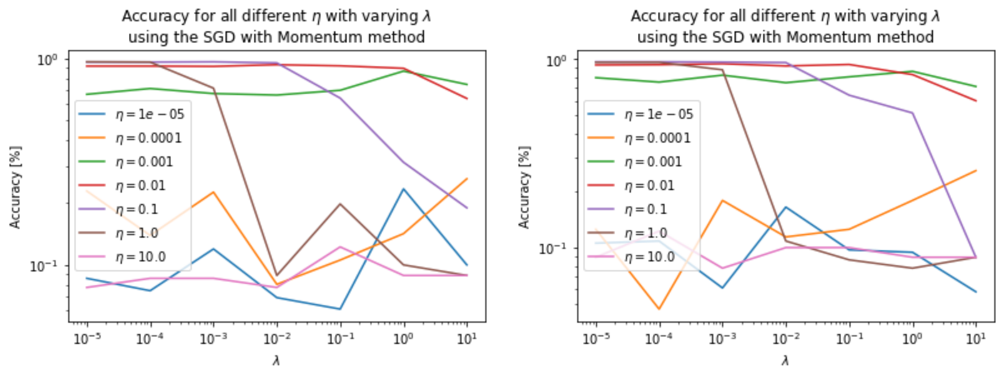
We now move over to the classification cases.
Let us first examine just the MNIST data alone with variable η and λ, for different numbers of neurons.



(a) 50 Neurons (b) 100 Neurons

Figure 9: Our accuracy for different values of λ and η. First for 50 Neurons and then for 100. Using our own FFNN code.

Let us now look at the Tensorflow version:



(a) 50 Neurons (b) 100 Neurons

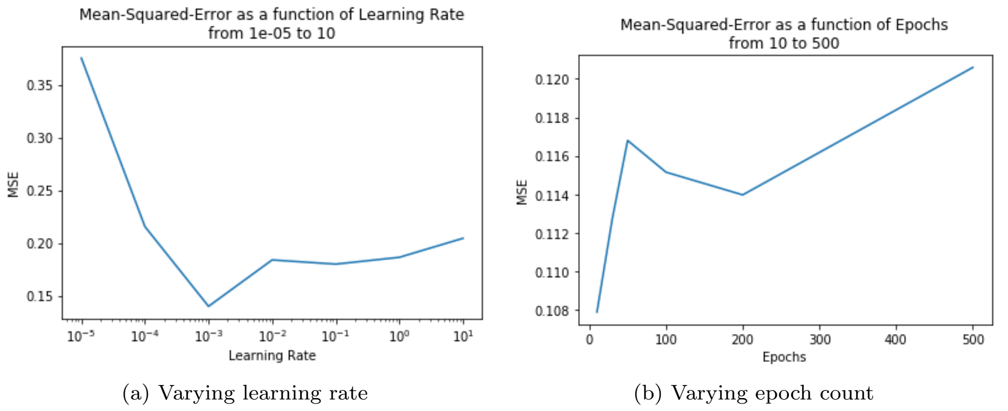
Figure 10: Our accuracy for different values of λ and η. First for 50 Neurons and then for 100. This time using the Tensorflow set of functions.

Finally, let us now examine Logistic Regression. We use trial and error and find a set of optimal parameters for three methods, GD, SGD and SGD with Mini-batches: Let us now examine how

	Iterations	Epoch Count	α	k	Batch Size	η	MSE	R^2
GD	800	500	0.6	-	-	1	0.019	0.89
SGD	600	200	0.7	2.3	-	0.006	0.12	0.32
SGD w/ mb	400	200	0.8	-	50	0.0005	0.11	0.38

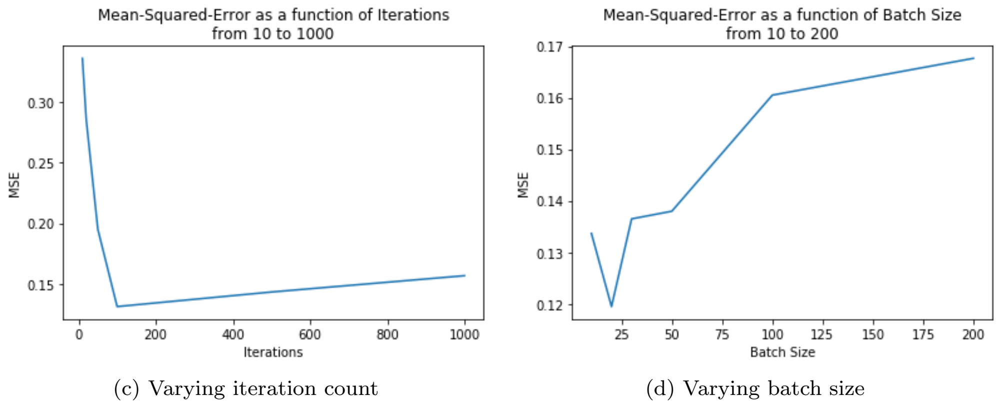
Table 1: A table showing different initial values for Logistic Regression, giving fairly good results. α is a parameter, so is k

Logistic Regression changes for the different variables. For simplicity's sake (and because it's the best model for big data sets, so it's the most relevant), let's use Stochastic Gradient Descent with Mini batches.



(a) Varying learning rate

(b) Varying epoch count



(c) Varying iteration count

(d) Varying batch size

Figure 11: Our Logistic Regression solution for varying values.

5 Discussion

Run times vs Accuracy for the Gradient Descent Methods

Looking at our data and what we've seen so far, it may be tempting to call Gradient Descent a surefire winner. With a mean square error as low as 0.02 at its lowest. However, this doesn't show the whole picture. Recall from the theory section where we discussed why we prefer Stochastic over Gradient.
Let's examine these run times. Since I have no proper big data set, let's instead set the polynomial order of our design matrix to be 20, and let it run.
We get:

	GD	SGD	SGD w/ mb
$\eta = 10^-6$	0.29	0.41	0.40
$\eta = 10^-5$	0.11	0.40	0.37
$\eta = 10^-4$	0.044	0.39	0.24
$\eta = 10^-3$	0.023	0.31	0.12
$\eta = 10^-2$	0.020	0.13	0.17
$\eta = 10^-1$	0.016	0.13	0.18
Run time [s]	474.19	8.76	22.34

Table 2: The Mean Square Error for a given learning rate for three Gradient Descent method, and finally the run time.

It should be obvious to you looking at this table why we shouldn't be using pure Gradient Descent. A design matrix with a polynomial degree of 20 isn't necessarily even that big. It becomes

$$X = [N \times 231]$$

where N is the length of the input arrays x and y, which also form the corresponding Frankefunction.

Discussing Activation Functions

Let us discuss a few of the different most popular activation functions, and weigh some of their pros and cons.
The **Sigmoid** function is perhaps one of the most popular activation functions and for good reason. It's characteristic S-shape means that small changes for x in turn gives large changes in y, meaning that y is ideally always large, giving us a distinct advantage when we're looking to classifiy values in a classification case.
In turn though, the S-shape also hurts us when we're far away from $x = 0$, because as x increases beyond ± 3 or so, we find an almost flat function, meaning that our gradients become very small, and thus, leaving us "dead in the water" so to speak, if we haven't yet hit our minimum
The **RELU** function is also a very popular pick, because of it's unique qualities. Studying the RELU function, we see that it returns zero for all negative values of x. This in turn means the neuron does not get activated, meaning that only certain neurons are active at a time, and that all active neurons are efficient/actually working/contributing. In turn, this helps computation times for larger data sets.

In turn of course, this also means that we can get permanently inactive neurons, as our gradient for negative x will always be zero.
This is however addressed by the even better Leaky RELU function, which retains all the previous benefits of the RELU method, but also accounts for the inactive neurons by introducing a new parameter close to zero, so that those neurons can also pass their information forward in the neural network. This makes the Leaky RELU function a good activation function all around.
So which one should you choose? Well, there is no clear cut answer to this. The honest answer is probably that you should try them all and see what gives you the best results, but a general rule of thumb would be that the ones discussed above are usually safe bets.

Discussing Methods - which method is the best?

So let's get down to the meat of it. Which method is best? The most logical way to tackle this, since we've been looking at two problems for this report, is to look at both regression and classification separately.

Classification

Again, like discussed above. The important thing to look at here would be run times vs. accuracy. Ideally, if a system is solvable within a reasonable margin using logistic regression, then it would follow naturally that that would be best. Neural Networks are computationally expensive, and require both more tweaking (adjusting of parameters) and (in my opinion) work. That being said, if we cannot get sufficiently accurate results with a logical regression, then we need to make use of a neural network.
Additionally, I'd also like to mention that logical regression is in itself not a classifier, but can be used to make one, where as Neural Networks are.

Regression

For regression however, it is in my opinion not as black and white. We can clearly see in the table above that this too is a case of accuracy vs. run times.
I personally think that the best solution a case of regression would be using the Stochastic Gradient Descent method. My reasoning is that it is both light weight and accurate, like previously mentioned, requires a lot less tinkering for a lot more. While you could of course use a neural network to model a regression case, the benefit is negligible, not to mention Neural Networks are a lot more prone to overfitting, whereas the Stochastic Gradient Descent is a lot more resistant.
So again, it depends on the complexity of the data set you're analysing, and if you can get away with a simpler method as opposed to a neural network, you should probably do that.

Confessions

You might've noticed by now that I have misunderstood part e) entirely. I don't know how it happened, and I didn't notice until friday night (an hour or so prior to the deadline). I was of course supposed to use the Logistic Regression method for classification purposes, not regression purposes
That being said, it should be easy to fix, all that we'd need to do was to instead cast a predictory array z against the true z value, and then binarily decide whether or not they are equal, as discussed in the theory section regarding accuracy. I'd do it if I had the time, and will fix it (after this project is graded). This is part of the reason why I'm a bit past the midnight delivery time, as I have been debating pulling an all-nighter to fix this.

6 Conclusion

In conclusion, we've worked with both our design matrix and the MNIST data set so as to visualize and solve both regression and classification problems. We've also looked at and discussed in length the different activation functions, and which ones to use when.
We've lastly shown and discussed some peculiarities with different algorithms, and we've come to the conclusion that for a given problem there is no set in stone answer for what is best, and you should instead choose a solution/solver/method with regards to the material you're working with. That being said, as a general rule of thumb, if you can get away with it, then the lighter computational methods are preferable to neural networks, in terms of computing power vs. result.

References

[1] URL: http://yann.lecun.com/exdb/mnist/.

[2] URL: https://compphysics.github.io/MachineLearning/doc/Projects/2020/Project2/pdf/Project2.pdf.

[3] URL: https://en.wikipedia.org/wiki/Gradient_descent.

[4] URL: https://www.researchgate.net/figure/An-illustration-of-a-neural-network_fig2_336701551.

[5] URL: https://developers.google.com/machine-learning/practica/image-classification/convolutional-neural-networks.

[6] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014.

[7] Attyuttam Saha. URL: https://medium.com/ai-in-plain-english/comparison-between-logistic-regression-and-neural-networks-in-classifying-digits-dc5e85cd93c3.