UCL COMPUTER SCIENCE, MSC MACHINE LEARNING

# Numerical Optimisation, Assignment 3: Project

COMPGV19

Edward Brown: 16100321

May 30, 2017

This project investigates using a Support Vector Machine (SVM) to classify handwritten digits in the famous MNIST handwritten digit dataset.

The first part of this study solves the primal problem in a variety of different ways. It compares backtracking and line search in gradient descent to two non-linear conjugate gradient descent methods. A hyper-parameter search is carried out for the stochastic gradient descent regime. Furthermore, a brief experiment is carried out using a momentum update in the gradient descent regime.

The next part uses Sequential Minimisation Optimisation [10] to solve the dual SVM formulation. It then goes on to explore an algorithm which, to the best of my knowledge, is novel and uses it to solve the dual optimisation problem. Two modifications to this algorithm are explored. I would add at this point that I believe it to be novel, since I cannot find it anywhere, but given my amateur mathematical ability, it is more than likely that someone else has used it before, or else it is a simple version of more sophisticated methods that already exist. Regardless, it has been interesting investigating it.

Please Note: The associated code takes about 5 minutes to run.

# CONTENTS

# 1 MNIST

The data-set used in this project the MNIST data-set. It is a common benchmark for machine learning algorithms. Each data-point is a 28 by 28 grey-scale image of a handwritten digit, so a 784 dimension space.

Figure 1: A handwritten 0

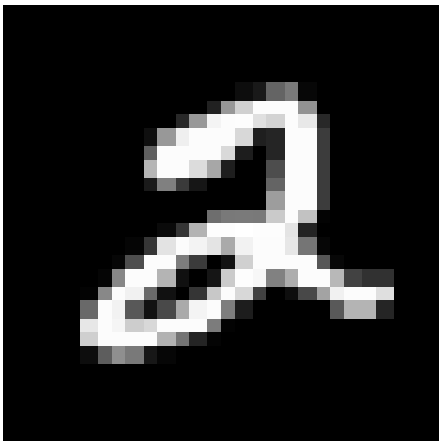

Figure 2: A handwritten 2

The motivation for this data-set is its obvious extension to higher dimension images. The dimensions of the data-set could be increased by simply increasing the resolution of the images, without the underlying problem changing much. This is especially useful given that a large problem of SVM classifiers is that they, in practice, take $O(n^2)$ to solve, and 784 dimensions is actually relatively low for modern day imaging data-sets.

However, the SVM is well suited to binary class classification problems [1]. Therefore, for the purposes of the optimisation problem, the MNIST data shall be used as a predictor of arbitrarily the digit 3 versus the rest.

For this project, given the need to run quickly, a smaller subset of the total dataset is used. Typically a random selection of 1000 images.

# 2 Part 1: Primal

## 2.1 Primal Formulation

The idea behind an SVM classifier is to find a hyperplane than separates two data classes. However, it goes further in that it wants to maximise the margin between these classes, for which the hyperplane cuts in half.

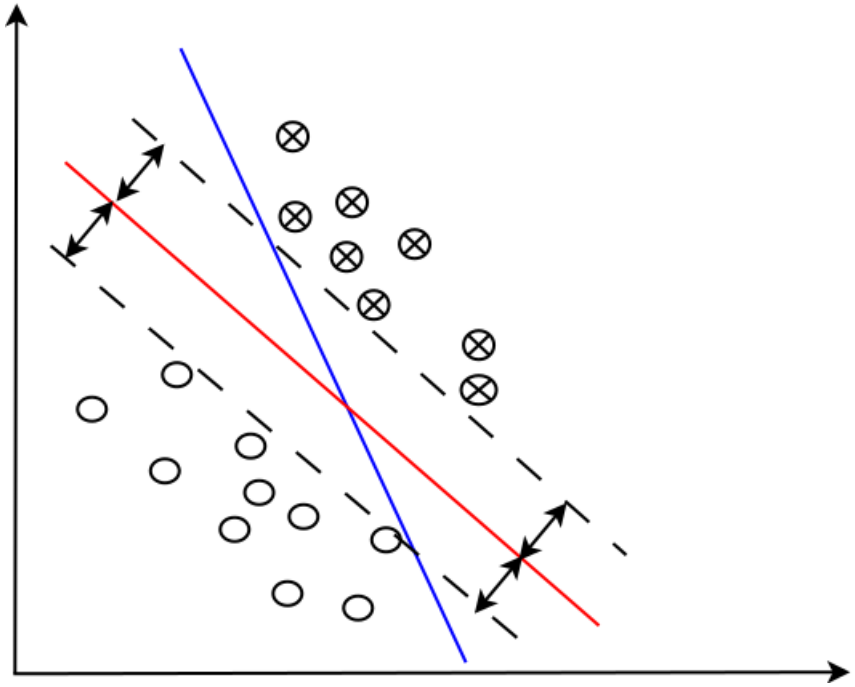

Figure 3: Separating Hyperplanes

As you can see in the figure above, there are an infinite amount of hyperplanes separating the points (by rotating/translating the hyperplane), but we want the one which maximises the margin between the classes. The blue line is a separating hyper plane, however, the red-line maximises the margin between the two clusters.

This problem can therefore be reduced to the optimisation problem:

$$\max_{w} \frac{2}{\|w\|}, \quad \text{subject to:}$$

$$\mathbf{w}^T\mathbf{x}_i + b \geq 1 \quad \text{for} \quad y_i = +1$$
$$\mathbf{w}^T\mathbf{x}_i + b \leq 1 \quad \text{for} \quad y_i = -1$$

where $\mathbf{w}^T\mathbf{x}_i + b = 1$ is the equation of the separating hyperplane.

This can then be equivalently reduced to:

$$\min_{\mathbf{w}} \|\mathbf{w}\|^2, \quad \text{subject to:} \quad y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 \quad \text{for} \quad i = 1...N \quad [2]$$

### 2.1.1 Inseparable Data Problem and Slack Variables

A problem with SVMs as a classifier is that sometimes the data is not linearly separable. That is, there is no hyperplane that actually separates the classes. The way to get around this problem is to introduce 'slack' variables. These variables is to make the constraints less rigid and allow errors, but aim to minimise these errors.

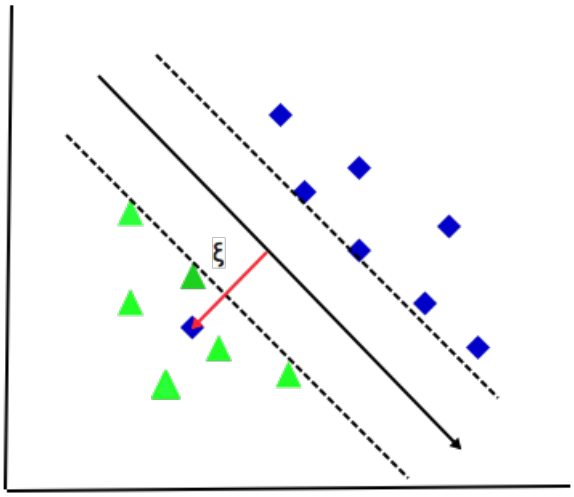


Figure 4: A data-point and its slack variable

As you can see in the above figure, the data clusters are linearly inseparable. However, the classifier allows deals with this with the introduction of slack variables, $\xi$. This problem can be formulated by:

$$\min_{\mathbf{w},\xi_i} \left(\|\mathbf{w}\|^2 + C\sum_i^N \xi_i\right), \quad \text{where} \quad y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i$$

Given that $\xi_i \geq 0$, the constraint can be rewritten as:

$$\xi_i = \max\left(0, 1 - y_i f(\mathbf{x}_i)\right) \quad \text{where} \quad f(\mathbf{x}_i) = \mathbf{w}^T\mathbf{x}_i + b$$

The optimisation problem then becomes:

$$\min_{\mathbf{w},\xi_i} \left(\|\mathbf{w}\|^2 + C\sum_i^N \max\left(0, 1 - y_i f(\mathbf{x}_i)\right)\right) \quad [2]$$

The algorithm is very similar to least-squares and logistic regression, except here the loss function is called the 'hinge loss' function. So called because of its graphical resemblance to a hinge [1].

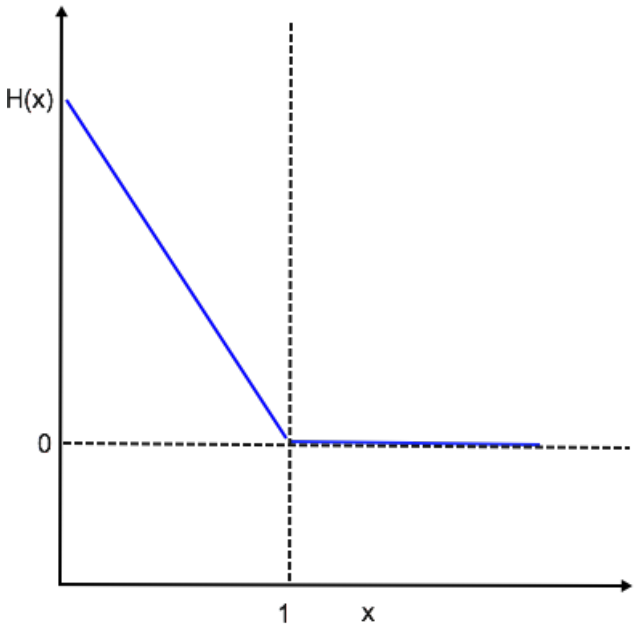


Figure 5: Hinge Loss function

## 2.2 Primal Optimisation Problem

### 2.2.1 Gradient Descent

As above, the primal optimisation problem is found by minimising the following cost function:

$$C(\mathbf{w}) = \|\mathbf{w}\|^2 + C\sum_i^N \max\big(0, 1 - y_i f(\mathbf{x}_i)$$

we can rewrite the above as follows:

$$C(\mathbf{w}) = \frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{N}\big(\sum_i^N \max(0, 1 - y_i f(\mathbf{x}_i)\big)$$

$$C(\mathbf{w}) = \frac{1}{N}\sum_i^N \big(\frac{\lambda}{2}\|\mathbf{w}\|^2 + \max(0, 1 - y_i f(\mathbf{x}_i)\big) \quad [2]$$

where $\lambda = \frac{2}{NC}$, and is the only parameter in the cost function that needs to be picked (instead of the constant $C$). This is the regularisation parameter.

In order to find the minimum of this cost function we can perform gradient descent so that our iterative update of the **w** vector is given by:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \, \nabla_{\mathbf{w}}\big(C(\mathbf{w}_t)\big)$$

where $\eta$ is chosen learning rate [2].

The hinge loss function is not differentiable so the sub-gradient, i.e. set based gradient, needs to be taken.

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \frac{1}{N} \sum_{i}^{N} \big(\lambda \mathbf{w}_t + \nabla_w L(x_i, y_i; w_t)\big)$$

where $L$ is the hinge loss function.

The subgradient of $L$ with respect to **w** are given by:

$$\frac{\partial L}{\partial \mathbf{w}} = -y_i x_i, \text{ for } y_i f(x) < 1,$$
$$\frac{\partial L}{\partial \mathbf{w}} = 0, \text{ otherwise}$$

If we represent the gradient for each data-point as:

$$\frac{\partial L}{\partial \mathbf{w}} = L'(\mathbf{x}_i, y_i, f(x_i)) = L'(x_i), \text{ for short.}$$

This means that our iterative update becomes:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \sum_{i}^{N} \big(\lambda \mathbf{w}_t - L'(\mathbf{x}_i)\big)$$

### 2.2.2 Non-linear Conjugate Gradient Descent

Conjugate Gradient descent is a method whereby an initial direction is chosen $\mathbf{p}_0$ and the function is minimised along this direction. Then a direction *conjugate* to the previous direction is generated. There are a variety of different ways to generated the next conjugate direction, having quite different results in practice. The function is then minimised along this direction. This cycle is repeated until a **0** vector is generated.

The algorithm can be carried out as follows [11]:

given $x_0$,

compute $L_0 = L(x_0), \nabla L_0 = \nabla L(x_0)$

Set $p_0 = -\nabla L_0^T, k = 0$

**while** $\nabla L_k^T \neq 0$ **do**

Compute $\alpha_k$ using e.g. line search $\alpha_k = \min_\alpha L(x_k + \alpha p_k)$

$x_{k+1} = x_k + \alpha_k p_k$

$\beta_{k+1} = \frac{\nabla L_{k+1}^T \nabla L_{k+1}}{\nabla L_k^T \nabla L_k}$

$p_{k+1} = -\nabla L_{k+1} + \beta_{k+1} p_k$

k = k + 1

**end** [11]

The above generation for the next $p_k$ direction is the Fletcher-Reeves version of the algorithm [12].

However, there include other updates that are valid conjugate directions such as the Polak-Ribiere (PR) [13].

$$\beta_{k+1} = \frac{\nabla L_{k+1}^T (\nabla L_{k+1} - \nabla L_k)}{\nabla L_k^T \nabla L_k}$$

The minimisation along the conjugate directions can be carried out using a variety of techniques, such as line search, backtracking or second order methods. For this project, we shall use line search obeying strong Wolfe conditions, but other methods like backtracking can also be used.

For this project, we shall compare the performance of Fletcher-Reeves versus Polak-Ribiere when applied to the MNIST data.

### 2.2.3 Stochastic Gradient Descent

The equation for the gradient descent update of $\mathbf{w}$ contains the sum over all data points. This can require a lot of memory and computation power, especially with modern-day data-sets. With million-pixel images and huge numbers of data-points, this update starts becoming intractable. [1][5].

A way to get around this issue is to perform Stochastic Gradient Descent. Stochastic gradient descent involves randomly picking a training data-point and then updating the weight based on that point [2]. You then repeat this update based on single data points until convergence. The weight update then simply becomes:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \left(\lambda \mathbf{w}_t - y_i \mathbf{x}_t\right) \text{ for } y_i f(x_i) < 1,$$
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \lambda \mathbf{x}_t \text{ otherwise.[2]}$$

where $\mathbf{x}_t$ is the randomly chosen data point for that iteration.

In practice, you can simply arrange the data set in a random order and then iterate through that shuffled data set, so that each data point is covered.

In order to improve the speed of this process however, we can perform this update in batches. We simply update the weight vector, $\mathbf{w}$, by the average of the contributions from each data point in the batch. Given the prominence powerful vectorisation libraries, this is a standard way of performing stochastic gradient descent. [5].

Under this regime, the update simply becomes:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \sum_{i}^{B} \left(\lambda \mathbf{w}_t - L'(x_i)\right)$$

where we operate on a batch of size $B$.[5]

For this project, we shall conduct a hyper-parameter sweep for a chosen regularisation parameter and analyse the results.

### 2.2.4 Momentum in Stochastic Gradient Descent

As an experiment for this project, I decided to trial stochastic gradient descent except with a momentum style descent direction. The point of momentum gradient descent is that it is similar to normal steepest descent except it includes a a velocity vector.

The update becomes:

$$v = \gamma v + \alpha \nabla L_w$$

$$w = w - v$$

where $\alpha$ is the normal learning rate but here $v$ is the new velocity vector.

The hypothesis that will be tested is that we can observe an appreciable improvement in the speed of convergence by adding a momentum to the descent.

Furthermore, a schedule of increasing the weighting to the velocity vector slowly as we approach the minimum shall be employed to see if we can improve performance.

## 2.3 PRIMAL SVM RESULTS/ANALYSIS

The primal SVM routine was written in MATLAB and applied to a dataset of 500 number 3's and 500 of any number but 3 from the MNIST dataset. This is reduced from the full training set simply for the purposes of speed. Given the scope of the project, it was not feasible to write the sort of parallelisation techniques needed to do in-depth hyper-parameter searches on the full data-set. The test set is a slightly larger dataset of 1000 3's and 1000 not 3's taken randomly from the full MNIST test data set.

Furthermore, although obviously an optimisation problem, it is also a Machine Learning problem, therefore it is reasonable to introduce stopping criteria that is the algorithm's performance on the test set. The problem of overfitting is a common issue in machine learning, so fully optimising on the training set is not always a desired result. Therefore it is reasonable to measure an algorithm's optimisation based on its performance on test sets.

### 2.3.1 GRADIENT DESCENT

Gradient descent with both back tracking and line search performed similarly. Within the standard deviations, they both took about the same to converge each run. Interestingly, the costs they arrived at were not actually that similar in some cases. This suggests that although the SVM problem for MNIST is convex, it might be very flat, meaning that if the tolerance for stopping is too high, (indeed for this study it was $10^{-4}$), the algorithm can be fooled into stopping early even if the global minimum is still a little bit away. The results of both these methods appear in figures 6 and 7.

### 2.3.2 NON-LINEAR CONJUGATE GRADIENT

Concerning the Non-linear Conjugate gradient descent, Polak-Ribiere vastly outperformed Fletcher-Reeves. Fletcher-Reeves proved itself to be quite unreliable, often choosing bad directions and resulting in a jump in cost/accuracy when it seemed to be converging. Indeed, the sample trajectory below shows Fletcher-Reeves displaying rather suboptimal behaviour.

As figure 6 shows. For a sample trajectory, the two gradient descent algorithms behaved very similarly, settling at similar costs. However, both the CG methods managed to find costs at almost an order of magnitude in size. This is actually very surprising. It really shows that the cost function is very flat in some places, causing the two gradient descent algorithms to stop early, where the CG algorithms could find much lower costs.

Figure 7 shows that, bar Fletcher-Reeves, the three methods took similar amounts of iterations to converge. In all trials, Fletcher-Reeves failed to converge within the max iteration

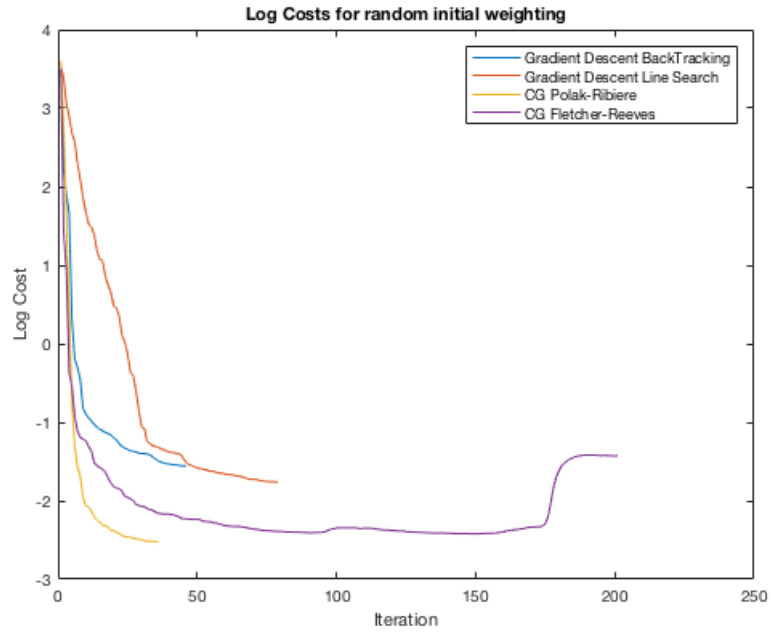


Figure 6: Sample Trajectory for the four primal optimisation methods

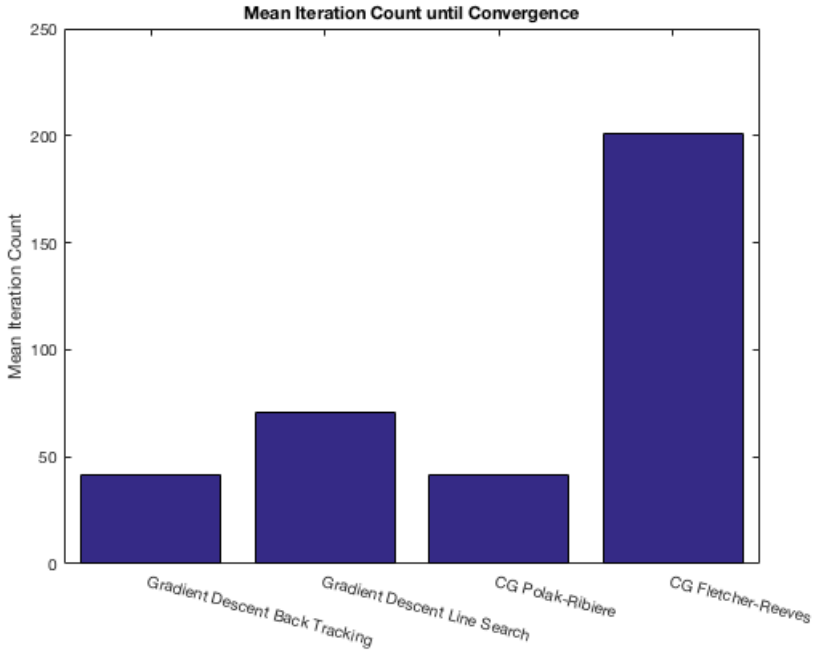


Figure 7: Mean Iteration Counts for the four primal optimisation methods

number. It seems Polak-Ribiere is much more robust in practice as a conjugate gradient method.

It should be noted however that this graph does not reflect the final costs. This was a slight oversight in the analysis and would be well worth investigating in a continuation of the project.

To continue this analysis further, I would suggest other conjugate direction generation techniques to see if there is a boost in performance. This could include Hestenes-Stiefel [14] and Dai-Yuan [15].

Furthermore, other methods like momentum gradient descent and Nesterov's accelerated gradient descent were not used in this study. Both of these regions would be interestingly to see if they could boost performance. Indeed, given the presence of some flat regions of the hyper-space, both these methods could indeed be useful.

### 2.3.3 Hyper-Parameter Search for best Regularisation

For the purposes of the primal problem, no features engineering was employed. Furthermore, the parameter search for a decent performing regulariser, $\lambda$ was simply made on the full gradient descent method. This lambda was found to be $10^{-5}$.

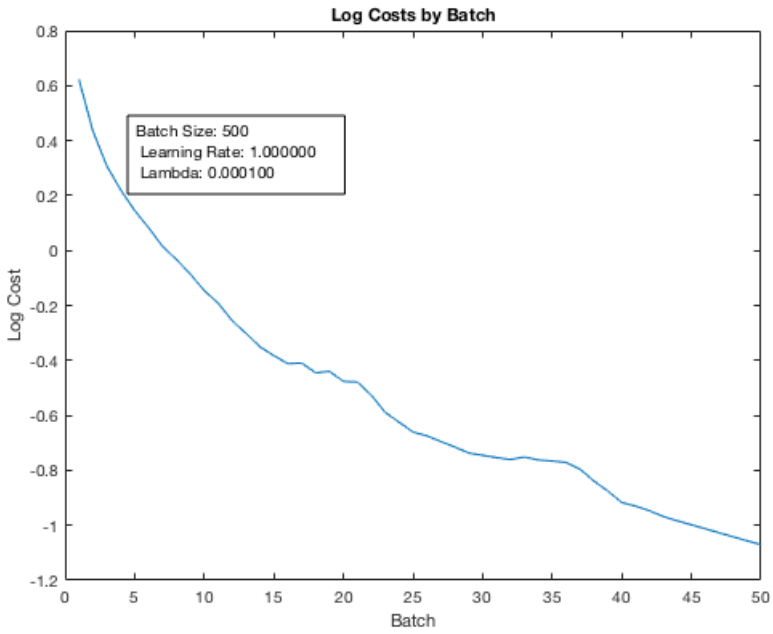


Figure 8: Log Cost for $\lambda = 10^{-5}$

The result for where $\lambda$ was $10^{-5}$ had a slight edge in performance over the other regularisation parameters tried, so for the purposes of comparing the performance of batch sizes in stochastic gradient descent, this shall be the regularisation parameter used.

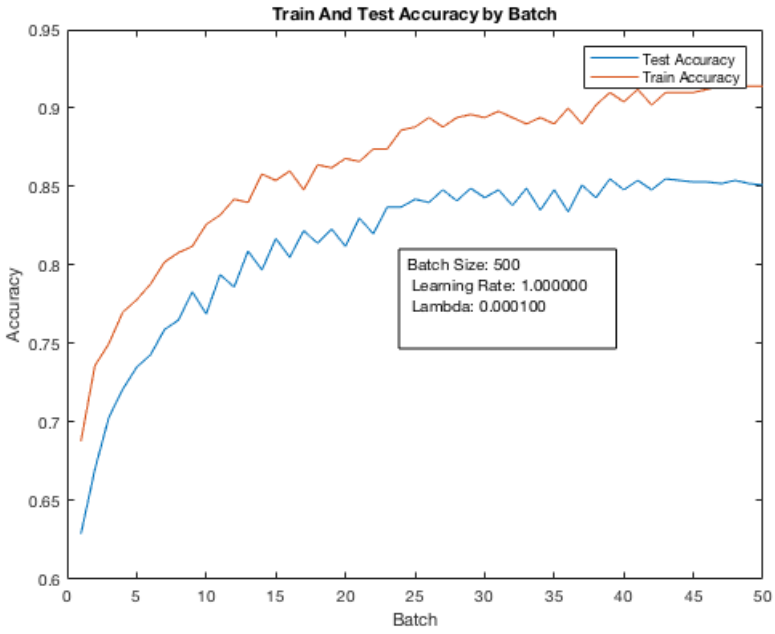


Figure 9: Accuracies for $\lambda = 10^{-5}$

### 2.3.4 Stochastic Gradient Descent Batch Comparison

For the purposes of this comparison, the following batch sizes were chosen: 1,10,100,500. Given that there were only 1000 images in the loaded data set, the batch size of 500 only has to run twice to do a full pass over every data-point. Instead of the points being picked fully at random, the data points were simply arranged in a random order and sorted into batches. That way we know each data point was passed over an equal amount of times.

It should be noted at this point that powerful parallelisation libraries do exist, but this is not necessarily exploited in this comparison. Therefore the comparison is done after each epoch, that is, over one full pass of the data-set.

Figure 10 shows there is a definite sweet spot for the choice of $= 10^{-5}$ where a test accuracy of 90% is reached. This can be summarised in the below table:

| Batch Size | Learning Rate |
|---|---|
| 1 | 10 |
| 10 | 10 |
| 100 | 100 |
| 500 | 10000 |

Table 1: Table to show quickly descending learning rates

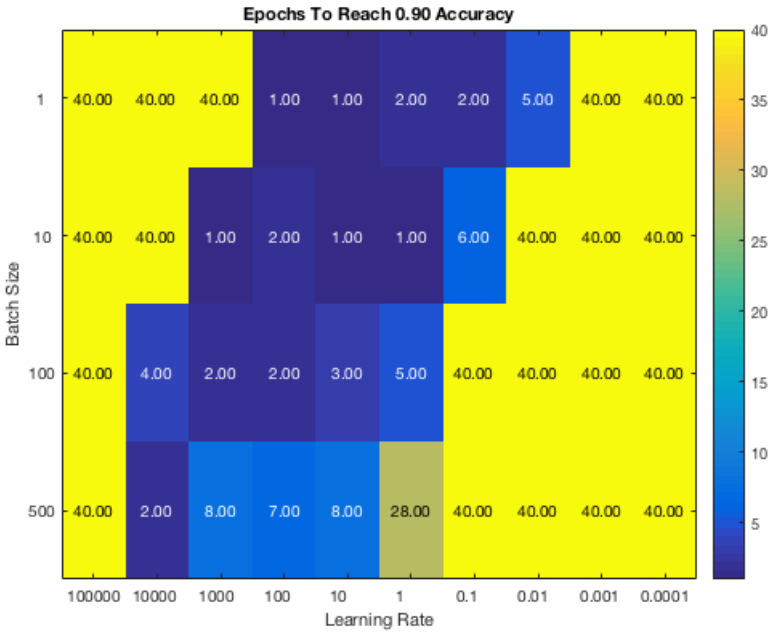


Figure 10: Grid showing how many full passes it took to reach 90% test accuracy

It should be noted that this sweet spot scales roughly linearly with the batch size. This should come as no surprise since the cost function is simply an addition of the contributions from each point. Obviously, greater resolution would be nice in the learning rate, but there was not enough time in this project.

'Early' stopping is a prominent notion in machine learning in neural networks and other classification regimes. It simply suggests that when optimising stochastically, we can stop the algorithm when it reaches a certain test accuracy, which will start to drop when the algorithm starts to overfit. So analyses like the one in 10 can be useful if we are simply looking to quickly reach a certain test threshold and are not interested in overfitting the training data.

The next analysis shows which learning rates were able to find the lowest costs.

Figure 11 paints a different story for the optimum learning rates. It is summarised in the table below.

What this table shows is that although these learning rates were slower to hit higher accuracy, they go on to converge to much lower costs in the long run than the fast movers. However, it should be noted at this point that lower costs does not actually mean better performance in the long run, as the training data may not well reflect test data.

What is interesting about figure 12 is that it does not correspond to either the fast movers or the lowest cost combinations. This is perhaps explained by the stochasticity of the method.

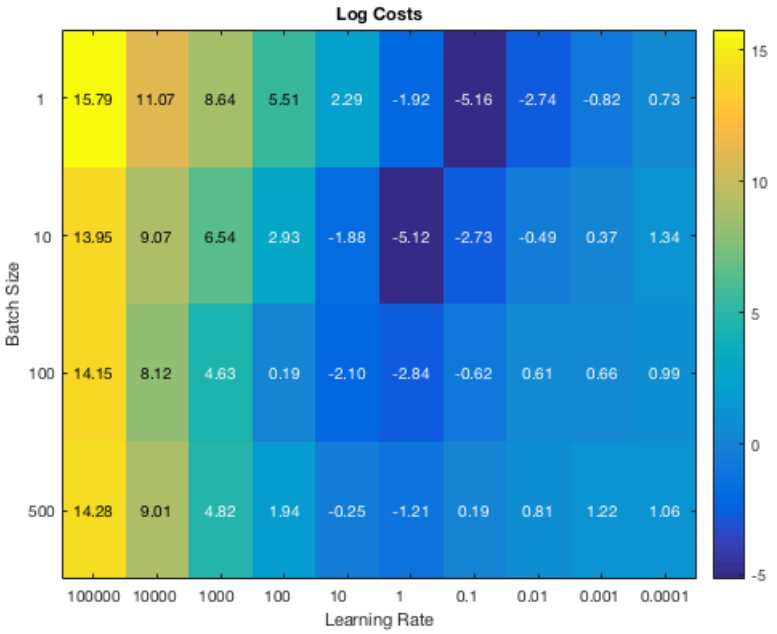


Figure 11: Grid showing how many full passes it took to reach 90% test accuracy

| Batch Size | Learning Rate | Log Cost |
|---|---|---|
| 1 | 0.1 | -5.16 |
| 10 | 1 | -5.12 |
| 100 | 1 | -2.84 |
| 500 | 1 | -1.21 |

Table 2: Optimally performing learning rates by batch size

The descent in total cost is not a smooth transition from batch to batch, indeed, there is no guarantee that passing over a batch will indeed improve the accuracy. If we look at the row for a batch size of 1, the discrepancy between learning rate 10 and 100 really highlights the randomness in the data. One particular outlier can throw off the test accuracy when covered stochastically. As mentioned above too, a low cost on the training data does not necessarily correspond to a good test accuracy, perhaps explaining the discrepancies in these accuracies.

The last, and perhaps least interesting heat map- figure 13, shows that the algorithms were able to fully overfit on some of the combinations. This means that there is a fully separating hyper-plane on the data. Due to the low data numbers, only 1000 images, this is maybe an expected result and but also evidence that an SVM is able to fully overfit and still produce decent test accuracies. This makes a little sense given that regularisation is built into the concept of the algorithm - the widest margin separator.

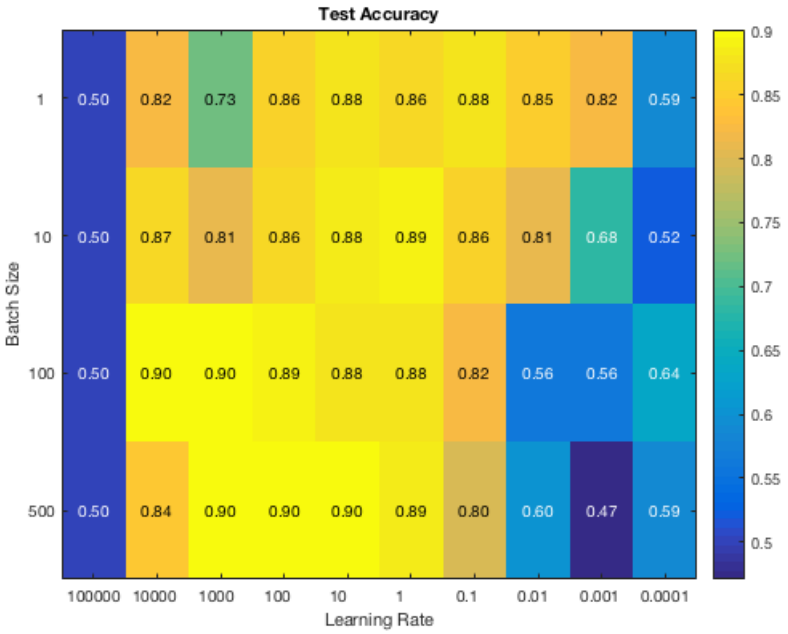


Figure 12: Heat Map showing the Test Accuracies by batch size and learning rate

| Batch Size | Learning Rate | Test Accuracy |
|---|---|---|
| 1 | 0.1 | 0.88 |
| 10 | 1 | 0.89 |
| 100 | 1000 | 0.9 |
| 500 | 1000 | 0.9 |

Table 3: Optimally performing learning rates by test accuracy

Figure 14 demonstrates a combination that achieves a 100% training accuracy. The separator is still able to achieve approximately 0.87% on the testing data. I would expect this value to be much higher with more training data. Furthermore, the more training data, the less likely the algorithm is able to overfit too.

### 2.3.5 SGD Momentum Analysis

The results from this experiment were slightly disappointing. A hyper-parameter search was conducted ahead of time to choose the best learning rate for the three different categories: Normal online SGD, SGD with momentum, SGD with a momentum schedule. Unfortunately, within error and averaging, there seems to barely be an improvement in the speed of convergence when momentum is applied online.

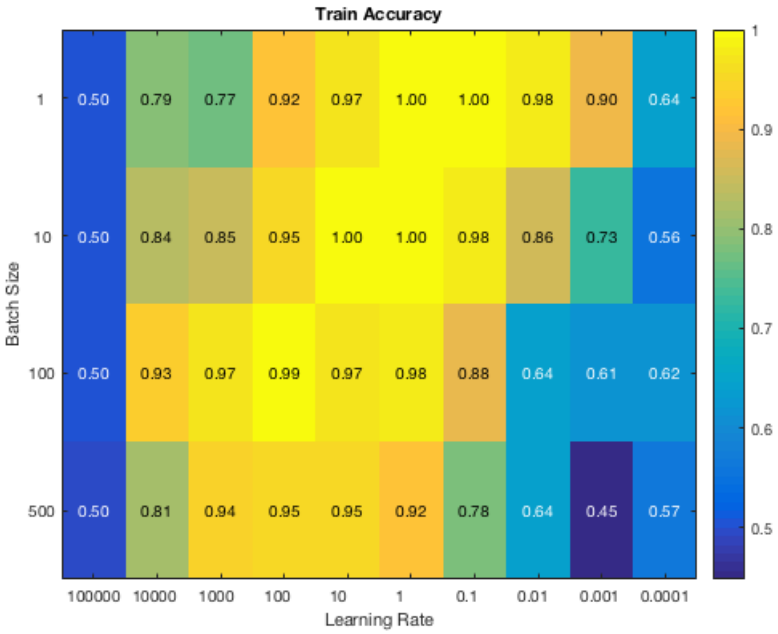


Figure 13: Heat Map showing the Train Accuracy by batch size and learning rate

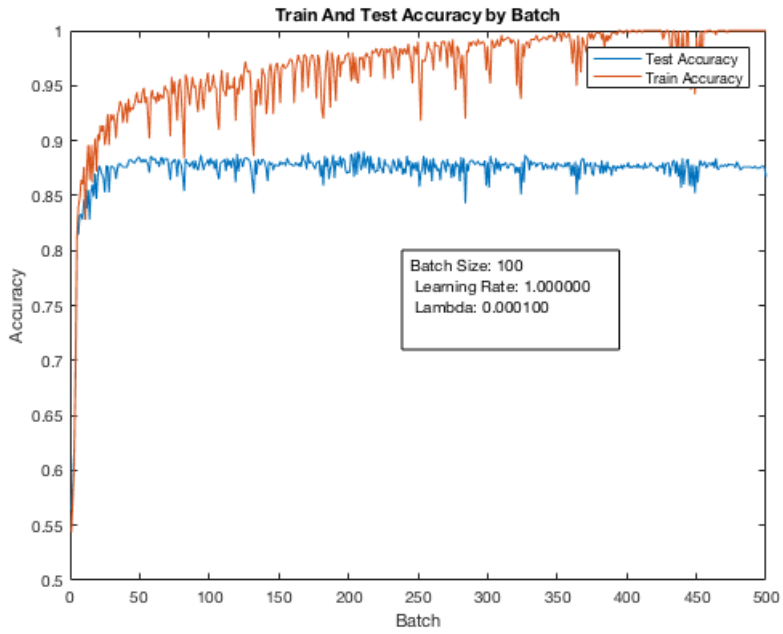


Figure 14: Combination that achieves 100% training accuracy

Figures 15 and 16 show a typical trajectory for the three categories, as you can see, the improvement over normal momentum updates is simply not there.

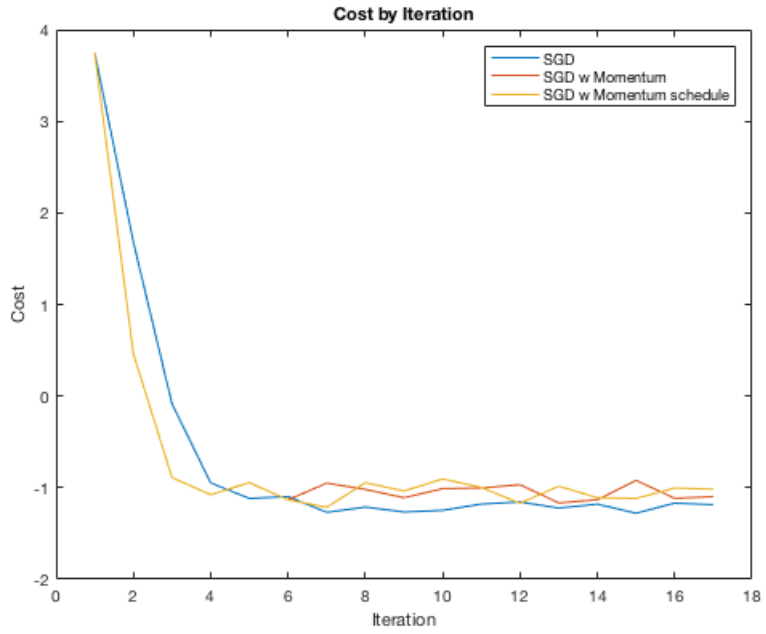


Figure 15: Cost by Iteration for the three categories

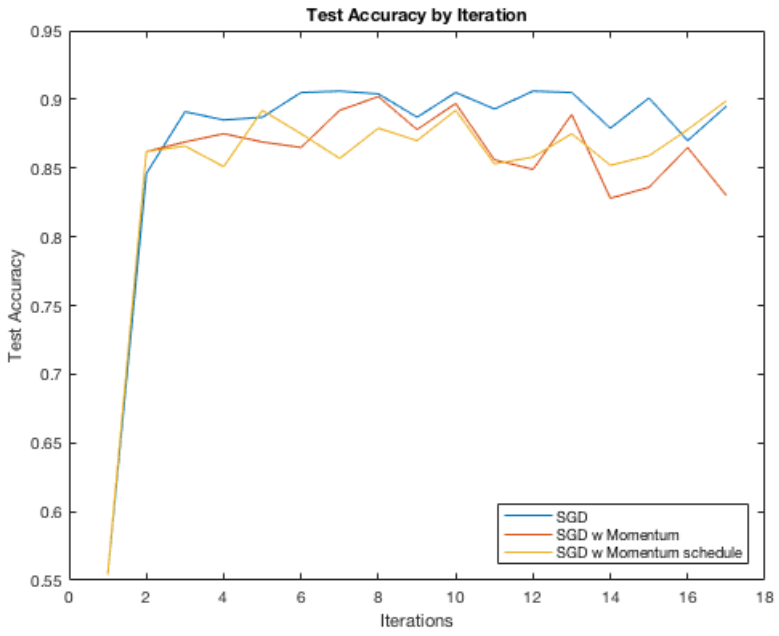


Figure 16: The corresponding test accuracies for the three categories

To further to this analysis, it would be prudent to look into different momentum schedules (functions to change gamma) too see if better performance is achievable. Furthermore, it is quite probable that momentum could help on the full gradient descent problem, so it

would be interesting to apply a similar method in that domain.

# 3 PART 2: DUAL

## 3.1 DUAL FORMULATION

### 3.1.1 OPTIMISATION PROBLEM

The above formulation is known as the primal SVM problem. There is another equivalent formulation, which under certain circumstances can perform the same job with less computation. It is known as the dual problem.

The idea is that minimisation of the loss function with respect to the weight vector, $\mathbf{w}$, can be replaced with a maximisation of the Lagrange Multipliers, $\alpha_i$.

To start with, it should be noted by the Representer Theorem that $\mathbf{w}$ can be represented by a linear combination of the training data [9].

$$\mathbf{w} = \sum_{i}^{N} \alpha_i y_i \mathbf{x}_i$$

Noting this, we can also reformulate the minimisation problem to a maximisation of the Lagrange Function:

$$\max_{\alpha} L(\mathbf{w}, \alpha) = \frac{1}{2}\mathbf{w}.\mathbf{w} - \sum_{j} \alpha_j y_j \mathbf{x}_j \big((\mathbf{w}.\mathbf{x}_j + b) y_j - 1\big).[6]$$

subject to $b = y_k - \mathbf{w}.\mathbf{x}_k$ for any $k$ where $\alpha_k > 0$.

Substituting in $\mathbf{w}$ and $b$, we arrive at:

$$\max_{\alpha} \big( \sum_{i} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \mathbf{x}_j \big)$$

subject to $C > \alpha_i > 0$, $\sum_i \alpha_i y_i = 0$. [6].

Once we have computed the $\alpha_i$s, we can then reform $b$ and $\mathbf{w}$, and solve the primal problem.

So now we have a different but equivalent optimisation problem. The real difference lies in the fact that the optimisation no longer relies on $\mathbf{w}$ but on a dot product of the training data and the $\alpha$'s. So what happens in the final optimisation is that most $\alpha$'s are 0 and the only non-zero elements are correspond to data which are on the edge or on the wrong side of the margin - the support vectors.

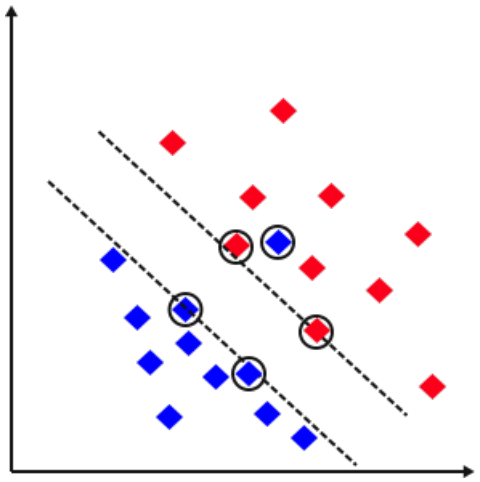

Figure 17: Support vector points are circled

This is significant since the SVM is able to computationally out-perform the primal version under certain circumstances because it has very sparse entries- just the support vectors. If the data has much more dimensions than data-points, the dual may start to become more efficient than the primal. [6]

For the purposes of the project, it would be interesting to compare the performance on the chosen optimiser of solving the dual vs. the primal version.

### 3.1.2 KKT CONDITIONS

It should be noted that KKT conditions are necessary and sufficient for the optimisation of a convex function [7]. If we consider the Lagrangian again:

$$L(\mathbf{w},\alpha) = \frac{1}{2}\mathbf{w}.\mathbf{w} - \sum_j \alpha_j y_j \mathbf{x}_j \big((\mathbf{w}.\mathbf{x}_j + b) y_j - 1\big).[6]$$

the KKT conditions would then be

$$\frac{\partial L}{\partial \mathbf{w}} = 0, \text{ and } \frac{\partial L}{\partial b} = 0$$

computing both these derivatives:

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_i \alpha_i y_i \mathbf{x}_i = 0$$

$$\frac{\partial L}{\partial b} = -\sum_i \alpha_i y_i = 0$$

which you will recognise from the dual formulation above. So solving the SVM problem is equivalent to finding the optimal KKT conditions.

### 3.1.3 Sequential Minimal Optimisation

Sequential Minimal Optimisation (SMO) is an algorithm which solves the dual SVM formulation [10]. As above, we wish to solve the optimisation problem:

$$\max_{\alpha}\Big(\sum_{i}\alpha_i - \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j \mathbf{x}_i\mathbf{x}_j\Big)$$

subject to $C > \alpha_i > 0$, $\sum_i \alpha_i y_i = 0$. [6].

The SMO algorithm solves this problem by breaking it down into smaller sub problems. The basic layout of the algorithm can be broken down into three steps:

1. Find a Lagrange multiplier $\alpha_1$ that breaks the KKT conditions mentioned above.

2. Find a second $\alpha_2$ and optimise both.

3. Repeat until no more $\alpha$ values are changing.

Broadly, the second step can be thought of as trying to find a second multiplier which has enough wiggle room to accommodate a change in the first multiplier.

For this project, a more simplified version of the original SMO will be employed due to its ease of use.

### 3.1.4 Algorithm Investigation: Residual Redistribution

This section is perhaps the most exciting part of the project. After investigating the performance of the SMO algorithm on the MNIST data, I started thinking trying to optimise the dual in other ways. I started testing around with holding the KKT conditions whilst performing gradient descent and arrived at an algorithm that seems to work very well for the SVM dual formulation.

**Derivation**
Here's the idea, as above the dual optimisation problem is reduced to:

To start with, it should be noted by the Representer Theorem that $\mathbf{w}$ can be represented by

a linear combination of the training data [9].

$$\mathbf{w} = \sum_{i}^{N} \alpha_i y_i \mathbf{x}_i$$

Noting this, we can also reformulate the minimisation problem to a maximisation of the Lagrange Function:

$$\max_{\alpha} L(\mathbf{w}, \alpha) = \frac{1}{2}\mathbf{w}.\mathbf{w} - \sum_{j} \alpha_j y_j \mathbf{x}_j \big((\mathbf{w}.\mathbf{x}_j + b) y_j - 1\big).[6]$$

subject to $b = y_k - \mathbf{w}.\mathbf{x}_k$ for any $k$ where $\alpha_k > 0$.

Substituting in $\mathbf{w}$ and $b$, we arrive at:

$$\max_{\alpha} \big( \sum_{i} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \mathbf{x}_j \big)$$

subject to $C > \alpha_i > 0$, $\sum_i \alpha_i y_i = 0$. [6].

Suppose we ignore the constraints, we could maximise this function using stochastic gradient descent in an iterative update of:

$$\alpha_{i,t+1} = \alpha_{i,t} + \eta \big(1 - \frac{1}{2} \sum_{j}^{N} (1 + \delta_{ij}) \alpha_j y_i y_j \mathbf{x}_i.\mathbf{x}_j)\big)$$

However, this ignores the constraints mentioned above. This first constraint, $C > \alpha_i > 0$, is simple to enforce, if the gradient update takes the value of $\alpha_i$ past either boundary, we simply clip that $\alpha_i$ to that boundary. Again, this is very similar to the SMO, where this concept of clipping is used. [10]

However, once we have performed a gradient update, the condition:

$$\sum_{i}^{N} \alpha_i y_i = 0$$

is no longer valid. Indeed, it has now changed by a value equal to the $\alpha_i$ update. If the constraint held before the update, then after the update, there is this residual, $R_i$. So that now after each iteration,

$$\sum_{j}^{N} \alpha_j y_j = R_i$$

Now the idea is to redistribute this residual amongst the remaining $\alpha_k$. So by what criteria do we redistribute this residual? Chunking algorithms and the SMO algorithm resort to various complicated heuristics, working sets and memory buffers to make the constraints

hold [10]. However, I propose a much lazier method. We can simply update all the alpha values in the following way:

$$\alpha_{k,t+1} = \alpha_{k,t} - \frac{R_i}{N\alpha_\mu} y_k \alpha_{k,t}$$

where $\alpha_\mu$ is the mean $\alpha$ value. And so substituting this back in:

$$\sum_j^N y_j \alpha_{j,t+1} = \sum_j^N (\alpha_{j,t} - \frac{R_i}{N\alpha_\mu} y_j \alpha_{j,t}) y_j$$

$$= R_i - \sum_j^N \frac{R_i}{N\alpha_\mu} \alpha_{j,t} y_j^2$$

$$= R_i - \frac{R_i}{N\alpha_\mu} \sum_j^N \alpha_{j,t}$$

$$= R_i - \frac{R_i \alpha_\mu}{\alpha_\mu} = 0$$

So the residual remains at zero!

If we initialize the $\alpha_i$ so that this residual is initially zero, a simple task, given that it is only binary classification, then the residual remains zero after every step, ensuring that the constraint holds, without updating the $\alpha_i$ values very much at each step.

So clearly, we are somewhat artificially updating the $\alpha$ values. However, you should note that once an $\alpha_i$ is 0, it cannot be changed by the residual being redistributed in subsequent iterations, since the update term for each $\alpha$ is linear in that value of $\alpha$. Hence, only $\alpha$ values which are near the $C$ constraint will change the most. So what about the $\alpha$ values that now break the KKT constraints by going above $C$? Well, on each iteration, the residual is either positive or negative, meaning on balance the $\alpha$ value should not change much. As the algorithm progresses too, the update terms become smaller and smaller, meaning less and less has to be redistributed.

**Modification 1**
A modification is not to redistribute the residual every iteration of the descent. We can choose to update it every 500 steps for example. This speeds the algorithm up because the residual calculation is linear in the data size, which becomes squared if we are iterating over the entire data-set.

**Modification 2**
Another modification to explore is perhaps the downfall of the whole algorithm, namely, whether we still get the same results when we do not redistribute the residual.

**Weaknesses**
A downside is that it can be numerically unstable. For example, a problem I encountered

was that if $N$ is too large, the redistribution applied to each $\alpha$ might be too small for the computer to register a difference (since the program only deals with 8-bit integers for example). Upping the precision of the calculations is often very costly.

## 3.2 Dual Results/Analysis

### 3.2.1 SMO

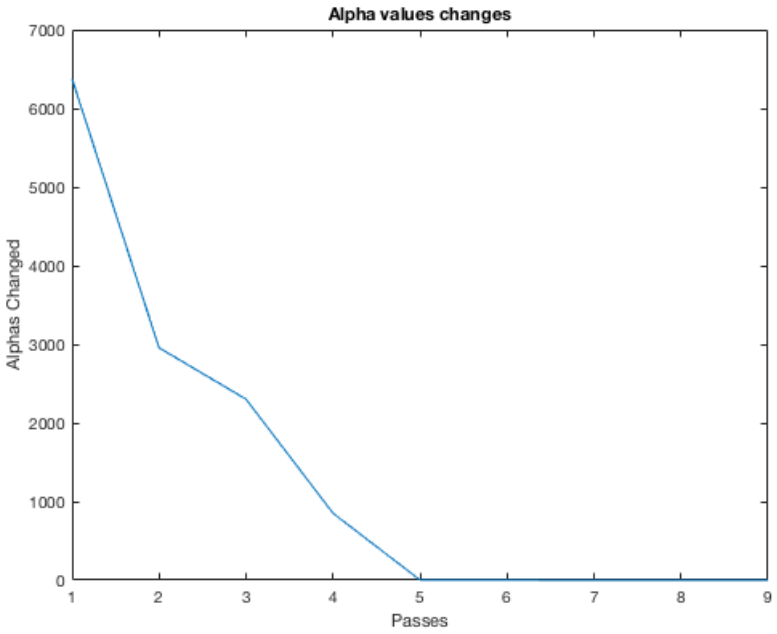


Figure 18: Alphas changed per iteration

As expected this simplified SMO algorithm achieves decent accuracy in a short amount of passes. It should be noted however, that this example is for a tolerance of $3^{-4}$ and anything more accurate, it took significantly longer to compute, so much so that I never actually saw it do it on my computer for $10^{-6}$, even waiting for > 10 minutes. It achieves a train accuracy of 94.5% and a test accuracy of 91.6% in 5 passes, in roughly 14000 $\alpha$ computations.

### 3.2.2 Residual Redistribution

Figure 19 shows the test accuracy and training accuracy per pass of the algorithm. This is admittedly a strange trajectory, occasionally actually dropping in training accuracy in a pass, although this does not happen very often, and it is only slight. It is a strange result that it works as well as it does. The gradient descent part of the algorithm ignores a constraint, the residual redistribution step also actually relies on the fact that it wont tip support vector already on the boundary of their constraints too far past it. However, the net result is seeming convergence to a decent accuracy. Indeed, it seems to outperform the full gradient descent algorithms in speed in the primal setting, although this comparison is perhaps not correct, given the complete difference in formulation. 92% test accuracy does not lie though.

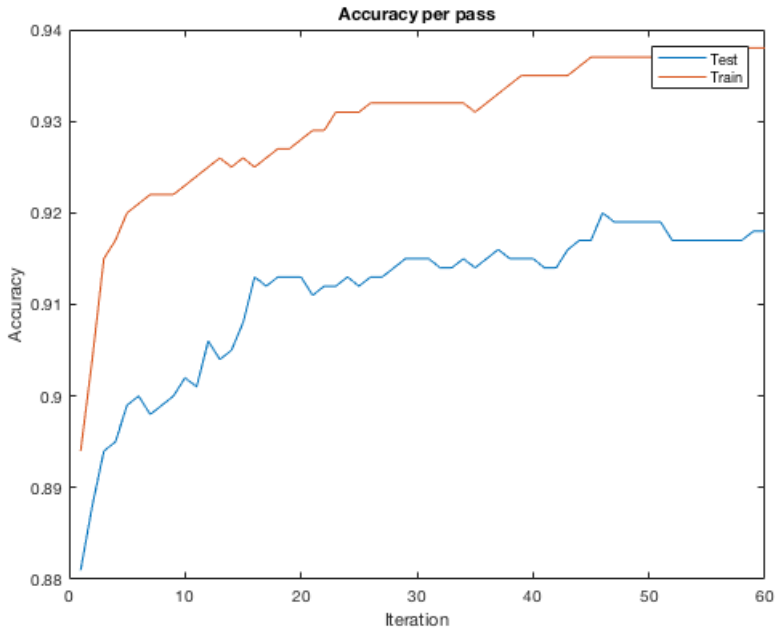


Figure 19: The corresponding test accuracies for the three categories

The drawbacks of this method are the need to compute the kernel matrix ahead of time. This is $O(dN^2)$ to compute and $O(N^2)$ to store. If the data is very high N, this is an incredibly expensive cost.

The actual algorithm I believe also takes $O(dN^2)$ to compute once the kernel is computed. Given this cost of running and the kernel precomputation, it is perhaps true to say that this algorithm is then best suited for the ground where d is high but N is low- perhaps it really is best for images!

### 3.2.3 Modification 1: Redistribute less often

The first modification is to redistribute the residual less often than every data point. The motivation behind this is to save cost. The redistribution is $O(N)$ within the loop iteration so a fractional increase can be achieved by redistributing less often

Figure 20 is a typical result for redistributing less often, it smoothes the trajectory but results in slower/convergence. Visually it looks similar to full algorithm, however, if you notice, it has settled at 91% a whole 1% accuracy below the full version.

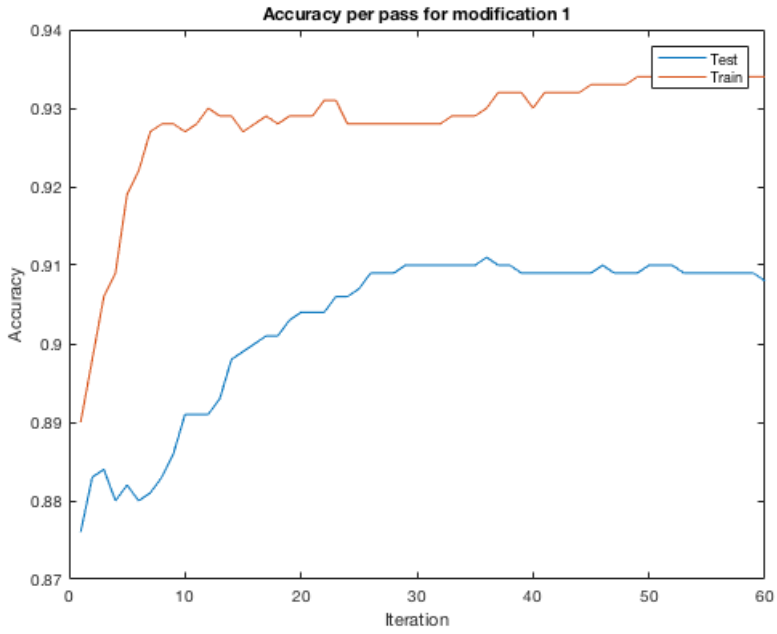


Figure 20: The corresponding test accuracies for the three categories

### 3.2.4 Modification 2: Do not redistribute

The second modification is perhaps against the point of the whole algorithm, but the result is very interesting. The algorithm converges to a similar point but it is much slower in getting there.

Figure 21 does not converge anywhere near as fast as the whole version.

To further the investigation into this algorithm, I would suggest trying different learning rates and attempting to see whether they affect the results. Furthermore, I would look into more complicated heuristics on how to update the alphas in the redistribution, because it is a lazy method at best.

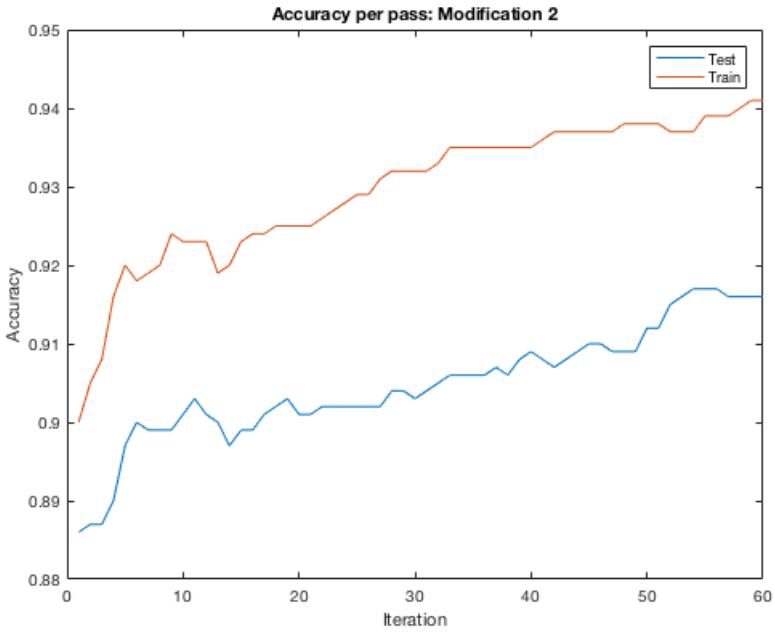


Figure 21: The corresponding test accuracies for the three categories

# 4 Conclusion

The study showed that for gradient descent, backtracking and line-search produced similar results on the MNIST data. Polak-Ribiere appears to be a much more robust conjugate gradient generation regime that Fletcher-Reeves which is prone to generating bad directions and therefore jumps in cost. Furthermore, it appears that the cost function for the SVM on MNIST data is quite flat because the CG methods were able to find costs a fair bit lower than the gradient descent methods.

A continuation to this project would be to look into approximating the hinge loss function with something else with a nicely differentiable gradient. That way quasi-newton methods could be explored.

Stochastic Gradient Descent itself seemed to show that it did not produce largely better results depending on the batch size. This perhaps suggests that, given the powerful parallelisation libraries available, the higher the batch size the better, up to memory considerations. However, the MNIST data is only 60,000 in absolute total, so this will be less and less a problem given the current rate of increase in computer memory.

Stochastic Gradient Descent with momentum did not seem to make an appreciable difference, although this does not rule it out, the project was narrow in its scope here.

The Residual Redistribution algorithm that I have come across appears to work well, albeit

with a somewhat jumpy trajectory. As a continuation to this project, it might be interesting to explore improvements to the algorithm, or testing it on different data-sets. Furthermore, its application to different kernels to the simple linear kernel would be an interesting study.

# REFERENCES


[1] K. P. Murphy, *"Machine Learning: A Probabilistic Perspective"*, MIT, 2013.

[2] A. Zisserman, *"Lecture 2: The SVM Classifier"*, Oxford University.

[3] J Kujala and T. Aho and T. Elomaa, *"A Walk from 2-Norm SVM to 1-Norm SVM"*, Tampere University of Technology.

[4] Koshiba, Yoshiaki and Abe Shigeo, *"Comparison of L1 and L2 support vector machines"*, Neural Networks, 2003. Proceedings of the International Joint Conference on,3:2054-2059.

[5] Leon Bottou, *"Large-Scale Machine Learning with Stochastic Gradient Descent"*; NEC Labs America, Princeton NJ 08542, USA.

[6] C. Guestrin, *"SVMs, Duality and the Kernel Trick"*, Carnegie Mellon University.

[7] R. Fletcher, *"Practical Methods of Optimisation"*; Wiley, Chichester, 1987.

[8] A. N. Tikhonov and V. Y. Arsenin, *"Solution of Ill-posed Problems"*. Washington: Winston Sons, 1977.

[9] B. SchÃűlkopf and R. Herbrich and A. J. Smola, *"A Generalized Representer Theorem"*. Computational Learning Theory. Lecture Notes in Computer Science, 2001.

[10] J. Platt, *"Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines"*; 1998.

[11] M. Betcke and . Rullan, *"Numerical Optimisation: Conjugate gradient methods*, 2017.

[12] M. R. Hestenes and E. Stiefel, *"Methods of Conjugate Gradients for Solving Linear Systems"*, Journal of Research of the National Bureau of Standards, 1952.

[13] E. Polak and G. Ribiere, *"Note sur la convergence de directions conjuguee"*, Rev. Francaise Informat Recherche Operationelle, 1969.

[14] M. R. Hestenes and E. Stiefel, *"Methods of conjugate gradients for solving linear systems"*, J. Research Nat. Bur. Standards 49, 1953.

[15] Y. H. Dai and Y. Yuan, *"A nonlinear conjugate gradient method with a strong global convergence property"*, SIAM J. Optim. 10, 1999.