Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Engineering

Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science

Incremental Learning with Support Vector Machines on Embedded Platforms

by
Shankar Kumar Jeyakumar

Supervisors:
Dr. Hassan Ghasemzadeh Mohammadi
Prof. Dr. Marco Platzner

Paderborn,
August 1, 2019

Abstract

Support Vector Machines (SVMs) are a class of fast and light *Machine Learning* (ML) algorithms known for their simplicity of mathematical formulation and ease of implementation. SVMs find a wide variety of applications ranging from weather forecasting [Zha+17], stock market predictions [HS06] in natural sciences to pedestrian detection [NDJ17], road sign recognition [Kir+08] and vehicle tracking [CCC13] in autonomous driving to name a few.

With systems such as autonomous driving as examples comprising of a large number of embedded devices that are interconnected and communicating with each other, the possibilities of introducing intelligence to these systems via ML is sought out for on the fly prediction and decision making, without having to outsource confidential data to third parties which might lead to concerns regarding data security and privacy. In other words, in house ML is a need.

In the era of *big data*, the the amount of information that such algorithms have to process has increased by many folds and with it comes a need for resource hungry hardware. Given the resource budget of an embedded system, such algorithms have to be simplified in a way that lets us apply them to such systems with the same performance, without the additional cost of infrastructure that comes with deploying such resource and computation intensive algorithms on traditional platforms such as CPUs and GPUs. In lieu of this, approximating the computations involved in the algorithm is one such way of being able to squeeze in the algorithm into an embedded system.

Traditionally, accelerating ML often involves a GPU to speed up only the inference phase of an ML algorithm. With the need to deploy these algorithms on embedded systems, GPUs might fail to fit the resource and power budget and provide the versatility of reconfigurable systems such as an FPGAs which would let us accelerate the training too by exploiting aspects such as data level and task level parallelism present in the algorithm.

This thesis aims to investigate the possibility of implementing the SVM algorithm on an embedded SoC with changes to the algorithm structure to accommodate learning from large volumes of data in an incremental manner with approximations to the computations. The finalized software-only approximated incremental algorithm achieved an accuracy difference of 0.63% (reduction) when compared to the base non-incremental algorithm and a difference of and 1.2% (reduction) when compared to a state of the art implementation of the SVM classifier. A speed up of nearly 4.0 (improvement) was achieved as a result of accelerating the training and inference phases of the classifier, validating the possibility of deploying approximated incremental ML algorithms to reconfigurable embedded devices.

Contents

1

Introduction

The concept of *Machine Learning* (ML) or *Artificial Intelligence* (AI) can be attributed to the earliest attempts at recreating an individual neuron of a human brain as an electric circuit, with which *neural networks* were born. Next came the *Turing test* [PSCA00] whose purpose was for a machine to convince a human that it is in fact another human and not a machine. In simpler terms, ML is a stepping stone in man's endeavor at achieving AI. Arthur Samuel was the first to actually coin the term while trying to have a game on his computer play itself [Sam59]. He goes on to lay the foundations of pattern recognition, the statistical backgrounds and evaluation principles that define the modern day field of ML, which became mainstream only in the late 1990s when researchers started to see possible applications of such algorithms to problems in nature[1].

ML today involves recognizing structure and patterns that exist coherently within presented data, being able to memorize this structure as a model and finally be able to infer this same structure when presented with new unseen data for the purpose of identifying or recreating the source. This is comparable to how a human is able to learn from experience and use this knowledge to handle similar scenarios. Application of ML is a vast growing field broadly classified into two - algorithms that require human intervention to facilitate learning, known as *supervised learning* and having the algorithm learn for itself, known as *unsupervised learning*. We will be focusing on *classification* problems that fall under supervised learning for this thesis. Classification can be defined simply as being able to assign a category or *label* to a given set of data to differentiate it from others. These problems can arise in the form of simple tasks of being able to tell apart a cat from a dog to complex decisions such as discerning the possibility of a genome sequence being prone to a particular disease [XJ19].

Support Vector Machines (SVM) specifically are among the most commonly used algorithms for classification analyses. [JR14] explains how SVM can be utilized in cancer treatment research to validate if a cancer mutation is the driver mutation. [Cao+] explains how SVMs are used with commendable classification capability to present a technique for fault diagnosis in induction motors. [Lin+18] illustrates how SVMs can help in object recognition for autonomous driving systems. These are just some of the use cases of this light but powerful algorithm. Originally envisioned as a simple linear *binary classifier*, they can be used for more complex non-linear classification, with subtle changes to its mathematical formulation which converts them from an initial non *kernalized* form to a class of algorithms called *kernel machines*[2] which are algorithms that deal with complex high dimensional data.

[1] https://www.nature.com/natmachintell/

[2] algorithms that employ kernel functions which transform lower dimensional data to higher dimensions

We have seen that ML finds applications in almost every scenario, and with it comes the problem for industries to either outsource the data to third parties to performing the learning and analysis on their data or investing in expensive hardware capable of handling such powerful algorithms and perform the learning in house to circumvent the issues of data privacy and confidentiality that comes with this outsourcing. In the era of *Industry 4.0*, and the *Internet of Things* (IoT), it opens up the possibility of embedded devices based on these principles to themselves have intelligence in them [Hui+04] and be able to communicate with each other and exchange information to perform various self aware tasks such as automatically monitoring the health of a machine [Zha+18] in an industrial environment to be able to predict future faults and prepare in advance for them. The possibilities are endless. This is an alternative to having expensive in house hardware capable of this ML. But with this comes the bottlenecks in memory, energy utilization and computational capabilities of such devices. An embedded system is usually limited in such resources and given the huge amounts of data that traditional ML algorithms deal with, we look at alternative ways to modify the algorithm to fit the resource budget of the system rather than just invest in better hardware.

It is a general notion that ML algorithms are usually resource and computation hungry and would require sophisticated and dedicated hardware to handle complex operations involved. This also would require suitable memory reserves to handle the big data flowing through it. But in order to enable ML for embedded devices we would have to make do with a restricted resource and computation budget. Incremental Learning is a promising technique commonly used to allow algorithms to deal with large amounts of such *big data*. Incremental learning techniques originally stemmed from batch processing which involved breaking the large data sets into manageable batches where each batch could fit on the embedded system for computation at any given point of time. Incremental learning today involves being able to learn from even single data samples that might occur at the stream of information to a ML model. The model is then successively updated with learning patterns in the incoming stream of data.

The field of approximate computing is a research into how computations can be replaced by their approximated versions which would suffice their original purpose. These might include hardware based approximations or software based algorithmic approximations. The goal of software based algorithmic approximations could be to replace precise computations with approximate ones that reach the actual precision given a tolerance level for this. Approximate techniques could also include studying statistical relations between computations and determining which of these computations might actually be required and which can be neglected with their overall impact on the performance of the algorithm. Examples of hardware based approximations on the other hand could involve half precision mathematical operations, using different data types etc in CPU based systems and using approximate hardware components such as approximate adders, multipliers and other such logic circuits when it comes to reconfigurable devices like FPGAs.

FPGAs are systems that lie in between those of pure ASICs and CPU based systems where ASICs being a spectrum limit of purely application based custom hardware systems and CPU based systems being the far end of the spectrum of general purpose computing. FPGAs are known for their massive parallel computation properties and parallel processing in general. With FPGAs, it is possible to create either an ASIC style system or a CPU style system based on the application requirement. Hence they are known by the name reconfigurable embedded systems. This reconfigurability can be leveraged to design custom approximate functions or half precision operators which allow us to implement approximated algorithms. With the advent of devices such as the PYNQ, which is a combination of a CPU and FPGA based system, it allows a Hardware-Software Co-design approach to creating custom ML algorithms that might include computationally intensive operations which can be offloaded to the FPGA, leveraging its acceleration capabilities. This way computations that perform well in a sequential manner can be run on the CPU and those that benefit from parallelism can be run on the FPGA. This is the reason why FPGAs are known as Field Programmable Gate Arrays which indicates reconfigurability on the fly.

Graphic Processing Units (GPU) are well known when it comes to acceleration of ML algorithms. Even though GPUs were initially only designed for faster image processing and gaming, they were quickly utilized for acceleration of ML algorithms. There are widespread cases were GPU based acceleration outperforms FPGA based acceleration. But GPUs are fixed architecture based systems with predefined widths for their computational units and unless the data and the ML algorithm can make use of the entire operational unit and its data path, the GPU might end up being under utilized. This is not the case with an FPGA where each computational unit that is designed is custom built to the width specification and hence exact resource allocation is possible. Another significant trade off between GPUs and FPGAs is the power consumption and energy efficiency where FPGAs have an edge over GPUs. There is also an aspect of the communication latency between GPUs complex hierarchy including the memory and CPU,which is nearly absent in FPGAs due to the custom routing abilities that the design tools offer. This flexible architecture lets us route the data parts as per our algorithm's computational needs. With all these advantages, there is a widespread increase in FPGAs being introduced into CPU based systems even by companies such as Intel specifically tailored to accelerating ML algorithms.

1.1 Motivation

With the need for in house ML in scenarios where data is not wished to be outsourced to third parties due to privacy and security concerns and a low cost solution to introducing ML to interconnected embedded devices, we look at ways to integrate ML and embedded systems. This class of embedded systems can comprise of devices such as sensors, Iot based devices which due to their widespread use and inter connectivity produce vast amounts of information. With such vast information, comes the possibility of studying it and optimizing processes to enhance productivity. To facilitate learning from *big data*, we modify the ML algorithm to incrementally learn from this data. Studying the statistical properties of the ML algorithm gives us an insight into what the computations result in and hence allows us to determine application critical computations. Using hybrid reconfigurable devices, these critical computations can be accelerated compensating for the lack of computational ability of such embedded systems. Finally knowing which of these computations can be dropped gives us the possibility of speeding up the algorithms on these embedded platforms.

1.2 Thesis Objective

This thesis aims to study the SVM algorithm, investigate various *Incremental Learning* techniques for the SVM that will let the algorithm to learn incrementally from larger amounts data, facilitating it to run on an embedded platform constrained in computation and memory resources. Furthermore, approximation techniques are researched to reduce the number of samples that the incremental algorithm actually has to go through to reduce the number of computations and hence the time and complexity of the learning process. Given that most computations involved in such ML algorithms tend to be iterative in nature, we look at techniques that let us take advantage of this in order to accelerate the learning process using modern hardware such as FPGAs.

1.3 Thesis Outline

- **Chapter 2** is an insight into what Support Vector Machines actually perform from a mathematical perspective.
- **Chapter 3** discusses the solution to this mathematical problem in Chapter 2 such that it can be converted to code.
- **Chapter 4** introduces us to the concept of Incremental Learning and the current literature and techniques associated with it. An overview at previous and current attempts at accelerating Machine Learning algorithms using different architectures is presented.
- **Chapter 5** discusses methods that we consider in order to evaluate our implementations with respect to a visual and statistical perspective.
- **Chapter 6** discusses four different algorithm implementations and their corresponding evaluation results, the first being the non Incremental Implementation of the SVM, followed by Incremental and Approximated versions.
- **Chapter 7** discusses the Hardware Implementation architecture and the corresponding results of the acceleration.
- **Chapter 8** consolidates evaluations from all algorithms and implementations into a combined comparison.
- **Chapter 9** summarizes the entire thesis, provides concluding remarks with achievements and possible improvements.

2

Support Vector Machines

The original SVM formulation was introduced in [CV95a]. It was initially envisioned for tasks such as handwriting recognition. Simply put, a SVM model creates a *hyperplane* or a *decision boundary* that can best divide a dataset into two categories or *classes*. The SVM has to define a margin alongside this decision boundary which is maximized such that the two classes are absolutely distinguishable. On encountering any new samples, the model is then able to assign them to one of the two classes, usually denoted as the $+ve$ and $-ve$ classes respectively. The technique gets its name from the fact that the points from the dataset that lie close to this hyperplane and help in forming the margins are called the *support vectors*. Once the model has been defined, which here means we have obtained the support vectors, the rest of the samples used for this learning can be discarded. In other words, the support vectors alone suffice to define our classifier.

As a start we will address only linearly separable points and how the SVM handles them. A particularly interesting technique called the *kernel trick*, which integrates smoothly into the SVM's mathematical formulation, allows us to also apply SVMs to non linearly-separable and overlapping data, discussed in later chapters. A visual representation of the idea of an SVM model can be seen in figure 2.1. The green points belong to the positive class and the yellow to the negative class. The solid line is the decision boundary that separates the two classes and the dashed lines represent the boundary that maximizes this separation.

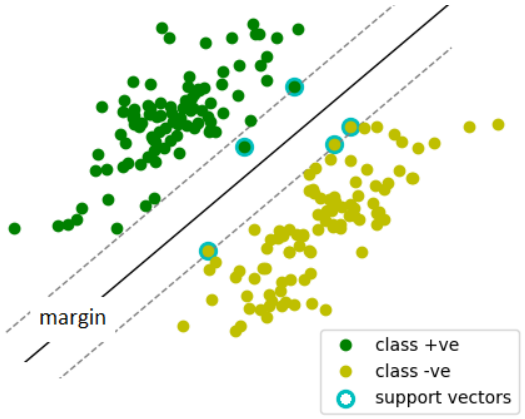


Figure 2.1: A binary SVM model to visualize the decision boundary, margins and support vectors

2.1 Mathematical Formulation

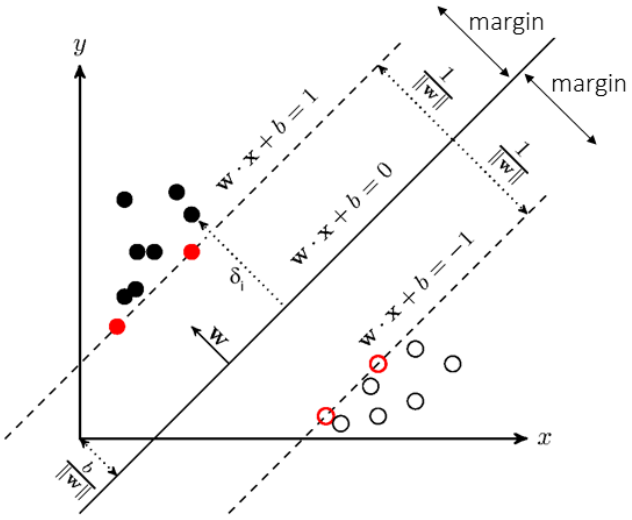


Figure 2.2: A binary SVM model with equations for the decision boundary and the margins forming the basis of the SVMs mathematical formulation. *Source*: [CV95a]

In simple terms, the SVM tries to generate a hyperplane that can separate a set of positive samples from negative samples with the *maximum* margin. Figure 2.2 represents the SVM model with the margins and the decision boundary marked. The black points represent the positive samples, white points represent the negative samples and the red points indicate the support vectors. Dotted lines represent the margins and the solid line represents the decision boundary or hyperplane. The margin can be defined as the distance of this hyperplane from the nearest positive and negative example. The SVM as linear discriminator can be stated as

$$\mathbf{w}^T\mathbf{x} + b \tag{2.1}$$

where $\mathbf{w}$ is the normal vector to the hyperplane, $\mathbf{x}$ is a sample input vector $\in \mathbb{R}$ and b is the *bias*. The upper margin represents the margin for the positive class by the equation $\mathbf{w}^T\mathbf{x} + b = 1$ and the lower for the negative class by the equation $\mathbf{w}^T\mathbf{x} + b$ = -1, which are both in the linear case, equations of a line. $\mathbf{w}^T\mathbf{x} + b = 0$ represents the separating decision boundary. The margin m can be defined as

$$m = \frac{1}{|\mathbf{w}|^2} \tag{2.2}$$

The goal is to *maximize* this margin $\frac{1}{|\mathbf{w}|^2}$, which can be rewritten as minimizing $\frac{1}{2}|\mathbf{w}|^2$ which can be modelled as a *primal* optimization[1] problem $\mathcal{P}$ to *minimize* $\mathbf{w}$ as

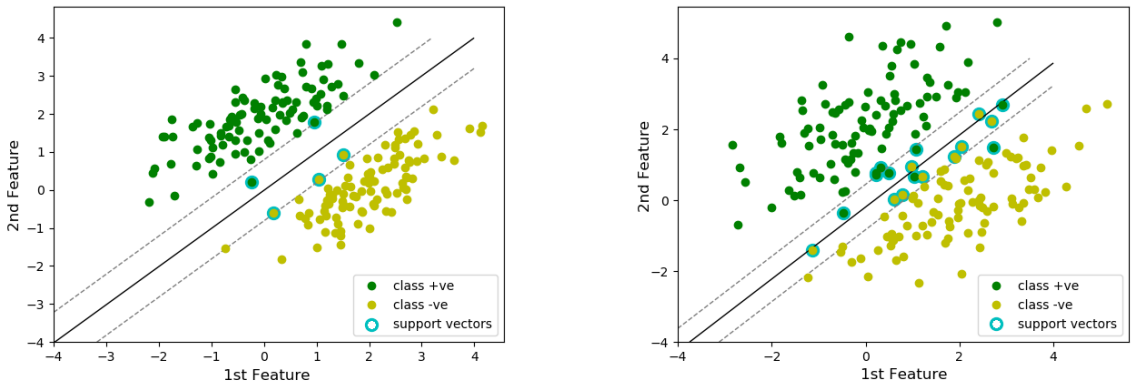
$$\begin{aligned} min\ \mathcal{P}(\mathbf{w}, b) &= \frac{1}{2}\mathbf{w}^2 \\ subject\ to\ \forall_i\ y_i(\mathbf{w}^T\mathbf{x}_i + b) &\geq 1 \end{aligned} \tag{2.3}$$

A sample $\mathbf{x}$ is assigned to a $+ve$ class or $-ve$ class from the sign of the output of the *linear discriminant function* $y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$.

[1] minimizing and maximizing a function until an optimal solution is reached

2.2 Hard and Soft Margins

So far we have the mathematical formulation for the SVM in a sample space which is linearly separable. In reality, samples are not so easily separable and tend to overlap or lie in situations where a linear separation is not possible. This can also arise when our samples can contain unwanted noise.



(a) Hard Margin SVM

(b) Soft Margin SVM

Figure 2.3: Represents the two kinds of SVM models with respect to separability of the samples where figure 2.3a represents the *hard margin* model and figure 2.3b represents the *soft margin* model.

In figure 2.3a notice that there are no samples that lie within the bounds of the positive and negative margins. This is the *hard margin* case where no samples are allowed to violate the boundary conditions defined in equation 2.7. In figure 2.3b some of the samples do violate the boundary conditions. This is the *soft margin* case. The idea of a soft margin in SVMs was first proposed in [CV95b] where we allow some samples to violate these boundary conditions. These violations are represented by a vector of *slack* variables $\boldsymbol{\xi} = (\xi_i ... \xi_n)$. Additionally, $\mathcal{C}$ represents the trade off between large and small margin violations.

Modifying the primal form in equation 2.3 to include the additional slack $\boldsymbol{\xi}$ and $\mathcal{C}$ parameters results in equation 2.4 where the only noticeable difference is the change in the upper bounds of α to $\mathcal{C}$.

$$\begin{aligned} \min_{\mathbf{w},b,\boldsymbol{\xi}} \mathcal{P}(\mathbf{w}, b, \boldsymbol{\xi}) &= \frac{1}{2}\mathbf{w}^2 + \mathcal{C}\sum_{i=1}^{n}\xi_i \\ \textit{subject to} \begin{cases} \forall_i \; y_i(\mathbf{w}^T\mathbf{x}_i) + b) \geq 1 - \xi_i \\ \forall_i \; \xi_i \geq 0 \end{cases} \end{aligned} \tag{2.4}$$

A large value of $\mathcal{C}$ leads to a narrow margin, which in turn decreases the number of misclassified samples and conversely a small value of $\mathcal{C}$ leads to a wide margin [BH+08], allowing more samples to be misclassified.

2.3 Construction of the Dual Problem

Since the constrains from equation 2.4 for this optimization problem are quite complex [VL63], we use the *lagrangian duality*[2] method [Ber76] to solve this problem. We rewrite equation 2.4 using $+ve$ lagrangian multipliers $\alpha_i \geq 0$ for the i^{th} sample as

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}) = \frac{1}{2}\mathbf{w}^2 + \mathcal{C}\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i(y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1 + \xi_i) \tag{2.5}$$

Differentiating equation 2.5 with respect to $\mathbf{w}$ to get the gradient,

$$\begin{aligned} \frac{d\mathcal{L}}{d\mathbf{w}} &= \mathbf{w} - \sum_{i=1}^{n}\alpha_i y_i x_i = 0 \\ \mathbf{w} &= \sum_{i=1}^{n}\alpha_i y_i x_i \end{aligned} \tag{2.6}$$

Differentiating equation 2.5 with respect to b to get the gradient,

$$\frac{d\mathcal{L}}{db} = -\sum_{i=1}^{n}\alpha_i y_i = 0 \tag{2.7}$$

Substituting equations 2.6 and 2.7 in 2.5 we get

$$\begin{aligned} max\ \mathcal{D}(\boldsymbol{\alpha}) &= \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{n} y_i\alpha_i y_j\alpha_j \mathbf{x}_i^T\mathbf{x}_j \\ subject\ to &\begin{cases} \forall_i\ 0 \leq \alpha_i \leq \mathcal{C} \\ \sum_i y_i\alpha_i = 0 \end{cases} \end{aligned} \tag{2.8}$$

Which is the dual formulation $\mathcal{D}(\boldsymbol{\alpha})$ with simpler constraints of equation 2.3 where n is the number of samples used for training, $\boldsymbol{\alpha}$ is the vector of lagrangian multipliers, $\forall_i\ \alpha_i \geq 0$ is the inequality constraint and $\sum_i y_i\alpha_i = 0$ is the linear equality constraint. The optimum orientation $\mathbf{w}^*$ of the hyperplane can be obtained by finding optimal αs^* that solve the dual problem in equation 2.7 given as

$$\mathbf{w}^* = \sum_i \alpha_i^* y_i \mathbf{x}_i \tag{2.9}$$

The dual formulation does not solve for the bias, attributed to [SK+05]. The optimal bias b^* can be solved as a straight forward one dimensional problem [BL07]. Finally, the linear discriminant function takes the form

$$\hat{y} = \mathbf{w}^{*T}\mathbf{x} + b^* = \sum_{i=1}^{n} y_i\alpha_i\mathbf{x}_i^T\mathbf{x} + b^* \tag{2.10}$$

whose sign determines whether a sample is classified to the $+ve$ or $-ve$ class.

[2] a method of solving the primal optimization problem in terms of a differently formulated dual problem

2.4 The Karush-Kuhn-Tucker (KKT) Conditions

The KKT conditions [Kar39] help determine if an optimal point is reached as part of the optimization routine. Recalling the dual formulation from equation 2.19, in order to solve for the optimum $\mathbf{w}$ and b, we consider the gradients as

- Stationarity: where our gradients will be zero for both the primal and dual variables

$$\frac{d\mathcal{L}}{d\mathbf{w}} = \mathbf{w} - \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i = 0 \tag{2.11}$$

- Primal feasibility:

$$\forall i \quad y_i(\mathbf{w}.\mathbf{x}_i + b) - 1 \geq 0 \tag{2.12}$$

- Dual feasibility:

$$\forall i \quad \alpha_i \geq 0 \tag{2.13}$$

- Complimentary slackness:

$$\forall i \quad \alpha_i(y_i(\mathbf{w})\mathbf{x}_i + b) - 1) = 0 \tag{2.14}$$

From equation 2.14 either $\alpha_i = 0$ or $y_i(\mathbf{w}.\mathbf{x}_i + b) - 1 = 0$, which leads to the conclusion that samples with $+ve$ lagrangian multipliers are *support vectors*. Hence solving the KKT conditions leads to solving the SVM problem, with a feasible solution being the support vectors themselves.

For any two pair of older lagrange multipliers α_a^{old} and α_b^{old} which was a feasible solution, to proceed with finding a newer solution α_a^{new} and α_b^{new}, according to the linearity constraint from equation 2.8

$$y_a\alpha_a^{old} + y_b\alpha_b^{old} = y_a\alpha_a^{new} + y_b\alpha_a^{new} \tag{2.15}$$

restricting the optimization to lie on a line, as seen when equation 2.15 is rewritten as

$$\begin{aligned} y_a\alpha_a^{old} &= -y_b\alpha_b^{old} \\ y_a &= -\frac{\alpha_b^{old}}{\alpha_a^{old}} y_b \end{aligned} \tag{2.16}$$

which is basically an equation of a line with a slope $m = -\frac{\alpha_b^{old}}{\alpha_a^{old}}$. We can either proceed in the positive direction or the negative direction of the line as shown in figure 2.4.

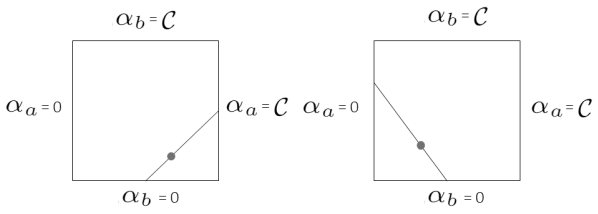


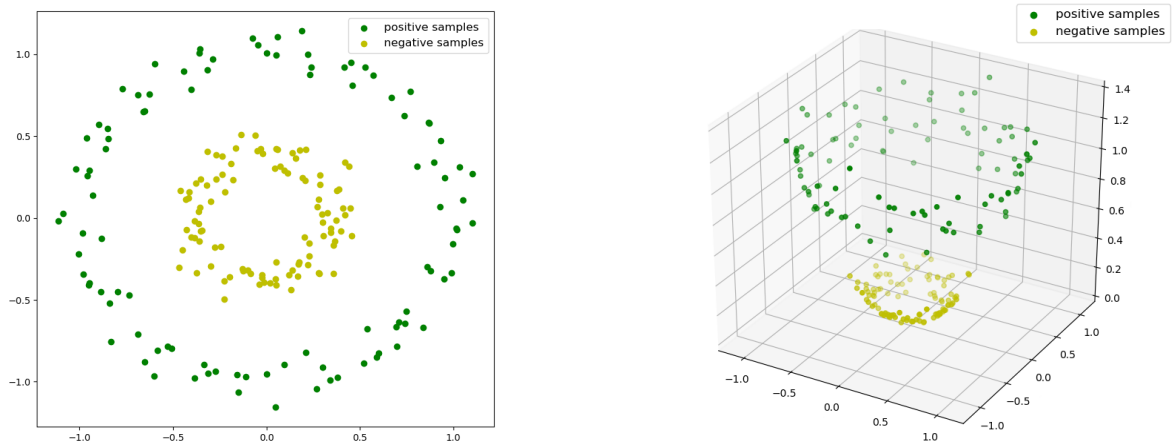
Figure 2.4: Box constraints for the optimization of αs [Pla99]

which changes the original constraints to a set of *box* constraints.

$$0 \leq \alpha_i \leq \mathcal{C} \tag{2.17}$$

2.5 Kernel Trick

The previous sections described the SVM formulation for data that is linearly separable, even if they are overlapping, but how does one deal with data that is not linearly separable? Figure 2.5a represents one such sample space. It is postulated in [Bur98] that a function ϕ exists that is able to transform samples from an original inseparable space to a higher dimensional separable space known as the *feature space*, shown in figure 2.5b.



(a) Samples not linearly separable

(b) Samples raised to a higher dimension

Figure 2.5: Represents an example of the before and after visualizations of samples subjected to the kernel function where figure 2.5a is an example of inseparable samples and figure 2.5b is the resultant of the kernel function.

Modifying the dual form equation 2.7 to include this ϕ function results in

$$max\ \mathcal{D}(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y_i \alpha_i y_j \alpha_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$$
$$subject\ to \begin{cases} \forall_i\ 0 \leq \alpha_i \leq \mathcal{C} \\ \sum_i y_i \alpha_i = 0 \end{cases} \tag{2.18}$$

But this ϕ function is unknown and can often only be determined by a brute force approach, which can be computationally intensive. In equation 2.18, we can observe that the operation $\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$ between the input samples is a *dot product*, which is an *inner product* in a *feature space*[3]. Following Mercer's theorem [MNY06], which states that we can either explicitly transform the samples to a higher dimension using a ϕ function and then perform the dot product required in equation 2.18, or we can replace it with a known *kernel function* without bothering about what the ϕ function could be. [RW06] further supported the idea stating that if in an algorithm, computations on the samples are only in terms of inner products, these samples can be transformed into a higher dimensional feature space by replacing these inner products with a kernel function. The mathematical formulation of the SVM is such that it easily integrates the kernel function into it. Upon replacing the dot product $\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$ with the kernel function along with the slack and margin violation parameters $\mathcal{C}$ in equation 2.7 we arrive at equation 2.19.

[3] the space where we consider each of the features of our input samples as a dimension

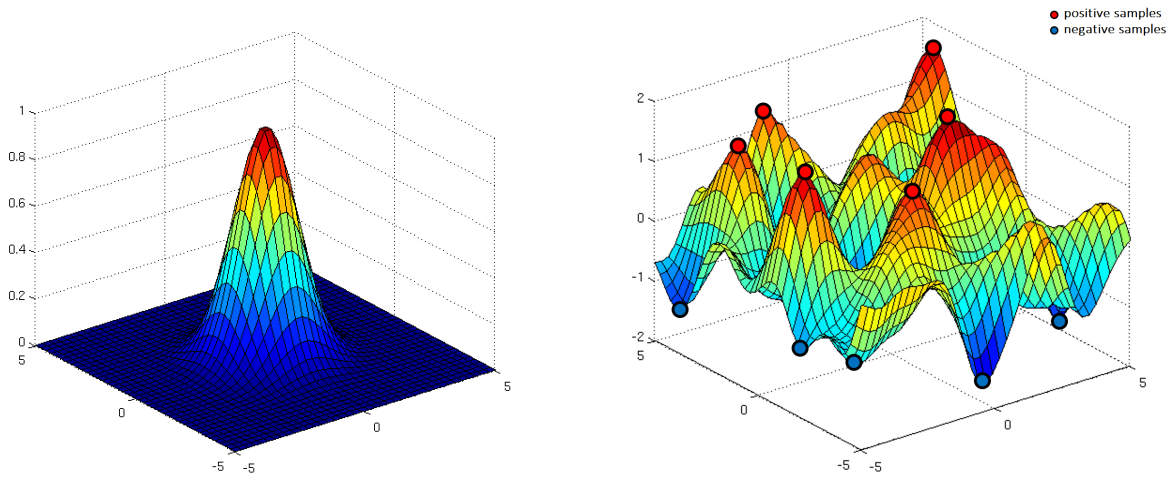
$$max\ \mathcal{D}(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y_i \alpha_i y_j \alpha_j \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j)$$
$$subject\ to \begin{cases} \forall_i\ 0 \leq \alpha_i \leq \mathcal{C} \\ \sum_i y_i \alpha_i = 0 \end{cases} \tag{2.19}$$

where $\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j)$ is the kernel function.

The most commonly used kernel function is the *squared exponential covariance* function, also known as the *Radial Basis Function* (RBF) kernel that takes the mathematical form of the equation 2.20.

$$\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 exp(-\frac{1}{2l^2}(\mathbf{x}_i - \mathbf{x}_j)^T(\mathbf{x}_i - \mathbf{x}_j)) \tag{2.20}$$

The parameters σ and l are known as hyperparameters[4]. They denote *variance* and *lengthscale* respectively. The variance determines the similarity between a pair of points and the lengthscale controls the smoothness of the curve or the height at which the points are separated from each other.



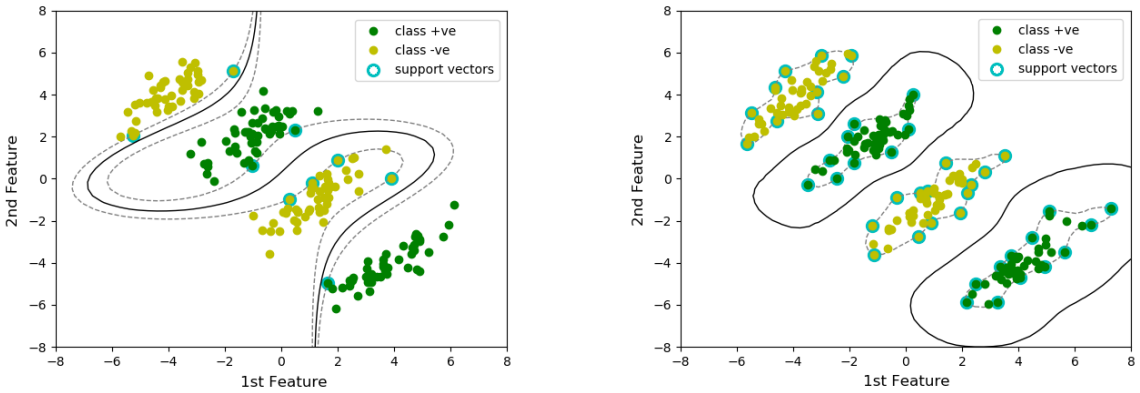
(a) The Radial Basis Function

(b) Contour of transformed points

Figure 2.6: Represents the RBF kernel in 3 dimensions [Duv14] where figure 2.6a shows the contours of the RBF kernel and figure 2.6b shows a pair of inseparable points mapped to separate dimensions by the kernel. *Source*:[Duv14]

Figure 2.6a represent a zero mean RBF kernel in 3 dimensions. Figure 2.6b represents the contour of the distribution of the points after they have been subjected to the RBF kernel transformation. The red and blue points were originally inseparable and after the kernel transformation, the points are mapped to higher dimensions which enables them to be separable in this higher dimensional space.

[4]a parameter whose value is set prior to the computations.



(a) The Polynomial Kernel

(b) Radial Basis Function Kernel

Figure 2.7: Represents example results for two commonly used kernels for SVMs where figure 2.7a is a result of the polynomial kernel and figure 2.7b is a result of the RBF kernel

Figure 2.7a and 2.7b represent the resultant decision boundaries and margins of an SVM subjected to training samples that were not separable linearly in the input space. The plots are a result of the points being raised to a higher dimension by a kernel and then the SVM being applied in the higher dimensional space and finally being lowered back to the original dimension [SS01].

In general when applying the kernel to ML algorithms, as the number of samples increases, kernel computations are usually time consuming and complex and increases linearly. SVMs in particular have an advantage over other kernalized algorithms since for all other samples other than the support vectors obtained during the training, the αs are zero and hence no computation takes place. This is beneficial for larger datasets.

3

Solving the SVM Quadratic Optimization Problem

Given that we have modelled the SVM as a quadratic programming problem, we look at ways to actually solve this mathematical problem by various commonly used techniques. Some common packages used for this are MINOS and LOQO [MS78] which have full fledged ways of solving the dual problem. In light of this thesis being research into incremental learning, we look at methods that involve decomposition of the optimization problem into manageable sub problems.

3.1 Decomposition methods

Generally when considering a quadratic programming method for optimization that considers all training samples at once, worst case run time complexity is of the order of $O(N^3)$, where N is the number of variables + number of constraints. This might be unfeasible when it comes to big data, given the huge volume of samples it encompasses. This technique was introduced first in [OFG97]. In an effort to avoid computing the kernel function for all samples which tends to be expensive as discussed in section 2.18, better ways of breaking down the entire sample space into smaller subsets were researched. The subsets were then itertatively solved until the entire sample space had been optimized. Formally general process for this technique involves choosing a subset $\mathcal{B}$ of αs and ignoring the rest. Modifying the dual form equation form 2.20 to include this constraint results in

$$\begin{aligned} \underset{\alpha^{`}}{max}\ \mathcal{D}(\alpha^{`}) &= \sum_{i=1}^{n} \alpha_i^{`} - \frac{1}{2} \sum_{i,j=1}^{n} y_i \alpha_i^{`} y_j \alpha_j^{`} \mathcal{K}(x_i, x_j) \\ subject\ to & \begin{cases} \forall i \notin \mathcal{B} \quad \alpha_i^{`} = \alpha_i, \\ \forall i \in \mathcal{B} \quad 0 \leq \alpha_i^{`} \leq \mathcal{C}, \\ \sum_i y_i \alpha_i^{`} = 0 \end{cases} \end{aligned} \tag{3.1}$$

The constraints in equation 3.1 now reflect the $\alpha^{`}s$ which are ones that will not be optimized on.

3.1.1 Iterative Chunking and Direction Search

[VL63] suggests that instead of solving for all the multipliers at once, a subset or *chunks* of these multipliers are taken, specifically those that violate the margin conditions [BL07], called the *working set* and optimized upon. This *iterative chunking* converts our large optimization problem into smaller mini optimization problems. After having optimized for the current working set, we need to decide on a direction in which we proceed to pick the next chunk of samples to optimize upon. This *direction search* is performed by choosing an initial $\boldsymbol{\alpha}$ and moving in a direction $u = u_1 ... u_n$ of non zero multipliers in an assumption that multipliers along this direction do not violate the constraints. Now our optimization problem converts our dual problem to maximizing the function over not the entire set of αs but over a restricted set which can be rewritten as

$$\underset{\lambda \in \mathbf{s}}{arg\ max}\ \mathcal{D}(\alpha + \lambda u) \tag{3.2}$$

where λ is a multiplier ≥ 0 in a set $\mathbf{s}$ of all non zero αs in a possible direction u .The shaded area in figure 3.1 represents this promising set and its maximum value by λ^{max}.

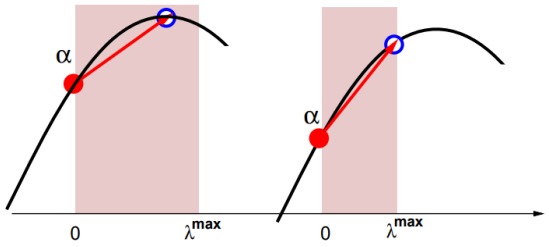


Figure 3.1: Direction search for selecting the working set, *source*: [BL07]

The viable λ within this region is calculated by

$$\lambda \ = \ max\Big(0,\ min\ \Big(\lambda^{max}, \frac{\mathbf{g}^T u}{u^T \mathbf{H} u}\Big)\Big) \tag{3.3}$$

whose derivation is discussed in [BL07]. $\mathbf{g}$ and $\mathbf{H}$ are the gradient and Hessian of the dual function.

3.1.2 Sequential Minimal Optimization

Sequential Minimal Optimization (SMO) technique was proposed in [Pla99] which is a very widely used optimization package for SVMs and even today found in state of the art ML libraries such as *Scikit-Learn*[1]. It was found to be computationally and resource friendly and less complex. The algorithm follows from the principles of the decomposition method and iterative chunking to optimize for the working set with the smallest amount of αs - two, with all other αs fixed. This greatly simplifies our optimization problem and the direction search to a one dimensional one since only two αs are involved. Modifying the dual form equation to reflect this results in

$$\mathcal{D}(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1} y_i \alpha_i y_j \alpha_j \mathcal{K}(x_i, x_j) \tag{3.4}$$

Notice that the indices for the αs are just $i, j = 1$ instead of $i,j = 1$ to n.

[1] https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

Algorithm 1 General SMO Algorithm

Result: optimum lagrange multipliers
1 compute initial *gradient*
while $\neq$ *stopping condition* **do**
2 | **select** *working set* $\alpha_i, \alpha_j \in B$
| optimize αs for selected *working set*
| compute gradient in the new search point
3 **end**

The explanation for the pseudo code 1 is as follows. For every iteration, we select a pair of αs. Upon completion of optimizing both these selected αs while keeping all other αs untouched, we compute the updated gradient at this current optimum point (line 2-3) using the old one. With this we can say that choosing the working set wisely will determine the speed at which the algorithm converges to the optimum of a set of training samples. A better working set selection was proposed in [SK+05] as an extension to Platt's SMO. We directly proceed to pick the pair of multipliers that have the highest chance of violating the KKT conditions.
For the ease of a software implementation and explanation for the pseudo code 2 we will first modify the box constraints or the *optimality criteria* as

$$0 \leq \alpha_i \leq \mathcal{C} \tag{3.5}$$

in terms of $y_i\alpha_i$ as

$$y_i\alpha_i \in [A_i, B_i] = \begin{cases} [0, +\mathcal{C}] & if\ y_i = +1,\ \mathcal{A}_i \\ [-\mathcal{C}, 0] & if\ y_i = -1,\ \mathcal{B}_i \end{cases} \tag{3.6}$$

αs could lie as $\mathcal{A}_i \leq \alpha_i \leq \mathcal{B}_i$ along with the equality constraint $\sum \alpha_i = 0$.

Pseudo Code

Algorithm 2 SMO with Maximum Violating Pair working set selection

Result: Return alphas
1 $\forall k \in \{1.....n\} \quad \alpha_k \leftarrow 0$ ▷ Initial Coefficients
2 $\forall k \in \{1.....n\} \quad g_k \leftarrow 1$ ▷ Initial Gradient
3 **while** *True* **do**
4 | $i \leftarrow arg\max_i y_i g_i \quad$ subject to $y_i\alpha_i < B_i$
5 | $j \leftarrow arg\min_j y_j g_j \quad$ subject to $A_j < y_j\alpha_j$ ▷ Maximal violating pair
6 | **if** $y_i g_i \leq y_j g_j$ **then**
7 | | **stop** ▷ Stopping criteria
8 | **else**
9 | | $\lambda \leftarrow min\{B_i - y_i\alpha_i, y_j\alpha_j - A_j, \frac{y_i g_i - y_j g_j}{K_{ii}+K_{jj}-2K_{ij}}\}$ ▷ Direction search
10 | | $\forall k \in \{1...n\} \quad g_k \leftarrow g_k - \lambda y_k K_{ik} + \lambda y_k K_{jk}$ ▷ Update gradient
11 | | $\alpha_i \leftarrow \alpha_i + y_i\lambda$
12 | | $\alpha_j \leftarrow \alpha_j - y_j\lambda$ ▷ Update coefficients
13 | **end**
14 **end**

An explanation to the pseudocode can be found in 2. Following the iterative chunking scheme, the initial coefficients of the lagrange multipliers are set to zero (line 1) and the gradients $\mathbf{g} = (g_1...g_n)$ of the objective function in equation 2.7 are set to 1 (line 2) respectively.

$$g_i = 1 - y_i \sum_j y_j\alpha_j\mathcal{K}(x_i, x_j) \tag{3.7}$$

4

Evaluation Methods

With the mathematics of the SVM laid out, we proceed with a base implementation of the SVM SMO with the maximum violating pair working set selection algorithm discussed in the previous section. The algorithm is trained and tested against three different datasets and its results are noted. Following this, 4 different incremental and approximated methods are implemented, each improving on the predecessor, trained and tested using the same three datasets and the results are compared to the base non incremental algorithm for performance and accuracy. To visualize the comparison, four different visual methods and three statistical methods are considered. A brief explanation of the methods follow.

4.1 Datasets

In order to demonstrate the learning and inference capabilities of our algorithm we choose three commonly used 'goto' datasets in the ML community to train and validate the performance and accuracy of our implementation.

The Handwritten Digits MNIST

The MNIST[1] dataset[2] is a collection of 70,000 images of handwritten digits ranging from 0 to 9. Each sample is a grayscale image of size 28x28, corresponding to 784 *features*. 60,000 samples are used for training and the remaining 10,000 are used for testing. A few of samples are shown in figure 4.1a.

Zalando Fashion MNIST

This dataset[3] is open-sourced by the research team[4] at *Zalando Fashion*, which consists of 70,000 samples divided into 10 classes of clothing and apparel, each of size 28 x 28 which again corresponds to 784 features, resembling the Handwritten Digits dataset in structure. A few samples of the classes are shown in 4.1b

CIFAR-10

The CIFAR(Canadian Institute For Advanced Research) - 10 dataset[5] is a collection of 70,000 samples of cars, birds, airplanes, animals, etc., grouped into 10 classes. Each sample is 32 x 32 in size and hence corresponds to 1024 features for each sample. Some examples from the dataset are shown in 4.1c.

[1] Modified National Institute of Standards and Technology
[2] http://yann.lecun.com/exdb/mnist/
[3] https://github.com/zalandoresearch/fashion-mnist
[4] https://research.zalando.com/
[5] https://www.cs.toronto.edu/ kriz/cifar.html

(a) Digits MNIST (b) Fashion MNIST (c) CIFAR

Figure 4.1: Represents samples of the three primary datasets used for training and validation of our algorithms, where figure 4.1a are samples from the MNIST, figure 4.1b are samples from the *fashion* MNIST and figure 4.1c are samples from the CIFAR dataset.

4.2 Visualization Methods

We have seen from the previous chapter that every feature of our sample is mapped to a higher dimension by the kernel function 2.18. For example, if we wished to visualize the hyperplane produced by the SVM post training or the specific support vectors it produces for different classes, it becomes difficult to visualize these high dimensional data in a 2 or 3 dimensional space. This is known as the *curse of dimensionality*. Two methods can be considered for overcoming this. *Feature selection* involves choosing which features are of importance to us and visualizing only those features. There is no dimensionality reduction involved. *Feature extraction* is a dimensionality reduction technique where various methods such as PCA, t-SNE are used to transform data from a higher dimension to a lower dimension. To study the differences between the visualization techniques, the *Iris* dataset[6], containing 4 features are used.

4.2.1 PCA for Dimensionality Reduction

Principal Component Analysis (PCA) is a common technique used for dimensionality reduction which involves reducing the number of *features*, each signifying a dimension - hence higher number of dimensions - to a lower number which can be represented in a 2 dimensional or 3 dimensional plot. If we were to consider the 784 features of the MNIST dataset, to view each feature we would need a 784 dimension plot(!) which is hard to visualize. Hence the technique takes a set of features that might be correlated and converts them into a *linear* set of uncorrelated values called the *principal components* by computing the variance given by

$$variance \;=\; E[(X-\mu)^2] \tag{4.1}$$

where $(X-\mu)^2$ is the square of the deviation of a sample from the mean μ and E is the expectation. This is computed without losing out too much information while trying to pack all the information contained in the 784 features into 2 or 3 features. The first component will have the largest variance and hence

[6] https://archive.ics.uci.edu/ml/datasets/iris

the largest amount of information, the second component will have the second largest variance and the remaining information, and so on. This way, to represent a sample with 784 features, we would have to obtain the first two principal components for a two dimensional representation and the first three principal components for a three dimensional representation.

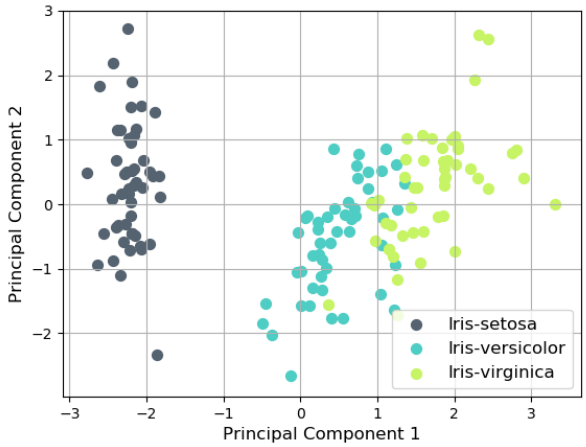


Figure 4.2: Represents the result of a PCA plot for samples from the iris dataset containing three classes

PCA has a disadvantage that it always tries to find the *linear* principal components and it can happen that our data is usually non linear. When mapping higher dimensional data to a linear plane, points that were actually dissimilar in the higher dimension may be superimposed, which can be seen in figure 4.2 and turn out to be similar in the reduced 2 dimensional plane. Basically PCA tries to squash points from a higher dimension to a common lower dimension. Comparisons with this squashed plot might give us a false sense of similarity between the points.

4.2.2 t-SNE for Dimensionality Reduction

t-distributed Stochastic Neighbor Embedded (t-SNE) is a visualization technique which is also used for dimensionality reduction, invented by [MH08]. Unlike PCA, this is a non-linear technique which can be used for visualizing high dimensional data in lower dimensions. The technique models higher dimensional points to lower dimensional point by grouping similar points as neighbours and dissimilar points as distant. There are two stages to the t-SNE algorithm. The first stage consists of modelling the probability distribution of points in the higher dimension in such a way that similar points are chosen first than the ones that are dissimilar. The second stage involves modelling the probability distribution of the same points in a lower dimension such that it minimizes the *Kullback–Leibler* (KL) [KL51] divergence[7] between the two transformations. A KL measure of 1 indicates complete dissimilarity and 0 indicates complete similarity. The conditional probability between two points is calculated as

$$p_{i|j} = \frac{exp(-\left\|x_i - x_j\right\|^2 / 2\sigma_i^2)}{\sum_{k \neq i} exp(-\left\|x_i - x_k\right\|^2 / 2\sigma_i^2)} \tag{4.2}$$

Additional hyperparameters determine the result of the transformation. *Perplexity* denotes the number of nearest neighbours to be considered for a chosen point. The metric for the pairwise distance calculation can also be chosen from options such as the *eucledian* distance, or the *manhattan* distance, etc [8]. Unlike the PCA, t-SNE is able to maintain the original distances [KL51] between the points due to the KL divergence. A similar plot of the t-SNE reduction for the iris dataset is shown in figure 4.3. It is noticeable

[7] a measure to identify the differences between two probability distributions
[8] https://scikit-learn.org/stable/modules/classes.htmlmodule-sklearn.metrics.pairwise

that the cluster of black points are far apart from the other two classes, retaining its similarity properties form the original high dimensional space.

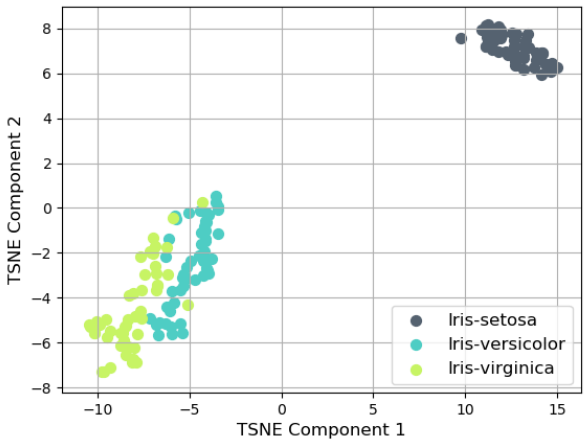


Figure 4.3: Represents the result of a t-SNE plot for iris dataset containing three classes

4.2.3 Andrew's plots

The *Andrew's curves* are a way to visualize the structure of high dimensional data without having to reduce its dimensions. Basically for a high dimensional point $\mathbf{x}$, the technique plots each of its dimensions as a value of the *fourier series* expansion given as

$$f_x(t) = \frac{x_1}{\sqrt{2}} + x_2 sin(t) + x_3 cos(t) + x_4 sin(2t) + x_5 cost(2t)... \; where \; -\pi \leq t \leq +\pi \tag{4.3}$$

where each dimension is a point between $+\pi$ *and* $-\pi$. This technique is able to maintain the mean, variance and original distances of the points [Spe03]. But sometimes this time series conversion of the data points might lead to changes in the meaning of the data. Figure 4.4 is an example of the andrew plot for the Iris dataset.

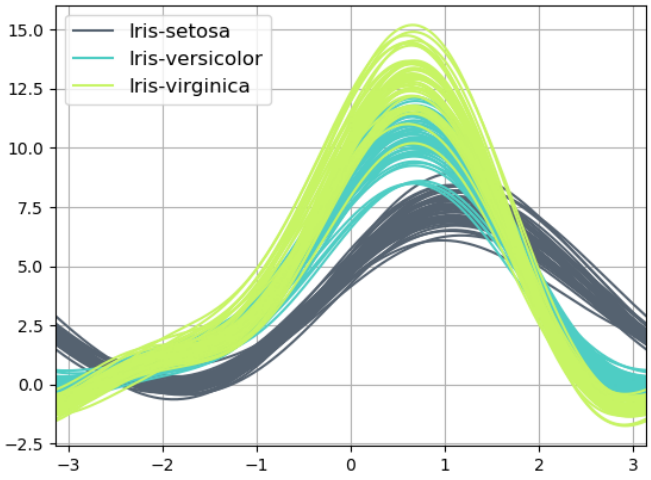


Figure 4.4: Andrew plot for Iris dataset

4.2.4 Parallel Coordinate Plots

Originally invented by [Ins09] as a coordinate transformation method, this technique is popularly used for visualizing high dimensional data similar to the andrew's plots in the sense no dimensionality reduction is done thereby preserving all the features of the sample point. Instead of plotting the values of each dimension as the value of a function at that particular point, the parallel coordinate plot simply normalizes all the values to within a range and directly plots the normalized value as a point on a new axis and connects them together to show the variance. The name parallel coordinates stems from this where each dimension of the original point becomes a parallel y axis and hence for an N dimensional point we would then have N parallel axes. Unlike the andrew's plot, these parallel lines do not correspond to a time series point, therefore preserving natural order.

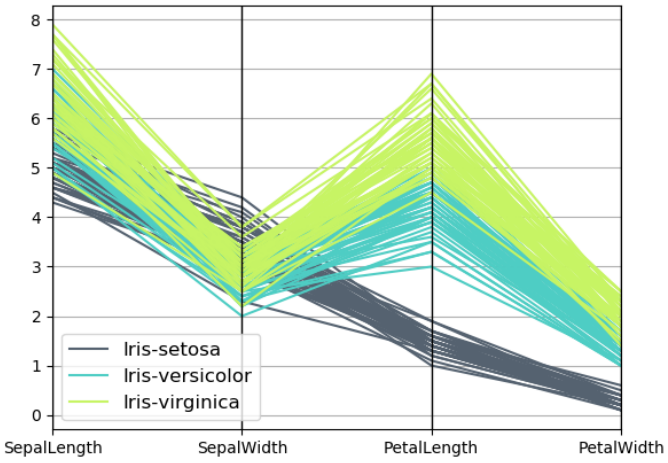


Figure 4.5: Parallel coordinate plot for iris dataset

Figure 5.33 is the plot for the iris dataset. Notice that there exists a parallel y axis for each of the four features or dimensions of a sample in the dataset.

4.3 Statistical Methods

The previous three methods will let us picturize the results of the output of the SMO after training and testing in the form of plots. For a statistical comparison, three techniques are used to validate the results of our algorithms.

4.3.1 Cosine Statistical Similarity

The *Cosine* similarity is not just a measure of distance but also the *orientation* due to considering the angle between two points. The similarity is given by

$$cos(\theta) = \frac{A.B}{|A||B|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{4.4}$$

where A_i and B_i are the features of the sample vectors respectively. By its nature a cosine table has values from 1 to 0 where 1 indicates complete similarity and 0 indicates dissimilarity.

Angle in Degrees	0°	30°	45°	60°	90°
Cosine Value	1	0.866	0.707	0.5	0

4.3.2 Confusion Matrix

Accuracy alone is not a good measure for the performance of a classification model. Considering a binary classifier with two classes, +ve and -ve, a *Confusion matrix* is tabular representation of data used to show how good a classifier is.

	Predicted Positives	**Predicted Negatives**
Positives	True Positives	False Negatives
Negatives	False Positives	True Negatives

- *True Positives* are values that actually belong to the positive class and the classifier also predicted them correctly as positive
- *False Negatives* are values that actually belong to the positive class but the classifier predicted them incorrectly as negative
- *False Positives* are values that actually belong to the negative class but the classifier predicted them incorrectly as positive
- *True Negatives* are values that actually belong to the negative class and the classifier also predicted them correctly as negative

Hence a confusion matrix is basically a contingency table[9] of the count of the true positives, false negatives, false positives and true negatives. This is extended to multi class classification by the same technique discussed in section 5.1.

4.3.3 Precision, Recall and f1-score

From the confusion matrix we can derive three parameters that can further help in studying the performance of a classifier without relying on just accuracy. *Precision* tell us out of all the positive prediction by the classifier, how many of them were actually positive.

$$\frac{tp}{(tp + fp)} \tag{4.5}$$

where tp is the number of true positives and fp the number of false positives. It can be stated as the ability of the classifier to not classify a negative label as positive. A higher value of precision indicates a lower number of false negatives. *Recall* is count of all the true positives found by our classifier out of the actual number of true positives.

$$\frac{tp}{(tp + fn)} \tag{4.6}$$

where tp is the number of true positives and fn the number of false negatives. It can be stated as the ability of the classifier to find all true positive labels. A higher value of recall indicates a lower number of false negatives. The *f1-score* is the weighted harmonic mean of the precision and recall, given as

$$2 * \frac{precision * recall}{precision + recall} \tag{4.7}$$

A larger value of precision and recall leads to a large value of the f1-score.

[9] one that denotes frequency distribution of the entries

5

Software Implementation

A general ML workflow for a classifier can be visualized as follows in figure 5.1.

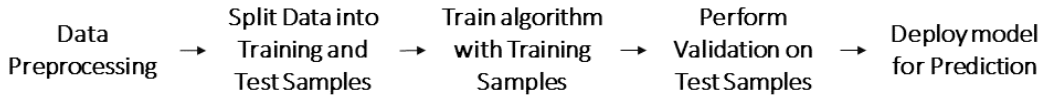


Figure 5.1: ML Workflow

Following the flowchart in figure 5.1, the data set we will be using is the MNIST dataset explained in the previous section. All results and observations that follow are with respect to this dataset. We will discuss 5 experiments which include a non - incremental implementation of a multi class SVM SMO algorithm followed by 4 different incremental and approximated versions of the algorithm. Each successive algorithm is an improvement over its predecessor. Results for each of them are provided alongside the implementation. A final comparison of every incremental technique against the base non - incremental version is provided as a summary, with accuracy and misclassification being the main metrics of comparison. For all algorithms the hyperparameters are set to $\mathcal{C} = 10.0$, γ for the RBF kernel = 0.05 and the $\epsilon = 2.0$. These hyperparameters are optimally chosen after subjecting the algorithm to an exhaustive *Grid Search*.[1] Additionally a cap of 1000 is set for the maximum number of iterations for the algorithm in case of being stuck in local minimum.

5.0.1 The Training and Test samples

Figure 5.2 and 5.4 represents the t-SNE reduced plots for the training and test samples sets we are going to use for our algorithms. While the plots cannot indicate the actual distribution of the samples in the higher dimensional space to lower dimensional space as a result of the t-SNE mapping being a non linear transformation [vH08], the plot helps us understand if the classes are actually differentiable than the other and which of them might pose difficulties in discerning. They also help understand the sparsity or density of a particular class of samples.

[1] https://scikit-learn.org/stable/modules/generated/sklearn.model$_s$*election.GridSearchCV.html*

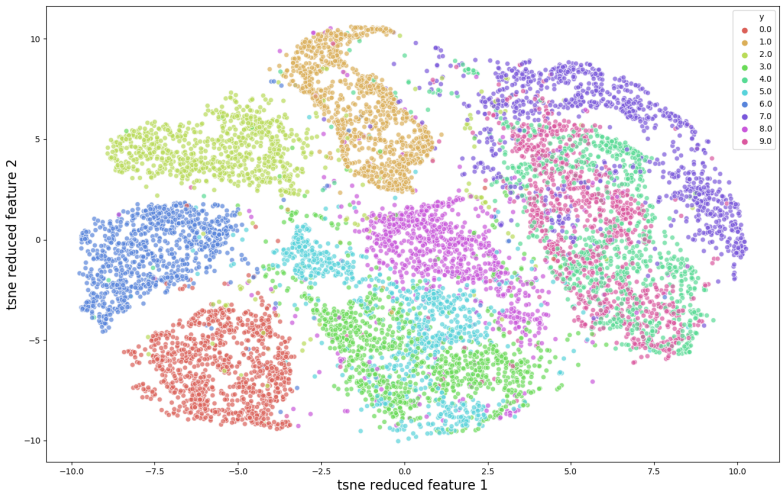


Figure 5.2: t-SNE reduced plot for the training data set

Figure 5.3: The color map in Figure 5.2 represents labels from 0 to 9 respectively and their corresponding sample clusters in the training dataset. The values along the x and y axes indicate two of t-SNE's dimensionality reduced components to view the original 784 dimension dataset in a 2 dimensional space.

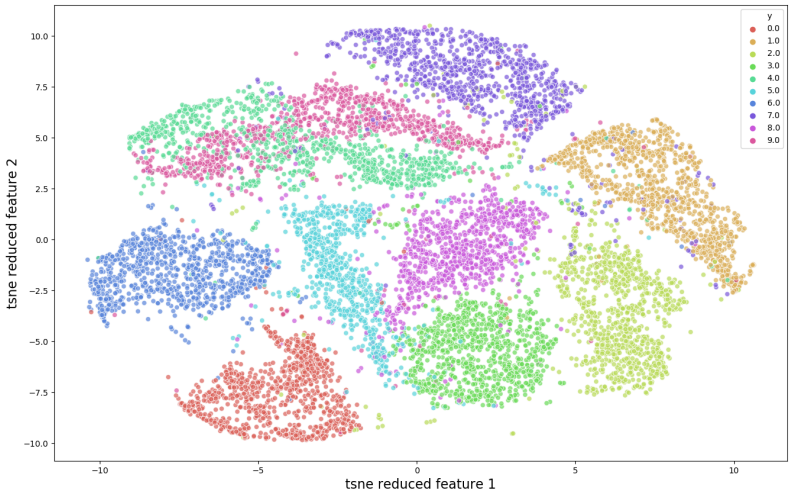


Figure 5.4: t-SNE reduced plot for the test data set

Figure 5.5: The color map in Figure 5.4 represents labels from 0 to 9 respectively and their corresponding sample clusters in the test dataset. The values along the x and y axes indicate two of t-SNE's dimensionality reduced components to view the original 784 dimension dataset in a 2 dimensional space.

5.1 Multi Class SVM

Originally envisioned for binary classification, SVMs can be extended to allow multi class classification by two techniques known as *One vs One* (OVO) and *One vs All* (OVA). The term multi class simply means that the data encompasses more than just two classes and each label corresponding to one class.

5.1.1 One vs One

This method involves training one classifier per pair of classes. Figure 5.6 shows this concept for three classes. If we have **N** classes, we have **N** * (**N** - 1)/2 number of classifiers which is obviously a disadvantage with the increase in computation and memory resources. Prediction involves a majority vote over all classifiers for a class and the one with the maximum votes gets assigned to the class. In figure 5.6, the red, blue and black shapes denote three respective classes. The dotted line indicates the classifier. Each of these classifiers are color coded in accordance to their classes.

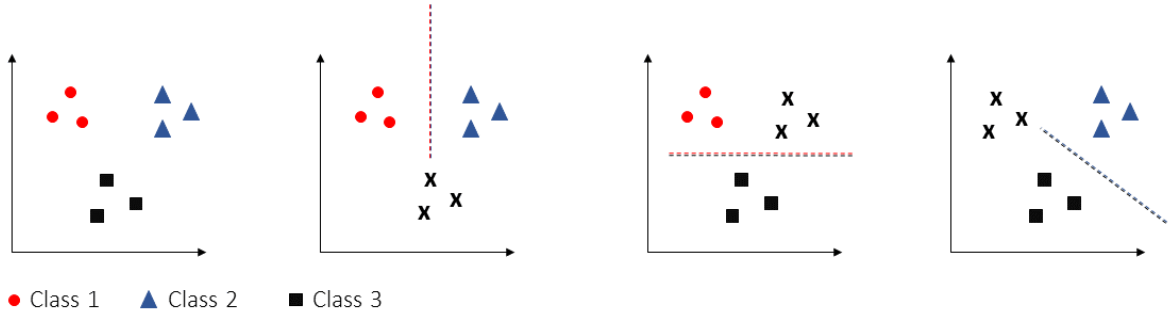


Figure 5.6: Representation of the OVO Multiclass technique

5.1.2 One vs All

This method involves training one classifier per class. Figure 5.7 shows this concept for three classes. Each class is pitted against the rest and the classification model is derived. Hence the advantage is that for **N** classes we have only **N** classifiers. Prediction is straightforward since we have only one classifier per class. For the reasons mentioned in [MCS] and the ease of implementation, we proceed with the OVA technique. In figure 5.7, the red, blue and black shapes denote three respective classes. The dotted line indicates the classifier. Each of these classifiers are color coded in accordance to their classes.

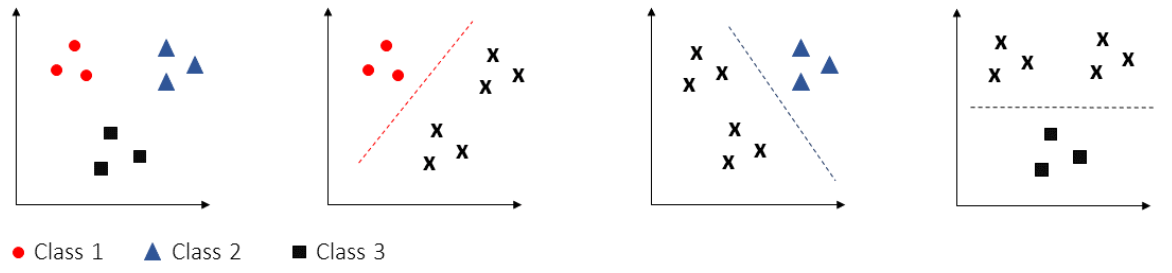


Figure 5.7: Representation of the OVA Multiclass technique

5.2 The Non-Incremental algorithm

The following algorithm and its results are going to be considered as our baseline implementation with which all other versions of the incremental and approximated algorithms are going to be compared with. The algorithm is split into a training phase and an inference phase of which the pseudo code for the training phase is presented in algorithm 3. The sequential minimal optimization procedure draws from the previously introduced algorithm 2 with the direction search and working set selection.

5.2.1 Pseudo Code

Algorithm 3 Non Incremental Multiclass SVM

Result: trained model with one classifier per class
Input: samples, labels
Output: trained classifier

```
1  Function train(training samples, labels):
2      loop for number of training samples do
3          loop for number of classes do
4              binarize labels                                  ▷ for multiclass classification
                Function smo(training samples, binarized labels):
5                  while true do
6                      if optimality criteria met or maximum iterations reached then
7                          break
8                      end
9                  end
10                 return support vectors, lagrange multipliers
11         end
12     end
13     return multi class classifier
```

The dataset we considered consists of 70,000 samples which is split to contain 60,000 training samples and 10,000 test samples. Each sample ranges in values from 0 to 255, which represent the intensity of the point in the image in the handwritten digit dataset. As a first step, we perform data preprocessing by normalizing the values from a scale of [0, 255] to [0, 1].

The pseudo code in 3 shows the training phase (lines 1-13) of the algorithm. The training phase takes in the 60,000 training samples and its corresponding labels as inputs (line 1). Since there are multiple classes, the multi class classifier technique one vs all method discussed in section 5.1 is used. A *binarization* procedure (line 4) involves picking a particular class, reassigning its label to the *+ve* class and treating all other samples apart from the chosen class as the *-ve* class. This inherently breaks down to a binary classification problem to which the binary classification can be applied. These binarized labels and the samples are passed to the SMO function (lines 5-10). This function is implemented using the principles discussed in the section 3.4, which includes performing an optimization and a direction search until the stopping conditions of the optimality criterion or maximum iterations are met. The SMO function then returns the support vectors and lagrange multipliers for that particular label (line 10). This process is repeated until all 10 classes are paired against themselves and the rest are subjected to the SMO function. This results in one classifier per class, which stores the support vectors and their corresponding optimal lagrange multipliers in the form of a trained model.

5.2.2 Results

For SVMs, it is interesting to visualize the decision boundary formed in the original feature space as a result of training. [MHT18] puts forth tested approaches towards visualizing classifier decision boundaries by dimensionality reduction techniques to view 2d and 3d plots of boundaries for high dimensional multi class data, with research into t-SNE and PCA for dimensionality reduction, both of which were discussed in section 4.2. With the disadvantages that PCA has with improper feature approximation, t-SNE was chosen for this purpose.

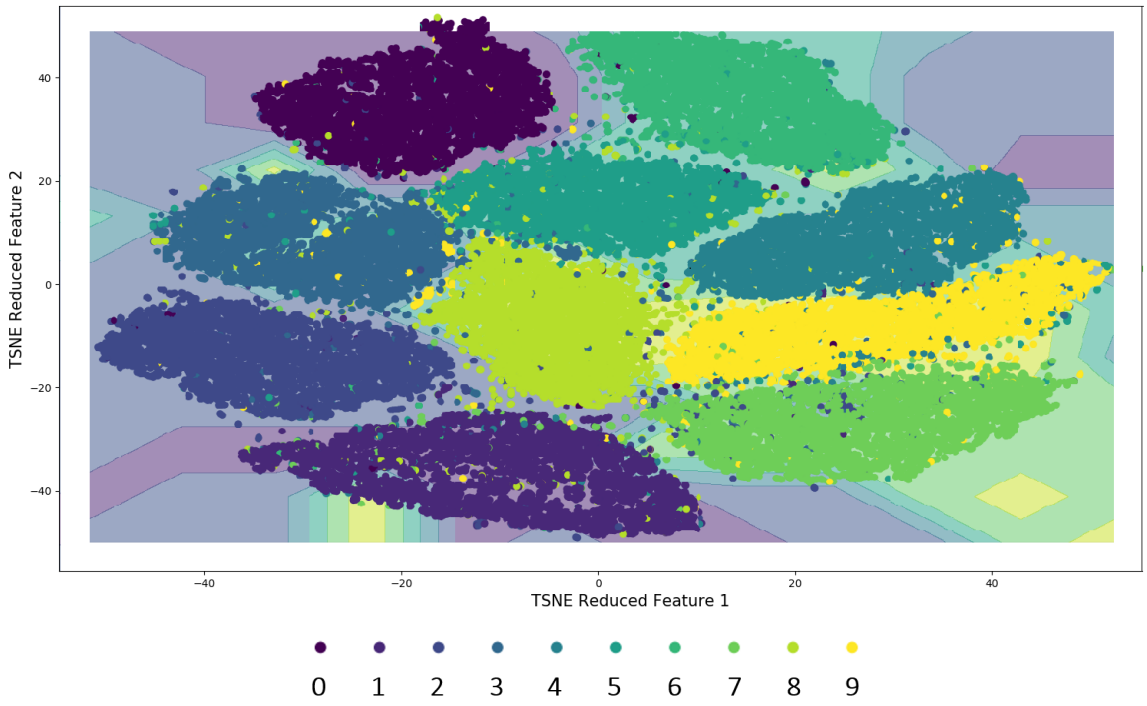
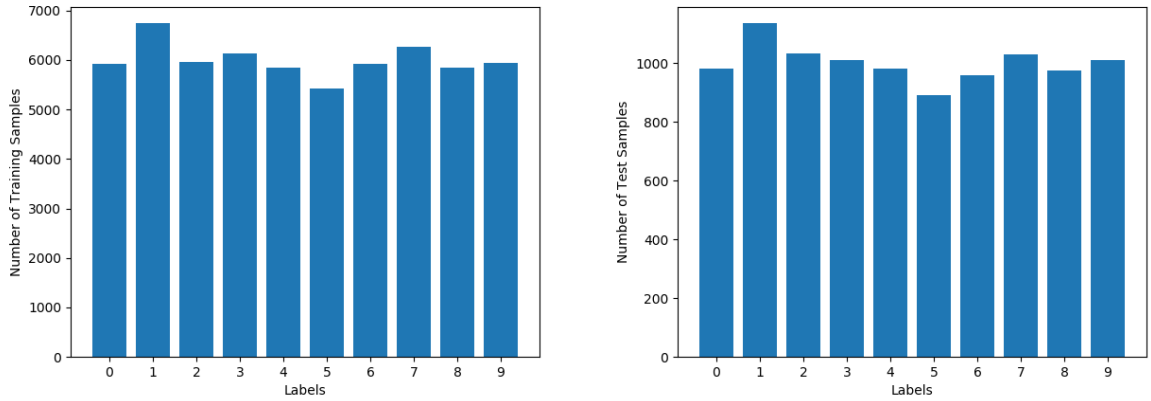


Figure 5.8: Decision boundary plot for the non incremental algorithm.

Figure 5.8 is a t-SNE dimensionality reduced visual representation of the decision boundaries formed for the 10 classes as a result of the training. With respect to the pseudo code in algorithm 3, at the end of each optimized step for a class, a decision boundary is produced. Each boundary represents a classifier and the filled regions represent the contours for the bounds of that particular class. These plots are especially useful when we would like to see the behaviour of the training process and its effect on the decision boundary. The color map represents labels from 0 to 9 respectively and their corresponding decision regions. The values along the x and y axes indicates two of t-SNE's dimensionality reduced components respectively to view the original 784 dimension dataset in a 2 dimensional space. The resultant of the t-SNE computation gives us values between -55 and +55 in both axes where our training samples lie.

It is noticeable that the plot is densely populated given the number of samples (60,000) that were used for training.

Figure 5.9a represents the number of samples per class used for the training phase of the algorithm. The labels are more or less uniformly distributed with a slightly higher number of 1s and 7s among the rest. It is important that an equal distribution of samples for all labels are present in the training dataset, which otherwise might lead to a bias towards a label with lower number of samples and an imbalance in the model [SZ05]. Hence we can say that the chosen dataset is a *balanced* dataset.

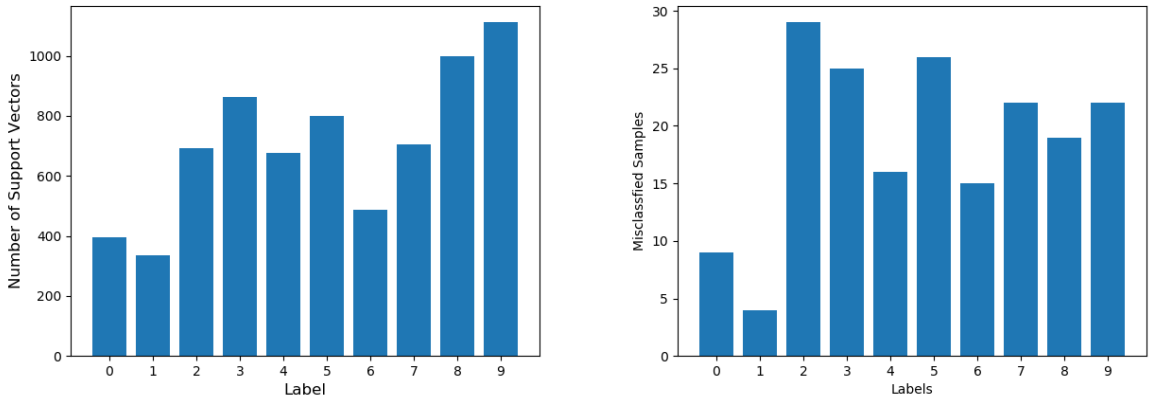


(a) Number of training samples per label

(b) Number of testing samples per label

Figure 5.9: Represents the distribution of samples for the training phase of the algorithm, where 5.9a denotes the count of samples for each label in the training set and 5.9b for the testing set.

Figure 5.10a represents the number of support vectors produced per label as a result of training. The more the number of support vectors, the harder it was for the algorithm to be able to differentiate the particular label from the rest of them. Hence it compensates by converting more number of the training samples into support vectors to back its decision boundary. Figure 5.10b is a plot of the number of samples per label that were misclassified from the test dataset.



(a) Support vectors per label

(b) Misclassified test samples per label

Figure 5.10: Represents the result of the training and testing phases of the algorithm, where 5.10a indicated the total number of support vectors produced per label and 5.10b indicated the samples that were misclassified.

We can infer that the model performs well for labels with lower misclassification rates such as 0, 1, 4 and 6 than the others 2, 3, 5, 7 and 8.
Table 5.1a depicts the 3 metrics used to quantify as explained in 4.3.3. A higher value is welcome, which interprets to having good classification capability. The confusion matrix in table 5.1b give us the results for the test samples used for prediction.

Label	**Precision**	**Recall**	**f1-score**
0.0	0.99	0.99	0.99
1.0	0.98	1.00	0.99
2.0	0.99	0.97	0.98
3.0	0.98	0.98	0.98
4.0	0.98	0.98	0.98
5.0	0.99	0.97	0.98
6.0	0.98	0.98	0.98
7.0	0.97	0.98	0.98
8.0	0.98	0.98	0.98
9.0	0.97	0.98	0.97

(a) Classification report

	0	1	2	3	4	5	6	7	8	9
0	971	3	0	0	0	0	3	1	1	1
1	0	1131	1	1	0	0	1	1	0	0
2	6	3	1003	2	4	0	1	7	5	1
3	0	1	3	985	0	5	0	6	6	4
4	1	1	1	0	966	0	3	1	1	8
5	2	1	0	8	1	866	4	1	4	5
6	2	6	0	1	3	1	943	0	2	0
7	0	7	5	0	1	0	0	1006	1	8
8	2	1	1	2	1	0	4	3	955	5
9	1	2	0	4	5	1	1	7	1	987

(b) Confusion matrix

Table 5.1: Represents the classification report in 5.1a and confusion matrix in 5.1b for the classification results.

The algorithm performs with an accuracy of **98.13**%.

The aim of this research was to be able to run the algorithm on an our embedded test platform restricted in memory and computational power. The embedded platform chosen is the *PYNQ* which has just 512 MB of RAM. The Linux operating system on it takes up about 100MB and the background process that run take up another 50MB to 80MB of RAM, leaving us with about 350MB of RAM. Combined with a 650MHz clock, it is pretty restrictive to run computationally intensive machine learning algorithms on large scale data. The entire dataset of 60,000 samples of the MNIST dataset would not fit in memory and cannot be operated upon at once. An excerpt from the memory profiling of the previous algorithm shows that at any given point of time the algorithm uses 356MB of RAM to run and an average of 37.4MB per sample at any instance of computation, which noticeably exceeds the available RAM. To be able to train all 60,000 sample we introduce batch processing of the samples as a step towards incremental learning to generate the SVM model we would have to divide our entire training sample set into batches, learn from each of them individually and finally create a combined model resulting from training the SVM on the individual batches. The following sections proceed to discuss the base incremental technique utilizing this batch processing approach.

5.3 Background study on Incremental Learning

Incremental learning refers to the process of updating an existing model's knowledge with new incoming data. This is followed either when the dataset is too large to fit in main memory or when the data itself is a gradual stream of incoming values. This is useful when dealing with big data. Algorithms that were first modified to use incremental learning were *decision trees* [CSF86] and *neural networks* [BFP99]. The first incremental method proposed for the SVM was the *Batch SVM* [Sye+99] which involved dividing the entire dataset into a series of batches and training on each of them at a time. The model's parameters are updated upon completion of every batch until all the batches are exhausted and we result with a complete trained model. This works well when all batches contain similar kind of data or is a *homogeneous* dataset. When the data is *heterogeneous*, addition penalty parameters are added to the *cost function* of the SVM [RÖ1] and an effort is made to balance this trade-off. A study into the impact of new incoming data was made in [JZJ05] such that the end model comprises of all the support vectors that would have been generated if the data was processed as a whole.

Other attempts at incremental learning specifically for classification include training one classifier per batch and combining all the resulting classifiers from all batches into one single classifier [WL07] where the support vectors produced by all the individual classifiers are simply averaged to form the support vectors of the final resultant classifier. These approaches are known as *ensemble* approaches. But a simple average over the support vectors leads to approximated imprecise results with averaging changing the meaning of the features of the samples forming the support vectors [WL07]. In an extension to ensemble methods, [HM07] considers a weighted majority voting technique on the classifiers resulting from each batch. [Gol89] considers using a *Genetic* algorithm to evolve a final resultant classifier from the set of classifiers from the batches. The ensemble methods tend to be resource and computationally more intensive than traditional incremental techniques. Studies in [Die00] indicated high storage usage as the number of ensemble classifier sets increased, which is not suitable given that the demonstrative embedded platform is already on a tight resource budget.

Another setting is the *Online Learning* method where samples from the dataset are trained one at a time [CP01]. Online learning is a subset of incremental learning, used for more for *Regression* than for *Classification*. Even in regression applications, it works well with linear SVM cases but upon introducing a kernel to deal with complex data, the computation time increases by many folds. Also in classification, in cases where the online learning algorithm has not encountered a particular label in training and come across it in a prediction phase, the algorithm is unable to classify this sample simply because it has not learned this label yet. A lot of redundancy is thus involved when it comes to SVMs with online learning for classification. To overcome this, a windowing technique [JB14] was proposed where instead of just a single sample, a *window* is opened to pick up a set of samples from the online stream, which is inching again towards a batch based learning system. With the window technique for SVM in classification, there always is a question as to when is the opportune moment to sample though the window such that we pick a set of samples from every class that the classifier might encounter during the prediction phase.

With all these points in mind we will first proceed with a baseline implementation of the support vector forwarding incremental technique and then improve on it. It is to be noted that in the first base incremental approach, where the support vectors are simply forwarded to the next batch, it was noted that the SVM seems to ignore the support vectors from the previous batch and replace them with support vectors learned from the newer batch [RÖ1]. This is usually a desired trait of the SVM which indicates its robustness and helpful in areas such as outlier detection [MS19]. During a prediction phase when we encounter samples from previous older batches, the classifier tends to perform badly for them. This is due to the fact that support vectors corresponding to the characteristics of the samples from that particular batch were discarded with new batch of data coming in. Hence in all following implementations after this section, we proceed with collecting all support vectors from each batch for their respective classes instead of appending them to the next training batch which might result in them being discarded.

5.4 Incremental Learning with Support Vector Forwarding

The earliest incremental type algorithm for the SVM was proposed by [Sye+99], which involved dividing a dataset into a number of smaller batches and subjecting them to training individually. Upon completion of training on each batch, the support vectors from the current batch are forwarded or appended to the samples of the next batch for re-training. This is under the premise that for a homogeneous dataset, the support vectors are a sufficient representations of the samples [RÖ1], and hence forwarding them to the next batch inherently involves transferring the characteristics of the previous batch to the next batch.

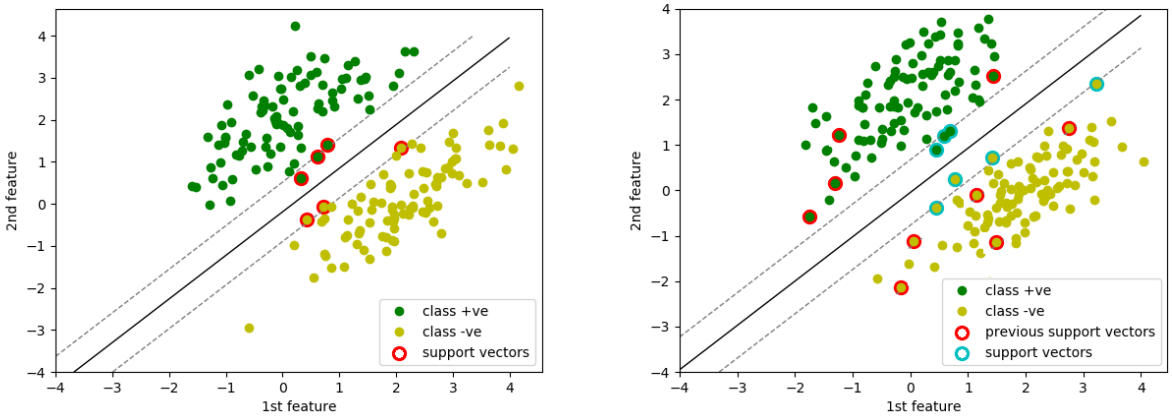
5.4.1 Pseudo Code

Algorithm 4 Incremental Multiclass SVM with Support Vector Forwarding

Result: trained model with one classifier per class
Input: samples, labels
Output: trained classifier

```
for number of batches do
    if batch number = 1 then
        Function train(training samples, labels):
            loop for number of training samples do
                loop for number of classes do
                    binarize labels                                  ▷ for multiclass classification
                    Function smo(training samples, binarized labels):
                        while true do
                            if optimality criteria met or maximum iterations reached then
                                break
                            end
                        end
                        return support vectors, lagrange multipliers
                end
            end
            return multi class classifier
    else
        append support vectors to training samples                   ▷ support vectors from previous batch
        Function train(training samples, labels):
            loop for number of training samples do
                loop for number of classes do
                    binarize labels
                    Function smo(training samples, binarized labels):
                        while true do
                            if optimality criteria met or maximum iterations reached then
                                break
                            end
                        end
                        return support vectors, lagrange multipliers
                end
            end
        return updated multi class classifier
    end
end
```

This way we keep forwarding the support vectors of the current batch for retraining along with samples of the next consecutive batch. The model can be pictorially seen in figure 5.12. Following the pseudo code in 4 which is similar to pseudo code 3 with the addition of a second stage of training. The first batch of training samples are subjected to the SVM SMO (lines 3-15) and the support vectors for every class (line 12) is stored. On encountering the second batch of data, indicated by a batch number > 1 (line 16), the stored support vectors from training on the previous batch are appended to the samples in the second batch (line 17) and are subjected to the SVM SMO (lines 18-29). For a batch size of 6 this results in 10,000 samples + support vectors from the previous batch. This process is repeated for the number of batches created. The classifiers are saved at every batch iteration such that we can use the model at any given point of time for prediction.



(a) Support vectors produced from the initial batch.

(b) Support vectors forwarded to the successive batch.

Figure 5.11: Represents the concept of the incremental algorithm where 5.11a represents support vectors generated from a previous batch and 5.11b represents these support vectors among training samples.

Figures 5.11a and 5.11b represent a demonstration of the first two stages of this algorithm. The samples chosen have two features with the first and second being marked on the x and y axes respectively. The solid black line represents the decision boundary and the dotted lines represent the positive and negative side margins. The points encircled in red are support vectors from the first batch in figure 5.11a. These points are then part of the training samples in figure 5.11b, along with support vectors of the result of the combined training in the second batch, represented as points encircled in turquoise.

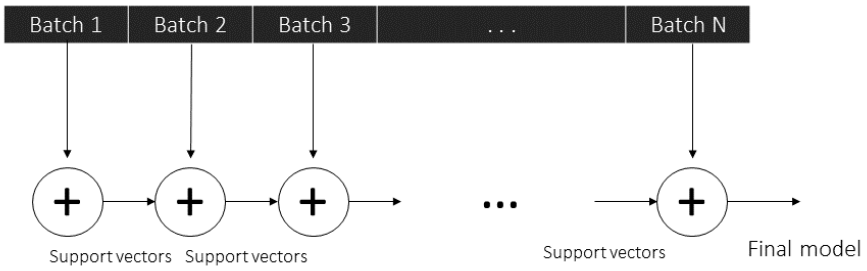


Figure 5.12: Incremental learning by support vector propagation

5.4.2 Results

Figure 5.8 is a t-SNE dimensionality reduced visual representation of the decision boundaries formed for the 10 classes as a result of the training. The plot is comparatively sparse since only the last batch of training samples and all the forwarded support vectors were considered for the plot. With respect to the pseudo code in algorithm 4, at the end of each optimized step for a class, a decision boundary is produced. Each boundary represents a classifier and the filled regions represent the contours for the bounds of that particular class. These plots are specially useful when we would like to infer the behaviour of the training process and its effect on the decision boundary. The color map represents labels from 0 to 9 respectively and their corresponding decision regions. The values along the x and y axes indicate two of t-SNE's dimensionality reduced components respectively to view the original 784 dimension dataset in a 2 dimensional space. The resultant of the t-SNE computation gives us values between -80 and +80 in both axes where our training samples lie.

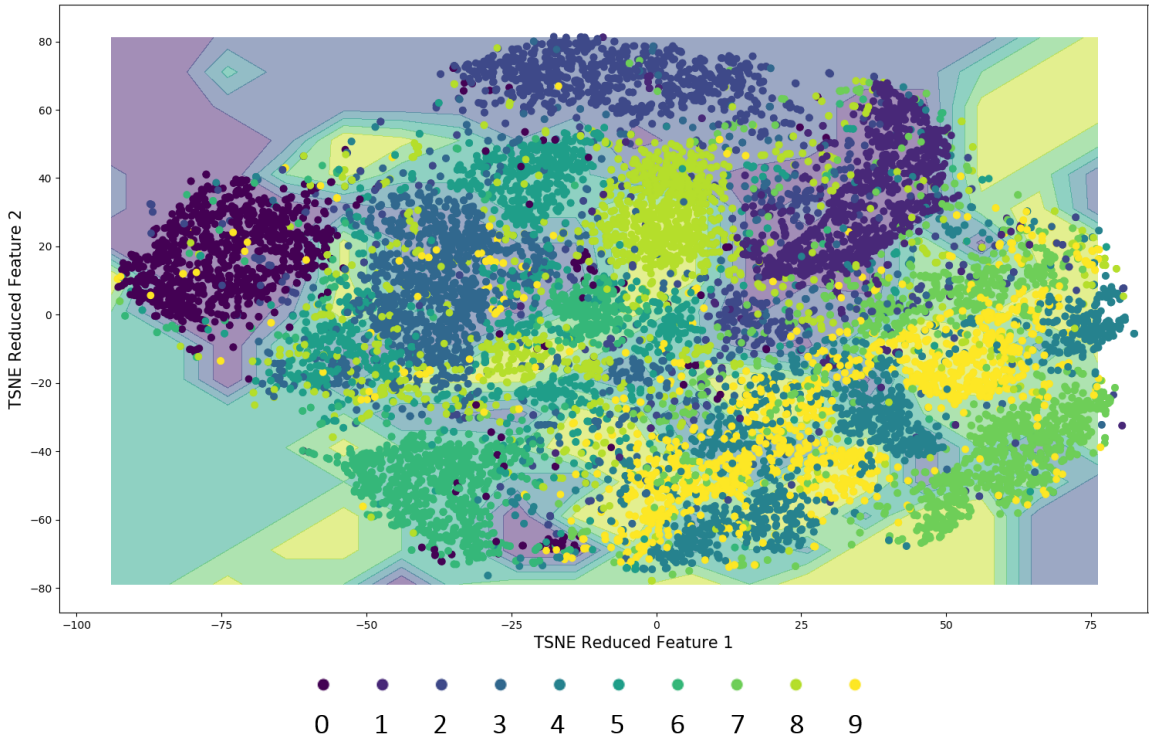
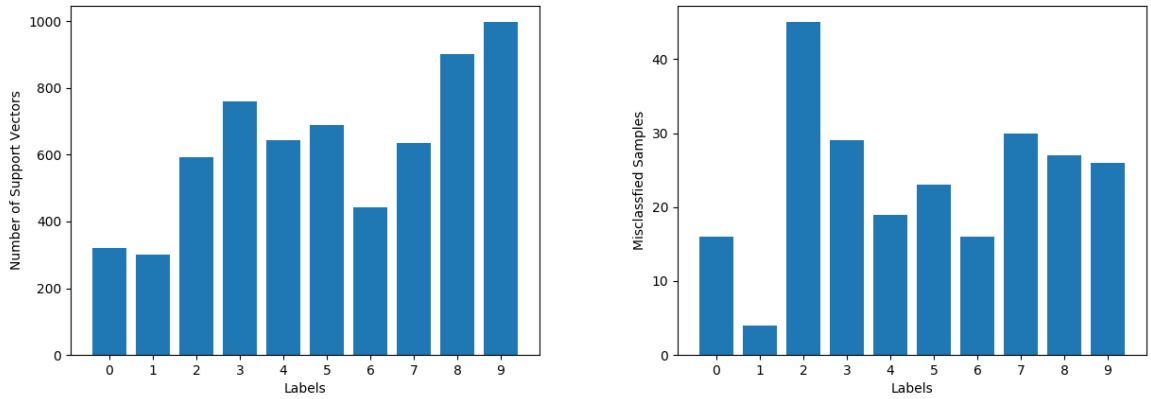


Figure 5.13: Decision boundary plot for the incremental learning algorithm with support vector forwarding.

As explained in the section 4.2, t-SNE does not maintain the distances and nearest neighbours during the transformation, it only indicates the separability of the classes. Hence we infer that if the points were separable in the high dimensional space after kernel transformation, they are separable in the transformed t-SNE plot too. Based on this we can observe from the initial t-SNE plot of the training samples from figure 5.2 how separable the classes are and the density of samples that result from the final stages of the algorithm as a result of the support vector forwarding.

Figure 5.14a represent the final number of support vectors produced as a result of training on all batches. Coinciding with our reasoning for the increase in support vectors from the decision boundary plot, labels 3,8 and 9 seem to have more support vectors when compared to the remainder of the labels. Figure 5.14b shows the misclassified samples from the test batch with the number for each label.



(a) Support vectors per label

(b) Misclassified test samples per label

Figure 5.14: Represents the result of the training and testing phases of the algorithm, where 5.14a indicated the total number of support vectors produced per label and 5.14b indicated the samples that were misclassified.

Table 5.2a depicts the 3 metrics used to quantify as explained in 4.3.3. A higher value is welcome, which interprets to having good classification capability of the classifier for the label whereas a lower value is vice versa.
Table 5.2b represents the confusion matrix for the resulting prediction run on the test samples. In accordance with figure 5.14b labels 2 and 7 have a higher misclassification rate than the rest of the labels.

Labels	**precision**	**recall**	**f1-score**
0.0	0.98	0.98	0.98
1.0	0.97	1.00	0.98
2.0	0.99	0.96	0.97
3.0	0.98	0.97	0.98
4.0	0.97	0.98	0.98
5.0	0.98	0.97	0.98
6.0	0.98	0.98	0.98
7.0	0.97	0.97	0.97
8.0	0.98	0.97	0.98
9.0	0.96	0.97	0.97

(a) Classification report

	0	1	2	3	4	5	6	7	8	9
0	964	2	1	1	0	1	8	1	2	0
1	0	1131	1	0	0	1	1	1	0	0
2	11	6	987	3	6	0	1	8	6	4
3	0	2	4	981	0	7	0	6	6	4
4	0	1	1	0	963	1	5	1	1	9
5	1	2	0	7	1	869	2	1	3	6
6	1	6	1	1	3	3	942	1	0	0
7	0	9	6	1	3	0	0	998	2	9
8	2	2	1	6	3	3	1	2	947	7
9	2	5	0	1	9	2	0	6	1	983

(b) Confusion matrix

Table 5.2: Represents the classification report in 5.2a and confusion matrix in 5.2b for the classification results.

The algorithm performs with an accuracy of **97.65**%.

It was noted that the accuracy is a factor of the batch size. Currently a batch size of 6 was chosen after a series of experiments with different batch sizes and an exhaustive *Grid Search*. With an increase in the number of batches, the accuracy dropped to 94.25%, but the algorithm did not see an increase in accuracy with a decrease in batch size. This can be attributed to the fact that a larger batch size leads to a larger gradient update for the direction search during training for the optimal lagrange multipliers [MR17]. It was also found that mini batches work well with the *stochastic gradient descent*[2] optimization method for the SVM [Li+14] and larger batches work well with the sequential minimal optimization. We will proceed with the batch size of 6 for the further implementations.

So far we tend to proceed with training all the samples in a batch. When an attempt was made at implementing this algorithm on the PYNQ, even though the algorithm is batch processed, it resulted in out of memory errors indicating that the number of samples were just too high to be operated upon during the later stages of the algorithm. This is due to the support vector forwarding which appended samples to the next training batch which results in a large increase of samples as all support vectors get accumulated. Leveraging the principles of the SVM formulation, it is possible to approximate the number of samples that are used for training based on their similarity to the samples already processed or sub sample the batch to pick only samples of interest. The next section discusses three approximated implementations of the incremental SVM.

[2] https://scikit-learn.org/stable/modules/sgd.html

5.5 Background study on Approximation techniques

In an effort to reduce the run time and computational complexity of the benchmark algorithm, we look at methods to reduce the number of samples that the algorithm has to actually go through for training to result in a trained model while trying to maintain similar performance and accuracy. *Approximate Computing* refers to a process where the actual computations in an algorithm are replaced by approximate ones which still is sufficient for its purpose [Mit16a], with sometimes a small trade-off in accuracy against a large gain in reduced computation time and resource utilization, which benefits algorithms that are destined to run on embedded platforms. The key point is to approximate in areas where the operations are relatively non critical which otherwise might result in unwanted consequences resulting from a faulty algorithm.

Traditionally, approximate computing is applied to four domains [Mit16b]. The first being approximating the circuits [Cho11] in the hardware on which the algorithm runs. These circuits usually refer to components such as adders, multipliers and other logical units. As an example, an approximate adder might discard the carry chain with multiple sub adders generating partial summation results [JHL15]. The second being approximating the storage method and resources used. This would involve for example the concept of *Approximate Memory Cells*[Sam+13], an attempt by NVIDIA research[3]. Usually memory cells have a built in redundancy for error corrections known as *Error-Correcting Code* (ECC) memory. Approximation in this area try to reduce the redundant cells by exposing errors that might occur due to missed writes or reads as controlled errors to the algorithm in a software level [Guo+16]. The third method involves software level approximations, where for example computations in the algorithm are approximated. This might also include operations with lower precision data types [Gup+15], or using statistical methods to determine similarities in operations and attempts to drop some of them. The fourth method involves approximating the entire system by means of a system level *Sensitivity analysis*[4]. This involves introducing errors in various parts of the entire system and observing the impact it has on the results of the algorithm. These results are compared against various performance metrics such as error magnitude, error significance, hamming distance etc.

We will focus our research into software level approximations. Generally approximating operations in machine learning algorithms such as *Deep Neural Networks*[5], where some of the parameters such as the *weights* can be randomly chosen and dropped [Son+19] is easier, which sometimes has additional beneficial results such as avoiding overfitting[6] [Sri+14]. [Xu+18] studies *Markov Sampling* as a viable technique to reduce the number of samples computed on the the incoming batch. In short, using the principles of *Markov Chains*[7], the algorithm is able to select samples that have a similar probability distribution as the samples it has already encountered. An interesting study on the use of the hyperplane distance in [LLW11] where the samples that lie close to the hyper plane are given more importance and are considered for training. [SHD17] is a study into applying clustering[8] techniques to reduce kernel computations [ZJ06], which can also then be applied to sub sample the input itself.

With all these points in mind we will proceed with three experiments on approximated incremental methods. The first drawing from research into [CGLQ15] using the samples that lie in a vicinity of the support vectors produced by previous batches. The second is based on considering only samples that are misclassified by the previously trained model and with the conclusions in [LLW11], a final algorithm combining the aspects of misclassification and points that lie near the hyperplane and its margins is implemented.

[3] https://www.nvidia.com/en-us/research/

[4] study of the uncertainty of the output of a mathematical process based on induced uncertainty at its input

[5] a neural network with more than two layers.

[6] a modelling error in machine learning where a model is biased towards the trained dataset and hence might perform badly in a general dataset.

[7] a model that can statistically describe the probability of an event occurring based only on information from previous and current events.

[8] a technique of grouping objects with a defined similarity measure

5.6 Input Approximation by K-Nearest Neighbours Clustering

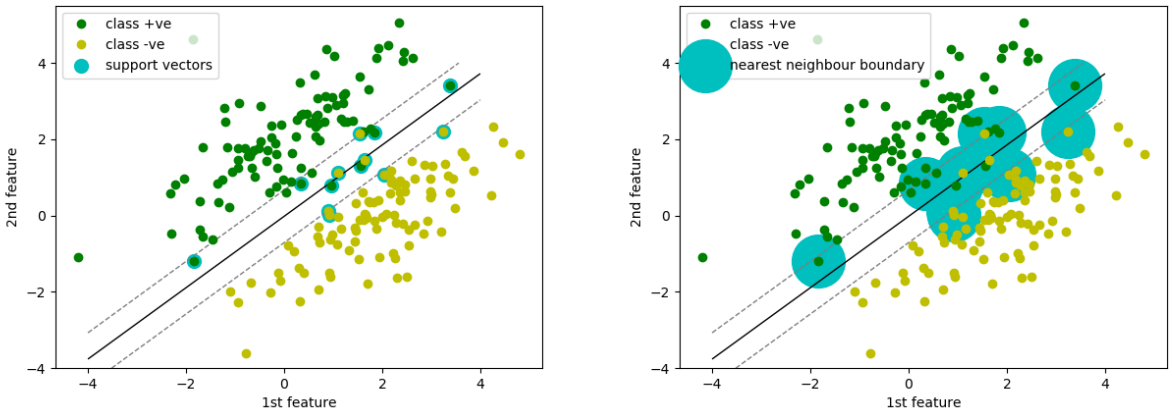
The points closest to the hyperplane and the margins are capable of altering an established decision boundary during to the training phase [R01]. Hence it is intuitive to find out which points in an incoming training batch lie close to the already established boundary and to subject these points to further training,

5.6.1 Pseudo Code

Algorithm 5 Incremental Multiclass SVM Nearest Neighbour Input Approximation

```
Result: trained model with one classifier per class
Input: samples, labels
Output: trained classifier
for number of batches do
    if batch number = 1 then
        Function train(training samples, labels):
            loop for number of training samples do
                loop for number of classes do
                    binarize labels                                  ▷ for multiclass classification
                    Function smo(training samples, binarized labels):
                        while true do
                            if optimality criteria met or maximum iterations reached then
                                break
                            end
                        end
                        return support vectors, lagrange multipliers
                end
            end
            return multi class classifier
    else
        Function Nearest Neighbour Approximator(training samples, support vectors):
            loop for number of support vectors do
                select k nearest neighbours from training samples
            end
            return nearest neighbours
        Function train(nearest neighbours, labels):
            loop for number of nearest neighbours do
                loop for number of classes do
                    binarize labels
                    Function smo(nearest neighbours, binarized labels):
                        while true do
                            if optimality criteria met or maximum iterations reached then
                                break
                            end
                        end
                        return support vectors, lagrange multipliers
                end
            end
        return updated multi class classifier
    end
end
```

instead of having to go through all the training samples. The *K-Nearest Neighbours* method, as the name suggests looks at k neighbours that lie nearest to a chosen point. The idea is that we extract the support vectors with the first batch of training samples that come in, and for the successive training batches we sub sample only the points that lie within a radius of x units as the neighbours. Following the pseudo code from 5, the first batch of 10,000 training samples are subjected to the SVM SMO (lines 2-15). The support vectors produced are stored (line 12) and before subjecting the second batch of training samples to the SVM training, a nearest neighbour approximation is undertaken (lines 17-21). For support vectors from each class, the k nearest neighbours to it form the new batch are computed. This can be visualized in figure 5.15b wherein k determines how many neighbours we are to consider for the search, essentially forming a radius around the support vectors from the previous batch. After this sub sampling of the training samples, the nearest neighbours are then subjected to the normal SVM training process (lines 22-24) resulting in an updated multi class classifier.



(a) Support Vectors produced from initial training batch

(b) Nearest neighbours to the support vectors

Figure 5.15: Represents the idea behind the nearest neighbour approximation method where 5.15a is the initial set of support vectors produced and 5.15b shows samples from the next batch that are neighbours to these previous support vectors.

The green points represent positive class and the yellow points represent the negative class. The samples chosen have two features with the first and second being marked on the x and y axes respectively. The solid black line represents the decision boundary and the dotted lines represent the positive and negative side margins. The encircled points represent the support vectors for the two classes. Figure 5.15b represents the support vectors with points in their nearest neighbour boundary in teal.

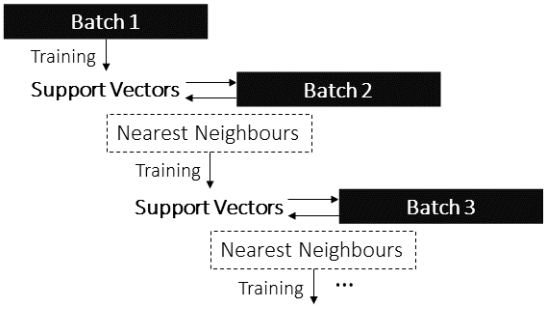


Figure 5.16: Incremental learning with nearest neighbour input approximation

5.6.2 Results

Figure 5.17 is a t-SNE dimensionality reduced visual representation of the decision boundaries formed for the 10 classes as a result of the training. The plot is comparatively sparse since only sub sampled training data is considered. With respect to the psedo code in algorithm 5, at the end of each optimized step for a class, a decision boundary is produced. Each boundary represents a classifier and the filled regions represent the contours for the bounds of that particular class. These plots are specially useful when we would like to infer the behaviour of the training process and its effect on the decision boundary. The color map represents labels from 0 to 9 respectively and their corresponding decision regions. The values along the x and y axes indicates two of t-SNE's dimensionality reduced components respectively to view the original 784 dimension dataset in a 2 dimensional space. The resultant of the t-SNE computation gives us values between -80 and +80 in both axes wherein our training samples lie.

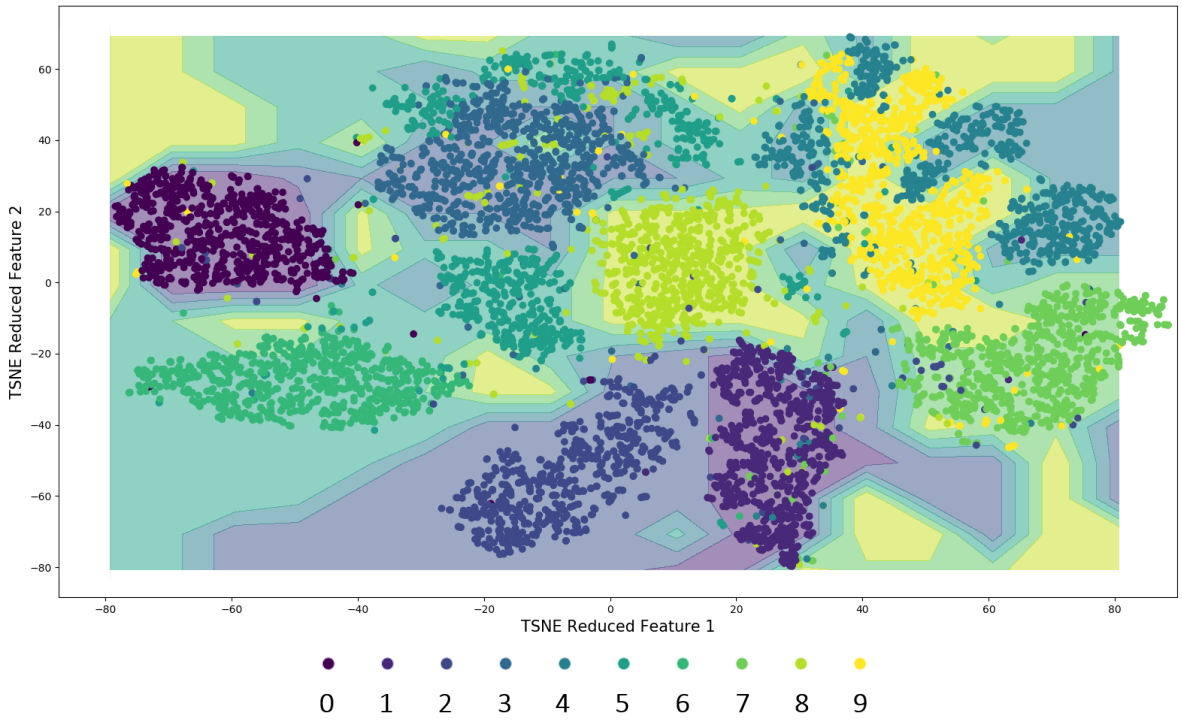
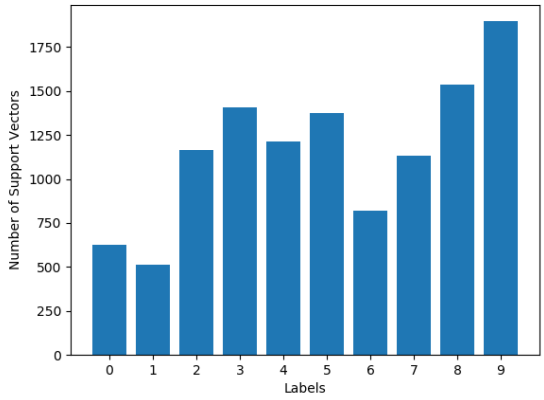
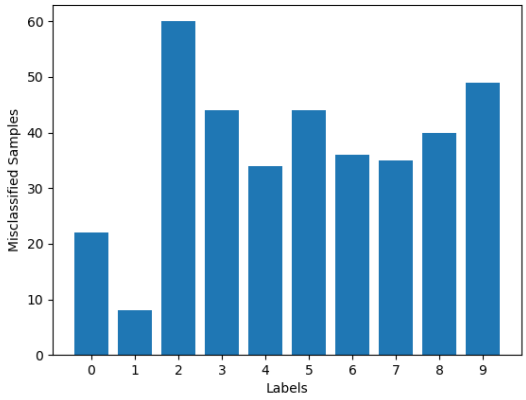


Figure 5.17: Decision boundary plot for the incremental algorithm with nearest neighbour approximation.

There is a dramatic decrease in the density of samples as a result of the Nearest Neighbour approximation which leads to decrease in runtime and computations with only the additional resource spent on computing all the nearest neighbours to the support vectors. This method also produces clear distinctive boundaries with quite a large margin. This is due to the fact that the algorithm produced a large number of support vectors, which helps in better definition of the decision boundaries of each class.
Figure 5.18a represents the spread of the number of support vectors produced per label at the end of the training phase.
Table 5.3a gives us the precision, recall and f1-scores of the results of the prediction on the test samples Table 5.3b represents the Confusion Matrix for the resulting prediction run on the test samples. In accordance with figure 5.18b labels 2 and 9 have a higher misclassification rate than the rest of the labels.
The algorithm performs with an accuracy of **96.28**%.



(a) Support vectors per label



(b) Misclassified test samples per label

Figure 5.18: Represents the result of the training and testing phases of the algorithm, where 5.18a indicated the total number of support vectors produced per label and 5.18b indicated the samples that were misclassified.

Labels	**precision**	**recall**	**f1-score**
0.0	0.97	0.98	0.98
1.0	0.97	0.99	0.98
2.0	0.98	0.94	0.96
3.0	0.96	0.96	0.96
4.0	0.96	0.97	0.96
5.0	0.97	0.95	0.96
6.0	0.97	0.96	0.97
7.0	0.95	0.97	0.96
8.0	0.96	0.96	0.96
9.0	0.93	0.95	0.94

(a) Classification report

	0	1	2	3	4	5	6	7	8	9
0	958	2	1	2	3	2	3	1	1	7
1	0	1127	2	1	1	1	2	0	1	0
2	8	7	972	12	3	0	4	13	11	2
3	0	2	4	966	1	9	0	12	3	13
4	1	2	1	1	948	1	5	2	5	16
5	3	4	0	11	7	848	7	3	2	7
6	9	10	0	1	6	7	922	0	3	0
7	0	5	8	1	2	0	0	993	3	16
8	1	1	1	9	5	7	2	6	934	8
9	3	4	0	5	14	3	2	13	5	960

(b) Confusion matrix

Table 5.3: Represents the classification report in 5.3a and confusion matrix in 5.3b for the classification results.

This method has the disadvantage that if the initial training batch does not contain samples from all classes or has an uneven distribution of support vectors for some classes in the first batch, then the consequent samples chosen according to the nearest neighbour method will result in a bias toward a particular class or label. The distribution of samples and support vectors after training on the first batch of data determines the course of the training algorithm and the final model produced. Since training an SVM in batches involves updating the decision boundary accordingly, we will carry forward the concept that the points closest to the hyperplane and margins are capable of altering the decision boundary to the next implementations.

The other observation was the high number of support vectors as evident from figure 5.18a that were generated. This is not good when trying to implement this algorithm on the PYNQ as previously discussed, the device is resticted in memory and upon running the algorithm, the system again crashed with the memory over flow problem. But following this nearest neighbour concept we formulated the final method for the software implementation, wherein instead of nearest neighbours to the support vectors, we consider the actual distances of the sample points from the support vectors to form a region similar to a second margin. Points from these region were then considered for training. This is discussed in section 5.8.

5.7 Input Approximation using Misclassified Samples

This approach involved dealing only with samples that we are unable to classify correctly. If the model is able to classify current samples correctly using the support vectors and lagrange multipliers generated from the previous batch, then these samples need not be retrained. We instead focus on samples that the algorithm has misclassified.

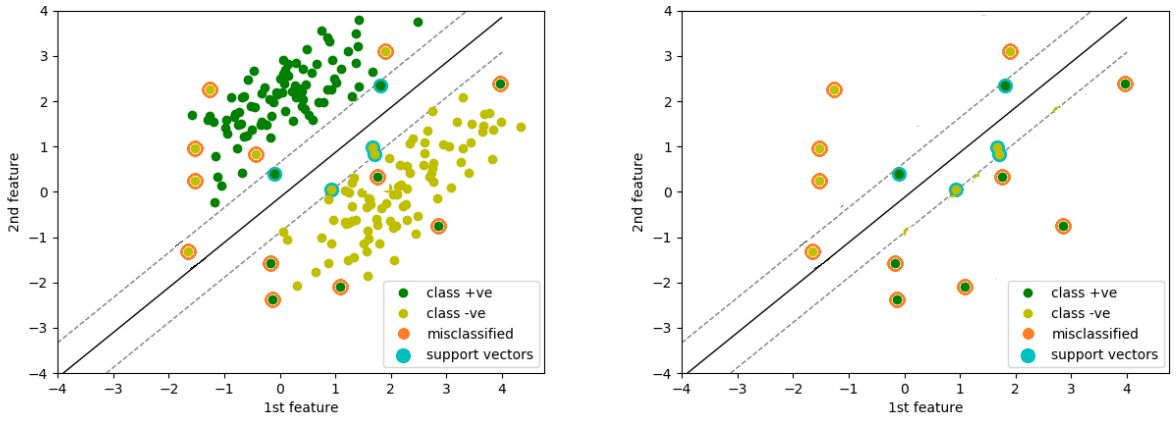
Algorithm 6 Incremental Multiclass SVM with Input approximation by Misclassified samples

Result: trained model with one classifier per class
Input: samples, labels
Output: trained classifier

```
for number of batches do
    if batch number = 1 then
        Function train(training samples, labels):
            for number of training samples do
                for number of classes do
                    binarize labels                                  ▷ for multiclass classification
                     Function smo(training samples, binarized labels):
                        while true do
                            if optimality criteria met or maximum iterations reached then
                                break
                            end
                        end
                        return support vectors, lagrange multipliers
                end
            end
            return multi class classifier
    else
        for number of training samples do
            Function Predict(training samples):
                return predicted label
            if predicted label ≠ actual label then
                misclassified sample = training sample
            end
        end
        Function train(misclassified sampless, labels):
            for number of misclassified samples do
                for number of classes do
                    binarize labels
                     Function smo(misclassified samples, binarized labels):
                        while true do
                            if optimality criteria met or maximum iterations reached then
                                break
                            end
                        end
                        return support vectors, lagrange multipliers
                end
            end
        return updated multi class classifier
    end
end
```

This technique is a form of *windowing technique* used commonly in *active learning*[9] where the algorithm decides on which samples to actually train upon [JB14].

Following the pseudo code in 6, The first 10,000 training samples are subjected to the SVM SMO training (lines 2-15) and the resulting model in the form of classifiers for each label is saved. On the next training sample batch, the algorithm first predicts the labels of the received samples (lines 17-19). Upon completion of the prediction phase, the samples that we incorrectly predicted or misclassified (lines 20-22) are saved from amongst the entire training batch. Only these samples are then sent to training (lines 24-37). This is repeated for all other batches. The process is shown in figure 5.20.



(a) Misclassified samples in a batch

(b) Misclassified samples filtered for training

Figure 5.19: Represents the concept of the incremental algorithm with misclassified sample approximation where figure 5.19a denoted in orange the samples that were misclassified and figure 5.19b shows the sparsity of them being filtered out.

To visualize this concept with an example, 5.15a represents the result of the SVM SMO training for the first batch of data. The green points represent positive class and the yellow points represent the negative class. The samples chosen have two features with the first and second being marked on the x and y axes respectively. The solid black line represents the decision boundary and the dotted lines represent the positive and negative side margins. The points encircled in orange represent samples that are misclassified in the current batch while the remaining training samples are discarded. Figure 5.15b represents the now sparse batch of training samples and the resultant support vectors. This results in a very sparse training set for the subsequent training phases. This process is repeated for the remaining n-1 batches.

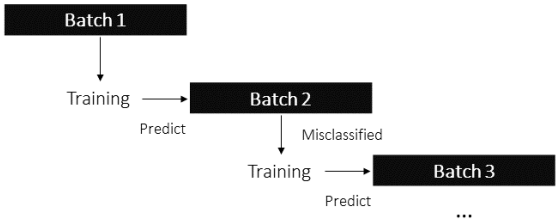


Figure 5.20: Incremental learning with input approximation using misclassified samples

[9] http://www-ai.cs.tu-dortmund.de/DOKUMENTE/JAIR/volume8/fuernkranz98a-html/node25.html.gz

5.7.1 Results

Figure 5.21 is a tSNE dimensionality reduced visual representation of the decision boundaries formed for the 10 classes as a result of the training. The plot is comparatively sparse since only sub sampled training data is considered. With respect to the psedo code in algorithm 4, at the end of each optimized step fora class, a decision boundary is produced. Each boundary represents a classifier and the filled regions represent the contours for the bounds of that particular class. These plots are specially useful when we would like to infer the behaviour of the training process and its effect on the decision boundary. The colormap represents labels from 0 to 9 respectively and their corresponding decision regions. The values along the x and y axes indicates two of tSNE's dimensionality reduced components respectively to view the original 784 dimension dataset in a 2 dimensional space. The resultant of the t-SNE computation gives us values between -65 and +80 in both axes wherein our training samples lie.

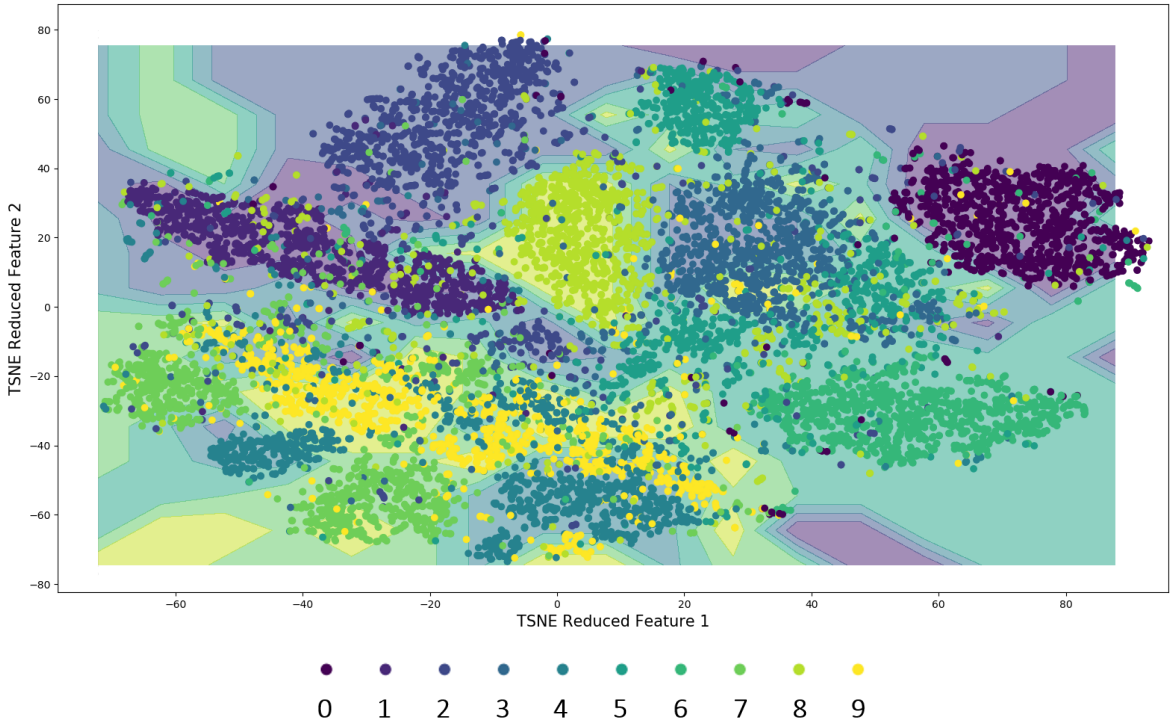
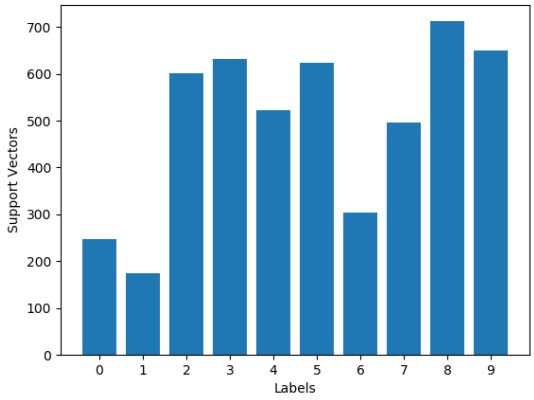


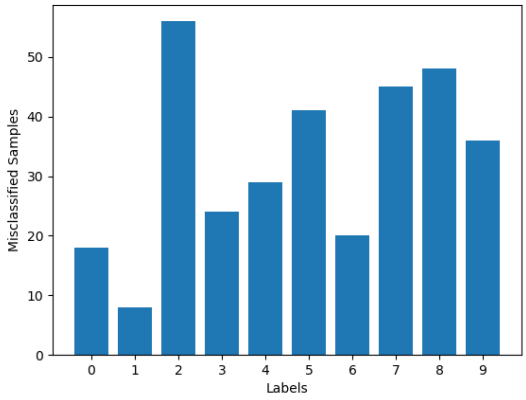
Figure 5.21: Decision boundary plot for the incremental learning with input approximation using misclassified algorithm

From the figure 5.21 we immediately notice a very sparse distribution of training samples. This is attributed to the fact that the entire set of training samples consists of the initial 10,000 samples followed by only the misclassified samples from the consecutive batches

Figure 5.22a represents the number of support vectors produced per label after the entirety of the training process. Labels 3 and 8 are not that easily separable when referring figure 5.9a, and hence a lot more support vectors are needed. But in comparison to the previous methods, very few support vectors are generated. Figure 5.22b represents a plot of the number of misclassified samples per label for the batch of test samples. Table 5.4a represents the precision, recall and f1-scores as a result of the prediction phase on the test samples. Table 5.4b represents the confusion matrix for prediction on the test samples. The algorithm performs with an accuracy of **96.75**%.



(a) Support vectors per label



(b) Misclassified test samples per label

Figure 5.22: Represents the result of the training and testing phases of the algorithm, where 5.22a indicated the total number of support vectors produced per label and 5.22b indicated the samples that were misclassified.

Labels	**precision**	**recall**	**f1-score**
0.0	0.98	0.98	0.98
1.0	0.94	0.99	0.97
2.0	0.99	0.95	0.97
3.0	0.96	0.98	0.97
4.0	0.98	0.97	0.97
5.0	0.97	0.95	0.96
6.0	0.98	0.98	0.98
7.0	0.97	0.96	0.96
8.0	0.98	0.95	0.96
9.0	0.93	0.96	0.95

(a) Classification report

	0	1	2	3	4	5	6	7	8	9
0	962	4	0	0	0	3	6	2	1	2
1	1	1127	1	1	0	0	0	3	2	0
2	9	9	976	12	3	0	0	12	9	2
3	0	2	1	986	0	6	0	4	4	7
4	1	4	2	0	953	0	2	1	0	19
5	2	6	0	12	1	851	8	1	2	9
6	5	4	0	0	5	3	938	0	3	0
7	0	16	7	2	2	0	0	983	0	18
8	0	12	1	8	2	8	2	3	926	12
9	1	11	0	3	11	3	1	5	1	973

(b) Confusion matrix

Table 5.4: Represents the classification report in figure 5.4a and confusion matrix in figure 5.4b for the classification results.

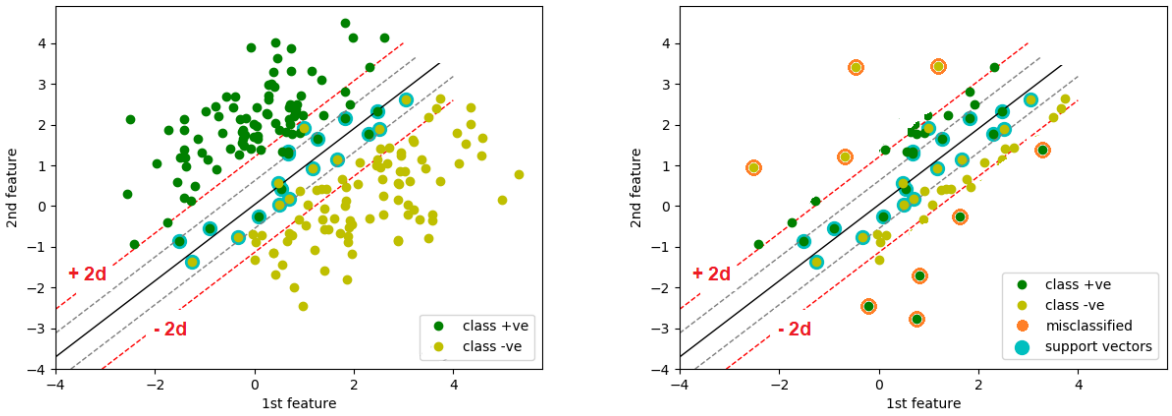
5.8 Input Approximation using Misclassified and Marginal Points

Algorithm 7 Incremental SVM with Input Approximation by Misclassified and Marginal Points

Result: trained model with one classifier per class
Input: samples, labels
Output: trained classifier

```
for number of batches do
    if batch number = 1 then
        Function train(training samples, labels):
            for number of training samples do
                for number of classes do
                    binarize labels                                  ▷ for multiclass classification
                    Function smo(training samples, binarized labels):
                        while true do
                            if optimality criteria met or maximum iterations reached then
                                break
                            end
                        end
                        return support vectors, lagrange multipliers
                end
            end
            return multi class classifier
    else
        for number of training samples do
            Function Predict(training samples):
                return predicted label
            if predicted label ≠ actual label then
                misclassified sample = training sample
            end
            Function calculate marginal distance(support vectors, lagrange multipliers):
                return marginal distance
            if training sample between marginal distance and 2 * marginal distance then
                marginal sample = training sample
            end
        end
        filtered samples = misclassified samples and marginal samples
        Function train(filtered samples, labels):
            for number of filtered samples do
                for number of classes do
                    binarize labels
                    Function smo(filtered samples, binarized labels):
                        while true do
                            if optimality criteria met or maximum iterations reached then
                                break
                            end
                        end
                        return support vectors, lagrange multipliers
                end
            end
        return updated multi class classifier
    end
end
```

This algorithm is an extension of algorithm 5.7. Instead of only considering the misclassified points, we also consider points that lie in a region of width 2 $*$ the margin from the decision boundary. This is backed by the fact that was established that points that lie close to the hyperplane and the margin are capable of altering the decision boundary and hence are interesting to be subjected to training [LLW11]. Figures 5.23a and 5.23b represents the result of the SVM SMO training for the first batch and second batch of data. The green points represent positive class and the cyan points represent the negative class with their first and second features on the x and y axes respectively. The training process starts off as



(a) Misclassified and Marginal Points

(b) Misclassified and Marginal points 2

Figure 5.23: Represents the idea behind the algorithm where figure 5.23a represents the support vectors and its defining boundary, and the additional 2 * margin. Figure 5.23b represents the filtering.

usual on the first 10,000 samples and the resultant support vectors and their lagrange multipliers are saved. The support vectors in every class occur in pairs, which makes sense since we need a a pair of support vectors to define the two sides of the margin. These pair of support vectors have a corresponding pair of lagrange multipliers. Every pair of support vectors and their lagrange multipliers are bifurcated into a positive class and negative class defined by their sign. These can be seen in figure 5.23a. The solid line represents the decision boundary and the upper dashed line represents the positive half of the margin and the lower dashed line represents the negative half of the margin, which are actually supported by the positive class support vectors and negative class support vectors respectively. By pairing up the positive support vectors and the positive lagrange multipliers we can recreate the positive side of the margin and vice versa for the negative support vectors and negative coefficients. We first determine the average distance of the support vectors of every class from the decision boundary. For any new incoming training samples, we calculate its distance from the hyperplane modeled by the support vectors of the previous batch. We then state a new boundary which lies in the region between $\boldsymbol{2d}$ and $\boldsymbol{d}$ where $\boldsymbol{d}$ is the distance of the support vector from the decision boundary or hyperplane (lines 23-27)
If the sample's distance lies within this margin, we consider this sample too for training. This is repeated for all classes and their respective support vectors. These can be seen in figure 5.23b where in samples that lie bounded by the margins marked in red dashed line at a distance of 2 * $\mathbf{d}$ from the hyperplane are considered for training (line 26). In addition to this, we predict the incoming samples with the model from training the previous batch. amongst these, we consider only the samples that were misclassified (lines 18-22). Now the training samples only contain these misclassified samples and the samples that fall in the aforementioned extended marginal region. Rest of the samples are discarded.

5.8.1 Results

Figure 5.24 shows the plot for the decision boundary, contours and the training samples as of the result of the algorithm. We can infer from the plot the following. It is immediately noticeable from the plot that the sample distribution is somewhere between sparse and dense. This is due to the fact that for one, the algorithm filters out samples in the training batch that are classified correctly. This makes the samples sparse. In addition to this the algorithm also considers samples that lie in the newly defined 2 x bounded marginal region. This in turn makes the samples denser in comparison. The combination of this results in an in between density of the samples. The clusters for individual labels are well defined and very well distinguishable from each other. The decision boundary and the decision regions are prominent unlike the remainder of the approximate methods discussed earlier.

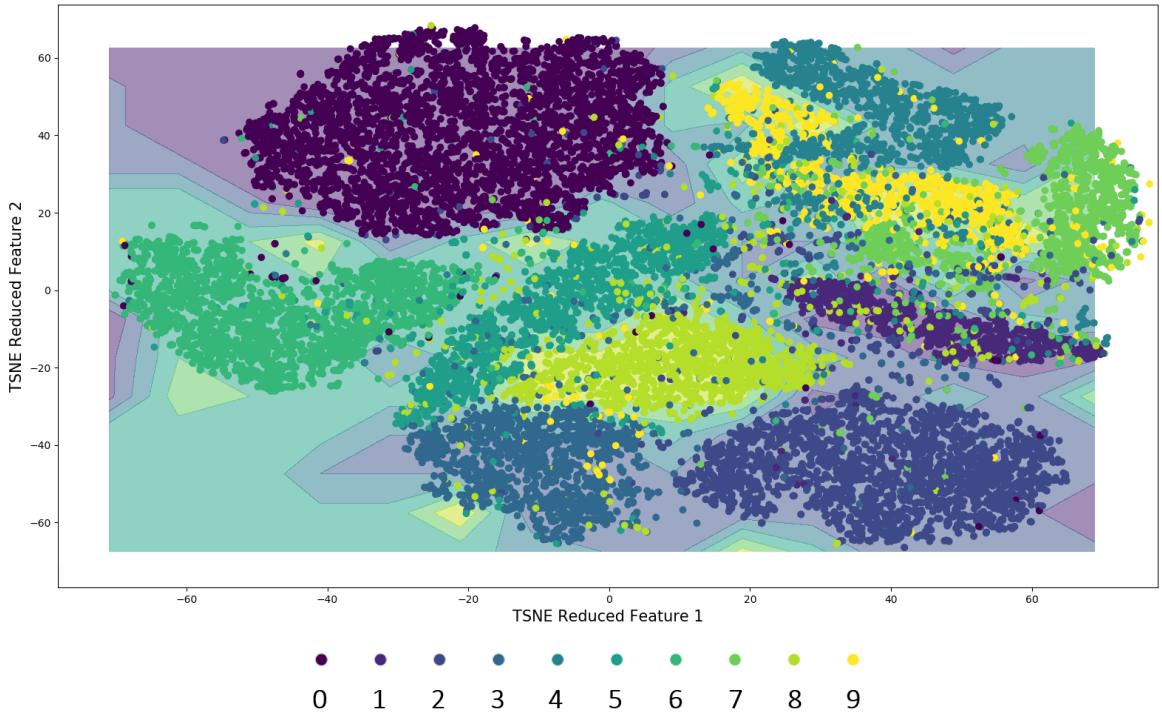
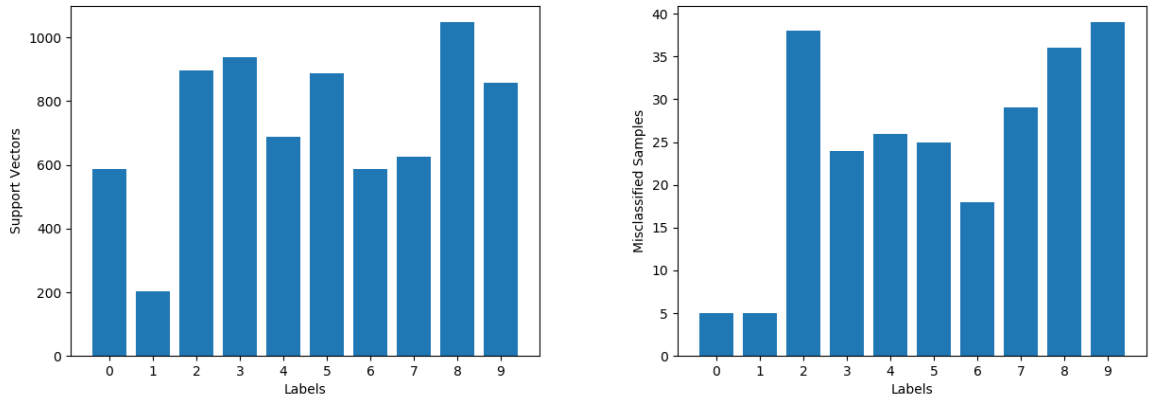


Figure 5.24: Decision boundary plot for the incremental algorithm with input approximation by misclassified and marginal points algorithm.

Figure 5.24 color map represents labels from 0 to 9 respectively and their corresponding decision regions. The values along the x and y axes indicates two of t-SNE's dimensionality reduced components respectively to view the original 784 dimension dataset in a 2 dimensional space. The resultant of the t-SNE computation gives us values between -80 and +80 in both axes wherein our training samples lie.

Figure 5.25a represents the final number of support vectors generated upon completion of training phase. This hold true for explanations in previous implementations for labels that are easily separable and those that are not. Figure 5.25b represents the number of samples per label that were misclassified in the test set.



(a) Support vectors per label

(b) Misclassified test samples per label

Figure 5.25: Represents the result of the training and testing phases of the algorithm, where 5.25a indicated the total number of support vectors produced per label and 5.25b indicated the samples that were misclassified.

Table 5.5a represents the classification report for the testing phase of the algorithm and table 5.5b represents the confusion matrix for the same.

Labels	precision	recall	f1-score
0.0	0.97	0.99	0.98
1.0	0.96	1.00	0.98
2.0	0.99	0.96	0.98
3.0	0.97	0.98	0.98
4.0	0.98	0.97	0.98
5.0	0.98	0.97	0.98
6.0	0.99	0.98	0.98
7.0	0.97	0.97	0.97
8.0	0.98	0.96	0.97
9.0	0.96	0.96	0.96

(a) Classification report

	0	1	2	3	4	5	6	7	8	9
0	975	1	0	0	0	0	1	0	1	2
1	0	1130	1	1	0	0	0	2	1	0
2	8	5	994	4	1	0	1	8	9	2
3	2	1	1	986	0	8	0	4	4	4
4	1	6	2	0	956	0	4	2	0	11
5	3	3	0	8	0	867	4	1	2	4
6	6	8	0	0	2	2	940	0	0	0
7	0	8	6	0	4	0	0	999	0	11
8	4	6	2	8	1	5	3	3	938	4
9	5	7	0	5	10	3	1	6	2	970

(b) Confusion matrix

Table 5.5: Represents the classification report in 5.5a and confusion matrix in 5.5b for the classification results.

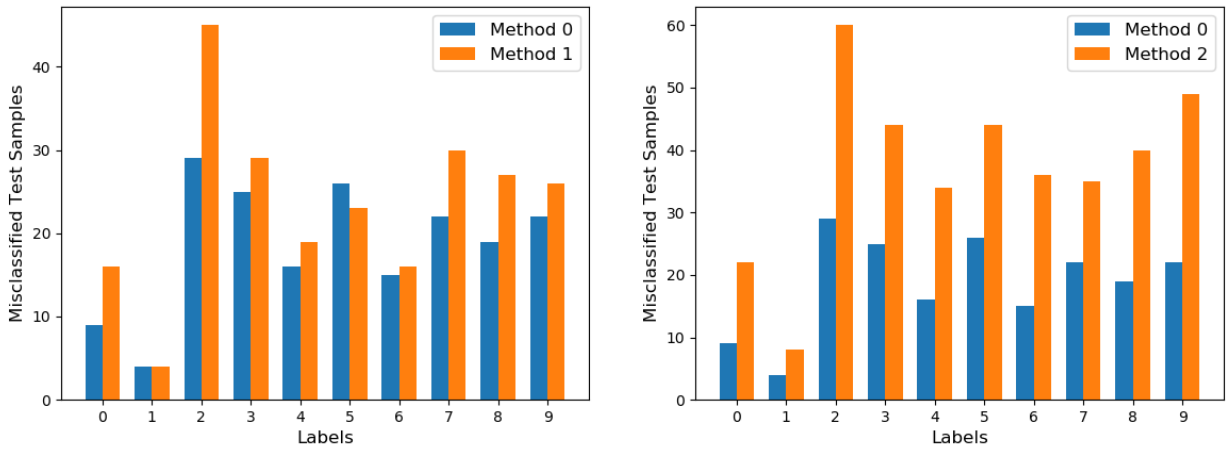
The algorithm performs with an accuracy of **97.55**%, highest among the incremental and approximated techniques.

5.9 Non-Incremental vs Incremental Approximated Algorithms

A comparison is made pairing each of the 4 incremental and approximate SVM algorithms with the non incremental algorithm since the non incremental algorithm was considered as the base metric for performance measurement. The naming convention followed here is as follows

Method 0	Non-Incremental SVM
Method 1	Incremental SVM with Support Vector Forwarding
Method 2	Incremental SVM with Input Approximation by Nearest Neighbours
Method 3	Incremental SVM with Input Approximation by Misclassified Samples
Method 4	Incremental SVM with Input Approximation by Misclassified and Marginal Points

The main metric for comparison of the different incremental algorithms was its closeness to the classification accuracy of the non incremental method. Along with this a measure of the number of missclassified samples also considered.



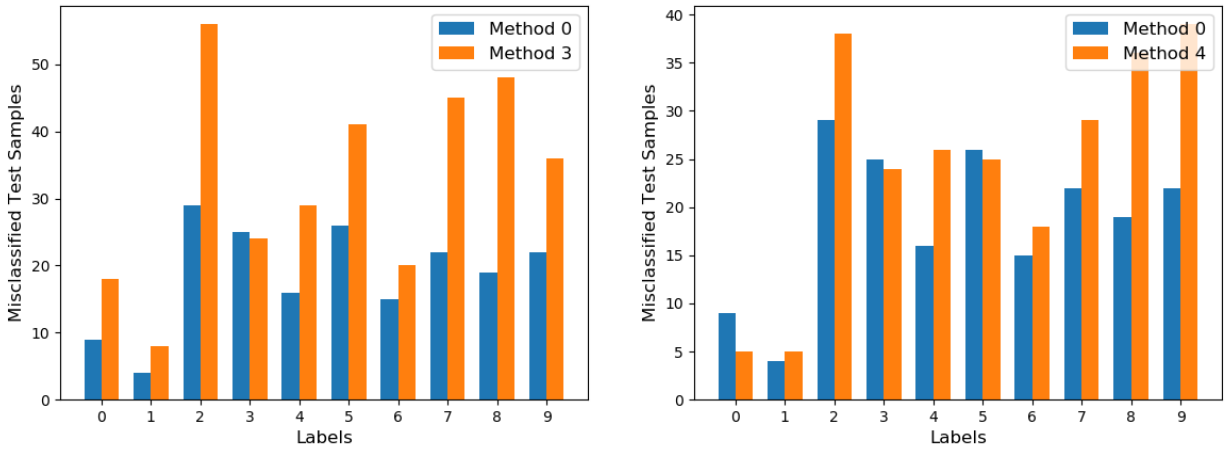
(a) Misclassified samples vs labels

(b) Misclassified samples vs labels

Figure 5.26: Represents the comparisions between methods where figure 5.27a represents for method 0 and 1 figure 5.27b is for method 0 and 2

Figure 5.26 represents the comparison in the number of missclassified test samples during the prediction phase for methods 0 and 1, followed by similar plot for method 0 and method 2 with the differences visible by the scale of the y axes
Figure 5.27 represents the comparison in the number of missclassified test samples during the prediction phase for methods 0 and 3, followed by similar plot for method 0 and method 4 with the differences visible by the scale of the y axes.

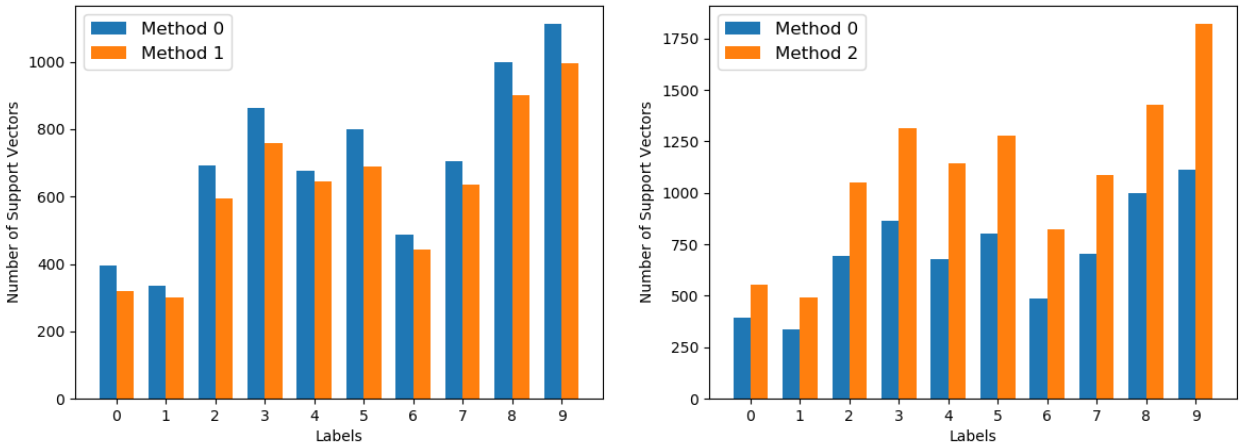
As an observation, the number of samples per label that were missclassified by the incremental and approximated algorithms are quite similar to those misclassified by the non incremental algorithm. This validates the approach of making sure that the incremental algorithms perform similar to the non incremental algorithm in terms of classification. Among the incremental algorithms (methods 1-4), with the comparison on the scale of the y axes, method 4 performs the most similar to method 0 with the lowest number of missclassified samples which in turn lead to an accuracy of 97.55%.
Figure 5.28b represents the comparison in the number of support vectors produced against labels during the training phase for methods 0 and 1, followed by similar plot for method 0 and method 2 with the differences visible by the scale of the y axes. Figure 5.29a represents the comparison in the number of support vectors produced against labels during the training phase for methods 0 and 3, followed by similar plot for method 0 and method 4 with the differences visible by the scale of the y axes.



(a) Misclassified samples vs labels

(b) Misclassified samples vs labels

Figure 5.27: Represents the comparisions between methods where figure 5.27a represents for method 0 and 3 figure 5.27b is for method 0 and 4



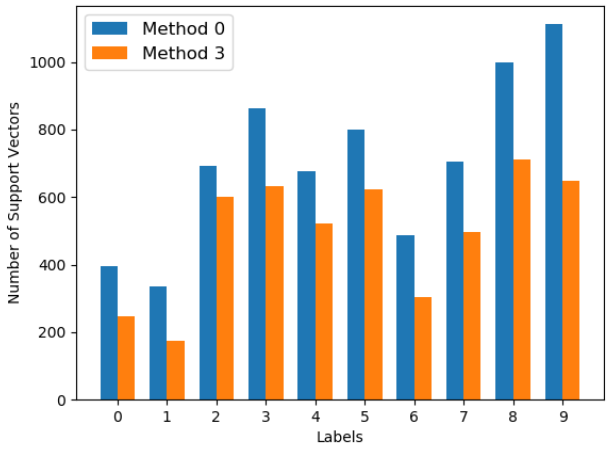
(a) Number of support vectors vs labels

(b) Number of support vectors vs labels

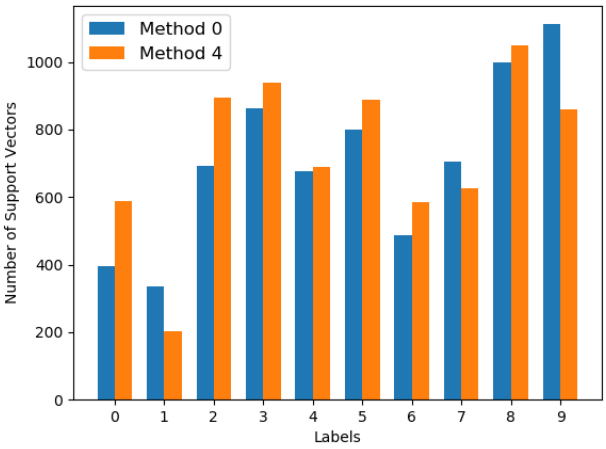
Figure 5.28: Represents the comparisons between methods where figure 5.28a represents for method 0 and 1 figure 5.28b is for method 0 and 2

The number of support vectors per label produced by two algorithms are not a measure of similarity in their performance. But in order to take into account the resource budget of the embedded system on which the algorithm is going to be deployed on, it is necessary to pick the one with the least number of support vectors produced and also, the prediction accuracy matches that of the non incremental algorithm. With these two factors, we can notice that even though method 3 produces the least number of support vectors, method 4 produces the highest classification accuracy with a relatively lower number of support vectors. Hence method 4 is chosen.

In an effort to compare if the support vectors are a sufficient set of samples to define the original set of training samples, a comparison is made between the support vectors produced by the incremental



(a) Number of support vectors vs labels



(b) Number of support vectors vs labels

Figure 5.29: Represents the comparisons between methods where figure 5.29a represents for method 0 and 3 figure 5.29b is for method 0 and 4

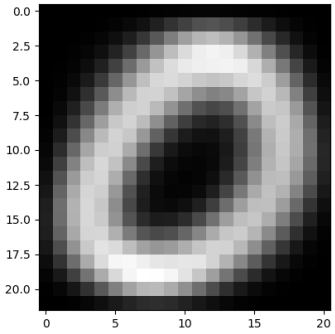


Figure 5.30: Training samples

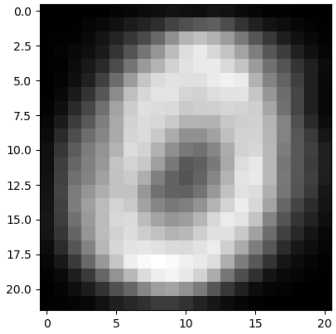
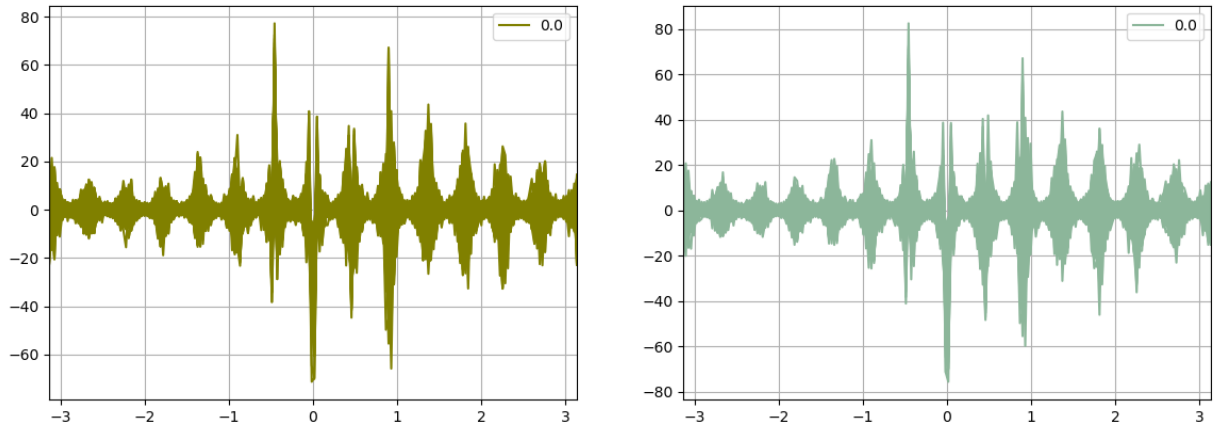


Figure 5.31: Support vectors

algorithm and the samples it was trained on. This is illustrated as an example in figures 5.30 and 5.31 for the label 0. These figures are plots of the images averaged over all samples for label 0. It is noticeable that the support vectors do resemble the training samples. The resemblance of the plot of the averaged support vectors to the original training labels indicates that, support vectors which were picked are unique enough to represent all different kinds of samples for the label 0 as evident in the figure for a subset of samples from the MNIST data previously.

With reference to the evaluation techniques mentioned in chapter on evaluation, figures 5.32a and 5.32b represent comparison of the Andrew's plots of the average of the training samples and the average of support vectors produced by the algorithm for the label 0. Again, it is observed that the averages are similar strengthening our inference that the set of support vectors produced by the training of the algorithm is indeed a sufficient set for representation of all the samples of the training set.



(a) Andrew plot of training samples

(b) Andrew plot of support vectors

Figure 5.32: Represents the Andrews plot for a comparison of samples from 5.32a for training samples and 5.32b for support vectors.

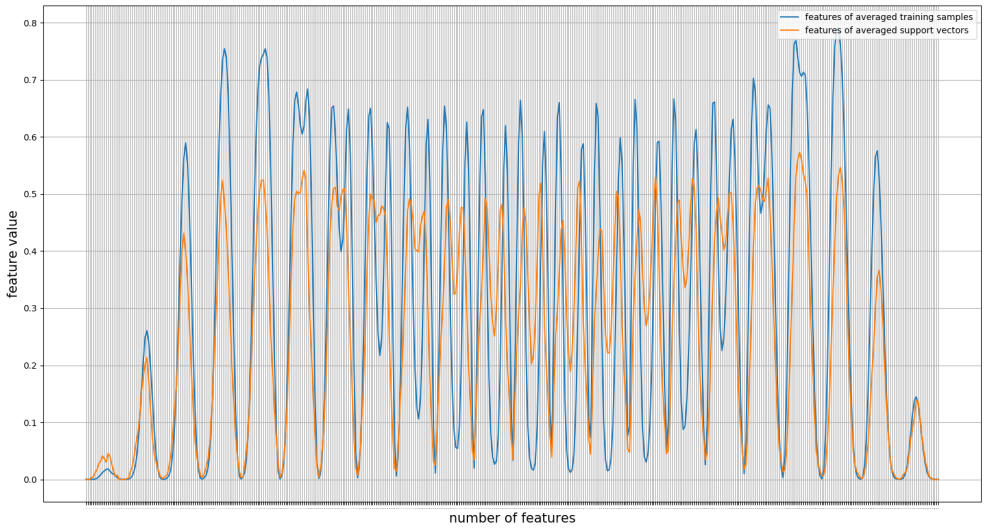


Figure 5.33: Feature Average plot of Support Vectors vs Misclassified Samples For Label 0

With respect to the third visualization method under evaluations, figure 5.33 represents the comparison between a parallel coordinate plot for the average of the support vectors produced by the algorithm and the average of the training samples for the label 0. The similarity between both plot again validates our previous inference.

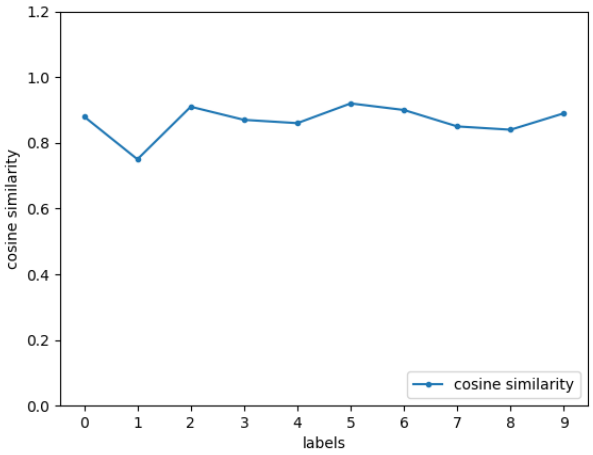
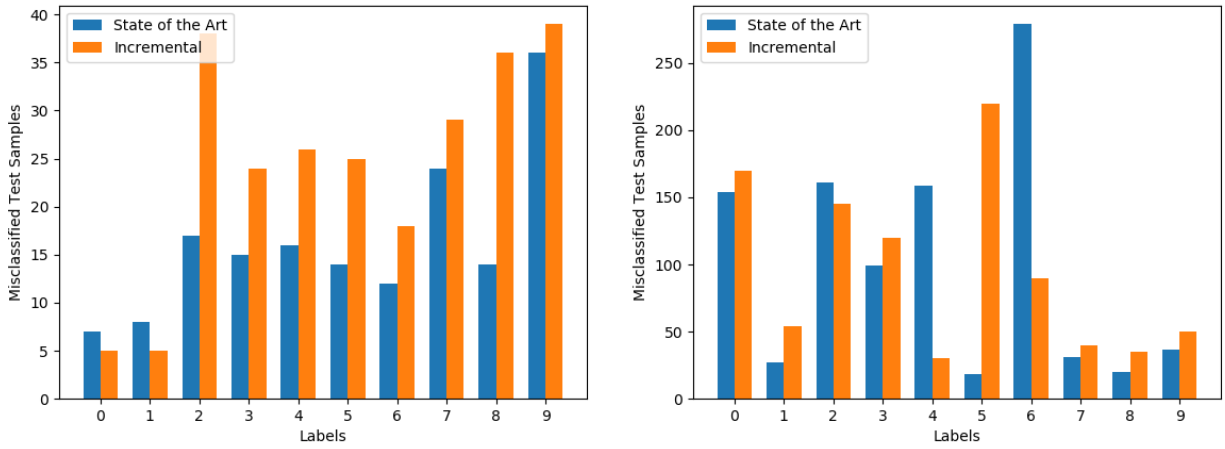


Figure 5.34: Cosine similarity between support vectors produced and training samples

With respect to the statistical evaluation method discussed earlier, the figure 5.34 represents a plot of the cosine similarity between the averages of the training samples and the support vectors produced by the algorithm for all the labels. This is a measure of the similarity between the support vectors produced by the algorithm and the training samples it was trained on for **all** the labels. Following our inference for label 0 from the previous methods, it can be generalized for all the labels that the support vectors produced by the algorithm for every label are an optimum and unique set of sufficient examples that are able to describe their respective counterparts in the training samples.

5.10 Incremental Approximated Algorithm vs State of the art



(a) misclassified labels for MNIST

(b) misclassified labels for Fashion MNIST

In order to validate our implemented algorithm against the state of the art implementation of an SVM classifier by scikitlearn, the figures 5.35a show a comparison of the classification performance for the testing phase for an additional data set apart from the MNIST. The number of misclassified samples per label between the two implementations are similar which demonstrate that our implementation of the incremental algorithm performs nearly as well as the state of the art. The incremental achieves 88% accuracy for the Fashion MNIST and the state of the art achieves 90.5%.

6

Hardware Implementation

6.1 Background study on Hardware Accelerators

We have seen that ML algorithms operate on vast amounts of data and involve complex computations. With the growing number of use cases of ML and its ability to solve real world problems, there is a rising need for hardware to support this. Traditional GPU and ASICs fail to meet the resource and power requirements of embedded systems which pave the way for FPGAs to be used as hardware accelerators. The earliest attempts at accelerating machine learning algorithms such as neural networks can be seen in [**ocr**], wherein digital signal processors were used for the purpose of dedicated computations in *Optical Character Recognition* systems. In the advent of simple *Neural Networks*, parallel processing systems were used to simulate these neural circuits.

FPGAs are *Reconfigurable* systems known for their parallel processing capabilities and often preferred over traditional GPUS in applications such as computer vision [Wie+12], image processing [TTY08], digital signal processing etc., to name a few. FPGAs as hardware accelerators for SVMs have have been researched extensively due to the relatively simple mathematical principles of the SVM which allow it to be easily translated to code, and the computations involved in the algorithm to be converted to parallel structures which FPGAs deal well with. These involve two types of hardware accelerations, training phase acceleration and prediction phase acceleration [AGS15]. Traditionally acceleration involved only the prediction phase wherein a trained model is available and saved and the model is then offloaded on to a FPGA to accelerate predictive operations which usually involve operations such as multiplications. Examples of these include Facial expression recognition on the Virtex-6 class FPGA [Pat+12]. The Xilinx CORDIC[1] IP core is dedicated Ip core for performing complex arithmetic operations specially developed for ML and statistics is a widely used in [Gom+10], [And01] and [GSLJ10].

FPGAs for training phase acceleration are pretty scarce as researched by [Mit16b], with the most notable being [ANI13], [Kua+12], [RB15] which, not surprisingly, utilized the SVM SMO type decomposition as discussed in the previous chapter. These were performed on Xilinx Vitex and Kintex family FPGA. It is interesting to note these classes of FPGAs used were powerful and expensive that do facilitate a higher resource budget. Such implementations on small, low cost FPGAs are scarce [Mit16b] especially with regard to training acceleartion of SVMs.

We will proceed with the algorithm that employs input approximation using misclassified and marginal samples 5.8 for hardware acceleration consideration on the PYNQ board.

[1] https://www.xilinx.com/support/documentation/ip$_d$*ocumentation/cordic/v*6$_0$

6.2 The Hardware Platform

The PYNQ Z1[2] board was used for this experiment. The board packs a 650 MHZ dual core ARM processor with an underlying Xilinx Zynq 7000 series FPGA. This board is particularly chosen because it is a combination of an FPGA and CPU system with a Linux Kernel running the Jupyter notebook platform on the ARM core. The Jupyter notebook can be used to interact with the CPU and FPGA specifically by means of direct programming in Python and C++. This lets us call the FPGA components or IP cores directly from software as one would import a standard machine learning library. This PYNQ board is an efficient representation of a small scale embedded SoC which can be used by industries for in house incremental ML.

6.3 Hardware Accelerator Architecture

The functions from the software implementation that were chosen to be accelerated in the FPGA are the *Kernel Matrix Computation function* and the *Prediction* function which were found to be the two most computationally intensive functions. The effects of having large number of samples and the complexity of the kernel computations on them are discussed in chapter 2.18. Since our algorithm involves a prediction phase inside the training phase as noticeable in algorithm 7, the Predict function was also considered to be accelerated.

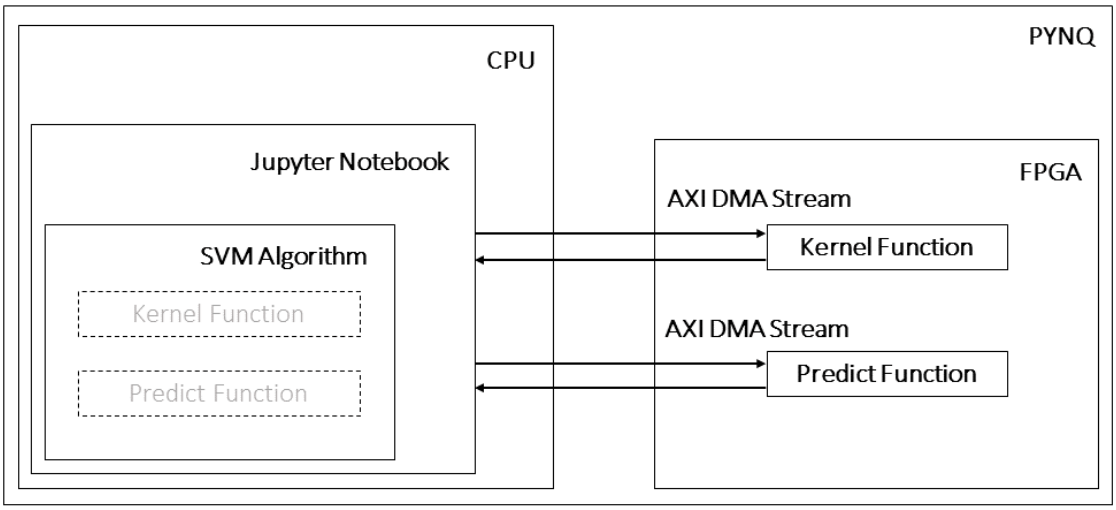


Figure 6.1: Hardware Accelerator Architecture

The figure 6.1 represents an overview of the accelerator architecture design on the PYNQ board. The two functions that are to be accelerated are shown to be replaced from the software implementation on the CPU to the FPGA as a hardware implementation. Since the main aim of the thesis was to enable learning with large amounts of dats and also since our data sets contain a fairly large amount of samples to facilitate movement of data between the CPU and FPGA for computations, the functions as IP cores on the FPGA were implemented to utilize the AXI DMA stream communication protocol. This protocol facilitates direct memory transfer between the ARM core and the FPGA via a memory mapped interface to minimize the latency in communication as much as possible.

[2]http://www.pynq.io/

6.3.1 Hardware Optimization

With the intention of accelerating the execution time of our algorithm, we proceed to first have a baseline implementation of the algorithm in Vivado HLS C++ and compared its execution time to the software only algorithm running on the CPU. The hardware implemented functions were found to perform similarly in execution time as their CPU implemented software counterpart. We then proceeded to modify the C++ code in such a way that various HLS pragmas , which are compiler directives for the Vivado HLS to try and optimize the hardware implementation. The following pragmas were then introduced in appropriate locations of the code. A brief overview of the pragmas is as follows:

DATAFLOW: This pragma allows us to overlap the computations between for loops and functions. Since our our functions were implemented in such a manner, it was possible to apply this pragma to attain task level pipelining.

LOOP UNROLL: This pragma deconstructs a loop structure into its individual components which is similar to dissolving a for loop. For loops in general operate in sequential computations on a CPU but the unrolled version is no longer a for loop and hence can be parallelised by the hardware implementation.

PIPELINE: This pragma lets us overlap computations that might originally be sequential but independent of each other. Our functions contained different parts such as streaming in the samples and computations on them which are viable candidates for pipelining since the computations would not have to wait until the previous one has completed.

INLINE: This pragma is straightforward in the sense that it replaces the function call at the call site by its definition. This is similar to dissolving a function and replacing it in the original call location. This in turn reduces latency in communication overheads arising due to multiple function calls. As an example, the Kernel matrix computation function calls an RBF Kernel function multiple times and hence results in a latency for function calls. This function was a viable candidate for inlining.

RESOURCE: With a restricted LUT and BRAM budget, the hardware implementation had to be tailored to allow maximum number of samples to be streamed in and operated on. Since our functions contained operations that involved independent streams and hence independent storage elements, implementing all of them on the BRAM tended to exceed the resource budget. In order to avoid this, the option was to either reduce the number of streamed in samples for the respective input arrays or usage of the resource pragma to allocate some input storage on the BRAM and the remaining implemented as a LUT RAM. This resulted in being able to stretch the maximum allowable number of samples that can be streamed in otherwise, thus increasing the throughput.

6.3.2 The Kernel Matrix Computer Function

This function implements the RBF Kernel discussed in chapter 2.18. A Kernel matrix of dimensions NxN is required during the optimization phase of each pair of αs , where N is the number of training samples. With respect to our algorithm, the initial batch consists of 10000 samples and the computations performed on them are repetitive in nature which translates to a sequential execution on the CPU. Hence this function is a viable candidate to be offloaded to the FPGA for acceleration. Listing **??** is an excerpt from the software implementation of algorithm 7 for the Kernel computer function. Xilinx HLS was used to convert this Python code to HLS C++. We can notice that in line 6, enumerate indicates a for loop that repeats for 10000 times per sample.

```
1     function rbf kernel(x, y):
2         exponent = - gamma * linalg.norm(x-y)**2
3         return exp(exponent)
4
5     function kernel_matrix(X_i, X_j):
6         for length(X):
7             result[j] = rbf kernel(X_i, X_j)
8         return result
```

Listing 6.1: Kernel Matrix Computer

The software only execution time of this function as part of the algorithm was around 00:00:02.78. Our aim is to improve on this by offloading it to the FPGA. The function takes in samples X and index as inputs which are implemented in the form of AXI DMA streams. The AXI DMA[3] streaming method allows direct memory access between the FPGA and the BRAM. The streaming of the samples can be pipelined using the pragma HLS PIPELINE. In the function *Kernel matrix* we see a for loop which runs for the number of samples that were streamed in. The pragma HLS UNROLL was used to unroll this for loop. The results are streamed out using a similar AXI DMA interface which is also pipelined using the pragma- HLS PRAGMA.

The FPGA was restricted in resources and could allow only 500 samples of dimensions N x 784 to be streamed in at any given point of time. But during the batch incremental training phase of the algorithm, we deal with 10000 samples at any given point of time, which corresponds to the first batch of training samples that are the maximum amount of samples that the algorithm takes in. Hence this is the reason why 10000 samples were chosen for the execution time comparison. The number of samples in successive batches are reduced by the approximation technique. These 10000 samples are operated on sequentially on the CPU. In an attempt to be able to process all 10000 samples too on the FPGA, the *batch processing* and *tiling* techniques were used to stream in 2500 samples at a time. The results of these 2500 samples were stored and the next batch of 2500 samples were streamed in simultaneously and processed upon. This was repeated for a batch size of 5 corresponding to 2500/500. The tiling and batch processing in software would have required 4 additional DMA transfers. Tiling and batch processing in the hardware overcomes this communication overhead and provides an additional speed up. With this hardware offloading of the function, the execution time of the function as part of the algorithm was around 00:00:01.26. This resulted in a speed up of 2.2. The table 6.3.2 summarizes this.

	Run Time
CPU only	0:00:02.78
FPGA Accelerated	0:00:01.26
Speed up	2.2

Kernel Function	BRAM	DSP	FFs	LUTs
Available	280	220	106400	53200
Used	266	23	31994	49832
Utilization	95%	10%	30%	93%

[3] https://www.xilinx.com/support/documentation/ip$_d$*ocumentation*/*axi*$_d$*ma*/*v*7$_1$/*pg*021$_a$*xi*$_d$*ma.pdf*

6.3.3 The Predict Function

The software only execution time of the predict function as part of the algorithm was around 00:04:24.69 which is significant as it is in minutes. Again, our focus is to improve on this by offloading it to the FPGA. This function is more complicated than the Kernel computer function since it involves operations with the Lagrange multipliers, support vectors and the training samples. These samples are input to the algorithm via individual AXI DMA streams which enables the IP core to simultaneously access all three set of samples for computations. These streams can then be pipelined using the pragma HLS PIPELINE. The predict function also involves the subfunction RBF Kernel as in the Kernel computer function. This function sees two opportunities of loop unrolling using pragma HLS UNROLL at points where the RBF Kernel is computed for all the incoming training samples and at the point where the resultant of RBF Kernel and Lagrange multipliers are accumulated with the training samples in order to return a prediction. The pragma HLS DATAFLOW was used to allow concurrent execution of loops and fucntions. Another optimization in the form of HLS INLINE pragma was used to dissolve the function definition of the RBF Kernel such that it is integrated into the predict function call site.

```
1       function rbf kernel(x, y):
2           exponent = - gamma * linalg.norm(x-y)**2
3           return exp(exponent)
4
5       function predict(X):
6           n_samples = length(X)
7           for i in range (length(X)):
8               result = 0.0
9               for m in range (length(lagrange multipliers)):
10                  result += lagrange multipliers * rbf kernel(support_vectors, X[i])
11              prediction[i] = result
12          return prediction
```

Listing 6.2: The Predict Function

The support vectors and the training samples are both of dimensions N x 784 and the Lagrange multipliers is a vector of dimension N x 1. Given the resource budget of the PYNQ, we are only able to stream in 100 training samples, 10 support vectors and 100 Lagrange multipliers at any given point of time. In order to accommodate this, after exceeding the BRAM resource budget, the array of support vectors were assigned to be stored in the LUT RAM instead of the BRAM using the HLS RESOURCE pragma. A similar attempt at tiling and batch processing was made to enable all 10000 of the first batch of training samples as part of the batch processing in the incremental algorithm. Unfortunately Vivado HLS does not allow multiple streams to exist within each other in the form of for loops. This was noticed on implementation of the Python code for this function in Vivado HLS. In order to facilitate the tiling and batch processing in hardware, one of the streams had to be streamed in entirely so that the rest could be tiled. From the pseudo code, we can observe that the dual coefficients can be separate individual stream and is a 1 dimensional vector when compared to the other two streams of multi dimensional vectors. Hence we ended up with tiling the support vectors and the training samples while streaming in the Lagrange multipliers directly. The use of the pragmas HLS UNROLL and HLS PIPELINE resulted in a execution time of 0:00:50.87 resulting in a speed up of 5.3. The table 6.3.3 summarizes this.

	Run Time
CPU only	0:04:24.69
FPGA Accelerated	0:00:50.87
Speed up	5.3

Predict Function	BRAM	DSP	FFs	LUTs
Available	280	220	106400	53200
Used	275	12	4949	40643
Utilization	98%	5%	4%	76%

6.3.4 Approximations in Hardware

We have discussed software based approximations in the algorithm level previous chapters. In addition to the offloaded functions, a hardware approximation approach was implemented in the form of fixed point computations instead of floating point computations. The library HLS ap_fixed_HLS lets us convert variables declared as floats into fixed point variables. A fixed point variable takes the form $ap\_fixed < 10, 5 >$ where the first number denotes the number of bits of the variable and the second number denotes number of bits allocated to the integer part of a floating point number. This operation was performed for all floating point variables declared in the two functions that were offloaded to the FPGA. It was noticed that there was a significant decrease in the BRAM utilization when compared to the floating point implementation but a drastic increase in LUT usage was observed.

Moving from standardized floating point to fixed point results in a precision drop of the number of digits after a decimal point that a variable holds. This causes a difference in the results produced by standard floating point computations and fixed point computations. The fixed point implementation of the functions as a part of the algorithm produced a slight decreease in the classification accuracy during the prediction phase. In addition to this, given the resource budget of the FPGA even though there was a decrease in the BRAM usage there was an increase in the LUT usage. This led to a trade off between using pragmas to accelerate parts of the functions versus using fixed point. The trade off in execution time was not worth proceeding with the fixed point implementation and hence the resource budget was allocated to using HLS pragmas for the LUT and BRAM utilization which results in a higher gain in acceleration than the fixed point approach. The results of this are summarized in table 6.3.4 and 6.3.4 for the Kernel Function and the predict function respectively.

Kernel Function	Execution Time	Classification Accuracy
Floating Point	0:00:02.78	97.55%
Fixed Point	0:00:02.15	95.15%

Predict Function	Execution Time	Classification Accuracy
Floating Point	0:04:24.69	97.55%
Fixed Point	0:04:11.25	96.25%

6.4 Design Choices

The initial implementation contained two separate IP cores - one for the Kernel Matrix Computer and the other for the Predict function. But each of these could only hold an array of 100 x 784 values while already reaching 95 percent BRAM utilization. Individually. Combining both of them resulted in an resource utilization of 190%. To fit within the resource budget we had to have both of these functions cut the 100 element array by half. This would let us use two individual IP cores but since only half the amount of samples can be processed, the communication overheard increased by twice the original amount.

Hence the next implementation was to share the array between the two functions which let us merge both functions into one IP core while maintaining a single array with a resource utilization of 95 percent. A flag was set to indicate which of the two functions is invoked, which could be controlled from the software. the third iteration involved using two separate hardware implementations of each function resulting in two bitstreams. This way each function can be optimized to its full potential without having to consider balancing them out based on resource utilization. Each respective bitstream could be then loaded when the particular function is called for a large batch of data to be operated on, this does provide an increase in the acceleration capability while introducing a slight additional bitstream loading time to be considered. The only downside is that if the batch size isn't large enough to consider switching bitstreams, the bitstream loading time would shave off a part of the speedup gained.

6.5 Results

Tables 6.5 and 6.5 represent the hardware acceleration results combining both the hardware implemented functions and integrating them into the rest of the SVM implementation in software. This completes the hardware-software codesign approach. From the table it is noticeable that for the MNIST data set we achieved a speed up of 3.49 for the training phase and 5.3 for the inference phase. Similarly from table 6.5 we can see that the implementation achieved a speed up of 3.78 during the training phase and 3.25 for the inference phase respectively.

MNIST	Training Time	Inference Time
CPU only	20:23:50.33	0:04:24.69
FPGA Accelerated	6:11:25.31	0:00:50.87
Speed up	3.49	5.3

Fashion MNIST	Training Time	Inference Time
CPU only	28:43:47.23	0:08:44.49
FPGA Accelerated	7:51:45.11	0:0:59.22
Speed up	3.78	3.25

7

Conclusion

The aim of the thesis was to implement an SVM algorithm in incremental manner which lets it run using incremental learning such that it is able to run on a board with tight resource and power budget. The experiment platform was based on PYNQ-Z1. We were supposed to design an SVM algorithm without using existing libraries with the notion that the end result algorithm can run on the board which has limited memory and resources. Our task was to be able to run the SVM algorithm on an embedded system were standard data sets were used. The embedded system is restricted in memory and resource. Hence the only way it will run is by an incremental learning method. In order to be able to implement this we cannot employ existing libraries. So the SVM algorithm in its non incremental form had to be implemented. The performance of this algorithm with the data sets is noted. The algorithm is then modified to achieve incremental learning. A base incremental algorithm is developed from non incremental algorithm. It is observed that when the algorithm is run on an embedded system, the computation time is large since all samples of huge data set undergo computation. In order to reduce number of samples, we have researched algorithm approximation. Three different techniques to approximate input samples are researched and implemented. Each implementation tries to improve on the accuracy of the predecessor. Final incremental and approximated algorithm which gives the maximum accuracy is then chosen to be run on the embedded system. The embedded system being a hybrid between embedded and FPGA, the idea is t accelerate the training and inference process of the algorithm by using the FPGA. The incremental algorithm is studied for potential points where computations can be offloaded to the FPGA. These computations are identified as functions and are implemented in hardware.

The base execution time of this implementation is noted and then we try to improve it by leveraging hardware optimization and approximation. Several hardware compiler directives were researched , used and introduced into functions implemented on the hardware with the expectation that the execution time will reduce further. It was noted that these compiler directives were successful in accelerating incremental algorithm even further by an average speed factor of 4.5. The precision of algorithm was compared to non incremental and state of the art implementations. Here we notice an accuracy difference of 0.63 (for non incremental) and 1.2 (for state of the art). This is expected with trying to approximate algorithms and perform incremental learning

7.1 Future Work

Our implementation was for the MNIST data set and a similar fashion MNIST data set which are considered mid range when it comes to budgeted embedded systems, but small in comparison to other big data. With a more powerful board of similar characteristics of the PYNQ but on a smaller footprint would facilitate the implementation on larger data sets like AlexNet. There is a scope for research into changing the ML algorithm structure and rewriting it with a perspective of being able to accelerate it using hardware since the original algorithm on which our implementation was based on was written focusing on implementing it on a CPU. In our implementation, hardware based approximation was only briefly researched. Implementations using lower precision custom data types would help in accelerating the algorithm by many folds. Further study into the algorithm itself would facilitate in implementing the entire algorithm or the SMO procedure in hardware. This was also briefly researched but with a trade off in the acceleration versus resource utilization.

Bibliography

[AGS15] S. Afifi, H. Gholamhosseini, and R. Sinha. In: (2015).

[And01] R. Andraka. In: (Dec. 2001).

[ANI13] Y. Ago, K. Nakano, and Y. Ito. "A Classification Processor for a Support Vector Machine with Embedded DSP Slices and Block RAMs in the FPGA". In: *2013 IEEE 7th International Symposium on Embedded Multicore Socs*. 2013.

[Ber76] D. P. Bertsekas. *Dynamic Programming and Stochastic Control*. Academic Press, Inc., 1976.

[BFP99] L. Bruzzone and D. Fernández Prieto. "An Incremental&Mdash;Learning Neural Network for the Classification of Remote&Mdash;Sensing Images". In: *Pattern Recogn. Lett.* (1999).

[BH+08] A. Ben-Hur et al. "Support Vector Machines and Kernels for Computational Biology". In: *PLoS computational biology* (2008).

[BL07] L. Bottou and C.-J. Lin. "Support Vector Machine Solvers". In: *Large Scale Kernel Machines*. 2007.

[Bur98] C. J. C. Burges. "A Tutorial on Support Vector Machines for Pattern Recognition". In: *Data Min. Knowl. Discov.* (1998).

[CCC13] Z. Chen, K. Chen, and J. Chen. "Vehicle and Pedestrian Detection Using Support Vector Machine and Histogram of Oriented Gradients Features". In: *2013 International Conference on Computer Sciences and Applications*. 2013.

[CGLQ15] S. A. Camelo, M. D. González-Lima, and A. J. Quiroz. "Nearest neighbors methods for support vector machines". In: *Annals of Operations Research* (2015).

[Cho11] M. Choudhury. "Approximate Logic Circuits: Theory and Applications". PhD thesis. Rice University, 2011.

[CP01] G. Cauwenberghs and T. Poggio. "Incremental and Decremental Support Vector Machine Learning". In: *Advances in Neural Information Processing Systems 13*. 2001.

[CSF86] J. C. Schlimmer and D. Fisher. "A Case Study of Incremental Concept Induction." In: *Proceedings of the Fifth National Conference on Artificial Intelligence*. 1986.

[CV95a] C. Cortes and V. Vapnik. "Support-Vector Networks". In: *Machine Learning*. 1995.

[CV95b] C. Cortes and V. Vapnik. "Support-vector networks". In: *Machine Learning* (1995).

[Die00] T. G. Dietterich. "Ensemble Methods in Machine Learning". In: *Proceedings of the First International Workshop on Multiple Classifier Systems*. 2000.

[Duv14] D. Duvenaud. "Automatic model construction with Gaussian processes". PhD thesis. 2014.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[GSLJ10] J. Gimeno Sarciada, H. Lamela, and M. Jimenez. In: (Apr. 2010).

[Guo+16] Q. Guo et al. "High-Density Image Storage Using Approximate Memory Cells". In: *SIGARCH Comput. Archit. News* (2016).

[Gup+15] S. Gupta et al. “Deep Learning with Limited Numerical Precision”. In: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning*. 2015.

[HM07] G. Hulley and T. Marwala. “Evolving Classifiers: Methods for Incremental Learning”. In: (2007).

[HS06] X.-F. Hui and J. Sun. “An Application of Support Vector Machine to Companies’ Financial Distress Prediction”. In: *Modeling Decisions for Artificial Intelligence*. 2006.

[Ins09] A. Inselberg. *Parallel Coordinates: Visual Multidimensional Geometry and Its Applications*. Springer-Verlag, 2009.

[JB14] L. Junfei and Z. Baolei. “Online learning algorithm of direct support vector machine for regression based on Cholesky factorization”. In: *2014 International Conference on Information Science, Electronics and Electrical Engineering*. 2014.

[JHL15] H. Jiang, J. Han, and F. Lombardi. “A Comparative Review and Evaluation of Approximate Adders”. In: *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*. 2015.

[JR14] E. J. Jordan and R. Radhakrishnan. “Machine learning predictions of cancer driver mutations”. In: *Proceedings of the 2014 6th International Advanced Research Workshop on In Silico Oncology and Cancer Investigation - The CHIC Project Workshop (IARWISOCI)*. 2014.

[JZJ05] Jianpei Zhang, Zhongwei Li, and Jing Yang. “A divisional incremental training algorithm of support vector machine”. In: *IEEE International Conference Mechatronics and Automation*. 2005.

[Kar39] W. Karush. “Minima of Functions of Several Variables with Inequalities as Side Conditions”. MA thesis. Department of Mathematics, University of Chicago, 1939.

[KL51] S. Kullback and R. A. Leibler. “On Information and Sufficiency”. In: *The Annals of Mathematical Statistics* (1951). DOI: 10.1214/aoms/1177729694.

[Li+14] M. Li et al. “Efficient Mini-batch Training for Stochastic Optimization”. In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2014.

[LLW11] C. Li, K. Liu, and H. Wang. “The incremental learning algorithm with support vector machine based on hyperplane-distance”. In: *Applied Intelligence* (2011).

[MCS] J. Milgram, M. Cheriet, and R. Sabourin. ““One Against One” or “One Against All”: Which One is Better for Handwriting Recognition with SVMs?” In: *Tenth International Workshop on Frontiers in Handwriting Recognition*. URL: https://hal.inria.fr/inria-00103955.

[MH08] L. van der Maaten and G. Hinton. “Visualizing Data using t-SNE ”. In: *Journal of Machine Learning Research* (2008).

[MHT18] F. C. M. Rodrigues, R. Hirata, and A. C. Telea. “Image-Based Visualization of Classifier Decision Boundaries”. In: *2018 31st SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*. 2018.

[Mit16a] S. Mittal. “A Survey of Techniques for Approximate Computing”. In: *ACM Comput. Survey.* (2016).

[Mit16b] S. Mittal. “A Survey of Techniques for Approximate Computing”. In: *ACM Computing Surveys* (2016).

[MNY06] H. Q. Minh, P. Niyogi, and Y. Yao. “Mercer’s Theorem, Feature Maps, and Smoothing”. In: *Proceedings of the 19th Annual Conference on Learning Theory*. 2006.

[MR17] P. M. Radiuk. “Impact of Training Set Batch Size on the Performance of Convolutional Neural Networks for Diverse Datasets”. In: *Information Technology and Management Science* (2017).

[MS19] M. Mohammadi and M. Sarmad. “Outlier Detection for Support Vector Machine using Minimum Covariance Determinant Estimator”. In: *Journal of AI and Data Mining* (2019).

[MS78] B. A. Murtagh and M. A. Saunders. “Large-scale linearly constrained optimization”. In: *Mathematical Programming* (1978).

[NDJ17] M. T. Nguyen, V. Dinh Nguyen, and J. W. Jeon. “Real-time pedestrian detection using a support vector machine and stixel information”. In: *2017 17th International Conference on Control, Automation and Systems (ICCAS)*. 2017.

[OFG97] E. Osuna, R. Freund, and F. Girosi. “An Improved Training Algorithm for Support Vector Machines”. In: IEEE, 1997.

[Pla99] J. C. Platt. “Fast Training of Support Vector Machines Using Sequential Minimal Optimization”. In: *Advances in Kernel Methods*. 1999.

[PSCA00] A. Pinar Saygin, I. Cicekli, and V. Akman. “Turing Test: 50 Years Later”. In: *Minds Mach.* (2000).

[RÖ1] S. Rüping. “Incremental Learning with Support Vector Machines”. In: *Proceedings of the 2001 IEEE International Conference on Data Mining*. 2001.

[RB15] M. B. Rabieah and C. Bouganis. “FPGA based nonlinear Support Vector Machine training using an ensemble learning”. In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 2015.

[RW06] C. Rasmussen and C. Williams. *Gaussian Processes for Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, 2006.

[SHD17] S. Si, C.-J. Hsieh, and I. S. Dhillon. “Memory Efficient Kernel Approximation”. In: *J. Mach. Learn. Res.* (2017).

[SK+05] S S. Keerthi et al. “A Fast Dual Algorithm for Kernel Logistic Regression”. In: *Machine Learning* (2005).

[Spe03] N. Spencer. “Investigating Data with Andrews Plots”. In: *Social Science Computer Review* (2003).

[Sri+14] N. Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *J. Mach. Learn. Res.* (2014).

[SS01] B. Scholkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, 2001.

[Sye+99] N. A. Syed et al. *Incremental Learning with Support Vector Machines*. 1999.

[SZ05] M. Sordo and Q. Zeng. “On Sample Size and Classification Accuracy: A Performance Comparison”. In: *Biological and Medical Data Analysis*. 2005.

[TTY08] Takashi Saegusa, Tsutomu Maruyama, and Yoshiki Yamaguchi. “How fast is an FPGA in image processing?” In: *2008 International Conference on Field Programmable Logic and Applications*. 2008.

[vH08] L. van der Maaten and G. Hinton. “Visualizing High-Dimensional Data Using t-SNE”. In: *Journal of Machine Learning Research* (2008).

[VL63] V Vapnik and A Lerner. “Pattern Recognition using Generalized Portrait Method”. In: *Automation and Remote Control* (1963).

[Wie+12] M. Wielgosz et al. In: (2012).

[WL07] Y.-M. Wen and B.-L. Lu. “Incremental Learning of Support Vector Machines by Classifier Combining”. In: *Advances in Knowledge Discovery and Data Mining*. 2007.

[XJ19] C. Xu and S. A. Jackson. “Machine learning and complex biological data”. In: *Genome Biology* (2019).

[Zha+17] W. Zhang et al. “Weather prediction with multiclass support vector machines in the fault detection of photovoltaic system”. In: *IEEE/CAA Journal of Automatica Sinica* (2017).

[ZJ06] D.-X. Zhou and K. Jetter. “Approximation with polynomial kernels and SVM classifiers”. In: *Advances in Computational Mathematics* (2006).

[Cao+] Cao Zhitong et al. “Support vector machine used to diagnose the fault of rotor broken bars of induction motors”. In: *Sixth International Conference on Electrical Machines and Systems, 2003. ICEMS 2003.*

[Gom+10] J. Gomes Filho et al. “A general-purpose dynamically reconfigurable SVM”. In: *2010 VI Southern Programmable Logic Conference (SPL)*. 2010.

[Hui+04] Hui Xiao et al. “Swarm intelligence based design of intelligent house embedded system”. In: *Fifth World Congress on Intelligent Control and Automation (IEEE Cat. No.04EX788)*. 2004.

[Kir+08] C. G. Kiran et al. “support vector machine learning based traffic sign detection and shape classification using Distance to Borders and Distance from Center features”. In: *TENCON 2008 - 2008 IEEE Region 10 Conference*. 2008.

[Kua+12] T. Kuan et al. “VLSI Design of an SVM Learning Core on Sequential Minimal Optimization Algorithm”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2012).

[Lin+18] Z. Lin et al. “Vehicle and Pedestrian Recognition Using Multilayer Lidar based on Support Vector Machine”. In: *2018 25th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*. 2018.

[Pat+12] R. A. Patil et al. “Power Aware Hardware Prototyping of Multiclass SVM Classifier Through Reconfiguration”. In: *2012 25th International Conference on VLSI Design*. 2012.

[Sam+13] A. Sampson et al. “Approximate storage in solid-state memories”. In: *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2013.

[Sam59] A. L. Samuel. “Some Studies in Machine Learning Using the Game of Checkers”. In: *IBM Journal of Research and Development* (1959).

[Son+19] Z. Song et al. “Approximate Random Dropout for DNN training acceleration in GPGPU”. In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019.

[Xu+18] J. Xu et al. “New Incremental Learning Algorithm With Support Vector Machines”. In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* (2018).

[Zha+18] R. Zhao et al. “Machine Health Monitoring Using Local Feature-Based Gated Recurrent Unit Networks”. In: *IEEE Transactions on Industrial Electronics* (2018).

Nomenclature

ϵ	Predefined Accuracy
$\mathcal{C}$	Large Margin vs Small Margin violation tradeoff
$\mathcal{D}$	Dual form
$\mathcal{H}$	Hessian Matrix
$\mathcal{K}$	Kernel Function
$\mathcal{P}$	Primal form
$\phi(x)$	Feature Vector
σ	Standard Deviation
$\mathbf{g}$	Gradient
ξ	Slack variable
b^*, b	Bias
u	Direction for Optimization
w	Slope
w^*, w	Optimization Direction
x	Sample

List of Figures

List of Tables

Declaration of Authorship

I hereby declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. According to my knowledge, the content or parts of this thesis have not been presented to any other examination authority and have not been published. I am aware that the respective work can be considered as a "fail" in the event of a false declaration. In case of justified suspicion, the thesis in digital form can be examined with the help of "Turnitin". For the comparison of my work with existing sources

- I agree to storage in the institutional repository to enable comparison with future theses submitted
- I do not agree to storage in the repository.

Further rights of reproduction and usage, however, are not granted here. In any case, the examination and evaluation of my work has to be carried out individually and independently from the results of the plagiarism detection service.

signature, city, date