WITS

# Parallel Democratic Support Vector Machines for Multi-Class Classification

*Author:*
Dario Vieira (806414)

Marco Lerena (1242414)

May 22, 2018

# 1 Introduction

This report outlines a parallel implementation of multi-class classification using support vector machines with ensemble learning. We first introduce the serial implementation which uses the Pegasos algorithm to train multiple SVMs, and a voting paradigm that performs multi-class classification using the trained set of SVM models. We then discuss how this implementation is parallelized using CUDA for the Pegasos algorithm, and MPI to distribute the multiple model generations. We outline our methodology, including the dataset used, the testing environment, and the experiment setup. Finally, we present the results of our experiments, alongside an analysis of the performance improvements achieved.

# 2 Background

## 2.1 Support Vector Machine

The support vector machine (SVM) is a supervised learning model used for classification problems with two classes. Given a labeled set of data-points, the SVM constructs a hyperplane that separates two classes using the widest margin possible. Producing a wide margin aims to lower the generalization error of the classifier. Constructing the hyperplane with the widest margin is an optimization problem.

## 2.2 Pegasos Algorithm

Pegasos stands for Primal Estimated sub-GrAdient SOlver for SVM, which as the name suggests, is a stochastic sub-gradient descent algorithm for solving the support vector machine optimization problem.

## 2.3 Ensemble Learning

Ensemble learning is a machine learning paradigm where a set of classifiers are trained to solve a single problem. The classifiers in the ensemble are known as a base learners, and are usually homogeneous i.e. all alike in some way. In our case, for example, the ensemble will be a set of support vector machines. Once an ensemble has been trained, the learners are then combined in order to solve the core problem. One common combination scheme is *majority voting*, in which each member of the ensemble casts a vote as its contribution to the solution. The voting combination scheme is what we will be using to perform multi-class classification with a set of binary SVM classifiers. This is elaborated on in section 3.

# 3 Algorithm Design

## 3.1 Context

In order to use Support Vector Machines for multi-class classification, we employ the ensemble learning scheme of majority voting. Multiple binary SVMs are produced, with each classifier being trained on a different combination of classes. The number of classifiers required is $\binom{c}{2}$, where $c$ is the number of classes that need to be classified.

For example, with classes = $\{1, 2, 3\}$, a set of $\binom{3}{3} = 3$ weight vectors are trained, one for each classifier; W(1,2), W(1,2), and W(2,3). Once the various weight vectors have been trained, we employ the voting procedure. For an unclassified data-point $x$, each classifier casts a vote, which is stored in a vote matrix.

For each $W(i, j)$:

- if, $h(x, W(i, j)) < 0$, $voteMatrix(i, j) = -1$
- else if, $h(x, W(i, j)) > 0$, $voteMatrix(i, j) = 1$,
  where $h(x, W(i, j)) = x \cdot W(i, j)$

This can be interpreted as follows; if voteMatrix(i,j) = -1, then classifier $i, j$ has voted that $x$ is in class $i$. Once all classifiers cast their votes, a tally of the votes is produced to make the final classification democratically.

## 3.2 Serial

The context above describes how we utilize an ensemble of SVMs with a voting scheme, to classify a data-point. It is important to keep this in mind when training to ensure the ensemble suits our purposes. The two algorithms below facilitate the main elements of our training process.

### 3.2.1 Multi-Model Generation

Algorithm 1 loads the training data, and launches Algorithm 2 $\binom{c}{2}$ times. Each time Algorithm 2 is launched, it is passed the subset of data required to train a set of weights for classes $i$ and $j$. The result of Algorithm 2 is stored in weight matrix at $W(i, j)$.

### 3.2.2 Pegasos

Algorithm 2 is the Pegasos algorithm called to train a set of weights for an SVM classifier. As stated above, the data sent to the algorithm is only the subset required to train a binary classifier for two ($i$ and $j$) of the c classes.

**Algorithm 1** Multi-Model Generation

Load Data
Set T and $\lambda$
Initialize 3D weight matrix W.
(of size c by c by d)
**for** i=1 to c **do**
  **for** j=0 to i **do**
    S = subset of data where target y = i, or y = j
    $W_{(i,j)}$ = **Algorithm 2(S,$\lambda$,T)** //Pegasos training algorithm
  **end for**
**end for**

**Algorithm 2** Serial Pegasos

**Input:**
Training data set: S
Regularisation Parameter: $\lambda$
Number of Iterations: T
**Output:**
Learned weights W
set W = 0
**for** t = 1 to T **do**
  choose a sample $(x_t, y_t)$ uniformly at random
  set $\eta_t = \frac{1}{\lambda t}$
  **if** $y_t h(x_t) < 1$ **then**
    $W_{t+1} = (1 - \eta_t)W_t + \eta_t y_t x_t$
  **end if**
  **if** $y_t h(x_t) \geq 1$ **then**
    $W_{t+1} = (1 - \eta_t)W_t$
  **end if**
**end for**

### 3.3 Parallel

For this project there are two frameworks we are tasked with using; MPI and CUDA. As stated above, the Multi-Model Generation and Pegasos algorithms are the main components that facilitate the learning process. Each of these components is parallelized using a different framework, namely:

- **MPI for Multi-Model Generation**

  Seeing as though each set of weights is generated in isolation, they can be generated simultaneously. MPI is perfectly suited for parallelizing this component, since minimal communication is required.

- **CUDA for Pegasos**

Being a learning algorithm, Pegasos iterations need to run consecutively i.e. we cannot split the iterations across different threads (loop carried dependency). However, the learning algorithm contains a dot product and weight updates, both of which can be done in parallel.

It is important to reinforce the fact that these two components are nested i.e. the Multi-Model Generation calls the Pegasos algorithm to generate a set of weights. In this report we focus on the implementation and performance of each parallel method on its own. At the end of this document, under further work, we discuss how this project can be extended, including a hybrid implementation.

#### 3.3.1 CUDA

As discussed above, CUDA has been used to parallelize the Pegasos algorithm. Once Algorithm 3 is called, it defines the necessary variables to be copied onto device global memory, namely;

- The weight array, initialized to $\mathbf{0}$
- The training data, both $X$ and $y$

On the first iteration of the training loop, the CUDA kernel, Algorithm 4, is launched and copies these variables onto device global memory, along with the index of the first data-point to use for training. In each subsequent kernel call, only the index of the next data-point to use for training is copied over. This allows the device to continually update the same set of weights until the training loop completes, this is only possible because of CUDAs constant memory between kernel calls. Once the loop completes, the trained weights are copied back from the device to the host (Algorithm 3) where they can be stored for the voting procedure.

In each iteration, Algorithm 4 first performs the necessary dot product in parallel using a reduction formula, and then performs the weight update in parallel. This is possible due to the fact that the kernel is launched with the same number of threads as the dimension of the data-points/weights. A decision needs to be made in between the dot product and the weight update, and so thread 0 has been tasked with making this decision. The other threads wait until this decision is made before they proceed to update their designated weight. This decision replaces the double if statement from the original PEGASOS algorithm (Algorithm 2).

**Algorithm 3** CUDA Pegasos

1: **Input:**
   Training data set: S
   Regularisation Parameter: $\lambda$
   Number of Iterations: T
2: **Output:**
   Trained weights W
3: set W = 0
4: **for** t = 1 to T **do**
5:  choose an index i from S uniformly at random
6:  set $\eta_t = \frac{1}{\lambda t}$
7:  **Algorithm 4 (Kernel)**
8: **end for**

**Algorithm 4** CUDA Kernel

1: **Input:**
   Training data point index $i$
   Weights: W
   Learning rate: $\eta$
2: **Output:**
   Updated weights W
3: h(x) = dot product of $X_i$ W
4: **if** $y_i h(X_i) < 1$ **then**
5:  $W = (1 - \eta_t)W + \eta_i y_i x_i$
6: **end if**
7: **if** $y_i h(x_i) \geq 1$ **then**
8:  $W = (1 - \eta_t)W$
9: **end if**

### 3.3.2 MPI

MPI Algorithm 5 has been used to parallelize the Multi-model Generation algorithm (Algorithm 1). This is done by initializing a separate node for each set of weights that needs to be generated. Each of these nodes calls the serial PEGASOS algorithm (Algorithm 2) on their portion of the data set in order to train the weights. The data is read in by only the lead node, which then broadcasts the data to all other nodes. The lead node also decides which set of weights each other node will be training, and broadcasts this information as well. As stated earlier, each set of weights required is trained to distinguish a different combination of classes. Thus, the lead node provides each other node with an $(i, j)$ tuple, indicating which classes data should be used for its model. The weights are then reduced to a single Matrix located only in the lead node. The lead node then contains the complete weight matrix that can be used to classify a point using the voting scheme.

**Algorithm 5** MPI Multi-model Generation

1: *Load Data*
2: Broadcast Data
3: *Assign each node a value for i and j*
4: Set T and $\lambda$
5: Initialize 3D weight matrix W.
(of size c by c by d)
6: S = Data set where target y == i or y==j
7: $W_{(i,j)}$ = **Algorithm 2(S,$\lambda$,T)** //Pegasos training algorithm
8: Reduce various weight vectors into a single matrix

# 4 Experiment Setup

The aim of our experiment is to measure the performance of the serial, CUDA, and MPI algorithms described above. It is important to reiterate that a hybrid implementation is not being tested. This means that when the MPI implementation parallelizes the multiple model generations, the training of a model on each node is done using serial Pegasos. While in the CUDA implementation, when a model is produced using CUDA Pegasos, each model is being produced one after the other in a serial fashion.

Below we describe the setup of our experiment, including the testing environment and the data used. We then present our results, followed by an analysis of each model's performance.

## 4.1 Testing Environment

1. The testing environment for the Serial and CUDA implementations is as follows:

   - CPU: Intel core i7-7700 3.60GHz
   - Number of threads on CPU: 8
   - GPU: NVIDIA Geforce GTX 1060 6GB
   - Operating System: Ubuntu 16.04 LTS 64bit

2. The testing environment for the MPI implementation is the MSL Cluster at the University of the Witwatersrand.

## 4.2 Data

The dataset used for our experiment is 1000 32x32 black/white images of handwritten numbers from '0' to '9' i.e. 10 classes. The amount of images in each class varies, but no class is particularly scarce of samples. The dataset has been been exported to CSV in two versions:

1. Rows containing the raw pixel values in vectorized (flattened) format i.e. 1024 dimensions per image.

2. Rows containing the raw pixel values reduced to 512 dimensions using PCA

### 4.3 Test Configurations

For our tests we use three configurations of data dimensions, $d$, in conjunction with three configurations of Pegasos iterations, $T$:

1. $d = 1024$ and $T = 100, 1000, 10000$

2. $d = 512$ and $T = 100, 1000, 10000$

3. $d = 256$ and $T = 100, 1000, 10000$

The tests with $d = 256$ utilize the first 256 dimensions of the PCA data described above.

## 5 Results

The following three tables show an analysis of the Times (Total, Processing & overhead) in seconds that each implemented algorithm took, as well as the accuracy (as a percentage when the final weights were tested) on a given T value (where T is the number of iterations).
The times were taken as follows:

- Total - The run time of the training of the weights - this was done as the voting was not part of the parallelization, and therefor was not included in the timing as it would only skew the results.

- Processing - This was done around the multi-model generation - this is because it can encompass the speedup seen in both parallelized versions of the implementation.

- Overhead(Serial & CUDA) - This is $Total - Processing$ - this is mainly data-reading and initialising all the variables.

- Overhead(MPI) - This is each MPI call that is made. In this case it includes only three MPI broadcasts and one MPI reduction.

Points to be noted from the below tables:

- As T is increased by multiples of 10, the processing time for each algorithm increases by a multiple of 10, this is correct because the parallelization was not implemented on the loop that runs T times, this can also be seen across all values of d, as the variation of d does not affect the T loop.

- MPI overheads increase as the d increases, but remains constant for the varying sizes of T. This is because the measured MPI overheads will only be affected by transferring different volumes of data, these transferred data are the weights and data values (both have sizes directly proportional to d).
- Accuracy is maintained across each pair of d and T for all the algorithms. This shows that the algorithms were parallelized correctly.
- Processing Time for Serial and MPI implementations are approximately doubled when the d is doubled. This is because these processes run weight updates and dot products in serial, and both of these are of size d. The CUDA implementation maintains approximately the same time time for each d, (and the corresponding T) this is because the parralelization of CUDA is around the d value.

Table 1: Table containing run-times for d = 256

| **Implementation** | **T** | **Total (s)** | **Processing (s)** | **Overheads (s)** | **Accuracy %** |
|---|---|---|---|---|---|
| Serial | 100 | 0.023 | 0.0004 | 0.023 | 82.07 |
| CUDA | 100 | 0.192 | 0.002 | 0.191 | 82.24 |
| MPI | 100 | 1.075 | 0.0005 | 0.171 | 82.23 |
| Serial | 1000 | 0.188 | 0.004 | 0.184 | 97.86 |
| CUDA | 1000 | 0.558 | 0.010 | 0.549 | 98.36 |
| MPI | 1000 | 1.190 | 0.016 | 0.091 | 98.19 |
| Serial | 10000 | 1.852 | 0.04 | 1.812 | 98.36 |
| CUDA | 10000 | 4.554 | 0.1 | 4.454 | 98.36 |
| MPI | 10000 | 1.337 | 0.209 | 0.245 | 98.36 |

Table 2: Table containing run-times for d = 512

| **Implementation** | **T** | **Total (s)** | **Processing (s)** | **Overheads (s)** | **Accuracy %** |
|---|---|---|---|---|---|
| **Serial** | 100 | 0.045 | 0.001 | 0.044 | 80.92 |
| **CUDA** | 100 | 0.195 | 0.001 | 0.194 | 81.25 |
| **MPI** | 100 | 0.818 | 0.001 | 0.102 | 82.89 |
| **Serial** | 1000 | 0.375 | 0.008 | 0.367 | 98.36 |
| **CUDA** | 1000 | 0.572 | 0.010 | 0.562 | 98.36 |
| **MPI** | 1000 | 0.948 | 0.019 | 0.204 | 98.03 |
| **Serial** | 10000 | 3.697 | 0.082 | 3.615 | 98.36 |
| **CUDA** | 10000 | 4.475 | 0.097 | 4.378 | 98.36 |
| **MPI** | 10000 | 1.932 | 0.520 | 0.235 | 98.36 |

Table 3: Table containing run-times for d = 1024

| Implementation | T | Total (s) | Processing (s) | Overheads (s) | Accuracy % |
|---|---|---|---|---|---|
| **Serial** | 100 | 0.096 | 0.002 | 0.094 | 58.43 |
| **CUDA** | 100 | 0.226 | 0.001 | 0.224 | 66.05 |
| **MPI** | 100 | 0.359 | 0.002 | 0.181 | 64.08 |
| **Serial** | 1000 | 0.760 | 0.016 | 0.743 | 88.44 |
| **CUDA** | 1000 | 0.632 | 0.011 | 0.621 | 94.96 |
| **MPI** | 1000 | 0.898 | 0.035 | 0.378 | 95.57 |
| **Serial** | 10000 | 7.413 | 0.165 | 7.248 | 98.77 |
| **CUDA** | 10000 | 4.986 | 0.107 | 4.879 | 98.77 |
| **MPI** | 10000 | 2.308 | 1.109 | 0.303 | 98.77 |

# 6 Conclusion

When looking at total time, MPI is the top performer when $T$ reaches 10000, regardless of the size of $d$. CUDA's total time shows promise when the number of dimensions increases to 1024. CUDAs processing time is also the best for larger values of d. CUDA's implementation can be improved dramatically, this is discussed in the section below.

# 7 Further Work

### 7.0.1 MPI CUDA Hybrid

As we mentioned in the earlier sections, this project did not investigate the performance of an MPI CUDA hybrid for ensemble learning with SVMs. The main reason for its exclusion from this project is that the MSL cluster used to run our MPI implementation does not have the compiler necessary to run CUDA. However, the MPI and CUDA algorithms described in this report should be easily combined to create this hybrid implementation. Further work in this area should attempt to measure its performance.

### 7.0.2 Further CUDA Parallelization

In the CUDA implementation of Pegasos, the kernel is the only computation called within the training loop. Making the T kernel calls drastically reduces the performance of this implementation. Moving this loop inside the kernel call would greatly improve performance. the main issue with doing so is that the $d$ threads called to perform dot product and weight update need to work on a single training iteration simultaneously. Depending on the implementation, moving this loop inside the kernel may cause the threads to function independently of each other, resulting in the weights being trained incorrectly. However, with clever synchronization of threads this can be done correctly. Further work should attempt to increase performance in this fashion.

# 8 References

Ensemble Learning: https://cs.nju.edu.cn/zhouzh/zhouzh.files/publication/springerEBR09.pdf
Pegasos Algorithm: https://www.cs.huji.ac.il/ shais/papers/ShalevSiSrCo10.pdf
Support Vector Machine: http://image.diku.dk/imagecanon/material/cortes_vapnik95.pdf