EE542 Final Project

Support Vector Machines in Hardware

Raghunandan Kumaran

Introduction

Cognitive systems capable of gathering information, detecting significant events, making decisions and/or coordinating operations are of immense value to a wide variety of application domains, from biomedical devices to automated military units. The core functionality of such machine learning involves mathematical kernels employing commonly used operators, typically implemented as software-based solutions executing on general-purpose machines. Unfortunately, such solutions require significant resources for execution and may consequently be unsuitable for portable applications. Efficient hardware implementations of machine-learning techniques yield a variety of advantages over software solutions: increased processing speed, reliability and battery life as well as reduced cost and complexity.

However, aside from a plethora of work in neural-network implementations, there are few hardware-based machine learning technologies. There have been a few attempts at implementing SVMs in hardware, even though their advantage and robustness far exceeds that of neural networks. Perhaps the reason for this could be that neural networks lend themselves naturally for hardware implementation, while SVM by their design look more suited for software or a processor based design. Another reason is that training a neural network is a lot simpler than SVM, since neural nets use a gradient based optimization while SVMs use quadratic optimization. This is also a main reason as to why there have been very little hardware implementations of SVMs, since it is very hard to implement in hardware with a reasonable degree of accuracy.

Typical applications of machine learning do not really need a capability to train and adapt very frequently. In a co-design approach to embedded systems, if and when new data is available and must be adapted to, it can be transmitted to a host machine where the model can be adapted to the new and then transmitted back to the hardware for use.
In this project my aim is to try a simple SVM implementation for both linear and non-linear kernels. I further plan to make use of this implementation for implementing an SVM based face detection scheme.

Support Vector Machines

The Support Vector Machine (SVM) algorithm is based on statistical learning theory. A simple and intuitive algorithm, it performs excellently for complex real-world problems that may be difficult to analyze theoretically. SVMs are an extension of linear models that are capable of nonlinear classification. Linear models are incapable of representing a concept with nonlinear boundaries between classes. SVMs employ linear models to represent nonlinear class boundaries by transforming the input, or *instance space*, into a new space using a nonlinear mapping. This transformation is facilitated through the use of kernels. The SVM algorithm can be treated linearly within the instance space, whereas the choice of various kernels may map the core operations transparently to a higher dimensional space. Consequently, complex pattern recognition and classification approaches can abstractly be represented linearly. Following this transformation, a Maximum Margin Hyperplane (MMH) that separates the instances by class is learned, thereby forming a decision boundary. The MMH comes no closer to a given instance than it must; in the ideal case it optimally separates classes. *Support vectors* are the training instances closest to the MMH. A set of support vectors thus defines the decision boundary for a given set of instances. This simplifies the representation of the decision boundary since other training instances can be disregarded.

SVM training is a complex quadratic optimization problem for obtaining the support vectors x_i (with class values y_i), their coefficients α, and a threshold value b. Support Vector classification (in a simple two-class problem) simply looks at the sign of a decision function. A test instance x is classified by the following decision function

$$\phi(x) = \operatorname{sgn}(\sum_j \alpha_i y_i K(x_i, x_j) + b)$$

The choice of the kernel function , $K(x_i,x_j)$ and the resultant feature space determines the functional form of the support vectors; thus, different kernels behave differently. Some of the commonly used kernels are :
Linear: $K(X,Y) = X.Y$

Polynomial: $K(X,Y) = (1 + X.Y)^d$

Radial Basis function: $K(X,Y) = \exp(-||X-Y||^2/2\sigma^2\ )$

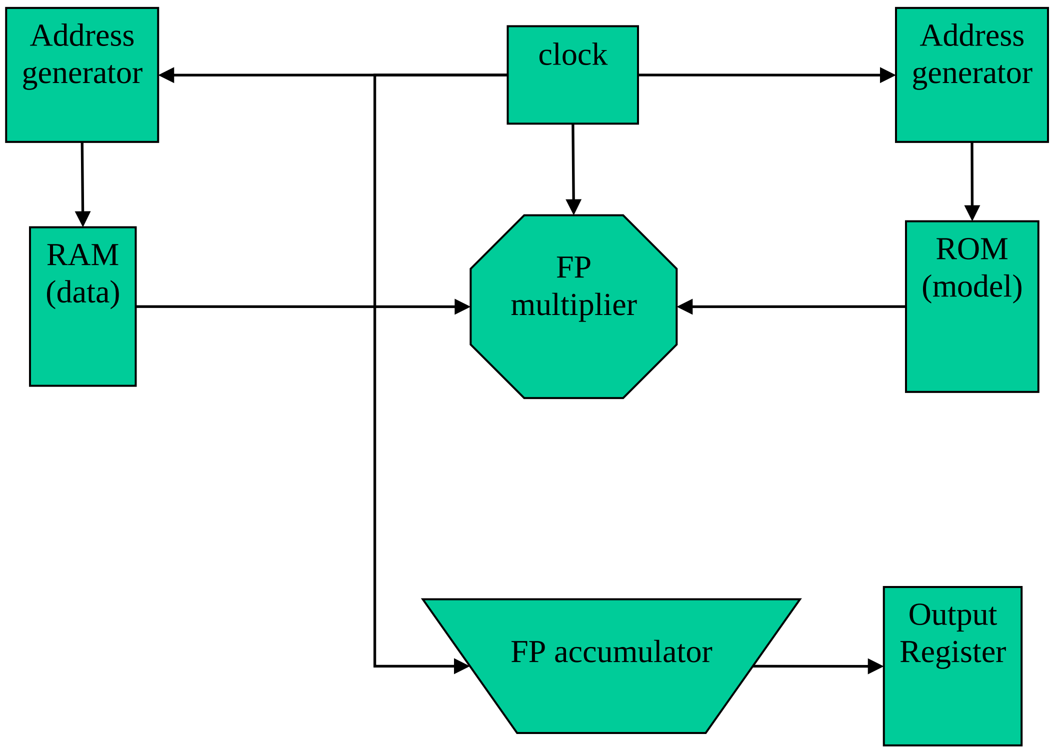
SVM based Face Detection

Face detection is an application where SVMs have demonstrated to be very effective and very efficient. There have been quite a few results that show that SVM based implementation outperform Neural nets.

The actual underlying problem is essentially a binary classification problem. We need to determine whether a give image is a face or non-face. Extracting features suitable for this problem is the main difference between this and any other classification problem. A very simple approach to face detection is to generate very simple features. A very common approach to generating suitable features for face detection is to perform histogram equalization on the image, concatenate the rows to form a single feature vector and then normalize the feature vector. This simple feature vector has worked quite well in many implementations. To be more robust to resolution and pose issues, a resolution pyramid could be used followed by affine transformation.

Once the feature vectors have been generated, the normal SVM training procedure can be employed. Features for the test images are generated in the same way as the training samples and are then classified using the trained SVM.

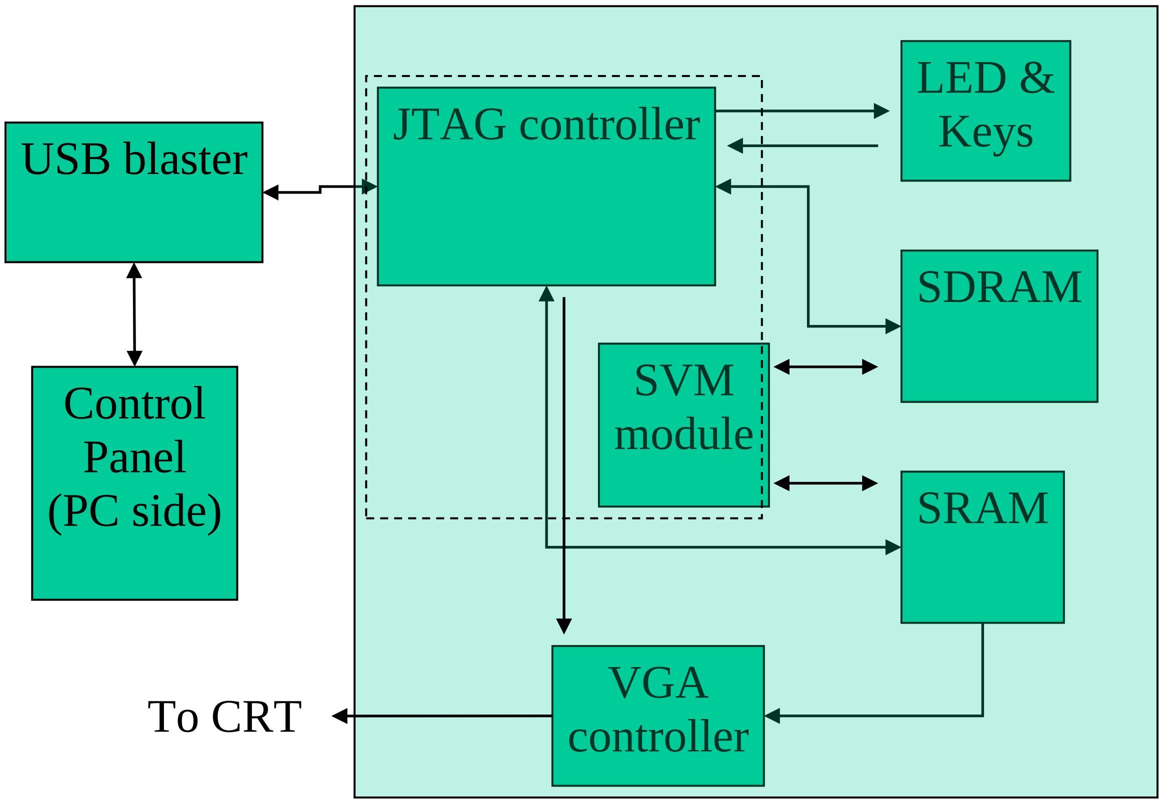
SVM Implementation



The above flowchart shows the basic idea behind the SVM implementation. Each incoming data vector is stored in a buffer, while the SVM model is stored in another buffer. The kernel computation unit performs the kernel computation and result accumulation for the vector. The result is then transferred to a register, where the MSB of the output is the class of the test vector.

If the data and the model is stored on off board memory, the IO is performed first to get them onboard, so as to have faster access. This of course depends on the size of the data set. If it is too large, then it is periodically buffered. The main computational units are the floating multiplier and the floating point accumulator units. Depending on the type of the kernel the multiplication is repeated as many times as required. In my case I implemented a linear and a quadratic kernel which is $(1+(X.Y))^2$. So in the linear case there are only N multiplications for each sample, where N is the number of features in the feature vector, while in the quadratic case there will be 2N multiplications for each sample. If other kernels are to be used, a Look up table based implementation is more suitable. For example if a Radial basis kernel is to be used a sufficiently large LUT containing pre-computed Gaussian values can be used. Once the individual multiplications are completed, they are summed up in the accumulator and the bias term is added to it to get the output. So for each sample there will be N+1 floating point additions. Therefore for each sample, in the linear kernel case N multiplications and N+1 additions are required.
This basic implementation was implemented as a schematic in Quartus.

Implementation on the DE1 board



The schematic of the Altera DE1 board is shown above. The basic code structure in verilog has been provided. This code includes the JTAG interface and the USB blaster. The control panel software communicates to the DE1 board through the USB blaster. The control panel can read from the memories SDRAM and SRAM and can also control the VGA controller.

For the DE1 board I implemented the code in verilog. I started with the DE1 control panel code and then added multiplexed interfaces to the SDRAM and SRAM and also setting up access for the VGA control from the SRAM.

The structure of the code is as follows
1) Store the input data (test image in the face detection) and the model on the SDRAM.
2) The output data is stored on the SRAM.
3) The SVM module is the main code that reads in the data and performs SVM computations and write the output to memory.
4) In the face detection case, the input image is stored is in the SDRAM and also the SRAM
5) The SVM module then reads in a 20x20 image from the SDRAM and then creates the feature vector as described previously.
6) The SVM computations are performed on this test vector and if it is detected to be a face region, a + is marked in the centre of the corresponding window on the SRAM.
7) The algorithm continues by scanning the image in row order.

There were quite a few problems with the verilog implementation. The problems start with verilog, since it is not well suited for complex computations. It started with accessing the data from memories, since I started out with a behavioral code instead of more suitable structural verilog code (I put it down to my largely C and software background !). This was causing bus contentions and timing errors as result the data was not being read properly. So I slowly moved parts of the code into structural and finally the data access and read and write was successful. Then started the problems with the floating multiplications and additions. I used the Altera fpmult and Altera fpadd_sub code for the floating point multiplications and additions. These codes have their own clock delays which needed to incorporated into the timing. Again, my way of handling this problem was a typical C code kind of verilog code with a lot of if-else nested structures which didn't seem to work. As a result the code is still not working. I'm still working on modifying the code, basically moving it to better structural verilog design in which the timing is lot more easy to handle. A better option which I haven't explored much, would be to move to SystemC.

Another change I feel I need to make is getting rid of the floating point computations by moving to a logarithmic number system (LNS). Timing issues are less severe in integer arithmetic than in floating point. Thus LNS should work better than floating point. For addition in LNS, a lookup table system can be implemented. But one draw back of lookup tables is that we need to know the dynamic range of data to determine the size of the lookup tables.

The part of the code for the face detection was again beset with the same kind of problems. The problem seems to be with the addressing for getting the 20x20 image from the memory. Since the image is stored sequentially in the memory an addressing algorithm needs to be used to access the pixel that form the 20x20 window. My current code for this is causing bus contentions, i.e the memory is not being in the read in the order I want. And added to this problems with the fpmult and fpadd_sub as described in the previous paragraph, this code is not working yet.

Results

I have the basic results for the schematic implementation. The input dataset used was the Iris dataset from the UCI machine learning repository. The data consists of measurement of the some features of the petals of the Iris flower to determine whether it is of the family Iris-setosa or Iris-vestibula. The data consists of 150 samples. Each sample has 4 real valued features.
The software SVM implementation was done using freely available svmlight software. Svmlight is a commonly used software that is very fast and works quite well for small to medium sized data sets.

Kernel type	Hardware	Software
Linear	92.3%	100%
Quadratic	88.1%	100%

The above table gives the accuracy of classification for the Iris dataset. We see that the accuracy for the software is 100%, i.e the 2 classes are perfectly separable. The hardware accuracy is significantly less, which I put down mainly to the timing issues. Due to the timing glitches the model and data features go out of sync, which causes the wrong data to be multiplied and thus results in the wrong result.

The verilog code for the SVM module is added in the appendix.

Conclusion

The project did not work as well as I expected. The schematic SVM implementation worked only reasonably well, but the main face detection part did not work at all, which was a huge disappointment. However there are some positives that I draw from this project. Firstly this was the first time since I have implemented anything this big in verilog and in hardware. So the experience of implementing a complete project was very useful. It also gave me a lot of insights into the vast differences between implementing in hardware and software. Nothing can be taken for granted when working with hardware, where even a small glitch can cause huge errors. Clock synchronization is vital when working with hardware and therefore a lot of care needs to be taken when working with behavioral verilog code.

I plan to fix the bugs in my code and get it to work as it should. A future improvement could be implementing this in SystemC which is much better than verilog. Once it is implemented successfully, the face detection algorithm can be easily extended to face recognition, which has better than utility than just face detection

References

- **Hardware-based support vector machine classification in logarithmic number systems** Khan, F.M.; Arnold, M.G.; Pottenger, W.M. Circuits and Systems, 2005. ISCAS 2005
- Bernhard Schölkopf, Christopher J.C. Burges, and Alexander J. Smola (editors). "**Advances in Kernel Methods: Support Vector Learning**". MIT Press, Cambridge, MA, 1999

Code references:
Altera corporation
Altera DE1 board cd

Appendix A:

svm_module.v

```
module
svm_module(iSelect_SR,iSelect_SDR,iDATA,oDATA,oADDR_SDR,oADDR_SR,islow
CLK,ifastCLK,oLED_GREEN,key3,oLED_RED,key2);

//Fixed for reading from SDRam and write to SRAM

input [1:0] iSelect_SR;
input [1:0] iSelect_SDR;
output reg [21:0] oADDR_SDR;
output reg [17:0] oADDR_SR;
output reg [15:0] oDATA;
input [15:0] iDATA;

input islowCLK;
input ifastCLK;
input key3;
input key2;

output reg [8:0]  oLED_GREEN;
output reg [17:0] oLED_RED;

parameter MAX_COUNT = 1000;
parameter MAX_ADDR = 10;
reg [15:0] idata;

parameter num_test_points = 30;
parameter vec_len = 3;
parameter model_start_address = 481;
parameter BIAS = 32'h3fe88f6f; //-1*b;

reg model_loaded;
reg data_loaded;
reg [17:0] count1;
reg [17:0] count2;
reg [17:0] count3;
reg [31:0] model [vec_len:0];
reg [31:0] data [vec_len:0];
reg [31:0] temp;
reg [31:0] result;
wire [31:0] result1;
reg [15:0] model_count;
reg [15:0] data_count;
reg [15:0] mult_count;
reg [15:0] data_len;
wire [31:0] out;
reg [31:0] data1;
reg [31:0] data2;
reg [3:0] byte_count;

wire clk_input;
reg do_mult;
reg ALL_DONE;
reg result_ready;
reg [5:0] mult_tick;

assign clk_input = ifastCLK & do_mult;
assign result1 = out;


altfp_mult0  m1  (clk_input,data1,data2,out);

initial begin
count1 = model_start_address;
count2 = 0;
count3 = 0;
```

```
model_count = 0;
data_count = 0;
data_len = 0;
result = 0;
//result1 = 0;
mult_count = 0;
do_mult = 0;
ALL_DONE = 0;
mult_tick = 0;
result_ready = 0;
oLED_GREEN[0] = 0;
byte_count = 0;
end

//always @(posedge iCLK or negedge key3)
//always @(posedge key2 or negedge key3)
always @(posedge islowCLK or negedge key3)
begin
   if(posedge islowCLK)
         begin
                  count1 = model_start_address;
                  count2 = 0;
                  count3 = 0;
                  model_count = 0;
                  data_count = 0;
                  data_len = 0;
                  result = 0;
                  //result1 = 0;
                  mult_count = 0;
                  do_mult = 0;
                  ALL_DONE = 0;
                  mult_tick = 0;
                  result_ready = 0;
                  oLED_GREEN[0] = 0;
                  byte_count = 0;
         end
   else if(iSelect_SDR == 2 && iSelect_SR == 2 && !ALL_DONE)
   begin
      if(!model_loaded )
      begin
//              oADDR_SDR = count1 + model_start_address;
              oADDR_SDR = count1;

              if (byte_count == 0)
                 temp[31:16] = iDATA;
              else if (byte_count == 1)
```

```
          temp[15:0] = iDATA;

        //oLED_RED = iDATA;
        count1 = count1 + 1;
        //oADDR_SDR = count1;
        byte_count = byte_count + 1;
        if (byte_count == 2)
        begin
                                model[model_count]  = temp;
                                if (model_count >= vec_len)
                                        model_loaded = 1;

                                model_count = model_count+1;
                                byte_count = 0;
        end


end
         else if (!data_loaded )
         begin
        oADDR_SDR = count2;
        if (byte_count == 0)
          temp[31:16] = iDATA;
        else if (byte_count == 1)
          temp[15:0] = iDATA;

        oLED_RED = count2;
        count2 = count2 + 1;
        byte_count = byte_count + 1;
        if (byte_count == 2)
        begin
                                data[data_count]  = temp;
                                if (data_count >= vec_len)
                                begin
                                        data_loaded = 1;
                                        data_count = 0;
                                        data_len = data_len +1;
                                        //oLED_GREEN[6:1] = data_len;
                                end
                                data_count = data_count+1;
                                byte_count = 0;
        end

end
else if (data_loaded && model_loaded && !result_ready )
begin
```

```
                                        data1 = model[mult_count];
                                        data2 = data[mult_count];
                                        do_mult = 1;
//                      oLED_RED = result1;

                                        temp = temp + 1;
                                        temp = temp + 1;
                                        temp = temp + 1;
                                        result = result + result1 ;
                                        if (mult_count >= vec_len)
                                        begin
                                                do_mult = 0;
                                                result_ready = 1;
                                                data_loaded = 0;
                                                mult_count = 0;
                                        end
                                        mult_count = mult_count+1;
                                        do_mult = 0;




        end
                else if (result_ready)
                begin
                 oLED_RED = result[31:16];
                 result  = result + BIAS;
                 oADDR_SR = count3;
                 oDATA = result[31:31];
                 //oDATA = result;
                 count3 = count3 + 1;
                 if (data_len > num_test_points)
                 begin
                    ALL_DONE = 1;
                    oLED_GREEN[0] = 1;
                 end
                 result_ready = 0;
                 result = 0;
        end

    end
end

endmodule
```