Random Kernel Perceptron on ATTiny2313 Microcontroller

Nemanja Djuric
Department of Computer and Information Sciences, Temple University
Philadelphia, PA 19122, USA
nemanja.djuric@temple.edu

Slobodan Vucetic
Department of Computer and Information Sciences, Temple University
Philadelphia, PA 19122, USA
vucetic@ist.temple.edu

ABSTRACT
Kernel Perceptron is very simple and efficient online classification algorithm. However, it requires increasingly large computational resources with data stream size and is not applicable on large-scale problems or on resource-limited computational devices. In this paper we describe implementation of Kernel Perceptron on ATTiny2313 microcontroller, one of the most primitive computational devices with only 128B of data memory and 2kB of program memory. ATTyny2313 is a representative of devices that are popular in embedded systems and sensor networks due to their low cost and low power consumption. Implementation on microcontrollers is possible thanks to two properties of Kernel Perceptrons: (1) availability of budgeted Kernel Perceptron algorithms that bound the model size, and (2) relatively simple calculations required to perform online learning and provide predictions. Since ATTiny2313 is the fixed-point controller that supports floating-point operations through software which introduces significant computational overhead, we considered implementation of basic Kernel Perceptron operations through fixed-point arithmetic. In this paper, we present a new approach to approximate one of the most used kernel functions, the RBF kernel, on fixed-point microcontrollers. We conducted simulations of the resulting budgeted Kernel Perceptron on several datasets and the results show that accurate Kernel Perceptrons can be trained using ATTiny2313. The success of our implementation opens the doors for implementing powerful online learning algorithms on the most resource-constrained computational devices.

Categories and Subject Descriptors
C.3 [**Computer Systems Organization**]: Special purpose and application-based systems – *Real-time and embedded systems*

General Terms
Algorithms, Performance, Experimentation.

Keywords
Kernel Perceptron, Budget, RBF kernel, Microcontroller.

SensorKDD'10, July 25th, 2010, Washington, DC, USA.

1. INTRODUCTION
Kernel Perceptron is a powerful class of machine learning algorithms which can solve many real-world classification problems. Initially proposed in [6], it was proven to be both accurate and very easy to implement. Kernel Perceptron learns a mapping $f: \mathrm{X} \rightarrow \mathrm{R}$ from a stream of training examples $S = \{(\mathbf{x}_i, y_i), i = 1 \ldots N\}$, where $\mathbf{x}_i \in \mathrm{X}$ is an M-dimensional input vector, called the data point, and $y_i \in \{-1, +1\}$ is a binary variable, called the label. The resulting Kernel Perceptron can be represented as

$$f(\mathbf{x}) = sign(\sum_{i=1}^{N} \alpha_i K(\mathbf{x}, \mathbf{x}_i)), \quad (1)$$

where α_i are weights associated with training examples, and K is the kernel function. The RBF kernel is probably the most popular choice for Kernel Perceptron because it is intuitive and often results in very accurate classifiers. It is defined as

$$K(\mathbf{x}, \mathbf{x}_i) = \exp(-\frac{\| \mathbf{x} - \mathbf{x}_i \|^2}{A}), \quad (2)$$

where $\| \cdot \|$ is the Euclidean distance, and A is the positive number defining the width of the kernel.

Training of the Kernel Perceptron is simple: starting from the zero function $f(\mathbf{x}) = 0$ at time $t = 0$, data points are observed sequentially, and $f(\mathbf{x})$ is updated as $f(\mathbf{x}) \leftarrow f(\mathbf{x}) + \alpha_i \cdot K(\mathbf{x}_i, \mathbf{x})$, where $\alpha_i = y_i$, if point $\mathbf{x}_i$ is misclassified ($y_i \cdot f(\mathbf{x}_i) \leq 0$) and $\alpha_i = 0$ otherwise. Examples for which $\alpha_i = y_i$ have to be stored and they are named the support vectors. Despite the simplicity of this training algorithm, Kernel Perceptrons often achieve impressive classification accuracies on highly non-linear problems. On the other hand, number of the support vectors grows linearly with the number of training examples on noisy classification problems. Therefore, these algorithms require $\mathcal{O}(N)$ memory and $\mathcal{O}(N^2)$ time to learn in a single pass from N examples. Because number of examples N can be extremely high, Kernel Perceptrons can be infeasible on noisy real-world classification problems. This is the reason why budgeted versions of Kernel Perceptron have been proposed with the objective of retaining high accuracy while removing the unlimited memory requirement.

Budget Kernel Perceptron [4] has been proposed to address the problem of the unbounded growth in resource consumption with training data size. The idea is to maintain a constant number of support vectors during the training by removing a single support vector every time the budget is exceeded upon addition of a new

support vector. Given a budget of T support vectors, Budget Kernel Perceptrons achieve constant space scaling $\mathcal{O}(T)$, linear time to train $\mathcal{O}(TN)$ and constant time to predict $\mathcal{O}(T)$. There are several ways one may choose to maintain the budget. The most popular approach in literature is removal of an existing support vector to accommodate the new one. For example Forgetron [5] removes the oldest support vector, Random [3] the randomly selected one, Budget [4] the one farthest from the margin, and Tightest [13] the one that impacts the accuracy the least. There are more advanced and computationally expensive approaches such as merging of two support vectors [12] and projecting a support vector to the remaining ones [9].

Although Budget Kernel Perceptrons are very frugal, requiring constant memory and linear space to train, they are still not applicable to one of the simplest computational devices, microcontrollers. These devices have extremely small memory and computational power. In this paper we consider microcontroller ATTiny2313, which has only 128B of memory available to store the model, maximum processor speed of 8MHz and whose hardware does not support floating-point operations. We propose a modification of Budget Kernel Perceptron that uses only integers and makes it very suitable for use on resource-limited microcontrollers. Our method approximates floating-point Budget Kernel Perceptron operations using fixed-point arithmetic that provides nearly the same accuracy, while allowing much faster execution time and requiring less memory. We use *Random Removal* as budget maintenance method [3, 11]. When the budget is full and new support vector is to be included to the support vector set, one existing support vector is randomly chosen for deletion. Although this update rule is extremely simple, the resulting Perceptron can achieve high accuracy when the allowed budget is sufficiently large. The pseudocode of Budget Kernel Perceptron is given as Algorithm 1.

It should be noted that a significant body of research exists on the topic of efficient hardware implementation of various machine learning and signal processing algorithms. In Compressed Kernel Perceptron [11] the authors consider efficient memory utilization through data quantization, but assume floating-point processor. Several solutions have been proposed for implementation of kernel machines on fixed-point processors as well. In [1], authors propose special hardware suitable for budgeted Kernel Perceptron. Similarly, in [2] authors consider implementation of Support Vector Machines [10] on fixed-point hardware. The two proposed algorithms, however, assume that the classifiers are already trained. Additionally, use of the problem-specific hardware is limited to a single problem. We, on the other hand, implement our method on general-purpose hardware, which makes the implementation much easier and more cost-efficient. Moreover, we combine the two proposed approaches, quantization of data and the use of fixed-point hardware, to obtain an accurate classifier that can be used on the simplest existing computational devices.

The paper is organized as follows. In Section 2 the proposed method is explained in detail. In Section 3 the results are presented and discussed. Finally, Section 4 concludes the paper.

Algorithm 1 - *Budget Kernel Perceptron*

Inputs : *data sequence* $((\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N))$, *budget* T
Output : *support vector set* $SV = \{SV_i, i = 1 \ldots I\}$

$I \leftarrow 0$; $i \leftarrow 1$
$SV = \emptyset$
for $i = 1 : N$
{
  if $\left((y_i \cdot \sum_{j=1}^{I} y_j \cdot K(\mathbf{x}_i, \mathbf{x}_j)) \le 0 \right)$
  {
    if $(I == T)$
      $new = random(I)$
    else
    {
      $I \leftarrow I + 1$
      $new \leftarrow I$
    }
    $SV_{new} = (x_i, y_i)$
  }
}

2. METHODOLOGY

In this paper we focus on implementation of Kernel Perceptron on the specific microcontroller. We first describe its properties and then explain the proposed algorithm.

2.1 Microcontroller

The microcontroller ATTiny2313 [14] has been chosen as the target platform because its specifications make it extremely challenging to implement a data mining algorithm. It has very limited speed (0-8 MHz, depending on voltage that ranges from 1.8 to 5.5V), low power requirements (when in 1.8V and 1MHz mode, it requires only 300μA and 540μW power), and very limited memory (2kB of program memory that is used to store executable code and constants; 128B of data memory that is used to store model and program variables). Because of its limitations it is also very cheap, with the price of around 1$ [15]. It supports 16-bit integers and supports fixed-point operations. Multiplication and division of integers is very fast, but at the expense of potential overflow problems with multiplication and round-off error with division. ATTiny2313 does not directly support floating-point numbers, but can cope with them using certain software solutions. Floating-point calculations can also be supported through C library, but only after incurring significant overhead in both memory and execution time.

2.2 Attribute quantization

In order to use the limited memory efficiently the data are first normalized and then quantized using B bits, as proposed in [11]. Using quantized data, instead of (1), the predictor is

$$f(\mathbf{x}) = sign(\textstyle\sum_i \alpha_i K(q(\mathbf{x}), q(\mathbf{x}_i))), \quad (3)$$

where $q(\mathbf{x})$ is the quantized data point $\mathbf{x}$. Quantization of data introduces quantization error, which can have significant impact on the classification, especially for the data points that are located near the separating boundary. The quantization loss suffered by the prediction model due to quantization is discussed in much more detail in [11]. Objective of this paper is to approximate (3) using fixed-point arithmetic.

2.3 Fixed-point method for prediction

Our algorithm should be implemented on microcontroller with extremely low memory. Although quantization can save valuable memory space, there are additional memory-related issues that must be considered. First, number of program variables must be minimal. Prudent use of variables allows more space to store the model and leads to increased accuracy. On the computational side, all unnecessary operations, such as excess multiplications and exponentials should be avoided. If this is not taken into consideration, the program can take a lot of memory and a lot of time to execute. Furthermore, due to inability of ATTiny2313 to deal with floating-point numbers in hardware, use of floating-point library results in even larger program size.

Instead of RBF kernel (2) our algorithm uses its modified version, as proposed in [2],

$$K(\mathbf{x},\mathbf{x}_i) = e^{-\frac{|\mathbf{x}-\mathbf{x}_i|}{2^A}}, \quad (4)$$

where several differences from RBF kernel in (2) can be noticed. First, instead of Euclidean distance, Manhattan distance is used. In [2], kernel function with Manhattan distance is used because multiplications can be completely omitted from their algorithm. For our application, we use the Manhattan distance to limit the range of distances between the quantized data, which will lead to improved performance of our algorithm. Furthermore, without the loss of generality, we represent the kernel width as 2^A, where A is any number.

Although the simplifications in equation (4) have been introduced, the algorithm cannot be implemented without the penalty in the speed of computation. This is because very simple devices are not designed to compute exponents or other operations involving floating-point arithmetic efficiently. This results in slow run-time and excessive memory usage. Therefore, an alternative way needs to be devised to perform kernel calculation. The idea of our method is to use only integers, which are handled easily by even the simplest devices, and still manage to use the RBF kernel for predicting the label of new point.

The normalized data point $\mathbf{x}$ is quantized using B bits and its integer representation $I(\mathbf{x})$ is

$$I(\mathbf{x}) = round(\mathbf{x} \cdot 2^B). \quad (5)$$

The value of $I(\mathbf{x})$ is in the range $[0,\ 2^B - 1]$. Since $\mathbf{x}$ is now approximated by $q(\mathbf{x}) = I(\mathbf{x})\, 2^{-B}$, we can define kernel function which takes quantized data

$$K_q(\mathbf{x},\mathbf{x}_i) \overset{\Delta}{=} K(q(\mathbf{x}), q(\mathbf{x}_i)) = e^{-\frac{|I(\mathbf{x})-I(\mathbf{x}_i)|}{2^{A+B}}}. \quad (6)$$

If we represent $|I(\mathbf{x}) - I(\mathbf{x}_i)|$ as d_i, and introduce for the simplicity of notation

$$g = e^{-\frac{1}{2^{A+B}}}, \quad (7)$$

we get

$$K_q(\mathbf{x},\mathbf{x}_i) = g^{d_i}. \quad (8)$$

We need to calculate $sign(\sum_{i=1}^{T} y_i \cdot K_q(\mathbf{x},\mathbf{x}_i))$, where T is the budget.

Since $sign(\sum_{i=1}^{T} y_i K_q(\mathbf{x},\mathbf{x}_i)) = sign(\sum_{i=1}^{T} y_i c K_q(\mathbf{x},\mathbf{x}_i))$ for any $c > 0$, we can replace g^{d_i} from (8) with $Cg^{d_i - d_{nn}}$, where d_{nn} is the distance between newly observed data point $\mathbf{x}$ and the nearest support vector and C is a positive integer. Then, to allow integer representation, we again replace it with integers

$$w(d_i - d_{nn}) \overset{\Delta}{=} round(C \cdot g^{d_i - d_{nn}}), \quad (9)$$

and approximate $K_q(\mathbf{x},\mathbf{x}_i) \approx w(d_i - d_{nn})$. It is clear that if $d_i = d_{nn}$ then $w = C$, if $d_i - d_{nn} = 1$ then $w = round(C\, g)$, and if $d_i - d_{nn}$ is sufficiently large then $w = 0$. We can notice that if the i-th support vector is close to data point $\mathbf{x}$ its weight w and its influence on classification will be large. Our final classifier is approximated as

$$f(\mathbf{x}) = sign(\sum_{i=1}^{T} y_i \cdot w(d_i - d_{nn})), \quad (10)$$

where y_i is the label of i-th support vector, and w is its weight which depends on the distance d_i from data point $\mathbf{x}$. The drawback of the proposed method is that the support vector set needs to be scanned every time a new data point $\mathbf{x}$ arrives in order to find the nearest support vector distance. In addition, there is a computational problem related to calculation of weights. These issues will be addressed in Section 2.4.

2.3.1 Error of weight approximation

By using rounding operation (9) we are inevitably introducing additional error, on top of the error made due to quantization of data point using B bits [11]. In order to understand the effect of the approximation error, let us define the relative error R as

$$R = \frac{|true\_value - approximation|}{true\_value}, \quad (11)$$

where *true_value* is the actual value of the number, and *approximation* is value of the number after rounding. It can be shown [7] that the relative error R we are making by the rounding (9) depends on the parameter C as $R \sim 1/C$. This shows that large C leads to small relative error. However, because the microcontroller supports only 16-bit integers, too large C will render our method useless on the chosen platform.

2.4 Weight calculation

To be able to use predictor (10) there is a need to calculate w_i for every support vector. For given parameters (A, B, C) we can calculate the weight for every possible distance $d = d_i - d_{nn}$ as illustrated in Table 1. In the naïve approach, we can precalculate all the weights and save them as array in microcontroller memory. If the dataset is M-dimensional, then d is in range $[0,\ M(2^B - 1)]$, and storing each weight could become impossible.

Table 1 Distances and corresponding weights

Distance (d)	0	1	...	d	…
Weight ($w(d)$)	Cg^0	Cg^1	...	Cg^d	…

Therefore, we must find a way to represent entire array of weights with only a subset of weights in order to save memory. Two

approaches are discussed next, both requiring very limited memory.

2.4.1 Sequential method

The most obvious way is to store weights $w(0)$ and $w(1)$, for distances 0 and 1, respectively. By storing these two weights we can calculate other weights in Table 1. Every other weight can be iteratively calculated, in integer calculus, as

$$w(d) = \frac{w(d-1)^2}{w(d-2)}. \quad (12)$$

The memory requirement of this approach is minimal, but there are certain problems. In order to calculate particular weight we need to calculate all the weights up to that one by repeatedly applying (12), which can take substantial time. In addition, the division of two integers results in rounding error that would propagate and increase for large d.

2.4.2 Log method

Different idea is to save only the weight of distance 0, which equals C, together with the weights at distances that are the power of 2, namely weights of distances 1, 2, 4, 8, 16 and so on. In this way the number of weights we need to save is logarithm of total number of distances, which is approximately $log(M\,2^B)$. Using the saved weights we calculate weight for any distance using Algorithm 2.

Algorithm 2 - *Find Weight*

Inputs : *array of weights W and corresponding distances D*
distance d for which the weight is being calculated
Output : *weight w*

i ← *Index of the nearest smaller distance to d in array D*
w ← *W*[*i*]
d ← *d* – *D*[*i*]
while (*d* > 0)
{
  i ← *Index of the nearest smaller distance to d in array D*
  w ← *w* · *W*[*i*] / *W*[0]
  d ← *d* – *D*[*i*]
}

As can be seen in Algorithm 2, if we are looking for the weight of the distance d that is the power of 2, the weight will immediately be found in the array W. If not, we are looking for the distance in array D that is the closest smaller value than d, and start the calculation from there. We repeat the process until convergence. In this way, we will always try to make the smallest error while calculating the weight, and the algorithm runs using $\mathcal{O}(log(M\,2^B))$ space and $\mathcal{O}(log(M\,2^B))$ time, which is efficient and fast even for highly-dimensional data and large bit-lengths.

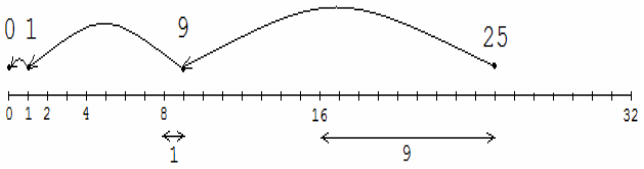

Figure 1 Illustration of *log* method

Log method is illustrated in Figure 1. Assume that we are looking for the weight of distance 25, and we saved weights at distances 0, 1, 2, 4, 8, 16 and 32. It is clear that 25 can be represented as 16 + 8 + 1. In the first step we will use $w(16)$, then $w(8)$, because 8 is the closest to 25−16, and finally $w(1)$. The sought-after weight will be found both very quickly and with minimal error.

3. EXPERIMENTAL RESULTS

We evaluated our algorithm on several benchmark datasets from UCI ML Repository whose properties are summarized in Table 2. The digit dataset *Pendigits*, which was originally multi-class, was converted to binary dataset in the following way: classes representing digits 1, 2, 4, 5, 7 (non-round digits) were converted to negative class, and those representing digits 3, 6, 8, 9, 0 (round digits) to the positive class. *Banana* and *Checkerboard* were originally binary class datasets. *Checkerboard* dataset is shown in Figure 8.

Table 2 Dataset summaries

Dataset	Training set size	Testing set size	Number of attributes
Banana	4800	500	2
Checkerboard	3500	500	2
Pendigits	2998	500	16

In all reported results, A is the kernel width parameter, B is bit-length, and C is positive integer, all defined in Section 2.3.

In Figure 2 the absolute difference between the true weights and the calculated ones using *sequential* and *log* methods for weight calculation described in Sections 2.4.1 and 2.4.2 is shown. It can be seen that *log* method makes very small error with minimal memory overhead over the whole range of distances. We use the *log* method as our default for weight calculation.

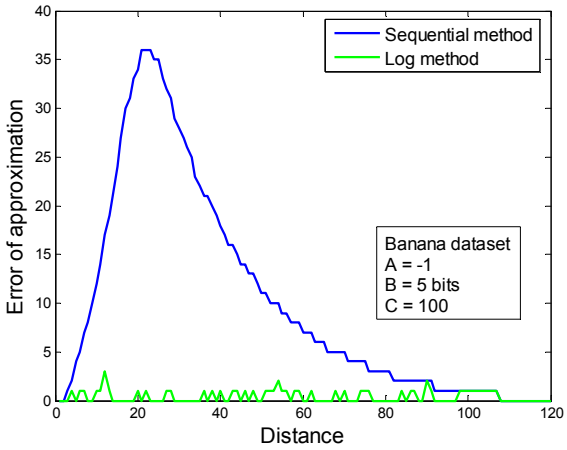

Figure 2 Error of two weight calculation methods

In the next set of experiments the quality of approximation was evaluated. First, the Random Kernel Perceptron was trained using equation (6) that requires floating-point calculations. Then, our fixed-point method was executed, and the predictions of the two algorithms were compared. The approximation accuracy is defined as

$$accuracy = \frac{\sum_{i=1}^{N} I(y_i^{fixed}, y_i^{floating})}{N}, \quad (13)$$

where I is the indicator function that is equal to 1 when the predictions are the same and 0 otherwise, y_i is the classifier prediction, and N is the size of the test set.

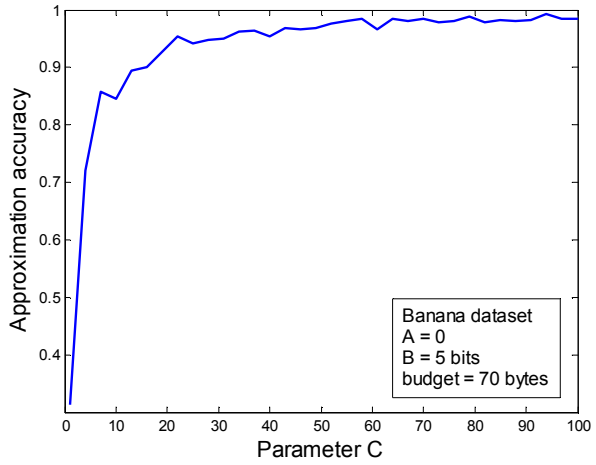

Figure 3 Approximation accuracy as the function of *C*

Figure 3 shows that the approximation accuracy is relatively large even for small values of C and that it quickly increases with C, as expected from discussion in Section 2.3.1. Since ATTiny2313 microcontroller supports 16-bit integers, the maximum value of a number is 65535. However, as can be seen from Algorithm 2, in order to calculate weights multiplications are required, and at no point the product should exceed this maximum value. Therefore, the best choice for C is the square root of 65535, which is 255. In the following experiments C was set to 255.

In Figure 4 the approximation accuracy is given for 3 datasets. Total budget was fixed to only 70 bytes. Each dataset was shuffled before each experiment and the reported results are averages after 10 experiments. The value of parameter A was changed in order to estimate its impact on the performance. High negative values of A correspond to the algorithm that acts like nearest neighbour. High positive values of parameter A correspond to the majority vote algorithm. It can be seen that over large range of A values the approximation accuracy was around 99%, and that is was lower at the extreme values of A. This behavior is due to the numerical limitations of double-precision format used when calculating equation (6). The exponentials of large negative numbers are too close to 0 and are rounded to 0, which results in much smaller accuracy. On the other hand, it can be seen that for large values of A corresponding to the algorithm that acts like majority vote the approximation accuracy starts to decrease. This was expected, because in this case all the weights, as calculated by our method, are practically the same since the weights decrease extremely slowly and very small differences between actual weights are lost when they are calculated using the *log* method. However, it should be noted that in this problem setting kernel width depends on A as 2^A, and the values where prediction accuracy starts to fall are practically never used. Experiments were conducted for different values of parameter B as well. However, the results for other bit-lengths were very similar, and are thus not shown.

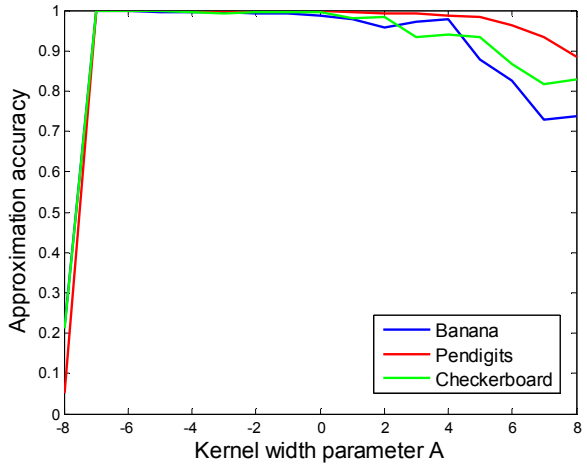

Figure 4 Approximation accuracy (*B* = 5 bits)

The next set of experiments was conducted to evaluate prediction accuracy of the proposed Random Kernel Perceptron implementation. The values of parameters A and B were changed in order to estimate their influence on the accuracy of methods. The results are the averages over 10 experiments and are reported in Figures 5, 6 and 7.

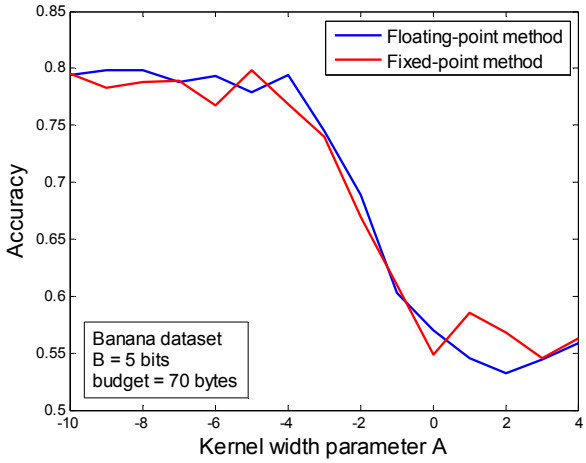

Figure 5 Accuracy of two methods

It can be seen that both floating- and fixed-point implementations achieve essentially the same accuracy, as was expected since the approximation has been proven to be very good. In some cases for very small values of A the accuracy of floating-point method drops sharply, as in Figure 6. This happens because of the numerical problems, where the predictions are so close to 0 that when calculated in double-precision they are rounded to 0. Our method does not have this problem. It is also worth noticing that for large A the accuracy drops sharply for both algorithms, which supports the claim that large kernel widths are usually not useful. It is exactly for these large values of A that the approximation accuracy drops as well. Therefore, as can be seen in Figures 5, 6 and 7, these values are not of practical relevance. Only results for bit-length of 5 bits were shown, since for the other bit-lengths the results were nearly the same.

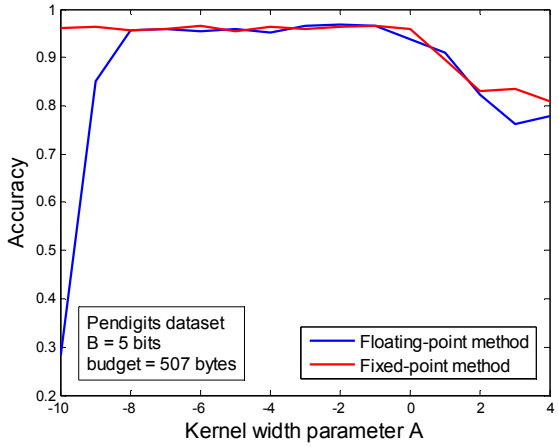

Figure 6 Accuracy of two methods

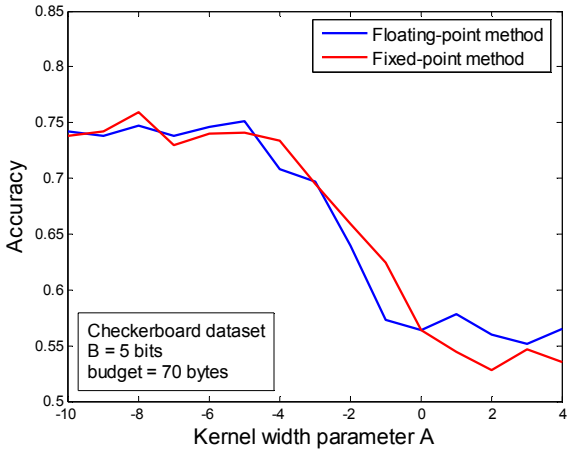

Figure 7 Accuracy of two methods

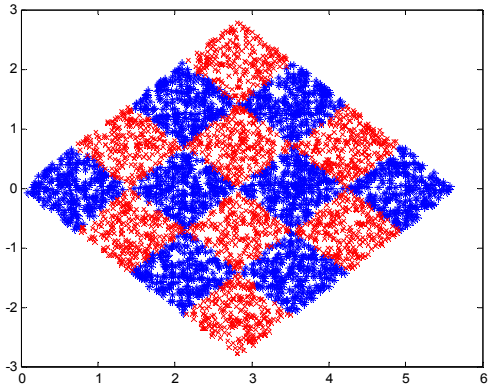

Figure 8 *Checkerboard* dataset

3.1 Hardware implementation

The algorithm was tested on microcontroller ATTiny2313. The program was written in C programming language using free IDE *AVR Studio 4* available for download from Atmel's website.

Different methods were used in order to assess the complexity of implementation on microcontroller. The details are given in Tables 3, 4 and 5. *Program* and *Data* represent the required program and data memory in bytes. *Time*, in milliseconds, is given after 100 observed examples, and kernel width parameter A is set to -6. Accuracy is calculated after all data points were observed. Numbers in the brackets represent the total size of memory block of the microcontroller. The microcontroller speed is set to 4 MHz. Number of support vectors was chosen so that the entire data memory of microcontroller is utilized in the case of fixed-point method. This number of support vectors was then imposed on both methods. Therefore, as the bit-length is increased, the number of support vector decreased.

For Tables 3 and 4 the target microcontroller was ATTiny2313. For Table 5 microcontroller with twice bigger memory was used because of the high dimensionality of *Pendigits* dataset. ATTiny48 was chosen which has 4kB of program memory and 256B of data memory.

It is clear that proposed fixed-point algorithm takes much less memory and is faster than the floating-point algorithm. Both time and space costs of fixed-point algorithm are nearly 3 times smaller. In fact, in order to run simulations using floating-point numbers, microcontroller ATTiny84 with four times larger memory was used since the memory requirement was too big for both ATTiny2313 and a more powerful ATTiny48, colored red in the tables. The accuracy of these two methods is practically the same, and the difference in computational cost is therefore even more striking. In addition, it can be concluded that accuracy depends greatly on the number of support vectors. The higher number of support vectors usually means higher accuracy. However, in our experiments, higher number of support vectors is achieved at the expense of smaller bit-lengths, so the results in the tables can be misleading. In some experiments larger support vector set does not necessarily lead to better performance because the quantization error is too big and this affects predictions considerably. It should be noted that the support vector set size/bit-length trade-off is not the topic of this work and will be addressed in our future study.

4. CONCLUSION

In this paper we proposed approximation of Kernel Perceptron that is well-suitable for implementation on resource-limited devices. The new algorithm uses only integers, without any need for floating-point numbers, which makes it perfect for simple devices such as microcontrollers.

The presented results show that the approach considered in this paper can be successfully implemented. The described algorithm approximates the kernel function defined in (6) with high accuracy, and yields good results on several different datasets. The results are, as expected, not perfect, which is the consequence of the highly limited computation capabilities of ATTiny2313 that required several approximations in calculation of the prediction. While the quantization and approximation error limit the performance, the degradation is relatively moderate considering very limited computational resources. The simulations on ATTiny microcontrollers proved that the algorithm is very frugal and fast, while being able to match the performance of its unconstrained counterpart.

Although the kernel function defined in (6) is used, which is a slightly modified RBF kernel, the fixed-point method can also be used to approximate the original RBF kernel (2). The proposed idea could probably be extended to some other kernel functions

Table 3 Microcontroller implementation costs

Banana	*2 bits*		*4 bits*		*6 bits*		*8 bits*	
ATTiny2313	***Fixed***	***Float***	***Fixed***	***Float***	***Fixed***	***Float***	***Fixed***	***Float***
Program [B] (2048B)	1744	6036	1720	6012	1720	6012	1748	6040
Data [B] (128B)	128	379	128	379	128	379	128	381
Time [ms]	1192	6604	1985	7505	1883	7610	1739	7496
Accuracy [%]	**67.32**	**65.12**	**81.08**	**81.00**	**79.36**	**79.60**	**78.00**	**77.80**
# of SVs	112		62		43		32	

Table 4 Microcontroller implementation costs

Checkerboard	*2 bits*		*4 bits*		*6 bits*		*8 bits*	
ATTiny2313	***Fixed***	***Float***	***Fixed***	***Float***	***Fixed***	***Float***	***Fixed***	***Float***
Program [B] (2048B)	1744	6036	1720	6012	1720	6012	1748	6040
Data [B] (128B)	128	379	128	379	128	379	128	381
Time [ms]	1568	7626	2226	6725	2410	7366	2232	7703
Accuracy [%]	**52.20**	**51.20**	**72.60**	**72.64**	**78.20**	**78.60**	**74.04**	**73.80**
# of SVs	112		62		43		32	

Table 5 Microcontroller implementation costs

Pendigits	*2 bits*		*4 bits*		*6 bits*		*8 bits*	
ATTiny48	***Fixed***	***Float***	***Fixed***	***Float***	***Fixed***	***Float***	***Fixed***	***Float***
Program [B] (4096B)	1836	6078	1810	6058	1812	6056	1816	5912
Data [B] (256B)	256	513	256	511	256	507	256	503
Time [ms]	3280	14186	3572	15352	4881	17102	5063	17373
Accuracy [%]	**93.80**	**94.20**	**92.76**	**93.92**	**86.44**	**86.32**	**80.72**	**81.68**
# of SVs	46		23		15		11	

that require operations with floating-point numbers, such as polynomial kernel. This would be of great practical importance for problems that do not work well with the RBF kernel. This issue will be pursued in our future work.

The proposed algorithm could be improved by some more advanced budget maintenance method, such as the one described in [8]. By using support vector merging, the performance of predictor would certainly improve, but it is to be seen if the limited device can support such an approach. Also, in this paper we assumed that parameters A and B, kernel width and bit-length, respectively, are given. Although this is reasonable if we want to use this method for applications where the classification problem is well understood, it would be interesting to implement on a resource-limited device an algorithm that can automatically find the optimal parameter values.

5. ACKNOWLEDGMENTS

This work is funded in part by NSF grant IIS-0546155.

6. REFERENCES

[1] Anguita, D., Boni, A., Ridella, S., Digital Kernel Perceptron, *Electronics letters*, 38: 10, pp. 445-446, 2002.

[2] Anguita, D., Pischiutta, S., Ridella, S., Sterpi, D., Feed–Forward Support Vector Machine without multipliers, *IEEE Transactions on Neural Networks*, 17, pp. 1328-1331, 2006.

[3] Cesa-Bianchi, N., Gentile, C., Tracking the best hyperplane with a simple budget Perceptron, *Proc. of the Nineteenth Annual Conference on Computational Learning Theory*, pp. 483-498, 2006.

[4] Crammer, K., Kandola, J., Singer, Y., Online Classification on a Budget, *Advances in Neural Information Processing Systems*, 2003.

[5] Dekel, O., Shalev-Shwartz, S., Singer, Y., The Forgetron: A kernel-based Perceptron on a fixed budget, *Advances in Neural Information Processing Systems*, pp. 259-266, 2005.

[6] Freund, Y., Schapire, D., Large Margin Classification Using the Perceptron Algorithm, *Machine Learning*, pp. 277-296, 1998.

[7] Hildebrand, F. B., Introduction to Numerical Analysis, 2nd edition, Dover, 1987.

[8] Nguyen, D., Ho, T., An efficient method for simplifying support vector machines, *Proceedings of ICML*, pp. 617–624, 2005.

[9] Orabona, F., Keshet, J., Caputo, B., The Projectron: a bounded kernel-based Perceptron, *Proceedings of ICML*, 2008.

[10] Vapnik, V., The nature of statistical learning theory, Springer, 1995.

[11] Vucetic, S., Coric, V., Wang, Z., Compressed Kernel Perceptrons, *Data Compression Conference*, pp. 153-162, 2009.

[12] Wang, Z., Crammer, K., Vucetic, S., Multi-Class Pegasos on a Budget, *Proceedings of ICML*, 2010.

[13] Wang, Z., Vucetic, S., Tighter Perceptron with Improved Dual Use of Cached Data for Model Representation and Validation, *Proceedings of IJCNN*, 2009.

[14] www.atmel.com/dyn/resources/prod_documents/doc2543.pdf ATTiny2313 Datasheet

[15] www.digikey.com