Dropout for Kernel Adaptive Filters

Eder Santana

***Abstract*— This paper presents a method for improving the generalization performance of the Kernel Adaptive Algorithms. We provide a general method and its applications at Kernel Least Mean Squares (KLMS) on a regression task using spike train data. We achieve this by randomly *droping out* part of the kernel machine active set for calculating its output for a given training sample and training only this random section at this time. This technique was inspired on the Dropout method, proposed by Hinton et. al. Here we also observe the improvement on generalization and also note that this technique accelerates the training of large kernel methods.**

Keywords—Kernel Methods, KLMS algorithm, spike train decoding, dropout.

I. INTRODUCTION

Kernel methods are known to have wide applicability and competitive results through several fields ranging from genetics [1] and neuroscience [2] to computer science [3] and economics [4]. Those methods are based in defining a feature detector for each sample in their active set and using them in the *kernel trick*. It is the large size of some feature detector datasets that produces both the best results and biggest practical problem of kernel machines.

Large datasets are difficult to handle because they ask for large memory capacities and large computations. For example, the Forgetron [5] that is a Kernel Perceptron which "forgets" a feature detector with small contributions to the whole output when a new elements must be added to the active set.

An equivalent example is the Random Budget Perceptron [6], which is like the Forgetron but forgets the a random element from its active set instead. There is also the Projectron [7] which limits the budget by accepting a feature detector only if it most of it's contribution can't be represented by a random element from its active set instead. On the other hand, for limiting the budget of the KLMS algorithm it was proposed a feature space quantization in the Quantized-KLMS [8][9]. The criterion of novelty and surprise had also been utilized for judging if a new sample should be admitted to the active set.

Nevertheless, those methods don't always keep the kernel machine from *overfitting* because all the elements in the active set are involved in complex co-adaptation. Beyond that, some problems allows the budget to be fixed, but requires this to be sufficiently large to achieve good results. At the end, what the kernel machine will provide is a computation and memory hungry program that is too specific to generalize well.

A similar problem also appears in the development of large deep neural networks [10]. Sometimes the net architecture necessary to achieve good results is large and only small training sets are available. In those scenarios different feature detectors of the network will adapt to represent parts of the training set and together they sometimes perform poorly on help-out test data. Hinton et. al. [10] observed that the *overfitting* of those networks can be diminished using *dropout* during training. *Dropout* can be described as randomly omitting hidden units of the network during training with a probability p, where usually $p = 0.5$. In equations, we can describe the dropout technique at a given layer of a neural network as:

$$y_{train} = \mathbf{m} \star \sigma(\mathbf{W}\mathbf{v}), \tag{1}$$

where the output vector of a layer, y, is given by the element wise product of a binary mask vector, $\mathbf{m}$, with the activation function σ. Also, $\mathbf{v}$ is the previous layer output vector, and $\mathbf{W}$ are connection weights. This way, a *different* network is trained for each training sample. At test time, the full network, or an average of all the *different* networks trained, should be used. Since twice as must outputs are used during test, Hinton et. al. [10] proposed to estimate the average network by multiplying the activation functions by p, thus:

$$y_{test} = p\sigma(\mathbf{W}\mathbf{v}). \tag{2}$$

In this paper we extend the technique described above to the KLMS algorithm, nevertheless, we believe our approach can be easily adopted at any other online kernel method, such as the Kernel Perceptron for example. To do this, we organized the reminder of the paper as follows. Section 2 is devoted to review the KLMS algorithm [11] and derive the Dropout-KLMS. In Section 3 we show some experiments on spike train signal processing to validate our method. We conclude the paper in Section 4.

II. DROPOUT-KLMS ALGORITHM

Given a set of pairs $\{\mathbf{x}_i, \mathbf{y}_i\}_{\{i=1\}}^{N}$ the Least Mean Squares (LMS) algorithm adapts a linear transformation

$$\bar{y}_i = \mathbf{w}^T\mathbf{x}_i, \tag{3}$$

to minimize the mean squares $J = E[(y_i - \bar{y}_i)^2]$ by following the contrary direction of the stochastic gradient as

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta e_i \mathbf{x}_i. \tag{4}$$

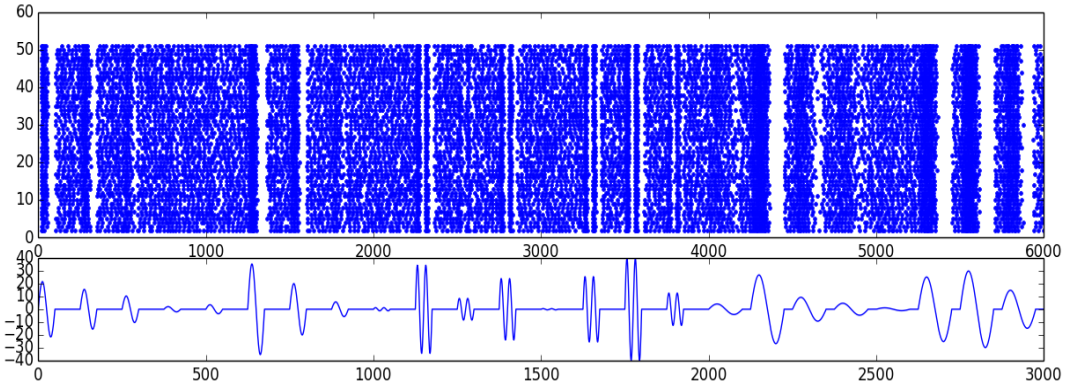

Fig. 1. Spike train of spiking neurons network (above) with its correspondent electrical stimulations (below).

Thus, assuming $\mathbf{w}_0 = 0$, we can calculate the output of the filter at time $t+1$ in terms of inner products of the input samples:

$$\mathbf{w}_{t+1} = \eta \sum_{i=0}^{t} e_i \mathbf{x}_i^T \mathbf{x}_{t+1} \tag{5}$$

To define the KLMS, first we remember the *kernel trick* $\langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle_{\mathcal{H}} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$, where $\mathcal{H}$ is a RKHS and κ its reproducing kernel. Using the same reasoning as above and the kernel trick, we can calculate the output of the KLMS filter [11] as

$$y_{t+1} = \eta \sum_{i=0}^{t} e_i \kappa(\mathbf{x}_i, \mathbf{x}_t) \tag{6}$$

and adapt that filter $\boldsymbol{\omega}_t = \sum_{i=0}^{t} e_i \kappa(\mathbf{x}_i, \cdot)$ following the gradient's contrary direction as

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t + \eta e_i \kappa(\mathbf{x}_{t+1}, \cdot) \tag{7}$$

In order to apply *dropout* to the KLMS algorithm, we need to find equivalent equations to (1) and (2) in RKHS. Thus, first we calculate the output of the filter at the training step $t+1$ as

$$\mathbf{y}_{train} = \mathbf{y}_t = \eta \sum_{i \in \mathcal{R}_t} \kappa(\mathbf{x}_i, \mathbf{x}_t), \tag{8}$$

where $\mathcal{R}_t$ is a random subset of $\{1,2,\dots,t-1\}$. Note that for each t, we sample a different $\mathcal{R}_t$, thus we are training different filters $\boldsymbol{\omega}$ at each training step. The average between all those filters can be approximated by

$$\mathbf{y}_{test} = \eta \sum_{i=1}^{N} p_i e_i \kappa(\mathbf{x}_i, \mathbf{x}_t), \tag{9}$$

where p_i is the relative frequency of the event $\{i \text{ is in } \mathcal{R}_t\}$. For the *1st case* where we fix the *dropout* probability, p, allowing $\mathcal{R}_t$ to grow with the number of training samples, we have $p\_i = p$ constant. *2nd case* is when we fix the size of $\mathcal{R}_t$ to L, thus growing the *dropout* probability, we end up with $p\_i = \sum_{j=1}^{N-i-1} L/j$. This last case is a particularity of the growing nature of adaptive kernel machines and resembles fixed budget algorithms during training, but keeping the whole architecture for test time. One can easily see that *1st case* just asks for smaller learning rates and *2nd case* needs an annealing of the learning rate. But, we also observed in practice that we can train the *dropout* KLMS without p_i and then learning a simple scale and bias for the final output y_t by linear regression with the desired training set d_t.

Also, we can analyze the statistical properties of the proposed algorithm as a regularization of the gradient as in [12]. Comparing the regular and the *dropout* gradient we have the gradient error:

$$\begin{aligned} \nabla \epsilon_i &= \left(e_i - e_i^{drop}\right) \eta \kappa(\mathbf{x}_i, \cdot) \\ &= \left(-\sum_{j=1}^{i} e_j \kappa(\mathbf{x}_j, \boldsymbol{x}_i) + \sum_{j \in \mathcal{R}_t} e_j^{drop} \kappa(\mathbf{x}_j, \mathbf{x}_i) \right) \eta \kappa(\mathbf{x}_i, \cdot), \end{aligned} \tag{10}$$

which is the error of the subgradient in RKHS, where. We leave the derivation of regret bound and further analysis of the proposed gradient for future work.

To validate the method above, we present simulations with spike train data in the next section. We choose this particular problem because we believe a practical and fast to train kernel machine should useful for several research problems in Neuroscience. Since calculating spike train kernels takes a relatively long time, this problem also allows us to show the speed up property of *dropout*.

III. Experiments

A spike train is a time series of stereotyped neural activity. For the present application we are interested only in the time that each spike occurred, thus we represent a spike train of neuron i as:

$$x_i = \sum_{\tau \in \mathrm{T}_i} \delta(t - \tau), \tag{11}$$

where T_i is a set with the time that neuron i fired. Equivalently, a spike train population, $\mathbf{x} = [x_1, x_2, \dots, x_m]$, is the set of time

where each one of the m neurons in the population fired a spike.

We assume that spike train is the effect of electrically stimulating the neural population with a source current d_t. In our simulations we generated a Brunel like spiking neurons network [13] using NEST simulator [14]. The network has 8000 excitatory, 2000 inhibitory neurons and a random Poisson input that stands for external source signals. We stimulated this network with sine pulses of 150 ms of duration and 250 ms between pulses as in [15]. The whole experiment has 6s of simulation time, the first 2s the sine pulses have frequency of 10Hz, the next 2s the stimulus frequency is of 20 Hz and 5 Hz in the last 2s. The whole experiment has a time resolution of 1 ms. Figure 1 shows an example of the stimulus signal the resulting spike activity.

This way, we can define a regression or decoding setting from the temporal spike times to the stimulus signal. In order to do that we used regressors defined as subsets of the population spike train at a time window:

$$\mathbf{x}_{\{t-\Delta t_1, t+\Delta t_2\}} = [\mathbf{x}_i]_{i=t-\Delta t_1}^{t+\Delta t_2} - t + \Delta t_1. \tag{12}$$

We subtracted $t - \Delta\, t$ in the equation above to take all regressors in the same relative temporal window *starting at time zero* as suggested in [15]. We used the first $2s$ of data as training set and the following $4s$ as test data. Note that this probes the performance of the KLMS algorithm to decode signals with half or twice the frequency it learned.

Also, here we used the memoryless Cross-Intensity (mCI) kernel [2] for spike trains, x_i and x_j, defined as

$$\kappa(x_i, x_j) = \sum_i \sum_j \exp\left(-\frac{|t_i - t_j|}{\tau}\right), \tag{13}$$

here x_i and x_j are two time windows of the same spike train. We calculate the spike train populations kernel between $\mathbf{x}_i$ and $\mathbf{x}_j$ as sum of (13) for each neuron in the population.

Other parameters of our experiment are the time step of $2\,ms$ between temporal windows, $\Delta t_1 = -50ms$, $\Delta t_2 = 20ms$, kernel size $\tau = 0.005$, learning rate $\eta = 0.0005$.

In Table I we compare the Mean Square Error (MSE) to the desired signal and simulation time between regular and *dropout* KLMS algorithms. Note that integer coefficients means that all the samples but an integer numbers are kept, thus growing the number of dropouts. Real coefficients mean the probability of a given sample to be dropped out. Coefficient 0 (zero) means that no elements were dropped out from active set while training. Also note that we only show the training time, since test times are equal for both methods. We repeated the experiment 10 times using the same stimulus and spike trains, but with different random dropouts for each sample at each repetition. We present mean ± standard deviation of our results.

Table I

Mean Square Error (MSE) during training and test phase as function of dropout coefficient.

Dropout coefficient	0	0.25	0.5	1	5	10
Train MSE	27.55	13.32 ±2.77	19.18 ±0.26	19.30 ±0.02	19.30 ±0.03	19.26 ±0.04
Test MSE	283	118.21 ±4.09	119.07 ±6e-10	119.07 ±4e-10	117.63 ±3	119.7 ±0.51
Train time* (s)	2018	249.74	498,22	2.56	11.95	21.15

*Simulations were performed on 2013 Core-i5 MacBook Air using custom Python code. That will be released with the paper.

We see that for this particular experiment we obtained better results for growing *dropout* probability with time, in other words, when we fix the number of members of the active set and drop everything else. We observe that this particular dataset is easy to overfitt and that the regularization provided by the *dropout* improved the results. But we note that no dropout KLMS provides results that can be useful on the go, without the extra need of a further regularization. In other words, the same output that is used to train KLMS can be readily used as an output of the filter since (6) and (8) are equal for this case. On the other hand and outputs of dropout KLMS may not resemble that desired since it is using only part of the kernel machine architecture to provide them.

IV. Conclusions

This paper proposes an equivalent of *dropout* for kernel adaptive filters. *Dropout* was reported to improve the generalization of deep feedforward neural networks. Here we also observe its ability to enhance kernel machines. The present implementation is such that it can be used as a further regularization of the gradient in RKHS. We also observed, in results not shown, that the proposed method can be used in parallel to limited budget methods. Future research should analyze kernel *dropout* performance to classification tasks in Perceptrons and SVMs.

Acknowledgment

E.S. would like to thank his brothers at Beta-Rho, mainly, Ciro Nascimento for useful discussions during the preparation of the manuscript.

References

[1] D. DeCaprio, J. P. Vinson, M. D. Pearson, P. Montgomery, M. Doherty, and J. E. Galagan, “Conrad: Gene prediction using conditional random fields,” *Genome Res.*, 2007.

[2] A. R. C. Paiva, I. Park, and J. C. Principe, “A reproducing kernel hilbert space framework for spike train signal processing,” *Neural Comput.*, vol. 21, no. 2. MIT Press, Cambridge, MA, USA, pp. 424–449, 2009.

[3] C. H. Lampert, “Kernel Methods in Computer Vision,” *Found. Trends. Comput. Graph. Vis.*, vol. 4, no. 3, pp. 193–285, Mar. 2009.

[4] A. Mitschele, S. Chalup, F. Schlottmann, and D. Seese, “Applications of Kernel Methods in Financial Risk Management.”

[5] O. Dekel, S. S. Shwartz, and Y. Singer, “The Forgetron: A kernel-based perceptron on a fixed budget,” in *Advances in Neural Information Processing Systems 18*, 2006, pp. 1342–1372.

[6] G. Cavallanti, N. Cesa-Bianchi, and C. Gentile, “Tracking the best hyperplane with a simple budget Perceptron,” *Mach. Learn.*, vol. 69, no. 2–3, pp. 143–167, Dec. 2007.

[7] F. Orabona, J. Keshet, and B. Caputo, “The projectron: a bounded kernel-based Perceptron,” *ICML ’08 Proc. 25th Int. Conf. Mach. Learn.*, Jul. 2008.

[8] B. Chen, S. Zhao, P. Zhu, and J. C. Principe, “Quantized Kernel Least Mean Square Algorithm,” *IEEE Trans. Neural Networks Learn. Syst.*, vol. 23, no. 1, pp. 22–32, Jan. 2012.

[9] S. Zhao, B. Chen, P. Zhu, and J. C. Príncipe, “Fixed budget quantized kernel least-mean-square algorithm,” *Signal Processing*, vol. 93, no. 9, pp. 2759–2770, Sep. 2013.

[10] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors.” Jul-2012.

[11] W. Liu, P. P. Pokharel, and J. C. Principe, “The Kernel Least-Mean-Square Algorithm,” *Signal Process. IEEE Trans.*, vol. 56, no. 2, pp. 543–554, 2008.

[12] I. Sutskever, “A simpler unified analysis of budget perceptrons,” in *Proceedings of the 26th Annual International Conference on Machine Learning*, 2009, pp. 985–992.

[13] N. Brunel, “Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons.,” *J. Comput. Neurosci.*, vol. 8, no. 3, pp. 183–208, 2000.

[14] M.-O. Gewaltig and M. Diesmann, “NEST (NEural Simulation Tool),” *Scholarpedia*, vol. 2, no. 4, p. 1430, 2007.

[15] L. Li, I. M. Park, S. Seth, J. S. Choi, J. T. Francis, J. C. Sanchez, and J. C. Principe, “An adaptive decoder from spike trains to micro-stimulation using kernel least-mean-squares (KLMS),” *Machine Learning for Signal Processing (MLSP), 2011 IEEE International Workshop on*. pp. 1–6, 2011.