PeerJ Preprints NOT PEER-REVIEWED

Beliefs Propagation in Log Domain: A Neural Inspired Algorithm for machine learning

Osama Ashfaq
Bahria University
Osama.Ashfaq.Bahria@gmail.com

Abstract

In this paper, we consider a variant of belief propagation algorithm in a tree graphical model where computations are carried out in the negative log-likelihood domain. Unlike the min-product algorithm, our goal is not limited to estimating the mode of the marginal distribution. We would like to obtain the entire marginal distribution as the sum-product algorithm does. We applied the algorithm to learn effective users features for A/B testing. We discussed scalable extension to the proposed algorithm for processing large amount of data.The primary goal of a parallel program is to reduce running time comparing to the sequential program by taking full advantage of computing power of multiprocessors. Threads are widely used in the implementation of parallelism in shared memory multiprocessor architectures. For UNIX/LINUX systems, *pthread* is the POSIX standard threading interface, which provides support a standardized way for creating and synchronizing threads. Here we presents how pthreads can be used successfully in parallelizing real scientific problems. We will illustrate it by implementing the shared memory parallel version of Jacobi iteration algorithm. Results of performance tests showed that the speedups can be up to p where p is the number of processors.

1 Introduction

Let $m_{j\to i}(x_i)$ be the message from random variable x_j to x_i, $\psi(\cdot)$ is the potential function over random variables. The sum-product [1, 2] algorithm can be written as

$$m_{j\to i}(x_i) = \sum_{x_j} \psi_{ij}(x_j, x_i) \prod_{k\in N(j)\setminus\{i\}} m_{k\to j}(x_j) \tag{1}$$

where $N(j) \setminus \{i\}$ be the set of neighboring nodes of x_j other than x_i. Let $\nu(x_j) = -\sum_{k\in N(i)\setminus\{i\}} \mu_{k\to j}(x_j)$ be the log belief, by taking the negative log of both sides of equation 1 we have

$$\begin{aligned} \mu_{j\to i}(x_i) &= -\log m_{j\to i}(x_i) \\ &= -\log(\sum_{x_j} \psi(x_j, x_i) \exp^{-\nu(x_j)}) \end{aligned} \tag{2}$$

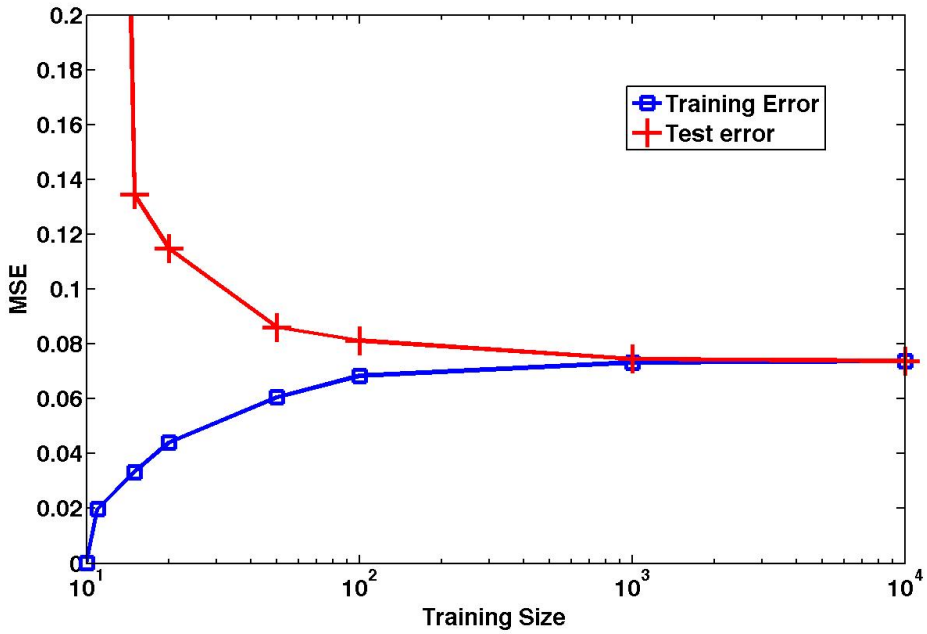
$$\begin{aligned} \nu(x_i) &= -\sum_{j\in N(i)\setminus\{i\}} \mu_{j\to i}(x_i) \\ &\simeq \sum_{j\in N(i)\setminus\{i\}} \sum_{x_j} w(x_i, x_j)\nu(x_j). \end{aligned} \tag{3}$$

Equation 2 involves log-sum-exp operations, which are extremely computational intensive. One possible way to get around it is to approximate the log message as a linear average of the log belief [3], with appropriate choice of weight coefficients $w(x_i, x_j)$.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2256v1 | CC BY 4.0 Open Access | rec: 6 Jul 2016, publ: 6 Jul 2016

PeerJ Preprints NOT PEER-REVIEWED

Figure 1: Approximation accuracy for training set and test set using batch learning rule



Equation 3 can be viewed as an input-output relation between neurons in a neural network, where the output of a neuron $\nu(x_i)$ is a weighted sum of its inputs $\nu(x_j)$. This suggests a way how brain implements belief propagation. To encode the belief of some random variable, we suppose brain employs a population of neurons, whose activities correspond to the log likelihoods of different states. There is strong physiological evidence [4, 5] to support this assumption and there are experimental finding suggesting that neurons transmit log beliefs in the network. Correspondingly, the network weights $w(x_i, x_j)$ between population i and population j encode potential function $\psi(x_i, x_j)$. The advantages for brain to work in log probability domain are the following. First, it only requires linear summation as in equation 3, which is a common way to describe the behaviors of neurons. It doesn't involve neural-implausible operations like point-wise multiplication in equation 1. Second, in negative log domain, less likely states correspond to higher activities. This enables animals to draw more attention toward rare events, but also provides better coding strategy in an energy efficient way, assuming the response range of a neuron is fixed. Third, Emo Todorov[6, 7] recently showed a general duality between estimation and optimal control. In his setting, the negative log of backward filtering density can be viewed as the optimal cost-to-go in control and reinforcement learning problems [8, 9]. Therefore, the study of how log probability propagates might help understand how brain optimizes its motor control [10, 11]. Such belief propagation algorithms has been successfully used in industry application such as face recognition and ontology discovery.

The weights $w(x_i, x_j)$ can be obtained from the potential function $\psi(x_i, x_j)$ by using the pseudo-inverse method. In this project, I would like to study the accuracy and generality of approximation in equation3. Moreover, when the potential function $\psi(x_i, x_j)$ is unknown, I would like to study how the neural network learn the connection weights w, by the plasticity of network. The network input, which comes via particle filters [12, 13], is consistent with distributed algorithms[14].

2 Pseudo-inverse Method

Therefore, we need to find a set of weights w that satisfy the approximation in equation 3 for arbitrary probability distribution $P(X_j) = \exp(-\nu(X_j))$. We tackled this problem by first generating a set of random distributions $P_l(X_j)$ for $l = 1 : L$. Each probability $P_l(X_j = k) = \exp(-\nu(X_j(l) = k)$ is uniformly drawn from $[0, 1]$, with constraints $\sum_{k=1}^{K} P_l(X_j = k) = 1$. K is the size of state

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2256v1 | CC BY 4.0 Open Access | rec: 6 Jul 2016, publ: 6 Jul 2016

PeerJ Preprints NOT PEER-REVIEWED

space of random variable X. For a fixed state k' in x_i, we then minimize the following objective functions

$$\begin{aligned} H(k') &= \sum_{l=1}^{L}[\sum_{k} w(k',k)\nu(X_j(l)=k) \\ &- \log\sum_{k}\psi(k',k)\exp(-\nu(X_j(l)=k))]^2 \end{aligned} \tag{4}$$

with respect to weights $w(k',\cdot)$. Let $\mathbf{y} = \{w(k',1), w(k',2), \ldots, w(k',K)\}^T$ be a $K \times 1$ weight vector, which can be obtained from any potential function $\psi(k',\cdot)$ by using the standard pseudo-inverse method. Let $A = \{\nu(X_j(l) = k)\}$ present a $L \times K$ matrix of log beliefs. Let $\mathbf{b}$ represent a $L \times 1$ column vector of log sums, that is, $b(l) = \{\log\sum_k \psi(k',k)\exp(-\nu(X_j(l) = k))\}$. To minimize the square error in equation 4 with respect to weight $\mathbf{y}$, we need to solve the equation $A\mathbf{y} = \mathbf{b}$, assuming columns in A are linear independent and $L \geq K$. Multiplying both sides by the psedo-inverse of A we have

$$\mathbf{y} = (A^T A)A^T\mathbf{b}$$

We first test this approximation as a function of size of training set L for a fixed potential function $\psi(k',\cdot)$. Approximation accuracy was measured in terms of the mean square error (MSE) between both sides of equation 3, $\epsilon = \sqrt{H(k')/L/K}$. We also generate a test set of probability distributions to measure the approximation accuracy using the learned weights $\mathbf{y}$. In the experiment, we choose the size of state space $K = 10$. We first learn the weight vector $\mathbf{y}$ from training sets with size $L = 10, 11, 12, 15, 20, 100, 1000, 10,000$. Then we use this learned weight vector to measure the approximation accuracy of a test set with size $10,000$. To reduce the variance in MSE [15], we average the square difference over 100 different trials, each of which has different random sets of training distributions and test distributions. As shown in Figure 1, the average error for the test size decreases as a function of size of training set. The only case when the approximation accuracy diverges (overfitting) is $L = K$. When $L > K$, which corresponds to an overdetermined system, the approximation error maintains below 0.15. The average error slowly converges to 0.08 when L is greater than 100. Since $\log p - \log q = \epsilon$ corresponds to $\frac{p-q}{q} = \exp(\epsilon) - 1$, this indicates that the percent error between the true belief and approximated belief is less than 15%, as along as the system is overdetermined. We also test the relation between approximation accuracy and the transition probability. It turns out the error curve shown in figure 1 is quite consistent for different choice of transition probability functions. Over the 100 different $\psi(k',\cdot)$ we tested, the standard deviation of average error is only 20% of the mean of the average error.

3 Sequential Learning

The above linear squares solution assumes the potential function ψ is known. Moreover, it requires the system stores a large amount of training data. Therefore, an online version of learning rule would be more favorable. In this section, we employed the perceptron learning algorithm to sequentially decrease the squared differences H in equation 4.

$$\mathbf{y}(l+1) = \mathbf{y}(l) + \eta * (b(l) - A(t,\cdot) * \mathbf{y}(l)) * A(l,\cdot)^T$$

where $\eta = 1/L$ is the learning rate. Unlike batch learning in Figure 1, left, both training error and test errors shown in Figure 3 decrease as the size of training set increases. Due to size limitation, more details about the sequential learning would be covered in the final report.

4 Method

Jacobi iteration algorithm can be used to solve a differential equation called Laplace's equation:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0 \tag{5}$$

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2256v1 | CC BY 4.0 Open Access | rec: 6 Jul 2016, publ: 6 Jul 2016

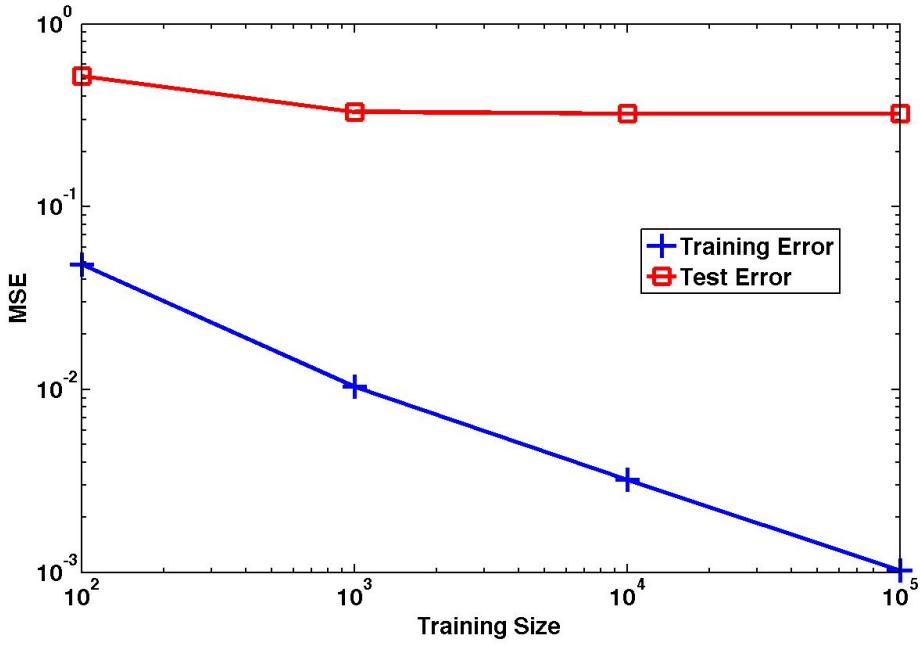


Figure 2: Approximation accuracy for training set and test set using sequential learning rule.

We need to solve this equation for each point (x, y) within a given two dimensional region. We can discrete this region into a large number of points and apply the finite different method to each point for solving the differential equation:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = \frac{1}{\delta^2}[f(x+\delta, y) + f(x-\delta, y) + f(x, y+\delta) + f(x, y-\delta) - 4f(x, y)] = 0$$

By using Jacoi iterative formula:

$$f^{k+1}(x, y) = \frac{1}{4}[f^k(x+\delta, y) + f^k(x-\delta, y) + f^k(x, y+\delta) + f^k(x, y-\delta)] \tag{6}$$

We can expect that after a number of iterations, the difference between$f^{k+1}(x, y)$ and $f^k(x, y)$ would be approaching zero. In those data this result is attributed to the so called Markov Decision Process [16, 17].

Thus the numerical computing method for solving the equation can naturally employ large $N \times N$ 2D array as its data structure. It provides an opportunity for data parallelism. We can have multiple processors to process different parts of the arrays [18]. The main ideas of Jacobi algorithm are outlined as follows:

Initiation At first we create two two-dimensional array A and B. We always write data on the elements of A and read data from elements of B. Thus we have to establish some initial values on B.

Iteration We recompute the value of each element of A as the average of values of its four neighboring elements in previous iteration, which are stored in B. The values of all elements will eventually converge to an accurate solution [19]. That means all the values will not be changed during iterations any more.

Check At the end of each iteration, if all differences between corresponding elements in A and B approach zero, the algorithm terminates. A global variable *maxDiff* is used to record the

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2256v1 | CC BY 4.0 Open Access | rec: 6 Jul 2016, publ: 6 Jul 2016

PeerJ Preprints NOT PEER-REVIEWED

maximum difference of all the elements in one iteration. Then we compared it to some desired extremely tolerant value *TOL*. If *maxDiff* is smaller than *TOL*, the algorithm stops. Otherwise,*maxDiff* will be reset to zero and A and B will be swapped. Therefore, A will always ready for writing in the next iteration and B keeps all values computed in current iteration.

In this case, the calculation in given iteration needs the values computed in the previous one, it is impossible to parallelize the iteration loop. However, we can implement the data parallelism by a row distribution. Since the computation of each row have to access the values of neighboring rows. To minimize the cache miss rate and the potential problems of cache coherent, we employ a BLOCK distribution: each processor will own and compute equal $\frac{N}{p}$ successive rows. In this way, we can balance the load.

5 Considered Issues and Parallel Implementation

The swap of values of matrices A and B may result in large number of read miss in the next iteration. We can improve both sequential and paralllel algorithm by recomputing the values of elements at B as the average of values of its four neighboring elements at A after all the values of A have been calculated. That's to say, before checking the minimum difference, there're actually two iterations carried out. Of course a barrier should always be inserted after each iteration so that we can ensure the writes of one matrix takes place after all the writes of the other matrix. Although swapping the pointers of matrices A and B will result in the same performance, it can not be implemented in some scientific languages, i.e. FORTRAN77.

In the check state, since each processor has to write *maxDiff*, we will encounter the problem of data coherence. Here we create a one dimensional array with length p. Each element in this array stores values of *maxDiff* computed by each processor. All of them then compare to *TOL*. By using this method, we can ensure the coherence of data and avoid the time consuming locks. We should also insert barrier before and after the check state so that we can prevent one processor from illegal stopping. All the processors should move on to the next iteration even only one process fails to reach the tolerant value, similar to the results found by boosting algorithm.

In summary, the parallel algorithm for Jacobi iteration can be implemented through the following procedures:

1. Specify the starting and ending rows for each processor
2. For each elements except for the boundaries: $A[i][j] = \sum_{neighbor} Bs$
3. Barrier
4. For each elements except for the boundaries: $B[i][j] = \sum_{neighbor} As$
5. Barrier
6. Calculate the *maxDiff* for each processor and compare all them with *TOL*. If all of them are smaller, program finished
7. Barrier
8. go back to step 2.

6 Performance

6.1 Testing Enviroments

We have carried out performance tests on IBM p655 cluster (pcluster) which is comprised of 32 8-CPU 1.18GHz Power4 nodes, each node with 16GB of RAM. Each processor has 32KB L1 data cache size and 1536KB L2 cache. The operating system is AIX. It uses the LoadLeveler queueing system.

In the Jacobi problem, we have used the following input data:

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2256v1 | CC BY 4.0 Open Access | rec: 6 Jul 2016, publ: 6 Jul 2016

- Boundary Condition. $North = 2, South = 3, East = 4, West = 5$
- Initial body values(all other elements other than boundaries): 0
- tolerant value: 1.0E-6
- Body dimensions: 128×128, 256×256, 512×512 and 1024×1024
- The threads - processors mapping is 1:1

6.2 Performance Measurements

Table 1. 2 and 3 and Fig. 2 illustrate the results obtained from performance tests with different size of array. The speedup is calculated according to the sequential algorithm.

As we can see from Fig.2, at a given number of processor, the speedup increases with the size of array. That may result from larger computation/communication ratio. And at the same time, when N is large, the speedup is larger than p. That may be because the L2 cache cannot hold the whole two dimensional array of double data at the same time. That will increase the read miss rate a lot.

Table 1: Speedup Results

N	# of Processor	Running Time	Speedups
128	1	25	1.00
128	2	14	1.78
128	4	7	3.57
128	8	6	4.17
256	1	385	1.00
256	2	206	1.87
256	4	107	3.59
256	8	56	6.9
512	1	4805	1.00
512	2	2555	1.88
512	4	1156	4.15
512	8	621	7.73
1024	1	57024	1.00
1024	2	26625	2.15
1024	4	13521	4.23
1024	8	7001	8.17

7 Conclusion

In this study, we show how we can use parallel program techniques to increase the performance and decrease the running time. We also realize the importance of cache miss rate in high performance computing, especially in the case of shared memory multiprocessor architecture [20].

References

[1] M. Baes and M. Burgisser. Hedge algorithm and dual averaging schemes. *arXiv*, 1112(1275), 2011.

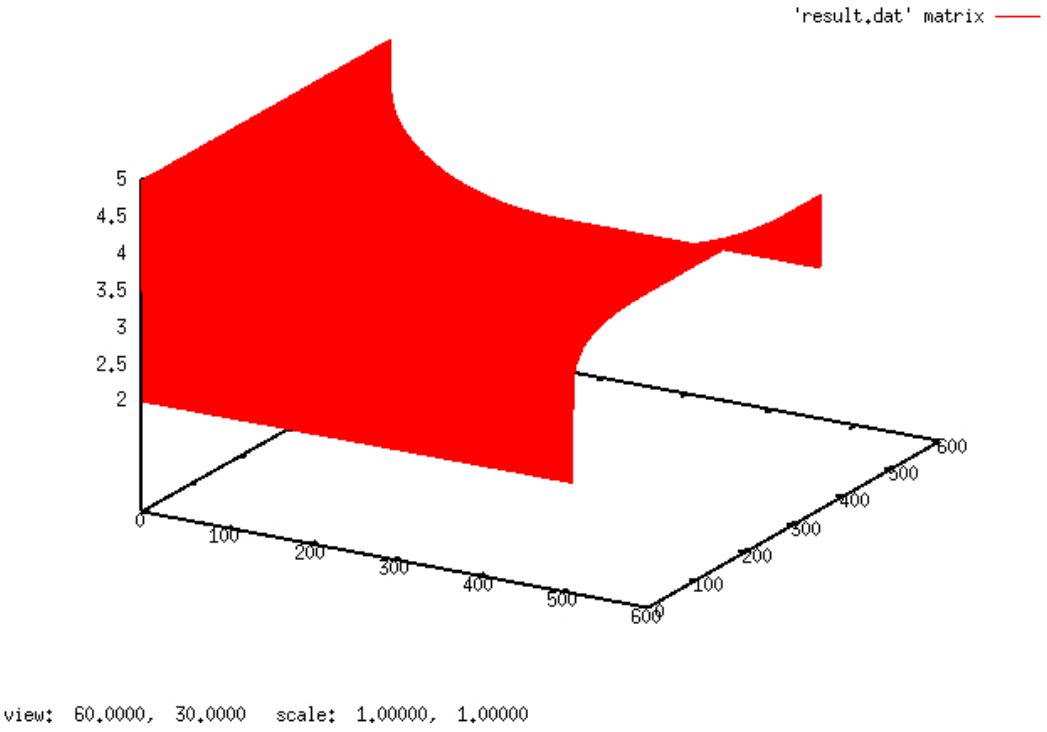
[2] Y. Freund and R. E. Schapire. A decision-theorectic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[3] Y. Li, L.G. Linda, and J. Bilmes. A generative/discriminative learning algorithm for image classification. *ICCV*, 2005.

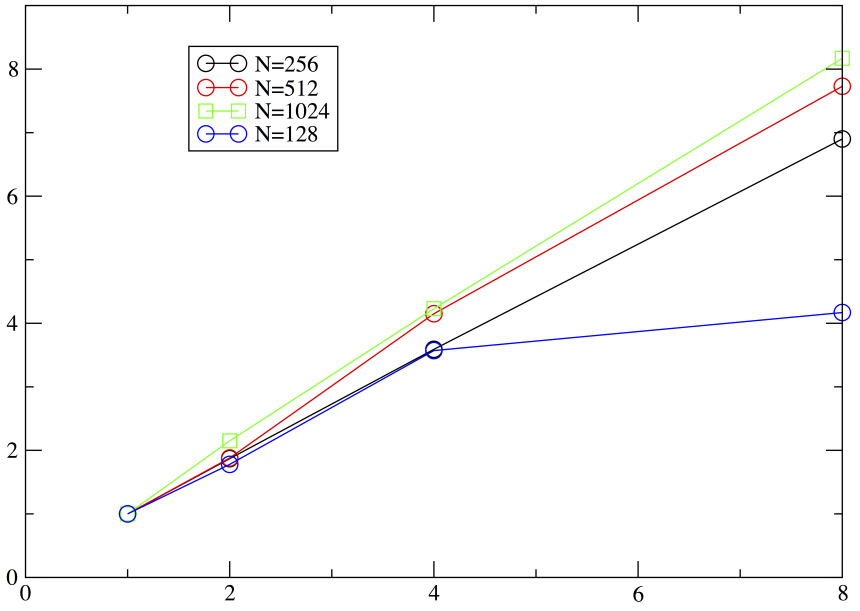
[4] R.P.N. Rao. Bayesian computation in recurrent neural circuits. *Neural Comput*, 16(1):1–38, 2004.

[5] K. Doya, S. Ishii, A. Pouget, and R. P. N. Rao. *Bayesian Brain: Probabilistic Approaches to Neural Coding*. Cambridge, MA: MIT Press, 2007.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2256v1 | CC BY 4.0 Open Access | rec: 6 Jul 2016, publ: 6 Jul 2016



Speedup with different size of arrays



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2256v1 | CC BY 4.0 Open Access | rec: 6 Jul 2016, publ: 6 Jul 2016

[6] E. Todorov. General duality between optimal control and estimation. *In proceedings of the 47th IEEE Conference on Decision and Control*, pages 4286–4292, 2008.

[7] J. Ditterich. Stochastic models and decisions about motion direction: Behavior and physiology. *Neural Networks*, 19:981–1012, 2006.

[8] Craig Boutilier. A pomdp formulation of preference elicitation problems. In *AAAI/IAAI*, pages 239–246, 2002.

[9] Yoav Freund, Robert Schapire, and N Abe. A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612, 1999.

[10] M. J. Kearns and L. G. Valiant. Learning boolean formulae or finte automata is as hard as factoring. Technical report, Department of Computer Science, Harvard University, 1988.

[11] L. Xiao. Dual averaging methods for regularized stochastic learning and online optimization. *The Journal of machine learning research*, 11, 2010.

[12] N.D. Daw and A.C. Courville. The pigeon as particle lter. *Advances in Neural Information Processing Systems*, 19, 2007.

[13] P. Dayan and N. D. Daw. Decision theory, reinforcement learning, and the brain. *Cognitive, Affective and Behavioral Neuroscience*, 8:429–453, 2008.

[14] Yanping Huang and Rajesh P Rao. Neurons as monte carlo samplers: Bayesian inference and learning in spiking networks. In *Advances in Neural Information Processing Systems 27*, pages 1943–1951. 2014.

[15] Tom Griffiths. Neural implementations of importance sampling. *NIPS preprint*, 2008.

[16] Y. Huang, A. L. Friesen, T. D. Hanks, M. N. Shadlen, and R. P. N. Rao. How prior probability influences decision making: A unifying probabilistic model. *Advances in Neural Information Processing Systems (NIPS)*, 2012.

[17] LR Bahl, Peter F Brown, Peter V De Souza, and Robert L Mercer. Maximum mutual information estimation of hidden markov model parameters for speech recognition. In *proc. icassp*, volume 86, pages 49–52, 1986.

[18] Bernhard Schölkopf and Alex Smola. Support vector machines. *Encyclopedia of Biostatistics*, 1998.

[19] L. Mason, J. Baxter, P. Barlett, and M. Frean. Functional gradient techniques for combining hypotheses. In *Advances in large margin classifiers*. MIT Press, Cambridge, 1999.

[20] P. Kara, P. Reinagel, and R.C. Reid. Low response variability in simultaneously recorded retinal, thalamic, and cortical neurons. *Neuron*, 27(3):635–646, 2000.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2256v1 | CC BY 4.0 Open Access | rec: 6 Jul 2016, publ: 6 Jul 2016