Branch Prediction with Neural Networks: Hidden layers and Recurrent Connections

Andrew Smith
Department of Computer Science
University of California, San Diego
La Jolla, CA 92307
atsmith@cs.ucsd.edu

Abstract

As Moore's law forces modern computer microarchitectures to increasingly rely on speculation, accurately predicting branches becomes more important in keeping the pipeline full. Previous work has shown that the perceptron, a simple linear discriminator, can be used as a powerful branch predictor. This paper expands on that research by experimenting with three other branch predictors, a neural network with one hidden layer (a feed-forward network), a neural network with one hidden layer and recurrent (feedback) connections (aka an Elman network), and a combined predictor, using a 2-bit saturating counter to vote between a perceptron and a feed-forward network. We show how the usually real-valued networks can be approximated with integer math using a look-up table to approximate the activation function and its derivative.

1 Introduction

Modern superscalar architectures achieve high instruction-level parallelism by speculation. For example, speculating on the direction of a conditional branch will allow the pipeline to remain full and executing useful instructions, if the speculation was correct. However, if the speculation was incorrect, those instructions that entered the pipeline after the misspeculated branch must be flushed out, and time is wasted while the pipline fills with the correct instructions. Because pipeline depths are continually increasing, this issue becomes more and more relevant to high-performance microarchitectures. [1]

The field of machine learning is rich with pattern recognition constructs that might be useful in this area. Traditional branch predictors, such as the 2-bit saturating counter (bimode) have shown some success. Research to improve the accuracy of bimode and similar simple predictors has concentrated chiefly on methods for eliminating destructive aliasing, or the harmful interferance of two branches which should not share the same predictor, but do, due to limited hardware. An alternative approach is to use more sophisticated learning techniques to create more accurate branch predictors.

Perceptrons are simple predictors which were shown in [1] to improve prediction accuracy over previous prediction schemes, given the same amount of hardware. The perceptron makes linear decision; the region of its input space for which it will decide to take a branch

is separated from the region in which it will decide to fall through by a simple hyperplane. Perceptrons, and their use in branch prediction is described in section 2.

Feed-forward neural networks, constructed out of several perceptrons have more power, in that the functions they can learn are not restricted to linear functions. Typically, several perceptrons (comprising the "hidden-layer") each receive a copy of the input, and one more perceptron takes the output of the hidden-layer perceptrons as its input, and makes a decision. This one-layer network is trained with a gradient-descent procedure called backpropagation. Feed-forward networks and their use in branch prediction is described in section 3.

Both perceptrons, and feed-forward neural networks are functions, in the sense that they produce the same output given the same input. This is a result of their structure; in the case of the perceptron, the function is a (function of the) weighted sum of the inputs, and the feed-forward network is a (function of the) weighted sum of the outputs of several perceptrons (see figures 1 and 2). If feedback connections are added to the network, the network now has state (and is no longer technically a function); the output now depends on the current input and all previous inputs. Such a network has the potential to learn correlations in a sequence of inputs over long spans of time, since values can be computed by the network and then stored indefinitely. This is a more natural computational model of time-series data such as a series of branch directions. This paper shows how a simple recurrent neural network (RNN) called the Elman network, can be implemented in integer math and applied to branch prediction in section 4.

2 The Perceptron

A perceptron is the simplest form of a neural network, with just one "artificial neuron." Like all neural networks, it learns from a training set of example inputs and outputs to approximate a function. The perceptron maps a set of input values $x_1, x_2, ..., x_d$ to an output value $y = -1$ or $+1$, by applying the following forumula:

$$y^* = w_0 + \sum_{i=1}^{d} x_i w_i$$

$$y = \text{sign}(\, y^* \,)$$

Where the w_i are the weights of the perceptron and w_0 is the bias. Alternatively w_0 can be considered as a weight connected to a unit that always has value 1. Figure 1 is a graphical example of a perceptron; each input is multiplied by the corresponding weight. The oval represents the artificial neuron, which sums its weighted inputs (as indicated by the $\sum$) and outputs the sign of the sum (as indicated by the "step-function" in the upper half of the unit).

The weights of the perceptron have a direct interpretation; a negative weight indicates a negative correlation between that unit and the output, an a positive weight is interpreted analogously. A negative bias weight indicates that the output is usually -1, and a positive weight indicates the output is usually $+1$.

2.1 Perceptron Learning

The classical perceptron learning algorithm descends the gradient of the error with respect to the weights. First choses a learning rate, α, which dictates the rate at which the weights are adjusted. α corresponds to how far along the gradient the weights are adjusted each step; if α is too big, the optimum set of weights could be missed, if α is too small, the weights will improve the performance of the perceptron very slowly.[bishop]

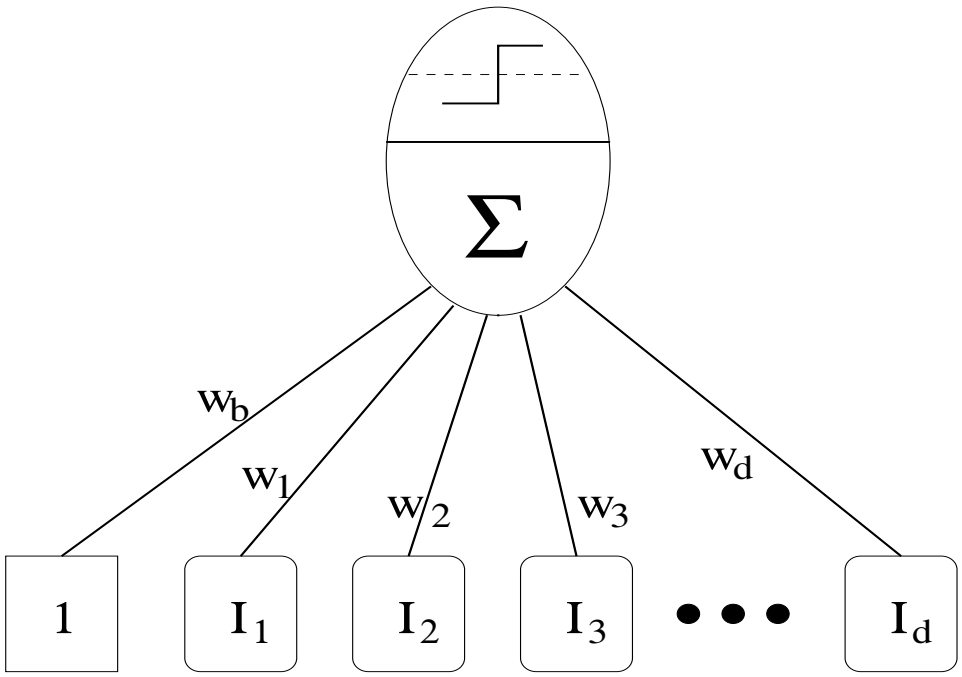

Figure 1: The Perceptron.

The learning algorithm cycles through each pattern in the training set and updates the perceptron if a patter is misclassified. A misclassification is when the target value t differs from the output of the perceptron y. When this occurs, weight w_i is adjusted by:

$$w_i \leftarrow w_i + \alpha t x_i$$

were x_i is the i^{th} input. Intuitively, if the target t and input x_i are the same, a small value is added to the corresponding weight, and if they are opposite, a small value is subtracted.

To be efficiently implemented in hardware, all calculations must be done in integer math. [1] found that using integer weights, and a learning rate of $\alpha = 1$ to be sufficient. Also, a threshold θ was introduced, so that if the perceptron was correct, but not confidently so (i.e. $|y^*| < \theta$), the perceptron is still trained.

2.2 Linear Separability

As stated before, perceptrons can only learn linearly separable functions. This follows from the fact that setting the equation for y^* to zero is the equation of a hyperplane in d-dimensional space which separates the values of $x_1..x_d$ for which $y^* > 0$ are separated from the values for which $y^* < 0$. The 2-layer feed-forward network overcomes this limitation by using several such hyperplanes.

3 The Feed-Forward Network

The 2-layer feed-forward network is a generalization of the perceptron. Figure 2 shows how several perceptrons can be organized into a 2-layer network (so-named because one layer operates on the input and the second layer operates on the first layer). The first-layer perceptrons, labeled $1..n$, comprise the "hidden layer," since when learning the weights,

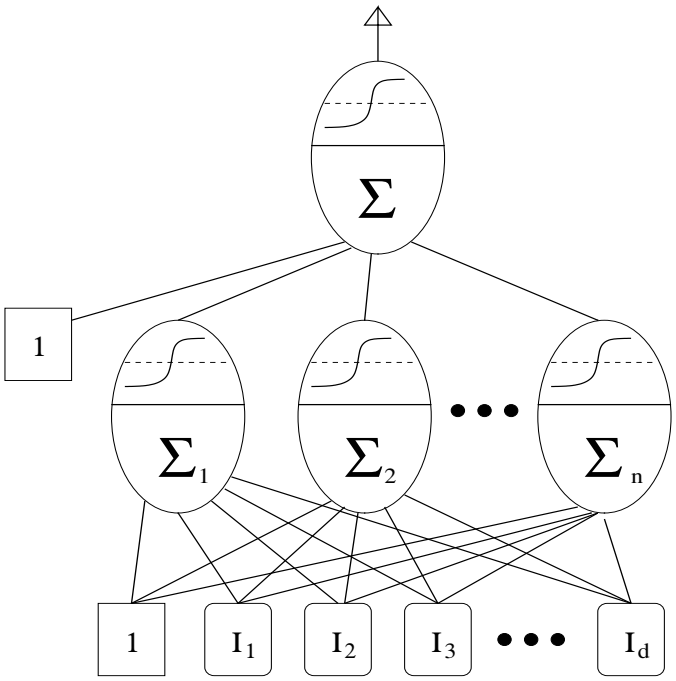

Figure 2: The Feed-Forward Network.

we have values for the output perceptron and the inputs, but not for these middel perceptrons. Another important difference between this network and the single perceptron is the activation function, which maps y^* to y. The single perceptron used the sign() function (as indicated by the step-function in the upper half of the oval), whereas more complex neural networks typically use a sigmoid (s-shaped) function such as tanh() (as indicated by the curve in the upper half of the perceptrons in figure 2).

Let w_{ij} represent the weight of the connection from input j to hidden unit i. Let the activation function be $f()$, and w_i be the weight from hidden unit i to the output unit. Then the function this network computes is:

$$h_i = f\left(w_{i0} + \sum_{j=1}^{d} w_{ij} x_j\right)$$

$$y = f\left(w_0 + \sum_{j=1}^{n} w_j h_j\right)$$

3.1 Feed-forward network learning

As with perceptrons, we learn the weights in a feed-forward network with gradient descent. The error function is the sum of the differences of the target for each input pattern t_i minus the network's output for that pattern y_i, squared:

$$E = \frac{1}{2} \sum_{i=1}^{N} (t_i - y_i)^2$$

The gradient of this error function can be calculated after passing throug the entire training set and then the weights can be adjusted. However, for branch prediction, we would like to update the weights after each prediction (and subsequent calculation of actual branch direction, t). Thus, for each pattern, the error is simply $(t-y)^2/2$. The derivatives of this with respect to w_i is

$$\frac{\partial E}{\partial w_i} = (t-y) f' \left(w_0 + \sum_{j=0}^{n} w_j h_j \right) w_i \tag{1}$$

If we define

$$\delta = \frac{\partial E}{\partial y}$$

or, the derivitave of the error with respect to the output unit, the partial derivative in equation 1 reduces to $\delta \, \partial y / \partial w_i$. We can define analogous derivatives for the hidden units:

$$\delta_j = \frac{\partial E}{\partial h_j} = w_j \delta$$

and then by the chain-rule, the derivative w.r.t. weight w_{ji} becomes:

$$\frac{\partial E}{\partial w_{ji}} f' \left(w_{j0} + \sum_{k=0}^{d} w_{jk} x_k \right) x_i$$

The error of the output unit was "back-propagated" to create an error for the hidden unit.[5]

Once the gradient of the error with respect to the weights is calculated, the weights can be updated by:

$$w \leftarrow w - \alpha \frac{\partial E}{\partial w}$$

Which moves the weights against the gradient, thus reducing the error. Section 5 discusses a practical implementation of this network in hardware

4 The Elman Network

Suppose we want our network not to merely be a function of its inputs, but compute some value of its inputs, remember it until it is needed, and then use it. Such a network has state, and can be implemented by providing feed-back connections.

Figure 3 is a simple example of such a network, called the Elman network. In the Elman network, after the hidden units are calculated, their values are used to compute the output of the network (as in the feed-forward network), and are also stored as "extra inputs" to be used the next time the network is operated. To formally define the semantics of our network, we must index the values of the hidden units with time, so $h_i^{(T)}$ represents the value of hidden unit i at time T. Let r_{ij} represent the recurrent weights from $h_j^{(T-1)}$ to $h_i^{(T)}$. Now, the network behavior at time T is defined:

$$h_i^{(T)} = f \left(w_{i0} + \sum_{j=1}^{d} w_{ij} x_j + \sum_{j=1}^{n} r_{ij} h_j^{(T-1)} \right)$$

$$y_i^{(T)} = f \left(w_0 + \sum_{j=1}^{n} w_j h_j^{(T)} \right)$$

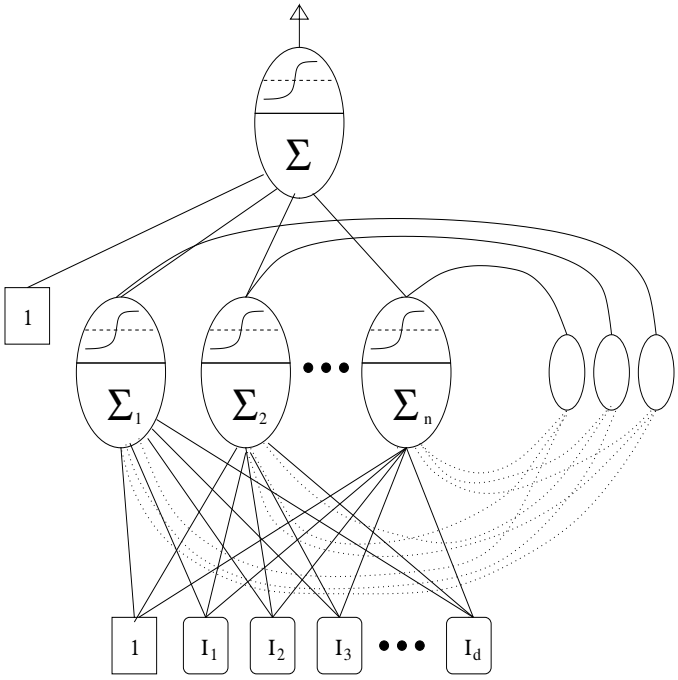

Figure 3: The Elman recurrent (feedback) newtwork. The small units to the right hold the values computed by the hidden units, to be used in the next operation of the network. The dotted connections indicate the recurrent connections; they provide a weighted sum of the previous values of the hidden units as input to the hidden units.

Note that this is identical to the feed-forward network except for the hidden units taking inputs consisting of weighted sums of the previous values of the hidden units.

Training such a network is not straightforward, since now the output of the network (and therefore, also the gradient) depends on the inputs, *and*, all previous inputs to the network. One approach used in the machine learning literature is to "unroll" the network through time, as in figure 4, and then use conventional backpropagation, called backpropagation through time (BPTT). This had the disadvantage, however, that each time a new pattern is presented to the network, it must be unrolled to the beginning, or unrolled h steps, called BPTT(h), in which case the network cannot learn to remember what happened more than h steps ago. [6]

A different approach, due to Williams and Zipser, is to maintain the derivatives of the value of each unit in the network, with respect to each weight in the network. [6] Then at each time-step, the derivatives of the error with resepect to weight w is the error at time T, $e^{(T)}$ times $\partial y^{(T)}/\partial w$, where $y^{(T)}$ is the output unit at time T. After the weights are adjusted, the derivatives of each unit with respect to each weight can be approximated by a simple recurrence relation. The derivative of unit u_k with respect ti weight w_{ij} at time $T+1$ can be approximated by:

$$\frac{\partial u_k^{(T+1)}}{\partial w_{ij}} = f'(s_k^{(T)}) \left[\sum_{v \in U} w_{kv} \frac{\partial u_v^{(T)}}{\partial w_{ij}} + \delta_{ik} z_j^{(T)} \right]$$

where δ_{ik} is the Kronecker delta, $z_j^{(T)}$ is the value of unit or input j at time T, U is the set of all units, and s_k is the weighted sum of inputs to unit k (i.e. $s_k^{(T)} = f^{-1}(u_k^{(T)})$).

With the simple Elman network, the full matrix of partial derivatives does not need to be calculated. The derivatives of the hidden units with respect to the output weights are zero, since there is no feedback from the output unit. For this network, we only need to store the derivatives of the output unit with respect to all the weights, and the derivatives of the hidden units with respect to the input weights and the recurrent weights.

This algorithm, called Real Time Recurrent Learning (RTRL), is intended to be used as the network is operating (hence "Real Time"), as opposed to periodically stopping it to unroll it and use BPTT. [6]

5 Efficient Implementations

Since the proposed branch prediction structures are intended to be implemented on a microchip, it is necessary that they operate efficiently. This somewhat precludes the use of real-valued (floating-point) numbers, which is the operating assumption of backpropagation and RTRL.

We propose an implementation of these algorithms that uses only integer math by interpreting each integer i as i/N, where N is some small power of 2 (we found 2048 to be adequate). Since the working range of numbers is only scaled (and not shifted), the sum $i + j$ represents the quantity $i/N + j/N = (i + j)/N$, which preserves the mapping. Whenever an algorithm calls for multiplying two numbers, the resulting product must be divided by N, since $(i/N)(j/N) = (ij)/N^2$. This is efficient since N is a power of 2.

The functions $f()$ and $f'()$ are stored in look-up tables of size $8N$, so each table represents the (scaled and truncated) values of the function acros the range $[-4/N\ ,\ 4/N]$. When the evaluation of a function at real numer r is needed, the table entry at position $Nr + 4N$ is indexed, or if that position is too low, $-N$ (corresponding to the asymptotic minimum of $f()$) is returned, or if that position is to high, N is returned (corresponding to the asymptotic maximum of $f()$).

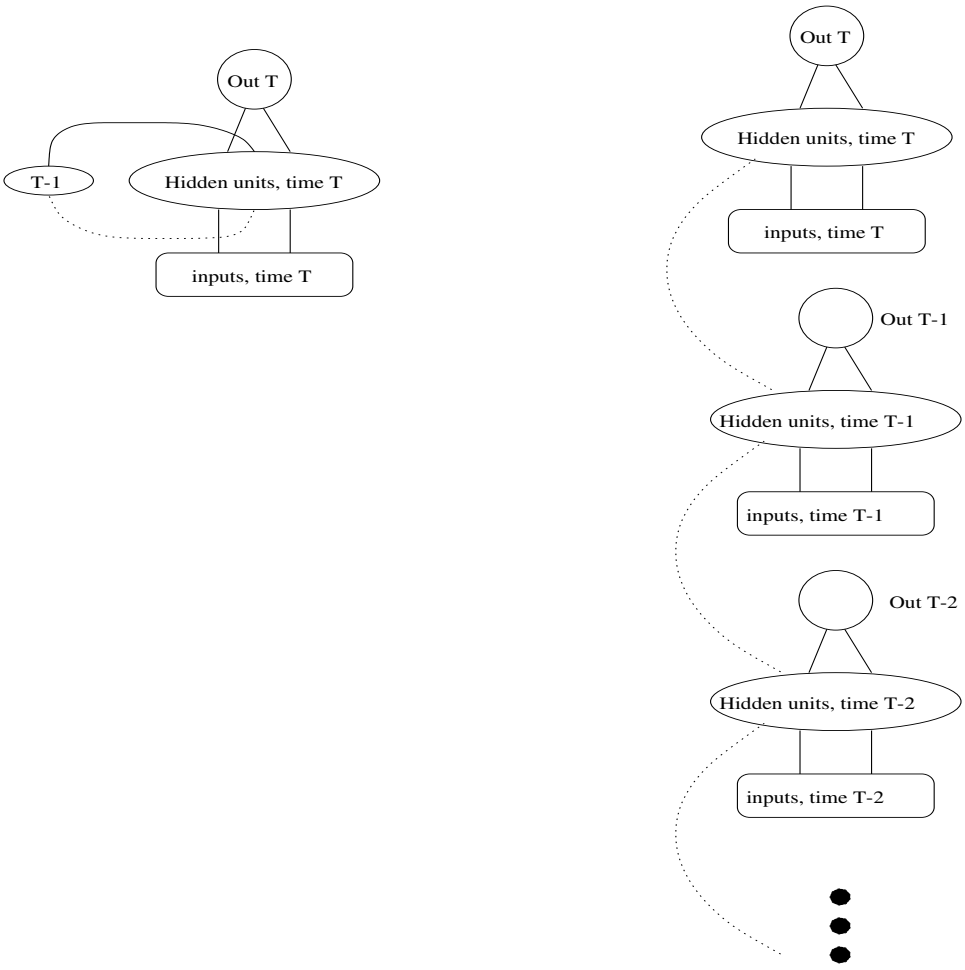

Figure 4: Unrolling an Elman network through time. On the left is the recurrent Elman network, and on the right is its interpretation as a long feedforward network. Under this interpretation, backpropagation can be used to calculate the derivatives of the errors (at each output unit) by unrolling the network to the beginning, or the derivatives can be approximated by unrolling it only a finite number of steps.

Since the minum value of $f'()$ is zero, but when working with real values $f'()$ only becomes very close to zero, we set the minimum value of $f'()$ to some small positive value. In this paper, the minimum value was $10/N$.

[In this paper, the function $f() = tanh()$, and therefore $f'() = tanh'() = 1 - tanh^2()$]

Given this mapping between floating-point numbers and integer representations, efficient, integer-math versions of backpropagation and RTRL can be implemented.

6 Results

The branch prediction schemes tested in this paper are all of similar format. A branch address is hashed into the table of predictors, and the global history is used as input to the selected predictor.

This paper uses SimPoint, as described in [3] to avoid having to simulate an entire benchmark. We used the Early-Single SimPoints of lenth 100 million instructions, given at [4]. Simulation was "fast-forwarded" until the relevant segment of execution. No warm-up was necessary, since the detailed simulation was so long.

We test our different branch predictors on 5 of the SPEC2000 benchmark programs. These programs were selected because their sim-point was relatively early on in the programs execution, and because the expected error in IPC from using that sim-point (as opposed to executing the entire program) was less than one percent. [4] We used CRAFTY-REF, GZIP-GRAPHIC-REF, BZIP-PROGRAM-REF, PARSER-REF, and TWOLF-REF.

To simulate the different neural branch predictors, the SimpleScalar simulator was modified to implement the following branch predictors:

- Perceptron branch predictor. Arguments: number, input size (ghist length).
- Two-Layer feed-forward neural networks. Arguments: number, input size, number of hidden units, extra delay incurred by predictor.
- Elman recurrent network, trained with RTRL. Arguments: number, input size, number of hidden units. (extra delay was not implemented, because these predictors always performed poorly and experimentation on them was stopped early.
- Combined Perceptron/Neural Net predictor, using 2-bit saturating counter to vote on which predictor's output is used. Arguments: number, input size (ghist length), number of hidden units (for Neural network), extra delay incurred if the neural network was used.

6.1 Architectural baseline

For this paper, the SimpleScalar defaults were used. SimplScalar simulates a superscalar out-of-order architecture with 4 integer ALUs, 1 integer mult/div unit, 4 FP ALUs, and 1 FP mult/div unit. The pipline can issuem, decode, and commit 4 instructions per cycle. There is a 3-level memory hierarchy. The L1 cache is 16k, and hits in 1 cycle. The L2 cache is 256k and hits in 6 cycles. Memory acces time is 18 cycles. The branch target buffer (BTB) has 2k entries and is 4-way set associative.

6.2 Different Hardware sizes

For this series of tests, we test the different types of branch predictors given different hardware budgets: 2k, 8k, 32k, 128k, and 512k. The input sizes for each predictor are set to what was determined to be optimal in [1], to 22, 34, 59, 62, and 62, respectively. We

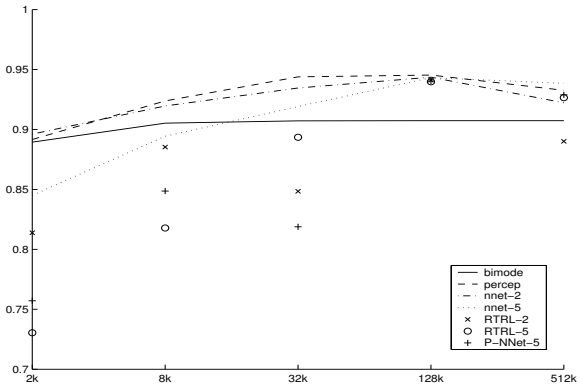

Figure 5: CRAFTY-REF: Prediction accuracy for different hardware budgets (x-axis).

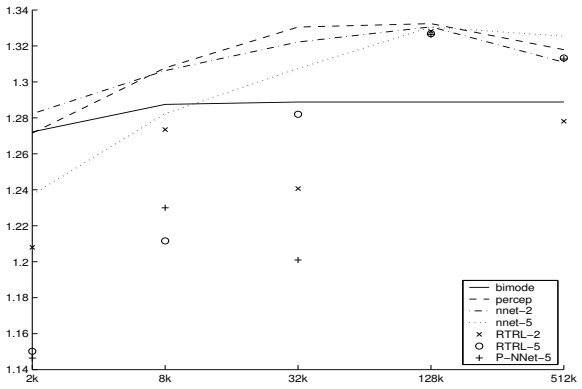

Figure 6: CRAFTY-REF: IPC for different hardware budgets (x-axis).

did not impose any penalty for predictors, since this test was a test of prediction accuracy. However, we report the resulting IPC as well: CRAFTY (figs. 5 and 6) GZIP (figs. 7 and 8) BZIP (figs. 9 and 10) PARSER(figs 11 and 12) and TWOLF (figs 13 and 14)

6.3 Different latencies

For this series of tests, the hardware budget is fixed at 64k, and we test the performance of a bimod predictor, perceptron with no extra delay, neural network with 3 hidden units and 1 extra delay cycle, and a neural network with 10 hidden units and 3 extra delay cycles. Figures 15 and 16 show the prediction accuracy of these networks, and the resulting IPC, respectively.

6.4 Different hidden-layer sizes

For this series of tests, the hardware budget is also fixed at 64k, and we test the prediction accuracy of a perceptron predictor, and neural network predictors with 1, 3, 8, and 15 hidden units, respectively. The point of this test was to explore the tradeoff between the number of Neural Networks, and the number of neurons in each network. Figures 17 and 18 show the prediction accuracy of these networks, and the resulting IPC, respectively.

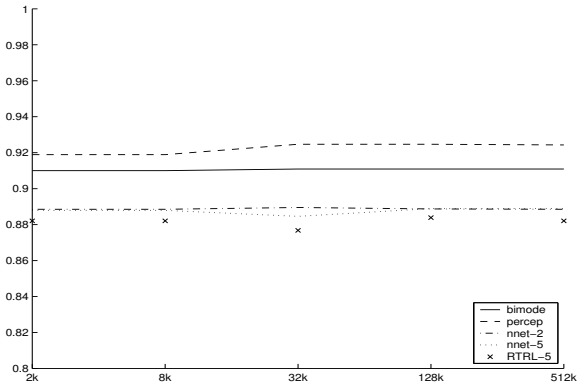

Figure 7: GZIP-GRAPHIC-REF: Prediction accuracy for different hardware budgets (x-axis).

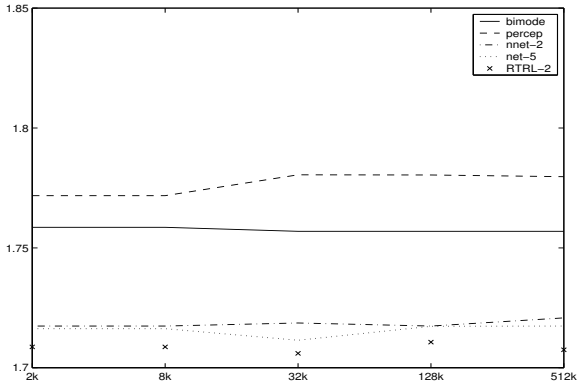

Figure 8: GZIP-GRAPHIC-REF: IPC for different hardware budgets (x-axis).

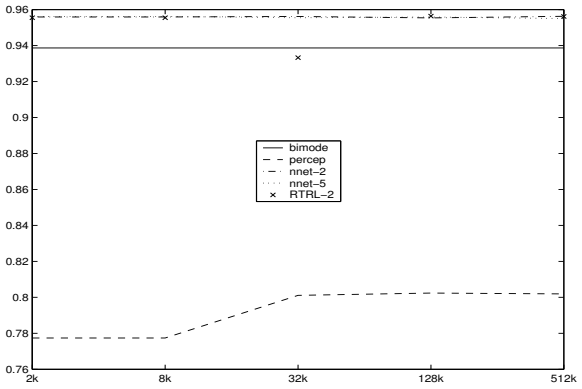

Figure 9: BZIP-PROGRAM-REF: Prediction accuracy for different hardware budgets (x-axis).

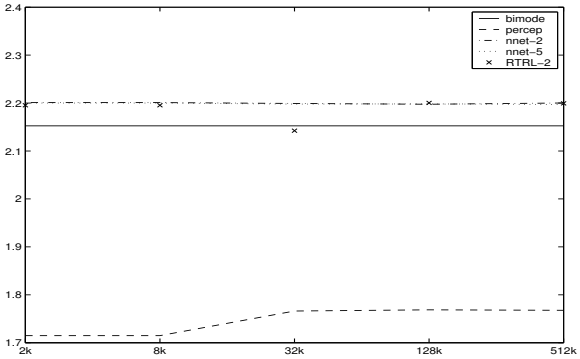

Figure 10: BZIP-PROGRAM-REF: IPC for different hardware budgets (x-axis).

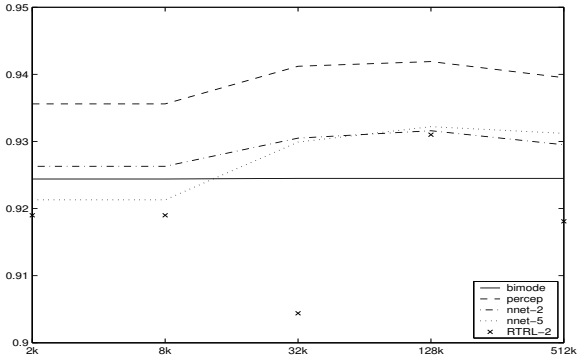

Figure 11: PARSER-REF: Prediction accuracy for different hardware budgets (x-axis).

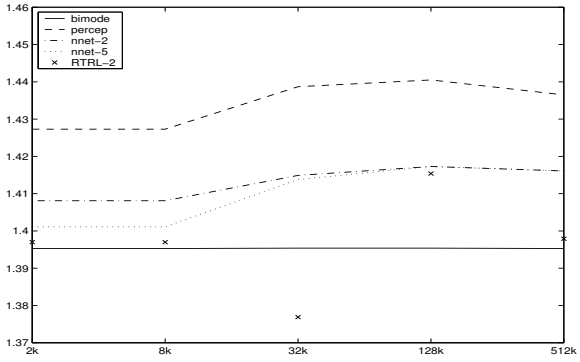

Figure 12: PARSER-REF: IPC for different hardware budgets (x-axis).

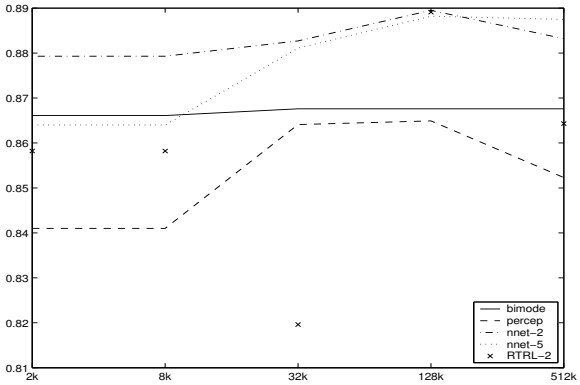

Figure 13: TWOLF-REF: Prediction accuracy for different hardware budgets (x-axis).

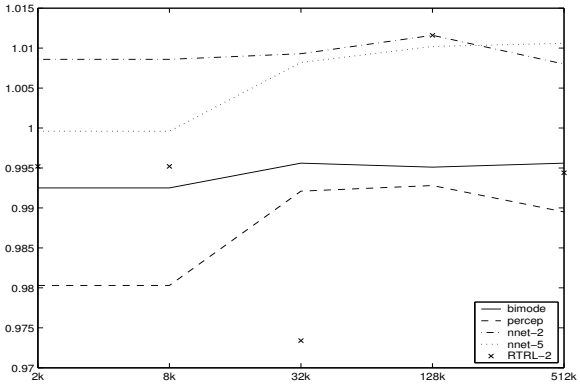

Figure 14: TWOLF-REF: IPC for different hardware budgets (x-axis).

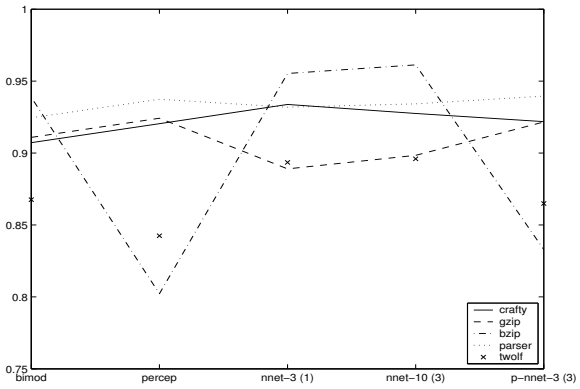

Figure 15: Branch Prediction Accuracy for predictors with different latencies.

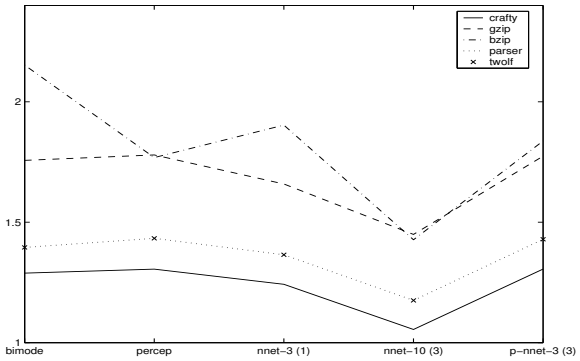

Figure 16: IPC for predictors with different latencies

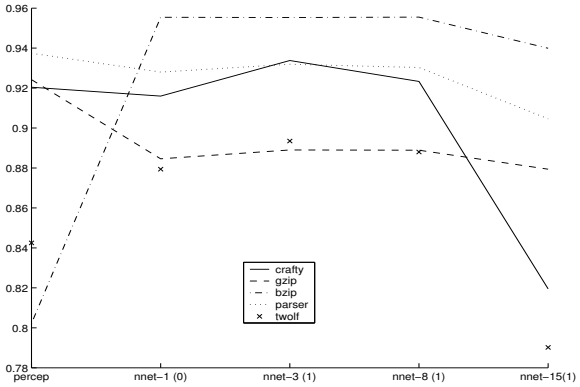

Figure 17: Branch Prediction Accuracy for diffferent networks.

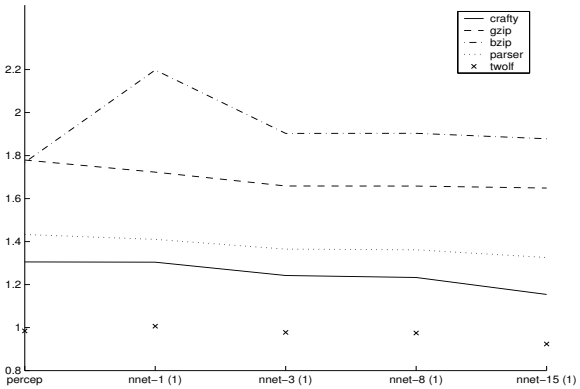

Figure 18: IPC for different networks

7 Conclusions

Recurrent networks learn far too slowly to be of value in the branch prediction problem. (In early experiments developing this simulation, it took over 100,000 training examples for a 3-unit network to learn to XOR it's current input with its previous input.) This slowness, combined with the relatively high memory requirements for the training algorithm (RTRL) eliminate them from the competition.

2-layer feed-forward networs, provide competitive accuracy to the perceptron and more traditional branch predictors, such as bi-mode. Unfortunately, with the added complexity comes added computation time, and in most cases, the increase in accuracy was mitigatged by the extra latencies required. Future work with neural network branch predictors should explore the design space, the tradeoff between the number of hidden units, the number of inputs, and the number of networks. Future work should also include an exploration of the learning rate; throughout this paper, a conservative estimate of a good learning rate was used (.02) to ensure the network didn't behave erratically.

The most consistent trend throughout this paper is that perceptrons are the best predictors. Unlike bimode predictors, the perceptron increases accuracy as the hardware budget increases. Future work should also include a study of whether the neural-network predictors could be more efficiently executed.

Future work should also include more benchmark studies. The consistently poor performance of the perceptrons on the BZIP-PROGRAM-REF benchmark suggests it is not optimal for all situations

8 Bibliography

[1] Jimenez and Lin, "Neural Branch Prediction" *HPCA*. pages 197-206. (2001)

[2] McFarling, "Combining Branch Predictors" Tech. Report *Proceedings of the 11th International Conference on Supercomputing.* pages 285-292. (1997)

[3] Sherwood, Perelman, Hamerly, and Calder. "Automatic. Characterization of Large Scale Program Behavior." *10th International Conference on Architecture Support for Programming Langauges and Operating Systems.* (2002)

[4]Sherwood, Perelman, Hamerly, Calder, Lau, Van Biesbrouck. "SimPiont" Available at http://www.cse.ucsd.edu/users/calder/simpoint

[5] Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press. (1994)

[6] Williams and Zipser. "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks." *Neural Computation.* pages 270-280. (1989)