Perceptrons for Dummies

Daniel A. Jiménez

Department of Computer Science
Rutgers University

Conditional Branch Prediction is a Machine Learning Problem

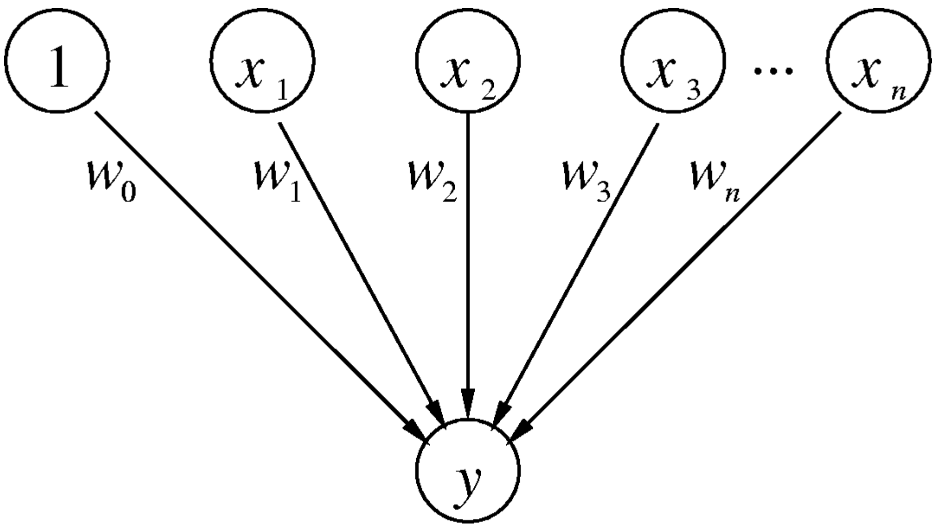
- The machine learns to predict conditional branches
- So why not apply a machine learning algorithm?
- Artificial neural networks
  - Simple model of neural networks in brain cells
  - Learn to recognize and classify patterns
- We used fast and accurate perceptrons [Rosenblatt `62, Block `62] for dynamic branch prediction [Jiménez & Lin, HPCA 2001]

Input and Output of the Perceptron

- The inputs to the perceptron are branch outcome histories
  - Just like in 2-level adaptive branch prediction
  - Can be global or local (per-branch) or both (alloyed)
  - Conceptually, branch outcomes are represented as
    - +1, for taken
    - -1, for not taken
- The output of the perceptron is
  - Non-negative, if the branch is predicted taken
  - Negative, if the branch is predicted not taken
- Ideally, each static branch is allocated its own perceptron

Branch-Predicting Perceptron

- Inputs (x's) are from branch history and are -1 or +1
- n + 1 small integer weights (w's) learned by on-line training
- Output (y) is dot product of x's and w's; predict taken if $y \geq 0$
- Training finds correlations between history and outcome



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

Training Algorithm

$x_{1..n}$ is the n-bit history register, x_0 is 1.
$w_{0..n}$ is the weights vector.
t is the Boolean branch outcome.
θ is the training threshold.

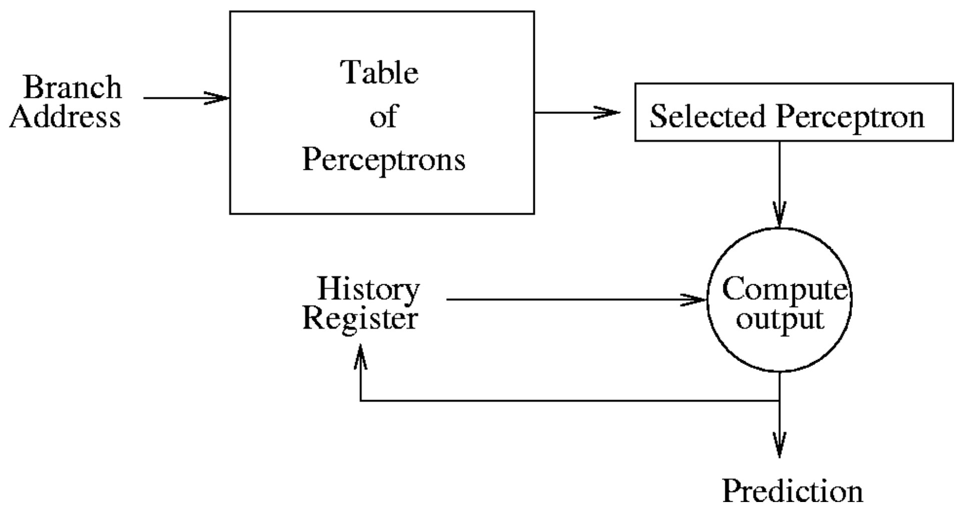
```
if |y| ≤ θ or ((y ≥ 0) ≠ t) then
    for each 0 ≤ i ≤ n in parallel
        if t = x_i then
            w_i := w_i + 1
        else
            w_i := w_i − 1
        end if
    end for
end if
```

What Do The Weights Mean?

- The bias weight, w_0:
  - Proportional to the probability that the branch is taken
  - Doesn't take into account other branches; just like a Smith predictor
- The correlating weights, w_1 through w_n:
  - w_i is proportional to the probability that the predicted branch agrees with the i^{th} branch in the history
- The dot product of the w's and x's
  - $w_i \times x_i$ is proportional to the probability that the predicted branch is taken based on the correlation between this branch and the i^{th} branch
  - Sum takes into account all estimated probabilities
- What's θ?
  - Keeps from overtraining; adapt quickly to changing behavior

Organization of the Perceptron Predictor

- Keeps a table of *m* perceptron weights vectors
- Table is indexed by branch address modulo *m*



[Jiménez & Lin, HPCA 2001]

Mathematical Intuition

A perceptron defines a hyperplane in n+1-dimensional space:

$$y = w_n x_n + w_{n-1} x_{n-1} + \ldots + w_1 x_1 + w_0$$

For instance, in 2D space we have: $y = w_1 x_1 + w_0$

This is the equation of a line, the same as $y = mx + b$

Mathematical Intuition continued

In 3D space, we have $y = w_1x_1 + w_2x_2 + w_0$

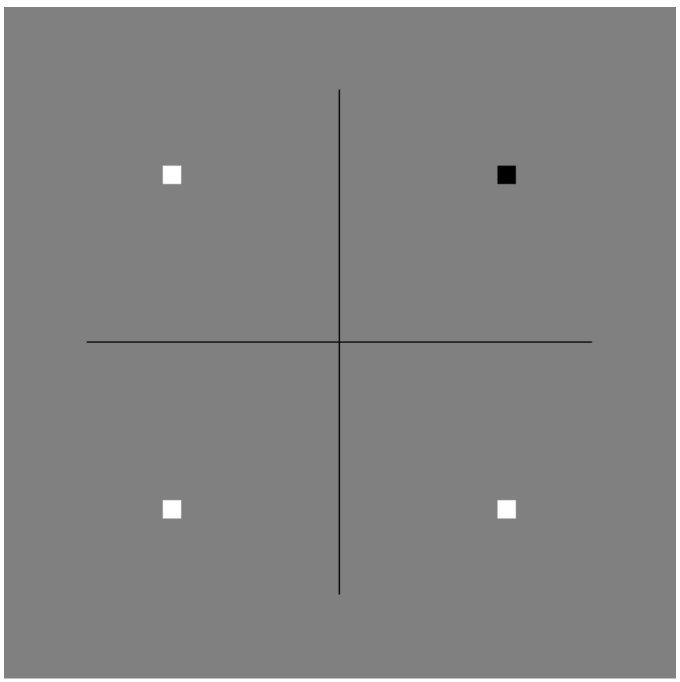
Or you can think of it as $z = m_1x + m_2y + b$

i.e. the equation of a plane in 3D space

This hyperplane forms a *decision surface* separating predicted taken from predicted not taken histories. This surface intersects the feature space. Is it a linear surface, e.g. a line in 2D, a plane in 3D, a cube in 4D, etc.

Example: AND

- Here is a representation of the AND function
- White means *false*, black means *true* for the output
- -1 means *false*, +1 means *true* for the input



-1 AND -1 = false

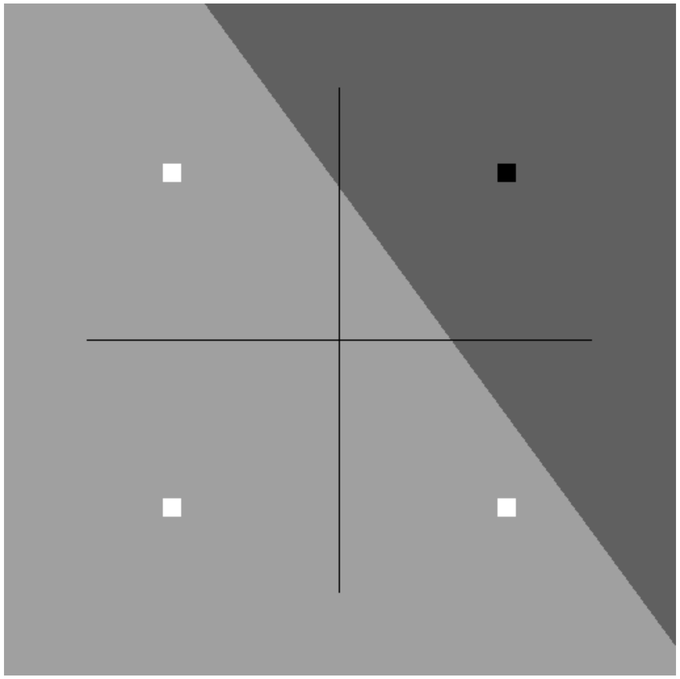
-1 AND +1 = false

+1 AND -1 = false

+1 AND +1 = true

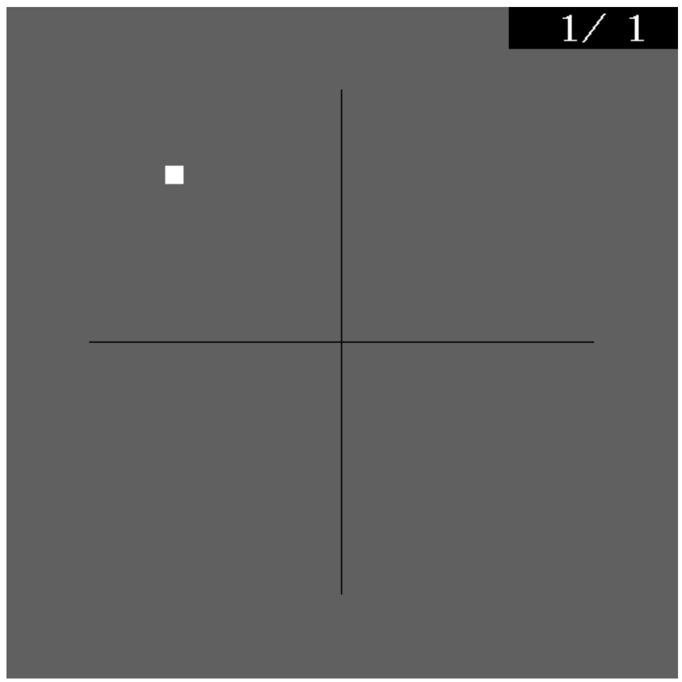
Example: AND continued

- A linear decision surface (i.e. a plane in 3D space) intersecting the feature space (i.e. the 2D plane where z=0) separates *false* from *true* instances



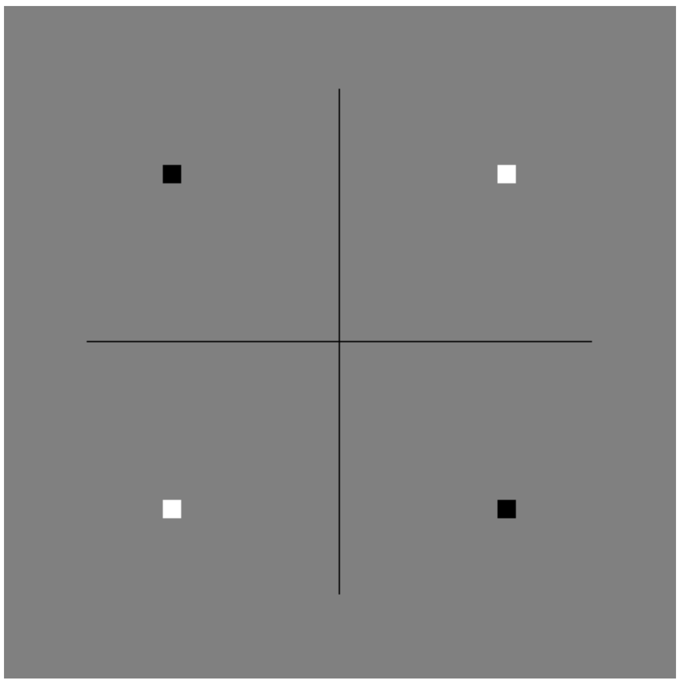
Example: AND continued

- Watch a perceptron learn the AND function:



Example: XOR

- Here's the XOR function:



-1 XOR -1 = *false*

-1 XOR +1 = *true*

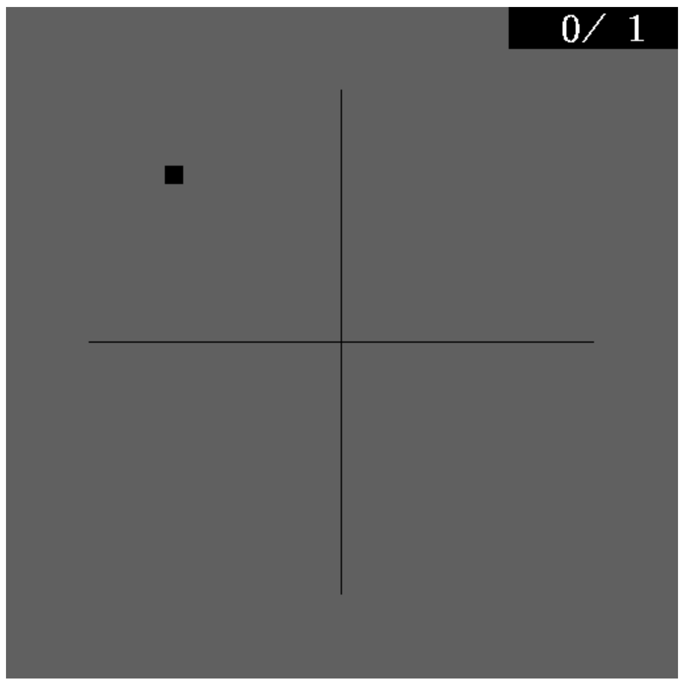
+1 XOR -1 = *true*

+1 XOR +1 = *false*

Perceptrons cannot learn such *linearly inseparable* functions

Example: XOR continued

- Watch a perceptron try to learn XOR



Concluding Remarks

- Perceptron is an alternative to traditional branch predictors
- The literature speaks for itself in terms of better accuracy
- Perceptrons were nice but they had some problems:
  - Latency
  - Linear inseparability

The End

Idealized Piecewise Linear Branch Prediction

Daniel A. Jiménez

Department of Computer Science
Rutgers University

Previous Neural Predictors

- The perceptron predictor uses only pattern history information
  - The same weights vector is used for every prediction of a static branch
  - The i^{th} history bit could come from any number of static branches
  - So the i^{th} correlating weight is aliased among many branches
- The newer path-based neural predictor uses path information
  - The i^{th} correlating weight is selected using the i^{th} branch address
  - This allows the predictor to be pipelined, mitigating latency
  - This strategy improves accuracy because of path information
  - But there is now even more aliasing since the i^{th} weight could be used to predict many different branches

Piecewise Linear Branch Prediction

- Generalization of perceptron and path-based neural predictors
- Ideally, there is a weight giving the correlation between each
  - Static branch *b*, and
  - Each pair of branch and history position (i.e. *i*) in *b*'s history
- *b* might have 1000s of correlating weights or just a few
  - Depends on the number of static branches in *b*'s history
- First, I'll show a "practical version"

The Algorithm: Parameters and Variables

- *GHL* – the global history length
- *GHR* – a global history shift register
- *GA* – a global array of previous branch addresses
- *W* – an $n \times m \times (GHL + 1)$ array of small integers

The Algorithm: Making a Prediction

Weights are selected based on the current branch and the i^{th} most recent branch

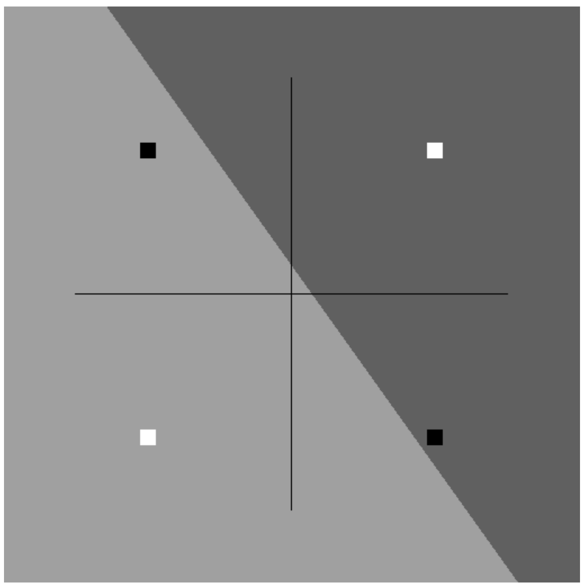
function *predict* (*address*: integer): boolean
begin
    $output := W[address, 0, 0]$
    for i in 1..*GHL* do
        if $GHR[i] =$ true then
            $output := output + W[address \bmod n, GA[i] \bmod m, i]$
        else
            $output := output - W[address \bmod n, GA[i] \bmod m, i]$
        end if
    end for
    $predict := output \geq 0$
end

The Algorithm: Training

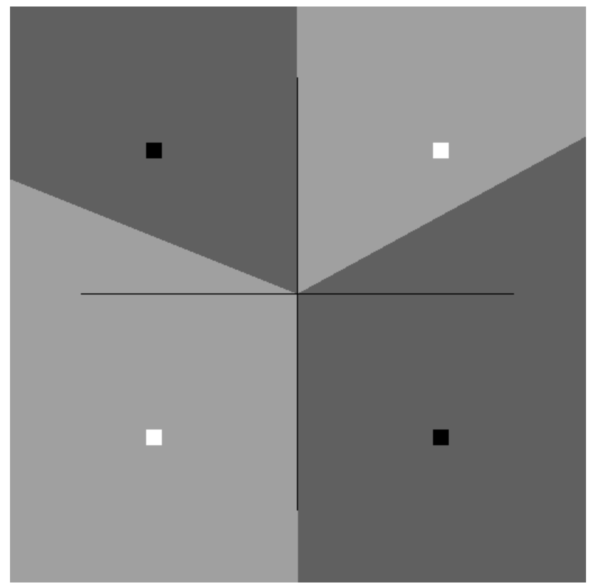
procedure *train* (*address*: integer; *taken*: boolean)
begin
    if $|output| < \theta$ or $output \geq 0 \neq$ taken then
        if *taken* $=$ true then
            $W[address \bmod n, 0, 0] = W[address \bmod n, 0, 0] + 1$
        else
            $W[address \bmod n, 0, 0] = W[address \bmod n, 0, 0] - 1$
        end if
        for i in $1..GHL$
            if $GHR[i] = taken$ then
                $W[address \bmod n, GA[i] \bmod m, i] = W[address \bmod n, GA[i] \bmod m, i] + 1$
            else
                $W[address \bmod n, GA[i] \bmod m, i] = W[address \bmod n, GA[i] \bmod m, i] - 1$
            end if
        end for
    end if
    $GA[2..GHL] := GA[1..GHL-1]$
    $GA[1] := address$
    $GHR[2..GHL] := GHR[1..GHL-1]$
    $GHR[1] := taken$
end

Why It's Better

- Forms a piecewise linear decision surface
  - Each piece determined by the path to the predicted branch
- Can solve more problems than perceptron



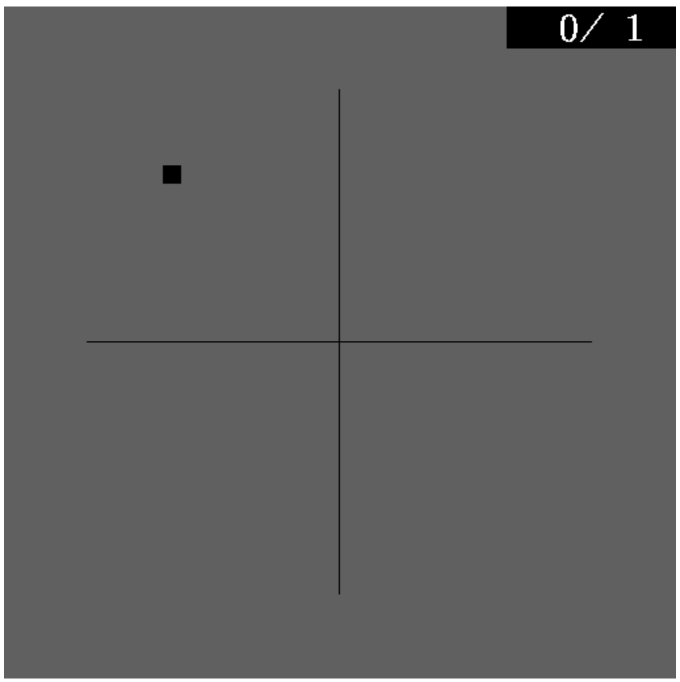
Perceptron decision surface for XOR doesn't classify all inputs correctly



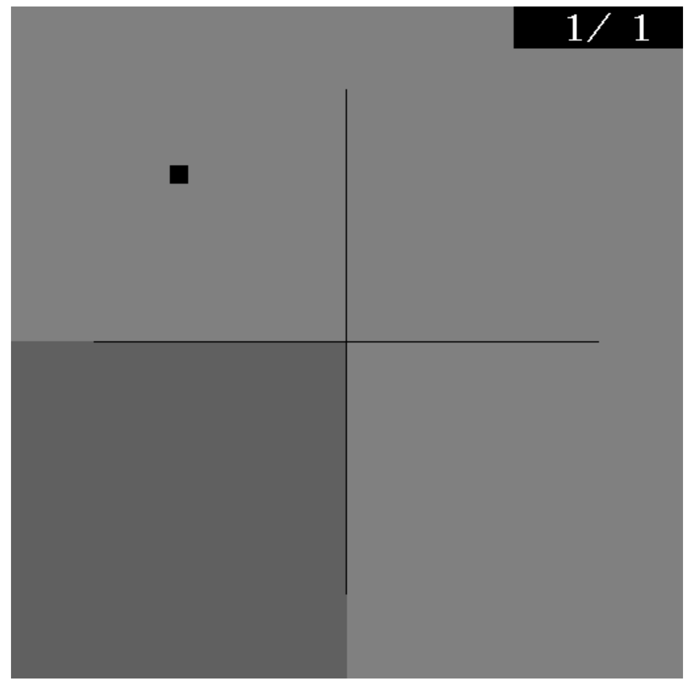
Piecewise linear decision surface for XOR classifies all inputs correctly

Learning XOR

- From a program that computes XOR using `if` statements



perceptron prediction

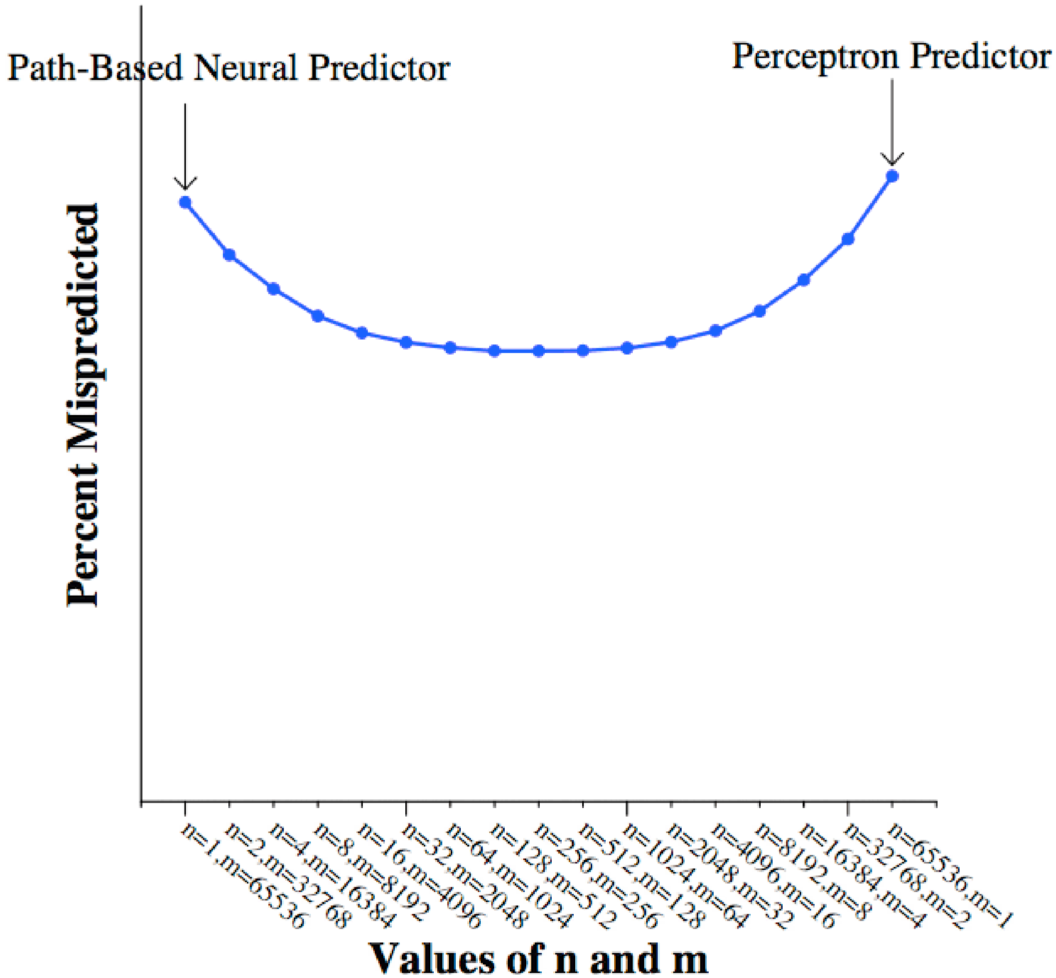


piecewise linear prediction

A Generalization of Neural Predictors

- When $m = 1$, the algorithm is exactly the perceptron predictor
  - $W[n,1,h+1]$ holds n weights vectors
- When $n = 1$, the algorithm is path-based neural predictor
  - $W[1,m,h+1]$ holds m weights vectors
  - Can be pipelined to reduce latency
- The design space in between contains more accurate predictors
- If n is small, predictor can still be pipelined to reduce latency

Generalization Continued



Perceptron and path-based are the least accurate extremes of piecewise linear branch prediction!

Idealized Piecewise Linear Branch Prediction

- Get rid of *n* and *m*
- Allow 1^{st} and 2^{nd} dimensions of W to be unlimited
- Now branches cannot alias one another; accuracy much better
- One small problem: unlimited amount of storage required
- How to squeeze this into 65,792 bits for the contest?

Hashing

- 3 indices of W : i, j, & k, index arbitrary numbers of weights
- Hash them into 0..N-1 weights in an array of size N
- Collisions will cause aliasing, but more uniformly distributed
- Hash function uses three primes H_1 H_2 and H_3:

function *hash* (i,j,k : integer): integer
begin
$$hi := i \times H_1$$
$$hj := j \times H_2$$
$$hk := k \times H_3$$
$$hash := (hi \text{ xor } hj \text{ xor } hk) \bmod N$$
end

More Tricks

- Weights are 7 bits, elements of *GA* are 8 bits
- Separate arrays for bias weights and correlating weights
- Using global and per-branch history
  - An array of per-branch histories is kept, alloyed with global history
- Slightly bias the predictor toward not taken
- Dynamically adjust history length
  - Based on an estimate of the number of static branches
- Extra weights
  - Extra bias weights for each branch
  - Extra correlating weights for more recent history bits
- Inverted bias weights that track the opposite of the branch bias

Parameters to the Algorithm

```
#define NUM_WEIGHTS 8590
#define NUM_BIASES 599
#define INIT_GLOBAL_HISTORY_LENGTH 30
#define HIGH_GLOBAL_HISTORY_LENGTH 48
#define LOW_GLOBAL_HISTORY_LENGTH 18
#define INIT_LOCAL_HISTORY_LENGTH 4
#define HIGH_LOCAL_HISTORY_LENGTH 16
#define LOW_LOCAL_HISTORY_LENGTH 1
#define EXTRA_BIAS_LENGTH 6
#define HIGH_EXTRA_BIAS_LENGTH 2
#define LOW_EXTRA_BIAS_LENGTH 7
#define EXTRA_HISTORY_LENGTH 5
#define HIGH_EXTRA_HISTORY_LENGTH 7
#define LOW_EXTRA_HISTORY_LENGTH 4
#define INVERTED_BIAS_LENGTH 8
#define HIGH_INVERTED_BIAS_LENGTH 4
#define LOW_INVERTED_BIAS_LENGTH 9
```

```
#define NUM_HISTORIES 55
#define WEIGHT_WIDTH 7
#define MAX_WEIGHT 63
#define MIN_WEIGHT -64
#define INIT_THETA_UPPER 70
#define INIT_THETA_LOWER -70
#define HIGH_THETA_UPPER 139
#define HIGH_THETA_LOWER -136
#define LOW_THETA_UPPER 50
#define LOW_THETA_LOWER -46
#define HASH_PRIME_1 511387U
#define HASH_PRIME_2 660509U
#define HASH_PRIME_3 1289381U
#define TAKEN_THRESHOLD 3
```

All determined empirically with an *ad hoc* approach

References

- Me and Lin, HPCA 2001 (perceptron predictor)
- Me and Lin, TOCS 2002 (global/local perceptron)
- Me, MICRO 2003 (path-based neural predictor)
- Juan, Sanjeevan, Navarro, SIGARCH Comp. News, 1998 (dynamic history length fitting)
- Skadron, Martonosi, Clark, PACT 2000 (alloyed history)

The End

Program to Compute XOR

```
int f () {
        int a, b, x, i, s = 0;

        for (i=0; i<100; i++) {
                a = rand () % 2;
                b = rand () % 2;
                if (a) {
                        if (b)
                                x = 0;
                        else
                                x = 1;
                } else {
                        if (b)
                                x = 1;
                        else
                                x = 0;
                }
                if (x) s++; /* this is the branch */
        }
        return s;
}
```