Implementing Perceptrons in P4

Jack Morris

CS6114 Final Project - December 2021

1 Introduction

This document describes the implementation of a perceptron in Programming Protocol-independent Packet Processors (P4), a programming language for networks devices. This implementation makes it possible to offload the calculations necessary for a perceptron (matrix multiplication and vector addition) to network devices. This is a preliminary exploration to demonstrate how inference for machine learning models can be accelerated by utilizing the computational capabilities of networking devices.[1]

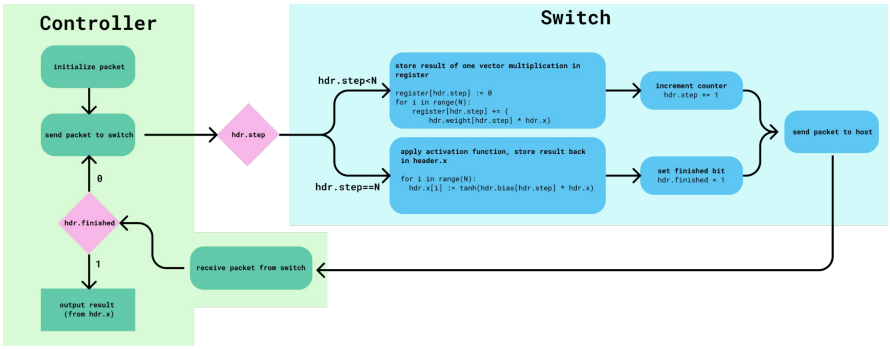

Figure 1: Flowchart of a perceptron modeled in P4. For input $x \in \mathbb{R}^N$, the packet must make N passes through the switch to complete the computation. This model is described in section 2.

1.1 Perceptron

The perceptron is an algorithm for learning a predictive function from data. Perceptrons are at the heart of the big breakthroughs in machine learning:

[1] Code for this project is available on GitHub: https://github.com/jxmorris12/p4-perceptron/.

speech recognition, machine translation, and self-driving cars. At the heart of perceptrons is a single computation:

$$\texttt{perceptron}(x|W,b) = f(Wx^T + b)$$

where $x \in \mathbb{R}^n$ is a vector input, $W \in \mathbb{R}^{nxn}$ is a learned matrix called the *weights*, $b \in \mathbb{R}^n$ is a learned vector called the *biases*. f is some nonlinear function called the *activation function*. One choice of activation function is $tanh(x)$. In practice, the choice of activation function is something of an engineering decision, and there may be many suitable choices. It is mostly important that the activation function is nonlinear - it is this property of nonlinearity that makes perceptrons powerful.

The goal of the perceptron is to learn W and b such that $\texttt{perceptron}(x|w,b)$ represents something useful about the world, for example, mapping a picture of a stoplight to a variable representing whether that light is red, yellow, or green.

1.2 How can we implement perceptrons without floats?

For this work, we focus on a simplification of the perceptron optimized to run quickly on network hardware. In particular, P4 lacks a floating-point type. We could circumvent this shortcoming by approximating somehow with integers, or storing the results of floating-point calculations in a lookup table. However, we are interested in the scenario where all the heavy computational work is done on the switch. So we choose to focus on a specific type of perceptron architecture, BinaryNet [1], which does not utilize any floating-point values (!).

1.3 BinaryNet

In the BinaryNet architecture, all weights and activations are binarized before the perceptron computation. For simplicity, we implement the deterministic binarization function reference in the paper; that is, we set $x^b = sign(x)$ before inputting a variable x to the perceptron. We assume that all weights and biases obtained via training of BinaryNet take on the values -1 or $+1$.

2 P4 Perceptron Model

Our P4 program implements the computation $tanh(Wx^T + b)$ in switch hardware, with the assumption that W, x, and b are binarized to either -1 or 1. This allows us to represent each value with a single bit, with 0 corresponding to a value of -1 and 1 corresponding to a value of 1.

The P4 program takes a packet as input with headers providing W, x, and b, as well as an initial step counter of 0. We unroll the matrix computation Wx^T over N steps, where N is the dimensionality of x and b. This means that a single round-trip packet transmission represents a single vector-wise product from the matrix computation. There are N total vector-wise products to compute, so the perceptron computation takes a total of N packet round trips.

```
header perceptron_t {
      bit<7> step_acc = 0;
      bit<1> finished = 0;
      bit<256> weight = [0, 1, 0, 1, 0, ..., 0];
      bit<16> bias = [0, 0, 0, 1, 1, ..., 1];
      bit<16> x = [1, 1, 1, 0, 1, ..., 1];
}
```

Figure 2: Example header fields for perceptron modeled in P4.

At each step i, a single vector-wise product $W[i] \cdot x$ is computed and stored in a register. After computing and storing products $\forall i \in 1..N$, the full results of Wx^T are stored in a register. In a single final step, each value $(Wx^T)[i]$ is retrieved from the register and added to the corresponding bias $b[i]$. Finally, the activation function $tanh((Wx^T)[i] + b[i])$ is applied. The result of the full computation is stored in the header for the original input, x. An additional bit $finished$ is set to 1 to indicate that the header x was overwritten with the output value $tanh(Wx^T + b)$.

3 Implementation

The perceptron is implemented in two main files, `perceptron.py` and `perceptron.p4`.

The Python script uses the Python library `numpy` for linear algebra. It generates a random W, x, b with values in $\{+1, -1\}$. It binarizes the values and creates a packet with the proper headers to send to the switch running `perceptron.p4`. The script loops, sending and receiving packets until the `packet.finished` bit is set. It then compares the value received from the P4 computation to one computed using `numpy` to ensure correctness.[2]

The P4 script realizes the aforementioned perceptron model.

The code for this project is available in this Github repository. To execute the code, run `make controller` and `make run` in separate windows. On the MiniNet controller, run `h1 python perceptron.py` to run a test computation.

4 Thoughts and potential improvements

This project was an interesting foray into implementing perceptrons in P4. The hardest part was getting the bitwise math right in P4 – the type checker in P4 proved very useful in distinguishing between numbers that could take -1 or 1 (represented using the `int<4>` type) and numbers that could only be 0 or 1 (represented using `bit<1>`).

Convolutional layers. The full BinaryNet architecture is a composition of

[2]Since the numbers are randomly generated, one can execute `perceptron.py` many times to see that, for any randomly generated W, x, b, the switch is able to properly compute $tanh(Wx^T + b)$.

binarized linear layers (like implemented here) and binarized convolutional layers. To run the full BinaryNet inference procedure on a networking device, one would need to implement a similar protocol for computing convolutions on the switch. This is also non-obvious because P4 lacks loops.

Stochastic binarization. The BinaryNet paper describes a superior binarization technique, where x is stochastically binarized – mapped to -1 with some probability $p = \sigma(x)$ and mapped to 1 with probability $1 - p$. This would be an improvement over the current deterministic binarization.

Single-step matrix computation. Another issue is that the matrix computation is split over N steps, meaning that it will take N packet round-trips to finish the computation. Although this is fine for $N = 16$, the BinaryNet paper also has linear layers as large as $N = 4096$. Ideally the full matrix computation could be done in one pass on the switch. It is not obvious how to do this because the P4 language lacks loops.

Variable N. One obvious practical limitation is that the program is hard-coded for $N = 16$. This number was chosen to compare to a single linear layer from the BinaryNet architecture, but it would be better to pass N via header and have W, x, and b as part of the payload.

Use multiple registers. This implementation stores intermediate computations in a scratchpad register. There is only one register globally, meaning that if you tried to do two computations in parallel, the scratchpads would overwrite each other and the answers would be wrong. We could use multiple registers to try and approximate a scratchpad for each possible flow (or at least, to scale to more than 1 computation at a given time). Alternatively, if we implement the single-step matrix computation described above, there won't be a need for a scratchpad register, so this would not be an issue.

Batching. The implementation described here applies only for a single input $x \in \mathbb{R}^N$. The implementation could be improved to operate on a batch of vectors $x \in \mathbb{R}^{BxN}$.

References

[1] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 4114–4122, Red Hook, NY, USA, 2016. Curran Associates Inc.