Interactive Programming for Artificial Intelligence

Deep Learning for Programmers

Dragan Djuric

An Interactive Tutorial with CUDA, OpenCL, MKL-DNN, Java, and Clojure

https://dragan.rocks aiprobook.com

DRAGAN DJURIC

DEEP LEARNING FOR PROGRAMMERS SAMPLE CHAPTER [DRAFT 0.1.0]

DRAGAN ROCKS

The book is available at
https://aiprobook.com/deep-learning-for-programmers.

Subscribe to the Patreon campaign, at
https://patreon.com/deep_learning,

and get access to all available drafts, and a number of nice perks: from various acknowledgments of your support, to complimentary handcrafted hardcover of the book once it is finished (only selected tiers).

All proceeds go towards funding the author's work on these open-source libraries.

By supporting the book you are also supporting the open-source ecosystem.

PUBLISHED BY DRAGAN ROCKS

HTTP://AIPROBOOK.COM

First printing, 19 May 2019

Sample Chapter

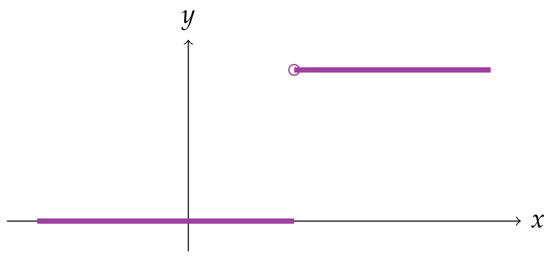
Bias and Activation Function

We continue building our network by adding the activation function and bias.

Threshold and Bias

In the current state, the network combines all layers into a single layer transformation. We can introduce basic decision-making capability by adding a cutoff to the output of each neuron. When the weighted sums of its inputs are below that threshold, the output is zero and when they are above, the output is one.

$$output = \begin{cases} 0 & W\mathbf{x} \leq threshold \\ 1 & W\mathbf{x} > threshold \end{cases} \tag{1}$$



Since we keep the current outputs in a potentially huge vector, it would be inconvenient to write a scalar-based logic. Prefer a vectorized function, or create one if a convenient one is not available.

Neanderthal does not have the cutoff function, but we can create one by subtracting threshold from the maximum of individual thresholds and the values of the signal and then mapping the signum function to the result. There are simpler ways to compute this, but using the existing functions, and doing the computation in-place has educational value. We will soon see that there are better things to use for transforming the output than the vanilla step function.[1]

```
(defn step! [threshold x]
  (fmap! signum (axpy! -1.0 threshold (fmax! threshold x x))))
```

[1] To avoid the clutter, the appropriate namespace declaration will not be shown in all chapters. Please refer to the source code that comes with this book for the details about requiring the functions that are used throughout the book.

```
(let [threshold (dv [1 2 3])
      x (dv [0 2 7])]
  (step! threshold x))

=>
#RealBlockVector[double, n:3, offset: 0, stride:1]
[   0.00    0.00    1.00 ]
```

The next few code samples we will follow a few steps in the evolution of the code, and will reuse weights and x. To simplify the example, we will use global `def` and not care about properly releasing the memory. It will not matter in a REPL session, but do not forget to do it in the real code.

```
(def x (dv 0.3 0.9))

(def w1 (dge 4 2 [0.3 0.6
                  0.1 2.0
                  0.9 3.7
                  0.0 1.0]
             {:layout :row}))

(def threshold (dv 0.7 0.2 1.1 2))
```

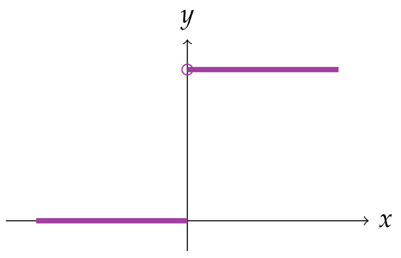
Since we do not care about additional object instances right now, it is more convenient to use the pure `mv` function instead of `mv!`. `mv` creates the resulting vector `y`, instead of mutating the one that has to be provided as an argument.

```
(step! threshold (mv w1 x))

=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[   0.00    1.00    1.00    0.00 ]
```

The bias is simply the threshold moved to the left side of the following equation.

$$output = \begin{cases} 0 & W\mathbf{x} - bias \leq 0 \\ 1 & W\mathbf{x} - bias > 0 \end{cases} \tag{2}$$



With zero threshold, the `step!` function can be used as shown in the following code.

```
(def bias (dv 0.7 0.2 1.1 2))

(def zero (dv 4))

(step! zero (axpy! -1.0 bias (mv w1 x)))

=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[   0.00    1.00    1.00    0.00 ]
```

As bias is the same as threshold there is no need for the additional zero vector, and the code becomes much simpler.

```
(step! bias (mv w1 x))

=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[   0.00    1.00    1.00    0.00 ]
```

Activation Function

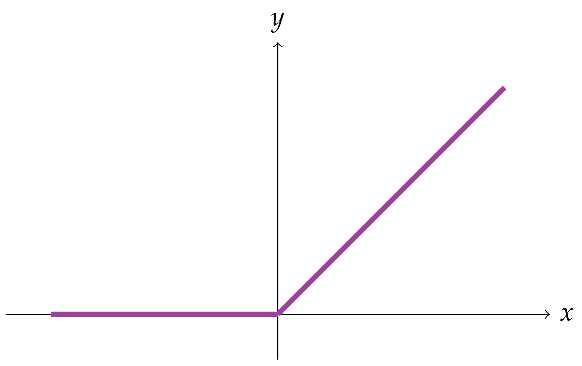
The decision capabilities supported by the step function are rather crude. A neuron either outputs a constant value (1), or zero. It is better to use functions that offer different levels of the signal strength. Instead of the step function, the output of each neuron passes through an *activation function*. Countless functions can be an activation function, but a handful proved to work exceptionally well.

Like neural networks themselves, the functions that work well are simple. Activation functions have to be chosen carefully, to support the *learning* algorithms, most importantly to be easily differentiable. Until recently, the *sigmoid* and *tanh* functions were the top picks. Recently an even simpler function, *ReLU*, became the activation function of choice.

Rectified Linear Unit (ReLU)

ReLU is short for Rectified Linear Unit. The name sounds mysterious, but it is a straightforward linear function that has zero value below the threshold, which is typically zero.

$$f(x) = max(0, x) \tag{3}$$



ReLU is even simpler to implement than the step function.

```
(defn relu! [threshold x]
  (axpy! -1.0 threshold (fmax! threshold x x)))
```

It might seem strange to keep the threshold as an argument to the `relu` function. Isn't ReLU always cut-off at zero? Consider it a bit of optimization. Imagine that here is no built-in optimized ReLU function. To implement the formula $f(x) = max(0, x)$ we either have to use a mapping over the `max` function, or to use the vectorized `fmax`, which requires an additional vector that holds the zeroes. Since we need to subtract the biases before the activation anyway, by fusing these two phases, we avoided the need for maintaining an additional array of zeroes.

```
(relu! bias (mv w1 x))
```

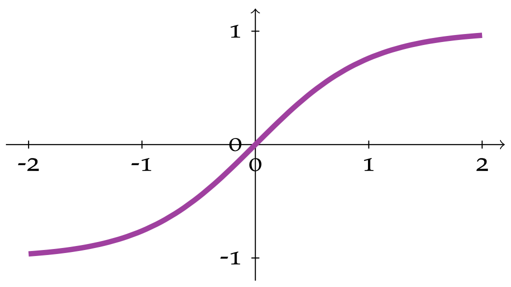
```
=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[   0.00    1.63    2.50    0.00 ]
```

This may, or may not, be the best choice for a complete library, but since we are doing this to learn, we stick to the Yagni principle. Of course, in real work it is best to use the `relu` function that comes with Neanderthal.

Hyperbolic Tangent (tanh)

Hyperbolic tangent is a popular activation function.

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{4}$$



Note how it is close to the identity function $f(x) = x$ in large parts of the domain between -1 and 1. As the absolute value of x gets larger, $tanh(x)$ asymptotically approaches 1. Thus, the output is between -1 and 1.

Since Neanderhtal has the vectorized variant of the `tanh` function in its `vect-math` namespace, the implementation is easy.

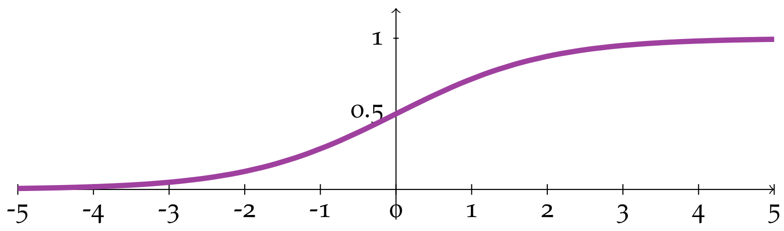
```
(tanh! (axpy! -1.0 bias (mv w1 x)))

=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[  -0.07    0.93    0.99   -0.80 ]
```

Sigmoid function

Before ReLU became popular, sigmoid was the most often used activation function. Sigmoid refers to a whole family of S-shaped functions, or, often, to a special case: the *logistic function*.

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1} \tag{5}$$



Standard libraries often do not come with an implementation of the sigmoid function. We have to implement our own. We could implement it in the most straightforward way, just following the formula. That might be a good approach if we are flexing our muscles, but may be not completely safe if we intend to use such implementation for the real work. `(exp 710)` is too big to fit even in `double`, while `(exp 89)` does not fit into `float` and produce the infinity (`##Inf`).

```
[(exp 709) (exp 710) (float (exp 88)) (float (exp 89))]

=>
[8.218407461554972E307 ##Inf 1.6516363E38 ##Inf]
```

Tackle that implementation as an exercise, after we take another approach instead. Let me pull the following equality out of the magic mathematical hat, and ask you to believe me that it is true.

$$S(x) = \frac{1}{2} + \frac{1}{2} \times tanh(\frac{x}{2}) \tag{6}$$

We can implement that easily by combining the vectorized `tanh!` and a bit of vectorized scaling.

```
(defn sigmoid! [x]
  (linear-frac! 0.5 (tanh! (scal! 0.5 x)) 0.5))
```

Let's program our layer with the logistic sigmoid activation.

```
(sigmoid! (axpy! -1.0 bias (mv w1 x)))

=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[   0.48    0.84    0.92    0.25 ]
```

You can benchmark both implementations with large vectors, and see whether there is a difference in performance. I expect that it takes only a fraction of the complete run time of the network. Consider that `tanh(x)` is correct, since it comes from a standard library, while a straightforward formula translation might not be good enough for a particular use case.

Layers with activations

The layers of our fully connected network now go beyond linear transformations. We can stack as many as we'd like and do the inference.

Figure 1 is an updated version of the simple neural network with one hidden layer from the previous chapter, shown in Figure 2. It explicitly shows bias as an additional node. Since it connects with each node in the following layer, it can be represented with a vector. The activations are not shown to avoid clutter, but assume that each neuron has an activation at the output.

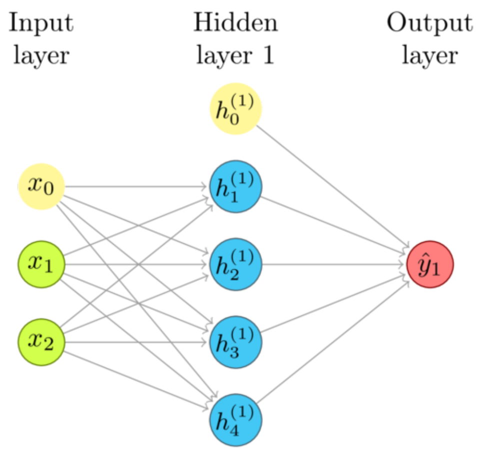


Figure 1: A simple network with bias shown as an additional node (yellow).

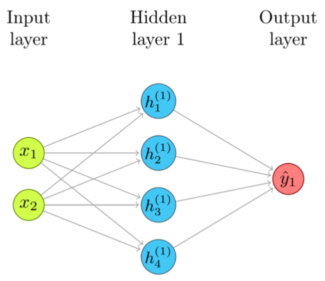


Figure 2: A simple network from the previous chapter repeated for convenience.

The code corresponding to this image builds on the existing example from the previous chapter.

```
(with-release [x (dv 0.3 0.9)
               w1 (dge 4 2 [0.3 0.6
                            0.1 2.0
                            0.9 3.7
                            0.0 1.0]
                       {:layout :row})
               bias1 (dv 0.7 0.2 1.1 2)
               h1 (dv 4)
               w2 (dge 1 4 [0.75 0.15 0.22 0.33])
               bias2 (dv 0.3)
               y (dv 1)]
  (tanh! (axpy! -1.0 bias1 (mv! w1 x h1)))
  (println (sigmoid! (axpy! -1.0 bias2 (mv! w2 h1 y)))))
```

```
=>
#RealBlockVector[double, n:1, offset: 0, stride:1]
[   0.44 ]
```

This is getting repetitive. For each layer we add, we have to herd a few more disconnected matrices, vectors, and activation functions in place. In the next chapter, we will fix this by abstracting it into easy to use layers.