Interactive Programming for Artificial Intelligence

Deep Learning for Programmers

Dragan Djuric

SAMPLE CHAPTER

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

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

Representing layers and connections

Our journey of building a deep learning library that runs on both CPU[1] and GPU[2] begins.

[1] We will support x86/amd64-based processors on all 3 major operating systems: Linux, Windows, and macOS.

[2] We will support all major vendors: Nvidia via CUDA, and AMD, Intel, and Nvidia via OpenCL.

Neural networks structure

Figure 1 shows a typical neural network diagram. As the story usually goes, we plug some input data into the input layer, and the network then propagates the signal through hidden layer 1 and hidden layer 2, via weighted connections, to produce the output at the output layer. For example, the input data is the pixels of an image, and the output are "probabilities" of this image belonging to a class, such as *cat* (y_1) or *dog* (y_2).

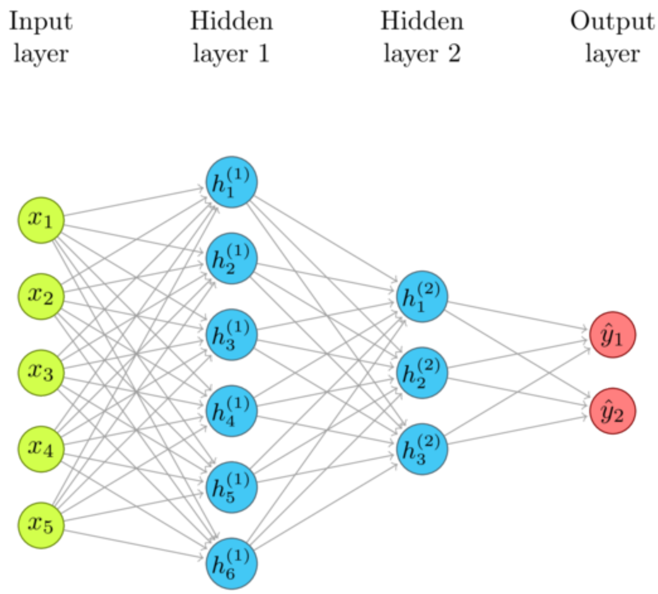

Figure 1: A typical neural network.

Neural Networks are often used to classify complex things such as objects in photographs, or to "predict" future data. Mechanically, though, there is no magic. They just approximate functions. What exactly are inputs and outputs is not particularly important at this moment.

The network can approximate (or, to be fancy, "predict"), even mundane functions such as, for example, $y = sin(x_1) + cos(x_2)$.

An even simpler network is shown in Figure 2.

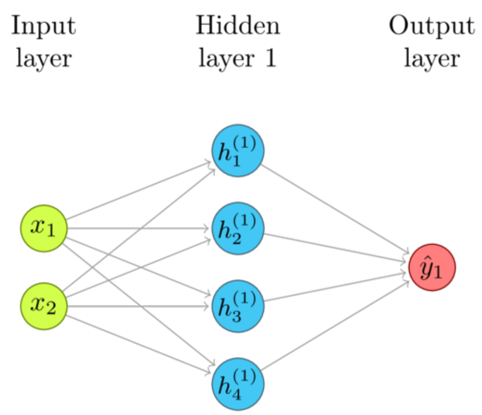

Figure 2: A simple network with two neurons in the input, two neurons in the output, and four neurons in the single hidden layer.

A neural network can be seen as a *transfer function*. We provide an input, and the network propagates that signal to calculate the output. On the surface, this is what many computer programs do anyway.

Different from everyday functions that we use, neural networks compute anything using only this architecture of nodes and weighted connections. The trick is in finding the right weights so that the approximation is close to the "right" value. This is what *learning* in deep learning is all about. At this moment, though, we are only dealing with *inference*, the process of computing the output using the given structure, input, and whatever weights there are.

Approaching the implementation

The most straightforward thing to do, and the most naive error to make, is to read about analogies with neurons in the human brain, look at these diagrams, and try to model nodes and weighted connections as first-class objects. This might be a good approach with business-oriented problems. First-class objects might bring the ultimate flexibility: each node and connection could have different polymorphic logic. In practice, that flexibility does not work well. Even if it could help with better inference, it would be much slower, and training such a wandering network would be a challenge.

Rather than in such enterprising "neurons", the strength of neural networks is in their *simple structure*. Each node in a layer and each connection between two layers has exactly the same structure and logic. The only moving parts are the numerical values in weights and thresholds. We can exploit that to create efficient implementations that fit well into hardware optimizations for numerical computations.

I would say that the human brain analogy is more a product of marketing than a technical necessity. A layer of a basic neural network basically does logistic regression. There are more advanced structures, but the point is that they do not implement anything close to biological neurons.

The math

Let's just consider the input layer, the first hidden layer, and the connections between them, shown in Figure 3.

We can represent the input with a vector of two numbers, and the output of the hidden layer 1 with a vector of four numbers. Note that, since there is a weighted connection from each x_n to each $h_m^{(1)}$, there are $m \times n$ connections. The only data about each connection are its weight, and the nodes it connects. That fits well with what a matrix can represent. For example, the number at w_{21} is the weight between the first input, x_1 and the second output, $h_2^{(1)}$.

Here's how we compute the output of the first hidden layer:

$$
\begin{aligned}
h_1^{(1)} &= w_{11} \times x_1 + w_{12} \times x_2 \\
h_2^{(1)} &= w_{21} \times x_1 + w_{22} \times x_2 \\
h_3^{(1)} &= w_{31} \times x_1 + w_{32} \times x_2 \\
h_4^{(1)} &= w_{41} \times x_1 + w_{42} \times x_2
\end{aligned}
$$

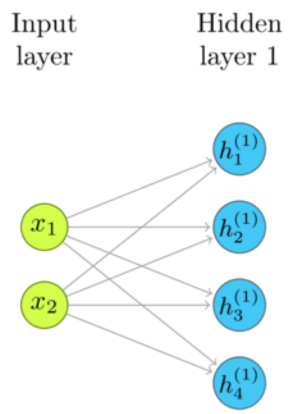

Figure 3: Connections between two layers

These are technically four dot products[3] between the corresponding rows of the weight matrix and the input vector.

[3] Read more in the chapter on Vector Spaces.

$$
\begin{aligned}
h_1^{(1)} &= \vec{w}_1 \cdot \vec{x} = \sum_{j=1}^{n} w_{1j} x_j \\
h_2^{(1)} &= \vec{w}_2 \cdot \vec{x} = \sum_{j=1}^{n} w_{2j} x_j \\
h_3^{(1)} &= \vec{w}_3 \cdot \vec{x} = \sum_{j=1}^{n} w_{3j} x_j \\
h_4^{(1)} &= \vec{w}_4 \cdot \vec{x} = \sum_{j=1}^{n} w_{4j} x_j
\end{aligned}
$$

Conceptually, we can go further than low-level dot products. The weight matrix *transforms* the input vector into the hidden layer vector. We do not have to juggle indexes and program low-level loops. The basic matrix-vector product implements the propagation from each layer to the next![4]

[4] Read more in the chapter on Matrix Transformations.

$$\mathbf{h}^{(1)} = W^{(1)}\mathbf{x}$$
$$\mathbf{y} = W^{(2)}\mathbf{h}^{(1)}$$

For example, for some specific input and weights[5] the network shown in Figure 3 computes in the following way:

[5]
$$h_1^{(1)} = 0.3 \times 0.3 + 0.6 \times 0.9 = 0.09 + 0.54 = 0.63$$
$$h_2^{(1)} = 0.1 \times 0.3 + 2 \times 0.9 = 0.03 + 1.8 = 1.83$$
$$h_3^{(1)} = 0.9 \times 0.3 + 3.7 \times 0.9 = 0.27 + 3.33 = 3.6$$
$$h_4^{(1)} = 0 \times 0.3 + 1 \times 0.9 = 0 + 0.9 = 0.9$$

$$\mathbf{h}^{(1)} = \begin{bmatrix} 0.3 & 0.6 \\ 0.1 & 2 \\ 0.9 & 3.7 \\ 0.0 & 1 \end{bmatrix} \begin{bmatrix} 0.3 \\ 0.9 \end{bmatrix} = \begin{bmatrix} 0.63 \\ 1.83 \\ 3.6 \\ 0.9 \end{bmatrix}$$

The hidden layer then propagates the signal in the same manner (Figure 4).

$$\mathbf{y} = \begin{bmatrix} 0.75 & 0.15 & 0.22 & 0.33 \end{bmatrix} \begin{bmatrix} 0.63 \\ 1.83 \\ 3.6 \\ 0.9 \end{bmatrix} = \begin{bmatrix} 1.84 \end{bmatrix}$$

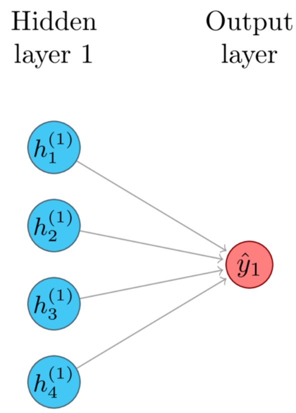

Figure 4: The second transformation.

The code

To try this in Clojure, we require some basic Neanderthal functions.

```
(ns dragan.rocks.dlfp.part-2.representing-layers-and-connections
  (:require [uncomplicate.commons.core :refer [with-release]]
            [uncomplicate.neanderthal
             [native :refer [dv dge]]
             [core :refer [mv!]]]))
```

The minimal code example, following the Yagni principle [6], would be something like this:

[6] Yagni: You Are not Going to Need It.

```
(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 h1 (dv 4))

(mv! w1 x h1)
```

After we evaluate this code, we get the following result in the REPL.

```
=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[   0.00    1.83    3.60    0.90 ]
```

The code directly corresponds to the formulas it has been based on. `w1`, `x`, and `h1` represent weights, input, and the first hidden layer. The function `mv!` applies the matrix transformation `w1` to the vector `x`, by multiplying `w1` by `x`. `mv` can stand for "matrix times vector", or "multiply vector", whatever you prefer as a mnemonic.

We should make sure that this code works well with large networks processed through lots of cycles when we get to implement the *learning* part, so we have to take care to reuse memory; thus we use the destructive version of `mv`, `mv!`. The memory that holds the data is outside of the JVM. We need to take care of its lifecycle and release it (automatically, using `with-release`) as soon it is not needed. This might be unimportant for demonstrative examples, but is crucial in "real" use cases. Here is the same example with proper cleanup.

```
(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})
               h1 (dv 4)]
  (println (mv! w1 x h1)))
```

```
=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[   0.00    1.83    3.60    0.90 ]
```

The output of the hidden layer, computed by the `mv!` function, is the input of the output layer (Figure 5). We transform it by yet another weight matrix, `w2`.

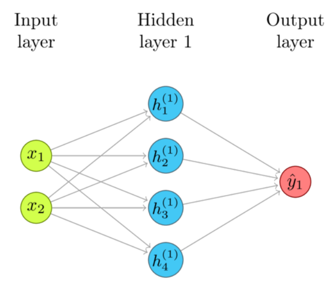

Figure 5: A simple network from Figure 2 repeated for convenience.

```
(def w2 (dge 1 4 [0.75 0.15 0.22 0.33]))

(def y (dv 1))

(mv! w2 (mv! w1 x h1) y)
```

```
=>
#RealBlockVector[double, n:1, offset: 0, stride:1]
[   1.84 ]
```

The final result is $y_1 = 1.84$. Who knows what it represents and which function it approximates. The weights we plugged in are not the result of any training nor insight. I have just pulled some random numbers out of my hat to demonstrate the computation steps.

This is not much but is a good first step

The network we have just created is still a simple toy.

It's not even a proper multi-layer perceptron, since we did not implement non-linear activation of the outputs. Funnily, the nodes we have implemented *are perceptrons*, and there *are* multiple layers full of these. You'll soon get used to the tradition of inventing confusing and inconsistent grand-sounding names for every incremental feature in machine learning. Without non-linearity introduced by activations, we could stack thousands of layers, and our "deep" network would still perform only linear approximation equivalent to a single layer[7].

We have not implemented thresholds, or *biases*, yet. We've also left everything in independent matrices and vectors, without a structure involving *layers* that would hold them together. And we haven't even touched the *learning* part, which is 95% of the work. There are more things that are necessary, and even more things that are nice to have, which we will cover. This code only runs on the CPU.

The intention of this chapter is to offer an easy start, so you *do try* this code. We will gradually apply and discuss each improvement in the following chapters. Run this easy code on your own computer, and, why not, improve it in advance! The best way to learn is by experimenting and making mistakes.

[7] If it is not clear to you why this happens, read more in the section on composition of transformations.