Interactive Programming for Artificial Intelligence

Deep Learning for Programmers

Dragan Djuric

SAMPLE CHAPTER

An Interactive Tutorial with CUDA, OpenCL, DNNL, Java, and Clojure

https://dragan.rocks

aiprobook.com

DRAGAN DJURIC

DEEP LEARNING FOR PROGRAMMERS [SAMPLE 1.0.0

DRAGAN ROCKS

Please check other books from the *Interactive Programming for Artificial Intelligence* book series at `https://aiprobook.com`

This 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 of further versions, and a number of nice perks.

All proceeds go towards funding the author's work on the Uncomplicate software libraries: Please check out `https://uncomplicate.org` and `https://github.com/uncomplicate`.

PUBLISHED BY DRAGAN ROCKS

HTTPS://AIPROBOOK.COM

Current version, 30 Aug 2021

This book would not have been possible without your support. Thank you!

Eric Fode, Erich Oliphant, Chewxy, Stéphane Tavera, tom winn, Alan Caulkins, Joe Norton, Andrew Dean, Anthony Carnemolla, Bobby Calderwood, David Bernal, Emmanuel Auclair, Carlton Schuyler, Siu Kin San, Lukasz Hall, Chris Curtis, Nada Amin, Walton, Forrest Galloway, Eric Blood, John Collins, Christopher Genovese, Matthew Chadwick, Boris, Hadil Sabbagh, Oleg Popov, Thomas Wengerek, R Lassoued, Eric Somdahl, DANIEL Fredric CARPENTER, Steven Collins, Saransh Mohapatra, ka yu Lai, Rune Brinckmeyer, Eric Ihli, Max Triano, and many others (list continues at the back of the book).

I would also like to thank Clojurists Together and Cognitect for supporting my work on the Uncomplicate libraries with generous grants.

Contents

Getting started

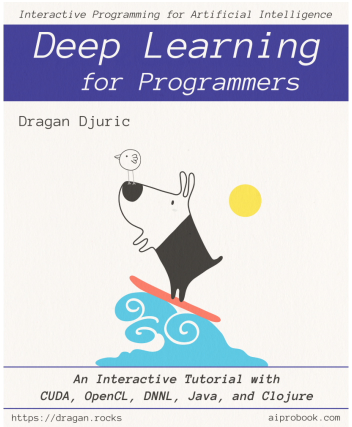
Introduction

Dear reader, the text you hold in your hands[1] is the first - imperfect, but useful, I hope - version of the only book in this field written with programmers in mind. As any good software, it's developed continuously. I plan to keep an endless stream of improvements, with milestones released as full versions 2, 3, etc. Hence, this is not the first edition, it's version one-point-zero.

What?

The full title of the book is "Deep Learning for Programmers: An Interactive Tutorial with CUDA, OpenCL, DNNL, Java, and Clojure" (Figure 1). Let's elaborate this.

The main topic is deep learning (DL), a subset of machine learning (ML) methods mainly based on neural networks (NN). At this moment, DL, very popular and successful in practice, is followed by countless books and articles. Alas, most of this literature is either oriented towards PhD researches, or less-technical users. There was a gaping hole where teaching how to *implement* this kind of software should have been, which I hope I'm helping cover - at least partially - with this book. Therefore, this book is *for programmers*, rather than ML researchers, mathematicians, cancer researchers, or business analysts.



Interactive Programming for Artificial Intelligence

Deep Learning
for Programmers

Dragan Djuric

An Interactive Tutorial with
CUDA, OpenCL, DNNL, Java, and Clojure

https://dragan.rocks aiprobook.com

Figure 1: The DLFP book

Every single line of code in this book is written in Clojure[2], a modern dialect of Lisp that compiles to Java bytecode and runs on Java Virtual Machine (JVM). I assume that you're already proficient in Clojure, but if you are not, the basics are fairly easy to learn[3].

Since our programs will run on a broad choice of exotic hardware, more notably GPUs, we'll have to work with software platforms such as OpenCL, or Nvidia's CUDA, in addition to the Java platform. These platforms are notoriously tricky to learn and use. Fortunately, I managed to hide them under the hood. You'll only need to learn to control them through Clojure, in a similar way you control the JVM. The only programming language we'll use is Clojure, and, yet, we get the full speed and power of these platforms.

In short, you're going to learn:

[1] More probably, on your screen.

[2] `https://clojure.org`

[3] There is a free online book "Clojure for the Brave and True" at `https://brave-clojure.com`, and if you prefer a deeper text, `https://clojure.org/books` is a good place to browse.

- the principles behind DL (and ML)
- only the necessary math and theory
- how to translate ML theory to code, step-by-step
- how to apply these techniques to implement NN from scratch
- how to implement other algorithms using vectors and matrices
- how to encapsulate these with elegant interfaces
- how to integrate with high-performance vendor libraries
- the nuts and bolts of a tensor-based deep learning library
- how to write simple, yet fast, number crunching software
- generally applicable high-performance programming techniques

Rather than being constrained to deep learning only, I hope that it opens a real-world portal to *high-performance scientific computing*.

How?

The title contains the phrase *An Interactive Tutorial*. Rather than throwing a pile of in-depth theory and specification at you, it starts from the simplest of the simple Hello World programs, which does almost nothing, but still does something. Then, in each chapter, we gently add one or two concepts, growing the program to do more. This book is not even trying to be a reference, it aims to be a great tutorial. I wrote it to be read in order, since each chapter builds on what has been learned earlier.

Thanks to Clojure and its REPL[4], this tutorial is *interactive*: as soon as I show you a line of code, you can evaluate it, and see the result *immediately*, even when the program is far from complete. There are no restarts, debug statements, `println`, nor carpets of boilerplate code.

You start the Clojure's JVM *once*, and work on the pieces of the program *while it is running all the time*, including the code that runs on *GPU*. If you're already familiar with what I'm talking about, great; but if that's not the case, I strongly advise that you keep your mind open and look around for demonstrations of Clojure's REPL[4] oriented programming. I recommend using Emacs + CIDER, but there are other tools that you might prefer. I don't care, as long as you are using the REPL-oriented process, rather than emulating interactive Python console[5].

[4] Short from Read Evaluate Print Loop.

[5] You might expect that Clojure REPL workflow is something like you've seen in Python, Ruby, and JavaScript ecosystems, but it's *not*; don't skip this.

Why?

Why bother with this? Why did I write this, despite millions of pages on DL that had already been available, and why should you read this, instead of choosing from thousands upon thousands of papers and hundreds of books on ML?

Why use Clojure, and why use Neanderthal and Deep Diamond? There is TensorFlow by Google, there is PyTorch by Facebook, there is MxNet by Amazon. Why not just learn these, and use the abundance of features they offer?

Google, Facebook, and Amazon certainly know what they're doing. But, they are solving different problems than the ones I have. If I was in competition to out-google Google, I would probably need to use (a better) TensorFlow. Most of TensorFlow's features might be a great fit for Google, but are bloated for the challenges that smaller companies have. What is a feature for these giants is often a shackle around little guy's ankles.

Moreover, you might've noticed that learning how to tame these beasts is not as easy as advertised. Tings you learn here will help you understand how these big frameworks work and how to use them optimally, even if TensorFlow and MxNet are the right tool for you. So, I guess that there's no downside...

Software is abundant, as well as superficial tutorials. The challenge is that the literature that would teach you foundations was not very good.

There are roughly two groups of texts:

The first is written by researchers, where their authors try to show off by discussing every research topic they'd ever been in contact with, rather than write tutorials that try to teach the area. After one of those, you probably won't be any wiser about how to construct even a hello world program.

There are many practical texts, that teach you how to classify images, generate voice, or produce a whole fake video, but are superficial on how that actually works under the hood. These are more recent and can be very useful once you know these "under the hood" things. But how to get to that point?

Rather than giving you a Big Mac Lunch Box, this book shows how to catch a fish, start a fire, and cook a moderate, yet healthy and delicious meal![6]

When?

Wherever you are in your Clojure journey, you can start right now! In that regard, the book is self-sufficient. We start from very basic things without assuming much about your math background, other

[6] Healthy, yet delicious? Impossible! Or, is it?

than high school or first year university linear algebra and calculus.[7] Even these are introduced through examples and without digression, as much as possible.

I hope that, as a programmer, you like the iterative and incremental approach to software development, because this is exactly how we'll work in this book. We start from one tiny lump of snow which we will roll and roll downhill many times until we build a huge avalanche. You'll know every single snowflake as your own pocket.

To stay true to these words, I won't write any introduction prose about the generalities of machine learning, deep learning, neural networks, their history, and whatever DL books spend the first 100 pages on. If you need such introduction, please check out the Deep Learning book (TDLB) by Ian Goodfellow et al.[8] This rare gem in the sea of plastic waste is a good companion book to this book. Whenever you find my explanations too laconic, or you need background information, look there first; I've purposefully made my presentation compatible with TDLB. Another good companion, especially to the first dozen chapters, is Michael Nielsen's Neural Networks and Deep Learning[9]. Its code is much more toy-ish than ours[10], and I doubt you'll need it often, but it might be a good additional resource for beginners.

The Interactive Programming for Artificial Intelligence series

So, you can comfortably use this book on its own and supplement it with TDLB for detailed theoretical background when needed. Whatever linear algebra and calculus is required to understand what we're doing, I explain and demonstrate on the go. Moreover, this book is a primer on how to write software based on linear algebra.

Numerical Linear Algebra for Programmers

However, a decent percentage of Clojure (and Java, and Python, and C++) programmers forgot whatever linear algebra they had learned in school. My on-the-go explanations might be good enough that you understand how the thing that we implemented works, but you might be still intrigued to more deeply understand *why* it works that way, and *how to recognize* places where you can apply various techniques yourself. Again, I wrote this book such that it shows these things, but I had to constrain myself to the topic of deep learning.

The "Numerical Linear Algebra for Programmers: An Interactive Tutorial with GPU, CUDA, OpenCL, MKL, Java, and Clojure" book (Figure 2) is a from-zero-to-hero guide to linear algebra for programmers who have never learned linear algebra or who have forgot too much of what they once knew. It follows an engineering math book, and teaches

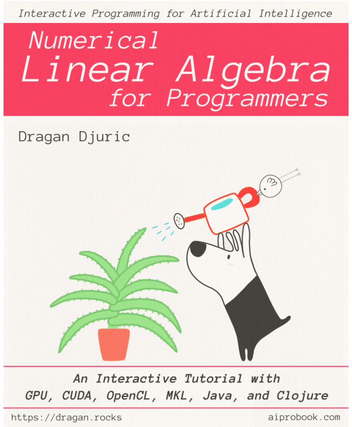

Interactive Programming for Artificial Intelligence
Numerical
Linear Algebra
for Programmers
Dragan Djuric
An Interactive Tutorial with
GPU, CUDA, OpenCL, MKL, Java, and Clojure
https://dragan.rocks
aiprobook.com

Figure 2: The LAFP book

[7] Take this with a grain of salt. I attended my high school and university in East Europe; curriculum in you part of the world might be more advanced (good) or lagging behind (then you might have to do some catching up).

[8] The book is freely available online at `https://www.deeplearningbook.org/`.

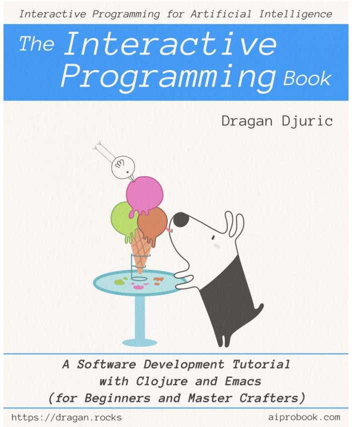
[9] Freely available at `http://neuralnetworksanddeeplearning.com/`

[10] It's Python, anyway. OTOH, it might be good as a baseline to compare to.

the same intuition and skills through Clojure code, with Neanderthal, of course. After teaching you the foundational linear algebra topics, it goes further and teaches how to apply them to typical programming tasks. You can check the contents of the LAFP book, available at `https://aiprobook.com/numerical-linear-algebra-for-programmers/` and see whether it could be a good fit for you (there's a free sample).

The Interactive Programming Book

This is the book that teaches the programming tools and the software development process. Although it is not a machine learning book, it teaches fundamental skills necessary for effective application of what the other books teach. It also complements the books on Clojure, which teach you the language itself, but not how to hold the pencil and how to press it to the paper. Figure 3 shows the cover page that I have already designed, but I'm yet to write it. Please check my blog `https://dragan.rocks` for news and updates. Once the first drafts are ready, they'll be available at `https://aiprobook.com`.



Interactive Programming for Artificial Intelligence

The Interactive Programming Book

Dragan Djuric

A Software Development Tutorial
with Clojure and Emacs
(for Beginners and Master Crafters)

https://dragan.rocks aiprobook.com

Figure 3: The Interactive Programming Book (TIPB)

Other books!

And that is not all! Two books (DLFP and LAFP) are already here, at version 1.0, one (TIPB) is in my mind, about to be started, but I plan to write at least a few more. One book will teach GPU programming that will cover CUDA and OpenCL. In DLFP we only use GPU, there we will learn how to write custom GPU programs. I expect that to be fairly thin, 150-200 pages.

Yet another will be a from-basics-to-GPU tutorial about probability and probabilistic data analysis. That's a tricky subject, so it will take some time, and may grow to be even thicker than DLFP. I hope that I'll be able to interest enough readers and subscribers so these two books can see the light of day, too.

So, that's 5 books in total planned so far. I'm looking forward to see Clojure with such a strong covering of these tricky topics!

Beyond that, it's too early to say, but who knows. . .

Let's go

So, check out the Appendix if you need to set up the libraries that we use in this book, take a look at the TOC to get a feel of what we're about to cover, and enjoy reading the first chapter without further delay.

Please do not hesitate to share any thoughts publicly online (and help spreading the word). I'm interested in both what's good and what should be improved. I'm learning from you, too!

Inference

Representing layers and connections

Our journey of building a deep learning library that runs on both CPU[11] and GPU[12] begins. We start from a clean slate, with only a matrix library, Neanderthal, in our toolbox.

Neural networks structure

Figure 4 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 could be the pixels of an image, and the output represents "probabilities" of this image belonging to a class, such as *cat* (y_1) or *dog* (y_2).

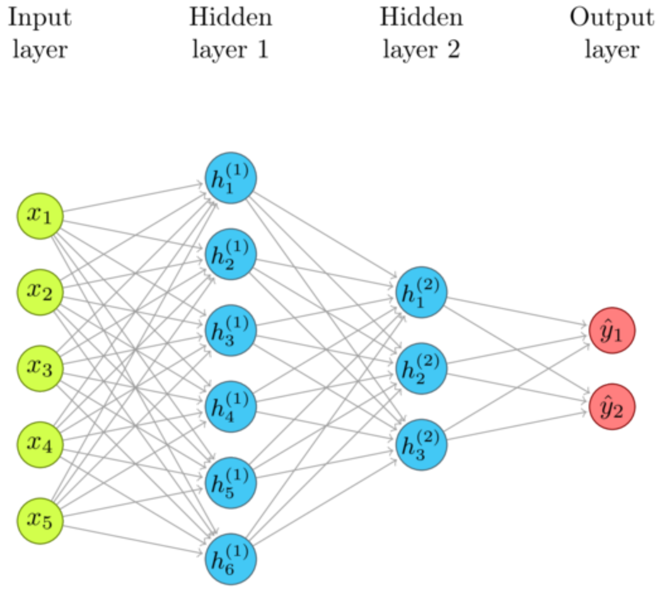


Figure 4: A typical neural network.

Neural Networks often classify complex things such as objects in photographs, translate text, or "predict" future data. Mechanically, though, there is no magic; they just approximate functions. What the

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

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

inputs and outputs are is not particularly important at the moment. The network can approximate (or, to be fancy, predict) even mundane functions such as, for example, $y = sin(x_1) + cos(x_2)$.

A network of the hello world level of simplicity is shown in Figure 5.

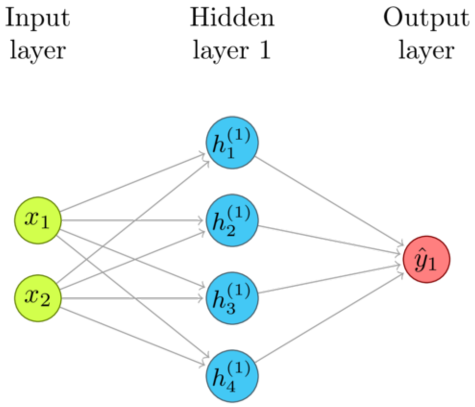


Figure 5: A simple network with two neurons in the input, one neuron in the output, and four neurons in the single hidden layer.

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 to 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 nodes has exactly the same structure and logic.

The only moving parts are the numerical values in weights. We can exploit that, and create efficient implementations that fit well into hardware optimizations for numerical computations.

It seems that the human brain analogy is more a product of marketing than a technical necessity. A simple neural network layer implements 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 6.

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, $h_2^{(1)}$.

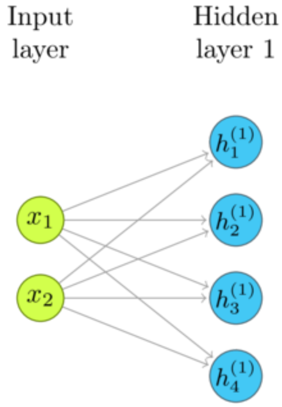


Figure 6: Connections between two layers

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

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

These are technically four dot products[13] between the corresponding rows of the weight matrix and the input vector.

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

[13] Read more about basic vector operations in the chapter on Vector Spaces (LAFP book).

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![14]

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

For example, for some specific input and weights[15] the network shown in Figure 6 computes in the following way:

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

The hidden layer then propagates the signal in the same manner (Figure 7).

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

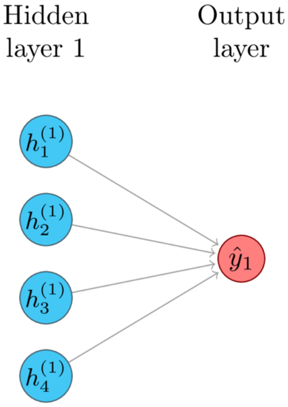


Figure 7: The second transformation.

The code

To try this in Clojure, we require a few 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 mv!]]]))
```

The minimal code example, following the Yagni principle [16], would be something like this:

```
(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}))
```

[14] Read more in the chapter on Matrix Transformations (LAFP book).

[15]
$$h_1^{(1)} = 0.3 \times 0.3 + 0.6 \times 0.9 = 0.09 + 0.54 = 0.63$$
$$h_2^{(1)} = 0.1 \times 0.3 + 2 \times 0.9 = 0.03 + 1.8 = 1.83$$
$$h_3^{(1)} = 0.9 \times 0.3 + 3.7 \times 0.9 = 0.27 + 3.33 = 3.6$$
$$h_4^{(1)} = 0 \times 0.3 + 1 \times 0.9 = 0 + 0.9 = 0.9$$

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

```
(def h1 (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]
[   0.63    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 `x` by `w1`. `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 in 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]
[   0.63    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 8). We transform it by yet another weight matrix, `w2`.

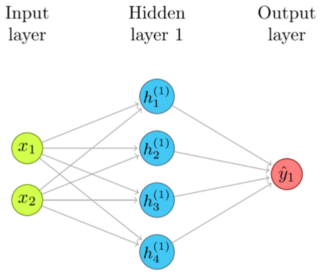


Figure 8: A simple network from Figure 5 repeated for convenience.

```
(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 magic 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.

Our toy is 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[17].

We have not implemented thresholds, or *biases*, yet. We have also left everything in independent matrices and vectors, without a structure involving *layers* that would hold them together. And we have not 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.

[17] If it is not clear to you why this happens, read more in the section on composition of transformations (LAFP book).

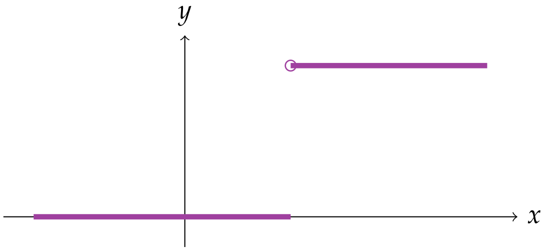
Bias and activation function

The current network implementation computes the same linear transformation as a one-layer network. This unwanted linearity of the whole structure can be avoided by adding bias and activation function at the output of each layer.

Threshold and Bias

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.[18]

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

[18] 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))
```

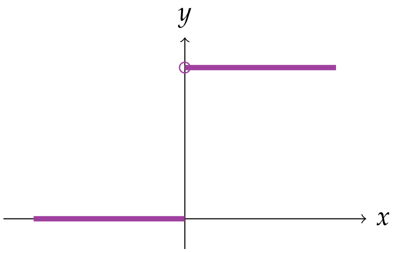
Since we do not care about additional object instances right now, it is more convenient to use the pure `mv` function instead of `mv!`. `mv` creates the resulting vector `y`, instead of mutating the one that has to be provided as an argument.

```
(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

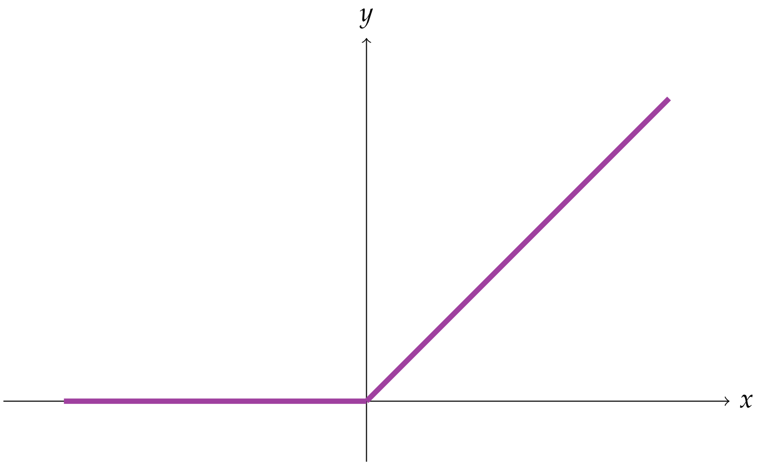
The decision capabilities supported by the step function are rather crude. A neuron either outputs a constant value (1), or zero. It is better to use functions that offer different levels of the signal strength. Instead of the step function, the output of each neuron passes through an *activation function*.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

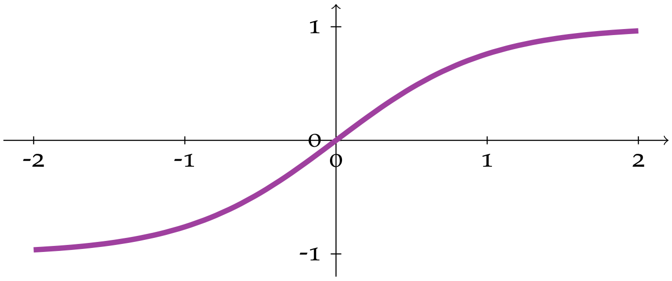
```
(relu! bias (mv w1 x))

=>
#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.

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



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.

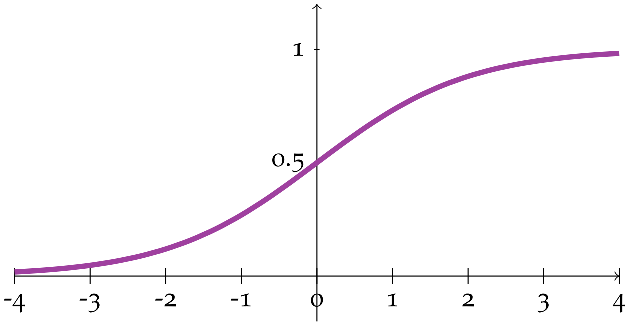
```
(tanh! (axpy! -1.0 bias (mv w1 x)))
```

```
=>
#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 9 is an updated version of the simple neural network with one hidden layer from the previous chapter, shown in Figure 10. 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.

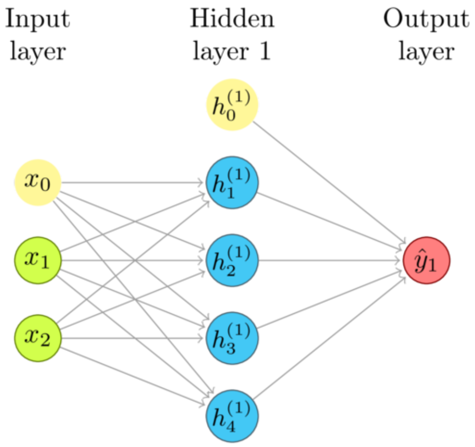


Figure 9: A simple network with bias shown as an additional node (yellow).

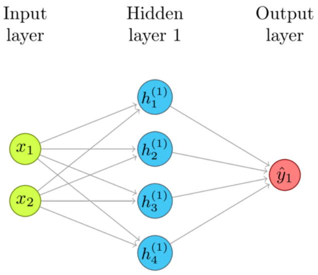


Figure 10: A simple network from the previous chapter repeated for convenience.

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.

Fully connected inference layers

It's time to consolidate the basic structure we have built so far into a layer type, that is easy to use as a stand-alone building block.

The updated network diagram

We'll update the neural network diagram from Figure 11, to reflect the recently included biases and activation functions and show it in Figure 12.

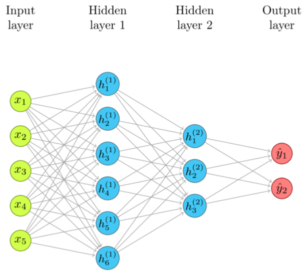


Figure 11: A *typical* network from an earlier chapter repeated for convenience.

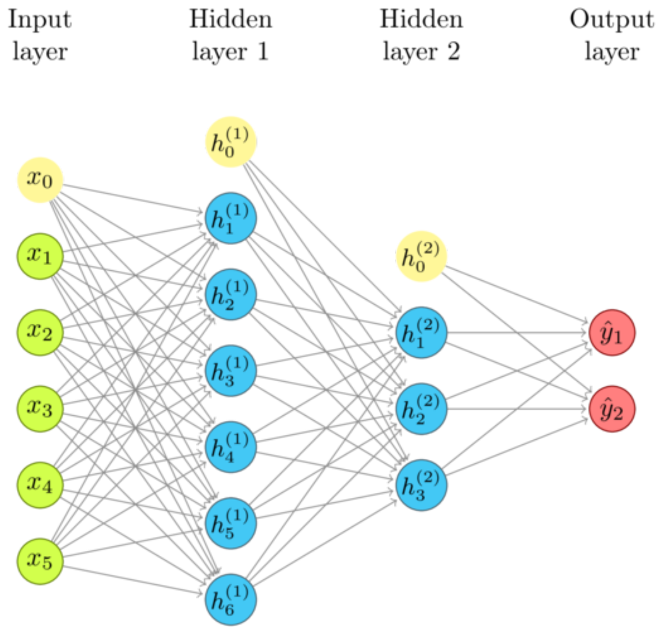


Figure 12: A typical neural network with bias shown as an additional node (yellow).

In each layer, bias is shown as an additional, *"zero"-indexed*, node, which, connected to the nodes in the next layer, gives the bias vector. Activation functions have not been shown to avoid clutter. Consider that each node has an activation at the output. If a layer did not have any activation, it would have been redundant.[19]

[19] Due to the composition of transformations.

The Layer type

We need a structure that keeps the account of the weights, biases and output of each layer. It should manage the life-cycle and release that memory space when appropriate. That structure can implement Clojure's `IFn` interface, so we can invoke it as any regular Clojure function.[20]

```
(import 'clojure.lang.IFn)
```

We will create `FullyConnectedInference` as a Clojure type, which is then compiled into a Java class. `FullyConnectedInference` implements the `release` method of the `Releaseable` protocol[21], and the `invoke` method of `IFn`[22]. We would also like to access and see the weights and bias values so we create new protocol, `Parameters`, with `weight` and `bias` methods.

```
(defprotocol Parameters
  (weights [this])
  (bias [this]))

(deftype FullyConnectedInference [w b h activ-fn]
  Releaseable
  (release [_]
    (release w)
    (release b)
    (release h))
  Parameters
  (weights [this] w)
  (bias [this] b)
  IFn
  (invoke [_ x]
    (activ-fn b (mv! w x h))))
```

Constructor function

At the time of creation, each `FullyConnectedInference` have to be supplied with an activation function, a matrix for weights, and the vectors for bias and output that have matching dimensions. Of course, we will automate that process with a function.

```
(defn fully-connected [activ-fn in-dim out-dim]
  (let-release [w (dge out-dim in-dim)
                bias (dv out-dim)
                h (dv out-dim)]
    (->FullyConnectedInference w bias h activ-fn)))
```

[20] As in the previous chapter, we leave out the namespace declaration. Refer to the source code for that detail.

[21] A widely used protocol from the `uncomplicate/commons` library, which serves as a hook for `with-release`.

[22] This is a standard Clojure mechanism.

Now a simple call to the `fully-connected` function that specifies the activation function and the dimensions of the input dimension (the number of neurons in the previous layer) and output dimension (the number of neurons in this layer) will create and properly initialize it.

`let-release` is a variant of `let` that releases its bindings (`w`, `bias`, and/or `h`) if anything goes wrong and an exception gets thrown in its scope. `with-release`, on the other hand, releases the bindings in all cases.

Activation functions

We can use a few matching functions that we discussed in the previous chapter.

```
(defn activ-sigmoid! [bias x]
  (axpy! -1.0 bias x)
  (linear-frac! 0.5 (tanh! (scal! 0.5 x)) 0.5))

(defn activ-tanh! [bias x]
  (tanh! (axpy! -1.0 bias x))  )
```

Using the fully-connected function

We are ready to re-create the existing example in a more convenient form. Note that we no longer have to worry about creating the matching structures of a layer; it happens automatically when we create each layer. I use the `transfer!` function to set up the values of weights and bias, but typically these values will already be there after the training (learning) of the network. We are using the same "random" numbers as before as a temporary testing crutch.

```
(with-release [x (dv 0.3 0.9)
               layer-1 (fully-connected activ-sigmoid! 2 4)]
  (transfer! [0.3 0.1 0.9 0.0 0.6 2.0 3.7 1.0] (weights layer-1))
  (transfer! (dv 0.7 0.2 1.1 2) (bias layer-1))
  (println (layer-1 x)))
```

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

The output is the same as before, as we expected.

Multiple hidden layers

Note that the layer is a *transfer function* that, given the input, computes and returns the output. As with other Clojure functions, the output value can be an input to the next layer function.

```
(with-release [x (dv 0.3 0.9)
               layer-1 (fully-connected activ-tanh! 2 4)
               layer-2 (fully-connected activ-sigmoid! 4 1)]
  (transfer! [0.3 0.1 0.9 0.0 0.6 2.0 3.7 1.0] (weights layer-1))
  (transfer! [0.7 0.2 1.1 2] (bias layer-1))
  (transfer! [0.75 0.15 0.22 0.33] (weights layer-2))
  (transfer! [0.3] (bias layer-2))
  (println (layer-2 (layer-1 x))))

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

If we do not count the `transfer!` calls, we have 3 lines of code that describe the whole network: two lines to create the layers, and one line for the nested call of layers as functions. It is already concise, and we will still improve it in the following chapters.

Micro benchmark

With rather small networks consisting of a few layers with few neurons each, any implementation will be fast. Let's create a network with a larger number of neurons, to get a feel of how fast we can expect these things to run.

Here is a network with the input dimension of 10000. The exact numbers are not important, since the example is superficial, but you can imagine that 10000 represents an image that has 400 × 250 pixels. We have put 5000 neurons in the first layer, 1000 in the second layer, and 10 at the output. Just some random large-ish dimensions. Imagine ten categories at the output.[23]

```
(with-release [x (dv 10000)
               layer-1 (fully-connected activ-tanh! 10000 5000)
               layer-2 (fully-connected activ-sigmoid! 5000 1000)
               layer-3 (fully-connected activ-sigmoid! 1000 10)]

  (quick-bench (layer-3 (layer-2 (layer-1 x)))))
```

Criterium's report looks like this.

[23] This code uses the `criterium` library for benchmarking.

```
Evaluation count : 36 in 6 samples of 6 calls.
             Execution time mean : 18.320566 ms
    Execution time std-deviation : 1.108914 ms
   Execution time lower quantile : 17.117256 ms ( 2.5%)
   Execution time upper quantile : 19.847416 ms (97.5%)
                   Overhead used : 7.384194 ns
```

On my old CPU (i7-4790k), one inference through this network takes 18 milliseconds. If you have another neural networks library at hand, you could construct the same network (virtually any NN software should support this basic structure) and compare the timings.

So far so good

Finishing this task in 18 ms is not slow. But what if we had lots of data to process? Does that mean that if I had 10 thousand inputs, I'd need to make a loop that invokes this inference 10 thousand times, and wait 3 minutes for the results?

Is the way to speed that up putting it on the GPU? We will do that soon, but there is something that we can do even on the CPU to improve the performance!

Increasing performance with batch processing

Sharing memory

GPU computing with CUDA and OpenCL

This chapter demonstrates a network running on the GPU, at last. We generalize the existing code so it can run on any supported GPU device: on an Nvidia GPU with CUDA, and on an AMD GPU with OpenCL. Our code will be so platform-agnostic that we will even be able to mix CUDA and OpenCL[24].

The inference layer type

We're starting from the existing layer type, and trimmed down to keep just the batch version. All functions that we have used, `axpy`, `mm`, etc., are polymorphic and general in respect to the device they execute on: CPU, Nvidia GPU, AMD GPU, or Intel GPU. The implementations of activation functions that we have used are general, too.

The dispatch to the right implementation is being done by the type of the vector or matrix structure at hand. The constructor function that we have used is hard-coded for using `double` floating point numbers, to exist in main memory, and use the native CPU backend[25].

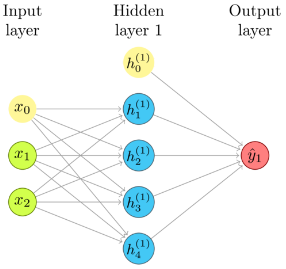


Figure 13: A neural network.

```
(defn fully-connected [activ-fn in-dim out-dim]
  (let-release [w (dge out-dim in-dim)
                bias (dv out-dim)]
    (->FullyConnectedInference w bias activ-fn)))
```

Generalize the code

There is only one thing that we have to do to make this code completely general: use general constructors from the `core` namespace, instead of the convenience methods from the `native` namespace. These methods are, in this case, `ge` (general matrix) instead of `dge` (double general native matrix), and `vctr` instead of `dv` (double native vector). The only difference in these methods is that they require an engine-specific factory as their first argument.

We accommodate the `fully-connected` constructor to accept the factory that determines the implementation technology as an input.

[24] Keep in mind that we do this only for fun, and that it is a bad idea for production code.

[25] `dge` and `dv` constructors.

```
(defn fully-connected [factory activ-fn in-dim out-dim]
  (let-release [w (ge factory out-dim in-dim)
                bias (vctr factory out-dim)]
    (->FullyConnectedInference w bias activ-fn)))
```

Now, we repeat the example of running the network with `native-double`. That is the same factory that is used by the `dge` and `dv` methods, available in the `native` namespace. We can use `native-float` in its place, to use single-precision floating point computations on the CPU, some of the GPU factories, configure another factory coded by a 3-rd party, or even use the same code provided by Neanderthal, but configured in a different way.

```
(with-release [x (ge native-double 2 2 [0.3 0.9 0.3 0.9])
               ones (vctr native-double 1 1)
               layer-1 (fully-connected native-double tanh! 2 4)
               a-1 (ge native-double 4 2)
               layer-2 (fully-connected native-double sigmoid! 4 1)
               a-2 (ge native-double 1 2)]
  (transfer! [0.3 0.1 0.9 0.0 0.6 2.0 3.7 1.0] (weights layer-1))
  (transfer! [0.7 0.2 1.1 2] (bias layer-1))
  (transfer! [0.75 0.15 0.22 0.33] (weights layer-2))
  (transfer! [0.3] (bias layer-2))
  (transfer (layer-2 (layer-1 x ones a-1) ones a-2)))
```

```
=>
#RealGEMatrix[double, mxn:1x2, layout:column, offset:0]
┌       ↓       ↓       ┐
→       0.44    0.44
└                       ┘
```

The display of the result in this example has been modified a little. Instead of doing a `println` as in the previous chapters, we transfer the resulting matrix to main memory. As `println` is typically not used in production code, `transfer` is more general and platform-agnostic way to ensure that the output values survive the `with-release` scope.

Don't forget that, in this example, we have used `with-release` for *all* bindings, even the output `a-2`. I do this because the code should support CPU *and GPU*. In the main memory releasing the data is of great help, but it is optional in a REPL session, since the memory eventually gets released by the JVM[26]. On the GPU, however, JVM can not do anything. The underlying GPU buffer that is not released explicitly, is not released at all until we release the whole context. Therefore, get in the habit of always taking care of the lifecycle and release all vectors, matrices and other structures as soon as possible.[27]

[26] With a few caveats since JVM might not do it as soon as we may hope.

[27] In most cases it is best to do this automatically by `with-release` or `let-release`. In more advanced cases, we can directly call `release`.

During interactive sessions, we would like to see the result in the REPL. But, how, if the data stored in the result that is being returned (`a-2`) is released just the moment before it is about to be printed. Here, the `transfer` method transfers the data from wherever it is (main memory or GPU memory) to the equivalent object in the main memory.

This particular network

We are going to run this code on different devices, so wrapping it into a function might be a good idea. Note that we provide `factory` as the argument, and everything else is general and the same for all platforms!

```
(defn this-particular-network [factory]
  (with-release [x (ge factory 2 2 [0.3 0.9 0.3 0.9])
                 ones (vctr factory 1 1)
                 layer-1 (fully-connected factory tanh! 2 4)
                 a-1 (ge factory 4 2)
                 layer-2 (fully-connected factory sigmoid! 4 1)
                 a-2 (ge factory 1 2)]
    (transfer! [0.3 0.1 0.9 0.0 0.6 2.0 3.7 1.0] (weights layer-1))
    (transfer! [0.7 0.2 1.1 2] (bias layer-1))
    (transfer! [0.75 0.15 0.22 0.33] (weights layer-2))
    (transfer! [0.3] (bias layer-2))
    (transfer (layer-2 (layer-1 x ones a-1) ones a-2))))
```

We can call this function and instruct it to use double-precision floating point computation on the CPU.

```
(this-particular-network native-double)
```

```
=>
#RealGEMatrix[double, mxn:1x2, layout:column, offset:0]
┌       ↓       ↓       ┐
→       0.44    0.44
└                       ┘
```

It can use single-precision floating point computation, still on the CPU.[28]

```
(this-particular-network native-float)
```

```
=>
#RealGEMatrix[float, mxn:1x2, layout:column, offset:0]
┌       ↓       ↓       ┐
→       0.44    0.44
└                       ┘
```

[28] If you are struggling to find the difference in these two printouts, note that the first prints "double", while the second prints "float".

CUDA on an Nvidia GPU

The same code, without changes, runs on the GPU! The only thing that it needs, is the factory that sets it up with appropriate engines.

For engines based on Nvidia's CUDA[29] platform, we use functions from `uncomplicate.clojurecuda.core`[30] namespace to choose and set up the GPU itself. We may have more than one graphics accelerator in our system, and Neanderthal has to know which one to use. `with-default` is a method that will choose the best device that you have, and set it up automatically. There are more fine grained methods in the ClojureCUDA[31] library if you need more control.

```
(require
 '[uncomplicate.clojurecuda.core :as cuda
   :refer [current-context default-stream synchronize!]])
```

Next, we use the `cuda-float` constructor to create a factory whose engines will use single-precision floating point computations in the default context and stream provided by ClojureCUDA. We may need more than one factory for advanced computations.

```
(require '[uncomplicate.neanderthal.cuda :refer [cuda-float]])

(cuda/with-default
  (with-release [cuda-factory (cuda-float (current-context) default-stream)]
    (this-particular-network cuda-factory)))

=>
#RealGEMatrix[float, mxn:1x2, layout:column, offset:0]
┌       ↓       ↓       ┐
→        0.44    0.44
└                       ┘
```

OpenCL on an AMD GPU

In case you have an AMD or Intel GPU, you will not be able to work with CUDA platform. For more universal coverage Neanderthal supports OpenCL[32] which is an open platform equivalent to CUDA, that supports all major hardware vendors: AMD, Intel, and even Nvidia.

Instead of ClojureCUDA, you'll use ClojureCL[33] to set up the execution environment. Other than a few differences in terminology, most of the knowledge of parallel computing on the GPU is transferable between CUDA and OpenCL.

```
(require
 '[uncomplicate.neanderthal.opencl :refer [opencl-float]]
```

[29] `https://developer.nvidia.com/cuda-toolkit`

[30] We are showing the namespace requires in the code below, but will skip that in the following chapters.

[31] `https://clojurecuda.uncomplicate.org`

[32] `https://www.khronos.org/opencl/`

[33] `https://clojurecl.uncomplicate.org`

```
 '[uncomplicate.clojurecl.core :as opencl
   :refer [*context* *command-queue* finish!]])

(opencl/with-default
  (with-release [opencl-factory (opencl-float *context* *command-queue*)]
    (this-particular-network opencl-factory)))

=>
#RealGEMatrix[float, mxn:1x2, layout:column, offset:0]
┌       ↓       ↓       ┐
→       0.44    0.44
└                       ┘
```

We can even mix CUDA and OpenCL

With Neanderthal, we can even combine code that partly runs on an Nvidia GPU and partly on an AMD GPU. For performance reasons, it is unlikely that we would do this often in "real" code. However, we can do it for fun and learning. This example has been included mainly to demonstrate how flexible Neanderthal and Clojure are. This is something that we would struggle to do in competing platforms!

```
(opencl/with-default
  (cuda/with-default
      (with-release [opencl-factory (opencl-float *context* *command-queue*)
                     cuda-factory (cuda-float (current-context) default-stream)
                     x (ge opencl-factory 2 2 [0.3 0.9 0.3 0.9])
                     ones-opencl (vctr opencl-factory 1 1)
                     layer-1 (fully-connected opencl-factory tanh! 2 4)
                     a-1 (ge opencl-factory 4 2)
                     a-1-cuda (ge cuda-factory 4 2)
                     ones-cuda (vctr cuda-factory 1 1)
                     layer-2 (fully-connected cuda-factory sigmoid! 4 1)
                     a-2 (ge cuda-factory 1 2)]
        (transfer! [0.3 0.1 0.9 0.0 0.6 2.0 3.7 1.0] (weights layer-1))
        (transfer! [0.7 0.2 1.1 2] (bias layer-1))
        (transfer! [0.75 0.15 0.22 0.33] (weights layer-2))
        (transfer! [0.3] (bias layer-2))
        (layer-1 x ones-opencl a-1)
        (transfer! a-1 a-1-cuda)
        (transfer (layer-2 a-1-cuda ones-cuda a-2)))))

=>
#RealGEMatrix[float, mxn:1x2, layout:column, offset:0]
```

```
⌜        ↓       ↓       ⌝
→        0.44    0.44
⌞                        ⌟
```

Micro benchmark

One aspect of GPU computing is *how to do it at all*. With Neanderthal, ClojureCL and ClojureCUDA it is not that hard. Another question is: *is it worth the trouble*?

Nvidia GTX 1080Ti

We are going to measure the same superficial example that we used in previous chapters. The heavily-optimized native CPU engine backed by Intel's MKL[34] computed one pass in 6 seconds. We hope that Nvidia's GeForce GTX 1080Ti[35], at 11 TFLOPS[36], will be able to do it much faster.

Please note the `(synchronize!)`[37] call. GPU calls are asynchronous[38], and here we are making sure that we block the main thread and wait for the computation to complete before we declare victory.

```
(cuda/with-default
  (with-release [factory (cuda-float (current-context)
                                     default-stream)]
    (with-release [x (ge factory 10000 10000)
                   ones (entry! (vctr factory 10000) 1)
                   layer-1 (fully-connected factory tanh! 10000 5000)
                   a1 (ge factory 5000 10000)
                   layer-2 (fully-connected factory sigmoid! 5000 1000)
                   a2 (ge factory 1000 10000)
                   layer-3 (fully-connected factory sigmoid! 1000 10)
                   a3 (ge factory 10 10000)]
      (time
       (do
         (layer-3 (layer-2 (layer-1 x ones a1) ones a2) ones a3)
         (synchronize!))))))
```

```
"Elapsed time: 122.529925 msecs"
```

122 milliseconds. It is well worth the trouble! This is roughly *50 times faster* than the optimized engine on my CPU![39]

AMD R9 290X

The computer that this book was written on also hosts an AMD GPU, R9 290X,[40] which has the theoretical peak performance of 5 TFLOPS.

[34] The fastest CPU thing around. See more at `https://software.intel.com/en-us/mkl`

[35] A Nvidia's flagship gaming GPU from 2017.

[36] Tera (10^12) Floating Point Operations (per second).

[37] This function is part of ClojureCUDA.

[38] Read more about asynchronous nature of GPU programming in another book in this series, Interactive GPU Programming with CUDA.

[39] Note that this depends on both the example and the hardware.

[40] An older AMD's flagship gaming GPU from 2013.

```
(opencl/with-default
  (with-release [factory (opencl-float *context* *command-queue*)]
    (with-release [x (ge factory 10000 10000)
                   ones (entry! (vctr factory 10000) 1)
                   layer-1 (fully-connected factory tanh! 10000 5000)
                   a1 (ge factory 5000 10000)
                   layer-2 (fully-connected factory sigmoid! 5000 1000)
                   a2 (ge factory 1000 10000)
                   layer-3 (fully-connected factory sigmoid! 1000 10)
                   a3 (ge factory 10 10000)]
      ;; The first time a BLAS operation is used in OpenCL
      ;; might incur initialization cost. Warm up the engine.
      (layer-1 x ones a1)
      (finish!)
      (time
       (do
         (layer-3 (layer-2 (layer-1 x ones a1) ones a2) ones a3)
         (finish!))))))
```

```
"Elapsed time: 330.683851 msecs"
```

This is roughly 3 times slower than Nvidia. It is still worth the effort, since it is *almost 20 times faster* than the CPU.

Someone may be disappointed by this not being close enough to GTX 1080Ti's speed, since R9 290X should be twice as slow by the specifications (5 TFOPS vs 11 TFLOPS). Instead of Nvidia's proprietary BLAS matrix routines, Neanderthal uses an open-source engine in its OpenCL backend. Although it's not *that* much behind, it can not match Nvidia's hardware optimizations at the same level. When we have an Nvidia's GPU, we can still use ClojureCL but if we need maximum performance, we should use ClojureCUDA.

A laconic performance comparison

Let us sum this up. In this example, we managed to accelerate a top-performance CPU code by *20 times* with an old AMD GPU, and *50 times* with a fairly recent, but not the best, Nvidia GPU, keeping the same code!

Learning

Gradient descent and backpropagation

The forward pass

The activation and its derivative

The backward pass

A simple neural networks API

Inference API

Training API

Initializing weights

Regression: learning a known function

By now, we have created the infrastructure that supports the basic task of supervised learning of something. It is not particularly sophisticated yet, and there are lots of techniques that we must support before it becomes useful in real-world scenarios. However, what we have is complete enough and simple enough that it offers a great opportunity for simple demonstration aimed at gradual understanding.

Neural networks approximate functions

When we step over the hype, what neural networks do is they approximate functions. Neural networks learn to do that based on a lot of examples taken out from the function's output. In real use cases neural networks learn to approximate functions that are black boxes.

The key ability of neural networks is that they approximate *unknown* functions. When we know a set of rules that transfer inputs to outputs, we can think of many ways of implementing that in programming languages, and all these ways are more efficient than using neural networks. When we don't know the process that transfers the inputs to the outputs, we cannot program it explicitly. If the process is known, but the rules are numerous, and it is not feasible to elicit them, that could be hard to implement, too.

A typical example familiar to programmers would be an expert system. Expert systems were a promising area of AI several decades ago. The idea is to find human experts for a certain area, help them define the rules she is using when making some decisions, program there rules in fancy DSLs with if/then/else flavor, and profit. It turns out that expert's time is expensive. Also, experts use lots of intuition; some rules work, but not always, with lots of 'however's. On top of that, due to the probabilistic nature of life, we cannot always rely even on the rules that we can define.

Let's say that we would like to analyze traffic of a website to defend against malicious visitors. We consulted with an expert, and he told us most of the known ways of detecting these. We implement some filters. If the user is from this range of IP addresses, if he uses a web browser with a certain user agent, if he comes via a proxy, if... Lots of rules are possible, and they would filter a lot of unwanted traffic. They would also filter some wanted traffic. But, most importantly, the attackers know these rules, too, and they can easily adapt their strategy.

The approach that neural networks take is implicit. If we feed past data to the network, and label good and bad visitors, *without saying why the good are good and the bad are bad* the network can figure out how to recognize them on its own. Even better, it can deal with the traffic that it has never seen. Of course, it may not do it perfectly, but it can learn this sort of stuff well enough.

To summarize, when we have lots of input/output examples, we can train a neural network to approximate the *unknown* function that produced these examples.

The example that we use in this chapter is something less ambitious than website traffic filtering. We are going to train a neural network on a *known* function. It is obvious that neural networks are not the right tool for that job, since it is much easier and precise to code the function in Clojure right away. It is a great first example, though, since it makes it possible to easily see what the network is doing.

Generating artificial data

Since we are simulating the data, we know the exact function that produces it. We use the function $f(\mathbf{x}) = sin(x_0) + cos(x_1) + tanh(x_2) + {x_3}^2$ as an easy example.

The implementation is straightforward. The function takes a Neanderthal vector as an input, and calculates a number according to the formula shown above.

```
(defn my-fn ^double [xs]
  (+ (math/sin (real/entry xs 0))
     (math/cos (real/entry xs 1))
     (math/tanh (real/entry xs 2))
     (math/sqr (real/entry xs 3))))
```

Here's the function in action. I give it a vector, an it returns the resulting number.

```
(my-fn (vctr native-float [0.3 0.1 0.9 0.33]))
```

```
=> 2.1157222504237048
```

Now we need lots of data that comes from this function. We would like if this data represented some domain well, so we are going to generate a lot of random vectors. We use the `rand-uniform` function to populate a matrix with 10000 examples. If you are wondering why we jumped from talking about vectors to populating a matrix, remember that the columns of a matrix are vectors. It is more efficient to work with a bunch of vectors tightly grouped in a matrix, than to keep them in a Clojure sequence.

```
(rand-uniform! (ge native-float 4 10000))

=>
#RealGEMatrix[float, mxn:4x10000, layout:column, offset:0]
┌       ↓       ↓       ↓       ↓       ↓       ┐
→       0.93    0.99    ⋰       0.94    0.01
→       0.97    0.07    ⋰       0.18    0.80
→       0.75    0.68    ⋰       0.43    0.99
→       0.36    0.05    ⋰       0.82    0.28
└                                               ┘
```

The columns of this matrix are the inputs that we are feeding to the function that generates the "fake" data.

```
(map my-fn (cols (rand-uniform! (ge native-float 4 10))))

=>
(1.966033196798584 1.6873884768278673 1.6556626674078674
 2.4881651136629275 1.9142285657995646 1.743036030649418
 2.266988861373788 1.5932202841070968 1.85334784193748
 2.139956338597604)
```

These are precise results calculated by the real, known, function `my-fn` with the data provided in the matrix that we had generated. The nice bonus of using a known function is that we can always generate more data, either for learning or for testing. Do not forget that in the real world, if the function is known, there is no need to learn its approximation from data.

Learning to approximate

In the previous example, we have created 10 observations of the function's output. There are data analysis methods that can learn something from such a small sample, but neural networks require more. We just assume that 10 thousand observations is enough in this case, but this is only an arbitrary chosen size. I hope this is going to be enough.

```
(def x-train (rand-uniform! (ge native-float 4 10000)))
x-train
```

```
=>
#RealGEMatrix[float, mxn:4x10000, layout:column, offset:0]
┌       ↓       ↓       ↓       ↓       ↓       ┐
→       0.06    0.01    ∵       0.35    0.80
→       0.24    0.10    ∵       0.90    0.09
→       0.40    0.74    ∵       0.55    0.32
→       0.33    0.55    ∵       0.48    0.36
└                                               ┘

(def y-train (ge native-float 1 10000 (map my-fn (cols x-train))))
y-train

#RealGEMatrix[float, mxn:1x10000, layout:column, offset:0]
┌       ↓       ↓       ↓       ↓       ↓       ┐
→       1.52    1.94    ∵       1.70    2.16
└                                               ┘
```

We have generated 10 thousand observations and we know the values of the function at these particular points, y-train. Then we create a neural network consisting of five fully connected layers.

The input layer is the data matrix. Since each input vector has four dimensions, the input layer will have four neurons. The hidden layers have 16, 64, and 8 neurons, respectively. The activation functions are sigmoid, tanh, and tanh, respectively. We have chosen these arbitrarily. Not only that this is not the optimal choice, but we do not even have a way to tell what the optimal architecture for any random function would be. That's one of the catches in neural networks, and many other AI methods. Since the network should approximate a real-valued function the output will be one-dimensional.

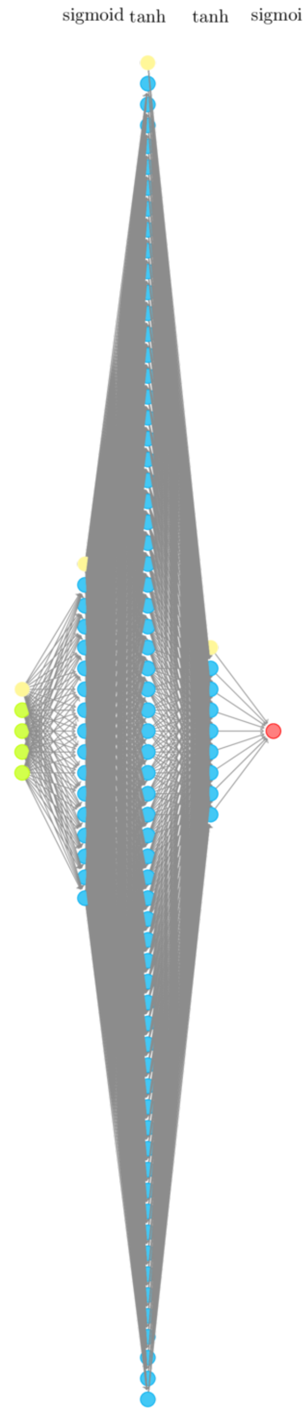


Figure 14: The 4-16-64-8-1 network (an unwieldy diagram).

```
(def inference (init! (inference-network
                       native-float 4
                       [(fully-connected 16 sigmoid)
                        (fully-connected 64 tanh)
                        (fully-connected 8 tanh)
                        (fully-connected 1 sigmoid)])))

(def training (training-network inference x-train))
```

Let us check whether the network seems to be properly initialized before we run the learning algorithm.

```
(weights (first (layers inference)))

=>
#RealGEMatrix[float, mxn:16x4, layout:column, offset:0]
```

```
┌       ↓       ↓       ↓       ↓       ┐
→       -0.20   -0.09    0.25   -0.01
→       -0.30   -0.13   -0.10    0.14
→         ⋱       ⋱       ⋱       ⋱
→       -0.42    0.01    0.19    0.02
→       -0.44    0.20   -0.15    0.01
└                                       ┘
```

```
(weights (second (layers inference)))
```

```
=>
#RealGEMatrix[float, mxn:64x16, layout:column, offset:0]
┌       ↓       ↓       ↓       ↓       ↓       ┐
→       -0.01    0.06     ⋱      0.02    0.02
→       -0.00   -0.00     ⋱      0.07    0.11
→         ⋱       ⋱       ⋱       ⋱       ⋱
→        0.01   -0.00     ⋱      0.13   -0.10
→       -0.02    0.15     ⋱     -0.04   -0.01
└                                               ┘
```

```
(weights (last (layers inference)))
```

```
=>
#RealGEMatrix[float, mxn:1x8, layout:column, offset:0]
┌       ↓       ↓       ↓       ↓       ↓       ┐
→        0.06    0.25     ⋱     -0.05   -0.01
└                                               ┘
```

These matrices contain relatively small numbers around zero. That seems right.

```
(sgd training y-train quadratic-cost! [[1 0.05] [1000 0.03] [100 0.01]])
```

```
(1.1418545755467493 0.6781750911065494 0.6780074343557965)
```

We run the algorithm for 1101 epochs with a few different learning rates. These values were chosen arbitrarily; we had *not* known in advance whether they are a good, bad, or a mediocre choice. To see whether the track we are on is any good, we have to rely on the value of the cost function. What we can see from the result, the starting value was 1.14, and it decreased to 0.68. This is not necessarily great, but it is a good sign.

The cost itself tells us only that the algorithm goes in the right direction, we need a more direct way to see how well it works. What is more familiar to a programmer than a unit test? Since we know the real function, it is easy to compare its results with the results from the neural networks. We do not make it formal yet. We will just generate a few random observations that we will use *for testing only. It is extremely important that the network does not see these observations during training.*

```
(def x-test (rand-uniform! (ge native-float 4 5)))
```

```
(def y-test (ge native-float 1 5 (map my-fn (cols x-test))))
```

Let's see what the network says.

```
(inference x-test)
```

```
=>
#RealGEMatrix[float, mxn:1x5, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ↓       ↓       ┐
   →       1.00    1.00    1.00    1.00    1.00
   └                                               ┘
```

This does not look very useful. The network should have returned something close to `y-test` values.

```
y-test
```

```
=>
#RealGEMatrix[float, mxn:1x5, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ↓       ↓       ┐
   →       2.49    2.08    1.32    2.34    2.93
   └                                               ┘
```

Regression requires linear output

The network returned a vector of ones because the output activation is the sigmoid function. Since the expected values are larger than 1, the output is saturated. Sigmoid is often spotted in the output layer of neural networks in various tutorials. That's because most tutorials start with classification examples, and often deal with classification of photos. There, the network usually has as many output neurons as there are categories that the output gets classified into, and it is expected that one neuron has a value close to one, while the others are closer to zero. Here, however, we are doing a different kind of task, regression.

In our case, there is only one neuron in the output, and it should directly return the value of the approximation. We do not want to mess up with the signal at the output, and do not need to do any activation there. Since we still need to fit that functionality into the existing architecture, we create a kind of do-nothing activation, similar to Clojure's `identity` function. The derivative of this linear function is the constant `1`.

```
(deftype LinearActivation []
  Activation
  (activ [_ z a!]
    (copy! z a!))
  (prime [this z!]
    (entry! z! 1)))

(defn linear
  ([]
   (fn [_] (->LinearActivation)))
  ([z!]
    z!))
```

We fix the network and repeat the process.

```
(def inference (init! (inference-network
                       native-float 4
                       [(fully-connected 16 sigmoid)
                        (fully-connected 64 tanh)
                        (fully-connected 8 tanh)
                        (fully-connected 1 linear)])))

(def training (training-network inference x-train))
```

Checking the inference on the untrained network, we, unsurprisingly, get useless answers.

```
(inference x-test)

=>
#RealGEMatrix[float, mxn:1x5, layout:column, offset:0]
┌       ↓       ↓       ↓       ↓       ↓       ┐
→       0.71    0.71    0.71    0.71    0.71
└                                               ┘
```

One epoch later, we see that the cost is quite high.

```
(sgd training y-train quadratic-cost! 1 0.05)

=> 1.0261352330502589
```

We repeat the inference, only to see that the network has not learned much, but it has changed.

```
(inference x-test)

=>
#RealGEMatrix[float, mxn:1x5, layout:column, offset:0]
```

```
⌜       ↓       ↓       ↓       ↓       ↓       ⌝
→        1.01    1.01    1.01    1.01    1.01
⌞                                               ⌟
```

One epoch later, the cost decreased.

```
(sgd training y-train quadratic-cost! 1 0.05)
```

```
=> 0.6637627432927955
```

As expected, the inference is still bad.

```
(inference x-test)
```

```
=>
#RealGEMatrix[float, mxn:1x5, layout:column, offset:0]
⌜       ↓       ↓       ↓       ↓       ↓       ⌝
→        1.20    1.20    1.20    1.20    1.20
⌞                                               ⌟
```

Are 10 epochs enough to see some improvement?

```
(sgd training y-train quadratic-cost! 10 0.05)
```

```
=> 0.11728069185146087
```

Hooray, now the loss decreased 10 times! How's the inference doing?

```
(inference x-test)
```

```
=>
#RealGEMatrix[float, mxn:1x5, layout:column, offset:0]
⌜       ↓       ↓       ↓       ↓       ↓       ⌝
→        1.96    1.96    1.96    1.96    1.96
⌞                                               ⌟
```

It does not seem to be any better. Let's do a 100 epochs more.

```
(sgd training y-train quadratic-cost! 100 0.05)
```

```
=> 0.10831777675424309
```

The loss doesn't seem to go much lower. The inference is still bad.

```
(inference x-test)
```

```
=>
#RealGEMatrix[float, mxn:1x5, layout:column, offset:0]
⌜       ↓       ↓       ↓       ↓       ↓       ⌝
→        2.06    2.06    2.06    2.06    2.06
⌞                                               ⌟
```

Maybe the learning rate is too big. Let's decrease it a bit.

```
(sgd training y-train quadratic-cost! 100 0.03)
```

```
=> 0.10831182885244489
```

The loss seems to stay at the same level, and the inference has not improved.

```
(inference x-test)
```

```
=>
#RealGEMatrix[float, mxn:1x5, layout:column, offset:0]
┌       ↓       ↓       ↓       ↓       ↓       ┐
→       2.06    2.06    2.06    2.06    2.06
└                                               ┘
```

This is puzzling. If the cost decreased, why the inference seems to be as bad as with the untrained network? Well, the inference *has* improved in some way. Although it is still quite bad for each particular example, and outputs are always near `2.06`, it is closer on average to true values than it was when all answers were `0.71`.

We try with 1000 epochs, and yet lower learning rate.

```
(sgd training y-train quadratic-cost! 1000 0.01)
```

```
=> 0.10829182024434049
```

It has not helped at all.

```
(inference x-test)
```

```
=>
#RealGEMatrix[float, mxn:1x5, layout:column, offset:0]
┌       ↓       ↓       ↓       ↓       ↓       ┐
→       2.07    2.07    2.07    2.07    2.07
└                                               ┘
```

Maybe we need to vary the learning rate a bit. Let's try that.

```
(sgd training y-train quadratic-cost! [[100 0.03][100 0.01][100 0.005][100 0.001]])
```

```
=>
(0.10828583755013534 0.10828376051097366 0.10828274862731632 0.10828255335206705)
```

We can see that, as the learning progresses, the cost stays roughly the same, which means that the network just strolls around, but does not progress much.

```
(inference x-test)
```

```
=>
#RealGEMatrix[float, mxn:1x5, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ↓       ↓       ┐
   →       2.07    2.07    2.07    2.07    2.07
   └                                               ┘
```

Before throwing the towel, canceling the book altogether, and admitting that we are useless without frameworks created by the Big Co., let us remember that the task that we are doing here is not classification, when it is enough that the network learns to discriminate between a few, or several, discrete categories. Here we are doing regression, which is more difficult, since the network has to learn to approximate the actual real value of the function. Also consider that we just constructed a network with a random structure, and use the vanilla gradient descent, without any advanced tricks. Maybe we need to give it more time. Let us see what it can do with 40000 epochs worth of lessons.

```
(time (sgd training y-train quadratic-cost! 40000 0.05))
```

```
=>
"Elapsed time: 135905.426141 msecs"
0.0028870595858461454
```

Now the cost is significantly lower. It can be directly seen when we test the inference.

```
(inference x-test)
```

```
#RealGEMatrix[float, mxn:1x5, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ↓       ↓       ┐
   →       2.51    1.97    1.40    2.48    2.99
   └                                               ┘
```

Right! Much closer to the real values. We can never expect to get the exact floating point values that the real function is returning, especially not with the test observations that the network hasn't seen during the learning phase, but the difference is within an acceptable range.

```
(axpy! -1 y-test (inference x-test))
```

```
#RealGEMatrix[float, mxn:1x5, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ↓       ↓       ┐
   →       0.02   -0.12    0.08    0.14    0.05
   └                                               ┘
```

If we wanted to improve the approximation, we should probably train the network for longer. However, do not assume that more training leads to better approximation of *unseen* data. As the learning progresses, the network will generally decrease the cost, but after some time, some local optimum is reached, and the cost may oscillate, or even start to increase. There is no guarantee when or if the network will reach some optimal state. Additionally, this cost is measured on the training data.

Fortunately, we do not even want to decrease the cost too much. As we will see in later chapters, this cost generally decreases with more training, but the metric that *is* important is the cost measured on the *test* or *validation* data, and this cost generally *increases* when the network is *overtrained*.

In practice, that might indicate *overfitting*. The network that is optimized for the training data too much, might work poorly on the data that it hasn't seen during the learning process, and this is exactly the data that we want it to work well with.

These are high-level things to worry about. For now, it is enough to see that our network works, and to get a feeling of how difficult the task of training is. We needed a huge number of epochs to get acceptable results, and may need even more to get something good. We have a tight implementation, without much resource waste. Imagine how long it would take with something that was less optimized.

GPU

On the CPU, this particular network took two minutes to learn something useful. Since GPU can be order(s) of magnitude faster, they should take only a few seconds. Right?

Nvidia GPU with CUDA

Let's try with our CUDA-based implementation.

```
(cuda/with-default
  (with-release [factory (cuda-float (current-context) default-stream)]
    (with-release [cu-x-train (ge factory 4 10000)
                   cu-y-train (ge factory 1 10000)
                   inference (init! (inference-network
                                     factory 4
                                     [(fully-connected 16 sigmoid)
                                      (fully-connected 64 tanh)
                                      (fully-connected 8 tanh)
                                      (fully-connected 1 linear)]))
                   training (training-network inference cu-x-train)]
      (transfer! x-train cu-x-train)
      (transfer! y-train cu-y-train)
      (time
         (sgd training cu-y-train quadratic-cost! 40000 0.05)))))
```

```
"Elapsed time: 23251.819897 msecs"
0.0028752992499551057
```

23 seconds! Faster than on the CPU, but not as much as we hoped. It is worth remembering that *the size of the task have to be demanding* to see these orders of magnitudes in speedup. With relatively small matrices, it's good that the GPU engine was not even slower than the CPU!

AMD GPU with OpenCL

Our implementation for older AMD hardware and drivers had some performance issues in earlier chapters.

```
(opencl/with-default
  (with-release [factory (opencl-float *context* *command-queue*)]
    (with-release [cl-x-train (ge factory 4 10000)
                   cl-y-train (ge factory 1 10000)
                   inference (init! (inference-network
                                     factory 4
                                     [(fully-connected 16 sigmoid)
                                      (fully-connected 64 tanh)
                                      (fully-connected 8 tanh)
                                      (fully-connected 1 linear)]))
                   training (training-network inference cl-x-train)]
      (transfer! x-train cl-x-train)
      (transfer! y-train cl-y-train)
      (finish!)
      (time
         (last (sgd training cl-y-train quadratic-cost! (repeat 40000 [1 0.05])))))))
```

```
"Elapsed time: 559357.909222 msecs"
=>
0.002766931535136348
```

This is terrible. It is even slower than the CPU. The reason is that the engine is optimized for large matrices, and can not find a way to saturate hardware potentials it has with such small chunks of data that it's being provided with.

Smaller is often better

We have seen that the brute force can help with efficiency, but eventually hits the wall. Since the architecture of the network was arbitrary, maybe we can get better results with a *smaller* network. As we are interested in experimenting with different sizes, we care only about the cost, not the actual learned values. We use the engine that seems to be the fastest with this task.

Let's try a `4-8-16-4-1` network. Since it is smaller, we hope that a smaller number of epochs would be enough.

```
(cuda/with-default
  (with-release [factory (cuda-float (current-context) default-stream)]
    (with-release [cu-x-train (ge factory 4 10000)
                   cu-y-train (ge factory 1 10000)
                   inference (init! (inference-network
                                     factory 4
                                     [(fully-connected 8 sigmoid)
                                      (fully-connected 16 tanh)
                                      (fully-connected 4 tanh)
                                      (fully-connected 1 linear)]))
                   training (training-network inference cu-x-train)]
      (transfer! x-train cu-x-train)
      (transfer! y-train cu-y-train)
      (time
         (sgd training cu-y-train quadratic-cost! 4000 0.05)))))
```

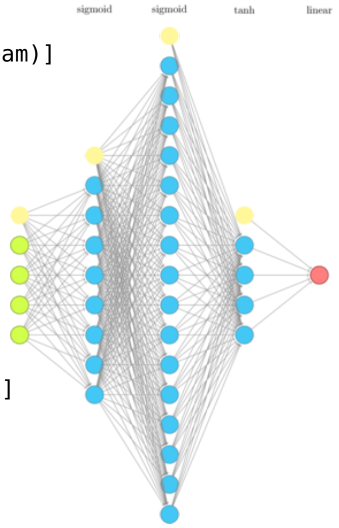


Figure 15: The 4-8-16-4-1 network.

```
"Elapsed time: 2049.335159 msecs"
=>
0.004670825056062358
```

4000 epochs took two seconds, and the error seems low enough to indicate that the network leaned something.

Does it learn more in 40000 epochs? Not that much, it seems.

```
"Elapsed time: 20982.76644 msecs"
=>
0.003014854083318096
```

From this, we can conclude that although the GPU did not take less time to compute these smaller layers, the learning algorithm itself got better results since smaller space can be explored with fewer steps.

Let's try the same code with an even smaller network: 2 hidden layers with 8 and 4 neurons.

```
"Elapsed time: 1527.71754 msecs"
=>
0.1007463554173708
```

What about changing the structure so the first hidden layer has 4 and the second layer has 8 neurons?

```
"Elapsed time: 1539.367704 msecs"
=>
0.004476395398100112
```

Lucky us, this 4-8 network can learn with similar cost as the larger 8-16-4, or the much larger 16-64-8 networks.

Let's try this small network with OpenCL.

```
(opencl/with-default
  (with-release [factory (opencl-float *context* *command-queue*)]
    (with-release
      [cl-x-train (ge factory 4 10000)
       cl-y-train (ge factory 1 10000)
       inference (init! (inference-network
                         factory 4
                         [(fully-connected 4 sigmoid)
                          (fully-connected 8 tanh)
                          (fully-connected 1 linear)]))
       training (training-network inference cl-x-train)]
      (transfer! x-train cl-x-train)
      (transfer! y-train cl-y-train)
      (finish!)
      (time
       (last (sgd training cl-y-train quadratic-cost! (repeat 4000 [1 0.05])))))))
```

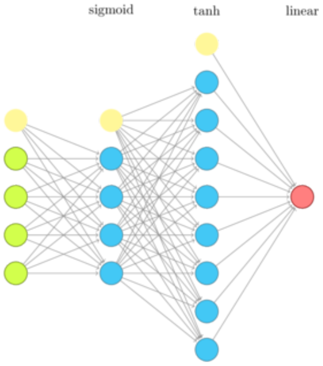


Figure 16: The 4-4-8-1 network.

```
"Elapsed time: 27948.357329 msecs"
=>
0.09556360326748764
```

We expect the CPU engine to work particularly well with these small networks, since it doesn't rely on parallelization that much.

```
(def inference-881 (init! (inference-network
                           native-float 4
                           [(fully-connected 8 sigmoid)
                            (fully-connected 8 tanh)
                            (fully-connected 1 linear)])))
(def training-881 (training-network inference-881 x-train))

(time (sgd training-881 y-train quadratic-cost! 4000 0.05))

"Elapsed time: 2920.363189 msecs"
=>
0.0047413169616575485
```

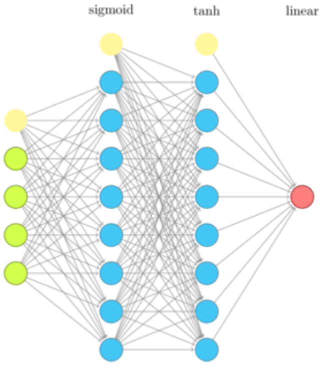


Figure 17: The 4-8-8-1 network.

Let's test the inference of this 4-8 network trained during 4000 epochs.

```
(inference-881 x-test)
```

```
=>
#RealGEMatrix[float, mxn:1x5, layout:column, offset:0]
⌜       ↓       ↓       ↓       ↓       ↓       ⌝
→       2.51    2.02    1.36    2.46    2.85
⌞                                               ⌟
```

```
y-test
```

```
=>
#RealGEMatrix[float, mxn:1x5, layout:column, offset:0]
⌜       ↓       ↓       ↓       ↓       ↓       ⌝
→       2.49    2.08    1.32    2.34    2.93
⌞                                               ⌟
```

Finally, let's compare the network's answers with the known outputs of the *known* function it approximates.

```
(axpy! -1 y-test (inference-881 x-test))
```

```
#RealGEMatrix[float, mxn:1x5, layout:column, offset:0]
⌜       ↓       ↓       ↓       ↓       ↓       ⌝
→       0.02   -0.06    0.05    0.13   -0.08
⌞                                               ⌟
```

It is acceptable, at least for these baby steps. Most approximated values are within a few percent of the real values for the examples in the test set that *the network has never seen before*.

The first milestone

After quite a few chapters we have created an implementation that is complete enough to be used for demo purposes. Depending on whether we expected a miracle, or we had already known how difficult are the problems that machine learning is applied to, we might be underimpressed or we might be jumping from joy right now.

The bottom line is that our implementation works quite efficiently, but is not as effective as we would like it to be. Since we can be quite confident that the code we wrote is quite tight, it is time to see whether we can improve the algorithm itself.

As we identified that the algorithm is fragile in regards to actual weight values, it makes sense to implement a way to keep them in check, so they stay in a zone where the gradient descent can progress well. We have also seen that the algorithm can spend a lot of time chasing local optimums. It may be a good idea if we can improve it to jump over pebbles and boulders in search for the valleys, and stop only in front of mountain peaks.

We will implement a few obvious improvements to the gradient descent that could be said to work well universally, and then we will revisit this same example for comparison. After we address the low hanging fruit, we may even try to learn some useful things from some real data, instead of playing with simulations.

Training optimizations

Weight decay

Momentum and Nesterov momentum

Adaptive learning rates

Regression: Boston housing prices

Dropout

Stochastic gradient descent

Classification: IMDB sentiments

Classification is another canonical machine learning task. While regression approximates the exact real value of the output, classification tries to categorize the output. The exact value of the output signal is not important, as long as the output is properly associated with one among two or more discrete classes.

That task seems to be easier than regression. Even if the output signal has a huge error, the category may be well chosen. On the other hand, if the signal is somewhere along the border between two categories, even if it is very precise, small differences may cause that it ends up in the wrong class.

While we might tolerate less precision in regression, we often leave much less space for miss-classification. Even if a software system classifies cars well for thousands of times, one error of classifying a car as a pedestrian could lead to a disaster.

A nice and popular example to introduce classification with neural networks is sentiment analysis of movie reviews.[41]

The IMDB sentiments dataset

Internet Movie Data Base (IMDB) provides extensive info about many forms of moving pictures, including user-contributed movie reviews. There is a freely available[42] data set of 100,000 reviews split into the train and test part.

Let's explore the structure of this data set, which is available in CSV format. We read the file that contains all 100,000 rows, but take only the first two.

```
(->> (io/resource "imdb-sentiment/imdb_master.csv")
     (slurp)
     (csv/read-csv)
     (take 2))
```

There are only 5 columns: `id`, `type`, `review`, `label`, `file`.

```
(["" "type" "review" "label" "file"]
```

[41] This example is often used in tutorials, and is also used in the *Deep Learning for Python* book, which makes it convenient for comparisons.

[42] It can be downloaded from Kaggle or elsewhere.

```
["0"
 "test"
 "Once again Mr. Costner has dragged out a movie for far longer than necessary.
 Aside from the terrific sea rescue sequences, of which there are very few I just
 did not care about any of the characters. Most of us have ghosts in the closet,
 and Costner’s character are realized early on, and then forgotten until much later,
 by which time I did not care. The character we should really care about is a very
 cocky, overconfident Ashton Kutcher. The problem is he comes off as kid who thinks
 he’s better than anyone else around him and shows no signs of a cluttered closet.
 His only obstacle appears to be winning over Costner. Finally when we are well
 past the half way point of this stinker, Costner tells us all about Kutcher’s
 ghosts. We are told why Kutcher is driven to be the best with no prior inkling
 or foreshadowing. No magic here, it was all I could do to keep from turning it
 off an hour in."
 "neg"
 "0_2.txt"])
```

We immediately see that only two fields are relevant for each review: the text of the review, and its label, which is a binary value; it can be `neg` or `pos`.

Figuring out how to teach NNs with the IMDB dataset

It is intuitively clear to us what this data mean; there is some text, and there is the expected sentiment value attached to that text, either positive or negative. We should put the text of each review at the input, and the network has to produce the sentiment value at its output.

The output should not be an issue. Assuming that the output is a number, we can easily devise a scheme that translates numbers above threshold with `pos`, and those below with `neg`, or something along these lines.

The problem, though, is that the network expects numbers at the input, while we just have a free-form text. We have to find a *meaningful* way to encode a string of variable length to a vector of fixed length, which is the only thing that neural networks can accept at the input.

We could try to utilize the fact that each character can be treated as its ASCII value.[43] This does not solve the issue that the resulting vector of numbers has variable length. More important flaw, though, is that this sequence of numbers is not very meaningful. Characters `a` and `b` are next to each other, while `a` and `z` are quite far away, but this distance does not have much to do with how words are formed, yet alone with the sentiment of the whole paragraph.

There are two straightforward ways to turn these strings into vec-

[43] For the sake of argument, we ignore that Java strings use unicode.

tors. Both of these encode *words* instead of single characters.

We can index each word, encode strings as variable lists of numbers, pad these lists so that they have the same length, and convert them to vector inputs. These inputs are not straightforward, but require a special layer at the network input to learn word embeddings.[44]

Alternatively, we could use one-hot encoding, a simpler sparse representation. One-hot encoding uses a large vector, where each word has an index, and the value of that index is `1` when the word is present in the text, and `0` when not. This method can only encode a finite number of chosen words, in a crude binary manner.

One-hot encoding's crudeness is a great opportunity to show how neural networks can make use of even when minimal effort is spent on preparing an unsophisticated input.[45]

Implementing one-hot encoding for the IMDB dataset

If the chosen dimension of the input is 10,000, we should choose 10,000 words that we will use. The most convenient way is to scan all reviews, count the frequencies of all words, and choose the most popular 10,000 words. Each word is represented by a specific position in the vector. For example, the word "car", which is surely frequent enough to make it to the top 10K, might be at position 325. Then, we scan the reviews again and set the values at the appropriate indexes to `1`. Such input can be used by plain dense layers that we have developed.

There is no need to use specialized data frame libraries for this task, since plain Clojure is quite powerful and can give us a pretty good run for our money. Let's explore the partial steps first.

The first review is in the second row, after the header. We extract it to its own var and use it as an example.

```
(def costner-review (second *1))
```

The first task is to split the review string into the constituent words. We use the `split` function and get the vector of words.

```
(string/split (costner-review 2) #" ")

;; This output is truncated to first 10 words
["Once" "again" "Mr." "Costner" "has" "dragged" "out" "a" "movie" "for" ...]
```

Right away, we can create a function that, given a review, discards the irrelevant fields, and creates a vector of the relevant ones, including the split review.

```
(defn split-review [review]
```

[44] Word embeddings are dense, low-dimensional representations of text that would require a bit more explanation that is out of scope of this book.

[45] Naturally, if we wanted to seriously work with textual input, we should have explored specialized techniques, including word embeddings.

```
  (vector (review 1) (string/split (review 2) #" ") (review 3)))
```

```
(split-review costner-review)
```

```
["test"
 ["Once" "again" "Mr." "Costner" "has" "dragged" "out" "a" "movie" "for" ...]
 "neg"]
```

Finding the 10,000 most frequent words

To find the most frequent words in the whole dataset, we should be able to find the frequency of each word in the split review. Clojure's `frequencies` function can already find the frequencies of all values in a sequence. We can use it as-is since we already keep the words in a vector, which satisfies the sequence abstraction.

```
(frequencies (string/split (costner-review 2) #" "))
```

```
{"else" 1,
 "Kutcher's" 1,
 "us" 2,
 "signs" 1,
 "driven" 1,
 "him" 1,
 "Mr." 1,
 "are" 4,
 "Kutcher" 1,
 ...}
```

We should accumulate these frequencies across the reviews. We create the `read-imdb-master` function, and use it to load the first 100 reviews that we will use in experimenting. Loading the whole sequence would take a few more seconds than necessary. It could be argued that we could load and parse all 100,000 rows, and store the sequence in a var to reuse. That would use a few hundred of megabytes, which is not a problem for contemporary desktop computers. However, it could be a problem when working with larger data sets.

Clojure sequences are lazy, which means that they can process infinite sequences, since they use memory only for entries that are currently processed. That functionality strictly forbids from storing references of the head of the sequence, since then the garbage collector cannot release the elements that have been processed.

In this exploratory example, we have this in mind even though supporting infinite data sets is not necessary.

```
(defn read-imdb-master
  ([]
   (->> (io/resource "imdb-sentiment/imdb_master.csv")
        (slurp)
        (csv/read-csv)
        (drop 1)))
  ([cnt]
   (take cnt (read-imdb-master))))

(def first-100-reviews (pmap split-review (read-imdb-master 100)))
```

The `word-frequencies` function computes the frequencies of all reviews, and then merges adds up all results by merging.

```
(defn word-frequencies [reviews]
  (apply merge-with + (pmap #(frequencies (% 1)) reviews)))
```

This function returns the frequencies of all words, and we can quickly check whether it works at all by taking a frequency of a common word.

```
(get (word-frequencies first-100-reviews) "is")
```

The word "is" seems to be present 327 times in the first 100 reviews, which seems about right.

```
=> 327
```

Printing all resulting words would clutter this page, but we can at least check how many of them there are.

```
(count (word-frequencies first-100-reviews))
```

The first 100 reviews contain 5978 distinct words.

```
5978
```

Checking how many distinct words are contained when all reviews are taken into account might be a nice and easy exercise. The result should be 696068.

We could use all 696068 words in the analysis. Uncommon words in the long tail would not help with learning, while they still require one additional dimension per word. We are going to use the 10,000 words that appear most frequently.

Once we have computed the frequencies of all words, we can sort them by popularity using Clojure’s built in functions, and just take a desired number of the most frequent ones. In the following example, we are taking 10 most frequent words from first 100 reviews.

```
(take 10 (sort-by val > (word-frequencies first-100-reviews)))
```

Unsurprisingly, the most popular words are "the", "a" etc. This gives us a hint that it could be a good idea to exclude the most popular words from our analysis as well. On the other hand, neural networks can learn to ignore features that are not important, so we are going to skip any clever data preparation that is not absolutely necessary.

```
(["the" 1060]
 ["a" 561]
 ["of" 521]
 ["and" 520]
 ["to" 488]
 ["is" 327]
 ["in" 308]
 ["I" 235]
 ["this" 227]
 ["that" 223])
```

Once we ordered the words by popularity, we can discard the frequencies, and just keep the sequence of words. From each vector `[word frequency]` we need the element at the index `0`. We codify our recent experiments into a reusable `word-vec` function.

```
(defn word-vec [reviews cnt]
  (->> (word-frequencies reviews)
       (sort-by val >)
       (map #(% 0))
       (take cnt)
       (into [])))
```

We repeat the same experiment using this convenient function.

```
(word-vec first-100-reviews 10)
```

```
["the" "a" "of" "and" "to" "is" "in" "I" "this" "that"]
```

Encoding reviews

When we have a list of words in a review, we have to encode it; meaning we have to flip the value at the index of each word in the vector that represents the review from `0.0` to `1.0`. Given the word "this", for example, we have to know that its index is `8`.

If we only had the list of the words, we would have to scan it each time to find that index, which would be comically inefficient. A way to avoid this problem is to build a hash-map dictionary that quickly finds the index of any word, or answers with nil if a word is not present in the dictionary.

This is simple to do with plain Clojure functions. We create the word-map function which, given a Clojure vector of words, builds a hash-map of their indices.

```
(defn word-map [word-vector]
  (into {} (map #(vector (word-vector %) %) (range (count word-vector)))))
```

Let's test it with the 10 most frequent words from the first 100 reviews.

```
(word-map (word-vec first-100-reviews 10))
```

```
{"of" 2,
 "this" 8,
 "is" 5,
 "that" 9,
 "a" 1,
 "and" 3,
 "I" 7,
 "to" 4,
 "the" 0,
 "in" 6}
```

The encoded IMDB dataset

We are now ready to encode the entire dataset into the final shape that can be consumed by neural networks. We create the Clojure vector of 10,000 most frequent words into wvec, and we build the matching hash-map of indices wmap.

```
(def wvec (word-vec (pmap split-review (read-imdb-master)) 10000))
(def wmap (word-map wvec))
```

We create the encompassing function encode-review, which, given a word-map, and the review in the [_ words sentiment] format, handles the review as we have discussed in detail, and populates the appropriate places in the input vector x and the output vector y, which will be consumed by the network

```
(defn encode-review [word-map review x y]
  (let [[_ words sentiment] (split-review review)]
    (doseq [idx (map word-map words)]
      (when idx (entry! x idx 1.0)))
    (entry! y 0 (case sentiment "neg" 0 "pos" 1)))
  x)
```

We demonstrate how it works on the first review, `costner-review`. The input vector is a 10,000 dimensional vector, and the output is a vector that has one entry, which is going to be `0` when sentiment is `"neg"`, and `1` when the review sentiment is `"pos"`, when encoded.

```
(def costner-code (encode-review wmap costner-review (fv 10000) (fv 1)))
```

```
=>
#RealBlockVector[float, n:10000, offset: 0, stride:1]
[   1.00    1.00    1.00          0.00    0.00 ]
```

Even though we do not need to decode this vector, it is a good idea to write the `decode-review` function, which we could use for testing. The last thing we need now is that we feed our network training process with garbled data.

```
(defn decode-review [word-vec code-vec]
  (filter identity
          (map #(if (< 0.5 (entry code-vec %))
                  (word-vec %)
                  nil)
               (range (dim code-vec)))))
```

```
(decode-review wvec costner-code)
```

It seems that the result makes sense.

```
=>
;; truncated output
("the" "a" "and" "of" "to" "is" "in" "I" "this" "it" "was" "as" "with" "for" "The"
 "movie" "are" "have" "not" "be" "by" "he" "an" "from" "who" "all" "has" "just" "or"
 "about" "out" "very" "when" "only" "really" "which" "no" "than" "there" "much" "time"
 "we" "could" "do" "any" "him" "then" "way" "well" "character" ...)
```

Finally, we automate the whole procedure in the `encode-review` function. It takes the map of word indexes and all reviews, create the appropriate input and output matrices, whose columns are vectors of the appropriate dimensions, 10,000 and 1, and have 25,000 rows, one per each review.

```
(defn encode-reviews [wmap reviews]
  (let-release [in (fge 10000 25000)
                out (fge 1 25000)]
    (doall (map #(encode-review wmap %1 %2 %3) reviews (cols in) (cols out)))
    [in out]))
```

We only use 50,000 reviews of the 100,000 that are available, since only the first half has the `pos` and `neg` sentiments recorded. When doing this at home, do not forget to shuffle the data, which is required for the *stochastic* gradient descent algorithm that we use.

```
(def data (doall (map #(encode-reviews wmap (shuffle %))
                      (split-at 25000 (read-imdb-master 50000)))))
```

```
([#RealGEMatrix[float, mxn:10000x25000, layout:column, offset:0]
   ⌜       ↓       ↓       ↓       ↓       ↓       ⌝
   →       1.00    1.00    ∵       1.00    1.00
   →       1.00    1.00    ∵       1.00    1.00
   →       ∵       ∵       ∵       ∵       ∵
   →       0.00    0.00    ∵       0.00    0.00
   →       0.00    0.00    ∵       0.00    0.00
   ⌞                                               ⌟
 #RealGEMatrix[float, mxn:1x25000, layout:column, offset:0]
   ⌜       ↓       ↓       ↓       ↓       ↓       ⌝
   →       0.00    1.00    ∵       1.00    0.00
   ⌞                                               ⌟
] [#RealGEMatrix[float, mxn:10000x25000, layout:column, offset:0]
   ⌜       ↓       ↓       ↓       ↓       ↓       ⌝
   →       1.00    1.00    ∵       1.00    1.00
   →       1.00    1.00    ∵       1.00    1.00
   →       ∵       ∵       ∵       ∵       ∵
   →       0.00    0.00    ∵       0.00    0.00
   →       0.00    0.00    ∵       0.00    0.00
   ⌞                                               ⌟
 #RealGEMatrix[float, mxn:1x25000, layout:column, offset:0]
   ⌜       ↓       ↓       ↓       ↓       ↓       ⌝
   →       0.00    1.00    ∵       1.00    1.00
   ⌞                                               ⌟
])
```

We give convenient names to these two pairs of input and output matrices.

```
(def x-test ((first data) 0))
(def y-test ((first data) 1))

(def x-train ((second data) 0))
(def y-train ((second data) 1))
```

Training the network

Finally, we are ready to train the network, which, compared to the preparation effort, seems almost trivial! We choose the 10000-16-16-1 network architecture with relu activations, and mini-batch size of 512, following popular resources that use this example.

```
(def x-minibatch (ge x-train (mrows x-train) 512))
```

```
(def inference (inference-network native-float 10000
                                  [(fully-connected 16 relu)
                                   (fully-connected 16 relu)
                                   (fully-connected 1 sigmoid)]))

(init! inference)
```

We use the sigmoid activation at the output, not the linear activation, because we are doing classification, and would like to have output between 0 and 1, which can be easily transformed into `neg` and `pos`.

To be able to measure the accuracy of the network output, we create the function that rounds the output to `0` or `1`, and compares it with the expected values.

```
(defn binary-accuracy!
  ([y a!]
   (- 1.0 (/ (asum (axpy! -1.0 y (round! a!))) (dim y)))))
```

When we apply this function to the untrained network, we expect that it is not more accurate than pure chance.

```
(binary-accuracy! y-test (inference x-test))
```

The accuracy is 0.5, just as we have expected.

```
=> 0.5
```

We create a training network with Adam layers, and train it for 20 epochs.

```
(def adam (training-network inference x-minibatch adam-layer))
(time (sgd-train adam x-train y-train quadratic-cost! 20 []))
```

```
"Elapsed time: 5376.243553 msecs"
=> 0.02493325881067765
```

The network achieved low cost, indicating that the training went well, achieving excellent performance.[46]

Let’s test the accuracy of this network on the test data.

```
(binary-accuracy! y-test (inference x-test))
```

The network’s accuracy is 88%.

```
=> 0.88324
```

[46] Considerably faster than mainstream libraries such as TensorFlow or PyTorch!

This is in line with the results achieved by the same network built with mainstream technologies.

Maybe we could improve the test cost by using Dropout.

```
(def adam-dropout (dropout adam))
(init! inference)
(time (sgd-train adam-dropout x-train y-train quadratic-cost! 20 [[0.005 0.0005]]))
```

```
"Elapsed time: 6246.483023 msecs"
=> 0.06372851434459736
```

```
(binary-accuracy! y-test (inference x-test))
```

The accuracy is roughly the same.

```
=> 0.87652
```

How does RMSprop stand in comparison to Adam?

```
(def rmsprop (training-network inference x-minibatch rmsprop-layer))
(init! inference)
(time (sgd-train rmsprop x-train y-train quadratic-cost! 20 [0.001 0.9]))
```

```
"Elapsed time: 5147.082232 msecs"
=> 0.013999668986643066
```

```
(binary-accuracy! y-test (inference x-test))
```

```
=> 0.87588
```

It seems that 88% is the limit for this network architecture. This is in line with the results reported by other literature. For this particular problem, the state of the art, achieved by more complex methods specialized for this purpose is 95%. Consider the fact that neural networks achieved slightly worse accuracy with no particular specialization, practically automatically.

Cross-entropy cost function

Sigmoid activation function that we use at the output layer saturates when the output is close to 0.0 or 1.0. This causes the derivative of quadratic cost to change very little when the network is wrong, and the learning progresses slowly. Regression with linear output does not have this problem, since linear output does not saturate.

In the sigmoid activation with quadratic cost[47] combination, the gradient with respect to weights depends on the derivative of sigmoid[48].

[47] Quadratic cost function, for reference.

$$C(w,b) \equiv \frac{1}{2n}\sum_x \|\mathbf{y}_x - \mathbf{a}_x\|^2 \qquad (7)$$

[48] The derivative of the sigmoid function, for reference.

$$\sigma' = \frac{d\sigma}{dx} = \sigma(x)\dot{(}1 - \sigma(x)) \qquad (8)$$

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{9}$$

However, when σ is saturated, σ' becomes close to 0, which makes δ^L close to zero.

Our example proved that the network *can* in fact learn well even though quadratic cost function is not an ideal match for the sigmoid activation. The theory proves that this causes issues, which we did not encounter because we have created a quite robust training infrastructure. However, we must solve this problem if we would like our software to work well with more challenging classification problems.

The right match for the sigmoid output is the Cross-entropy cost function.

$$C(w,b) = \frac{1}{n}\sum_x [\mathbf{y}_x \ln \mathbf{a}_x + (1 - \mathbf{y}_x)\ln(1 - \mathbf{a}_x)] \tag{10}$$

It is not immediately obvious why this function is a good choice, or why it is acceptable as a cost function at all. It is acceptable because it is non-negative, and it is close to 0 when the neuron's output is close to the target value. Why is it better than quadratic cost, when the formula for δ^L includes σ' in both cases? Seeing how the error looks like will help us understand why.[49]

$$\delta^L = \frac{(\sigma(z) - \mathbf{y}) \odot \sigma'(z)}{\sigma(z) \odot (1 - \sigma(z))} = \frac{\sigma'(z) \odot (\sigma(z) - \mathbf{y})}{\sigma'(z)} = \sigma(z) - \mathbf{y} \tag{11}$$

The σ' term cancells, and the error in the last layer only depends on the σ function, not on the problematic value of σ', which is near zero when the activation is saturated.

The quadratic cost function has been constructed so that, when combined with the sigmoid activation, the problematic term disappears. With some other activation function, this nice optimization would not work.

We have to change our infrastructure a bit before we use the following implementation of the `sigmoid-crossentropy-cost!` function.

```
(defn sigmoid-crossentropy-cost!
  ([y a]
   (with-release [ylna (mul! (log a) y)
                  y-1 (linear-frac 1.0 y -1.0)]
     (/ (real/asum (axpy! -1.0 ylna
                          (mul! y-1 (log! (linear-frac! -1.0 a 1.0)))))
        (ncols y))))
  ([activ-fn z! y a]
   (if (instance? SigmoidActivation activ-fn)
     (axpy! -1.0 y (copy! a z!))
```

[49] We skip the complete calculation, which you can find in Michael Nielsen's *Neural Networks and Deep Learning* book.

```
    (dragan-says-ex "This crossentropy implementation can only be used with sigmoid activation."
                    {:activ-fn-type (type activ-fn)}))))
```

The optimization works only when the sigmoid activation is used with the crossentropy cost. Additionally, the cost calculation up to now assumed that the input argument is `y-a`, which was enough for calculating the quadratic cost, while crossentropy needs both `y` and `a`. Finally, this optimization should be applied only in the last layer; the formula in the hidden layers does not care how the gradient that comes from the upper layers are calculated. To support this, we can extract the `backward-error` function from the `backward-error` function, and call it at appropriate times.

Here is the relevant part of `NeuralNetworkTraining`.

```
Backprop
  (forward [_ hyperparam]
    (doseq [layer forward-layers]
      (forward layer hyperparam))
    (output last-layer))
  (backward-error [_]
    (backward-error last-layer))
  (backward-error [_ cost! y]
    (backward-error last-layer cost! y))
  (backward [_ hyperparam]
    (backward last-layer hyperparam)
    (doseq [layer backward-layers]
      (backward-error layer)
      (backward layer hyperparam)))
```

`FullyConnectedAdam` and `FullyConnectedRMsprop` have to be adapted accordingly.

```
Backprop
  (forward [_ _]
    ...)
  (backward-error [_]
    (mul! (prime activ-fn z) a))
  (backward-error [_ cost! y]
    (cost! activ-fn z y a))
  (backward [_ [t eta lambda rho1 rho2 epsilon]]
    (let [...]
      (mm! (/ 1.0 (dim ones)) z (trans a-1) 0.0 g)
      ...))
```

Updating the existing infrastructure is a good way to see whether you grasp well the implementation we have covered so far.

Training the network (again)

Although the implementation slightly changed, we still support the existing API.

```
(def x-minibatch (ge x-train (mrows x-train) 512))

(def inference (inference-network native-float 10000
                                  [(fully-connected 16 relu)
                                   (fully-connected 16 relu)
                                   (fully-connected 1 sigmoid)]))

(init! inference)
(def adam (training-network inference x-minibatch adam-layer))
(time (sgd-train adam x-train y-train sigmoid-crossentropy-cost! 20 []))
```

```
"Elapsed time: 5344.798563 msecs"
=> 0.20265182852745056
```

```
(binary-accuracy! y-test (inference x-test))
```

```
=> 0.88236
```

There is no significant advancement. It seems that 88% is the ceiling for this basic network. This is actually a great result; our home-grown software matches mainstream tools in accuracy, while being much simpler and faster!

Doing it on the GPU

The same code should work on the GPU. Is it reasonable to expect it to be faster, since the dimension of the minibatch is 10,000 × 512?

```
(cuda/with-default
    (with-release [factory (cuda-float (current-context) default-stream)]
      (with-release [cu-x-train (ge factory 10000 25000)
                     cu-y-train (ge factory 1 25000)
                     cu-x-minibatch (ge factory 10000 512)
                     inference (init! (inference-network
                                       factory 10000
                                       [(fully-connected 16 relu)
                                        (fully-connected 16 relu)
                                        (fully-connected 1 sigmoid)]))
                     adam (training-network inference cu-x-minibatch adam-layer)]
        (transfer! x-train cu-x-train)
        (transfer! y-train cu-y-train)
        (time
         (sgd-train adam cu-x-train cu-y-train
                    sigmoid-crossentropy-cost! 20 []))))))

"Elapsed time: 705.944645 msecs"
0.10233936458826065
```

Yes! This network, despite being quite small, with only two hidden layers of 16 neuron each, runs an order of magnitude faster on this GPU! The key to this speedup is that the first layer is large, so the weight matrix in the first layer has 5,120,000 entries, which is not particularly large, but is not small.

Tensors

Classification and metrics: MNIST handwritten digits recognition

Tensors and ND-arrays

Tensor transformations

DNNL: Tensor operations on CPU

Tensor-based neural networks

cuDNN: Tensor operations on GPU

Convolutional networks

The convolution operation

Convolutional layers on the CPU with DNNL

Convolutional neural networks (CNN): Fashion-MNIST

CNN on the GPU with cuDNN

Appendix

Setting up the environment and the JVM

Many more early supporters helped me create this book. Thank you!

Vladimir Urosevic, Bill Vanyo, Nolan, Pei-Lan Hsu, Nick Brandaleone, Dorab Patel, Alex Lykostratis, Alan Thompson, Alister Lee, Adam Tait, Matthew hoopes, Christian Meichsner, aaelony, David Pham, Anders Murphy, Gurvesh Sanghera, Yiu Ming Huynh, Jacob Maine, Brian Abbott, Paula Gearon, Andrei Ursan, Michael Alyn Miller, Ragnar Dahlén, Nils Grunwald, atashi, Huahai Yang, Joel Wong, James Davidson, Fjolne Yngling, Carin Meier, Al King, Scott Jappinen, Ruud Reus, Brandon Adams, Kamil Toman, Henk Timmermans, Oliver Holworthy, Ning Sun, Jeffrey Cummings, Ronen Narkis, Timo Kauranen, Matthew Pettis, Daniel Lorencic, Anna S, , Henrik Suzuki, Elpizo Choi, Edwin Park, Klaus Gutenbrunner, Peter McTaggart, DUANE SEARSMITH, Phill Wolf, Davide Del Vecchio, Arthur Ulfeldt, Robert J. Berger, Alex Nelson, William Swaney, Joshua Done, Xavier Tapia, Don Jackson, Amar, Chris Blom, David M Woodhead, James Tolton, Not Invader Zim, Jason Gilbertson, John A Fingerhut, drew verlee, Christoph Ulrich, Ben Yorke, stask, Mandimby RAVELOARINJAKA, Gijs Stuurman, Tracy Shields, Daniel Jolicoeur, Andrei Zhurau, brian gorman, Stephen Stephens, Scott MacFarlane, Alexander, Bernard Hugueney, Jacob Filimonov, Damian, Pixel Pie, Haochen Xie, Alex, Bartosz Macięga, Mattias Arro, bahadir cambel, Dan Dorman, Thomas Gebert, Joshua M, Łukasz Kożuchowski, ezazel, Benjamin Loy, Jason Mulvenna, MR A BIN SALMAN, Leon Talbot, Kamuela Franco, Timothy Dean, James Conroy-Finn, Johann Böhme, refset, Amresh Venugopal, E., Key Tiong Tan, Robert Crim, Jon Anthony, Zack Teo, Martin, Leander, James Good, Rahul Nair, Agam Brahma, Daniel Tan, Magnus, , John Doe, Fed Reggiardo, Snorre Magnus Davøen, Venkatesh KS, Keith Mantell, Xceno Malloy, Jonathan Lin, Alex Doroshenko, Hjalmar Ekengren, Tim Wiess, Mojave Matt, Alfred Thompson, Randy Lien, Steven Harms, Brian Hurlow, Andy Mullenix, Marian, Vijay Edwin, David Millett, Peder Refsnes, Felipe Gerard, Ken Rawlings, Christian, Tim Johann, Adam Elga, Jason Thomas, Peter Denno, Matt Burbidge, Dario, Brian Brunner, Ales Huzik, Glenn Jahnke, didier, Aiken Drum, Szabolcs, Carsten Behring, Martin, Pauli Ojala, Brian Hurlow, Rohit Thadani, Soumya Chatterjee, Sam Heaton, Otis Clark, Martin Jung, sun, Paul Bellamy, Daniel Carleton, Steve Mayer, Away From People, Valerii Praid, Csaba Endre Simon, Martin van Amersfoorth, omnidelic, Andrew Baxter, Alex Shevchenko, Petter Egesund, Adam Morgan, Ari Lerner, JWtoch, Bor Hodoščék, Pavithra Solai Jawahar, Jonathan Smith, Christopher Hugh Graham, Pedro Cuenca, Ram Tripathi, pkvov, Kaz Librowski, Matt Hodgson, Tom Brooke, Jules White, max, Chad Harris, Rangel Spasov, Juan Ruiz, Tim Chase, 313 Ventures, Andrei Duhnea, Jason, Ned Twigg, Pedro Gomes, Hari Krishnan, Daniel Lee, Jon Irving, Nicholas Stares, Anthony Ru-

bin, Antti Rämö, ZAKH, Jason Waack, Jan van Esdonk, Mark Watson, leo garcia, Jahyun Gu, Nenad Mitrovic, Tory S. Anderson, Fernando Dobladez, Adolfo De Unanue, Mogens Brødsgaard Lund, Matus Lestan, Daniel Wood, Sooheon, tattarattat, Mike Coleman, TAKESHI NAKANO, Stephen Telford, Manas Marthi, Dmitry, Mikhail Sakhnov, Erik Olivier.