Contents

Chapter 1

Introduction

The best things about object-oriented programming are flexibility, extendability and reusability. The field of neural networks is a very wide, complicated, and continuously growing. It requires different methods, state-of-the-art solutions, and various heuristics. Building a flexible neural networks framework and a modeling tool that would allow you to extend and modify it based on your specific problem, is a good task for OOP.
In this work I will describe my implementation of neural networks in Pharo in an object-oriented manner. One important think to note is that I do not claim to offer any new approach to building neural networks. This field has been well studied long before I got introduced to it. The main goal of this project is to bring neural networks to Smalltalk. To show how this beautiful language can be used for the problems of machine learning and data analysis. And to make my contribution into something that may later become a full-scale machine learning toolkit.

1.1 Structure

First chapter In this chapter will explain what is Pharo, how it can be installed, and how to acquire the code of this project. I will also talk about the ideas behind implementing neural networks, as well as the other machine learning algorithms, in an object-oriented manner, and give a brief overview of the work done before in this field.

Second chapter Here I will give you the short introduction into machine learning and neural networks. I will talk about different kinds of learning, different network architectures, different ways of training them. Feel free to skip that chapter if you already have a good knowledge of the concepts in question. But do have a look at a section [?] where I talk about the confusing terminology and explain my reasoning behind naming the classes in this project.

Third chapter Here I talk about my implementation of a single-layer perceptron as a collection of neurons. Each neuron in this representation is a separate object. This approach may not be the best one performance-wise, but it can be used for modeling purposes.

Fourth chapter This chapter is dedicated to a multi-layer neural network. It is implemented as a collection of layers. Neurons are not the logical units anymore which allows us to perform some very expensive computations much faster using the vector-matrix operations implemented in the PolyMath package.

Fifth chapter In this final chapter I demonstrate how my multi-layer neural network can be used for the classic task of classifying the handwritten digits from MNIST dataset. This chapter gives you the detailed explaination of how to use the neural networks package and how to extend it to fit your specific needs. So if you want to start using the package right ahead - jump straight to the fifth chapter.

1.2 Related work

In 1995 Barry Ellingsen developed MANN (Modular Artificial Neural Network) system and described it in a technical report *An Object-Oriented Approach to Neural Networks*[6]. I took some inspiration from his work, but my implementation was also infuenced by modern packages, such as TensorFlow or scikit-learn.
There were several attempts of implementing neural networks in Pharo. The most notable one would be the work of Alexandre Bergel done in his package

1.3 Installation

To use my library you must first install Pharo. It is available for all major operating systems

1.3.1 Getting Pharo

Pharo is a modern, open-source, dynamically typed language supporting live coding and inspired by Smalltalk. The important principle behind Pharo is that it doesn't just copy the past, but reinvents the essence behind Smalltalk. Pharo is not read-only, it integrates the changes made by community, daily[4].
To download Pharo, visit the official website: `http://pharo.org/download`.

1.3.2 Getting the code

The code of this project can be acquired from Smalltalkhub using this Metacello script (`Do It` in a Playground of your Pharo image):

```
Metacello new
   repository: 'http://smalltalkhub.com/mc/Oleks/NeuralNetwork/main';
   configuration: 'MLNeuralNetwork';
   version: #development;
   load.
```

Or you can get it from the GitHub repository: `https://github.com/olekscode/MLNeuralNetwork`.
In future this project will be moved to GitHub.

Parts of this project depend on the following packages. They will be automatically installed and updated by Metacello.

- **PolyMath** - a library for numerical methods. It provides MATLAB-like vectors and matrices. It is simmilar to numpy library in Python.
- **Roassal** - a library for agile visualizations. The only part dependent on Roassal is MLVisualizer class. All data and metrics are provided in a type that is supported either by Pharo base, or by PolyMath. So if you wish to use something other visualization tool - you are free to do that. However, all examples in this [thesis] will be visualised with MLVisualizer which uses Roassal.
- **IdxReader** - a package for reading the data in idx format, designed by [Guilermo Polito]. The MLDataReader class uses it to read the MNIST dataset of handwritten digits. If you don't want to use this dataset - you can ignore this dependency. Everything else will work just fine.

1.4 Example of usage

To create an instance of MLNeuralNetwork class you must provide an inforamtion about the network architecture, defined by a simple one-dimensional array of integers $\#(s_1 s_2 ... s_n)$. The size of an array n represents the number of layers in the network, and each number s_i is the number of neurons in that layer.

```
neuralNet := MLNeuralNetwork new initialize: #(784 500 10).
```

This code will create a neural network with 784 input units, one hidden layer with 500 neurons, and an output layer with 10 neurons (this network can be used to classify the input data into 10 classes).

1.5 How to contribute?

Contributions are welcome and encouraged. The easiest way to contribute is to make a pull request to the GitHub repository of this project: `https://github.com/olekscode/MLNeuralNetwork`. If you have any questions or suggestions regarding this project, write me an email to `olk.zaytsev@gmail.com`.
If you want to implement some machine learning algorithms in Pharo - join us on the mailing list or on the #polymath channel of Pharo's server on Discord.

Chapter 2

Machine learning with neural networks

In this chapter I will provide the reader with the basic understanding of the concept of neural networks and the surrounding field. We will look at the general topic of machine learning, explain the place of neural networks as machine learning algorithms, explore different types of neural network architectures and so on. When it comes to OOP, it is very important to understand the entity that we are trying to represent in our code on a deep level. We must know what exactly is, and what isn't a neural network, what are the different types of networks, how are they different. What is the relation between a neural network and a neuron? Finally, when it comes to weights, are they properties of a neuron, or of a whole network? If it's neuron, then which neuron should store these weights, the one that receives the signal, or the one that emits it?

2.1 What is machine learning?

In recent years Machine Learning became one of the most promising and rapidly developing fields in Computer Science. It tackles the problems that classical programming and sometimes also humans can't handle. In this section I will give the short introduction to the field of Machine Learning.
In his book *Information Theory, Inference, and Learning Algorithms* [9] David MacKay writes:

> Machine learning allows us to tackle tasks that are too difficult to solve with fixed programs written and designed by human beings. From a scientific and philosophical point of view, machine learning is interesting because developing our understanding of machine learning entails developing our understanding of the principles that underlie intelligence.

Definition The intuitive definition of machine learning was given by Arthur Samuel in 1959:

> Machine Learning is a field of study that gives computers the ability to learn without being explicitly programmed.

This definition is nice and easy to understand. Though, to work with machine learning as a scientific field (indeed, it is a field of computer science, strongly related to some mathematical fields, such as computational statistics and mathematical optimization), we need a more formal definition. One can be found in Tom Mitchell's book *Machine Learning* (1997) [17]. This definition is widely known and often reffered to as a well-posed learning problem:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Machine learning problems By the way we measure performance P on task T machine learning tasks can be divided into three main classes:

- **Supervised Learning** - the agent receives the set of examples with labels ("right answers") to learn from
- **Unsupervised Learning** - no explicit feedback is provided. The agent should learn patterns in an unlabeled dataset
- **Reinforcement Learning** - the agent receives series of reinforcements - rewards or punishments (for example, winning or loosing the chess game)

Here are the formal definitions for the problems of supervised and unsupervised machine learning[1].

Task of supervised learning Given a training set of m example input-output pairs

$$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})$$

where each $y^{(j)}$ is generated by an unknown function $y^{(j)} = f(x^{(j)})$
Goal: discover the function h that approximates function f.

Task of unsupervised learning Given a large unlabeled dataset of m input examples

$$x^{(1)}, x^{(2)}, ..., x^{(m)}$$

where each $x^{(i)} \in \mathbb{R}^n$.
Goal: learn a model for the inputs and then apply it to a specific machine learning task.

2.2 Neural networks

An artificial neural networks is one of the most developed and widely used algorithms of machine learning. It is the mathematical model of brain's activity that is able to tackle both problems of classification and regression. Neural network can function as a model of supervised, unsupervised or reinforcement learning.

[1] Formal definitions of supervised and unsupervissed learning problems are inspired by [15] and [13] respectively

Definition Simon Haykin [14] offers the following definition:

> A neural network is a massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experiential knowledge and making it available for use. It resembles brain in two respects:
>
> 1. Knowledge is aquired by network from its environment through a learning process.
> 2. Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.

Since their invention in 50-s neural networks have been used to model human brain and approach the goal of creating human-like artificial intelligence. Nowadays it is more common to think of neural networks as of the statistical models that perform well on some extremely complicated tasks. For example, Hastie et. al. [18] view neural networks as nonlinear statistical models, the two-stage regression or classification models. David MacKay [9] sees them as parallel distributed computational systems consisting of many interacting simple elements. And Goodfellow et. al. (MIT) [11] write the following

> Modern neural network research is guided by many mathematical and engineering disciplines, and the goal of neural networks is not to perfectly model the brain. It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function.

2.3 Model of a neuron

A biological neural network (brain) consists of cells called neurons. Human brain is composed of about 10 billion neurons, each connected to about 10,000 other neurons. The same applies to artificial neural network - they consists of many artificial neurons - mathematical models of biological ones. I will start this section by describing the structure of a biological neuron. Then I will provide a formal description of an artificial neuron as a mathematical model.

Biological neurons According to the medical definition, provided by Merriam-Webster dictionary[2],

> Neuron is a cell that carries messages between the brain and other parts of the body and that is the basic unit of the nervous system.

Figure 2.1 shows the simplified schematic of a biological neuron. Each neuron consists of a cell body, or soma, that contains a cell nucleus. Branching out from the cell body are a number of fibers called dendrites and a single long fiber called the axon. The axon stretches out for a long distance, much longer than the scale in this diagram indicates. Typically, an axon is 1 cm long (100 times the diameter of the cell body), but can reach up to 1

[2] http://www.merriam-webster.com/dictionary/neuron

meter. A neuron makes connections with 10 to 100,000 other neurons at junctions called synapses. Signals are propagated from neuron to neuron by a complicated electrochemical reaction. The signals control brain activity in the short term and also enable long-term changes in the connectivity of neurons. These mechanisms are thought to form the basis fur learning in the brain. Each neuron receives electrochemical inputs from other neurons at the dendrites. If the sum of these electrical inputs is sufficiently powerful to activate the neuron, it transmits an electrochemical signal along the axon, and passes this signal to the other neurons whose dendrites are attached at any of the axon terminals. It is important to note that a neuron fires only if the total signal received at the cell body exceeds a certain level. The neuron either fires or it doesn't, there aren't different grades of firing.

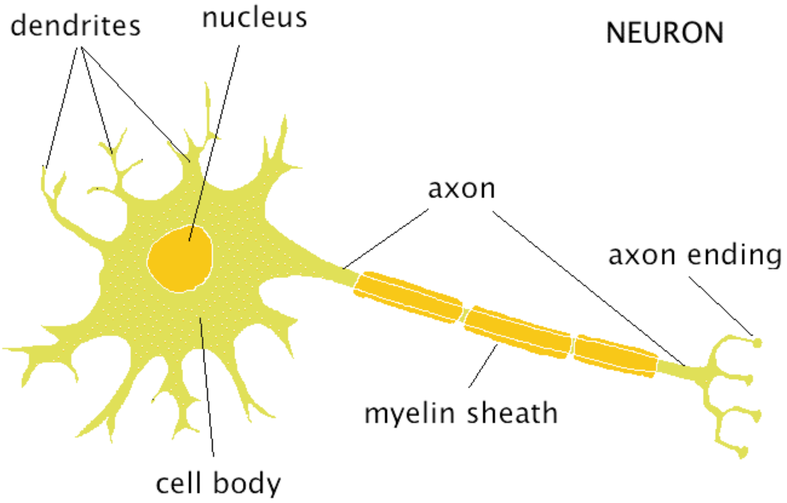

Figure 2.1: A schematic of biological neuron

Artificial neurons In fact, artificial neuron can be viewed as a parametric function of n inputs $g_w : X^n \to Y$, where X is the space of all possible input values, Y - the space of output values and w - the vector of numeric weights $w_1, w_2, ..., w_n$, the tunable parameters that are being updated as the network learns. In other words, function g_w is defined with the rule

$$y = g_w(x) \tag{2.1}$$

where $x \in X^n$, $y \in Y$ and $w \in \mathbb{R}^n$.
Function g_w is a superposition of two functions, $s_w : X^n \to \mathbb{R}$ and $f : \mathbb{R} \to Y$:

$$g_w(x) = f(s_w(x))$$

where s_w is defined as follows:

$$s_w(x) = \sum_{i=0}^{n} w_i x_i$$

f is a nonlinear activation function in case of classification and an identity function $\forall x f(x) = x$ in case of regression.

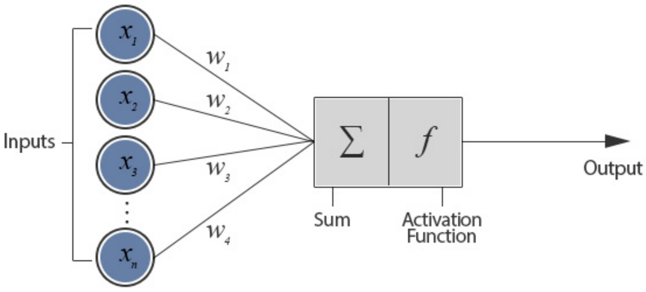

Figure 2.2: Model of an artifitial neuron[3]

2.4 Structure and representation

Graphical representation Neural networks are composed of nodes (neurons), connected by direct links (synaptic connections). If we think of neurons as vertices and synaptic connections as edges, neural networks can be represented by weighted directed graphs called **network diagrams**.
If the network is feedforward (without cycles), the graph will be k-partide, where k is the number of layers. If it is also fully-connected, it will be a complete k-partide graph. See Figure 2.3 for specific examples.

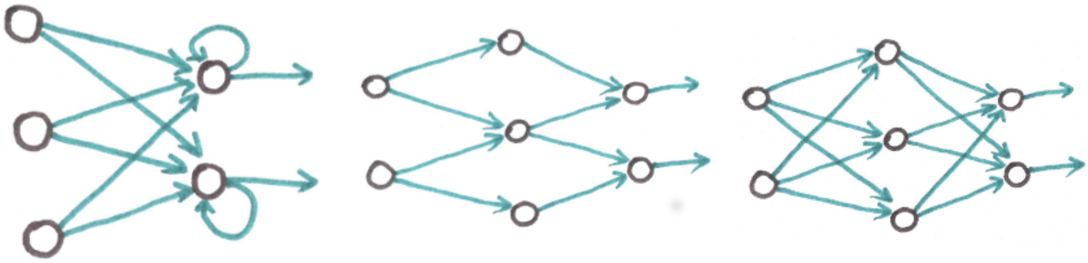

Figure 2.3: (1) Recurrent neural network represented by directed graph. (2) 3-layered feedforward neural network represented by 3-partide directed graph. (3) 3-layered fully-connected feedforward neural network represented by a complete 3-partide directed graph

Network topologies As it was already said, neural networks consist of neurons. These neurons are connected with directed links (synaptic connections) with numeric weights that determine the strength and sign of the connection.
Neurons are grouped into layers. First layer is called **input layer**, the last one - **output layer**. All the layers between input and output layers are called **hidden layers**. Number of hidden layers is one of the tunable metaparameters that define the architecture of a

[3] Source: `http://www.theprojectspot.com/tutorial-post/introduction-to-artificial-neural-networks-7`

neural network. According to [18], typically the number of hidden units is somewhere in the range of 5-100.
To satisfy the linear model of a regression each layer, except for the output one, has an additional bias unit $b = 1$.
For a specific example of neural network architecture see Figure 2.4.
By the type of connections neural network can be either feedforward or recurrent:

- **Feedforward network** - has connections only in one direction (outputs of neurons from layer k can be connected only to neurons of layers $k + c$ where $c > 0$). The network diagram of a feedforward netwond forms a directed acyclic graph.

- **Recurrent network** - feeds its outputs to its own inputs. This network has at least one cycle (at least one connection from neuron in layer k to neuron in layer $k - c$ where $c \geq 0$).

If in neural network with N layers $\forall k : 0 \leq k \leq N - 1$ every neuron in layer k is connected to all the neurons of layer $k + 1$, the network is called **fully-connected**.
Figure 2.4 depicts the schematic of a feedforward neural network with one hidden layer.

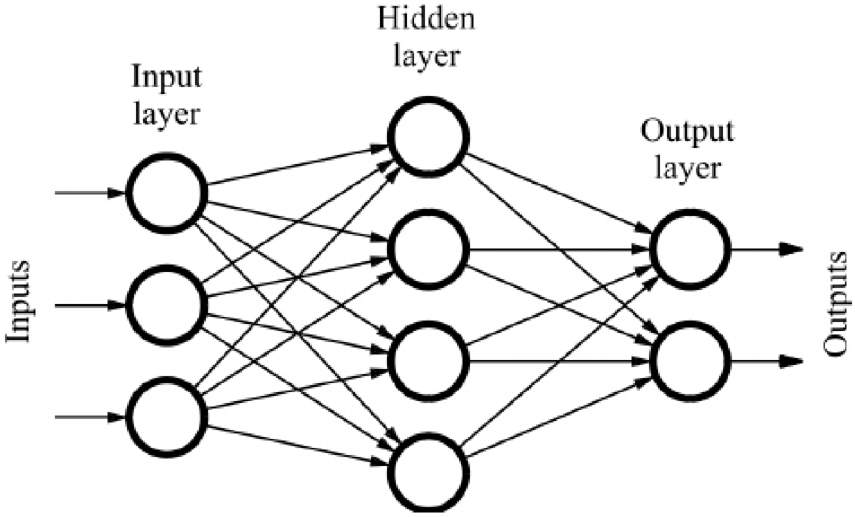

Figure 2.4: Feedforward artificial neural network with one hidden layer. Number of neurons in each layer: 3 (2-dimentional input + bias unit) in the input layer, 4 in the hidden layer, 1 in the output layer (1-demensional output)[5]

2.5 Training a neural network

In this section I will describe gradient descent and the backpropagation learning algorithm. One of the most important features of backpropagation, which makes it so popular is that it can be easily parallelized.

[5] Source: https://www.researchgate.net/figure/234055177_fig1_Figure-61-Sample-of-a-feed-forward-neural-network

2.5.1 Gradient descent

The most widely used algorithm for training neural networks (and optimizing many other machine learning problems) is called gradient descent. It is based on the fact that derivative (gradient) of a function characterizes how how it grows:

- The sign of a derivative over some parameter denotes the direction in which the function grows
- The ablsolute value of derivative tells us how fast the function grows

Therefore, derivative of a function at some point can tell us in which direction and how far should we move to minimize the function.
This helps us construct an algorithm for function minimization which is based on adding some gradient to the value of parameter.
Let's say we have a cost function $J(t; w)$ that depends on the target output t that we expect our model to produce, and the tunable parameter w (weight). To minimize this function we change the weight w by the value of gradient $-\eta\frac{\partial}{\partial w}J(t; w)$ until the function reaches the minimum (gradient becomes equal to 0.

$$w = w - \eta\frac{\partial}{\partial w}J(t; w) \tag{2.2}$$

The parameter η is called the learning rate. It controlls the speed of learning. If η is big, we are making big steps in the directions in which our function descends. This means that the model learns faster, but also creates the risk of overshooting - we can jump too far to the point where the function increases again. This may result in divergence - the situation when our model keeps getting worse instead of getting better. Another problem arises when the η is too small. First of all, such learning algorithm could take unreasonable amount of time to converge. And if the cost function that we are trying to minimize is nonlinear, we may get stuck in a local minimum. Choosing a good leraning rate is one of the hardest tasks of machine learning.

2.5.2 Backpropagation

As the number of hidden layers grows, the problem arises, as we can only compute the error of output layer by finding the deviation of hypothesis h_w from the desired output y. The fitness function for regression is the sum of squared errors

$$J(w) = \sum_{k=1}^{K}\sum_{i=1}^{n}(y_i - f_k(x_i))^2 \tag{2.3}$$

For classification it is defined as

$$J(w) = \sum_{i=1}^{n} y_{ik}\log f_k(x_i \tag{2.4}$$

The task of learning is to minimize the fitness function $J(w)$. There are different learning algorithms for doing that. In this paper I will explain only the idea of the classical backpropagation algorithm.

Backpropagation is an abbreviation for "backward propagation of errors". According to [18], backpropagation is a two-pass procedure, used to compute the gradients for the updates in gradient descent algorithm:

- **Forward pass** -the current weights are fixed and the predicted values are computed
- **Backward pass** - the errors δ_{ki} are computed and then backpropagated to give errors s_{ij}. Both sets of errors are then used to compute the gradients for updates.

The main advantage of backpropagation is that it has local nature, and thus it can be efficiently implemented on a parallel architecture computer.

Chapter 3

Single-layer perceptron

In this chapter I will describe my implementation of a single-layer perceptron in Pharo. It will support multiclass classification (one or many neurons). Each neuron will be implemented as an object. The code of this project can be found is my NeuralNetwork repository on Smalltalkhub[1].

I will start by illustrating the design issues and different approaches to the implementation of every single part of this project. It will be quite long, so here is my final design:

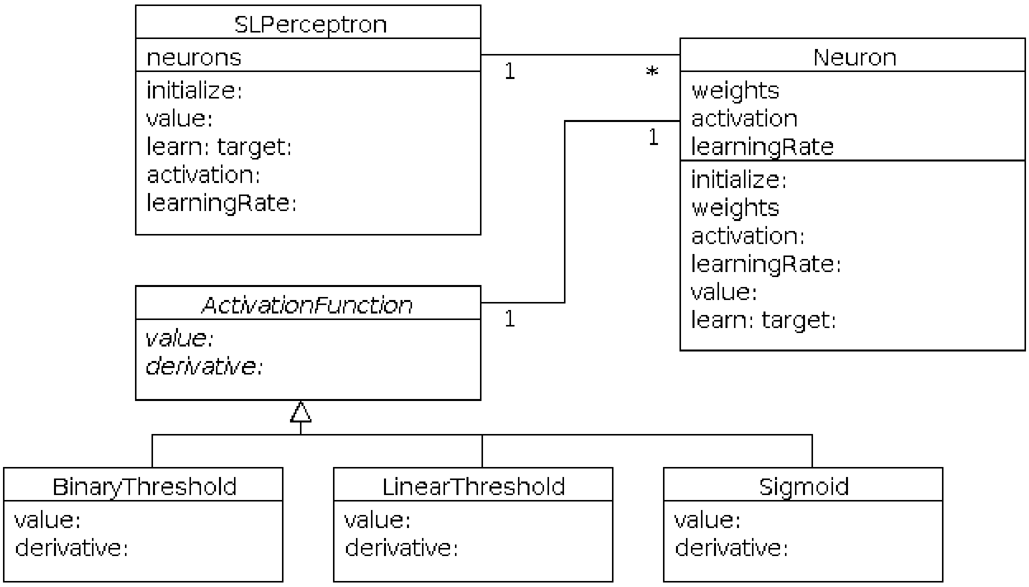

Figure 3.1: Object-oriented design of a single-layer perceptron

3.1 What is a perceptron?

First of all, we need to define a perceptron. It is the most basic form of an artificial neural network, still, most people fail to clearly define what it actually is.

[1] http://smalltalkhub.com/#!/ Oleks/NeuralNetwork

For now I will refer to a perceptron as an artificial neural network that follows the perceptron learning procedure.
This definition implies some restrictions to what perceptrons are and what can they do.

3.1.1 Restrictions of perceptrons

- They can only converge on a linearly-separable input (see XOR problem, Minski & Pappet). But if the input is linearly-separable, the perceptron is guaranteed to converge on it, regardless of the initial weights and learning rate (see Perceptron Convergence Theorem, proven in 1962 by Block and Novikoff)

- Perceptrons (as defined above) can have only one layer of neurons. The thing is (week 3 of the Neural Networks for Machine Learning course by Geoffrey Hinton), the perceptron learning procedure can only be applied to a single layer of neurons. Neurons in hidden layers would need some sort of feedback in order to calculate their errors and update the weights. That's why we need a different learning algorithm (e. g. backpropagation, which will be implemented on the next stage).

- Perceptrons can only learn online (one example at a time). That's because the perceptron learning is based on adding (or subtracting) the input vector to the weights based on the error of a binary classifier (this error can be -1, 0, or 1).

3.2 Design issues

3.2.1 How to represent weights?

When it comes to the object-oriented implementation of neural networks, this is probably the most important question that has to be answered. Should weights belong to the neuron? If yes, should it be the sending or the receiving neuron? Or maybe they should belong to a layer? Or maybe to the whole network? Maybe we should even implement them as separate objects?
Being a feedforward network with only one layer, and therefore having no weights that connect two neurons, single-layer perceptron simplifies this problem. Basically, we have three options:

1. Input weights of each neuron are stored as a vector inside that neuron.

2. The matrix of all input weights is stored in the network.

3. The weights are implemented as objects and connected to neurons.

Second option is the most efficient (vector-matrix multiplication), but not very object-oriented. What is neuron in this implementation? Clearly, the network is just a matrix of weights + some learning rules. Should the neuron be an activation function with a learning rate? But then again, storing them in the network would be even more efficient. So basically, we don't need a Neuron class. All we need is a matrix and several functions for manipulating it. That doesn't sound object-oriented to me.

Third option in this case would be an over-engineering. It would just make the whole thing way more complicated. Implementing weights as objects would probably make some sense in a multi-layer neural network, where each weight is a connection between two neurons (we can think of inputs as of fake neurons). It connects two neurons, sends a signal between them and has a "strength" which can be updated. As a result, neurons would not know about other neurons. They would just receive, process, and emit the signals. I assume that such implementation would not be very fast, but it could be used for modeling purposes. I will write more about this idea in a post dedicated to multi-layer networks.
First option looks like most appropriate for single-layer perceptrons. And it's very easy to implement, so I'll stick to it.

3.2.2 Activation functions

There are two ways of representing activation functions in this project:

1. Implement them as methods
2. Implement them as classes

The first approach will be faster and consume less memory. We create a base class Neuron with abstract methods activation and activationDerivative. Each subclass will be a special type of neuron, such as BinaryThresholdNeuron, SigmoidNeuron, that implements a corresponding activation function.

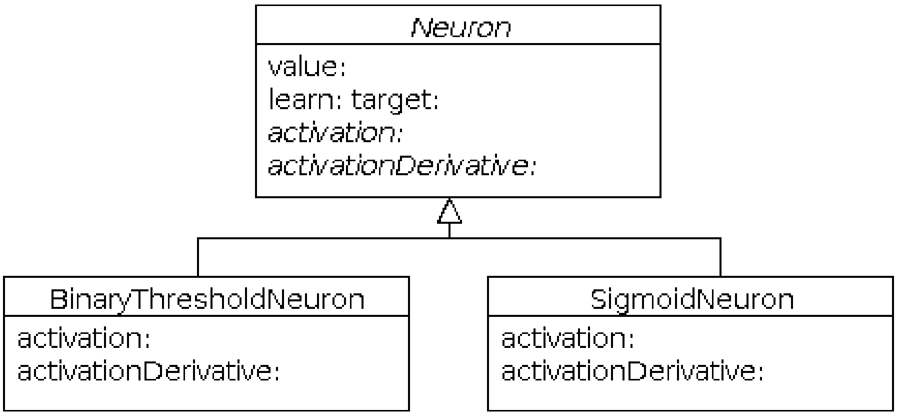

Figure 3.2: Implementing activation functions as methods of specific subclasses on Neuron

Another way of implementing activations is to create a base class ActivationFunction with two abstract methods, value: and derivative:. This approach is more flexible, because if someone wants to use a new activation function, he will be able to implement it as a subclass, only defining what it is and what is its derivative. Then he will be able to pass an object of this class to an existing neuron. It doesn't seem logical to reimplement the whole neuron every time we need to create a function.

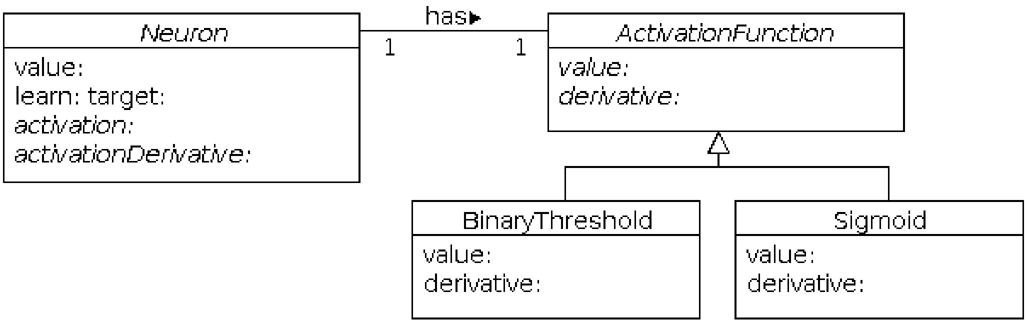

Figure 3.3: Implementing activation functions as separate classes

So the real question can sound like this: *Are neurons defined by their activations? Does having a different activation means being a completely different type of neuron?*

3.2.3 Shared or separate activation and learning rate?

Both activation and learning rate can be either shared by all neurons of a perceptron or be stored separately in each neuron.The question is: do we need to have neurons with different activations and different learning rates?

Let's assume that we don't. Indeed, in most cases all neurons of a network (or a layer) share the same learning rate and have the same activation. If the network has many neurons (and most networks do), then we will be storing the same number just as many times. And if the activation function is implemented as a class, then we will be creating a separate instance of that class for each neuron.

However, if we wanted to parallelize the computations done by neurons, it would be better to have a separate learning rate and separate activation for each neuron (or each block of neurons). Otherwise they will just block each other trying to access the shared memory on every single step. And besides, the total memory occupied by this "heavy" neuron would still be rather small. I think, such neuron (or a group of them) would easily fit in the local memory of a single core of GPU.

But single-layer perceptrons do not usually perform heavy computations. They are more useful for the modeling purposes. That's why we should probably take the "separate" approach and allow the user to build a network out of completely different neurons (like building blocks).

By the way, for multi-layer network the nice idea would be to share the same activation and learning rate inside one layer, but allow user to have completely different layers. In the end, he should be able to build some complicated network like the convolutional network on the picture. But that's not the topic of this post.

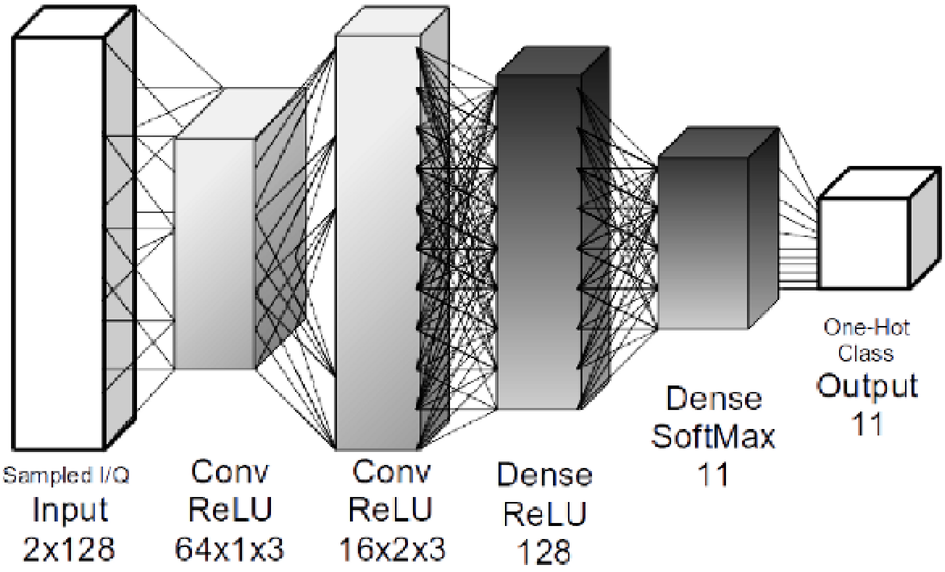

Figure 3.4: A convolutional neural network with four hidden layers

3.2.4 Data shuffling

Online perceptrons are sensitive to the order in which training examples are received. Weight updates are made after each training example, that's why the training vectors #(#(0 1) #(1 1)) and #(#(1 1) #(0 1)) will result in different weight vectors. Depending on the order of examples, the perceptron may need a different number of iterations to converge.
That's why, to test the complexity of such learning, the perceptron has to be trained by examples randomly selected from a training set.

3.3 Implementation

Putting it all together, here is my design of a single-layer peceptron:

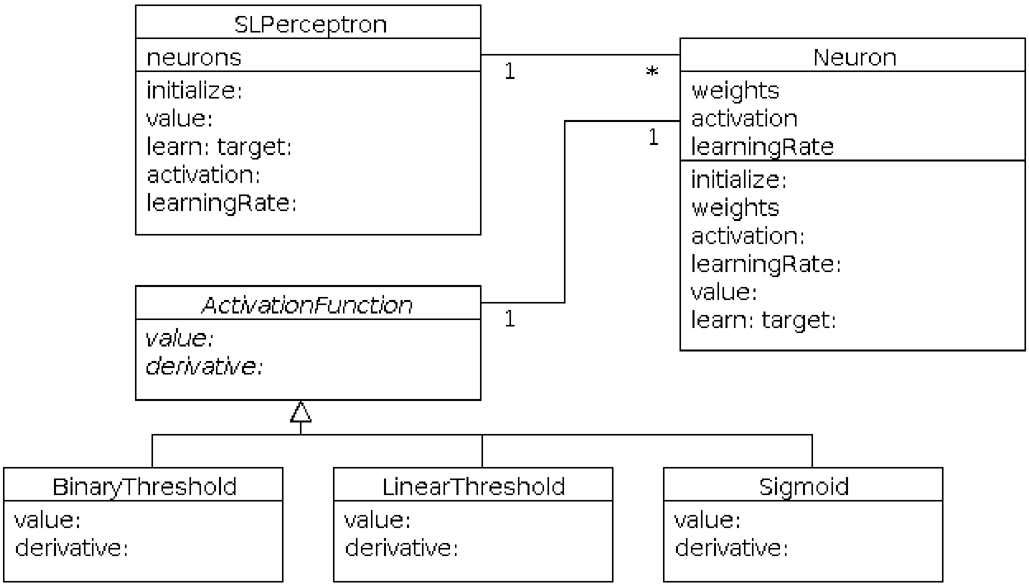

Figure 3.5: Object-oriented design of a single-layer perceptron

3.3.1 Neuron class

```
Object subclass: #Neuron
  instanceVariableNames: 'weights activation learningRate'
  classVariableNames: ''
  package: 'NeuralNetwork'
```

The weights are initialized with random numbers in range [0, 1]. I'm not sure if this is a good range, but on simple examples it works just fine.
BinaryThreshold is a default activation function and the default learning rate is 0.1. These parameters can be changed using the accessors activation: and learningRate:.

```
initialize: inputSize
  "Creates a weight vector and initializes it with random values. Assigns default values to activation and
    learning rate"

  activation := BinaryThreshold new.
  learningRate := 0.1.

  weights := DhbVector new: (inputSize + 1).

  1 to: (inputSize + 1) do: [ :i |
    weights at: i put: (1 / (10 atRandom))].
  ^ self
```

We will also need to prepend 1 as a bias unit to every input vector.

```
prependBiasToInput: inputVector
  "this method prepends 1 to input vector for a bias unit"
  ^ (#(1), inputVector) asDhbVector.
```

According to “Numerical Methods” book, each function should implement the value: method. I want to emphasize that from the mathematical point of view neuron is a function.

Though the inner representation uses DhbVector, I want a user to write something like perceptron value: #(1 0). instead of perceptron value: #(1 0) asDhbVector.

```
value: inputVector
  "Takes a vector of inputs and returns the output value"

  | inputDhbVector |
  inputDhbVector := self prependBiasToInput: inputVector.
  ^ activation value: (weights * inputDhbVector).
```

We need accessors for setting the learning rate an activation. I also added a simple accessor for weights for debugging purposes. All these accessors are trivial, so I will not put the code here.

And of course, the perceptron learning rule.

```
learn: inputVector target: target
  "Applies the perceptron learning rule after looking at one training example"

  | input output error delta |
  output := self value: inputVector.
  error := target - output.

  input := self prependBiasToInput: inputVector.

  delta := learningRate * error * input *
    (activation derivative: weights * input).
```

3.3.2 SLPerceptron class

Single-layer perceptron (according to my design) is a container of neurons. The only instance variable it has is the neurons array.

```
Object subclass: #SLPerceptron
  instanceVariableNames: 'neurons'
  classVariableNames: ''
  package: 'NeuralNetwork'
```

To create an instance of SLPerceptron we need to specify the size of the input vector and the number of classes which equals to the number of neurons in our perceptron (multiclass classification).

```
initialize: inputSize classes: outputSize
  "Creates an array of neurons"
  neurons := Array new: outputSize.

  1 to: outputSize do: [ :i |
    neurons at: i put: (Neuron new initialize: inputSize). ]
```

The output of a single-layer perceptron is a vector of scalar outputs from each neuron in the layer.

```
value: input
  "Returns the vector of outputs from each neuron"
  | outputVector |

  outputVector := Array new: (neurons size).

  1 to: (neurons size) do: [ :i |
    outputVector at: i put: ((neurons at: i) value: input) ].

  ^ outputVector
```

If we ask SLPerceptron to learn, he will pass that request to all his neurons (basically, SLPerceptron is just a container of neurons that provides an interface for manipulating them).

```
learn: input target: output
  "Trains the network (perceptron) on one (in case of online learning) or multiple (in case of batch learning
    ) input/output pairs"

 1 to: (neurons size) do: [ :i |
   (neurons at: i) learn: input target: (output at: i) ].
```

3.4 Testing

I test my SLPerceptron with BinaryThreshold activation function on 4 linearly-separable logical functions: AND, OR, NAND, and NOR, and it converges on all of them.
Here is the test for AND function. Other 3 look exactly the same (only the expected output values are different).

```
testANDConvergence
  "tests if perceptron is able to classify linearly-separable data"
  "AND function"

  | perceptron inputs outputs k |
  perceptron := SLPerceptron new initialize: 2 classes: 1.
  perceptron activation: (BinaryThreshold new).

  "logical AND function"
  inputs := #(#(0 0) #(0 1) #(1 0) #(1 1)).
  outputs := #(#(0) #(0) #(0) #(1)).

  1 to: 100 do: [ :i |
    k := 4 atRandom.
    perceptron learn: (inputs at: k) target: (outputs at: k) ].

  1 to: 4 do: [ :i |
    self assert: (perceptron value: (inputs at: i)) equals: (outputs at: i) ].
```

And this test (or rather a demonstration) shows that single-layer perceptron can not learn the XOR function (not linearly-separable).

```
testXORDivergence
  "single−layer perceptron should not be uneble to classify data that is not linearly−separable"
  "XOR function"

  | perceptron inputs outputs k notEqual |
  perceptron := SLPerceptron new initialize: 2 classes: 1.
  perceptron activation: (BinaryThreshold new).

  "logical XOR function"
  inputs := #(#(0 0) #(0 1) #(1 0) #(1 1)).
  outputs := #(#(0) #(1) #(1) #(0)).

  1 to: 100 do: [ :i |
    k := 4 atRandom.
    perceptron learn: (inputs at: k) target: (outputs at: k) ].

  notEqual := false.

  1 to: 4 do: [ :i |
    notEqual := notEqual or:
      ((perceptron value: (inputs at: i)) ~= (outputs at: i)) ].

  self assert: notEqual.
```

I was also trying to test the Sigmoid function, but that test failed. This means that either perceptrons (as defined at the beginning of this post) can not have sigmoid as their activation, or that I don't have a good enough understanding of how to implement a perceptron with sigmoid.

Chapter 4

Multi-layer neural network

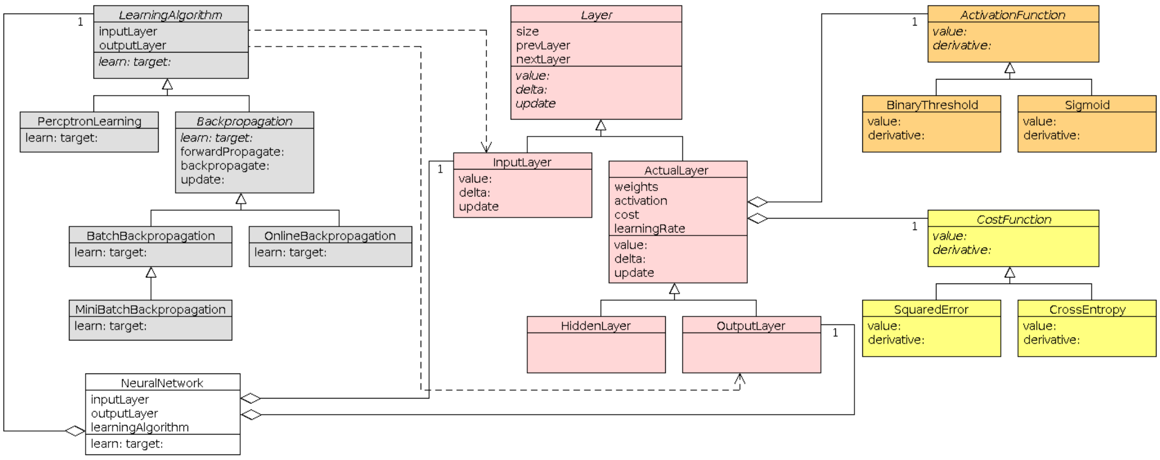

Figure 4.1: Implementing activation functions as methods of specific subclasses on Neuron

4.1 Why not ”multi-layer perceptron”?

When writing about neural networks, most authors start by describing the single-layer perceptron, showing that it is restricted to the problems that are linearly separable. Then they introduce us to a concept of a much more powerful feedforward neural network, called a multi-layer perceptron. There is one big problem though with using a word ”perceptron”. Just like ”neural networks”, this term is extremely misleading. Most of the time people mean different things when they talk about perceptrons. In some literature perceptron is a neural network, in other - it is a single neuron.

Personally, I like the view of Geoffrey Hinton, expressed in his famous MOOC: perceptron is an artificial neural network that learns by following perceptron learning procedure.

This procedure is based on a concept of error, which can only be calculated for a single output layer of neurons. Because the error is sort of a difference between the expected output, and the one that was received:

$$\text{error} = (\text{expected output}) - (\text{actual output})$$

But what is the expected output of a hidden layer? We can't tell by just looking at the data. Finding this error would require us to solve the problem of optimisation. We can't just update the weights using the simple error-based rule. Therefore, the above definition implies that perceptrons can only have one layer of neurons.
Therefore, in this work, I will be referring to perceptrons as to the single-layer neural networks that learn by updating their weights according to the perceptron learning procedure. Feedforward neural networks with two or more layers of neurons will be called the multi-layer neural networks.
Another thing that makes perceptrons special is the fact that they are guaranteed to converge on a linearly separable data. This is not the case with multi-layer networks
Many famous software libraries share this terminology. For example, scikit-learn with its MLPClassifier (Multi-Layer Perceptron Classifier).

4.2 Design issues

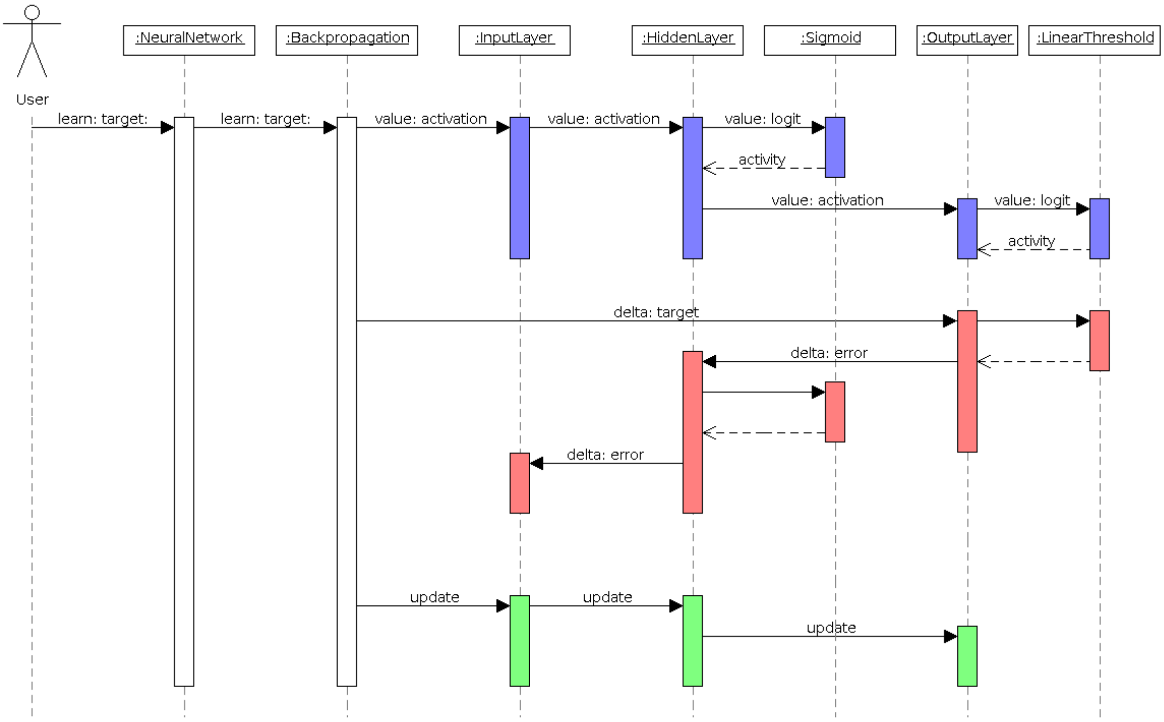

Figure 4.2: Implementing activation functions as methods of specific subclasses on Neuron

4.2.1 LearningAlgorithm class: Strategy Pattern

Backpropagation is the most widely used algorithm for training multi-layer neural networks. But not the only one. There are many other algorithms available, such as… . Though, at this point I will not be implementing anything except backpropagation, I don't want to force it onto the user and make him redesign half of the framework in order to use some other algorithm.

But even backpropagation comes in different forms. It can be update the weights after each training example (online) or after looking at a whole dataset (batch). The most efficient though is the one that looks at smaller subsets of a training data before updating the weights (mini-batch). Generally speaking, these are types of learning, and they don't affect the idea of backpropagation. But the learning algorithm as a sequence of actions changes a lot:

> Online: forward - backward - update - forward - backward - update ... Batch: forward - backward - forward - backward - ... - update

Another way in which implementations of backpropagation can be different is how the learning rate is chosen. The easiest solution is to let the user pick a constant learning rate, and keep it unchanged for the whole period of learning. It is not a very good approach, because:

1. It can be very hard to choose a good learning rate. If it is too small, the learning will be too slow - the network will need thousands of epochs to reach some relatively good performance. If the learning rate is too big, the algorithm will eventually start diverging - the cost will be growing, meaning that the performance is only getting worse.

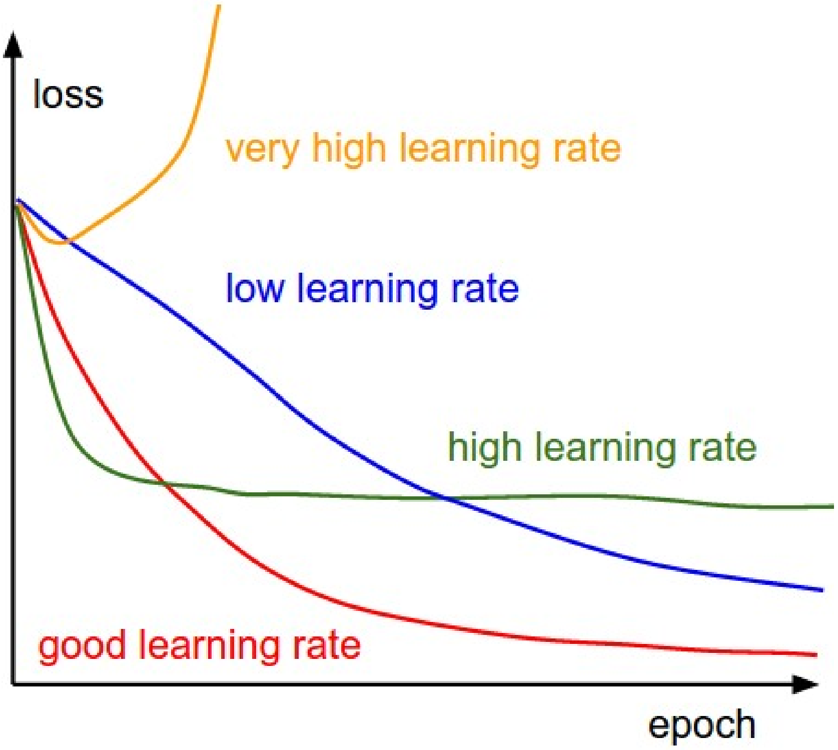

Figure 4.3: The demonstration of how the change of a learning rate affects the process of learning

2. (Batch learning) At the beginning, when we start from some random position in a weight space, we can be really far from the desired optimum. We want our learning

rate to be big enough, so that we can get relatively close to that optimum in as little steps as possible. But as we get closer, we want to reduce our learning rate, in order to approach the optimum more slowly. Otherwise we may "jump too far" and start diverging.

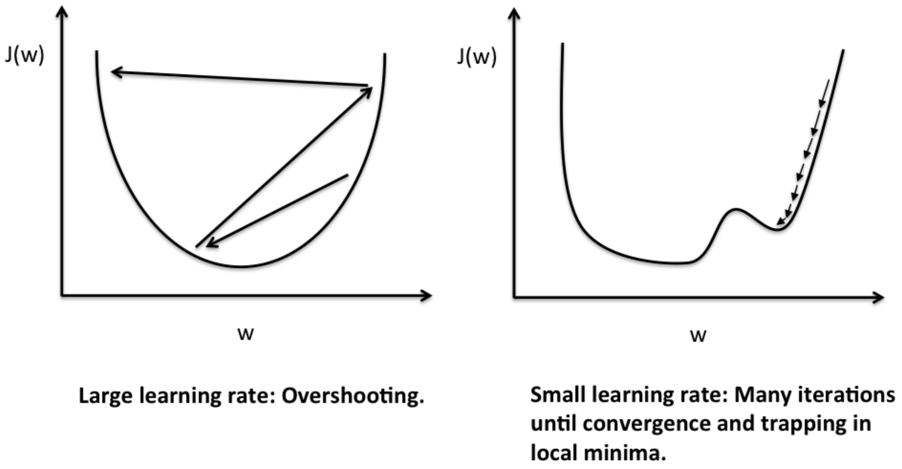

Figure 4.4: Different types of problems when choosing a learning rate

3. When it comes to multi-layer neural networks, the surface of a cost function in a multidimensional weight space that we are trying to optimize, becomes very complex and highly nonlinear. It may have many local optima in which we may get stuck with a small learning rate. We may need an algorithm that would increase the learning rate every time we get into the local optimum.

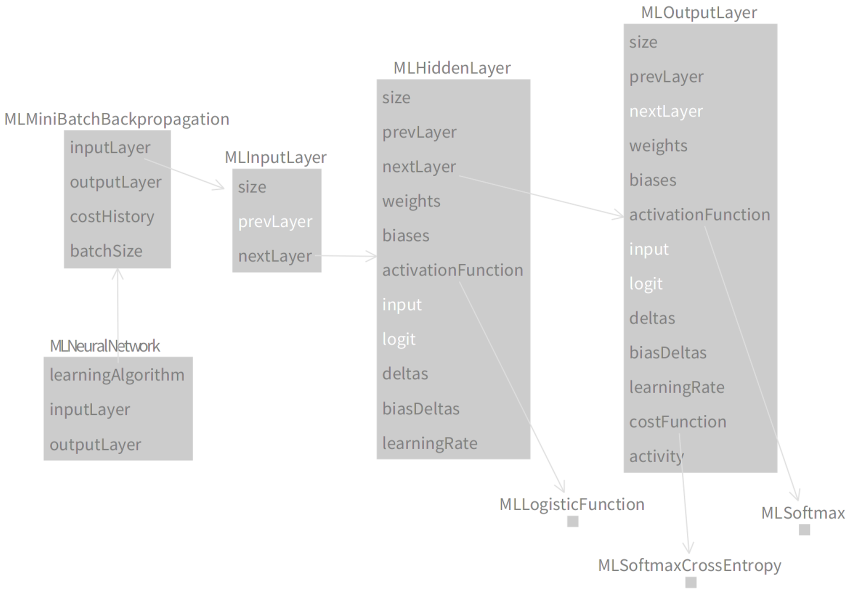

Figure 4.5: The graph of connections between the objects. Not all connections are visualized

4.2.2 Weight initialization

Neural network is a machine learning algorithm that changes its weights to improve the performance on a given dataset. Each configuration of weights represents a point in the multidimentional space. The process of learning is based on minimizing the cost function, defined on that weight space. Therefore, setting the initial weights means choosing the starting point for the process of minimization. If the initial weights are close enough to the global minimum (best set of weights), the learning will be fast. If they are far - the network will take a lot of time to learn, or fail to converge at all.
Initializing all weights with zeroes is the easiest thing to do. However, a network with such weights may never learn anything. All weight updatades Δw will be equal to zero.
It is recommended to initialize the weights with a small random values. Initializing a network with random weight can be a massive boost in performance. Pinto et al.[19] evaluated thousands of architectures on a number of object recognition tasks and found that random weights performed only slightly worse than pretrained weights. And [5] showed that while random weights are no substitute for pretraining and finetuning, their use for architecture search can improve the performance of state-of-the-art systems.
According to [1], a good way to initialize the weights of a layer L is by choosing random values from range $\left[-\frac{1}{2k}, \frac{1}{2k}\right]$, where k is the number of neurons in layer L.

Chapter 5

Handwritten Digit Recognition

In this chapter I will describe the application of my NeuralNetwork to the task of digit recognition.

5.1 Getting and normalizing the data

In this section I will explain where to get the data for digit recognition and how to read it into the system, storing it in the objects of Dataset class, described in the previous chapters.

5.1.1 Understanding the importance of data normalization

The first thing I did after creating this package was trying to use it for digit recognition on MNIST. The single-layer network with 784 input and 10 output neurons worked just as expected (in the following sections I will talk about it in more details), but when I training a two-layer neural network (784 input neurons, 500 neuron in the hidden layer, and 10 output neurons), the only output it was able to produce were vectors of $Floatnan$ values. I used Pharo debugging and inspecting tools, wrote several tests and found out that the weight update, performed after one mini-batch training epoch with 100 training examples, set some of the weights to be equal to Float nans. All the algebraic operations applied to NaNs return NaNs. So after few more steps, all the weights turn into NaN values, and the network produces only NaN values.

What is the reason of this? The errors of this kind are pretty hard to fix, because it's hard to find out what causes them. But Pharo provides advanced debuging tools that allow you to inspect the state of any object in your system at every stage of your algorithm. After inspecting the deltas (accumulated values used for weight updates) I realised that they grow (or decrease) very rapidly and have a slight probability of overflowing the Float (after 10,000 steps even a slight probability of overflow makes it very probable that at least one weight gets turned into Float nan, and this will result in all the other weights becoming NaNs as well).

There are many ways of preventing this from happening. But the easiest one is to normalize the inputs by converting each pixel value from Integer in range [0, 255] to Float in [0, 1] range. This can be done using the following formula

$$v := \frac{v - min}{max - min}$$

It will convert any collection of values into [0,1] range, preserving the distribution and all the statistics. In our case we just have to divide every pixel value by 255.

5.1.2 MNIST database of handwritten digits

The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. You can download it here: `http://yann.lecun.com/exdb/mnist/`.
The website provides 4 files:

- **train-images-idx3-ubyte.gz** - training set images (9912422 bytes)
- **train-labels-idx1-ubyte.gz** - training set labels (28881 bytes)
- **t10k-images-idx3-ubyte.gz** - test set images (1648877 bytes)
- **t10k-labels-idx1-ubyte.gz** - test set labels (4542 bytes)

5.1.3 Reading the data

As it was said in the previous chapters, objects of MLNeuralNetwork class accept the Dataset object as an input. Dataset stores input and output values, used for supervised learning, and provides an interface for

- normalizing the input values
- getting all the input data as an array of PMVectors (in case of MNIST, these will be 28x28 images unrolled into a vector with 784 values)
- getting all the output data as an array of PMVectors (each vector storing the expected output values of all the output neurons)
- getting a random input/output pair
- getting a random subset of a given size

A single example input of a neural network has to be a vector. Therefore, we have to unroll the 28x28 images provided by MNIST databese into the vectors with 784 numbers, representing the colors of pixels. Example output must be a vector of numbers, representing the desired output value of each neuron in the output layer. The best way of doing this is converting the digits 0-9 into one-hot vectrors containing the bits, of which only one can be equal to 1.

$$\begin{array}{c}
0 \longrightarrow 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
1 \longrightarrow 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
2 \longrightarrow 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
\dots \\
9 \longrightarrow 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1
\end{array}$$

Therefore, after reading the data from idx files, we must:

1. unrol the images, represented by matrices of numbers, into vectors
2. turn labels into one-hot vectors

Creating MnistReader

Let's define a MnistReader class that will read the MNIST database from idx files into two Dataset objects: training dataset and test dataset.

```
unroll: arrayOfArrays
  "Unrolls the (m, n) matrix into a vector of size m*n"

  | rows cols vector |
  rows := arrayOfArrays size.
  cols := (arrayOfArrays at: 1) size.

  vector := PMVector new: (rows * cols).

  1 to: rows do: [ :i |
    1 to: cols do: [ :j |
      vector at: ((i - 1) * cols + j)
        put: ((arrayOfArrays at: i) at: j) ]].

  ^ vector
```

```
onehot: digit
  "Turns a digit (0-9) into a one-hot vector. A PMVector with 10 numbers, all of which are 0, except for
    the one that represents the given digit. Because indexing in Smalltalk starts from 1, the index of 1 will
    be determined by adding 1 to the value of digit"

  | onehot |
  onehot := PMVector new: 10.

  1 to: 10 do: [ :i |
    onehot at: i put: 0 ].

  onehot at: (digit + 1) put: 1.

  ^ onehot
```

Now we define a method that will read MNIST images from a given file, unroll them using the method defined above, and return an array of unrolled images.

```
readImages: path
  "Reads MNIST images from an idx file specified by path using IdxReader. Retuns a matrix of images
    unroled into PMVectors with 784 elements"

  | reader matrix |
  reader := IdxReader onStream: (File named: path) readStream.
  matrix := reader next.

  ^ matrix collect: [ :v | self unroll: v ].


readLabels: path
  "Reads MNIST labels from an idx file specified by path using IdxReader. Returns an array of one-hot
    PMVectors"

  | reader array |
  reader := IdxReader onStream: (File named: path) readStream.
  array := reader next.

  ^ array collect: [ :v | self onehot: v ].


readTrainFrom: basePath
  "Reads MNIST images and labels into a test dataset. This method assumes that the files are named
    exactly the same as on Yan LeCun's website and are located in a folder identified by basePath"

  | images labels dataset |
  images := self readImages: basePath, 'train-images.idx3-ubyte'.
  labels := self readLabels: basePath,  'train-labels.idx1-ubyte'.

  dataset := MLDataset new input: images output: labels.
  ^ dataset normalizeInputs.


readTestFrom: basePath
  "Reads MNIST images and labels into a training dataset. This method assumes that the files are named
    exactly the same as on Yan LeCun's website and are located in a folder identified by basePath"

  | images labels dataset |
  images := self readImages: basePath, 't10k-images.idx3-ubyte'.
  labels := self readLabels: basePath,  't10k-labels.idx1-ubyte'.

  dataset := MLDataset new input: images output: labels.
  ^ dataset normalizeInputs.
```

We can use our MnistReader like this

```
base := '/home/user/data/mnist/'.
trainData := MnistReader readTrain: base.
testData := MnistReader readTest: base.
```

Now we have two Dataset objects that can be used to train a neural network and evaluate the accuracy. In the following sections I will show how this can be done.

5.2 Training a neural network

Let's train a neural network to classify these digits, using 60,000 input/output examples in `trainData` dataset for supervised learning and evaluate the performance on 10,000 examples in the `testData` dataset.

5.2.1 Choosing the network architecture

It would be a good idea to use some well-known architecture that was trained on other platforms. This way we can easily compare the performance. Here are the two architectures I will be working with in this chapter

1. Single-layer neural network with 784 inputs and 10 neurons. All neurons have softmax activation function. The learning rate is constant and set to 0.5. The network uses Cross-entropy cost function

1Experiment 1 2Experiment 2

	1	2
Architecture	784, 10	784, 500, 10
Activations	Softmax	Logistic, Softmax
Cost	Cross-entropy	Cross-entropy

Table 5.1: Table to test captions and labels

	1	2
Batch size	100	100
Epochs	1000	500
Learning rate	0.5	0.1

Table 5.2: Table to test captions and labels

The results obtained after training

	1	2
Accuracy	92%	88%
Training time	0:00:07:02.85	0:06:15:31.29
Time per epoch	0:00:00:00.42	0:00:00:45.06

Table 5.3: Table to test captions and labels

5.2.2 Single-layer neural network (784, 10)

```
neuralNet := MLNeuralNetwork new initialize: #(784 10).
neuralNet outputLayer activationFunction: (MLSoftmax new).
neuralNet costFunction: (MLCrossEntropy new).
neuralNet lerningRate: 0.5.

neuralNet learningAlgorithm: (MLMiniBatchBackpropagation new).
neuralNet batchSize: 100.


neuralNet learn: trainingData epochs: 1000.
```

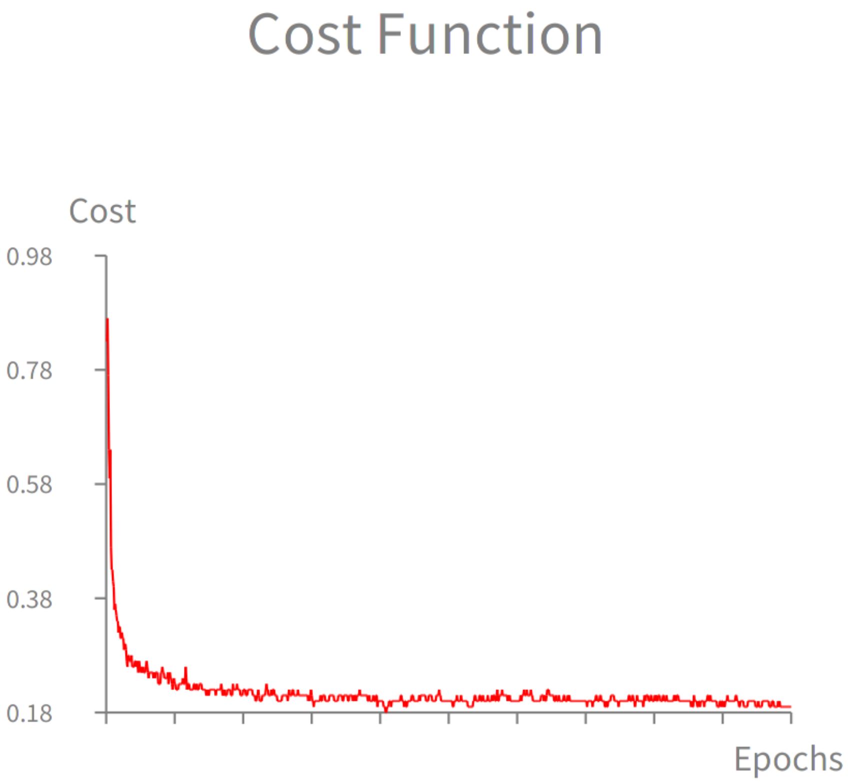

Figure 5.1: Cost history of a single-layer neural network

5.2.3 Two-layer neural network (784, 500, 10)

```
neuralNet := MLNeuralNetwork new initialize: #(784 500 10).
(neuralNet layerAt: 1)
neuralNet outputLayer activationFunction: (MLSoftmax new).
neuralNet costFunction: (MLCrossEntropy new).
neuralNet lerningRate: 0.5.

neuralNet learningAlgorithm: (MLMiniBatchBackpropagation new).
neuralNet batchSize: 100.


neuralNet learn: trainingData epochs: 1000.
```

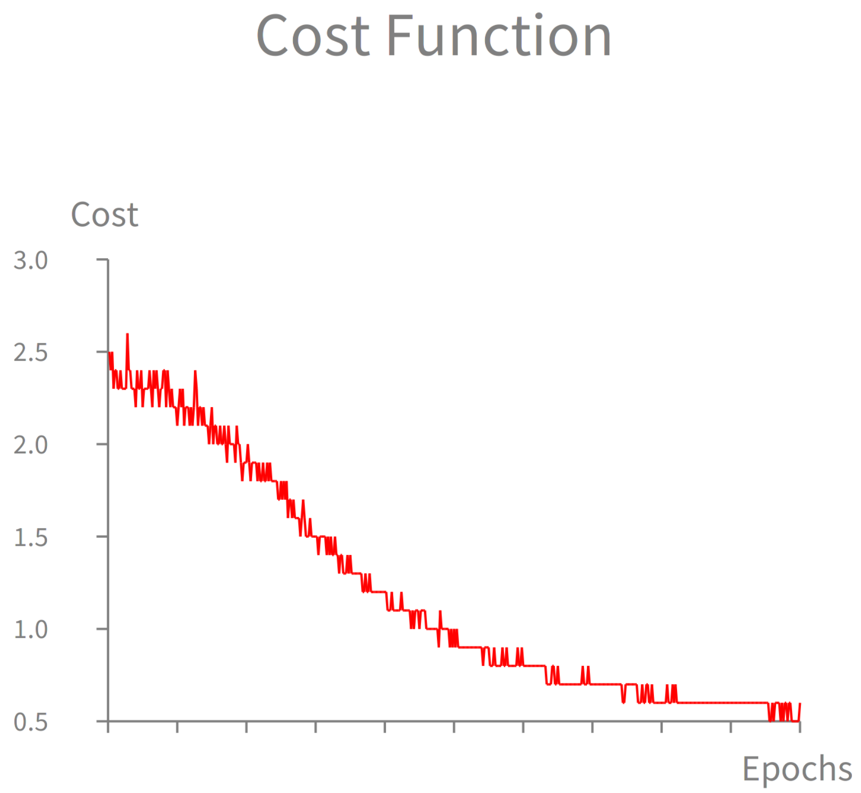

Figure 5.2: Cost history of a two-layer neural network with 500 hidden units. The network was trained with mini-batches of size 100 during 500 epochs and with learning rate 0.1

The Roassal[3] visualization of a random prediction made by a trained neural network with softmax output layer. The activity of 10 output neurons is probabilistic and sums up to 1. The neuron with the highest activity indicates the prdicted class. The bar of a correct (expected) class is green. If you see only one colorful bar, it means that the network made a correct prediction. If the network made a mistake and the highest bar is different from the targeted class, it is highlited in red. Then you can see a green bar indicating the correct class, and the red bar of the class that was incorrectly predicted.

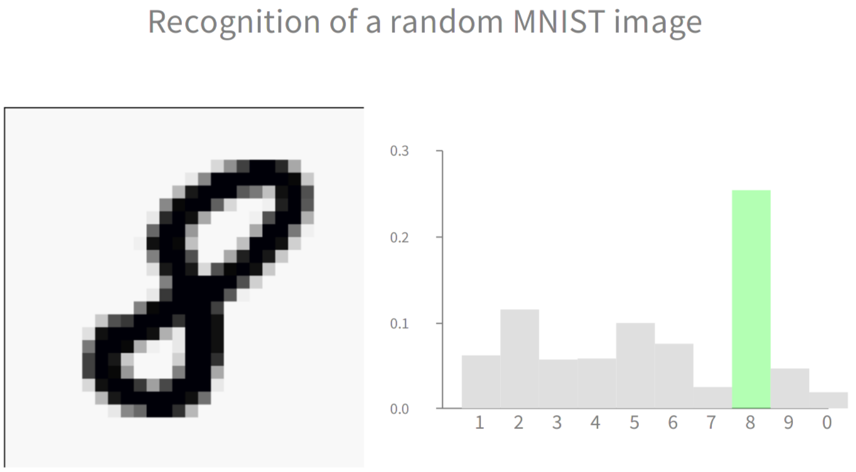

Figure 5.3: Predicting random MNIST image

5.3 Comparing to TensorFlow

Architectures of neural networks described above are chosen to be the same as the ones described in several other works. This way we can reproduce the historical experiments and compare the performance.

The first network with a single layer of softmax neurons was repreduced from a famous TensorFlow tutorial. Experiments show that the accuracy of both networks is about the same and slightly variates around the value of 92%. On the other hand, the time spent on learning by TensorFlow is signisicantly (almost 180 times) smaller. This demonstrtes the implortance of parallelization of neural networks and shows the power of GPU optimization.

Taking into consideration that Pharo virtual machine can only run on a single CPU core, 7 minutes spent on training a neural network that would be trained in 3 seconds by TensorFlow, is not a bad result at all.

Summary

I have implemented a modular and extendable neural network package in Pharo. This is the first implementation of a neural network in Pharo, capable of digit recognition. It is proved experimentaly that the neural networks created with this package are capable of classifying the handwritten digits of MNIST dataset with 92% accuracy. Which is exactly the same as the simplest one-layer softmax model created with TensorFlow. Hovewer, the speed of learning appears to be about 180 times lower than the one on TensorFlow. This can be explained by the fact that the latter is parallelized on multiple CPU cores and also uses a GPU for the most time-consuming operations, while the virtual machine of Pharo can only run on a single CPU core. This slow performance demonstrates the need in parallelization and native boost of some operations. Therefore, it is safe to assume that the best and the easiest solution for implementing neural networks in Pharo that could find real-life applications (especially deep networks with more than one hidden layer of neurons) would be by creating the bindings for TensorFlow or any other open-source library of that type.

This work demonstrates the capabilities of Pharo in the field on machine learning and artificial intelligence. It layed a foundation for several other project that would bring us closer to having a full-scale machine learning toolkit implemented in Pharo.

Bibliography

[1] В. М. Трушевський, Г. А. Шинкаренко, and Н. М. Щербина. *Метод скінченних елементів і штучні нейронні мережі. Теоретичні аспекти та застосування.* ЛНУ імені Івана Франка, 2014.

[2] Adele Goldberg and David Robson. *Smalltalk-80: The language and its implementation.* Addison-Wesley, 1983.

[3] Alexandre Bergel. *Agile Visualization.* 2016.
Available at `http://agilevisualization.com`.

[4] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep into Pharo.* Square Bracket Associates, first edition, 2013.

[5] Andrew M. Saxe, Pang Wei Koh, Zhenghao Chen, Maneesh Bhand, Bipin Suresh, and Andrew Y. Ng. On random weights and unsupervised feature learning. *ICML*, 2011.

[6] Barry Kristian Ellingsen. An object-oriented approach to neural networks. Technical report, 1995.

[7] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

[8] David E. Rumelhart, Geoffrey E. Hintonand, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[9] David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms.* Cambridge University Press, 2003.

[10] Didier H. Besset. *Object-Oriented Implementation of Numerical Methods. An Introduction with Pharo.* Square Bracket Associates, first edition, 2016.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. Available at `http://www.deeplearningbook.org`.

[12] Mohamad H. Hassoun. *Fundamentals of Artificial Neural Networks.* MIT Press, 1995.

[13] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors. 2009.

[14] Simon Haykin. *Neural Networks. A Comprehensive Foundation.* Prentice Hall, second edition, 2005.

[15] Stuart Russell and Peter Norvig. *Artificial Intelligence. A Modern Approach.* Prentice Hall, third edition, 2010.

[16] Stéphane Ducasse, Dimitris Chloupis, Nicolai Hess, and Dmitri Zagidulin. *Pharo by Example 5.* Square Bracket Associates, first edition, 2017.

[17] Tom Mitchell. *Machine Learning.* McGraw-Hill Science/Engineering/Math, 1997.

[18] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning. Data Mining, Inference and Prediction.* Springer, second edition, 2013.

[19] Nicolas Pinto, David D. Cox, and James J. DiCarlo. Why is real-world visual object recognition hard? *PLOS. Computational Biology*, 2008.