

Object-oriented Implementations of Artificial Neural Networks

Oleksandr Zaytsev

May 10, 2017

Contents

1	Introduction	2
1.1	Related work	2
1.2	What is Pharo?	2
1.3	Installing the library	2
1.3.1	Getting Pharo	2
1.3.2	Installing the Dependencies	2
1.4	Example of usage	3
1.5	How to contribute?	3
2	Machine learning with neural networks	4
2.1	What is machine learning?	4
2.2	Neural networks	5
2.3	Model of a neuron	6
2.4	Structure and representation	8
3	Single-layer perceptron	10
3.1	What is a perceptron?	10
3.1.1	Restrictions of perceptrons	11
3.2	Design issues	11
3.2.1	How to represent weights?	11
3.2.2	Activation functions	12
3.2.3	Shared or separate activation and learning rate?	13
3.2.4	Data shuffling	14
3.3	Implementation	14
3.3.1	Neuron class	15
3.3.2	SLPerceptron class	16
3.4	Testing	17
4	Multi-layer neural network	19
4.1	Why not "multi-layer perceptron"?	20
4.2	Design issues	20
4.2.1	Weight initialization	20
4.3	Implementation	21
4.4	Testing	21

Chapter 1

Introduction

In this [thesis] I will describe my implementation of artificial neural networks in Pharo.

1.1 Related work

1.2 What is 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. [By now it has more than a hundred contributors] [1].

1.3 Installing the library

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

1.3.1 Getting Pharo

1.3.2 Installing the Dependencies

Here is the complete list of dependencies:

- **PolyMath** - a library for numerical methods. It provides MATLAB-like vectors and matrices. It is similar 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 [Guillermo 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 information about the network architecture, defined by a simple one-dimensional array of integers $\#(s_1 s_2 \dots s_n)$. 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?

Chapter 2

Machine learning with neural networks

[...]

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* [5] 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) [6]. This definition is widely known and often referred 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 losing the chess game)

Here are the formal definitions for the problems of supervised and unsupervised machine learning¹.

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

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

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

$$x^{(1)}, x^{(2)}, \dots, x^{(m)}$$

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.

Definition Simon Haykin [4] 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 acquired by network from its environment through a learning process.
2. Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.

¹Formal definitions of supervised and unsupervised learning problems are inspired by [8] and [7] respectively

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. [3] view neural networks as nonlinear statistical models, the two-stage regression or classification models. David MacKay [5] sees them as parallel distributed computational systems consisting of many interacting simple elements. And Goodfellow et. al. (MIT) [2] 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 dictionary²,

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 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 for 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.

²<http://www.merriam-webster.com/dictionary/neuron>

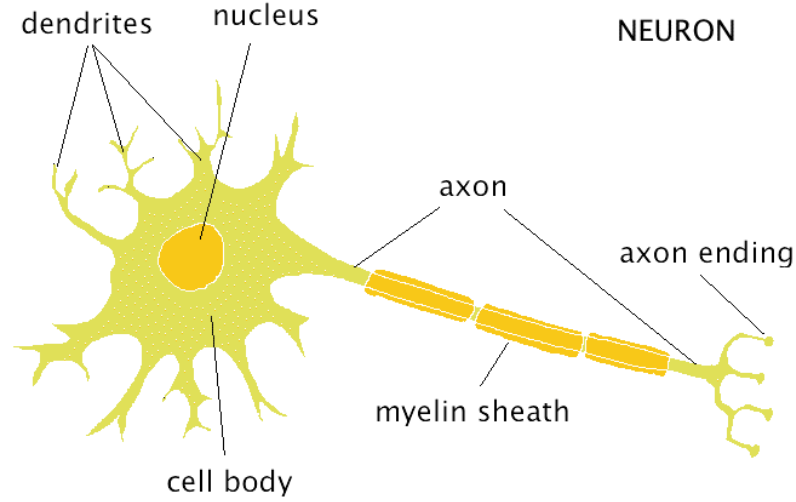


Figure 2.1: A schematic of biological neuron

Artificial neurons In fact, artificial neuron can be viewed as a parametric function of n inputs $g_w : X^n \rightarrow Y$, 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, \dots, w_n , the tunable parameters that [...]. In other words, function g_w is [defined with the rule]

$$y = g_w(x) \quad (2.1)$$

where $x \in X^n$, $y \in Y$ and $w \in \mathbb{R}^n$.

Function g_w is a superposition of two functions, $s_w : X^n \rightarrow \mathbb{R}$ and $f : \mathbb{R} \rightarrow Y$:

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

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.

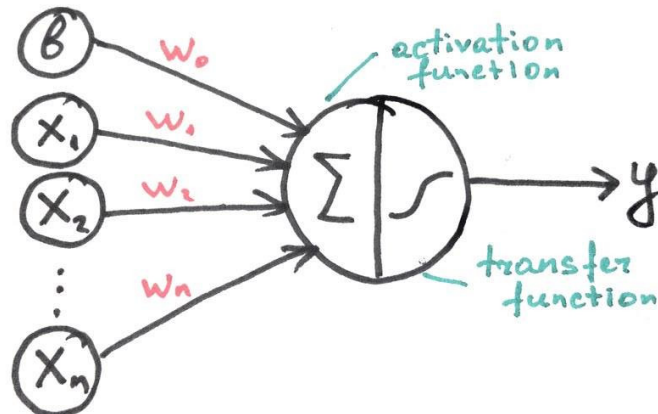


Figure 2.2: Model of artificial neuron

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 -partite, where k is the number of layers. If it is also fully-connected, it will be a complete k -partite graph. See Figure 2.3 for specific examples.

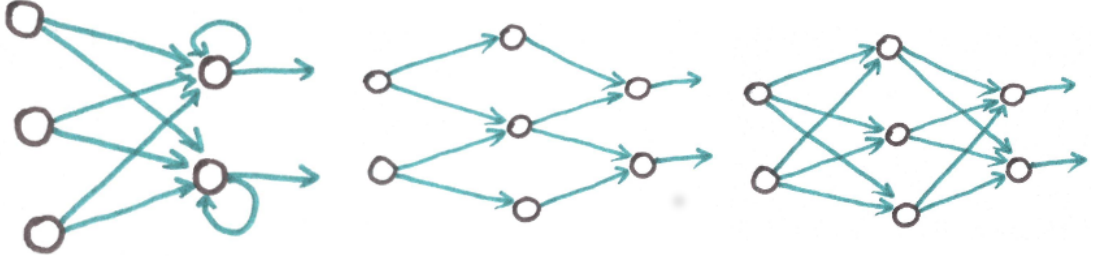


Figure 2.3: (1) Recurrent neural network represented by directed graph. (2) 3-layered feedforward neural network represented by 3-partite directed graph. (3) 3-layered fully-connected feedforward neural network represented by a complete 3-partite directed graph

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 neural network. According to [3], 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 network 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.

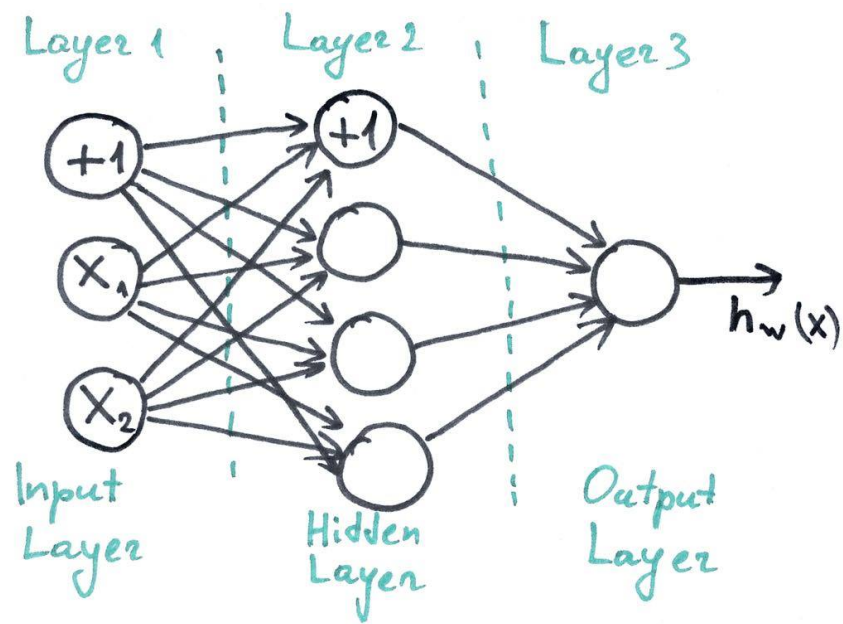


Figure 2.4: Feedforward artificial neural network with one hidden layer. Number of neurons in each layer: 3 (2-dimensional input + bias unit) in the input layer, 4 in the hidden layer, 1 in the output layer (1-dimensional output)

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 in my NeuralNetwork repository on Smalltalkhub¹.

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:

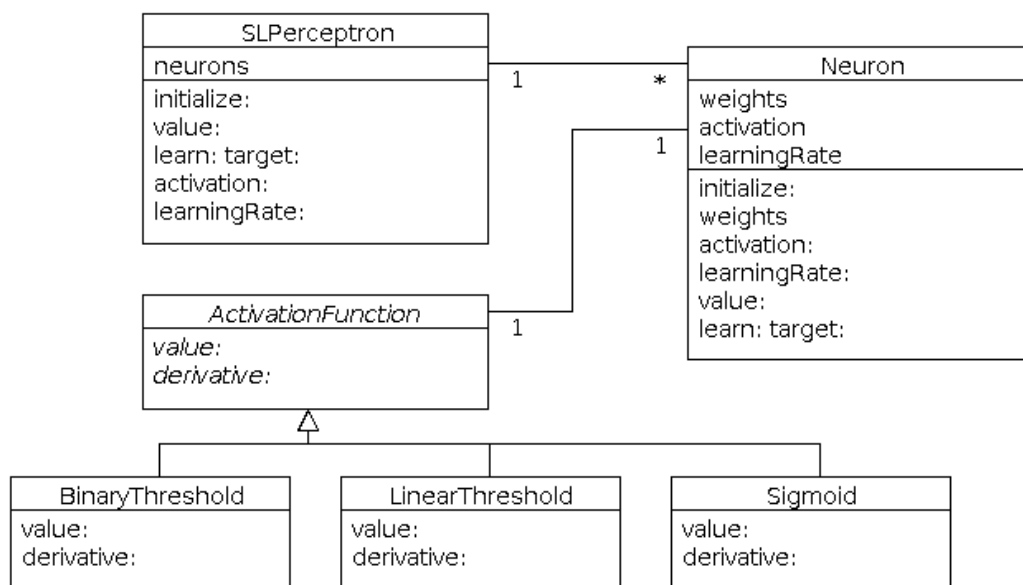


Figure 3.1: Object-oriented design of a single-layer perceptron

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.

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.

¹<http://smalltalkhub.com/#!/Oleks/NeuralNetwork>

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 its very easy to implement, so Ill 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.

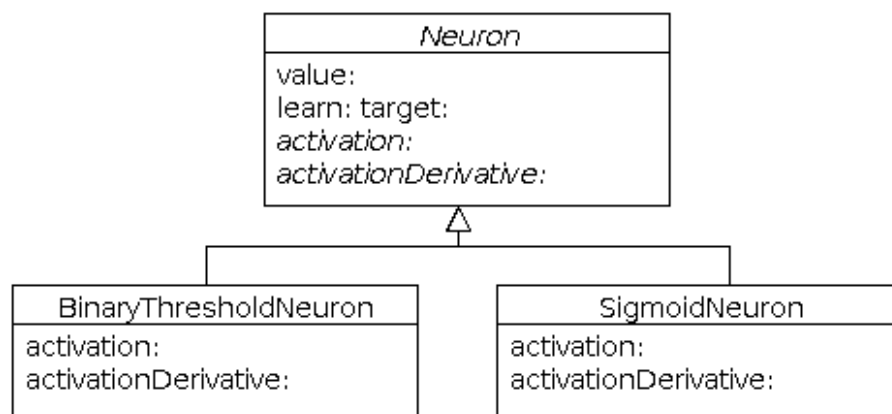


Figure 3.2: Implementing activation functions as methods of specific subclasses on `Neuron`

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 doesnt seem logical to reimplement the whole neuron every time we need to create a function.

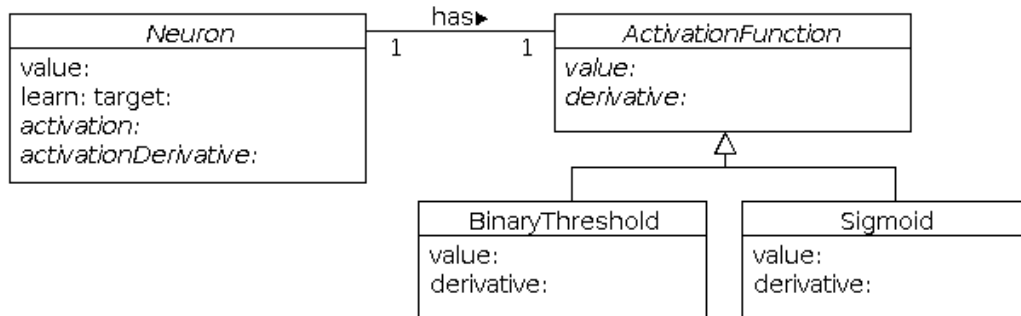


Figure 3.3: Implementing activation functions as separate classes

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?

Lets assume that we dont. 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. Thats 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 thats not the topic of this post.

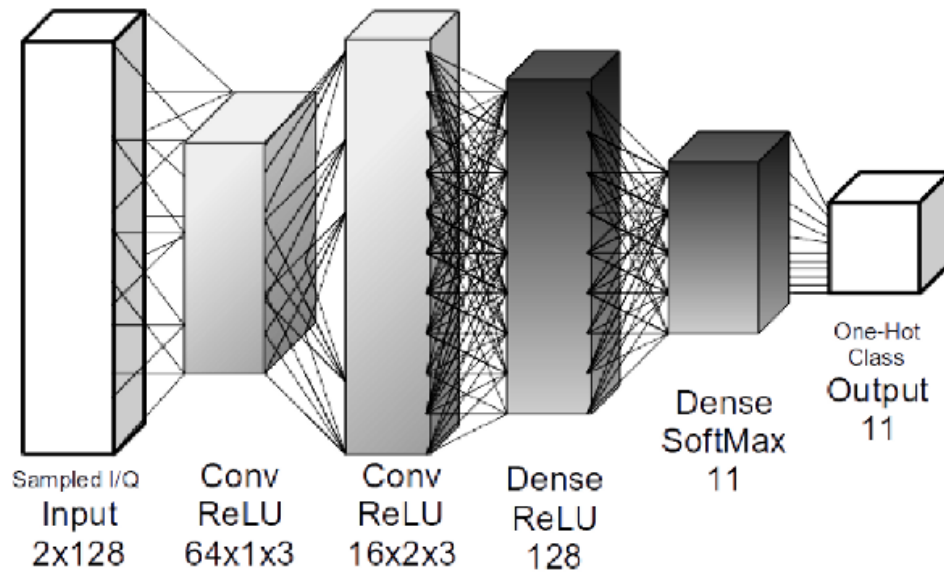


Figure 3.4: A convolutional neural network with four hidden layers

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 perceptron:

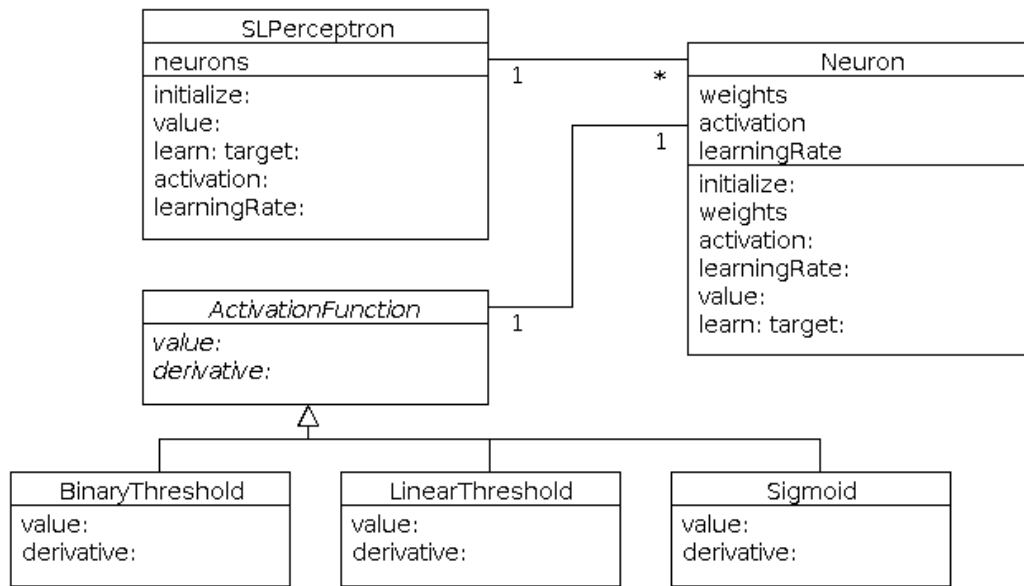


Figure 3.5: Object-oriented design of a single-layer perceptron

3.3.1 Neuron class

```

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

```

The weights are initialized with random numbers in range $[0, 1]$. Im 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. Assign

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 and 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 ex

  | 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) c

    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

```

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
"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 dont have a good enough understanding of how to implement a perceptron with sigmoid.

Chapter 4

Multi-layer neural network

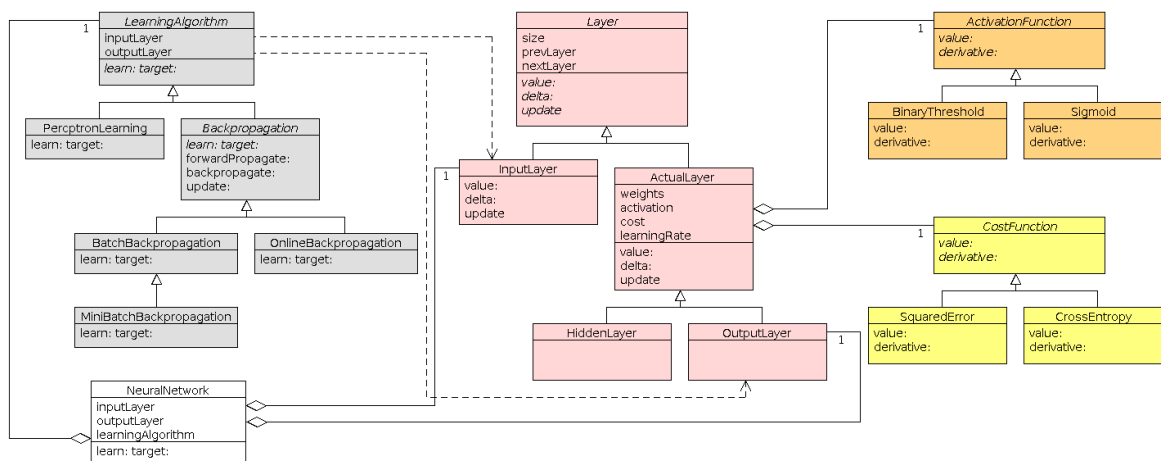


Figure 4.1: Implementing activation functions as methods of specific subclasses on Neuron

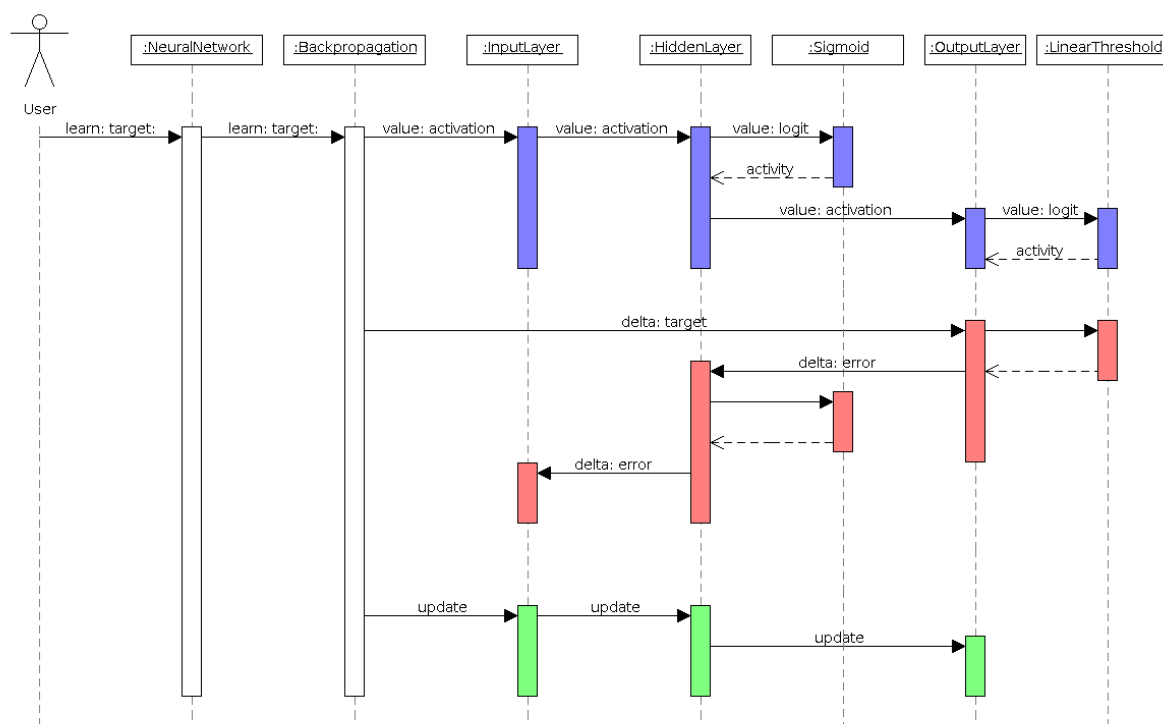


Figure 4.2: Implementing activation functions as methods of specific subclasses on Neuron

4.1 Why not "multi-layer perceptron"?

When writing about neural networks, many authors (most of them in fact) start by describing the single-layer perceptron, talking about its restrictions (XOR problem), and then introduce the awesome multi-layer perceptron with backpropagation learning procedure and its amazing capabilities. Many famous software libraries share this terminology. For example, scikit-learn with its MLPClassifier (Multi-Layer Perceptron Classifier). But in his Neural Networks for Machine Learning course on Coursera Geoffrey Hinton argues that we shouldn't use such term as multi-layer perceptron, because the perceptron learning procedure can not be applied to hidden layers. Personally, I just think that the term perceptron is even more misleading than neural network. Very few people can clearly define what perceptron is, and most of us mean different things when speaking of perceptrons. So I prefer to apply the term perceptron only to those neural networks which adjust their weights using the perceptron learning procedure (which is a simple error-correction rule that can not be applied to multiple layers).

4.2 Design issues

4.2.1 Weight initialization

Neural network is an [optimization system] that changes its weights to improve the performance on a given dataset. Each configuration of weights represents a point in the multidimensional 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.

[Unsupervised pre-training]

Initializing all weights with zeroes is the easiest thing to do. However, a network with such weights may never learn anything. All weight updates 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. [?] evaluated thousands of architectures on a number of object recognition tasks and found that random weights performed only slightly worse than pretrained weights. And [9] 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 [10], a good way to initialize the weights of a layer L is by choosing random values from range $[-\frac{1}{2k}, \frac{1}{2k}]$, where k is the number of neurons in layer L .

4.3 Implementation

4.4 Testing

Summary

The project is on the early stage. Please, contribute.
Here is the list of future features:

- Recurrent neural networks
- Deep convolutional neural networks

Bibliography

- [1] Alexandre Bergel, Damien Cassou, Stphane Ducasse, and Jannik Laval. *Deep into Pharo*. Square Bracket Associates, first edition, 2013.
- [2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. Available at <http://www.deeplearningbook.org>.
- [3] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning. Data Mining, Inference and Prediction*. Springer, second edition, 2013.
- [4] Simon Haykin. *Neural Networks. A Comprehensive Foundation*. Prentice Hall, second edition, 2005.
- [5] David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.
- [6] Tom Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1997.
- [7] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors. 2009.
- [8] Stuart Russell and Peter Norvig. *Artificial Intelligence. A Modern Approach*. Prentice Hall, third edition, 2010.
- [9] Andrew M. Saxe, Pang Wei Koh, Zhenghao Chen, Maneesh Bhand, Bipin Suresh, and Andrew Y. Ng. On random weights and unsupervised feature learning.
- [10] . . . , . . . , and , 2014.