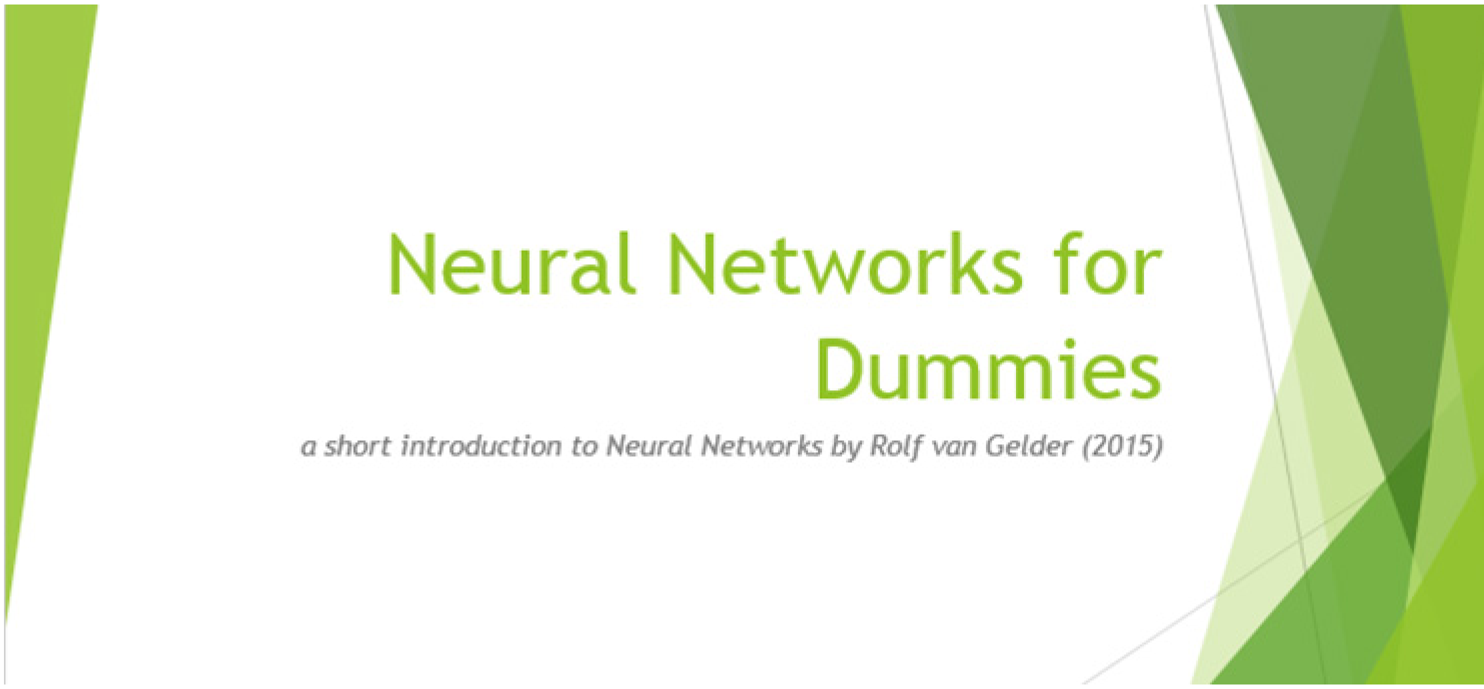


Neural Networks for Dummies

a short introduction to Neural Networks by Rolf van Gelder (2015)

[February 1, 2017]

Neural Networks for Dummies

Contents

Neural Networks for Dummies

Introduction

Neural networks are 'self-learning' computer programs and are normally created and configured to solve one specific problem.

They are used in cases when it's too hard to use a traditional computer program.

Like in the case I will explain later on: we want to build a network that will recognize handwritten digits.
If you want to recognize handwritten digits with a traditional program, it probably will need many thousands lines of code. Our demo network will only use less than 1000 lines of code, and the best part: the same program code can be used for solving different problems (with just some minor adjustments).

Nowadays many neural networks are used in my different fields:
For instance Thunderbird (mail client) uses a neural network for identifying spam mails.
When the user marks an email as spam, it will train the network with that information.
(More about training later)

Other applications: license plate recognition, OCR, face recognition and so on.

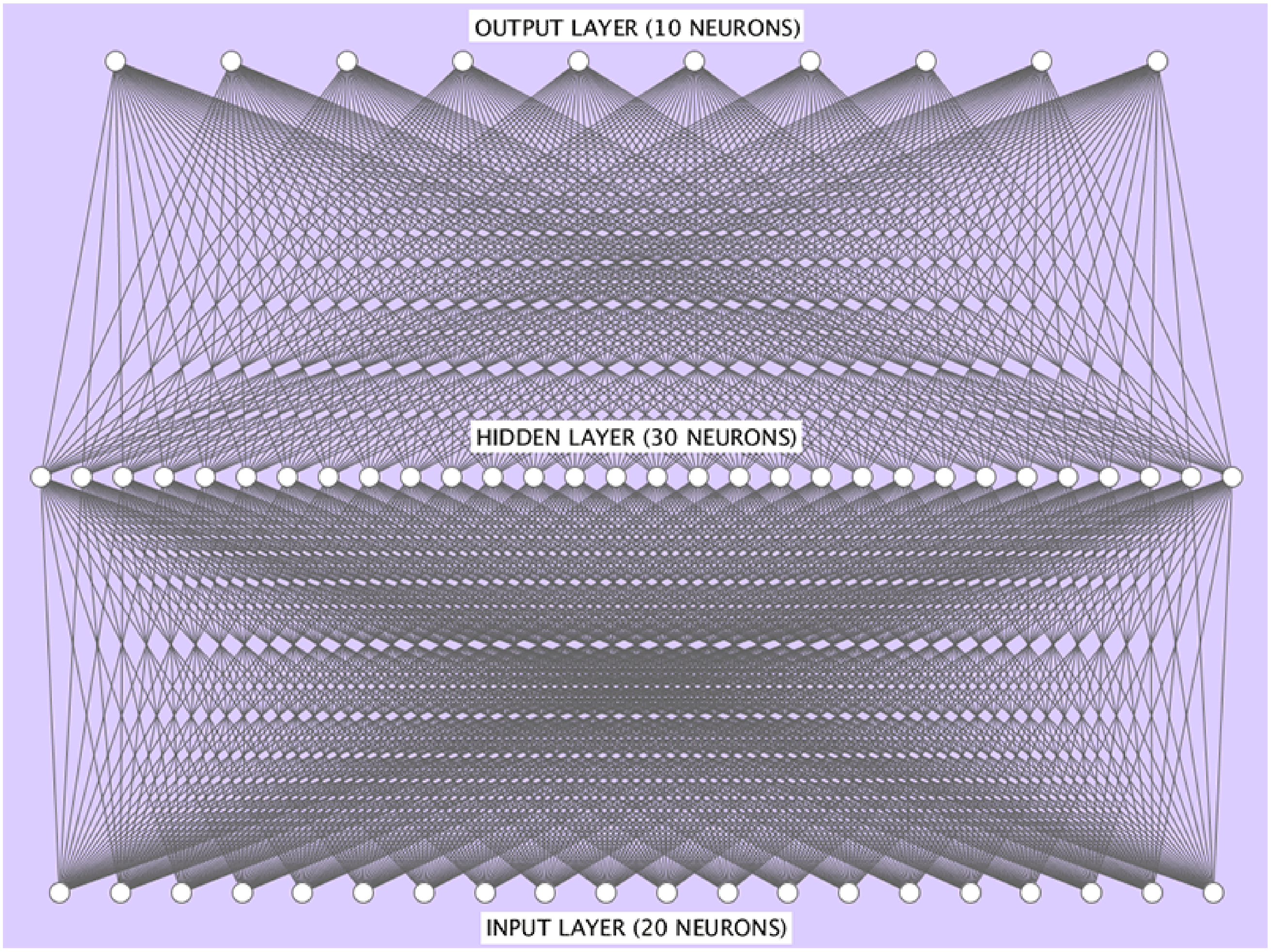
In fact neural networks simulate a tiny part of a human brain.

Like the human brain, a neural network is built with so-called '***neurons'***

A neural network has several layers, built with neurons.

There is always one '***input layer***', zero or more '***hidden layers***' and one '***output layer***'.

Schematic of a neural network with one hidden layer:



All neutrons are identical (same code) with one exception: neurons in the input layer don't have inputs themselves.

Depending on the problem it has to solve, the number of input neurons, the number of hidden layers and the number of neurons of the hidden layers and the number of output neurons will be chosen.

Every neuron in a layer is connected to all the neurons in the next layer (see above illustration).

In the above example there are already (20 * 30) + (30 * 10) = 900 connections (lines)!

In our test case (further on) we will use 196 input neurons, 100 hidden neurons and 10 output neurons. That's (196 * 100) + (100 * 10) = 20,600 connections!

During training and testing often the number of hidden layers and the number of hidden layer neurons will be changed to find the best configuration for performing that specific task.

How does it work?

Well, so far, no one really understands how it actually works... Especially the hidden layers are quite 'mysterious'.

There are two things you can do with the network: '***train the network***' and '***test and/or use the network***'.

Training a Network

Without training a network, the output of the network will be completely random. The network has no idea what to do with the input, so it just makes a guess.

Training is like teaching children: You tell them 2 + 2 = 4. And you tell them 1 + 6 = 7 and so on.
If you repeat that often enough, the child learns how to add two numbers.

Same for the network: during its training period you put inputs in and you tell it what the output should be.

Every neuron has two properties: '***weight***' and '***bias***' (sometimes also called '***threshold'***).

These two properties determine what the output value will be and what the neuron will pass on to the neurons in the next layer, based on its inputs.

In short: the weight defines how important the value is; the bias defines how high the weight should be before passing it on to the next layer. Together they calculate the output value.

When a training input is processed, it generates output values based on the current state of the neurons.
These output values in the output layer are compared to the right answer. The properties will be changed a little bit, network will be tested again with the same input and the new output will be compared with the right answer. If the new output is better than the previous one, the new properties will be saved.
So, the more often you train the network, the more reliable the output will become.

Testing / Using a Network

After every training session, the network will be tested: did the reliability improve?

When the reliability is good enough, the network is ready for its job!

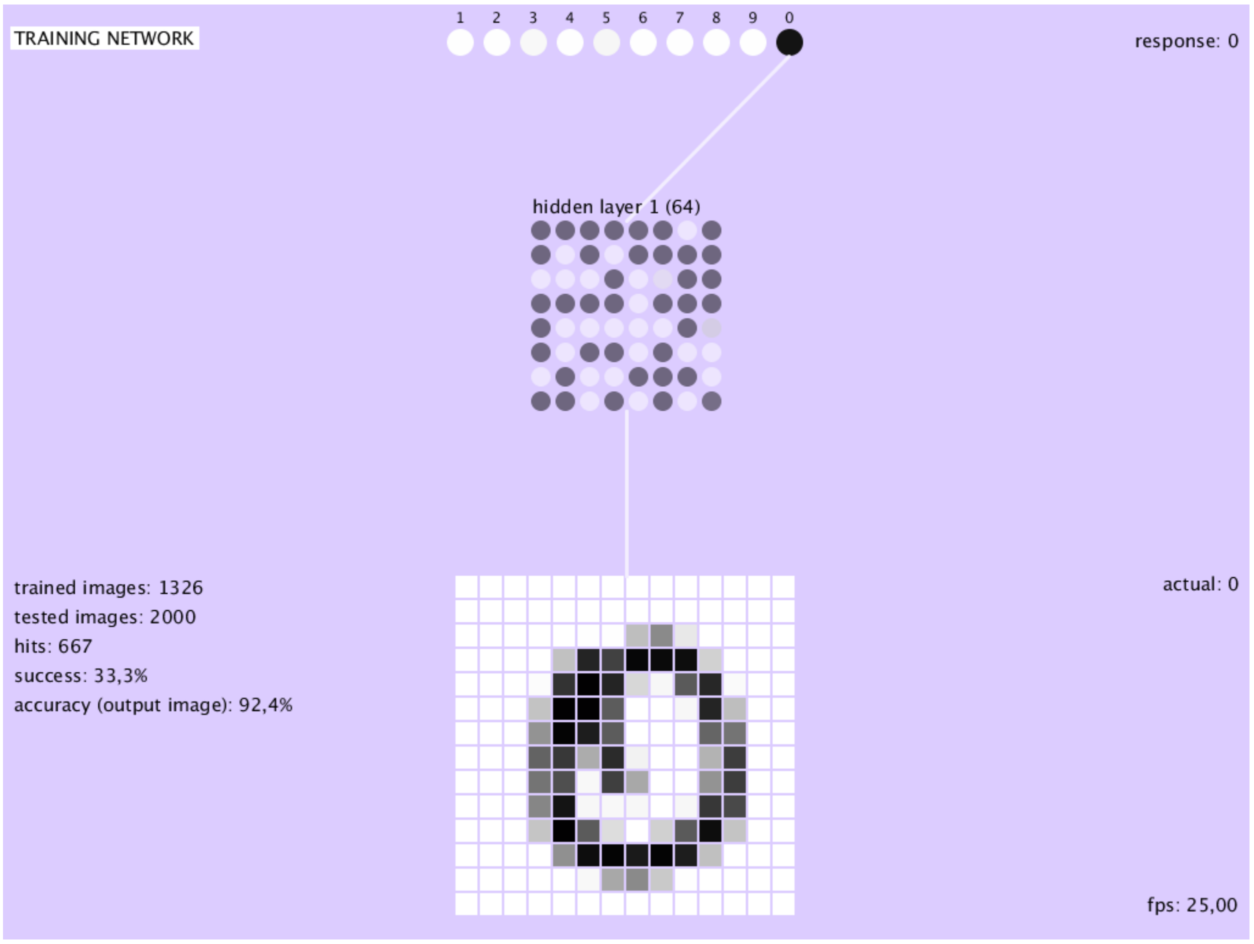
Training and testing use different sets of data: '***training-data***' and '***testing-data***'.
After training it, the network should be 'clever' enough to also solve inputs it never saw before.

These training- and testing datasets can be huge: millions of inputs. The basic rule is: the more the better!

The Test Case: Recognition of Handwritten Digits

You can download and play with the demo Network yourself! Just download and run the '*rvg_neural_net.exe*' app from http://cagewebdev.com/wp-content/uploads/2016/10/rvg_neural_net20_win32.zip

Screen grab of the demo network in action (1 hidden layer with 64 neurons):



trained images: 1326
tested images: 2000
hits: 667
success: 33,3%
accuracy (output image): 92,4%

Goal of the Network

The goal is that our network will recognize handwritten digits.

I warn you: the result won't be 100% accurate!

People also make mistakes reading handwritten digits: for instance many people will say it's a 'one' while in fact it's meant to be a 'seven' or the other way around. Depends on the handwriting style of the person who wrote it.

So, our little network will also be fooled by that.

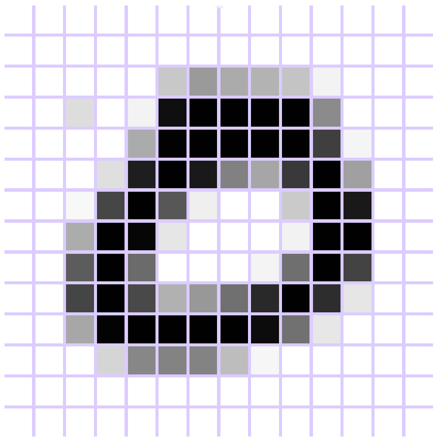
The training- and test-sets for our network are from the '***MNIST***' database. MNIST, a foundation based in New York asked many people to write down digits (and also the same digits many times).
In total there are 60,000 training- and 10,000 handwritten testing-digits in our datasets.

Structure of the Network

Our network has 196 input neurons, a certain amount of hidden layers with a certain amount of neurons, which we can adjust, and 10 output neurons.

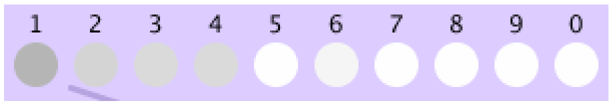
Why 196 inputs? Because the images in the MNIST database are simplified to 14 x 14 (=196) pixels. So, there should be an input for every pixel.

For instance (which apparently is a 'zero'):



Like said: we can play around with the number of hidden layers and the number of neurons in the hidden layer(s).

The output layer, in our test case, has 10 neurons:



The darkness of the dots indicates what the network thinks the right answer is.
The darker the color, the more sure it is that it's the right answer.

In the above example, apparently the network thinks it's a 'one' and it's not very sure about it (could be much darker, up to pitch black).

Step one: testing the untrained Network

Just for fun: let's test the untrained network.

Without any training the weights and biases of the neurons will be totally random.

So, we easily can predict the outcome of this experiment will be around 10% accuracy (=success).

If you guess a number between 0 and 9, in about 10% of the cases you will have guessed it right.

A testing session means: putting 1000 (=adjustable amount) images into the network and measure the number of hits (correct answers). Based on the total number of hits, it calculates the percentage of accuracy of the network (success %).

The images, by the way, are selected at random from the datasets.

After downloading and starting the demo app (***rvg_neural_net.exe***), you click the ***right mouse button*** to start a testing session. It will test 1000 test images per session.

As you will see, the success percentage is somewhere around the 10% (since the network is not trained yet, as explained above).

Step two: training the Network

Next: let's see if training the network helps...

During a training session we'll put in a batch of 1000 (=adjustable amount) random images from the training-set.

For every image, the network will compare the actual output from the network with the wanted output and adjust the weights and biases of the neurons accordingly.
Teaching it: "Hey Network, this should be a 'two' and not a 'five'!"

After two training sessions (2000 images) it turns out that the accuracy already has gone up to around 65%.

So, our network is actually learning!

We can train it again and again and see what happens to the accuracy.

You can train the demo network (***rvg_neural_net.exe***) by clicking the ***left mouse button*** somewhere on the canvas.
After one or more training sessions, just run the test session again (click the ***right mouse button***) and watch the success percentage go up.

Step three: experimenting with the Hidden Layers

Now we can experiment with the number of hidden layers.
And the number of neurons per hidden layer.
It's a purely empirical process: it's impossible to predict if the outcome will be better or worse.

Sometimes 1 hidden layer works much better than 5 hidden layers or the other way around.

There are no formulas to calculate the most efficient number of hidden layers and neurons for solving the problem. It's just a matter of experimenting. That's part of the fact that no one really knows how it works.

To increase the number of hidden layers in the demo network (***rvg_neural_net.exe***) press the ***'L'-key***.
To decrease the number of hidden layers in the demo network (***rvg_neural_net.exe***) press the ***'l'-key***.

To increase the number of neurons in the hidden layer(s) in the demo network (***rvg_neural_net.exe***) press the ***'N'-key***.
To increase the number of neurons in the hidden layer(s) in the demo network (***rvg_neural_net.exe***) press the ***'n'-key***.

Appendix I: Quick Reference '*rvg_neural_net.exe*' Demo Network

How to interact with the demo app (***rvg_neural_net.exe***)?

Mouse

- ***Left-click***: start a **training** session (= feed the network 1000 training images)
- ***Right-click***: start a **testing** session (= feed the network 1000 testing images)

Keyboard

- ***Spacebar***: test the next image (one by one)

- ***'b'***-key: decrease **training batch size**
- ***'B'***-key: increase **training batch size**

- ***'f' or 'F'-key***: toggle between **slow** (non-animated) and **fast** (animated) training

- ***'l'-key***: decrease the **number of hidden layers** (min = 0)
- ***'L'-key***: increase the **number of hidden layers** (max = 5)

- ***'n'-key***: decrease the **number of neurons in the hidden layers** (min = 25)
- ***'N'-key***: increase the **number of neurons in the hidden layers** (max = 144)

- ***'-'-key***: decrease the **framerate** (speed) for auto testing
- ***'+'-key***: increase the **framerate** (speed) for auto testing