Neural network programming: introduction

Erik Spence

SciNet HPC Consortium

16 April 2019

Today's code and slides

You can get the slides and code for today's class at the SciNet Education web page.

`https://support.scinet.utoronto.ca/education`

Click on the link for the class, and look under "File Storage", find the file "intro_I.pdf", and "lecture1_code.tar.gz".

About this class, SciNet's DAT112

The purpose of this class is to introduce you to basic and intermediate neural network programming in Python. Some notes about the class:

- This class is a combination of SciNet's "Introduction to Neural Network Programming" (DAT111), "Advanced Neural Networks" (DAT212), plus some new material.
- As such, the material will start introductory and then build up to cover more-advanced topics.
- We'll meet Tuesdays and Thursdays, 11:00am - 12:00pm, in the SciNet teaching room, for six weeks.
- All classes will be recorded, and the lecture material made available.
- There will be 3 assignments, which will be due the week after they are assigned. The first will be assigned in the second week.
- This class qualifies for 16 credits toward a SciNet Data Science Certificate.

About this class, continued

Software requirements for this class:

- We'll be using Python 3.7.X. Python 3.6.X will also likely work. The code may work with Python 2.7.X, but no promises.
- Starting class 3 we'll be using the Keras neural-network programming framework. You may use whichever Keras backend (Theano, TensorFlow, CNTK) you like.
- You will need the usual machine-learning packages: numpy, matplotlib, scikit-learn.
- The final assignment may need the use of GPU resources to complete in a reasonable amount of time. I'm currently exploring how those resources might be made available.

The first 3 weeks we will cover the basics of how neural networks work. The second half of the coure will be a survey of many of the types of neural networks. Ask questions!

Class topics

This class will cover the following topics:

- Introduction to neural networks.
- Programming neural networks using Keras.
- Various computer vision topics, using neural networks.
- Recurrent Neural Networks.
- Various types of generative networks.
- Reinforcement learning with neural networks.
- And more...

If there's a type of neural network you'd like to see covered, please let me know.

Today's class

Today's class will cover the following topics:

- Motivation for neural networks.
- Neurons.
- Gradient descent optimization.
- Training example.

Please ask questions if something isn't clear.

Neural networks?

Let us begin at the beginning. What are neural networks? Neural networks, also called artificial neural networks,

> *are a computational model used in machine learning, ..., which is based on a large collection of connected simple units called artificial neurons, loosely analogous to axons in the biological brain. ... Such systems can be trained from examples, rather than explicitly programmed, and excel in areas where the solution or feature detection is difficult to express in a traditional computer program.*
>
> *Wikipedia*

If you're doing pattern recognition of any type, neural networks are worth considering.

Neural networks are commonplace

Neural networks are particularly good at detecting patterns, and for certain problems perform better than any other known class of algorithm. Neural networks are used for

- Image recognition, object detection.
- Natural language processing (voice recognition).
- Novelty detection (detection of outliers).
- Next-word predictions.
- Text sentiment analysis.
- System control (self-driving cars).
- Medical diagnosis.

Neural networks are finding their way into everything.

Neural networks, motivation

Consider the problem of hand-written digit recognition:

How would you go about writing a program which can tell you what digits are displayed?

- All the algorithms you might use to describe what a given number "looks like" are extremely difficult to implement in code. Where do you even start?
- And yet humans can easily tell what these digits are.
- This is one of the classic problems which have been solved using neural networks.

Again, anywhere patterns show up, neural networks can be used.

Neural networks, the approach

Rather than focus on the details of what individual numbers look like, we will instead ignore those details altogether. We will use a supervised machine-learning approach:

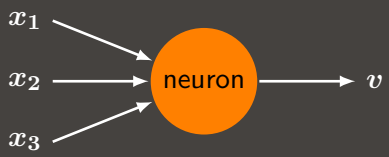
- Break the dataset of numbers into two or three groups: training, testing, and optionally validation.
- Feed the training data to the neural network and train it to recognize one number from another.
- Let the neural network figure out the details for itself.

By the end of lecture 5 you should be achieving greater than 99% classification accuracy from your neural network on the hand-written digits data set.

But first, let's start with the basics.

Neurons

Neural networks are built upon "neurons". This is just a fancy way of saying a "function that takes multiple inputs and returns a single output".



The function which the neuron implements is up to the programmer, but it must contain free parameters so that the network can be trained. These functions usually take the form

$$f(x_1, x_2, x_3) = f\left(\sum_{i=1}^{3} w_i x_i + b\right) = f\left(\mathbf{w} \cdot \mathbf{x} + b\right)$$

Where $\mathbf{w}$ are the 'weights' and b is the 'bias'. These are the trainable parameters.

Neurons, continued

What might this look like in practise?
Consider the data on the right.

cloudy (c)	hot (h)	windy (w)	beach?
1	3	2	1
3	3	2	1
1	3	3	0
1	1	2	0
1	2	2	0
2	3	1	1
2	1	2	0

Our neuron's function might look like:

$$f(c,h,w) = \begin{cases} 1 & w_1 \times c + w_2 \times h + w_3 \times w + b \geq 0 \\ 0 & w_1 \times c + w_2 \times h + w_3 \times w + b < 0 \end{cases}$$

Where we must now optimize w_1, w_2, w_3, b to match the data.

Training our neuron

How do we optimize our neuron's weights and biases? We need to define some sort of "cost function" (sometimes called "loss" or "objective" function):

$$C = \frac{1}{2} \sum_i \left(f(\mathbf{w} \cdot \mathbf{x}_i + b) - v_i \right)^2$$

where v_i are the correct answers, based on the data, associated with each $\mathbf{x}_i$. Here we are using the "quadratic" cost function.

We then use an optimization algorithm to search for the minimum of C, given $\mathbf{x}$ and v.

Neurons, continued more

But there's a difficulty here:

- Having an output being just "1" or "0", as in our previous example, is difficult to deal with, since the function is discontinuous.
- Very small changes in the weights, $\Delta \mathbf{w}$, can lead to discontinuous changes in the output. This makes using calculus difficult.
- Instead let's change the function of our neuron to use a "sigmoid function" (also called the "logistic function").

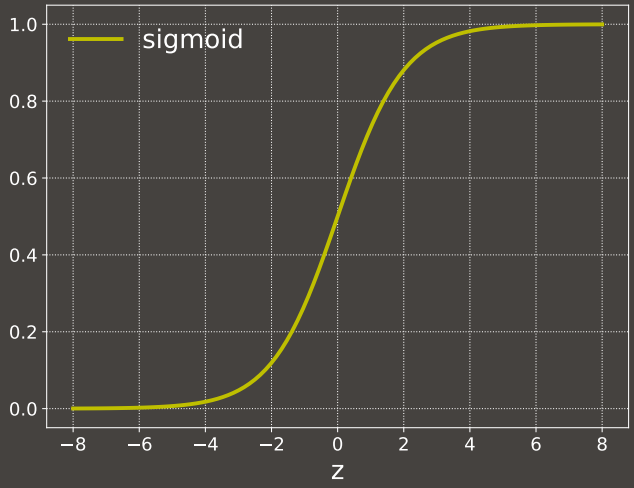
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

And so our neuron function becomes

$$f(x_1, x_2, x_3) = f\left(\mathbf{w} \cdot \mathbf{x} + b\right) = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + b)}}$$

Where $\mathbf{w}$ are again the 'weights' and b is the 'bias'.

Why the sigmoid function?



Because it ranges from 0 to 1 smoothly.

Using sigmoid neurons

By using the sigmoid neurons, we can approximate the changes in the cost function, C:

$$\Delta C \approx \sum_j \frac{\partial C}{\partial w_j} \Delta w_j + \frac{\partial C}{\partial b} \Delta b$$

meaning that the changes in the cost function are linear in changes to the weights and biases. The cost function changes 'smoothly' with changing parameters.

This allows us to use minimization algorithms to find the optimal values of $\mathbf{w}$ and $\boldsymbol{b}$.

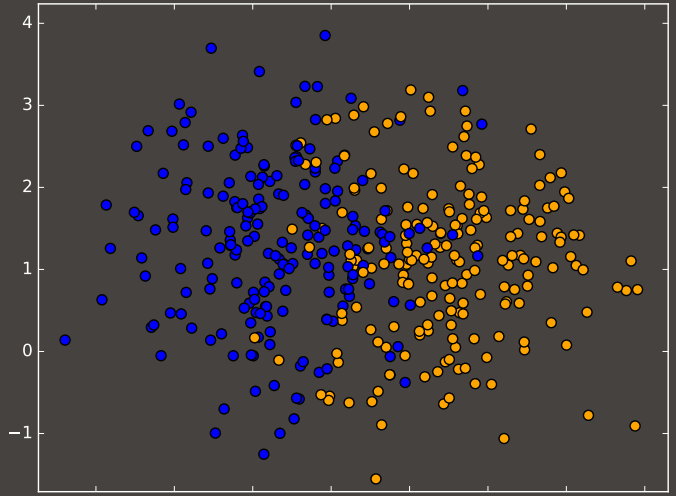
Our first example

We use the *sklearn.datasets.make_blobs* command to generate some toy data.

```
# example1.py
import sklearn.datasets as skd, sklearn.model_selection as skms

def get_data(n):
  pos, value = skd.make_blobs(n, centers = 2, center_box = (-3, 3))
  return skms.train_test_split(pos, value, test_size = 0.2)
```

```
In [1]: import example1 as ex1, plotting_routines as pr
In [2]:
In [2]: train_pos, test_pos, train_value, test_value = ex1.get_data(500)
In [3]:
In [3]: train_pos.shape, train_value.shape, train_pos[0], train_value[0]
Out[3]: ((400,2), (400,), array([ 2.10552892, 1.31996395]), 1)
In [4]:
In [4]: pr.plot_dots(train_pos, train_value)
```

Our data



The goal

What are we trying to accomplish?

- We want to create a neural network which, given a 2D position, can correctly classify the data point (0 or 1).
- To keep things simple, we will begin with a one-neuron network.
- We will use the sigmoid function for our neuron, with the 2 values of the position variable, (x_1, x_2), as the inputs.
- We will use a technique called "Gradient Descent" to minimize the cost function, and find the best value of w_1, w_2 and b.

$$f(x_1, x_2) = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + b)}} = \frac{1}{1 + e^{-((w_1, w_2) \cdot (x_1, x_2) + b)}}$$

Gradient Descent

So, again, how do we train our network? We are trying to minimize our cost function:

$$
\begin{aligned}
C = \sum_i C_i &= \frac{1}{2} \sum_i \left(f(\mathbf{w} \cdot \mathbf{x}_i + b) - v_i \right)^2 \\
&= \frac{1}{2} \sum_i \left(\frac{1}{1 + e^{-\left((w_1, w_2) \cdot (x_1, x_2)_i + b \right)}} - v_i \right)^2
\end{aligned}
$$

Where, again, the v_i are the correct classifications for each $(x_1, x_2)_i$.

We will use an optization algorithm called 'gradient descent'. The idea behind gradient descent is to calculate the gradient of our cost function, with respect to the weights and biases, and then move "downhill".

Gradient descent, continued

Suppose that our function has only one parameter.

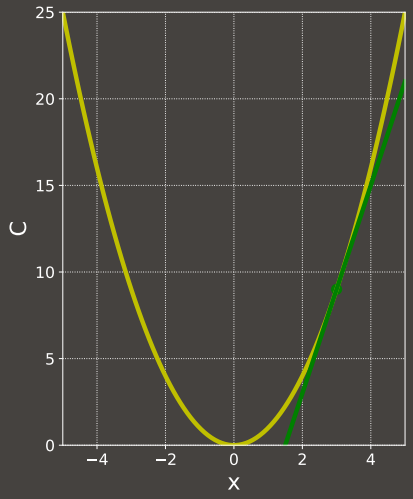
$$C = x^2$$

and we wish to minimize the function. Gradient descent says to move according to the formula:

$$x_{i+1} = x_i - \eta \frac{\partial C}{\partial x_i}$$

where η is called the step size. We then repeat until some stopping criterion is satisfied.

If we have multiple parameters (weights and biases), we step them all.



Gradient Descent, continued more

$$f_i = \frac{1}{1 + e^{-\left((w_1, w_2)\cdot(x_1, x_2)_i + b\right)}}$$

$$C = \sum_i C_i = \sum_i \frac{1}{2} \left(f_i - v_i\right)^2$$

So to find the minimum of our cost function, we need

$$\frac{\partial C}{\partial w_1} = \sum_i \left(f_i - v_i\right) f_i \left(1 - f_i\right) x_{1i} \qquad \frac{\partial C}{\partial w_2} = \sum_i \left(f_i - v_i\right) f_i \left(1 - f_i\right) x_{2i}$$

$$\frac{\partial C}{\partial b} = \sum_i \left(f_i - v_i\right) f_i \left(1 - f_i\right)$$

Gradient Descent, continued even more

So now

$$w_1 \to w_1 - \eta \sum_i (f_i - v_i) f_i (1 - f_i) x_{1i}$$
$$w_2 \to w_2 - \eta \sum_i (f_i - v_i) f_i (1 - f_i) x_{2i}$$
$$b \to b - \eta \sum_i (f_i - v_i) f_i (1 - f_i)$$

$$f_i = \frac{1}{1 + e^{-\left((w_1, w_2) \cdot (x_1, x_2)_i + b\right)}}$$

Now we're ready tackle the problem!

Training our neuron, code

```
# first_network.py
import numpy as np, numpy.random as npr

def sigma(x, model):
  z = model['w1'] * x[:,0] + \
      model['w2'] * x[:,1] + \
      model['b']
  return 1. / (1. + np.exp(-z))

def build_model(x, v, eta, num_steps = 10000,
  print_best = True):

  model = {'w1': npr.random(), \
           'w2': npr.random(), \
           'b' : npr.random()}
  scale = 100. / len(v)
  best = 0.0
  f = sigma(x, model)
```

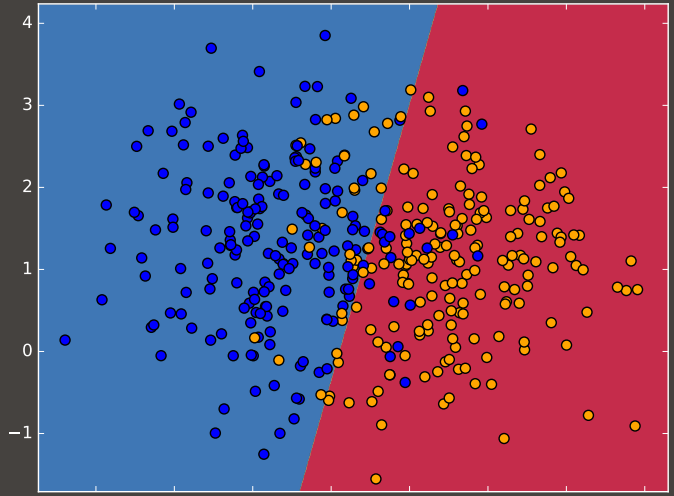
```
  for i in range(0, num_steps):
    # Calculate derivatives.
    dCdw1 = sum((f - v) * f * (1 - f) * x[:,0])
    dCdw2 = sum((f - v) * f * (1 - f) * x[:,1])
    dCdb = sum((f - v) * f * (1 - f))

    # Update parameters.
    model['w1'] -= eta * dCdw1
    model['w2'] -= eta * dCdw2
    model['b'] -= eta * dCdb

    f = sigma(x, model)
    score = sum(np.round(f) == v) * scale
    if (score > best):
      best, bestmodel = score, model.copy()

    # Print out, if requested.
  return bestmodel
```

Our first example, continued

Assume that we've still got our data in memory.

```
In [5]: import first_network as fn
In [6]:
In [6]: model = fn.build_model(train_pos, train_value, eta = 5e-5)
Best by step 0: 48.0 %
Best by step 1000: 87.2 %
Best by step 2000: 87.2 %
:
Best by step 7000: 87.2 %
Best by step 8000: 87.2 %
Best by step 9000: 87.2 %
Our best model gets 87.2 percent correct!
In [7]:
In [7]: pr.plot_decision_boundary(train_pos, train_value, model, fn.predict)
In [8]:
```

Our fit



Our first example, test data

```
In [8]:
In [8]: import numpy as np
In [9]:
In [9]: f = fn.predict(test_pos, model)
In [10]:
In [10]: sum(np.round(f) == test_value) / len(test_value)
Out[10]: 0.88026315789473684
In [11]:
```

88%!

This results for this example are strongly dependent on how separated your data's blobs are.

Some notes on our first example

A few observations about our first example.

- Our model only has three free parameters, $\boldsymbol{w_1}$, $\boldsymbol{w_2}$, $\boldsymbol{b}$. As such, it is only capable of a linear fit, for data with two independent variables.
- The reason why the fit worked as well as it did is because I separated the data enough that a linear split would be a reasonable thing to do.
- Depending on how well your data were split, your result may not have been as good.
- Note that using gradient descent wasn't even necessary, as we could have just used Least Squares to solve this particular problem exactly.
- If we want to be able to categorize more-complex data, such as hand-written digits, we're going to need a more-complex approach.

But wasn't that just logistic regression?

In actual fact, all we did was just a very long-winded version of logistic regression.

- As mentioned earlier, the "logistic function" (the sigmoid function) is used to perform 'logistic regression'.
- Logistic regression is a standard classification algorithm.
- Normally you would use the logistic regresssion model built into sklearn.linear_model, rather than code it yourself.
- The purpose of doing it the long way was to introduce the concepts of
  - a cost function
  - minimization algorithms (gradient descent)
- When we build proper neural networks, with more than one neuron, things will get considerably more complicated.

We're laying the ground work for much bigger things.

Summary

A summary of today's class.

- Neural networks are used for pattern recognition, often exceeding human performance.
- Neurons are just (nonlinear) functions. These functions contain trainable weights and biases.
- We use a loss function to measure the inaccuracy of a given neural network, for some data set.
- Gradient descent, and its variations, is commonly used to optimize the neural network's weights and biases, by minimizing the loss function.

We will continue developing these ideas next class.