Neural network programming: neural networks

Erik Spence

SciNet HPC Consortium

28 April 2022

SciNet compute • calcul CANADA

Today's code and slides

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

`https://scinet.courses/1210`

Click on the link for the class, under "Lectures" click on "Neural Networks".

The best contact address is courses@scinet.utoronto.ca.

Today's class

This class will cover the following topics:

- Introduction to fully-connected neural networks.
- Backpropagation algorithm.
- Example with fake data.
- Example on the MNIST data.

Please ask questions if something isn't clear.

SciNet compute • calcul CANADA

A review of last class

Recall what we did last class.

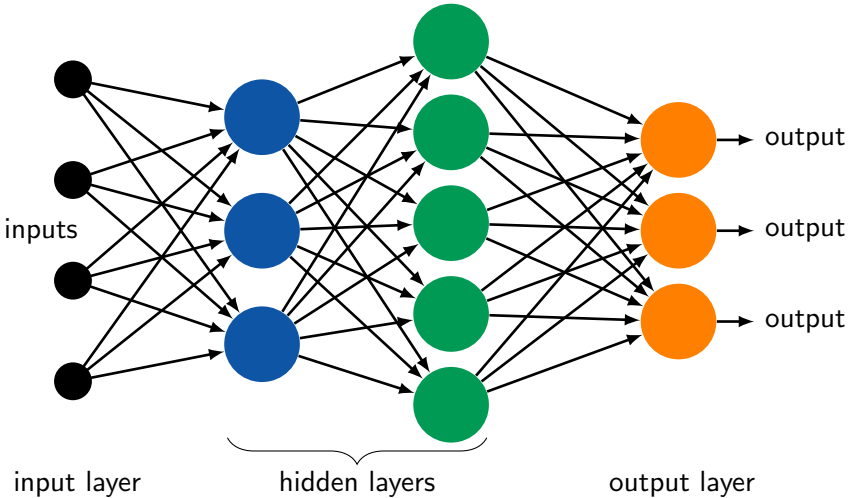
- We built a neural network, consisting of a single neuron.
- We defined a cost function, C, which measured the inaccuracy of the neural network's predictions.
- We created data that came in the form $(\mathbf{x}, y)$, so that we could perform supervised learning.
- We used a minimization algorithm, Gradient Descent, to minimize the cost function, using the data. The minimization was accomplished by modifying the neuron's weights and bias.
- We then tested the resulting network against the test data.

This was a good start, but now it's time to do real neural networks.

SciNet compute • calcul CANADA

Neural networks

Suppose we combine many neurons together, into a proper network, consisting of "layers".



Some notes about neural networks

Some details about the graphic on the previous slide:

- The input neurons do not contain any functions. They merely represent the input data being fed into the network. There is one input neuron for each "feature" in the data set ($\boldsymbol{x_1}$ and $\boldsymbol{x_2}$, for example).
- Each neuron in the "hidden" layers and the output layer all contain an "activation function" (such as sigmoid) with its own free parameters, $\mathbf{w}$ and $\boldsymbol{b}$.
- Each neuron outputs a single value. This output is passed to all of the neurons in the subsequent layer. This type of layer is known as a "fully-connected", or "dense", layer.
- The number of free parameters in the neurons in any given layer depends upon the number of neurons in the previous layer.
- The output from the output layer is aggregated into the desired form to calculate the cost function.

SciNet compute • calcul CANADA

Seriously?

You might legitimately wonder why on Earth we would think this would lead anywhere.

- As it happens, this topology is similar to some simple biological neural networks.
- Each layer takes the output of the previous layer as its input.
- Each layer makes "decisions" about the information that it receives.
- In this way the later layers are able to make more complex and abstract decisions than the earlier layers.
- A many-layered network can potentially make sophisticated decisions.

However, there are subtleties in training such a network.

SciNet compute • calcul CANADA

Training a neural network

How do we train such a network?

- Suppose that we decide to try to use gradient descent to train the network from three slides ago.
- Each of the neurons has its own set of free parameters, $\mathbf{w}$ and $\boldsymbol{b}$. There are lots of free parameters!
- To update the parameters we need to calculate every $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ *for every weight and bias, in every neuron!*
- But how do we calculate those derivatives, especially for the parameters associated with the neurons that are several layers away from the output?

Actually, as it happens, this is a solved problem.

The backpropagation algorithm

To find the gradients of the cost function with respect to the weights and biases we use the "backpropagation algorithm". First let's go over some terminology.

Let the input layer be the zeroth layer. If $\mathbf{x} \in \mathbb{R}^{500\times 2}$ is the input data, then let $\mathbf{a_1} \in \mathbb{R}^{m_1\times 500}$ be the vector of outputs from the m_1 neurons in the first (hidden) layer:

$$\mathbf{z_1} = \mathbf{w_1}\mathbf{x}^T + \mathbf{b_1} \qquad \mathbf{a_1} = \boldsymbol{\sigma}\left(\mathbf{z_1}\right)$$

with $\mathbf{w_1} \in \mathbb{R}^{m_1\times 2}$, $\mathbf{b_1} \in \mathbb{R}^{m_1\times 1}$ and $\boldsymbol{\sigma}(z)$ the sigmoid function. Similarly,

$$\mathbf{z_\ell} = \mathbf{w_\ell}\mathbf{a_{\ell-1}} + \mathbf{b_\ell} \qquad \mathbf{a_\ell} = \boldsymbol{\sigma}\left(\mathbf{z_\ell}\right)$$

with $\mathbf{w_\ell} \in \mathbb{R}^{m_\ell\times m_{(\ell-1)}}$, $\mathbf{b_\ell} \in \mathbb{R}^{m_\ell\times 1}$, $\mathbf{a_\ell} \in \mathbb{R}^{m_\ell\times 500}$, where m_ℓ is the number of neurons in the ℓth layer.

The backpropagation algorithm, continued

$$\delta_M = \frac{\partial C}{\partial z_M} = \nabla_{\mathbf{a}_M} C \circ \sigma'(\mathbf{z}_M)$$

is the "error" in the last (Mth) layer. Recall that

$$C = \frac{1}{2}\sum_i (a_{Mi} - y_i)^2,$$

and thus

$$\delta_M = (\mathbf{a}_M - \mathbf{y}) \circ \sigma'(\mathbf{z}_M).$$

We now claim that

$$\delta_\ell = \left[(\mathbf{w}_{\ell+1})^T \delta_{\ell+1}\right] \circ \sigma'(z_\ell)$$

Some algebra reveals that

$$\frac{\partial C}{\partial b_\ell} = \delta_\ell$$

and that

$$\frac{\partial C}{\partial w_\ell} = \mathbf{a}_{\ell-1}\delta_\ell$$

The derivation of these quantities is not too difficult, just algebra.

Note that we are now using a_M as the output of our network.

Our second example

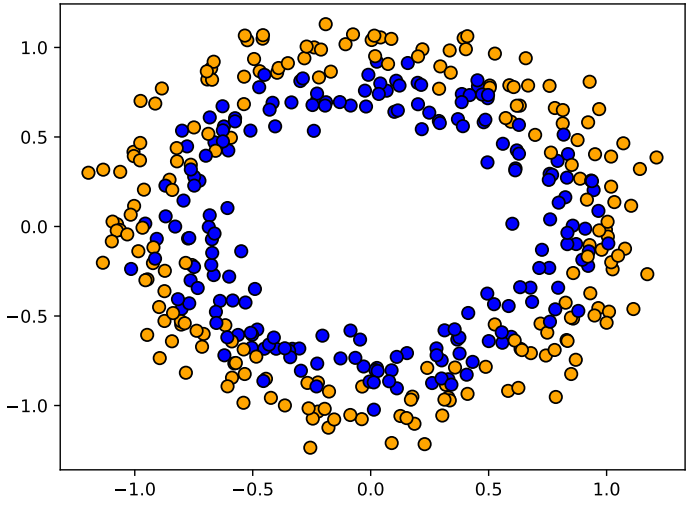
We use the *sklearn.datasets.make_circles* command to generate some toy data.

```
# example2.py
import sklearn.datasets as skd, sklearn.model_selection as skms

def get_data(n):
  pos, value = skd.make_circles(n, noise = 0.1)
  return skms.train_test_split(pos, value, test_size = 0.2)
```

```
In [1]: import example2 as ex2, plotting_routines as pr
In [2]:
In [2]: train_pos, test_pos, train_value, test_value = ex2.get_data(500)
In [3]:
In [3]: pr.plot_dots(train_pos, train_value)
In [4]:
```

SciNet compute • calcul CANADA

Our second example, data



The goal

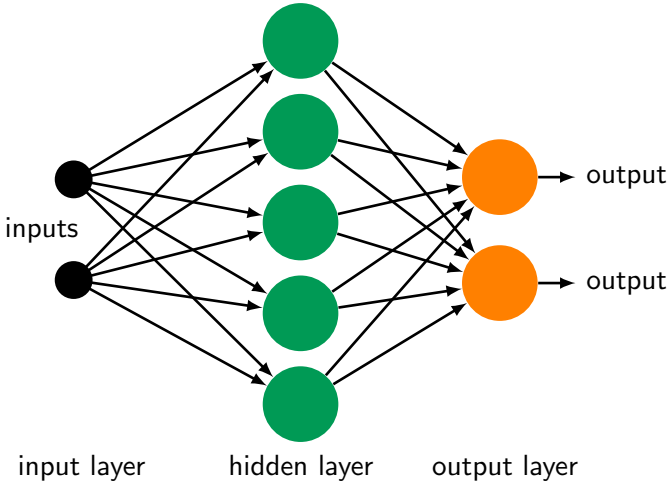
What are we trying to accomplish?

- Just like with our first example, we want to create a network which, given a 2D position, can correctly classify the data point (0 or 1).
- However, obviously a linear fit will not work in this case.
- This time we will create a network with three layers, an input layer, one hidden layer, and an output layer.
- We will use the sigmoid function for all neurons, with the 2 values of the position variable, $(\boldsymbol{x_1}, \boldsymbol{x_2})$, as the inputs to the hidden layer, and the outputs of the hidden layer as inputs to the output layer.
- Once again, we'll use gradient descent to minimize the cost function, to find the best values of our weights and biases.

SciNet compute • calcul CANADA

Our neural network

Note that the number of neurons in the hidden layer is arbitrary.



Training our network, code

```
# second_network.py
import numpy as np
import numpy.random as npr

# Sigmoid function.
def sigma(z):
  return 1. / (1. + np.exp(-z))

# Sigmoid prime function.
def sigmaprime(z):
  return sigma(z) * (1. - sigma(z))

# Returns just the predicted values.
def predict(x, model):
  _, _, _, a2 = forward(x, model)
  return np.argmax(a2, axis = 0)
```

```
# second_network.py, continued

# Predict the output value, given the model
# and the positions.
def forward(x, model):

  # Hidden layer.
  z1 = model['w1'].dot(x.T) + model['b1']
  a1 = sigma(z1)

  # Output layer.
  z2 = model['w2'].dot(a1) + model['b2']
  a2 = sigma(z2)

  return z1, z2, a1, a2
```

SciNet compute • calcul CANADA

Training our network, code, continued

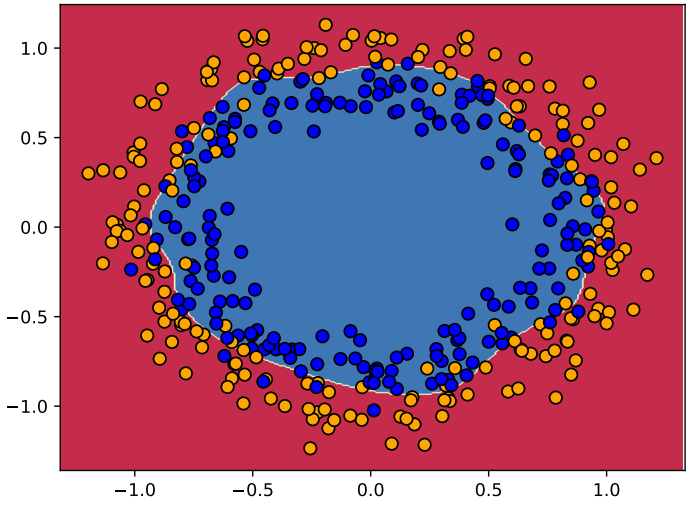
```
# second_network.py, continued
def build_model(num_nodes, x, y, eta, output_dim, num_steps = 10000, print_best = True):
  input_dim = np.shape(x)[1]
  model = {'w1': npr.randn(num_nodes, input_dim), 'b1': np.zeros([num_nodes, 1]), \
           'w2': npr.randn(output_dim, num_nodes), 'b2': np.zeros([output_dim, 1])}

  z1, z2, a1, a2 = forward(x, model)

  for i in range(0, num_steps):
    delta2 = a2;      delta2[y, range(len(y))] -= 1  # (a_M - y);      delta2 *= sigmaprime(z2)
    delta1 = (model['w2'].T).dot(delta2) * sigmaprime(z1)
    dCdb2 = np.sum(delta2, axis = 1, keepdims = True)
    dCdb1 = np.sum(delta1, axis = 1, keepdims = True)
    dCdw2 = delta2.dot(a1.T);     dCdw1 = delta1.dot(x)

    model['w1'] -= eta * dCdw1;      model['b1'] -= eta * dCdb1
    model['w2'] -= eta * dCdw2;      model['b2'] -= eta * dCdb2
    # Then check for best fit, and rerun the forward pass through the model.
```

Our second example, continued

Once again, assume that we've still got our data in memory.

```
In [4]: import second_network as sn
In [5]:
In [5]: model = sn.build_model(10, train_pos, train_value, eta = 5e-3, output_dim = 2)
Best by step 0: 51.2 %
Best by step 1000: 85.0 %
Best by step 2000: 86.0 %
⋮
Best by step 7000: 87.2 %
Best by step 8000: 87.2 %
Best by step 9000: 87.2 %
Our best model gets 87.5 percent correct!
In [6]:
In [6]: pr.plot_decision_boundary(train_pos, train_value, model, sn.predict)
In [7]:
```

Our fit



SciNet compute • calcul CANADA

Our second example, test data

```
In [7]:
In [7]: f = sn.predict(test_pos, model)
In [8]:
In [8]: sum(f == test_value) / len(test_value)
Out[8]: 0.82999999999999
In [9]:
```

83%!

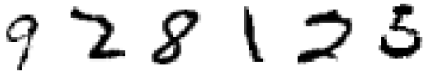
Some notes on our second example

A few observations about our second example.

- The choice of η was by trial-and-error. There are more-sophisticated techniques which can be used. We may discuss these in a later class.
- Our model has as many free parameters as you like, depending on the number of nodes you use. As such, it is capable of getting an extremely good fit.
- It is not uncommon for the number of parameters in the network to greatly exceed the number of observations. Your machine-learning instincts should be warning you: this situation is ripe for over-fitting.
- Nonetheless, there are techniques that are used to improve the generalization of the model. We'll visit these next class.

SciNet compute • calcul CANADA

Handwritten digits



One of the classic datasets on which to test neural-network techniques is the MNIST dataset.

- A database of handwritten digits, compiled by NIST.
- Contains 60000 training, and 10000 test examples.
- The training digits were written by 250 different people; the test data by 250 different people.
- The digits have been size-normalized and centred.
- Each image is grey scale, 28 x 28 pixels.

We can use our existing code to classify these digits.

Our network

How would we design a network to analyse this data?

- Each image is 28 x 28 = 784 pixels. Let the input layer consist of 784 input nodes. Each entry will consist of the grey value for that pixel.
- The output will consist of a one-hot-encoding of the networks analysis of the input data. This means that, if the input image depicts a '7', the output vector should be $[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]$.
- Thus, let there be 10 output nodes, one for each possible digit.
- To start, let's just use a single hidden layer.
- As it happens, the code which we used in the second example can solve this problem.

SciNet compute • calcul CANADA

Hand written digits, continued

The code for reading the data is contained in mnist_loader.py.

```
In [9]: import mnist_loader
In [10]: train_x, train_y, val_x, val_y, test_x, test_y =
...:         mnist_loader.load_mnist_1D_small('mnist.pkl.gz')
In [11]:
In [11]: train_x.shape
Out[11]: (500, 784)
In [12]: model = sn.build_model(30, train_x, train_y, output_dim = 10,
...:         eta = 5e-4, num_steps = 20000)     # Takes about 1 minute.
Best by step 0: 10.8 %
Best by step 1000: 46.4 %
Best by step 2000: 59.2 %
⋮
Best by step 19000: 85.6 %
Our best model gets 85.8 percent correct!
In [13]:
```

Handwritten digits, test data

```
In [13]:
```

```
In [13]: test_x.shape
Out[13]: (100, 784)
```

```
In [14]:
```

```
In [14]: f = sn.predict(test_x, model)
```

```
In [15]:
```

```
In [15]: sum(f == test_y) / len(test_y)
Out[15]: 0.58999999999999997
```

```
In [16]:
```

```
In [16]: # number of parameters in the model
In [17]: (784 + 1) * 30 + (30 + 1) * 10
Out[17]: 23860
```

```
In [18]:
```

59%! Not great. Clearly we have some over-fitting going on. How do we deal with this?

Over-fitting

Over-fitting occurs when a model is excessively fit to noise in the training data, resulting in a model which does not generalize well to the test data.

This can be a serious issue with neural networks. How do we deal with this?

- More data! Either real (original), or artificially created.
- Regularization.
- Dropout.

The first is self-explanatory. We'll go over the later two cases next week.

SciNet compute • calcul CANADA

Summary

Things to remember from today's class.

- Neural networks are built out of collections of neurons.
- Such networks are organized into layers: an input layer, an output layer, and an arbitrary number of hidden layers.
- Backpropagation is used to calculate the derivatives of the cost function with respect to the weights and biases.
- These derivatives are used in the Gradient Descent algorithm.
- Neural networks are prone to overfitting, especially when trained on an insufficient amount of data.
- This is the last time we will code the network details by hand.

SciNet compute • calcul CANADA

Linky goodness

Introductory Neural network classes:

- `http://neuralnetworksanddeeplearning.com`

Backpropagation:

- `http://colah.github.io/posts/2015-08-Backprop`
- `http://cs231n.github.io/optimization-2`

SciNet compute • calcul CANADA