Compute Ontario Summer School Artificial Neural Networks: introduction

Erik Spence

SciNet HPC Consortium

12 June 2024

Today's code and slides

You can get the slides and code for today's class at the 2024 Compute Ontario Summer School web site:

`https://training.computeontario.ca/courses/course/view.php?id=96`

Scroll down to this session. You can find the slides and code there.

Don't forget to mark yourself 'present' on the Attendance 'activity'.

The code for this morning's session is in "nn1_code.tar.gz".

Who am I?

My name is Erik Spence. I am assisted by Nastaran Shahparian and Collin Wilson.

- I am an Applications Analyst at SciNet (`https://www.scinethpc.ca`).
- SciNet is a High-Performance-Computing (HPC) consortium, one of six in Canada, run by the University of Toronto.
- Nastaran and Collin are with SHARCNET, another of Canada's HPC consortia.
- These consortia run massively parallel computers, with tens of thousands of cores, to perform computations that couldn't be done otherwise.
- Our job at SciNet and SHARCNET is to help users get their code to run on these machines.
- We also educate users on how to write fast, efficient code.

The Compute Ontario Summer School is presented by members of Ontario's three HPC consortia.

About this class

The purpose of this course is to introduce you to basic and intermediate neural network programming in Python.

- The material will start introductory and then build up to cover intermediate topics.
- We'll meet Wednesday and Thursday, June 12 and 13, 9:00am - 12:00pm, and Friday, June 14, 9:00am - 12:00pm and 1:30 - 4:30pm, online.
- Classes will consist of lectures with some hands-on sessions.
- All classes will be recorded, and the lecture material made available.
- Please mark yourself 'present' in the Attendance activity, near the top of the class web page.

Ask questions! But please put your questions in the Zoom chat!

About this class, continued

Software requirements for this class:

- I'll be using Python 3.10.X. Python 3.7–3.9 will also likely work.
- I won't be teaching Python syntax explicitly, unless asked.
- Starting tomorrow morning we'll be using the Keras neural-network programming framework, which is now built into TensorFlow. I will be using TensorFlow 2.12 ("pip install tensorflow").
- You will need the usual machine-learning packages: numpy, matplotlib, scikit-learn.

Today and tomorrow's classes will cover the basics of how neural networks work. Friday will be a survey of some of the types of neural networks. Ask questions!

Course topics

The next two days we will cover the following topics:

- Introduction to neural networks,
- Programming neural networks using Keras,
- Convolutional neural networks,
- Autoencoders,
- Diffusion networks,
- Recurrent Neural Networks,
- Attention networks, Transformers.

There are many many different families and classes of neural networks. We won't have time to cover them all.

This morning's class

This morning we will cover the following topics:

- Motivation for neural networks,
- Neurons,
- Gradient descent optimization,
- Training example,
- Hands-on session,
- Introduction to fully-connected neural networks,
- Backpropagation algorithm,
- Example with fake data,
- Example on the MNIST data,
- Hands-on session.

Please ask questions if something isn't clear.

Neural networks are already commonplace

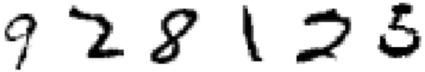
Neural networks are particularly good at detecting patterns, and for certain problems perform better than any other known class of algorithm. Neural networks are used for

- 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:

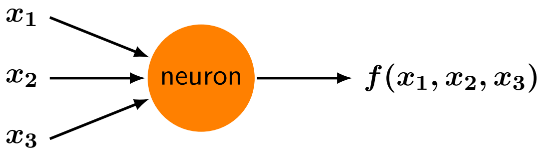
- Break the data set of numbers into two or three groups: training, testing, and optionally validation.
- Feed the training data to the neural network and train it to recognize one number from another.
- Let the neural network figure out the details for itself.

By the end of tomorrow you will 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 (x_1, x_2, x_3, for example) and returns a single output".



The function, called the "activation 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 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 more

What function should we use for f? One introductory function that is often used is the "sigmoid function" (also called the "logistic function").

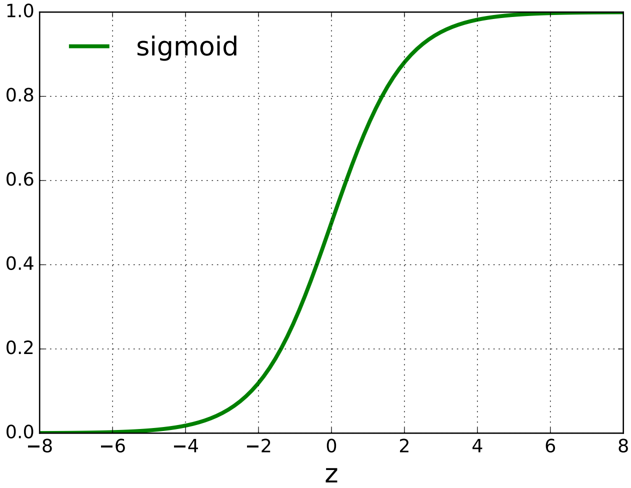
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

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'. Other functions are also an option, as we'll see later in this course.

Why the sigmoid function?



Because it ranges from 0 to 1 smoothly.

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):

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

where y_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 y.

Today's code

I'll be using ipython, running in the same directory that contains the code for today's class.

```
ejspence@mycomp ~> ls
mnist.pkl.gz   nn1_code.tar.gz
ejspence@mycomp ~>
ejspence@mycomp ~> tar -zxf nn1_code.tar.gz
ejspence@mycomp ~> ls
mnist.pkl.gz   nn1_code   nn1_code.tar.gz
ejspence@mycomp ~>
ejspence@mycomp ~> cd nn1_code
ejspence@mycomp nn1_code> ls
example1.py    first_network.py    mnist_loader.py     second_network.py
example2.py    plotting_routines.py
ejspence@mycomp nn1_code>
ejspence@mycomp nn1_code> ipython --pylab
In [1]:
```

Do not copy-and-paste code from PDFs!

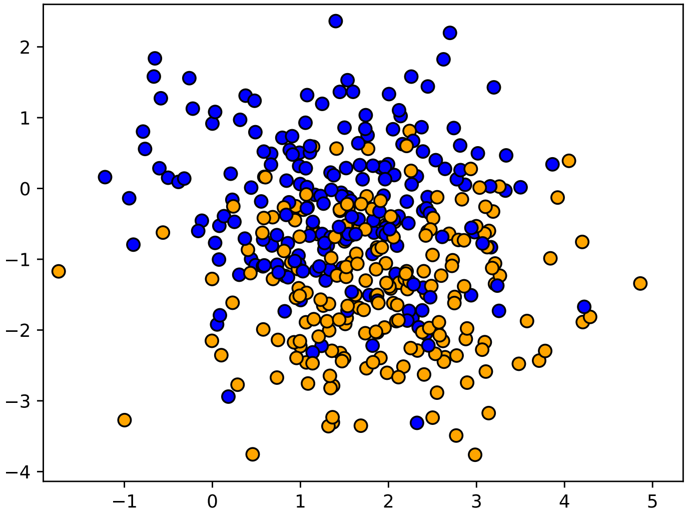
Our first example

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

```python
# 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), cluster_std = 1.0)
  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, $(\boldsymbol{x_1}, \boldsymbol{x_2})$, as the inputs.
- We will use a technique called "Gradient Descent" to minimize the cost function, and find the best value of $\boldsymbol{w_1}$, $\boldsymbol{w_2}$ and $\boldsymbol{b}$.

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

Gradient Descent

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

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

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

We will use an optimization 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.

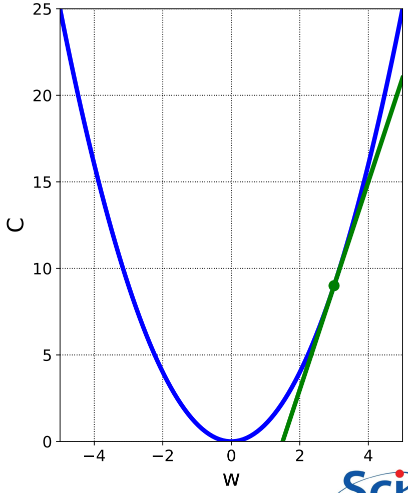
$$C = w^2$$

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

$$w_{j+1} = w_j - \eta \frac{\partial C}{\partial w_j}$$

where j is the iteration number and 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

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

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

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

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

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

Gradient Descent, continued even more

So now

$$\begin{aligned} w_1 &\to w_1 - \eta \sum_i (f_i - y_i)\, f_i\, (1 - f_i)\, x_{1i} \\ w_2 &\to w_2 - \eta \sum_i (f_i - y_i)\, f_i\, (1 - f_i)\, x_{2i} \\ b &\to b - \eta \sum_i (f_i - y_i)\, f_i\, (1 - f_i) \end{aligned}$$

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

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, y, eta, num_steps = 10000,
  print_best = True):

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

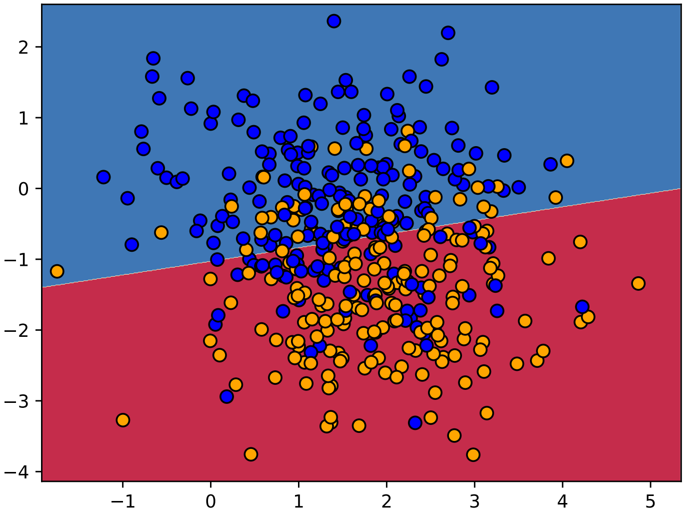
```
  for i in range(0, num_steps):
    # Calculate derivatives.
    dCdw1 = sum((f - y) * f * (1 - f) * x[:,0])
    dCdw2 = sum((f - y) * f * (1 - f) * x[:,1])
    dCdb = sum((f - y) * f * (1 - f))

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

    f = sigma(x, model)
    score = sum(np.round(f) == y) * 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 regression 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.

Hands-on 1

Let's take a break, and spend some time practising. What things can we modify to explore what we've learned so far?

- Change the noise level of the data (the 'cluster_std' argument).
- Change the bounding boxes around the centres of your data (the 'center_box' argument).
- Try other data-creating functions, such as
    - skd.make_moons(),
    - skd.make_classification(n_features = 2, n_redundant = 0, n_clusters_per_class = 1)

Well continue exploring these ideas after the break.

A review

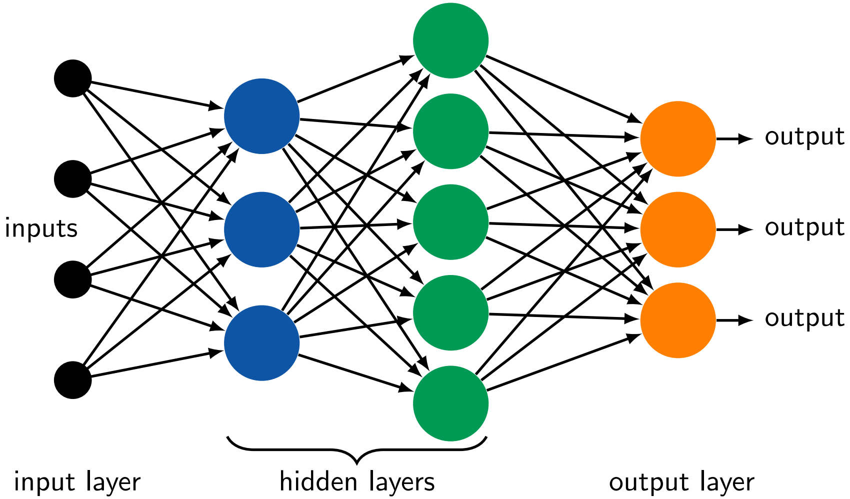
Let's review what we've done so far.

- 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.

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 activation function. 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.

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.

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

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

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

$$\boldsymbol{C} = \tfrac{1}{2} \sum_i \left(\boldsymbol{a_{Mi}} - \boldsymbol{y_i}\right)^2,$$

and thus

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

We now claim that

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

Some algebra reveals that

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

and that

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

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

Note that we are now using $\boldsymbol{a_M}$ as the output of our network.

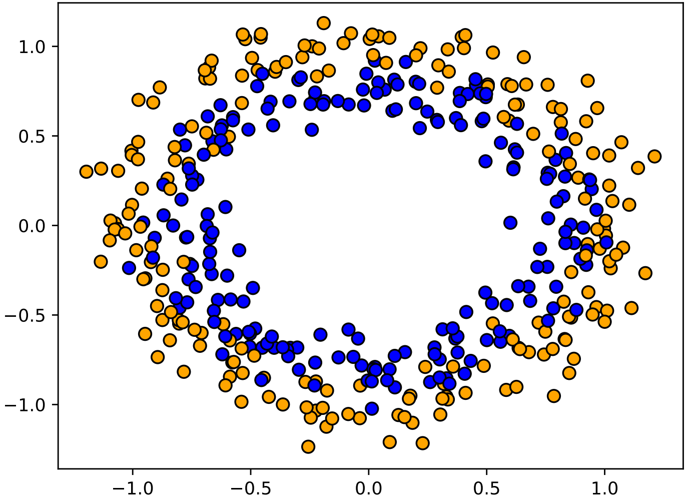
Our second example

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 [11]: import example2 as ex2, plotting_routines as pr
In [12]:
In [12]: train_pos, test_pos, train_value, test_value = ex2.get_data(500)
In [13]:
In [13]: pr.plot_dots(train_pos, train_value)
In [14]:
```

Our second example, data



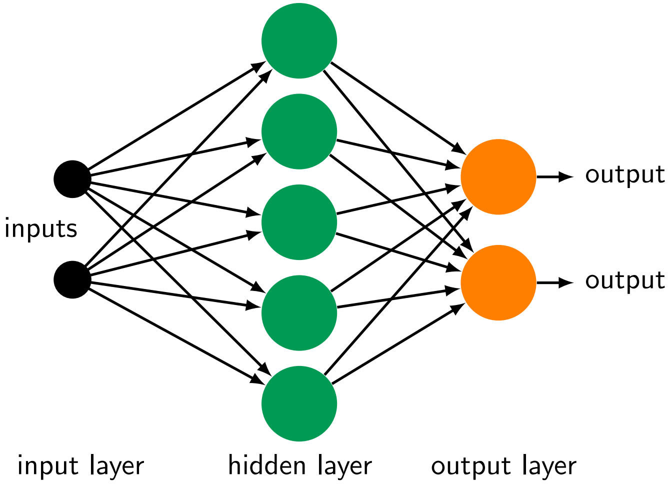
The goal

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, (x_1, 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.

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
```

Training our network, code, continued

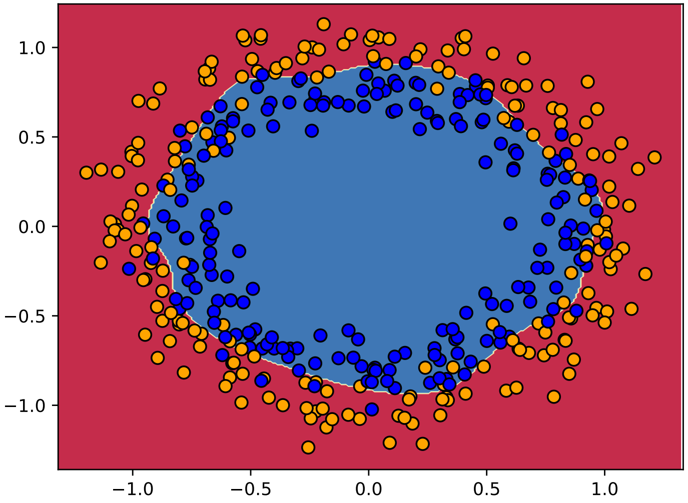
```
# second_network.py, continued
def build_model(num_nodes, x, y, eta, output_dim, num_steps = 10000, print_best = True):
  input_dim = np.shape(x)[1]
  model = {'w1': npr.randn(num_nodes, input_dim), 'b1': np.zeros([num_nodes, 1]),
           'w2': npr.randn(output_dim, num_nodes), 'b2': np.zeros([output_dim, 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 [14]: import second_network as sn

In [15]:

In [15]: 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 [16]:

In [16]: pr.plot_decision_boundary(train_pos, train_value, model, sn.predict)

In [17]:
```

Our fit



Our second example, test data

```
In [17]:
In [17]: f = sn.predict(test_pos, model)
In [18]:
In [18]: sum(f == test_value) / len(test_value)
Out[18]: 0.82999999999999
In [19]:
```

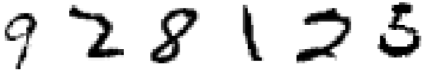
83%!

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.

Handwritten digits



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

- 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 analyze 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 $[\mathbf{0, 0, 0, 0, 0, 0, 0, 1, 0, 0}]$.
- 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.

Hand written digits, continued

The code for reading the data is contained in mnist_loader.py. Make sure you point to where the data file is stored.

```
In [19]: import mnist_loader
In [20]: train_x, test_x, train_y, test_y = mnist_loader.load_mnist_1D_small('../mnist.pkl.gz')
In [21]:
In [21]: train_x.shape
Out[21]: (500, 784)
In [22]: 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 19000: 85.6 %
Our best model gets 85.8 percent correct!
In [23]:
```

Handwritten digits, test data

```
In [23]:
In [23]: test_x.shape
Out[23]: (100, 784)
In [24]:
In [24]: f = sn.predict(test_x, model)
In [25]:
In [25]: sum(f == test_y) / len(test_y)
Out[25]: 0.58999999999999997
In [26]:
In [26]: # number of parameters in the model
In [27]: (784 + 1) * 30 + (30 + 1) * 10
Out[27]: 23860
In [28]:
```

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 most-often occurs when there are more free parameters in the model than there are 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 dropout tomorrow.

Summary

A summary of today's session.

- 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.
- 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.

Summary, continued

Our summary, continued:

- 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, since it is so easy to generate large neural networks.
- This is the last time we will code the network details by hand.
- We'll switch to a programming using a neural network framework next.

We will continue developing these ideas tomorrow.

Hands-on 2

Let's finish off with a break, and spend some time practising. What things can we modify to explore what we've learned so far?

- Change the number of neurons in our hidden layer. How does increasing or decreasing the number of neurons affect the
  - accuracy?
  - overfitting?
- Change the step size of our backpropagation. How does this affect the result?

We will continue to develop these ideas tomorrow.

Linky goodness

Introductory Neural network classes:

- http://neuralnetworksanddeeplearning.com

Backpropagation:

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