Introduction to neural network programming

Erik Spence

SciNet HPC Consortium

25 September 2017

Today's data, code and slides

You can get the slides 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".

The code and data for today's class are stored on our git repository.

git clone https://gitrepos.scinet.utoronto.ca/public/nn.git

Note that the data may take a moment to download. You can also alternatively get yourself set up on SciNet.

Getting set up on SciNet

```
ejspence@gpc-f103n084-ib0 ~>
ejspence@gpc-f103n084-ib0 ~> git clone https://gitrepos.scinet.utoronto.ca/public/nn.git
ejspence@gpc-f103n084-ib0 ~>
ejspence@gpc-f103n084-ib0 ~> cd nn/code
ejspence@gpc-f103n084-ib0 nn/code>
ejspence@gpc-f103n084-ib0 nn/code>  # Only do these two steps on SciNet!
ejspence@gpc-f103n084-ib0 nn/code> module load gcc/4.8.1 anaconda2/4.4.0
ejspence@gpc-f103n084-ib0 nn/code>
ejspence@gpc-f103n084-ib0 nn/code> ipython --pylab
In [1]:
```

Note that the code has been tested with Python 3, and appears to work.

About this class

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

- The material is introductory. If you are already using neural networks it's likely you are already familiar with much of this content.
- We'll be using Python 2.7.X.
- I'll be using the Keras neural-network programming framework, with a *Theano* back end (though you can use *TensorFlow* if you like).
- You will need the usual machine learning packages: numpy, matplotlib, scikit-learn.
- You'll obviously also need Theano (or Tensorflow) and Keras installed.

Ask questions!

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.
- Neural networks are based on a "biologically inspired" approach to solving such classification problems.
- This is one of the classic problems which have been solved using neural networks.

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 different approach:

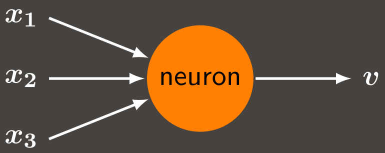
- Break the dataset of numbers into two or three groups: training, testing, and optionally validation.
- As with other supervised machine-learning algorithms, feed the training data to the neural network and train it to recognize one number from another.
- Rather than focus on details of the numbers, let the neural network figure out the details for itself.

This is the goal of the class. By the end of the class you should be acheiving greater than 98% 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? 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").

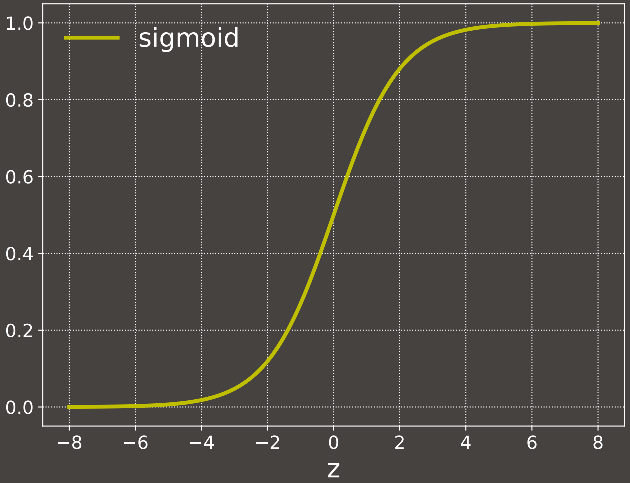
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

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 b.

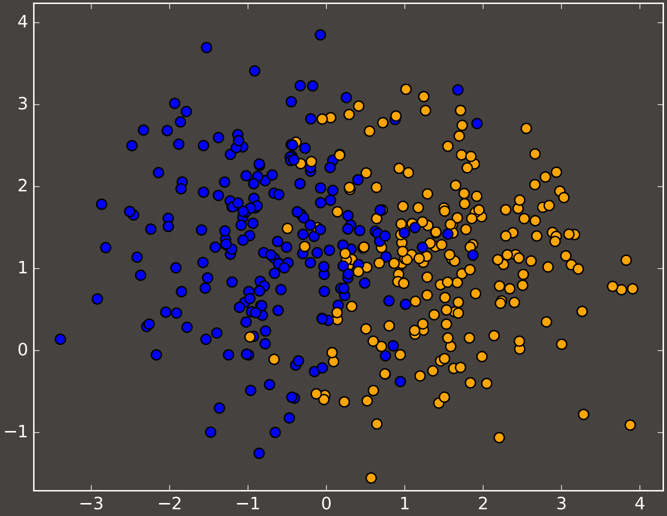
Our first example

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:

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

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

The idea behind gradient descent is to calculate the gradient of our function, and then move "downhill".

Gradient descent, continued

Suppose that our function has only one parameter.

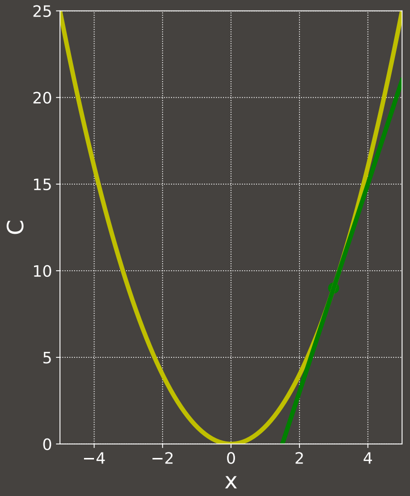
$C = x^2$

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, 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

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

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

Now we're ready tackle the problem!

Training our neuron, code

```python
# 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. / float(len(v))
  best = 0.0
  f = sigma(x, model)
```

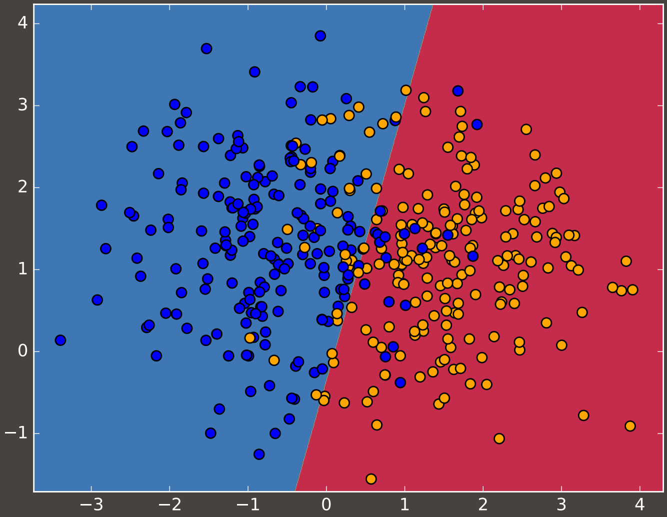
```python
  for i in xrange(0, num_steps):
    # Calculate derivatives.
    dCdw1 = sum((f - v) * f * (1 - f) * x[:,0])
    dCdw2 = sum((f - v) * f * (1 - f) * x[:,1])
    dCdb = sum((f - v) * f * (1 - f))

    # 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) / float(len(test_value))
Out[10]: 0.88026315789473684
In [11]:
```

88%!

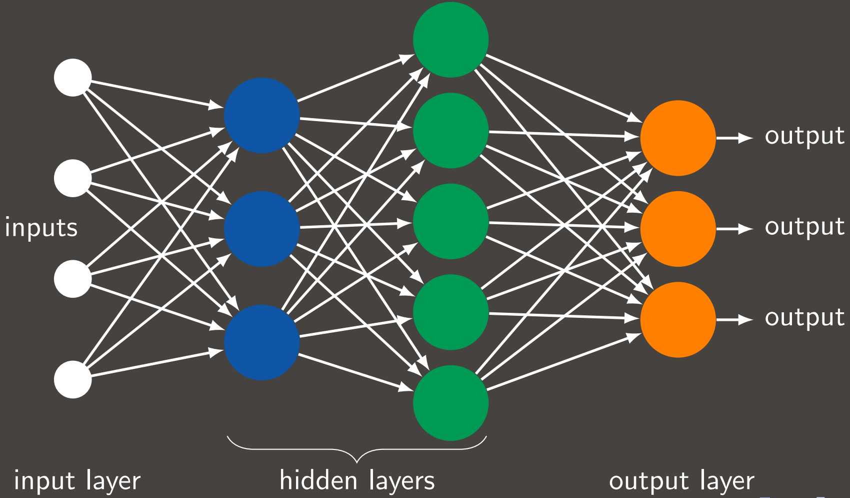
Some notes on our first example

A few observations about our first example.

- Our model only has three free parameters, w_1, w_2, 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.

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.
- Each neuron in the 'hidden' layers and the output layer all contain functions with their 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 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 b. There are lots of free parameters!
- To update the parameters we need to calculate every $\frac{\partial C}{\partial w_i}$ and $\frac{\partial C}{\partial b}$ *for 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 $\mathbf{a}^1 \in \mathbb{R}^{k\times 500}$ be the vector of outputs from the k 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 $\mathbf{w}^1 \in \mathbb{R}^{k\times 2}$ and $\mathbf{b}^1 \in \mathbb{R}^{k\times 1}$. 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

$$\delta^M = \frac{\partial C}{\partial z^M} = \nabla_{\mathbf{a}^M} C \circ \sigma'\left(\mathbf{z}^M\right)$$

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

$$C = \tfrac{1}{2} \sum_i \left(a_i^M - v_i\right)^2,$$

and thus

$$\delta^M = \left(\mathbf{a}^M - \mathbf{v}\right) \circ \sigma'\left(\mathbf{z}^M\right).$$

We now claim that

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

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 beyond the scope of this class, but it's not too difficult.

Note that we are now using a_i^M as the output of our network.

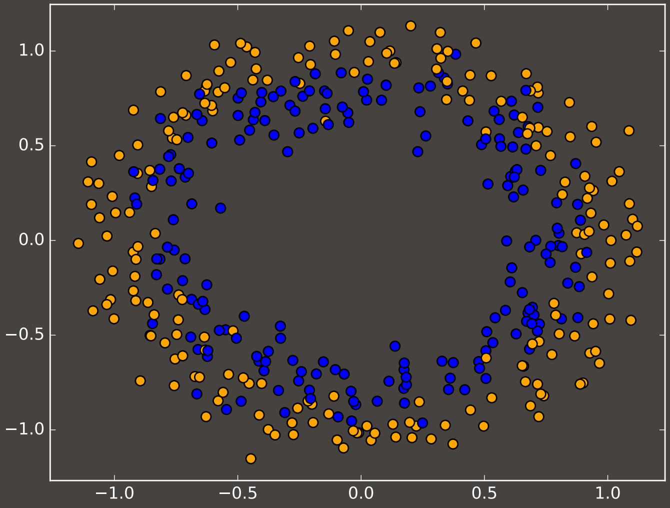
Our second example

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
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



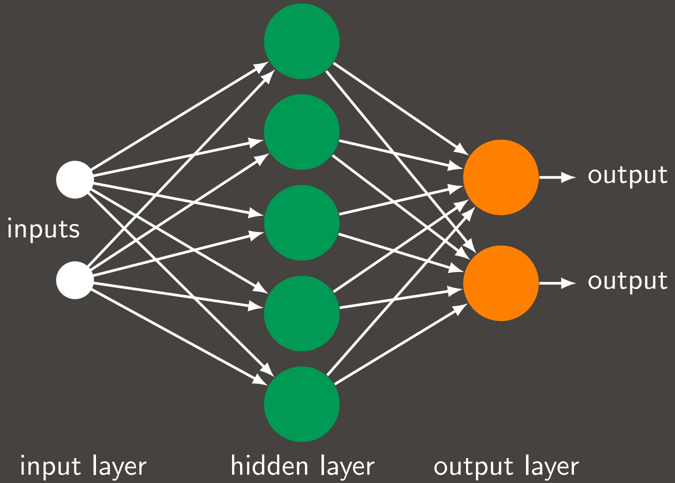
The goal

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

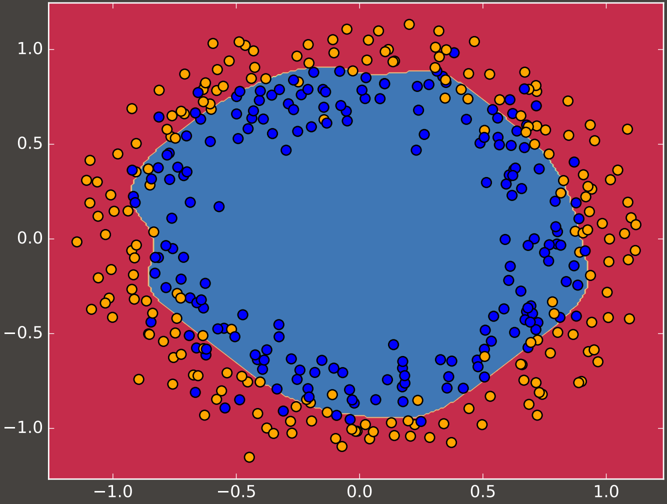
```
# second_network.py, continued
def build_model(num_nodes, x, v, eta, output_dim, num_steps = 10000, print_best = True, lam = 0):
  input_dim = np.shape(x)[1]
  model = {'w1': npr.randn(num_nodes, input_dim), 'b1': np.zeros([num_nodes, 1]), \
           'w2': npr.randn(output_dim, num_nodes), 'b2': np.zeros([output_dim, 1])}

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

  for i in xrange(0, num_steps):
    delta2 = a2;     delta2[v, range(len(v))] -= 1  # (a^M - v)
    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 * (lam * model['w1'] + dCdw1);     model['b1'] -= eta * dCdb1
    model['w2'] -= eta * (lam * model['w2'] + 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 = 0.01, output_dim = 2)
Best by step 0: 51.2 %
Best by step 1000: 83.8 %
Best by step 2000: 84.8 %
⋮
Best by step 7000: 85.5 %
Best by step 8000: 85.8 %
Best by step 9000: 86.0 %
Our best model gets 86.0 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) / float(len(test_value))
Out[18]: 0.72999999999999
In [19]:
```

73%!

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'll discuss these in the advanced NN 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 later in the class.

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

```
In [19]: import mnist_loader

In [20]: train_in, train_out, val_in, val_out, test_in, test_out = \
...:         mnist_loader.load_mnist_1D_small('../data/mnist.pkl.gz')

In [21]:

In [21]: train_in.shape
Out[21]: (500, 784)

In [22]: model = sn.build_model(30, train_in, train_out, output_dim = 10,
...:         eta = 4e-5, num_steps = 20000)     # Takes about 1 minute.
Best by step 0: 14.4 %
Best by step 1000: 64.7 %
Best by step 2000: 72.3 %
.
.
.
Best by step 19000: 98.6 %
Our best model gets 98.8 percent correct!

In [23]:
```

Handwritten digits, test data

```
In [23]:

In [23]: test_in.shape
Out[23]: (100, 784)

In [24]:

In [24]: f = sn.predict(test_in, model)

In [25]:

In [25]: sum(f == test_out) / float(len(test_out))
Out[25]: 0.8199999999999999

In [26]:

In [26]: # number of parameters in the model
In [27]: (784 + 1) * 30 + (10 + 1) * 2
Out[27]: 23572

In [28]:
```

82%! 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.

Regularization

Regularization is an *ad hoc* technique by which parameters in a fit are penalized to prevent individual parameters from becoming excessively important to the fit.

- This shows up all the time in contexts where over-fitting is to be expected, or the problem is ill-posed (inverse problems).
- It goes by many names: Tikhonov regularization, ridge regression...
- This usually manifests itself as a modification to the cost function, though there are other forms as well.

$$C = \frac{1}{2}\sum_i \left(a_i^M - v_i\right)^2 + \frac{\lambda}{2}\sum_j w_j^2$$

where λ is the "regularization parameter", and the sum over j is over all weights in the network.

Regularization, continued

If we call our previous cost function C_0, then the new cost function is

$$C = C_0 + \frac{\lambda}{2} \sum_j w_j^2$$

The addition of another term in the cost function changes the derivatives used to perform gradient descent.

$$\frac{\partial C}{\partial w_k} \to \frac{\partial C_0}{\partial w_k} + \lambda w_k$$

where the derivatives of C_0 are calculated the usual way, using backpropagation. Using gradient descent leads to

$$\begin{aligned} w_k &\to w_k - \eta \frac{\partial C_0}{\partial w_k} - \eta \lambda w_k \\ &\to (1 - \eta\lambda)\, w_k - \eta \frac{\partial C_0}{\partial w_k} \end{aligned}$$

Training our network, code, continued

```
# second_network.py, continued
def build_model(num_nodes, x, v, eta, output_dim, num_steps = 10000, print_best = True, lam = 0):
  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, _, a1, a2 = forward(x, model)

  for i in xrange(0, num_steps):
    delta2 = a2;     delta2[v, range(len(v))] -= 1  # (a^M - v)
    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 * (lam * model['w1'] + dCdw1);     model['b1'] -= eta * dCdb1
    model['w2'] -= eta * (lam * model['w2'] + dCdw2);     model['b2'] -= eta * dCdb2
    # Then check for best fit, and rerun the forward pass through the model.
```

Hand written digits, continued more

The code for the regularization is already part of second_network.py.

```
In [28]: model = sn.build_model(30, train_in, train_out, output_dim = 10,
...:         eta = 4e-5, num_steps = 20000, lam = 5)
Best by step 0: 9.2%
Best by step 1000: 66.8 %
Best by step 2000: 81.4 %
:
:
Best by step 19000: 93.2 %
Our best model gets 93.2 percent correct!
In [29]:
In [29]: f = sn.predict(test_in, model)
In [30]:
In [30]: sum(f == test_out) / float(len(test_out))
Out[30]: 0.87
In [31]:
```

Regularization, continued more

Why the improvement in the fitting of the test data?

- The regularization keeps the network from depending on any one particular parameter, or set of parameters, too much.
- This results in the network not focusing too much on any given feature, resulting in better generalization to the test data.
- Hyperparameter optimization techniques, such as those needed to choose a good value of λ, are beyond the scope of this class. Trial-and-error isn't a bad place to start, but you can do better.

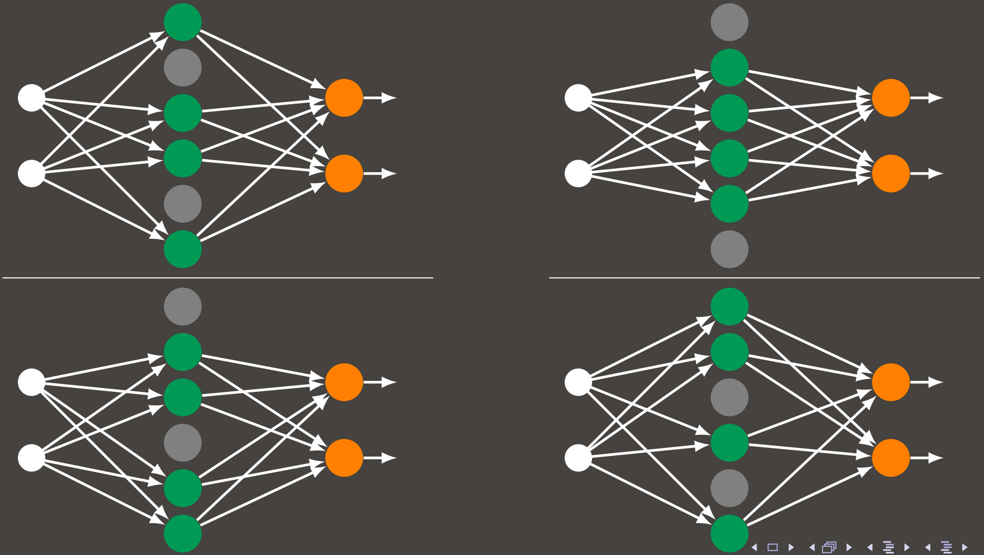
Regularization isn't used as often as it used to be. Dropout appears to much more popular an approach.

Dropout

Dropout is a uniquely-neural-network technique to prevent over-fitting.

- The principle is simple: randomly "drop out" neurons from the network, at each iteration of the minimization algorithm.
- Like regularization, this results in the network not putting too much importance on any given weight, since the weights keep randomly disappearing from the network.
- It can be thought of as averaging over several different-but-similar neural networks.
- Different fractions of different layers can be specified for dropout.
- Obviously, the output from the dropout layers must be scaled.
- We're not going to run an example of this.

Dropout, visualized



Neural network frameworks

Now that we have a sense of how neural networks work, we're ready to switch gears and use a 'framework'. Why would you do that?

- Coding your own networks from scratch can be a bit of work. (Though it's easier and cleaner if you use classes).
- Neural network (NN) frameworks have been specifically designed to solve NN problems.
- Python, of course, is not a high-performance language.
- The NN frameworks which have been developed are compiled before being used, thus being much faster than interpreted Python.
- The NN frameworks are also designed to use GPUs, which make things significantly faster than just using CPUs.

The training of neural networks is particularly well suited to solution using GPUs.

Theano

Let's review several of the more-popular NN frameworks out there. First is the Theano programming package.

- Theano is not strictly a NN framework. It was designed to handle multi-dimensional array manipulation.
- Developed by the LISA/MILA lab at the Université de Montréal.
- Written in Python; uses a numpy-like syntax, but generates C code which is compiled before being used.
- Supports symbolic differentiation.
- Is the grand-daddy of NN programming approaches.

Though commonly used in NN programming, it's been falling out of favour due to its long compile times (for some classes of problems) compared to other frameworks.

TensorFlow

TensorFlow is Google's NN framework.

- Released as open source in November 2015.
- The second-generation NN framework developed internally at Google.
- Allegedly capable of running on multiple cores.
- Provides APIs for Python, C++, Java and other languages.
- Can be quite slow.

This framework continues to grow in popularity.

Caffe

Caffe was developed as part of a Ph.D. project at U.C. Berkeley.

- Stands for Convolutional Architecture for Fast Feature Embedding.
- Currently maintained by the Berkeley Vision and Learning Center.
- Written in C++, but has Python and MATLAB interfaces.
- Particularly strong in the image-recognition and analysis area.
- Weak if you're not doing convolution networks (we'll get to this shortly), meaning you're studying text, sound, or time series.
- Only a few native input formats, and only one native output format, HDF5 (though you can get around this using the Python interface).

This is a commonly-encountered framework where ever images are being used.

Torch

Another framework used for neural networks is Torch.

- First released in 2002. Quite mature at this point.
- Written in Lua. Never heard of Lua? Welcome to the club.
- Pytorch was release by Facebook in January.
- Like Theano, Torch is more flexible than just NN. It is more of a generic scientific computing framework.
- Unlike Theano, Torch does not support symbolic differentiation.
- Very strong on GPUs.
- Very fast. Often the fastest depending on the problem being considered.
- Limited visualization capabilities.
- Used and maintained by Facebook, Twitter and other high-profile companies.

Keras

Because Theano is not a native NN framework, we will use Keras on top of Theano.

- Keras is a NN framework.
- It can run on top of either Theano or TensorFlow.
- Because it's a proper framework, all of the NN goodies you need are already built into it.
- Designed for fast development of networks.
- Is compatible with both Python 2 and 3.
- By default it will try to use TensorFlow on the back end; you may switch it if you wish.
  - On Macs/Linux, edit your $HOME/.keras/keras.json file.
  - On Windows, edit your %USERPROFILE%/.keras/keras.json file.
  - You can also set the KERAS_BACKEND environment variable.

Prepping the data

Theano needs the data in a specific format:

- If the input data is 2D, it must have a 'depth' added, to make it 3D, even if the depth is 1.
- If the input data is 1D no depth is needed.
- The labels must be changed to a categorical format (one-hot encoding).

```
In [31]:
In [31]: import keras.utils as ku
Using Theano backend.
In [32]:
In [32]: train_out2 = ku.to_categorical(train_out, 10)
In [33]: val_out2 = ku.to_categorical(val_out, 10)
In [34]:
```

Our network using Keras

Let us re-implement our second network using Keras.

- A "Sequential" model means the layers are stacked on one another in a linear fashion.
- A "Dense" ("fully-connected") layer is the regular layer we've been using.
- Use "input_dim" in the first layer to indicate the shape of the incoming data.
- The "activation" is the output function of the neuron.

```
# model1.py
import keras.models as km
import keras.layers as kl

def get_model(numnodes):
  model = km.Sequential()
  model.add(kl.Dense(numnodes, input_dim = 784,
    activation = 'sigmoid'))
  model.add(kl.Dense(10,
    activation = 'sigmoid'))
  return model
```

```
In [34]: import model1 as m1
In [35]: model = m1.get_model(30)
In [36]: model.output_shape
Out[36]: (None, 10)
In [37]:
```

Our network using Keras, continued

```
In [37]: import keras.optimizers as ko
```

```
In [38]: model.compile(optimizer = ko.SGD(lr = 1), metrics = ['accuracy'],
...:           loss = "mean_squared_error")
```

```
In [39]:
```

```
In [39]: fit = model.fit(train_in, train_out2, epochs = 100, batch_size = 10,
...:           validation_data = (val_in, val_out2), verbose = 2)
Train on 1000 samples, validate on 200 samples
Epoch 1/100
0s - loss: 0.0742 - acc: 0.4610 - val_loss: 0.0759 - val_acc: 0.4100
Epoch 2/100
0s - loss: 0.0709 - acc: 0.5250 - val_loss: 0.0732 - val_acc: 0.4150
⋮
Epoch 100/100
0s - loss: 0.0107 - acc: 0.9600 - val_loss: 0.0234 - val_acc: 0.8750
```

```
In [40]:
```

Note the over-fitting: training versus validation data.

About that optimization flag

The optimization flag was set to "opt.SGD".

- This stands for "Stochastic Gradient Descent".
- This is similar to regular gradient descent that we used previously.
  - Regular gradient descent is ridiculously slow on large amounts of data.
  - To speed things up, SGD uses a randomly-selected subset of the data (a "batch") to update the coefficients.
  - This is repeated many times, using different batches, until all of the data has been used. This is called an "epoch".
- In practise, regular gradient descent is never used, stochastic gradient descent is used instead, since it's so much faster.
- The only real advantage of regular gradient descent is that it's easier to code, which is why I used it previously.

Our network using Keras, continued

Some notes about the compilation of the model.

- We must specify the loss function with the "loss" argument.
- We must specify the optimization algorithm, using the "optimizer" flag.
- The optimizer can be specified explicitly, as in this example, or just put 'sgd', and get the default values.
- I specified it explicitly so that I could specify the value of η (using 'lr', the 'learning rate').
- The 'metrics' argument is optional, but is needed if you want the accuracy to be printed.

Our network using Keras, continued more

Now check against the test data.

Again we see the over-fitting rearing its head.

We can do better!

```
In [40]:
In [41]: test_out2 = ku.to_categorical(test_out, 10)
In [42]:
In [42]: score = model.evaluate(test_in, test_out2)
In [43]:
In [43]: score
Out[43]: [0.020172787383198738, 0.87]
In [44]:
```

The next steps

We can do better. What's the plan? There are a few simple approaches:

- Use more data.
- Change the activation function.
- Change the cost function.
- Change the optimization algorithm.
- Change the way things are initialized.
- Add regularization, to try to deal with the over-fitting.

We'll try some of these next, but there are some not-so-simple approaches:

- Completely overhaul the network strategy.

We'll take a look at this as well.

The next steps, an aside

There are alot of things we can tweak to make the network do better on the testing data. How do we know what to do?

- In many ways, implementing a network is an art.
- Certain forms and functions and parameters are known to lead to certain types of behaviour, and thus are used in certain situations.
- Choosing the correct values of parameters can often seem like a matter of trial-and-error.
- And choosing the correct activation functions, number of nodes, can also seem like trial-and-error.
- But there are more-sophisticated ways of finding the optimum parameter choices. We'll discuss this in the advanced NN class.

Practise is often needed to know how to approach various types of problems. Consult your colleagues, and the literature.

Other activation functions: softplus

Two commonly-used functions:

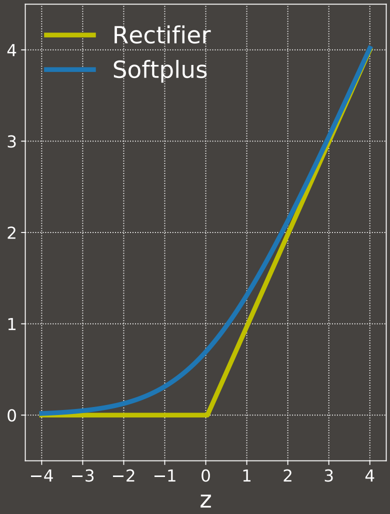
- Rectifier (also called Rectifier Linear Units, or RLUs):

  $$f(z) = \max(0, z).$$

- Softplus:

  $$f(z) = \ln(1 + e^z).$$

- Good: doesn't suffer from the vanishing-gradient problem.
- Bad: unbounded, could blow up.

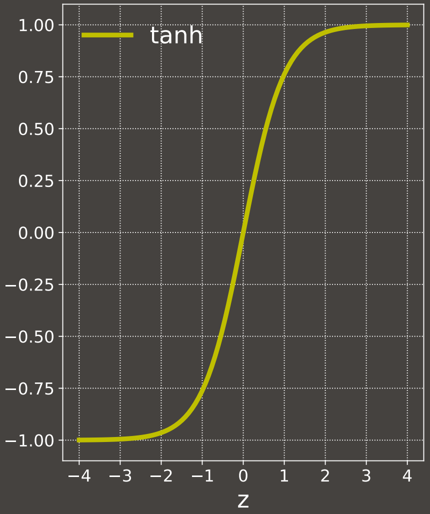


Other activation functions: tanh

Another commonly-used activation function is tanh:

$$f(z) = \tanh(z).$$

- Good: stronger gradients than sigmoid, faster learning rate, doesn't suffer from the vanishing-gradient problem.
- Good: because the function is anti-symmetric about zero. This also results in faster learning, at least for deeper networks.



Other activation functions: softmax

One of the more-commonly used output-layer activation functions is the softmax function:

$$s(z_j) = \frac{e^{z_j}}{\sum\limits_{k=1}^{N} e^{z_k}},$$

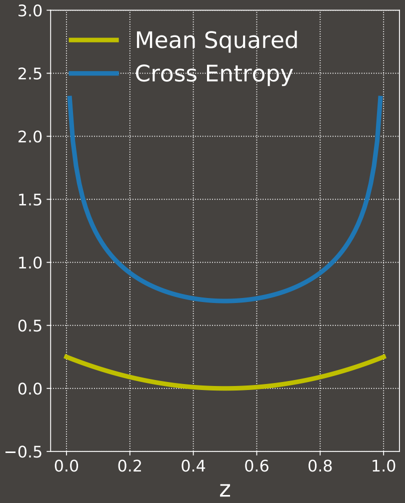
where N is the number of output neurons. The advantage of this function is that it converts the output to a probability.

Other cost functions: cross entropy

Probably the most-commonly used cost function is cross entropy:

$$C = -\frac{1}{n}\sum_i^n \left[v_i \log(a_i) + (1 - v_i)\log(1 - a_i)\right]$$

- Good: the gradient of cross entropy is directly proportional to the error; learning is faster than with mean squared error.
- Because $0 \leq a \leq 1$, this is usually used with softmax output.
- $v = 0.5$ in the example on the right.



Other optimization algorithms

There are many algorithms used to minimize the cost function.

- Gradient Descent, and its variations (RMSprop, Adam).
- Newton's method, uses the second derivatives of the cost function. Its variations ("Quasi-Newton") are gaining in popularity, especially DFP, and L-BFGS.
- Conjugate Gradient, which is like a combination of Gradient Descent and Newton's method.
- Levenberg-Marquardt (damped-least-squares) algorithm. Only works on squared cost functions (doesn't work on cross entropy).

If you find that your network won't train on a given optimization algorithm, it may be worth the effort to try a different one.

Our Keras network revisited

```
# model2.py
import keras.models as km, keras.layers as kl

def get_model(numnodes):
  model = km.Sequential()
  model.add(kl.Dense(numnodes, input_dim = 784, activation = 'tanh'))
  model.add(kl.Dense(10, activation = 'softmax'))
  return model
```

```
In [44]:
In [44]: import model2 as m2
In [45]:
In [45]: model = m2.get_model(30)
In [46]:
```

Our Keras network revisited, continued

```
In [46]: model.compile(loss = "categorical_crossentropy", optimizer = "sgd",
...:           metrics = ['accuracy'])

In [47]:

In [47]: train_in, train_out, val_in, val_out, test_in, test_out = \
...:          mnist_loader.load_mnist_1D_large('../data/mnist.pkl.gz')

In [48]:

In [48]: fit = model.fit(train_in, train_out, epochs = 100, batch_size = 20,
...:          validation_data = (val_in, val_out), verbose = 2)     # Takes about 4 minutes.
Train on 50000 samples, validate on 10000 samples
Epoch 1/100
2s - loss:  0.6853 - acc:  0.8261 - val_loss:  0.3783 - val_acc:  0.9021
Epoch 2/100
2s - loss:  0.3661 - acc:  0.9002 - val_loss:  0.3049 - val_acc:  0.9163
⋮
Epoch 100/100
2s - loss:  0.0543 - acc:  0.9860 - val_loss:  0.1342 - val_acc:  0.9633

In [49]:
```

Our Keras network revisited, continued more

Now check against the test data.

96%! Better!

This is about the best we can do with this approach, at least without using regularization. To push this even further, we need to change our network's architecture.

```
In [49]:
In [49]: score = model.evaluate(test_in, test_out)
In [50]:
In [50]: score
Out[50]: [0.12195164765194058, 0.96540000000000004]
In [51]:
```

Other topics

This workshop is short. There is not time to cover every topic. Some topics you should look into further, if you're going to use NNs in your research:

- preprocessing data: remove unnecessary degrees of freedom, scale and centre the data.
- parameter initialization: how the weights and biases are initialized matters.
- activation functions: there are several activation functions which are used in specific areas of neural networks. Learn which ones apply to your field.
- more cost functions: there are other cost functions which are used in specific fields.
- training failures: the disappearing gradient problem, the exploding gradient problem.

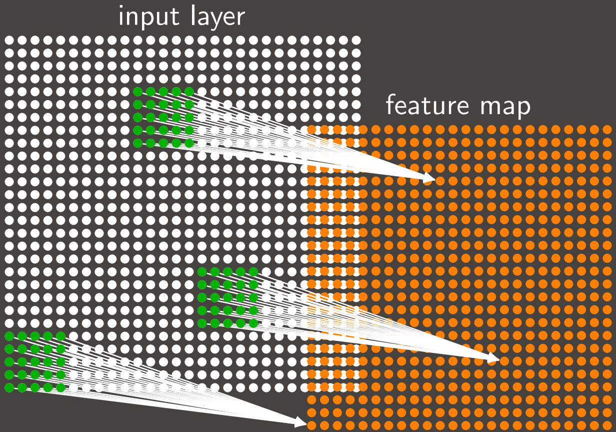
What next?

What we've done so far is pretty good, but it's not going to scale well.

- These are small images, and only black-and-white.
- Imagine we had a more-typical image size (200 x 200) and 3 colours? Now we're up to 120,000 input parameters.
- We need an approach that is more efficient.
- A good place to start would be an approach that doesn't throw away all of the spatial information.
- The data is 28 x 28, not 784 x 1.
- We should redesign our network to account for the spatial information. How do we do that?
- The first step called a Convolution Layer.

Convolution layers: feature maps

Create a set of neurons that, instead of using all of the input data, only takes input from a small area of the image. This is called a "feature map".



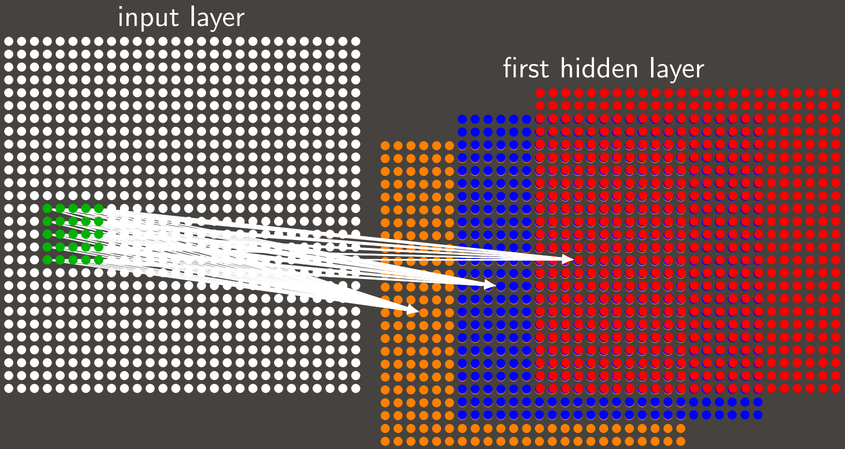
Feature maps

Some notes about feature maps.

- Notice that the feature map is smaller (24 x 24) than the input layer (28 x 28).
- The size of the feature map is partially set by the 'stride', meaning the number of pixels we shift to use as the input to the next neuron. In this case I've used a stride of 1.
- The **weights and biases are shared by all the neurons in the feature map**.
- Why? The goal is to train the feature map to recognize a single feature in the input, regardless of its location in the image.
- Consequently, it makes no sense to have a single feature map as the first hidden layer. Rather, multiple feature maps are used as the first layer.
- Feature maps are also called "filters" and "kernels".

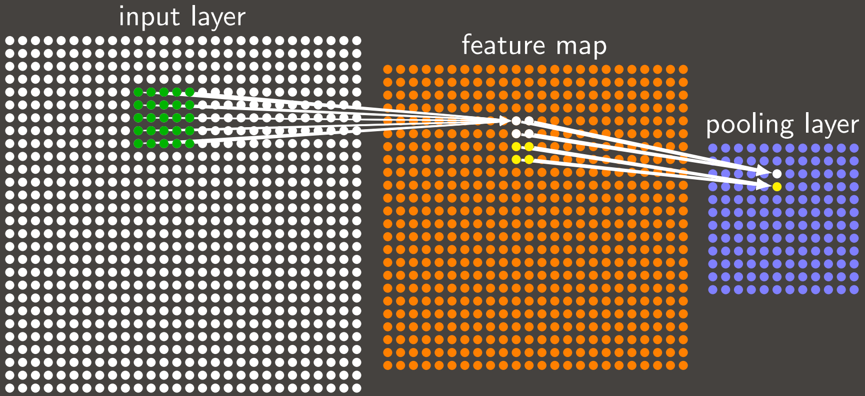
Convolution layers, continued more

The first hidden layer, a "convolution layer", consists of multiple feature maps. The same inputs are fed to the neurons in different feature maps.



Pooling layers

Each feature map is often followed by a "pooling layer".



In this case, 2 x 2 feature map neurons are mapped to a single pooling layer neuron.

Pooling layers, continued

Some notes about pooling layers.

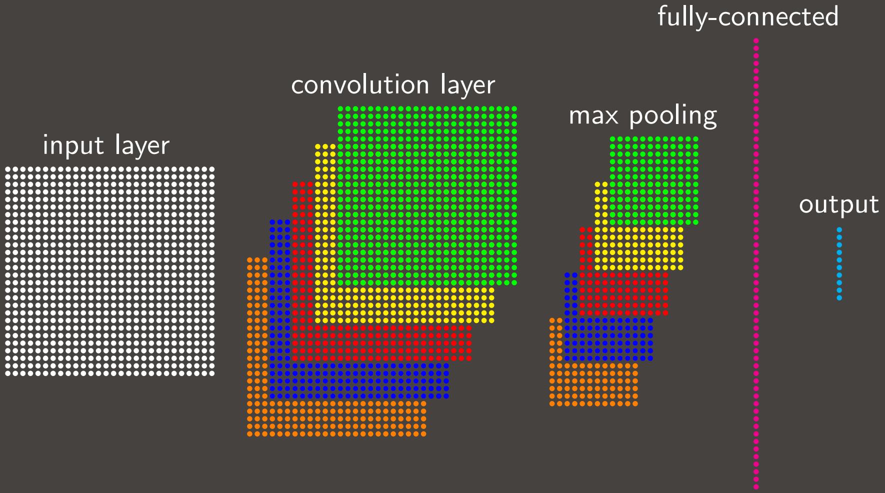
- The purpose of the layers is reduce the size of the data, and thus the number of parameters.
- The reduction in data also helps with over-fitting.
- Rather than use one of the activation functions we've already discussed, pooling layers use other functions.
- These functions do not have parameters in them which need to be fit.
- The most common function used is 'max', simply taking the maximum input value.
- Other functions are sometimes used, average pooling, L2-norm pooling.

2D data formatting

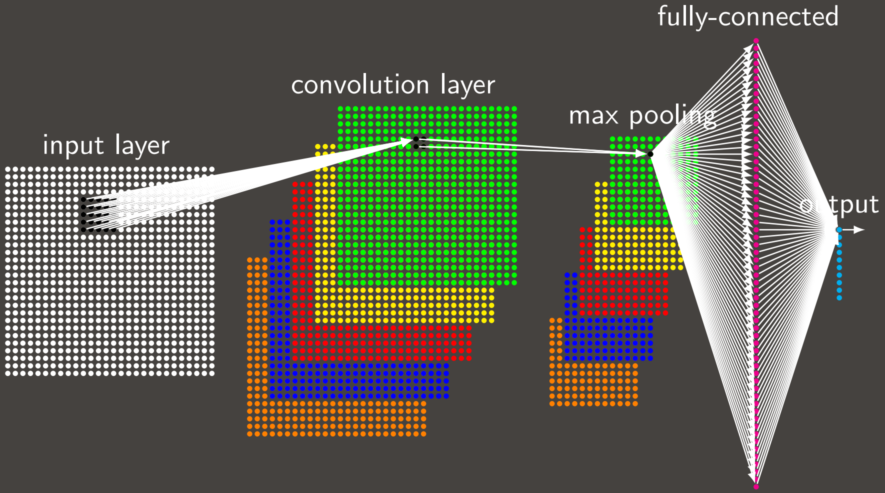
```
In [51]: train_in, train_out, val_in, val_out, test_in, test_out = \
...:          mnist_loader.load_mnist_2D('../data/mnist.pkl.gz')
In [52]: train_in.shape
Out[52]: (50000, 28, 28, 1)
In [53]:
In [53]: import keras.backend as K
In [54]:
In [54]: K.image_data_format()
Out[54]: 'channels_last'
In [55]:
```

Generally, 2D images are actually 3D, to deal with colours. Where the third dimension (the "channels") shows up in the dimensionality is indicated with the "image_data_format" function.

Our network, latest version



Our network, latest version



Our network revisited again

```
# model3.py
import keras.models as km, keras.layers as kl

def get_model(numfm, numnodes):

  model = km.Sequential()
  model.add(kl.Conv2D(numfm, kernel_size = (5, 5),
    input_shape = (28, 28, 1), activation = "relu"))

  model.add(kl.MaxPooling2D(pool_size = (2, 2),
    strides = (2, 2)))

  model.add(kl.Flatten())
  model.add(kl.Dense(numnodes, activation = "tanh"))
  model.add(kl.Dense(10, activation = "softmax"))
  return model
```

```
In [55]:
In [55]: import model3 as m3
In [56]:
In [56]: model = m3.get_model(20, 100)
In [57]:
```

The "Flatten" layer converts the 2D output to 1D, so that the fully-connected layer can handle it.

Our network revisited again, continued

```
In [57]: model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 24, 24, 20)        520
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 20)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2880)              0
_________________________________________________________________
dense_1 (Dense)              (None, 100)               288100
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params:  289,630.0
Trainable params:  289,630.0
Non-trainable params:  0.0
_________________________________________________________________
```

Our network revisited again, more

```
In [58]:
In [58]: model.compile(loss = "categorical_crossentropy", optimizer = "sgd",
...:            metrics = ['accuracy'])
In [59]:
In [59]: fit = model.fit(train_in, train_out, epochs = 30, batch_size = 20,
...:          validation_data = (val_in, val_out), verbose = 2)    # Takes about 15 minutes.
Train on 50000 samples, validate on 10000 samples
Epoch 1/30
33s - loss:  0.1219 - acc:  0.9658 - val_loss:  0.1083 - val_acc:  0.9707
Epoch 2/30
33s - loss:  0.0997 - acc:  0.9709 - val_loss:  0.0940 - val_acc:  0.9753
⋮
Epoch 30/30
17s - loss:  0.0060 - acc:  0.9996 - val_loss:  0.0464 - val_acc:  0.9866
In [60]:
```

Our network revisited again, some more

Now check against the test data.

98.87%! Only 113 / 10000 wrong! Not bad!

You can improve this even more by adding another convolution layer-max pooling layer after the first pair, adding dropout to both layers, and expanding the dataset.

But we'll leave that as an exercise.

```
In [60]:
In [60]: score = model.evaluate(test_in, test_out)
In [61]:
In [61]: score
Out[61]: [0.034107370334264121, 0.98870000000000002]
In [62]:
```

Notes on Convolution Networks

The previous network is called a Convolution Neural Network (CNN), and is quite common in image analysis.

- Often more than a single convolution layer-pooling layer combination will be used.
- This will lead to improved performance, in this case.
- In practice people come up with all manner of combinations of convolution, pooling and fully-connected layers in their networks.
- Trial-and-error is your friend here. Reviewing the literature, you will find themes, but also much art.

Using GPUs

An important note. Graphical Processing Units (GPUs) are particularly good at running NN-training calculations.

data size	CPU only		CPU-GPU	
	epoch time	total time	epoch time	total time
50000	41 s	21 min 4 s	4 s	2 min 43s
250000	198 s	100 min	26 s	15 min

These numbers are for our previous network. These were run on a Power 8 CPU, and a P100 GPU.

Unfortunately, multi-GPU functionality is not yet available in Keras running on Theano (though Theano and TensorFlow both support multiple GPUs).

Deep Learning

What is Deep Learning?

- Quite simply: a neural network with many hidden layers.
- Our last network probably qualified as Deep Learning, though barely.
- Up until the mid-2000s neural network research was dominated by "shallow" networks, networks with only 1 or 2 hidden layers.
- The breakthrough came in discovering that it was practical to train networks with a larger number of hidden layers.
- But it only became practical with the advent of sufficient computing power (GPUs) and easily-accessible huge data sets.
- State-of-the-art networks today can contain dozens of layers.

Other neural network types

What we've looked at are called feed-forward networks, since the data only goes in one direction. There are other classes of neural network too, which we will save for the advanced course.

- Recurrent Neural Networks (RNNs). These are networks where output from the network is used as input to the network. These are used for networks that must remember previous information.
- Modular Neural Networks. In this scenario various sub-networks are glued together through an intermediary network.
- Generative adversarial networks. Two networks work together, one generating data and one judging the data.
- And many many others....

Linky goodness

Neural network classes:

- `http://neuralnetworksanddeeplearning.com`
- `http://www.cs.utoronto.ca/~fidler/teaching/2015/CSC2523.html`
- `http://cs231n.stanford.edu`

Backpropagation:

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