CO Summer School Central: Introduction to neural networks with Python I

Erik Spence

SciNet HPC Consortium

26 June 2018

SciNet compute • calcul CANADA

Material for this class

The slides, code and data for this class can be found here:

`https://scinet.courses/451`

All the material for the Summer School can be found here:

`https://scinet.courses/438`

This is also the link to confirm your attendance for this class. Make sure that you confirm within 15 minutes of the end of the class; note that you can only confirm if you are registered for the class.

SciNet compute • calcul CANADA

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 will be using Python 3.X.
- We will be using the Keras neural-network programming framework. You may use whichever back end you like.
- You will need the usual machine learning packages: numpy, matplotlib, scikit-learn.
- You'll obviously also need a back end (Theano, Tensorflow, CNTK, *etc.*) and Keras installed.

Ask questions!

SciNet compute • calcul CANADA

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.

SciNet compute • calcul CANADA

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

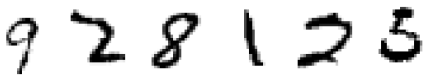
- Image recognition, object detection.
- Natural language processing (voice recognition).
- Novelty detection (detection of outliers).
- Next-word predictions.
- Text sentiment analysis.
- System control (self-driving cars).
- Medical diagnosis.

Neural networks are finding their way into everything.

SciNet compute • calcul CANADA

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.

SciNet compute • calcul CANADA

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.

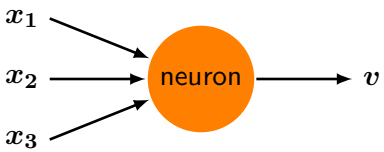
This is the goal of this class, and the class tomorrow. By the end of the second class you will be acheiving greater than 99% classification accuracy from your neural network on the hand-written digits data set.

But first, let's start with the basics.

SciNet compute • calcul CANADA

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):

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

where the sum is over all the data points, 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.

SciNet compute • calcul CANADA

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").

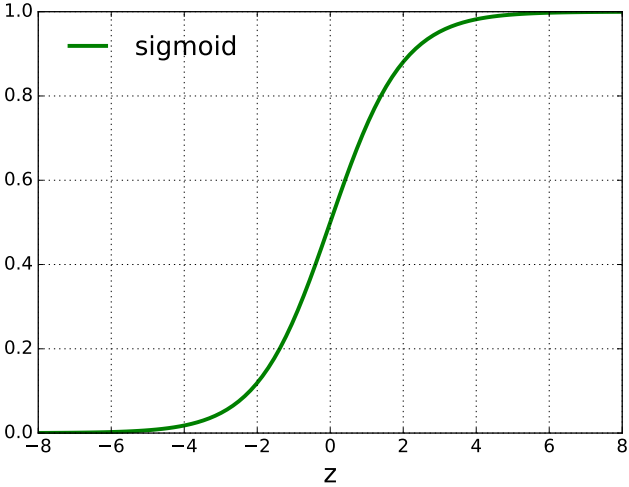
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

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.

SciNet compute • calcul CANADA

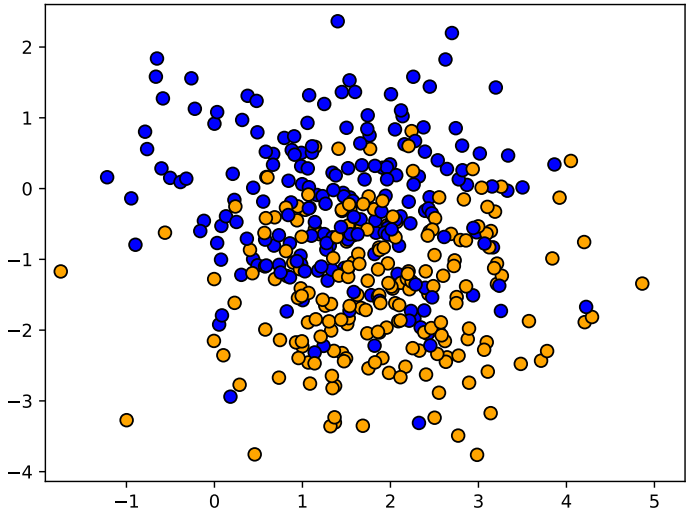
Our first example

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_x, test_x, train_y, test_y = ex1.get_data(500)
In [3]:
In [3]: train_x.shape, train_y.shape, train_x[0], train_y[0]
Out[3]: ((400,2), (400,), array([ 2.10552892, 1.31996395]), 1)
In [4]:
In [4]: pr.plot_dots(train_x, train_y)
```

Our data



The goal

What are we trying to accomplish?

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

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

Gradient Descent

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

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

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

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

Gradient descent, continued

Suppose that our function has only one parameter.

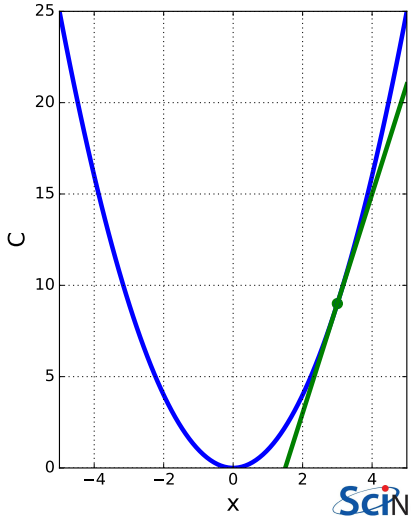
$$C = x^2$$

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

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

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

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



Gradient Descent, continued more

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

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

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

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

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

Gradient Descent, continued even more

So now

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

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

Now we're ready tackle the problem!

Training our neuron, code

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

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

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

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

```
for i in range(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_x, train_y, 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!
```

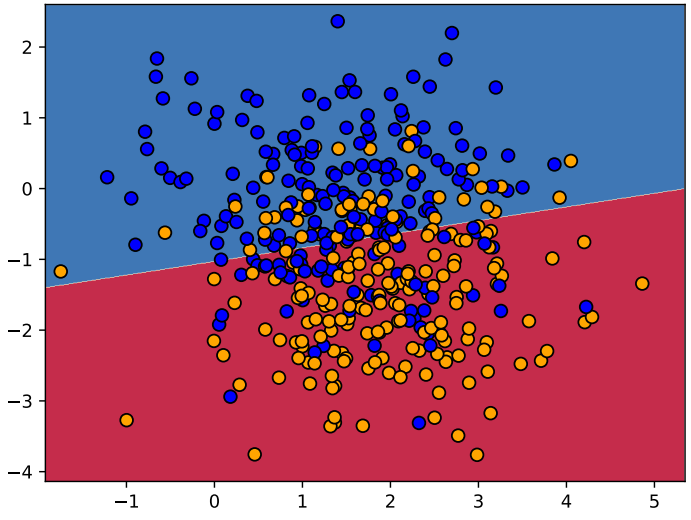
```
In [7]:
```

```
In [7]: pr.plot_decision_boundary(train_x, train_y, model, fn.predict)
```

```
In [8]:
```

SciNet compute • calcul CANADA

Our fit



Our first example, test data

```
In [8]:
In [8]: import sklearn.metrics as skm
In [9]:
In [9]: test_pred = fn.predict(test_x, model)
In [10]:
In [10]: skm.accuracy_score(test_y, test_pred)
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.
- Note that, strictly speaking, all we've done is logistic regression on the data.
- However, if we want to be able to categorize more-complex data, such as hand-written digits, we're going to need to generalize this to a more-complex approach.

SciNet compute • calcul CANADA

But wasn't that just logistic regression?

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

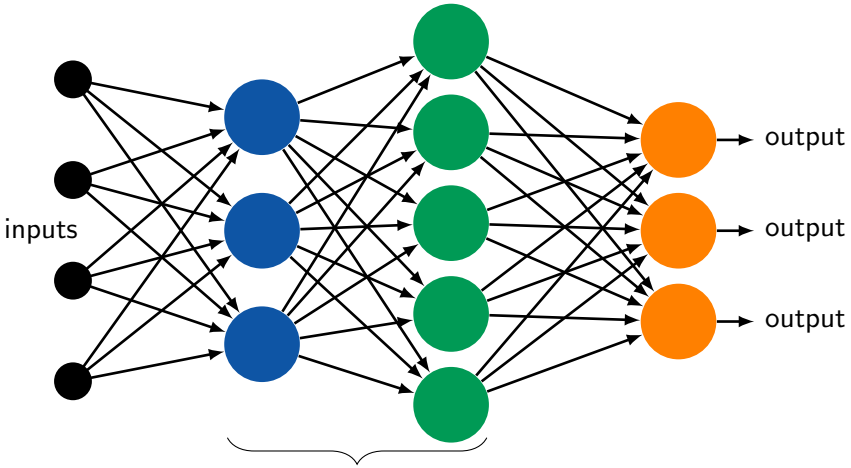
- As mentioned earlier, the "logistic function" (the sigmoid function) is used to perform 'logistic regression'.
- Logistic regression is a standard classification algorithm.
- Normally you would use the logistic regresssion model built into sklearn.linear_model, rather than code it yourself.
- The purpose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

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

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.
- 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.

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 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 $\frac{\partial C}{\partial w_i}$ 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.

SciNet compute • calcul CANADA

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:

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

with $\mathbf{w_1} \in \mathbb{R}^{k\times 2}$ and $\mathbf{b_1} \in \mathbb{R}^{k\times 1}$. Similarly,

$$\mathbf{z}_\ell = \mathbf{w}_\ell \mathbf{a}_{\ell-1} + \mathbf{b}_\ell \qquad \mathbf{a}_\ell = \sigma\left(\mathbf{z}_\ell\right)$$

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

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

and thus

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

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

Note that we are now using a_{Mi} as the output of our network.

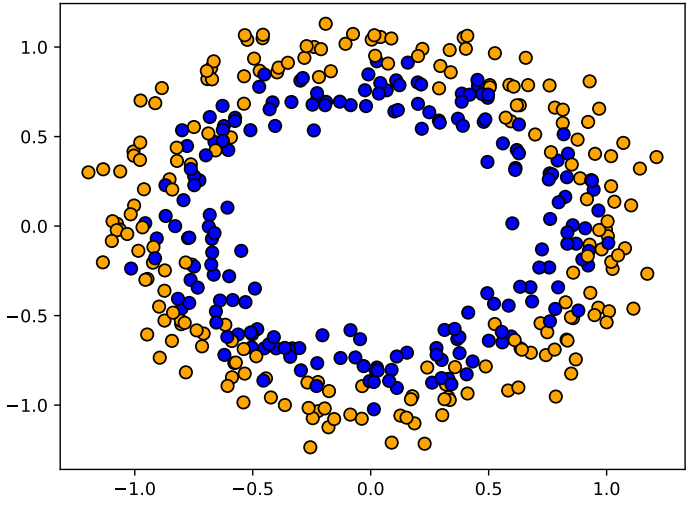
Our second example

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_x, test_x, train_y, test_y = ex2.get_data(500)
In [13]:
In [13]: pr.plot_dots(train_x, train_y)
In [14]:
```

Our second example, data



The goal

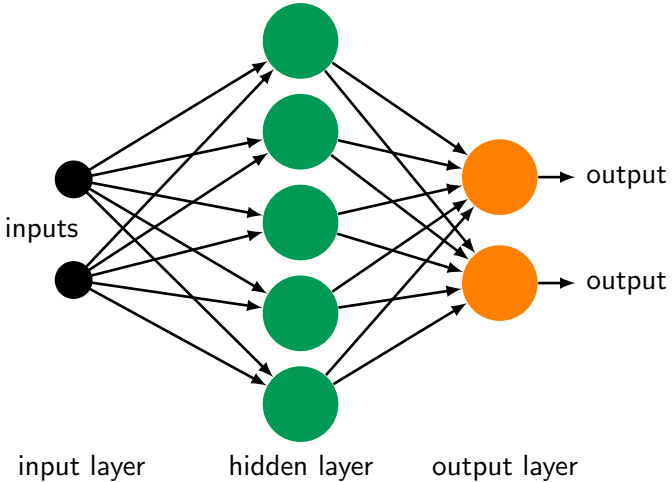
What are we trying to accomplish?

- 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.

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

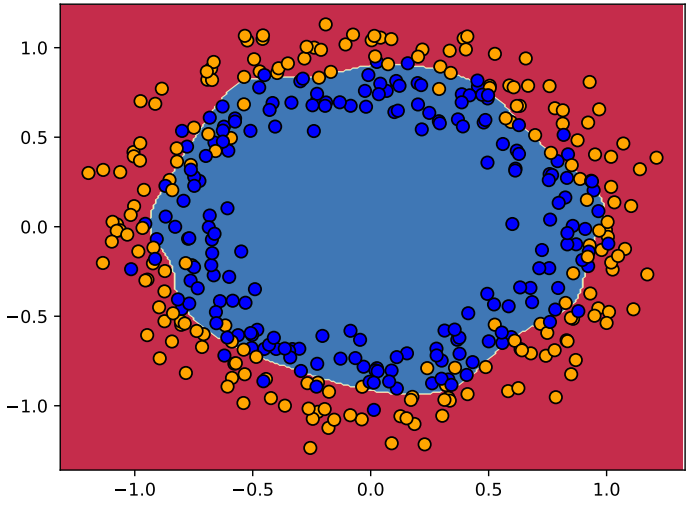
```
# second_network.py, continued
def build_model(num_nodes, x, v, eta, output_dim, num_steps = 10000, print_best = True):
  input_dim = np.shape(x)[1]
  model = {'w1': npr.randn(num_nodes, input_dim), 'b1': np.zeros([num_nodes, 1]), \
           'w2': npr.randn(output_dim, num_nodes), 'b2': np.zeros([output_dim,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

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

  for i in range(num_steps):
    delta2 = a2;      delta2[v, range(len(v))] -= 1  # (a_M - v);      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_x, train_y, eta = 0.01, 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_x, train_y, model, sn.predict)
In [17]:
```

Our fit



Our second example, test data

```
In [17]:
In [17]: test_pred = sn.predict(test_x, model)
In [18]:
In [18]: skm.accuracy_score(test_y, test_pred)
Out[18]: 0.82999999999999
In [19]:
```

83%!

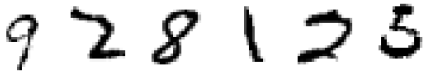
Some notes on our second example

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.
- 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.

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.

SciNet compute • calcul CANADA

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 [19]: import mnist_loader
In [20]: train_x, train_y, val_x, val_y, test_x, test_y = \
...:         mnist_loader.load_mnist_1D_small('../data/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 2000: 59.2 %
⋮
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]: test_pred = sn.predict(test_x, model)
```

```
In [25]:
```

```
In [25]: skm.accuracy_score(test_y, test_pred)
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 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 in a few slides.

SciNet compute • calcul CANADA

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 we 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 neural networks which are built using frameworks are compiled before being used, thus being much faster than 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 GPUs.

SciNet compute • calcul CANADA

TensorFlow

TensorFlow is Google's NN framework.

- Released as open source in November 2015.
- The second-generation machine-learning framework developed internally at Google, successor to DistBelief.
- More flexible than some other neural network frameworks.
- Capable of running on multiple cores and GPUs.
- Provides APIs for Python, C++, Java and other languages.
- Can be a bit of challenge to learn (many ways to do the same thing).
- With Tensorflow 2.0, Keras has become the main API.

This framework is popular, though not necessarily the fastest.

SciNet compute • calcul CANADA

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.
- Weaker if you're not doing convolution networks (we'll get to this next week), 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 wherever images are being used.

SciNet compute • calcul CANADA

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 2018.
- Torch is more flexible than just neural networks. It is more of a generic scientific computing framework.
- 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.

This framework is growing in popularity, through the pytorch interface.

SciNet compute • calcul CANADA

Keras

We will use Keras on top of your favourite backend.

- Keras is a NN framework, but it's only the top-most level.
- More accurately, it's an API standard for creating neural networks.
- It runs on top of a 'back end': Theano, TensorFlow, CNTK, MXNet, TypeScript, JavaScript, PlaidML, Scala, CoreML, and others.
- Because it's a proper framework, all of the NN goodies you need are already built into it.
- Very popular; 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.
- Use the usual "pip install keras" to install it from the Linux command line.

SciNet compute • calcul CANADA

Getting the data, using Keras

Because it is so commonly used, the MNIST data set is built into most NN frameworks.

Keras is no different, so we'll just grab it from Keras directly.

```
In [28]:
In [28]: from keras.datasets import mnist
Using Theano backend.
In [29]:
In [29]: (x_train, y_train), (x_test, y_test) = \
                    mnist.load_data()
In [30]:
In [30]: x_train.shape
Out[30]: (60000, 28, 28)
In [31]:
```

Prepping the data

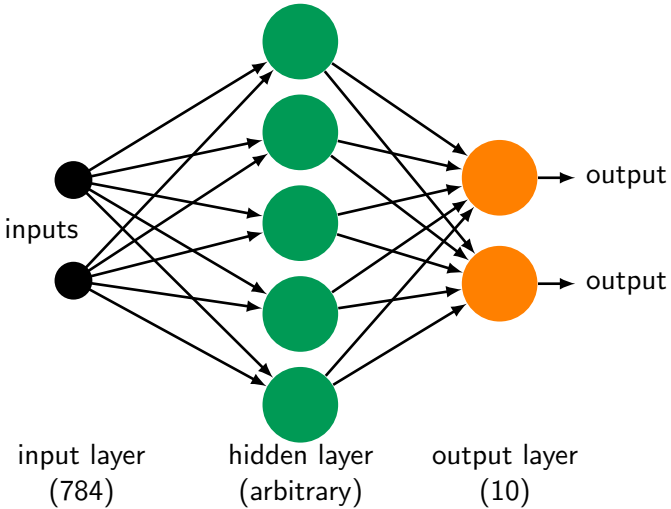
As with last time, we need the data in a specific format:

- Instead of 28 x 28, we flatten the data into a 784-element vector.
- We only take the first 500 data points for training, to be consistent with the hand-coded example.
- The labels must be changed to a categorical format (one-hot encoding).

```
In [31]:
In [31]: import keras.utils as ku
In [32]:
In [32]: x_train2 = x_train[0:500, :, :].reshape(500, 784)
In [33]: x_test2 = x_test[0:100, :, :].reshape(100, 784)
In [34]:
In [34]: y_train2 = ku.to_categorical(y_train[0:500], 10)
In [35]: y_test2 = ku.to_categorical(y_test[0:100], 10)
In [36]:
```

SciNet compute • calcul CANADA

Our neural network



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.
- The "name" of the layer is optional.

```
# 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', name = 'hidden'))
  model.add(kl.Dense(10, name = 'output',
    activation = 'sigmoid'))
  return model
```

```
In [36]: import model1 as m1
In [37]: model = m1.get_model(30)
In [38]: model.output_shape
Out[38]: (None, 10)
In [39]:
```

Our network using Keras, continued

```
In [39]:
```

```
In [39]: model.summary()
Layer (type)                 Output Shape              Param #
=================================================================
hidden (Dense)               (None, 30)                23550
_________________________________________________________________
output (Dense)               (None, 10)                310
=================================================================
Total params:  23,860
Trainable params:  23,860
Non-trainable params:  0
_________________________________________________________________
```

```
In [40]:
```

Our network using Keras, continued more

```
In [40]:
In [40]: model.compile(optimizer = 'sgd', metrics = ['accuracy'], loss = "mean_squared_error")
In [41]:
In [41]: fit = model.fit(x_train2, y_train2, epochs = 1000, batch_size = 5, verbose = 2)
Epoch 1/1000
0s - loss: 0.1963 - acc: 0.1170
Epoch 2/1000
0s - loss: 0.1338 - acc: 0.1720
⋮
Epoch 999/1000
0s - loss: 0.0394 - acc: 0.8440
Epoch 1000/1000
0s - loss: 0.0394 - acc: 0.8440
In [42]:
```

SciNet compute • calcul CANADA

About that optimization flag

The optimization flag was set to "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 in previous classes.

SciNet compute • calcul CANADA

Our network using Keras, notes

Some notes about the compilation of the model.

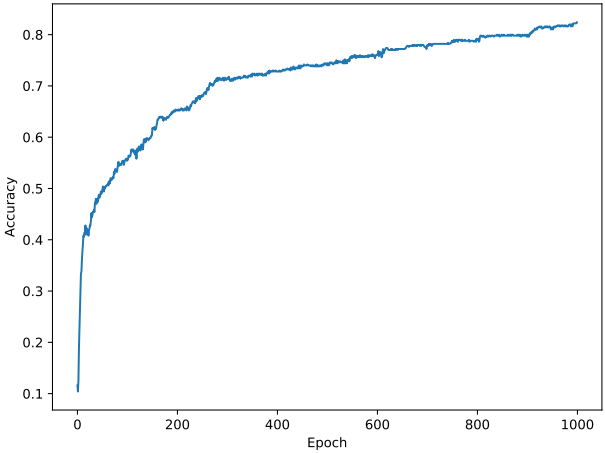
- We must specify the loss (cost) function with the "loss" argument.
- We must specify the optimization algorithm, using the "optimizer" flag.
- The optimizer can be generic ('sgd'), as in this example, or you can specify parameters using the optimizers in the keras.optimizers module.
- I sometimes specify the optimizer explicitly so that I can 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.

SciNet compute • calcul CANADA

Plotting the training

The fit.history dictionary contains useful information about the training. Sometimes plotting can give you some insight into the quality of the training, and whether or not it's finished.

```
In [42]:
In [42]: import matplotlib.pyplot as plt
In [43]: plt.plot(fit.history['acc'])
In [44]: plt.xlabel('Epoch')
In [45]: plt.ylabel('Accuracy')
In [46]:
```



Our network using Keras, continued even more

Now check against the test data.

Once again, we see the over-fitting rearing its head (84% versus 62%).

We can do better! Let's look at how we can address the overfitting.

```
In [46]:
In [47]: score = model.evaluate(x_test2, y_test2)
In [48]:
In [48]: score
Out[48]: [0.056402873396873471, 0.62]
In [49]:
```

SciNet compute • calcul CANADA

Regularization

Regularization is an *ad hoc* technique by which parameters in a model 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.

$$C = \frac{1}{2}\sum_i (a_{M_i} - v_i)^2 + \frac{\lambda}{2}\sum_j w_j^2$$

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 to the cost function changes the derivatives used to perform gradient descent.

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

where the derivatives of C_0 are calculated the usual way, using backpropagation. Our gradient descent parameter update now becomes

$$\begin{aligned} w_k &\rightarrow w_k - \eta \left[\frac{\partial C_0}{\partial w_k} + \lambda w_k \right] \\ &\rightarrow (1 - \eta\lambda)\, w_k - \eta \frac{\partial C_0}{\partial w_k} \end{aligned}$$

Regularization using Keras

Regularization, as you might expect, is built right into Keras.

- Regularization is done on a layer-by-layer basis.
- This allows you to only regularize some layers, and leave others out, if you so desire.
- Keras has three types of regularizers:
  - Kernel regularization: regularizes against the weights of the layer.
  - Bias regularization: regularizes against the biases of the layer.
  - Activity regularization: regularizes against the output of the layer.
- Because the regularization is specified layer-by-layer, the regularization is initialized as part of the layer declaration, rather than as part of the cost function.
- You can specify l1, l2 and custom regularizers.

Any combination of the regularizers can also be used.

SciNet compute • calcul CANADA

Regularization using Keras, continued

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

def get_model(numnodes, lam = 0.0):

  model = km.Sequential()
  model.add(kl.Dense(numnodes,
    input_dim = 784, name = 'hidden',
    activation = 'sigmoid',
    kernel_regularizer = kr.l2(lam)))

  model.add(kl.Dense(10,
    activation = 'sigmoid',
    kernel_regularizer = kr.l2(lam)))

  return model
```

```
In [49]: import model3 as m3
In [50]: model3 = m3.get_model(30, lam = 0.001)
In [51]: model3.compile(optimizer = 'sgd',
      metrics = ['accuracy'], loss = "mean_squared_error")
In [52]:
In [52]: fit = model3.fit(x_train2, y_train2,
      epochs = 1000, batch_size = 5, verbose = 2)
Epoch 1/1000
0s - loss: 0.1704 - acc: 0.1230
:
Epoch 1000/1000
0s - loss: 0.0433 - acc: 0.9860
In [53]: model3.evaluate(x_test2, y_test2)
100/100 [==============================] - 0s 11us/step
Out[53]: [0.06426530569791794, 0.78000000000000003]
In [54]:
```

Regularization using Keras, continued more

Why the improvement in the fitting of the test data?

- The regularization keeps the network from depending on any one particular weight, or set of weights (or biases), too much.
- This results in the network not focusing too much on any given feature, resulting in better generalization to the test data.
- As you might imagine, the regularization parameter is yet another hyperparameter in the development of our model.
- We've discussed some approaches for optimizing hyperparameters in previous classes.

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

SciNet compute • calcul CANADA

Dropout

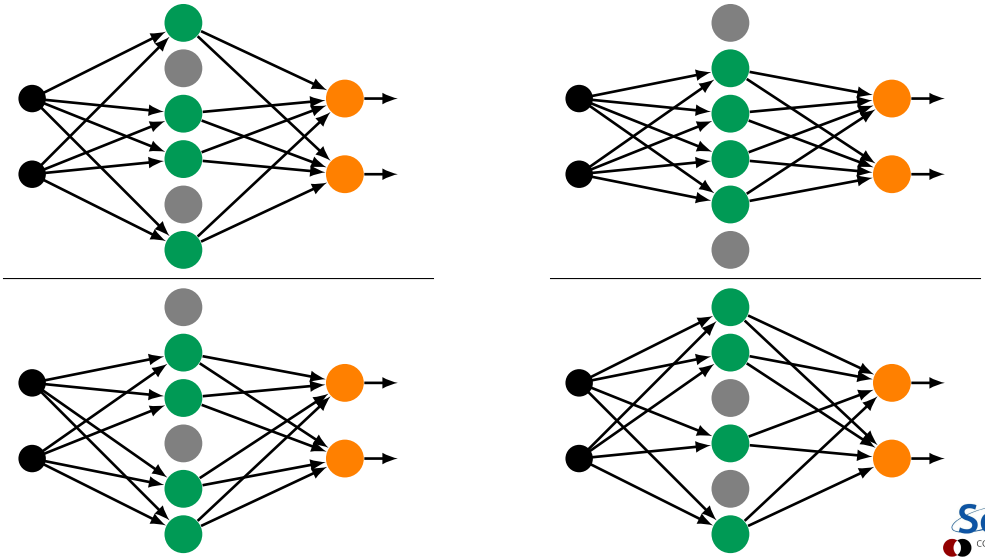
Dropout is a uniquely-neural-network technique to prevent over-fitting.

- The principle is simple: randomly "drop out" neurons from the network during each batch of the stochastic gradient descent.
- 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. A general rule of thumb is 30 - 50%.
- In the final model (after training):
  - the neurons in the dropout layer are no longer dropped out, and
  - the output from the neurons in the dropout layer is scaled by $(1 - p)$, where p is the probability of being dropped out.

This form of over-fitting control is quite common to encounter.

SciNet compute • calcul CANADA

Dropout, visualized



Dropout using Keras

```
# model4.py
import keras.models as km
import keras.layers as kl

def get_model(numnodes, d_rate = 0.4):

  model = km.Sequential()

  model.add(kl.Dense(numnodes,
    input_dim = 784, name = 'hidden',
    activation = 'sigmoid'))

  model.add(kl.Dropout(d_rate,
    name = 'dropout'))

  model.add(kl.Dense(10, name = 'output',
    activation = 'sigmoid'))
  return model
```

```
In [54]: import model4 as m4
In [55]: model4 = m4.get_model(30, d_rate = 0.2)
In [55]: model4.compile(loss = "mean_squared_error",
    ...:  optimizer = 'sgd', metrics = ['accuracy'])
In [56]: fit = model4.fit(x_train2, y_train2,
    ...:  epochs = 1000, batch_size = 5, verbose = 2)
Epoch 1/1000
0s - loss: 0.1727 - acc: 0.1600
⋮
Epoch 1000/1000
0s - loss: 0.0389 - acc: 0.8140
In [57]:
In [57]: model4.evaluate(x_test2, y_test2)
100/100 [==============================] - 0s
11us/step
Out[57]: [0.04947481036186218, 0.76000000000000001]
In [58]:
```

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, tomorrow.

SciNet compute • calcul CANADA

Other activation functions: relu

Two commonly-used functions:

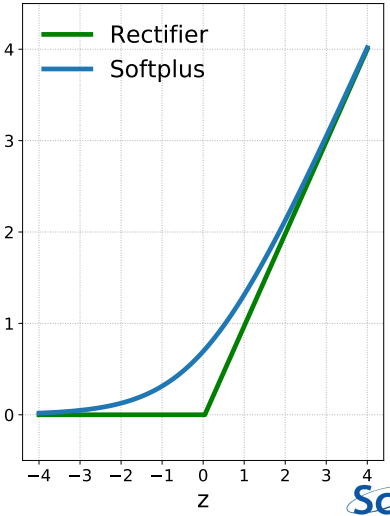
- Rectifier (also called Rectifier Linear Units, or RELUs):

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

- Softplus:

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

- Good: doesn't suffer from the vanishing-gradient problem.
- Bad: unbounded, could blow up.
- Other variants: leaky RELU, and SELU (scaled exponential).

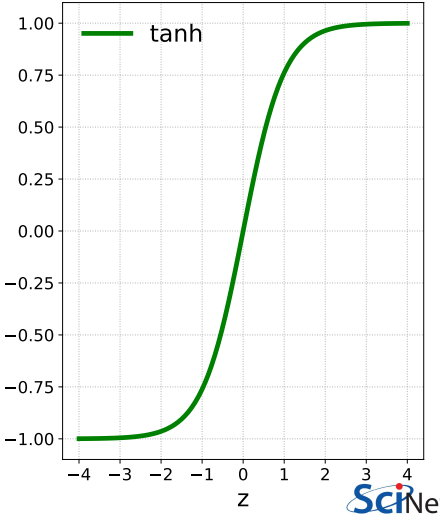


Other activation functions: tanh

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:

$$s(\mathbf{z}_j) = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}},$$

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

This is the activation function that is always used on the output layer when doing classification.

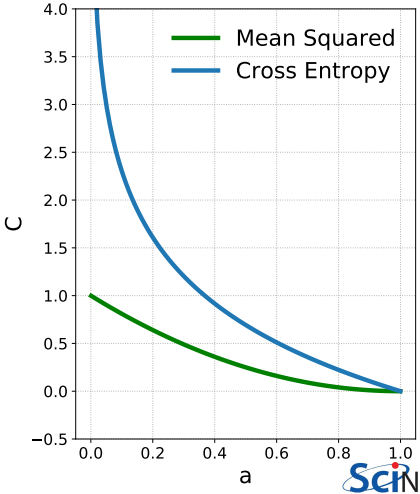
SciNet compute • calcul CANADA

Other cost functions: cross entropy

The most-commonly used cost function for categorical output data 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 always used with the softmax activation function as output.
- $v = 1$ in the example on the right.



Our Keras network, revisited

What's our new strategy for our MNIST neural network?

- Use all of the data.
- Change our hidden layer activation function to tanh.
- Change our output layer activation function to softmax.
- Use the cross-entropy cost function.
- Use the Adam minimization algorithm.
- We won't add regularization or dropout, as the data set is larger than the number of parameters in the model.

Using regular gradient descent would also probably work. Using the rectifier linear unit activation function on the hidden layer is also an option.

SciNet compute • calcul CANADA

Our Keras network, revisited

```
# model5.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, name = 'hidden',
    activation = 'tanh'))

  model.add(kl.Dense(10, name = 'output',
    activation = 'softmax'))

  return model
```

```
In [58]:
In [58]: x_train.shape
Out[58]: (60000, 28, 28)
In [59]: x_test.shape
Out[59]: (10000, 28, 28)
In [60]:
In [60]: x_train = x_train.reshape(60000, 784)
In [61]: x_test = x_test.reshape(10000, 784)
In [62]:
In [62]: y_train = ku.to_categorical(y_train, 10)
In [63]: y_test = ku.to_categorical(y_test, 10)
In [64]:
```

Our Keras network revisited, continued

```
In [64]: import model5 as m5
In [65]: model5 = m5.get_model(30)
In [66]:
In [66]: model5.compile(loss = "categorical_crossentropy", optimizer = "adam",
...:            metrics = ['accuracy'])
In [67]:
In [67]: fit = model5.fit(x_train, y_train, epochs = 100, batch_size = 100, verbose = 2)
Epoch 1/100
2s - loss:  0.0688 - acc:  0.4576
Epoch 2/100
2s - loss:  0.3661 - acc:  0.7246
⋮
Epoch 100/100
2s - loss:  0.0103 - acc:  0.9338
In [68]:
```

Our Keras network revisited, continued more

Now check against the test data.

93%! Better!

```
In [68]:
In [68]: score = model5.evaluate(x_test, y_test)
In [69]:
In [69]: score
Out[69]: [0.010993927612225524, 0.92949999999999999]
In [70]:
```

The next steps, an aside

There are alot of things we can tweak to make the network do better on the testing data (number of layers, neurons, activation functions, etc.). 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 may discuss this in a later class.

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

SciNet compute • calcul CANADA

Other topics

This class 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 sometimes matters.
- activation functions: there are several activation functions which are used in specific areas of neural networks. Learn which ones are used in your field.
- more cost functions: there are other cost functions which are used in specific applications.
- training failures: the disappearing gradient problem, the exploding gradient problem.

But at this point we have covered enough of the very basics to get you started.

SciNet compute • calcul CANADA

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

SciNet compute • calcul CANADA

Linky goodness, continued

Keras:

- `https://keras.io`

Other frameworks:

- `http://deeplearning.net/software/theano`
- `https://www.tensorflow.org`
- `http://caffe.berkeleyvision.org`
- `http://torch.ch`

SciNet compute • calcul CANADA