

Quantitative Applications for Data Analysis: neural networks

Erik Spence

SciNet HPC Consortium

2 April 2024

Today's slides

Today's slides can be found here. Go to the "Quantitative Applications for Data Analysis" page, under Lectures, "Neural networks".

<https://scinet.courses/1346>

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

Neural networks are finding their way into everything.

Neural networks, motivation

Consider the problem of hand-written digit recognition:

9 2 8 1 2 3

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

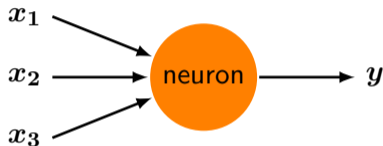
Rather than focus on the details of what individual numbers look like, we will instead ignore those details altogether. We will use a completely different approach:

- 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.

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

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

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

Neurons, continued

What function should we use for f ? One which is usually used, at least when initially teaching about neural networks, is the "sigmoid function" (also called the "logistic function").

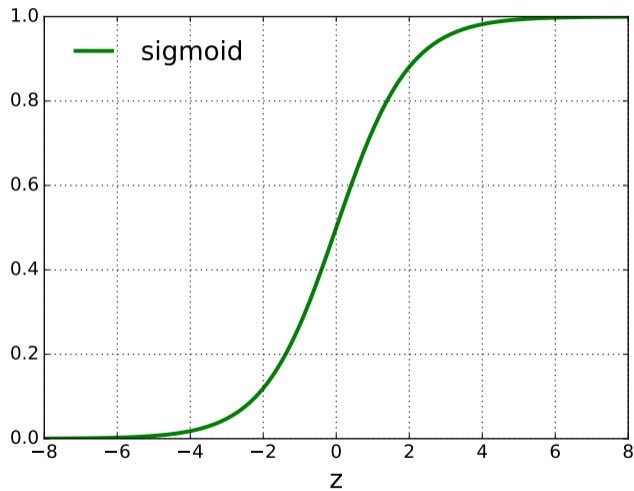
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

And so our neuron function becomes

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

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

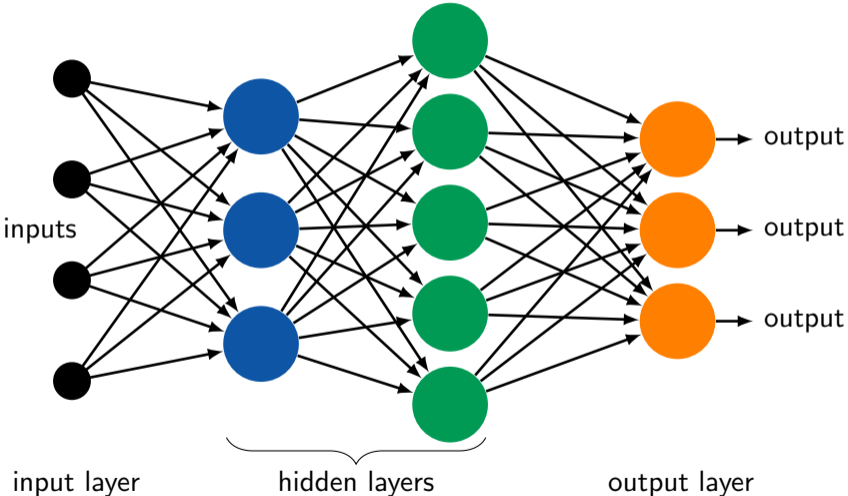
Why the sigmoid function?



Because it ranges from 0 to 1 smoothly.

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 \mathbf{b} .
- Each neuron outputs a single value. This output is passed to all of the neurons in the subsequent layer. This type of layer is known as a "fully-connected", or "dense", layer.
- The number of free parameters in the neurons in any given layer depends upon the number of neurons in the previous layer.
- The output from the output layer is aggregated into the desired form to calculate the cost function.

Seriously?

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

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

However, there are subtleties in training such a network.

Training our neural network

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

$$C = \frac{1}{2} \sum_i (g(\mathbf{x}_i) - \mathbf{y}_i)^2$$

where g is our neural network, and \mathbf{y}_i are the correct answers, based on the data, associated with each \mathbf{x}_i . Here we are using the "quadratic" cost function.

We then use an optimization algorithm to search for the values of \mathbf{w} and \mathbf{b} which generate the minimum of C , given the data \mathbf{x} and \mathbf{y} . We will use the Gradient Descent algorithm to find this minimum.

Gradient descent

Suppose the function we want to minimize has only one parameter.

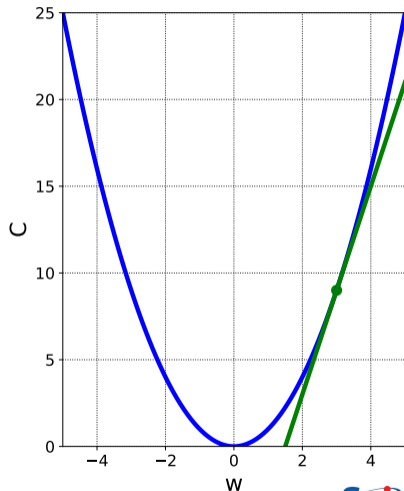
$$C = w^2$$

Suppose we've guessed that the minimum of C is w_i , and we wish to improve the guess. Gradient descent says to move according to the formula:

$$w_{i+1} = w_i - \eta \frac{\partial C}{\partial w_i}$$

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.



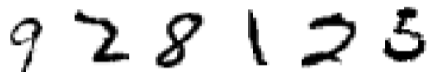
Training a neural network

How do we apply Gradient Descent to a neural network?

- Suppose that we decide to try to use gradient descent to train the network from five slides ago (slide 9).
- Each of the neurons has its own set of free parameters, \mathbf{w} and \mathbf{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 algorithm is called Backpropagation, but we won't cover it today.

Handwritten digits

A row of six handwritten digits: 9, 2, 8, 1, 2, 3. The digits are black on a white background and appear to be from the MNIST dataset.

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 create a neural network to classify these digits.

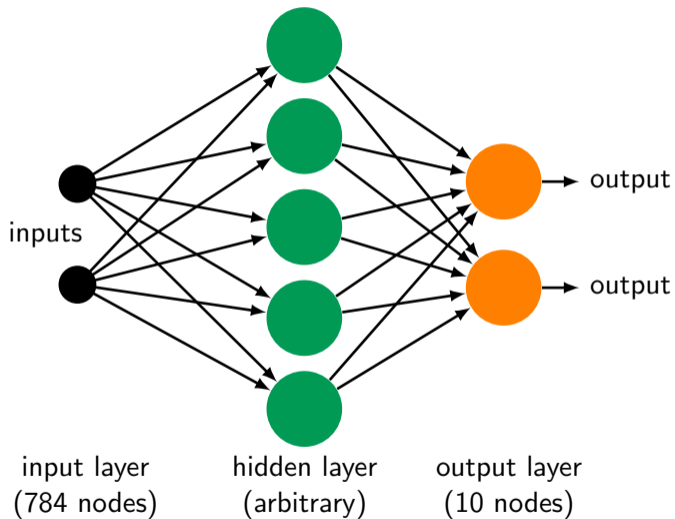
Our network

How would we design a network to analyse this data?

- Each image is $28 \times 28 = 784$ pixels. Let the input layer consist of 784 input nodes. Each node 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.

Fortunately, packages exist which make coding such a network quite easy.

Our neural network



Neural network frameworks

Now that we have a plan for our network, how are we going to code it? The standard way is to use a neural network 'framework'. Why would you do that?

- Coding your own networks from scratch can be a bit of work.
- 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.
- Standard NN frameworks include TensorFlow, Torch, MXNet, Caffe and many many others.

We will use Keras on a TensorFlow backend.

Keras

We will use Keras on top of Tensorflow.

- Keras is a NN framework, but it's only the top-most level.
- More accurately, it's an API standard for creating neural networks.
- Designed for fast development of networks.
- The original version ran on top of a 'back end', which by default is now TensorFlow, as Keras is being absorbed into TensorFlow.
- Historically it ran on top of many other backends also: Theano, 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.
- Because the recommended way is to use Keras through Tensorflow, that is the way we will be using it.

Getting the data

Let us implement our neural network using Keras. First let us get the MNIST data.

- To install keras you may need to install the devtools package.
- The data can be automatically downloaded by Keras.
- The data comes pre-split into training and testing data sets.
- We are going to use the data as 1-D for this example, so we need to reshape.

```
In [1]:  
-----  
In [1]: from tensorflow.keras.datasets import mnist  
-----  
In [2]:  
-----  
In [2]: (x_train, y_train), (x_test, y_test) =  
-----  
         mnist.load_data()  
-----  
In [3]:  
-----  
In [3]: x_train.shape  
Out[3]: (60000, 28, 28)  
-----  
In [4]:  
-----  
In [4]: x_test.shape  
Out[4]: (10000, 28, 28)  
-----  
In [5]:
```

Prepping the data

The data needs to be 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 [5]:  
-----  
In [5]: import tensorflow.keras.utils as ku  
-----  
In [6]: import tensorflow.keras.models as km  
-----  
In [7]: import tensorflow.keras.layers as kl  
-----  
In [8]:  
-----  
In [8]: x_train = x_train.reshape(60000, 784)  
-----  
In [9]: x_test = x_test.reshape(10000, 784)  
-----  
In [10]:  
-----  
In [10]: y_train = ku.to_categorical(y_train, 10)  
-----  
In [11]: y_test = ku.to_categorical(y_test, 10)  
-----  
In [12]:
```

Our network using Keras

We implement our neural network using Keras, with 30 neurons in the hidden layer.

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

```
In [12]: model = km.Sequential()
-----
In [13]: model.add(kl.Dense(30, input_dim = 784,
                             activation = 'sigmoid'))
-----
In [14]: model.add(kl.Dense(10,
                             activation = 'sigmoid'))
-----
In [15]: model.summary()
-----
Layer (type) Output Shape Param #
=====
dense_1 (Dense) (None, 30) 23550
-----
dense_2 (Dense) (None, 10) 310
=====
Total params: 23,860
Trainable params: 23,860
Non-trainable params: 0
```

Our network using Keras, continued more

Now that the network is constructed, it must be compiled.

- The loss function must be specified.
- The optimizer indicates what minimization algorithm to use. Here we use Stochastic Gradient Descent (SGD), which is a variation on regular Gradient Descent:
 - ▶ chop up the data into chunks of size 'batch_size',
 - ▶ perform regular gradient descent on each chunk,
 - ▶ repeat until all chunks have been used. This is an 'epoch'.
- The 'metrics' flag indicates what to print out during the training of the network.
- The 'fit' command is used to execute the training.
- The number of epochs, and the batch size, are parameters which apply to the optimization algorithm. We won't go over what they mean today.

Our network using Keras, continued

```
In [16]:
```

```
In [16]: model.compile(loss = 'mean_squared_error', optimizer = 'sgd', metrics = ['accuracy'])
```

```
In [17]:
```

```
In [17]: fit = model.fit(x_train, y_train, epochs = 200, batch_size = 128)
```

```
Epoch 1/200
```

```
60000/60000 [=====] - 0s - loss: 0.1368 - acc: 0.1933
```

```
Epoch 2/200
```

```
60000/60000 [=====] - 0s - loss: 0.0923 - acc: 0.3126
```

```
⋮
```

```
Epoch 199/200
```

```
60000/60000 [=====] - 0s - loss: 0.0225 - acc: 0.8947
```

```
Epoch 200/200
```

```
60000/60000 [=====] - 0s - loss: 0.0225 - acc: 0.8947
```

```
In [18]:
```


Our network using Keras, continued even more

Now check against the test data.
88.5%!

This isn't great. We can do
better.

```
In [18]:  
-----  
In [18]: score = model.evaluate(x_test, y_test)  
7616/10000 [=====>.....] - ETA: 0s  
-----  
In [19]:  
-----  
In [19]: score  
[0.024616853955388068, 0.8851999999999999]  
-----  
In [20]:
```

The next steps

We can do better. What might we do? There are a few simple approaches that might be explored.

- 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 over-fitting.

The most important technique, however:

- Completely overhaul the network strategy.

This field is huge; we've barely scratched the surface.

Other activation functions: relu

Two alternative activation functions:

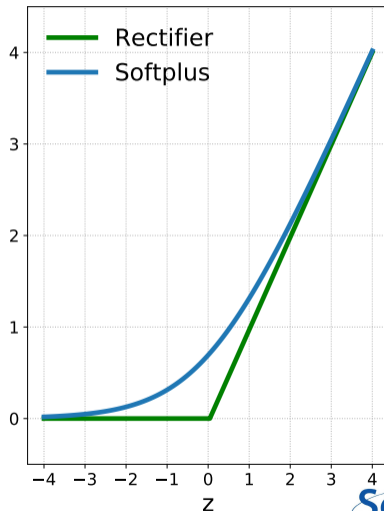
- Rectifier (also called Rectifier Linear Units, or RELUs):

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

- Softplus:

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

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

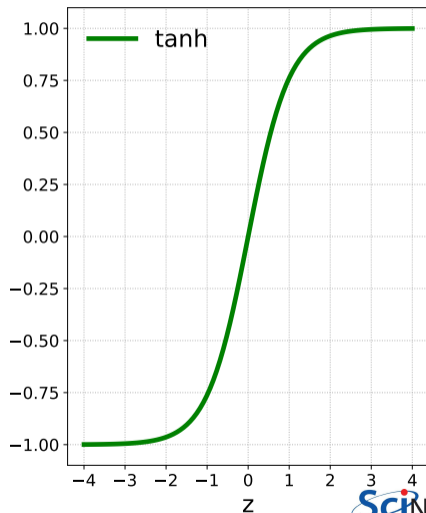


Other activation functions: tanh

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:

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

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

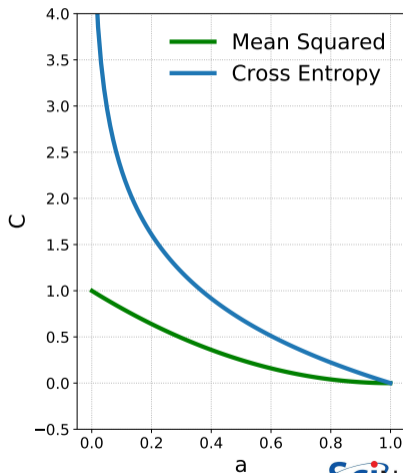
This activation function is always used on the output layer for single-class categorization problems.

Other cost functions: cross entropy

The most-commonly used cost function for categorization problems is cross entropy:

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

- 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.
- $y = 1$ in the example on the right.



Our Keras network revisited

```
In [20]: model = km.Sequential()
In [21]: model.add(kl.Dense(30, input_dim = 784, activation = 'tanh'))
In [22]: model.add(kl.Dense(10, activation = 'softmax'))
In [23]:
In [23]: model.compile(loss = "categorical_crossentropy", optimizer = "sgd",
...:                 metrics = ['accuracy'])
In [24]:
In [24]: fit = model.fit(x_train, y_train, epochs = 200, batch_size = 128, verbose = 2)
Epoch 1/200
2s - loss: 0.6853 - acc: 0.8261 - val_loss: 0.3783 - val_acc: 0.9021
Epoch 2/200
2s - loss: 0.3661 - acc: 0.9002 - val_loss: 0.3049 - val_acc: 0.9163
:
Epoch 200/200
2s - loss: 0.0543 - acc: 0.9860 - val_loss: 0.1342 - val_acc: 0.9633
In [25]:
```

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. To push this even further, we need to change our network's architecture.

```
In [25]:
```

```
score = model.evaluate(x_test, y_test)
```

```
In [26]:
```

```
score
```

```
[0.12195164765194058, 0.96540000000000004]
```

```
In [27]:
```


Deep Learning

You've probably heard the term. What is Deep Learning?

- Quite simply: a neural network with many hidden layers.
- 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.

Linky goodness

Neural network classes:

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

I am offering a neural network programming class starting in April. The class is taught in Python.