Part 3 - Even More Fun

"If you don't play, you don't learn."

In this part of the guide we'll explore further ideas just because they're fun. They aren't necessary to understanding the basic of neural networks so don't feel you have to understand everything here.

Because this is a fun extra section, the pace will be slightly quicker, but we will still try to explain the ideas in plain English.

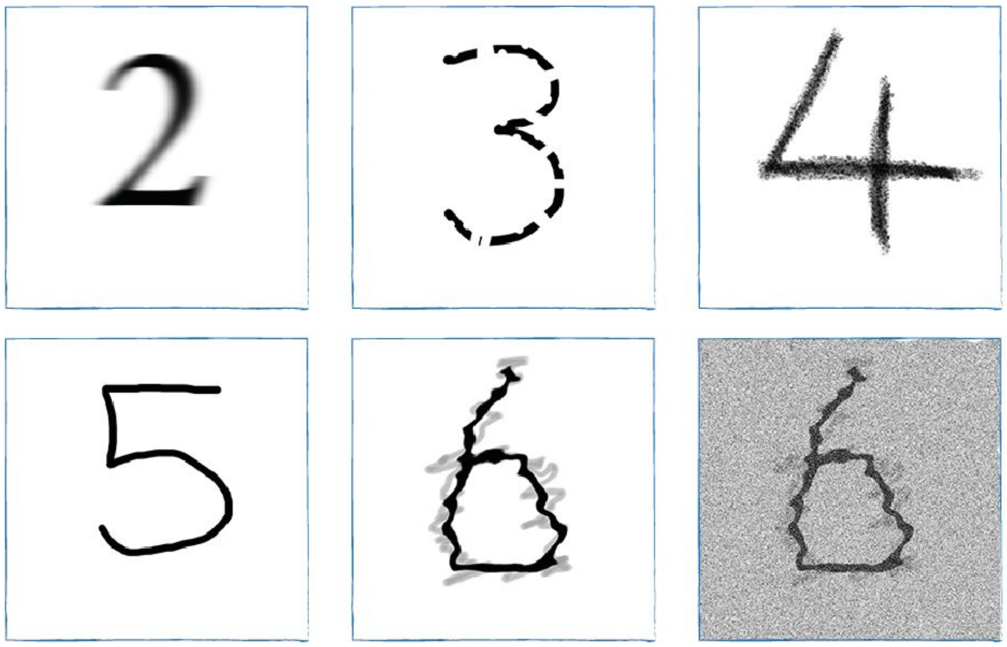
Your Own Handwriting

Throughout this guide we've been using images of handwritten numbers from the MNIST dataset. Why not use your own handwriting?

In this experiment, we'll create our test dataset using our own handwriting. We'll also try using different styles of writing, and noisy or shaky images to see how well our neural network copes.

You can create images using any image editing or painting software you like. You don't have to use the expensive Photoshop, the GIMP is a free open source alternative available for Windows, Mac and Linux. You can even use a pen on paper and photograph your writing with a smartphone or camera, or even use a proper scanner. The only requirement is that the image is square (the width is the same as the length) and you save it as PNG format. You'll often find the saving format option under File > Save As, or File > Export in your favourite image editor.

Here are some images I made.



The number 5 is simply my own handwriting. The 4 is done using a chalk rather than a marker. The number 3 is my own handwriting but deliberately with bits chopped out. The 2 is a very traditional newspaper or book typeface but blurred a bit. The 6 is a deliberately wobbly shaky image, almost like a reflection in water. The last image is the same as the previous one but with noise added to see if we can make the neural network's job even harder!

This is fun but there is a serious point here. Scientists have been amazed at the human brain's ability to continue to function amazingly well after suffering damage. The suggestion is that neural networks distribute what they've learned across several link weights, which means if they suffer some damage, they can perform fairly well. This also means that they can perform fairly well if the input image is damaged or incomplete. That's a powerful thing to have. That's what we want to test with the chopped up 3 in the image set above.

We'll need to create smaller versions of these PNG images rescaled to 28 by 28 pixels, to match what we've used from the MNIST data. You can use your image editor to do this.

Python libraries again help us out with reading and decoding the data from common image file formats, including the PNG format. Have a look at the following simple code:

```
import scipy.misc
img_array = scipy.misc.imread(image_file_name, flatten=True)

img_data  = 255.0 - img_array.reshape(784)
img_data = (img_data / 255.0 * 0.99) + 0.01
```

The scipy.misc.imread() function is the one that helps us get data out of image files such as PNG or JPG files. We have to import the scipy.misc library to use it. The "flatten=True" parameter turns the image into simple array of floating point numbers, and if the image were coloured, the colour values would be flattened into grey scale, which is what we need.

The next line reshapes the array from a 28x28 square into a long list of values, which is what we need to feed to our neural network. We've done that many times before. What is new is the subtraction of the array's values from 255.0. The reason for this is that it is conventional for 0 to mean black and 255 to mean white, but the MNIST data set has this the opposite way around, so we have to reverse the values to match what the MNIST data does.

The last line does the familiar rescaling of the data values so they range from 0.01 to 1.0.

Sample code to demonstrate the reading of PNG files is always online at gihub:

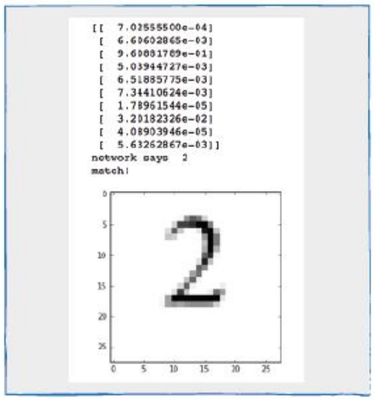
- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part3_load_own_images.ipynb

We need to create a version of our basic neural network program that trains on the MNIST training data set but instead of testing with the MNIST test set, it tests against data created from our own images.

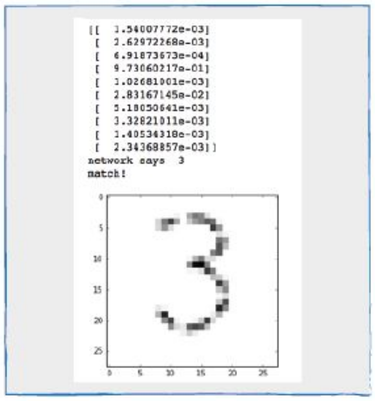
The new program is online at github:

- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part3_neural_network_mnist_and_own_data.ipynb

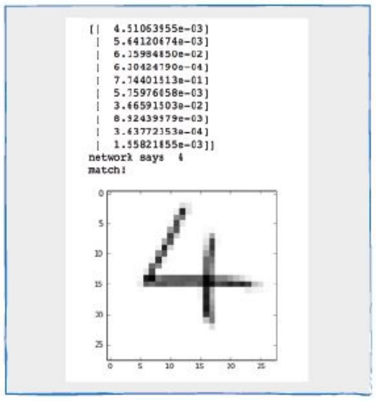
Does it work? It does! The following summarises the results of querying with our own images.



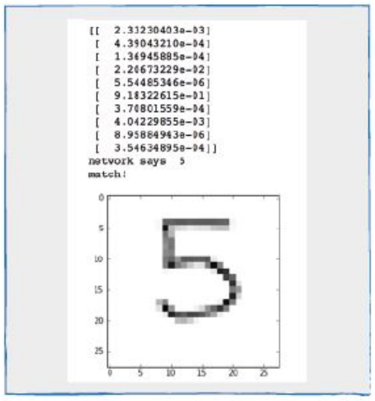
```
[[  7.02555500e-04]
 [  6.60602865e-03]
 [  9.60081709e-01]
 [  5.03944727e-03]
 [  6.51885775e-03]
 [  7.34410624e-03]
 [  1.78961544e-05]
 [  3.20182326e-02]
 [  4.08903946e-05]
 [  5.63262867e-03]]
network says  2
match!
```



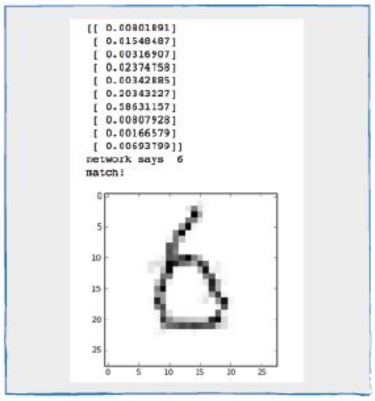
```
[[  1.54007772e-03]
 [  2.62972268e-03]
 [  6.91873673e-04]
 [  9.73060217e-01]
 [  1.02681001e-03]
 [  2.83167145e-02]
 [  5.18050641e-03]
 [  3.32821011e-03]
 [  1.40534318e-03]
 [  2.34368857e-03]]
network says  3
match!
```



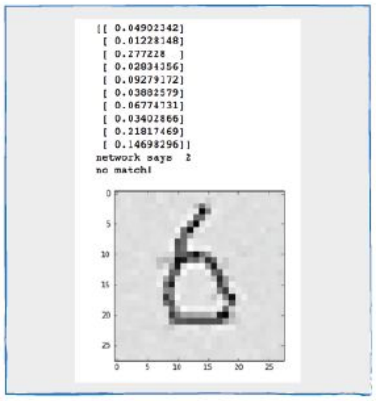
```
[[  4.51063955e-03]
 [  5.64120674e-03]
 [  6.15984850e-02]
 [  6.30424790e-04]
 [  7.74401513e-01]
 [  5.75976658e-03]
 [  3.66591503e-02]
 [  8.92439979e-03]
 [  3.63772153e-04]
 [  1.55821855e-03]]
network says  4
match!
```



```
[[  2.33230403e-03]
 [  4.39043210e-04]
 [  1.36945885e-04]
 [  2.20673229e-02]
 [  5.54485346e-06]
 [  9.18322615e-01]
 [  3.70801559e-04]
 [  4.04229855e-03]
 [  8.95884943e-06]
 [  3.54634895e-04]]
network says  5
match!
```



```
[[ 0.00801891]
 [ 0.01548487]
 [ 0.00316907]
 [ 0.02374758]
 [ 0.00342885]
 [ 0.20343227]
 [ 0.58631157]
 [ 0.00807928]
 [ 0.00166579]
 [ 0.00693799]]
network says  6
match!
```



```
[[ 0.04902342]
 [ 0.01228148]
 [ 0.277228  ]
 [ 0.02834356]
 [ 0.09279172]
 [ 0.03882579]
 [ 0.06774731]
 [ 0.03402866]
 [ 0.21817469]
 [ 0.14698296]]
network says  2
no match!
```

You can see that the neural network recognised all of the images we created, including the deliberately damaged “3”. Only the “6” with added noise failed.

Try it with your own images, especially handwritten, to prove to yourself that the neural network really does work.

And see how far you can go with damaged or deformed images. You’ll be impressed with how resilient the neural network is.

Inside the Mind of a Neural Network

Neural networks are useful for solving the kinds of problems that we don't really know how to solve with simple crisp rules. Imagine writing a set of rules to apply to images of handwritten numbers in order to decide what the number was. You can imagine that wouldn't be easy, and our attempts probably not very successful either.

Mysterious Black Box

Once a neural network is trained, and performs well enough on test data, you essentially have a mysterious **black box**. You don't really know **how** it works out the answer - it just does.

This isn't always a problem if you're just interested in answers, and don't really care how they're arrived at. But it is a disadvantage of these kinds of machine learning methods - the learning doesn't often translate into understanding or wisdom about the problem the black box has learned to solve.

Let's see if we can take a peek inside our simple neural network to see if we can understand what it has learned, to visualise the knowledge it has gathered through training.

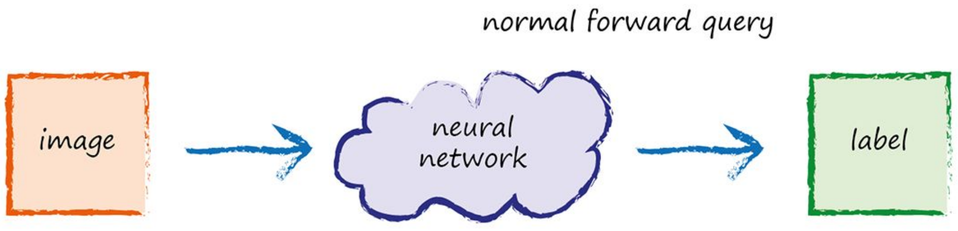
We could look at the weights, which is after all what the neural network learns. But that's not likely to be that informative. Especially as the way neural networks work is to distribute their learning across different link weights. This gives them an advantage in that they are resilient to damage, just like biological brains are. It's unlikely that removing one node, or even quite a few nodes, will completely damage the ability of a neural network to work well.

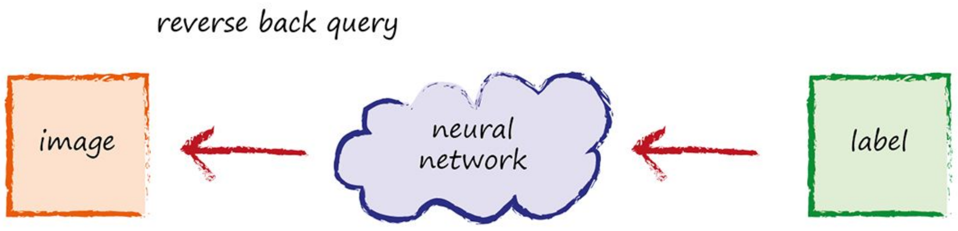
Here's a crazy idea.

Backwards Query

Normally, we feed a trained neural network a question, and out pops an answer. In our example, that question is an image of a human handwritten number. The answer is a label representing a number from 0 to 9.

What if we turned this around and did it backwards? What if we fed a label into the output nodes, and fed the signal backwards through the already-trained network, until out popped an image from the input nodes? The following diagram shows the normal forward query, and this crazy reverse **back query** idea.





We already know how to propagate signals through a network, moderating them with link weights, and recombining them at nodes before applying an activation function. All this works for signals flowing backwards too, except that the inverse activation is used. If **y = f(x)** was the forward activation then the inverse is **x = g(y)**. This isn't that hard to work out for the logistic function using simple algebra:

$$y = 1 / (1 + e^{-x})$$

$$1 + e^{-x} = 1/y$$

$$e^{-x} = (1/y) -1 = (1 - y) / y$$

$$-x = \ln [(1-y) / y]$$

$$x = \ln [y / (1-y)]$$

This is called the **logit** function, and the Python scipy.special library provides this function as **scipy.special.logit()**, just like it provides **scipy.special.expit()** for the logistic sigmoid function.

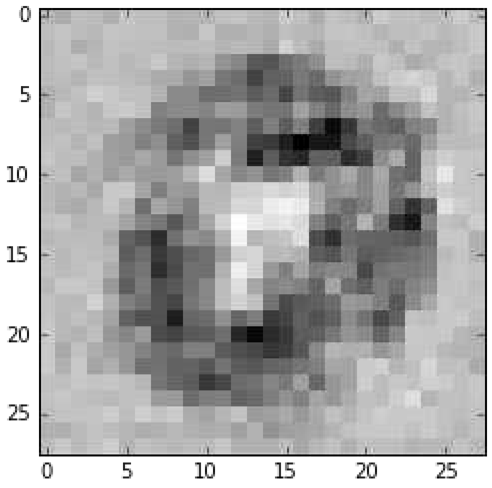
Before applying the logit() inverse activation function, we need to make sure the signals are valid. What does this mean? Well, you remember the logistic sigmoid function takes any value and outputs a value somewhere between 0 and 1, but not including 0 and 1 themselves. The inverse function must take values from the same range, somewhere between 0 and 1, excluding 0 and 1, and pop out a value that could be any positive or negative number. To achieve this, we simply take all the values at a layer about to have the logit() applied, and rescale them to the valid range. I've chosen the range 0.01 to 0.99.

The code is always available online at github at the following link:

- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part3_neural_network_mnist_backquery.ipynb

The Label "0"
Let's see what happens if we do a back query with the label "0". That is, we present values to the output nodes which are all 0.01 except for the first node representing the label "0" where we use the value 0.99. In other words, the array **[0.99, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01]**.

The following shows the image that pops out of the input nodes.



That is interesting!

That image is a privileged insight into the mind of a neural network. What does it mean? How do we interpret it?

The main thing we notice is that there is a round shape in the image. That makes sense, because we're asking the neural network what the ideal question is for an answer to be "0".

We also notice dark, light and some medium grey areas:

- The **dark** areas are the parts of the question image which, if marked by a pen, make up supporting evidence that the answer should be a "0". These make sense as they seem to form the outline of a zero shape.

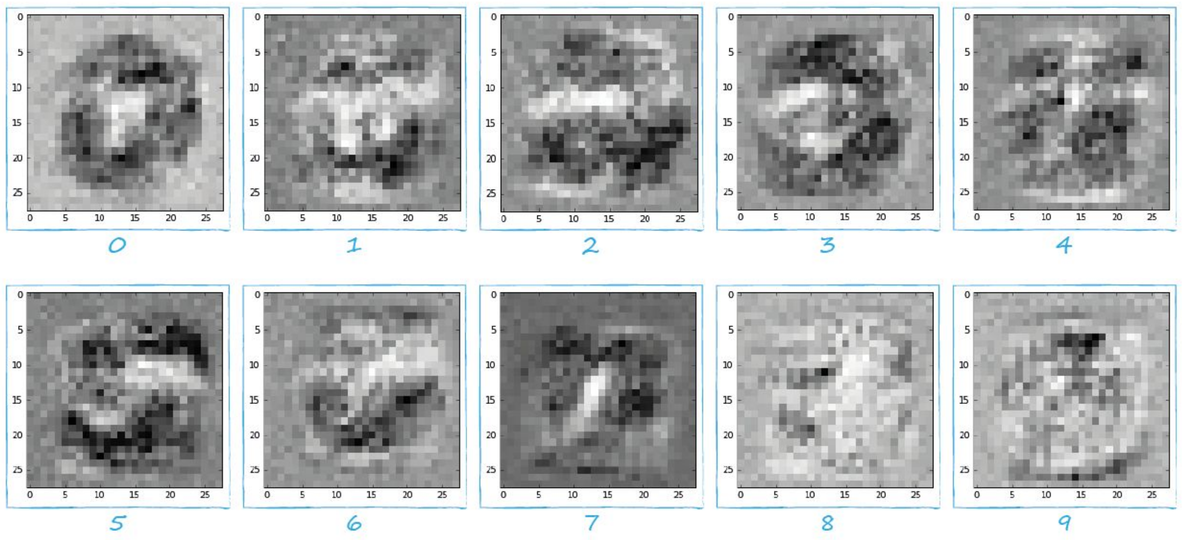
- The **light** areas are the bits of the question image which should remain clear of any pen marks to support the case that the answer is a “0”. Again these make sense as they form the middle part of a zero shape.

- The neural network is broadly indifferent to the **grey** areas.

So we have actually understood, in a rough sense, what the neural network has learned about classifying images with the label “0”.

That’s a rare insight as more complex networks with more layers, or more complex problems, may not have such readily interpretable results. You’re encouraged to experiment and gave a go yourself!

More Brain Scans
The following shows the results of back querying the rest of the digits.



Wow! Again some really interesting images. They're like ultrasound scans into the brain of the neural network.

Some notes about these images:

- The "7" is really clear. You can see the dark bits which, if marked in the query image, strongly suggest a label "7". You can also see the additional "white" area which must be clear of any marking. Together these two characteristics indicate a "7".

- The same applies to the "3" - there are dark areas which, if marked, indicate a "3", and there are white areas which must be clear.

- The “2” and "5" are similarly clear too.

- The "4" is interesting in that there is a shape which appears to have 4 quadrants, and excluded areas too.

- The "8" is largely made up of a “snowman” shaped white areas suggesting that an eight is characterised by markings kept out of these “head and body” regions.

- The “1” is rather puzzling. It seems to focus more on areas which much be kept clear than on areas which must be marked. That’s ok, it’s what the network happens to have learned from the examples.

- The “9” is not very clear at all. It does have a definite dark area and some finer shapes for the white areas. This is what the network has learned, and overall, when combined with what it has learned for the rest of the digits, allows the network to perform really well at 97.5% accuracy. We might look at this image and conclude that more training examples might help the network learn a clearer template for “9”.

So there you have it - a rare insight into the workings of the mind of a neural network.

Creating New Training Data: Rotations

If you think about the MNIST training data you realise that it is quite a rich set of examples of how people write numbers. There are all sorts of styles of handwriting in there, good and bad too.

The neural network has to learn as many of these variations as possible. It does help that there are many forms of the number “4” in there. Some are squished, some are wide, some are rotated, some have an open top and some are closed.

Wouldn’t it be useful if we could create yet more such variations as examples? How would we do that? We can’t easy collect thousands more examples of human handwriting. We could but it would be very laborious.

A cool idea is to take the existing examples, and create new ones from those by rotating them clockwise and anticlockwise, by 10 degrees for example. For each training example we could have two additional examples. We could create many more examples with different rotation angles, but for now let’s just try +10 and -10 degrees to see if the idea works.

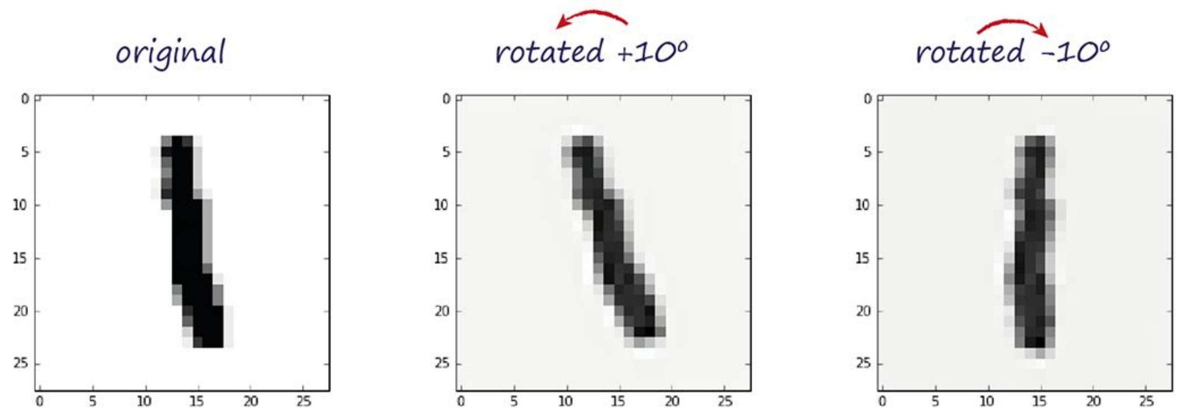
Python’s many extensions and libraries come to the rescue again. The ndimage.interpolation.rotate() can rotate an array by a given angle, which is exactly what we need. Remember that our inputs are a one-dimensional long list of length 784, because we’ve designed our neural networks to take a long list of input signals. We’ll need to reshape that long list into a 28*28 array so we can rotate it, and then unroll the result back into a 784 long list of input signals before we feed it to our neural network.

The following code shows how we use the ndimage.interpolation.rotate() function, assuming we have the **scaled_input** array from before:

```
# create rotated variations
# rotated anticlockwise by 10 degrees
inputs_plus10_img =
scipy.ndimage.interpolation.rotate(scaled_input.reshape(28,28), 10,
cval=0.01, reshape=False)
# rotated clockwise by 10 degrees
inputs_minus10_img =
scipy.ndimage.interpolation.rotate(scaled_input.reshape(28,28), -10,
cval=0.01, reshape=False)
```

You can see that the original scaled_input array is reshaped to a 28 by 28 array, then scaled. That reshape=False parameter prevents the library from being overly helpful and squishing the image so that it all fits after the rotation without any bits being clipped off. The cval is the value used to fill in array elements because they didn’t exist in the original image but have now come into view. We don't want the default value of 0.0 but instead 0.01 because we've shifted the range to avoid zeros being input to our neural network.

Record 6 (the seventh record) of the smaller MNIST training set is a handwritten number "1". You can see the original and two additional variations produced by the code in the diagram below.



You can see the benefits clearly. The version of the original image rotated +10 degrees provides an example where someone might have a style of writing that slopes their 1's backwards. Even more interesting is the version of the original rotated -10 degrees, which is clockwise. You can see that this version is actually straighter than the original, and in some sense a more representative image to learn from.

Let's create a new Python notebook with the original neural network code, but now with additional training examples created by rotating the originals 10 degrees in both directions. This code is available online at github at the following link:

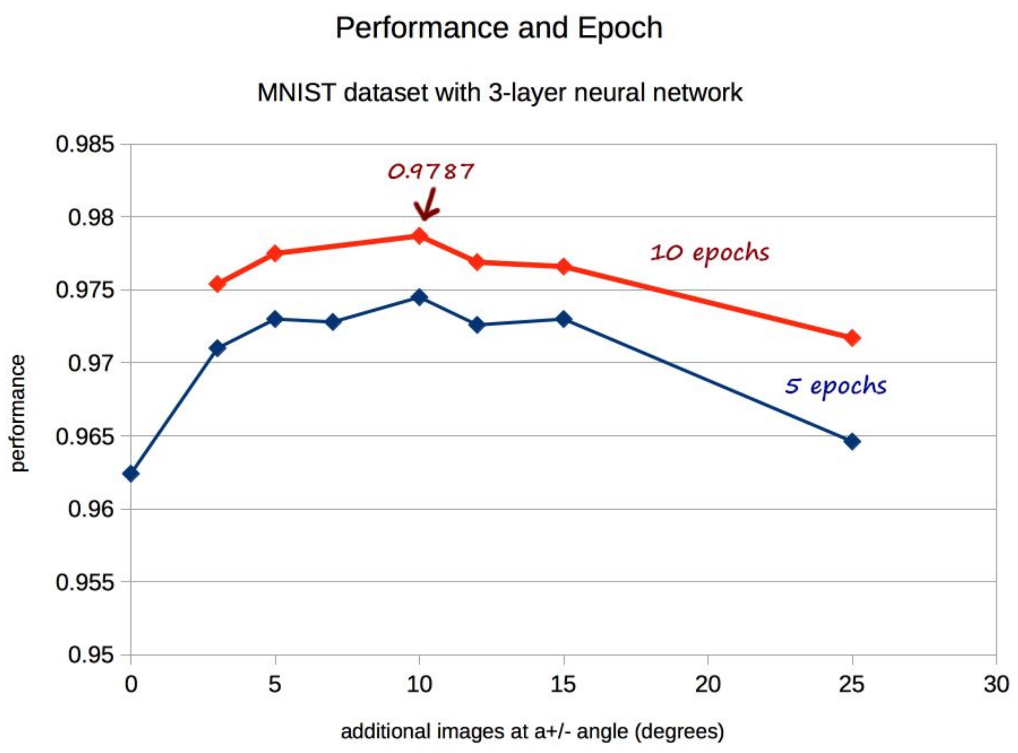
- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part2_neural_network_mnist_data_with_rotations.ipynb

An initial run with learning rate set to 0.1, and only one training epoch, the resultant performance was **0.9669**. That's a solid improvement over 0.954 without the additional rotated training images. This performance is already amongst the better ones listed on Yann LeCunn's website.

Let's run a series of experiments, varying the number of epochs to see if we can push this already good performance up even more. Let's also reduce the **learning rate to 0.01** because we are now creating much more training data, so can afford to take smaller more cautious learning steps, as we've extended the learning time overall.

Remember that we don't expect to get 100% as there is very likely an inherent limit due to our specific neural network architecture or the completeness of our training data, so we may never hope to get above 98% or some other figure. By "specific neural network architecture" we mean the choice of nodes in each layer, the choice of hidden layers, the choice of activation function, and so on.

Here's the graph showing the performance as we vary the angle of additional rotated training images. The performance without the additional rotated training examples is also shown for easy comparison.



You can see that with 5 epochs the best result is **0.9745** or 97.5% accuracy. That is a jump up again on our previous record.

It is worth noticing that for large angles the performance degrades. That makes sense, as large angles means we create images which don't actually represent the numbers at all. Imagine a "3" on its side at 90 degrees. That's not a three anymore. So by adding training examples with overly rotated images, we're reducing the quality of the training by adding false examples. Ten degrees seems to be the optimal angle for maximising the value of additional data.

The performance for 10 epochs peaks at a record breaking **0.9787** or almost **98%**! That is really a stunning result, amongst the best for this kind of simple network. Remember we haven't done any fancy tricks to the network or data that some people will do, we've kept it simple, and still achieved a result to be very proud of.

Well done!