Creating New Training Data by Rotations

If you think about the MNIST training data, you realize 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.

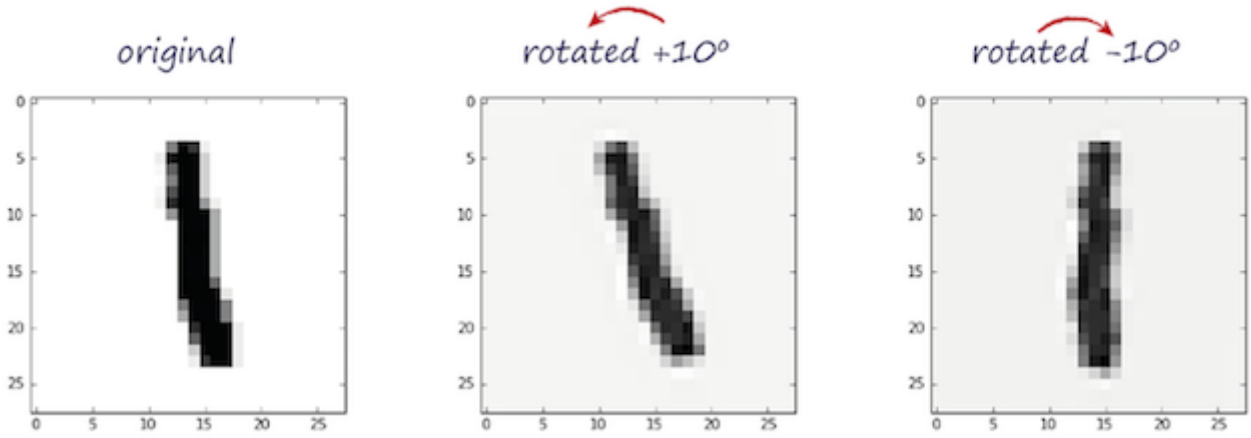
Wouldn't it be useful if we could create yet more such variations as examples? How would we do that? We can't easily collect thousands of more examples of human handwriting. We could, but it would be very laborious. A cool idea is to take the existing examples and create new ones from those by rotating 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 (
inputs_plus10_img = scipy.ndima
# rotated clockwise by 10 degre
inputs_minus10_img = scipy.ndi
```

You can see that the original scaled_input array is reshaped to a 28 by 28

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 $1's$ backward. 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:

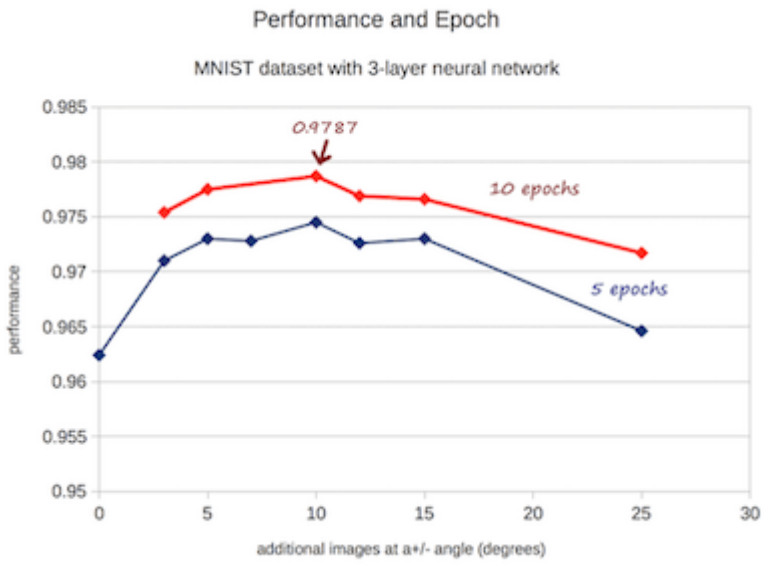
- https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part2_neural_network_mnist_data_with_rotations.ipynb

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 mean 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 maximizing the value of additional data. The performance for ten 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!