Doing Multiple Runs

How doing multiple runs can bring improvement in the accuracy of result?

The next improvement we can do is to repeat the training several times against the data set. Some people call each run through an *epoch*. So a training session with 10 epochs means running through the entire training data set 10 times. Why would we do that? Especially if the time our computers take goes up to 10or 20 or even 30 minutes? The reason it is worth doing is that we're helping those weights do that gradient descent by providing more chances to creep down those slopes. Let's try it with 2 epochs. The code changes slightly because we now add an extra loop around the training code. The following shows this outer loop color coded to help see what's happening.

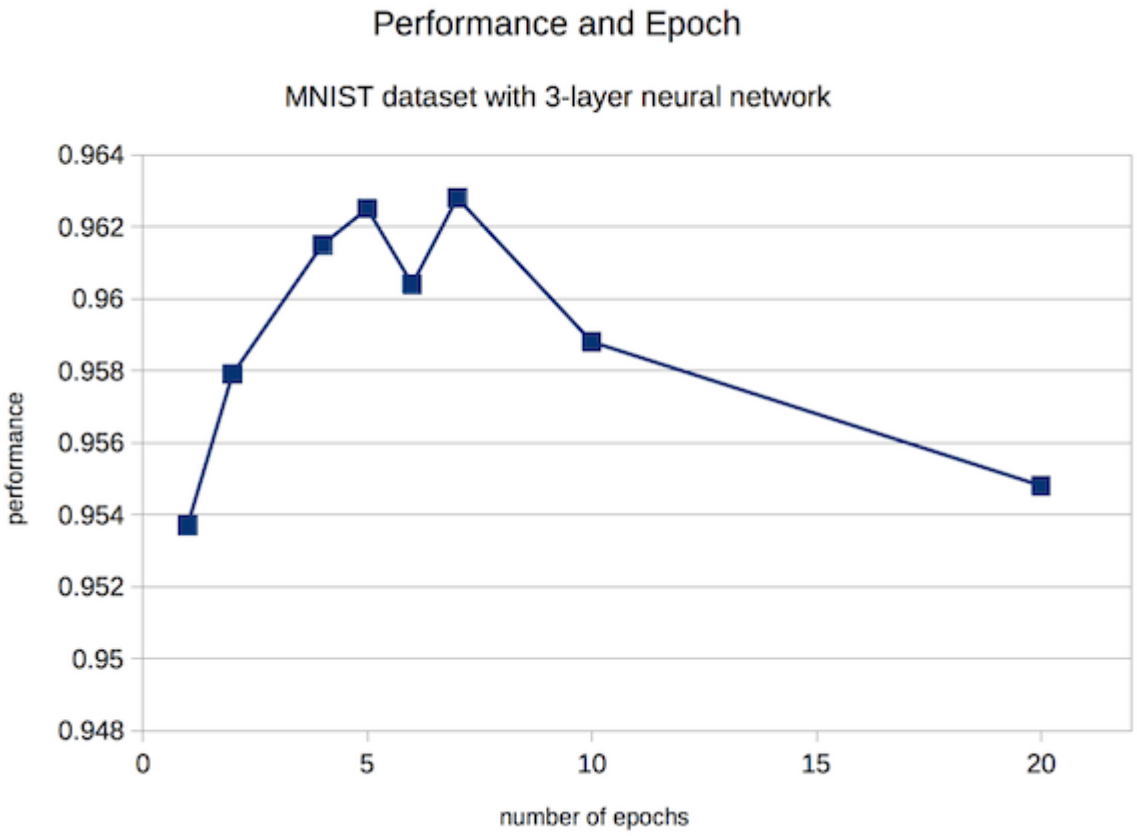
```
# train the neural network

# epochs is the number of times
epochs = 2

for e in range(epochs):
    # go through all records in
    for record in training_data
        # split the record by t
        all_values = record.spl
        # scale and shift the i
        inputs = (numpy.asfarra
        # create the target out
        targets = numpy.zeros(o
        # all_values[0] is the
        targets[int(all_values
        n.train(inputs, targets
        pass
    pass
```

The resulting performance with 2 epochs will be slightly improved as compared to performance generated over just 1 epoch (see the figure below). Just like we did with tweaking the learning rate, let's experiment with a few different epochs and plot a graph to visualize the effect this has. Intuition

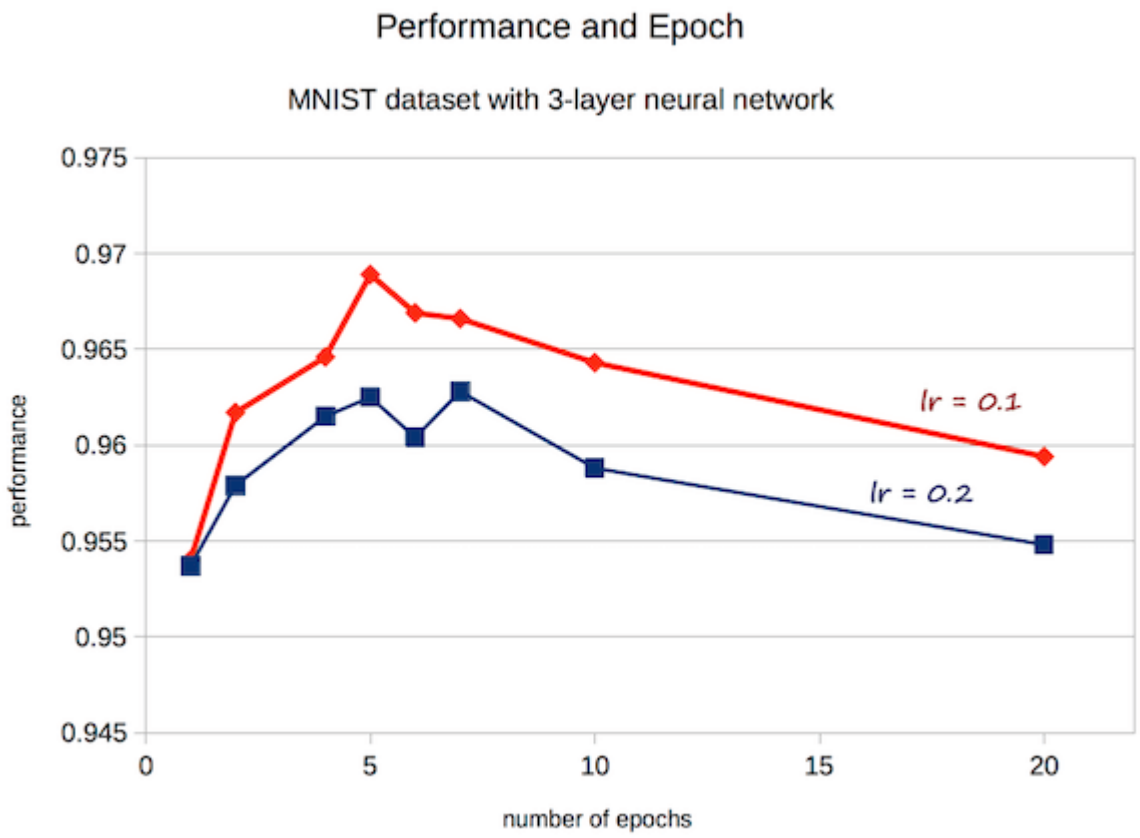
suggests the more training you do, the better the performance. Some of you will realize that too much training is actually bad because the network overfits to the training data, and then performs badly against new data that it hasn't seen before. This *overfitting* is something to beware of across many different kinds of machine learning, not just neural networks. Here's what happens:



You can see the results aren't quite so predictable. You can see there is a sweet spot around 5 or 7 epochs. After that performance degrades, and this may be the effect of overfitting. The dip at 6 epochs is probably a bad run with the network getting stuck in a bad minimum during gradient descent. Actually, I would have expected much more variation in the results because we've not done many experiments for each data point to reduce the effect of expected variations from what is essentially a random process. That's why I've left that odd point for six epochs in, to remind us that neural network learning is a random process at heart and can sometimes not work so well, and sometimes work really badly.

Another idea is that the learning rate is too high for larger numbers of epochs. Let's try this experiment again and tune down the learning rate from 0.2 down to 0.1 and see what happens. The peak performance is now up to 0.9628, or 96.28%, with 7 epochs (see the figure below). The following graph

shows the new performance with learning rate at 0.1 overlaid onto the previous one.



You can see that calming down the learning rate did indeed produce better performance with more epochs. That peak of 0.9689 represents an approximate error rate of 3%, which is comparable to the networks benchmarks on Yann LeCun's website.

Intuitively it makes sense that if you plan to explore the gradient descent for much longer (more epochs), you can afford to take shorter steps (learning rate), and overall you'll find a better path down. It does seem that 5 epochs are probably the sweet spot for this neural network against this MNIST learning task. Again keep in mind that we did this in a fairly unscientific way. To do it properly you would have to do this experiment many times for each combination of learning rates and epochs to minimize the effect of randomness that is inherent in *Gradient Descent*.