Improve your neural networks – Part 1 [TIPS AND TRICKS]

March 28, 2017 Andy

A bag of tricks

In the **last post**, I presented a comprehensive tutorial of how to build and understand neural networks. At the end of that tutorial, we developed a network to classify digits in the MNIST dataset. However, the accuracy was well below the state-of-the-art results on the dataset. This post will show some techniques on how to improve the accuracy of your neural networks, again using the **scikit learn** MNIST dataset.

When we are thinking about "improving" the performance of a neural network, we are generally referring to two things:

1. Improve the accuracy
2. Speed up the training process (while still maintaining the accuracy)

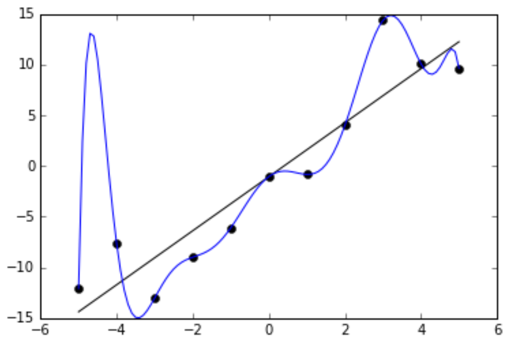
(1) and (2) can play off against each other. This is because, in order to improve the accuracy, we often need to train our network with more data and more iterations. This slows down the training however, and makes it more expensive. The "tips and tricks" in this post will address both of these issues. All code will be in Python.

1 Regularisation and over-fitting in neural networks

1.1 Over-fitting

As was presented in the **neural networks tutorial**, we always split our available data into at least a training and a test set. We do this because we want the neural network to *generalise* well. This means that we want our network to perform well on data that it hasn't "seen" before during training. If we just throw all the data we have at the network during training, we will have no idea if it has *over-fitted* on the training data. If it has, then it will perform badly on new data that it hasn't been trained on. We need another data set, the test set, to check and make sure our network is *generalising* well.

What happens when a machine learning model over-fits during training? A simple way of thinking about it is that it becomes *over-complicated* given the data it has to train on. This is often best illustrated using a linear regression example, see the image below from Wikipedia:



By Ghiles (Own work) [CC BY-SA 4.0], via Wikimedia Commons

The black dots are the training data. As can be observed, the data-points pretty obviously follow a linear trend with increasing *x*, but there is a bit of noise in the relationship i.e. not all the data points sit on the black linear trend line. The blue line is a high order polynomial model, which has *perfectly* fit the training data. However, intuitively we would say that the black linear line is the better model. Why is that? It is because the blue model is *unjustifiably* complicated given the data – it is so complicated that it has "chased after" the random noise of the data. We can see that small changes in the horizontal axis can lead to huge changes in the vertical axis. This is not a good model and will therefore perform badly on any test set that it is tried on.

Over-fitting is something we also have to be wary of in neural networks. A good way of avoiding this is to use something called *regularisation.*

1.2 Regularisation

Regularisation involves making sure that the weights in our neural network do not grow too large during the training process. During training, our neural networks will converge on local minimum values of the cost function. There will be many of these local minima, and many of them will have roughly the same cost function – in other words, there are many ways to skin the cat. Some of these local minimum values will have large weights connecting the nodes and layers, others will have smaller values. We want to force our neural network to pick weights which are smaller rather than larger.

This makes our network less complex – but why is that? Consider the previous section, where we discussed that an over-fitted model has large changes in predictions compared to small changes in input. In other words, if we have a little bit of noise in our data, an over-fitted model will react strongly to that noise. The analogous situation in neural networks is when we have large weights – such a network is more likely to react strongly to noise. This is because large weights will amplify small variations in the input which could be solely due to noise. Therefore, we want to adjust the cost function to try to make the training drive the magnitude of the weights down, while still producing good predictions.

The old cost function was (see the **neural networks tutorial** for an explanation of the notation used):

$$J(w,b) = \frac{1}{m}\sum_{z=0}^{m}\frac{1}{2}\parallel y^z - h^{(n_l)}(x^z)\parallel^2$$

J(w,b)=1m∑z=0m12∥yz−h(nl)(xz)∥2

In this cost function, we are trying to minimize the mean squared error (MSE) of the prediction compared to the training data. Now we want to vary the cost function to:

$$J(w,b) = \frac{1}{m}\sum_{z=0}^{m}\frac{1}{2}\parallel y^z - h^{(n_l)}(x^z)\parallel^2 + \frac{\lambda}{2}\sum_{all}\left(W_{ij}^{(l)}\right)^2$$

J(w,b)=1m∑z=0m12∥yz−h(nl)(xz)∥2+λ2∑all(Wij(l))2

Notice the addition of the last term, which is a summation of all the weight values in each layer, multiplied by the λλ constant divided by 2 (the division by 2 is a little trick to clean things up when we take the derivative). This λλvalue is usually quite small. This shows that any increase in the weights must be balanced by an associated decrease in the mean squared error term in the cost function. In other words, large weights will be penalised in this new cost function if they don't do much to improve the MSE.

To incorporate this new component into the training of our neural network, we need to take the partial derivative. Working this through gives a new gradient descent step equation. The old equation:

$$W^{(l)} = W^{(l)} - \alpha \left[\frac{1}{m} \Delta W^{(l)}\right]$$

W(l)=W(l)–α[1mΔW(l)]

is now replaced by the new:

$$W^{(l)} = W^{(l)} - \alpha \left[\frac{1}{m} \Delta W^{(l)} + \lambda W^{(l)}\right]$$

W(l)=W(l)–α[1mΔW(l)+λW(l)]

1.3 Regularisation in the training code

The gradient descent weight updating line in the code of the **neural network tutorial** can simply be updated to the following, to incorporate regularisation into the Python code:

```
W[l] += -alpha * (1.0/m * tri_W[l] + lamb * W[l])
```

Where "lamb" is the regularisation parameter, which must be chosen. If you recall from the tutorial, without regularisation the prediction accuracy on the scikit learn sample MNIST data set was only 86%. This was with a learning rate (αα) of 0.25 and 3,000 training iterations. Using the same parameters, and a regularisation parameter (λλ) equal to 0.001, **we now get a prediction accuracy of 95%!** That is a 9% increase in prediction accuracy by altering a single line of code and adding a new parameter.

Therefore, it is safe to say that in our previous example without regularisation we were over-fitting the data, despite the mean squared error of both versions being practically the same after 3,000 iterations.

2. Selecting the right parameters

In our training code for neural networks, we have a number of free parameters. These parameters are the learning rate αα, the number and size of the hidden layers and now the regularisation parameter λλ. We also have to make a choice about what activation function to use. All of these selections will affect the performance of the neural network, and therefore must be selected carefully. How do we do this? Usually by some sort of brute force search method, where we vary the parameters and try to land on those parameters which give us the best predictive performance.

Do we still use the test set to determine the predictive accuracy by which we tune our parameters? Is it really a test set in that case? Aren't we then using all our data to make the network better, rather than leaving some aside to ensure we aren't over-fitting? Yes, we are. We need to introduce a new set of the training data called the *validation* set.

To create a validation set, we can use the scikit learn function called **train_test_split**. Usually, we want to keep the majority of data for training, say 60%. The remaining data we can split into a test set and a validation set. The code below shows how to do this:

```
from sklearn.model_selection import train_test_split
X_train, X_holdover, y_train, y_holdover = train_test_split(X, y, test_size=0.4)
X_valid, X_test, y_valid, y_test = train_test_split(X_holdover, y_holdover, test_size=0.5)
```

Now we have training, validation and test data sets, and we're ready to perform parameter selections.

2.1 Brute-force search example

Often model parameter selection is performed using the brute-force search method. This method involves cycling through likely values for the parameters in different combinations and assessing some measure of accuracy / fitness for each combination on the validation set. We then select the best set of parameter values and see how they go on the test set. The brute-force search method is easy to implement but can take a long time to run, given the combinatorial explosion of scenarios to test when there are many parameters.

In the example below, we will be using the brute-force search method to find the best parameters for a three-layer neural network to classify the scikit learn MNIST dataset. Note: this data set isn't the "real" MNIST dataset that is used often as a benchmark (it's a cut down version), but it is good enough for our purposes. This example will be using some of the same functions as in the **neural network tutorial**. The parameters that we are going to test are:

1. The number of hidden layers
2. The learning rate
3. The regularisation constant

Let's first setup some lists for the parameter cycling:

```
hidden_size = [10, 25, 50, 60]
alpha = [0.05, 0.1, 0.25, 0.5]
lamb = [0.0001, 0.0005, 0.001, 0.01]
```

It is now a simple matter of cycling through each parameter combination, training the neural network, and assessing the accuracy. The code below shows how this can be done, assessing the accuracy of the trained neural network after 3,000 iterations. Careful, this will take a while to run:

```
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
digits = load_digits()
X_scale = StandardScaler()
X = X_scale.fit_transform(digits.data)
y = digits.target

# setup the parameter selection function
def select_parameters(hidden_size, alpha, lamb, X, y):
    X_train, X_holdover, y_train, y_holdover = train_test_split(X, y, test_size=0.4)
    X_valid, X_test, y_valid, y_test = train_test_split(X_holdover, y_holdover,
                                       test_size=0.5)
    # convert the targets (scalars) to vectors
    yv_train = convert_y_to_vect(y_train)
    yv_valid = convert_y_to_vect(y_valid)
    results = np.zeros((len(hidden_size)*len(alpha)*len(lamb), 4))
    cnt = 0
    for hs in hidden_size:
        for al in alpha:
            for l in lamb:
                nn_structure = [64, hs, 10]
                W, b, avg_cost = train_nn(nn_structure, X_train, yv_train,
                                          iter_num=3000, alpha=al, lamb=l)
                y_pred = predict_y(W, b, X_valid, 3)
                accuracy = accuracy_score(y_valid, y_pred) * 100
                print("Accuracy is {}% for {}, {}, {}".format(accuracy, hs, al, l))
                # store the data
                results[cnt, 0] = accuracy
                results[cnt, 1] = hs
                results[cnt, 2] = al
                results[cnt, 3] = l
                cnt += 1
    # get the index of the best accuracy
    best_idx = np.argmax(results[:, 0])
    return results, results[best_idx, :]
select_parameters(hidden_size, alpha, lamb, X, y)
```

Note that the above code uses functions developed in the **neural networks tutorial**. These functions can be found on this site's GitHub **repository**.   After running this code, we find that the best accuracy (98.6%) is achieved on the validation set with 50 hidden layers, a learning rate of 0.5 and a regularisation parameter of 0.001.

Using these parameters on the test set now gives us an accuracy of 96%. Therefore, using the brute-force search method and a validation set, along with regularisation, improved our original naïve results in the **neural networks tutorial** from 86% to 96%! A big improvement, clearly worth the extra time taken to improve our model.

In the next part of this series we'll look at ways of speeding up the training.