Practical Tips for using Backpropagation

Keith L. Downing

August 31, 2017

1 Introduction

In practice, backpropagation is as much an art as a science. The user typically needs to try many combinations of parameter settings before finding that which best suits the current task. Few concrete rules exist to cover a wide range of application domains. There are many factors to consider, and we touch on a few of them below. A host of additional tips can be found in the neural network literature, particularly in the Deep Learning book (Goodfellow, Bengio and Courville, 2016).

This document is far from a complete list of but merely a collection of issues that I have encountered in my own experiences with supervised learning using neural networks.

2 Hidden Layers and Nodes

Although there are some theoretical results, there is no hard and fast rule as to how many hidden layers and nodes in each layer are required for a given problem. There are a few considerations.

1. If all of your training and test cases are, in fact, linearly separable, then you can get by without a hidden layer. Unfortunately, most real-life data sets are not linearly separable.
2. If the data are not linearly separable, then, in theory, a **single** hidden layer is sufficient for learning the data. However, the exact number of nodes in that layer cannot be accurately calculated ahead of time. Also, the practially-best solution may involve multiple hidden layers.
3. The more layers and nodes in an ANN, the more connections, and thus the more weights that need to be learned. Adding more connections than necessary can easily increase training time well above that needed to solve the problem.
4. Excess connections increase the risk of overfitting the ANN weight vector to the training set. Thus, the ANN may perform perfectly on the training set but poorly on the test cases. This indicates that the ANN has *memorized* the training cases but cannot *generalize* from them to handle new cases. This is highly undesirable. To prevent this (in cases where a perceived demand for many connections exists) it helps to include a "validation set" and to halt training when the error rate for validation testing begins to rise.

3 Activation Functions

In working with shallow nets, it is often wise to choose between a sigmoid (a.k.a. the logistic function), a hyperbolic tangent (tanh) or a simple linear (i.e. identity) function. This choice often involves a simple analysis of the situation. If all nodes should be outputting positive values, then the sigmoid is preferable to tanh. If outputs should be bounded above and below, then the sigmoid (range (0,1)) or tanh (range (-1,1)) are preferable to a linear function, unless explicit upper and lower bounds will be imposed on the linear function.

Although the logistic function (a.k.a. sigmoid) has been a common activation function for many years – partly due to its biological relevance but mostly due to the fact that it is continuous and differentiable yet quite similar to a fixed-thresholded step function – it has serious shortcomings when used in deep networks. Essentially, the gradients (i.e. derivatives of error with respect to weights) deteriorate rather quickly when derived from long chains of sigmoids.

One of the key breakthroughs in neural network research was the discovery that the Rectified Linear Unit (RELU) preserves significant gradients across these long chains, thus giving more informed weight modifications throughout deep networks. Hence, the RELU and its relatives – the Exponential Linear Unit (ELU) and the Leaky RELU – are quite common fixtures in deep networks.

4 Ranges for Desired Values

When converting desired outputs into target activations for the output nodes of an ANN, it is wise to use a restricted range of values. This pertains to ANNs that use any kind of sigmoidal activation function.

For example, if the sigmoid's output range is [0, 1], then it pays to use a subrange such as [0.1, 0.9] for the desired output values. The problem lies in the nature of the sigmoid function, which has **almost** the same minimum or maximum value for large sections of the domain, i.e., the area far from the threshold point. In trying to attain this asymptote of the sigmoid, the backpropagation algorithm will often increase n's incoming weights indefinitely, and these weight increases can propagate backwards. Once weights become very large, backpropagation has a hard time reducing them quickly, or at all. The whole system can easily spiral out of control, with most weights approaching infinity and most sigmoids becoming *saturated*, i.e. having weighted sum inputs that push them to the extreme ends of their range.

By using target values well below the asymptotic limit of the sigmoid, one insures that targets can be achieved exactly and weights will remain more stable.

5 Ranges for activation levels

For any application, it is important to decide whether low values of a particular factor, such as an input parameter, should have either a) a low effect upon downstream nodes, or b) the **opposite** effect of a high value. In the former case, your nodes should output in the range [0,1], while the latter case probably calls for a [-1, 1] range.

6 Ranges for Initial Weights

Although the initial settings of weights may have little effect in many architectures, it can have significant effects in others. As Glorot and Bengio (2010) describe in "Understanding the difficulty of training deep feedforward neural networks," major performance increases can result by initializing all weights to random values within the range [-$\sqrt{M}$,$\sqrt{M}$], where M is the number of neurons in the upstream layer, which is normally the number of inputs to a neuron in the downstream layer.

7 Information Content of Nodes

One should exercise caution when using *tricky* encodings of information on the input and output ends of an ANN, even though these can yield more compact networks (i.e., those with fewer nodes and connections).

For example, if the color of an automobile is an input parameter to an ANN for assessing insurance risk, then a compact encoding might involve a single input neuron, n, whose activation value would indicate everything from dark colors (low activation levels) to bright colors (high activation levels).

Keep in mind that the goal of the ANN is to learn a proper, **fixed** set of weights emanating from n that will reduce total error and thus give good predictions of insurance risk based on color (and other input variables).

Whatever these weights may be, they represent the total effect of color upon all downstream nodes. And the activation level of n will further modulate that effect, but changes to that activation are severely constrained as to possible changes in effect. In essence, they are constrained to linear behavior. If the activation level goes up, then the absolute value of the effect will also increase.

So only if there does indeed exist a linear relationship between the color and the net effect (on insurance risk), can this encoding work effectively. However, it could easily be the case that both very bright colors (such as hot red) and dark colors (black) are typical colors of cars owned by drivers with a *street racer* mentality. In fact, there is a strong correlation in the United States between these hot colors and speeding tickets. I am only speculating about the black colored cars for the purpose of this example. Conversely, colors in the middle of the spectrum, such as pale blue, are more typical *family car* colors.

The problem is that if both black and red have effects that trigger a high insurance risk, then by tuning weights to realize that influence, one cannot avoid producing weights that will also give blue a similar effect. In this example, the weights would have to be very high to allow both a low color input (black) to achieve a similar influence to that of a high-color input (red).

In this case, it is wiser to use k input nodes, where each represents a single color. Then, backpropagation is free to tune the individual effects of each color (by modifying its outgoing weights) without being hampered by any artificial connections between the colors (as imposed by the more compact representation).

Conversely, if the input factor truly does have a linear effect upon the output factors, for example, if weight is an input condition and risk of heart-attack is an output class, then compressing the range of inputs into a single input node makes more sense.

The same can be said about output nodes and the information encoded in them. In this case, the input arcs to an output node represent the net effect of the ANN's information upon the outputs. Once again, a linear

relationship holds such that the sum of the weighted inputs has a non-decreasing effect upon the output. [1].

Once those weights are fixed by learning, they enforce a linear relationship between any given node, n_1 in the previous layer and a particular output node, n_2. Thus, the target concepts to which activation levels of n_2 correspond should have a linear relationship to the information represented by n_1. If not, then the ANN will not be able to determine an optimal mapping between inputs and outputs.

The above advice is heuristic in nature, not absolute. A tricky encoding here or there may not destroy ANN performance, but it is important to be aware of the gains (compact ANNs) and losses (inability to properly differentiate certain cases) associated with witty encoding schemes.

8 Momentum

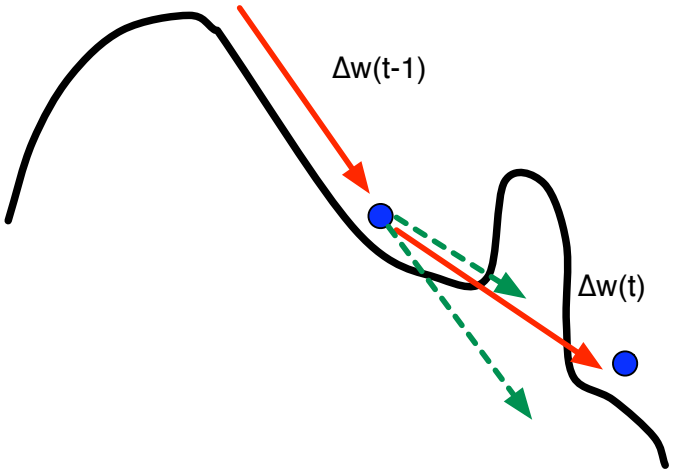

Figure 1: Illustration of the effect of momentum in breaking through local minima. Given the current location of the system on the error landscape (upper blue dot), the standard weight-updating algorithm would move it toward the lowest point of the local minimum, as shown by the upper dotted line. However, the previous move (line labeled $\Delta w(t-1)$) provides momentum (long dotted line). The combination of the two (dotted line) vectors gives the resultant weight change, which breaks through the hump that encloses the local minimum.

Backpropagation can easily get stuck at a local minimum, such as that pictured in Figure 8. There, all gradients point upward, toward higher error, so all progress stagnates.

To combat the problem, a momentum term is added to the weight-update function such that the current weight change is a combination of the standard change and that change made during the **previous** learning pass of backpropagation, known as the *momentum term*. Equation 1 expresses this update formula, where α is the momentum factor. In many implementations, α decreases with the training epoch such that momentum has a large effect in the early epochs but a minor role later.

$$\Delta w_{ij}(t) = -\eta \frac{\partial E_i}{\partial w_{ij}} + \alpha \Delta w_{ij}(t-1) \tag{1}$$

[1]Here, we ignore less conventional, non-monotonic, activation schemes such as radial-basis functions

As shown in Figure 8, this momentum can help the search process blast through the walls that surround a local minimum to continue its descent on the error landscape.

9 Optimizing Trial and Error

The lack of uncertainty as to the proper neural network configuration for a particular task often necessitates a trial-and-error process wherein a wide variety of nets must be built and tested. Tools that can speed up this process are extremely useful. Some of these tools, such as Tensorflow, Theano, Cafe and Keras, are off-the-shelf and appropriate for many needs, but in other cases, you may need to develop some of your own aids (often on top of the aforementioned systems). In general, it is well worth the effort to build and/or tailor a **general** system for neural network experimentation. The savings are enormous, even in the short term, since a single neural network project can easily entail the building and running of hundreds of network prototypes before you find the right one.