More on fitting neural networks

This note is a continuation of our high-level introduction to neural networks.

Local optima

The cost function for neural networks is not unimodal, and so is certainly not convex (a stronger property). We can see why by considering a neural network with two hidden units. Assume we've fitted the network to a (local) optimum of a cost function, so that any small change in parameters will make the network worse. Then we can find another parameter vector that will represent exactly the same function, showing that the optimum is only a local one.

To create the second parameter vector, we simply take all of the parameters associated with hidden unit one, and replace them with the corresponding parameters associated with hidden unit two. Then we take all of the parameters associated with hidden unit two and replace them with the parameters that were associated with hidden unit one. The network is really the same as before, with the hidden units labelled differently, so will have the same cost.

Models with "hidden" or "latent" representations of data, usually have many equivalent ways to represent the same model. When the goal of a machine learning system is to make predictions, it doesn't matter whether the parameters are well-specified. However, it's worth remembering that the values of individual parameters are often completely arbitrary, and can't be interpreted in isolation.

In practice local optima don't just correspond to permuting the hidden units. Some local optima will have better cost than others, and some will make predictions that generalize better than others. When I've fitted small neural networks, I've tried optimizing many times and used the network that cross-validates the best. However, researchers pushing up against available computational resources will find it difficult to optimize a network many times.

One advantage of large neural networks is that fitting more parameters tends to work better(!). The intuition I have is that there are many more ways to set the parameters to get low cost, so it's less hard to find one good setting.[1] Although it's difficult to make rigorous statements on this issue. Understanding the difficulties that are faced in really high-dimensional optimization is an open area of research. (For example, `https://arxiv.org/abs/1412.6544`.)

Regularization by early stopping

We have referred to complex models that generalize poorly as "over-fitted". One idea to avoid "over-fitting" is to fit less! That is, stop the optimization routine before it has found a local optimum of the cost function. This heuristic idea is often called "early stopping".

The most common way to implement early stopping is to periodically monitor performance on a validation set. If the validation score is the best that we have seen so far, we save a copy of the network's parameters. If the validation score fails to improve upon that cost over some number of future checks (say 20), we stop the optimization and return the weights we've saved.

David MacKay's textbook mentions early stopping (Section 39.4, p479). This book points out that stopping the optimizer prevents the weights from growing too large. Goodfellow et al.'s deep learning textbook (Chapter 7) makes a more detailed link to L2 regularization. MacKay argued that adding a regularization term to the cost function to achieve a similar effect seems more appealing: if we have a well-defined cost function, we're not tied to a particular optimizer, and it's probably easier to analyse what we're doing.

However, I've found it hard to argue with early stopping as a pragmatic, sensible procedure. The heuristic directly checks whether continuing to fit is improving predictions for held-out

1. The high-level idea is old, but a recent (2018) analysis described it as "The Lottery Ticket Hypothesis", `https://arxiv.org/abs/1803.03635`

data, which is what we care about. And we might save a lot of computer time by stopping early. Moreover, we can still use a regularized cost function along with early stopping.

Regularization corrupting the data or model

There are a whole family of methods for regularizing models that involve adding noise to the data or model during training. Like early-stopping, I found this idea unappealing, as it's hard to understand what objective we are fitting, and it makes the models we obtain depend on which optimizer we are using. However, these methods are often effective…

Adding Gaussian noise to the inputs of a linear model during gradient training has the same average effect as L2 regularization[2]. We can also add noise when training neural networks. The procedure will still have a regularization effect, but one that's harder to understand. We can also add noise to the weights or hidden units of a neural network. In some applications, adding noise has worked better than optimizing easy-to-define cost functions (like L2 regularizers).

Other regularization methods randomly replace some of the weights with zeros ("drop-out"[3]) or features with zeros (such as in "denoising auto-encoders"[4] or a 2006 feature-dropping regularizer.). These heuristics prevent the model from fitting delicate combinations of parameters, or fits that depend on careful combinations of features. If used aggressively, "masking noise" makes it hard to fit anything! Often *large* models are needed when using these heuristics.

Further Reading

Most textbooks are long out-of-date when it comes to recent practical wisdom on fitting neural networks and regularization strategies. However, `http://www.deeplearningbook.org/` is still fairly recent, and is a good starting point. The MLP notes are also more detailed on practical tips for deep nets.

I'll mention two of the recent tricks that I think are particularly worth knowing that make it easier to fit deep networks. But like most ideas, they don't always improve things, so experiments are required. And the research landscape and available tools are changing rapidly.

The first trick, *Batch Normalization* (or "batch norm"), is "old" enough to be covered in the deep learning textbook. The discussion in the previous note about initialization pointed out that we don't want to saturate hidden units. Batch normalization rescales the activations for a unit across a training batch to a target variance, making gradient-based training of neural nets easier in many tasks. In hindsight I'm amazed this trick is so recent: it's a simple idea that someone could have come up with in a previous decade, but didn't. Variants are still being actively explored, so I recommend chasing the recent literature if interested.

Another trick is the use of *residual layers*. There are different variants, especially when combined with batch norm in different places. However, the core idea is to take a standard non-linearity g and transform one hidden layer to another with $r(\mathbf{h}) = \mathbf{h} + g(W\mathbf{h})$. The weights W are used to fit the "residual" difference from a default transformation, that leaves the hidden unit values unchanged. Related, more complicated, layers were previously developed in the recurrent neural network literature, such as the "LSTM" cell, and are also worth knowing about. It is often easier to fit deep stacks of residual layers (or LSTMs, or GRUs), than standard layers.

2. Non-examinable: there's a sketch in these slides: `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec9.pdf`. More detail in Bishop's (1995) neural network textbook, section 9.3.
3. `https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf`
4. `http://icml2008.cs.helsinki.fi/papers/592.pdf`