Practical Advice for Building Deep Neural Networks

by Matt Z and Daniel S

In our machine learning lab, we' ve accumulated tens of thousands of training hours across numerous high-powered machines. The computers heave*, * the only ones to learn a lot in the process, though: we ourselves have made a lot of mistakes and fixed a lot of bugs.

Here we present some practical tips for training deep neural networks based on our experiences (tested mainly in TensorFlow). Some of the suggestions may seem obvious to you, but they matter *!* to one of us at some point. Other suggestions are not apply or might even be bad advice for your particular task; use discretion!

We acknowledge these are all well known methods. But, too, stand on the shoulders of giants here! Our objective with this article is simply to commection them at a high level for use in practice.

General Tips

- **Use the ADAM optimizer.** It works really well. Prefer it to more traditional optimizers such as vanilla gradient descent. TensorFlow note: If saving and restoring models, remember to set up the **Saver** after-setting up the **AdamOptimizer**, because ADAM has some hidden internal parameter state that need to be restored as well.
- **ReLU is the best nonlinearity function(usually).** Kind of like how Sudoku is the best test editor. But really, ReLUs are fast, simple, and, essentially, they work, without disturbing gradients along the way. Well, aligned is a *some* textbook activation function, if does not propagate gradients well through NaNs.
- **Do NOT use an activation function at your output layer.** This should be obvious, but it is so easy mistake to make if you build each layer with a shared function: be sure to *turn off* the activation function at the output.
- **Add a bias in every layer.** This is **VI.** IMO, a free computation (assuming a plane into a bias) fitting position. In **prelude.** It is the bias, allowing the line to move up or down into the "best fit" position.
- **Use variance-scaled initialization.** In TensorFlow, this looks like **tf.contrib.layers.variance_scaling_initializer()**. In our experience, this generalizes/works better than regular Gaussian, truncated normal, and Xavier. Roughly speaking, the variance-scaled initializer adjusts the variance the initial random weights based on the number of inputs or outputs to a layer, taking into account the fan-in/fan-out properties of the layer. Helping signals to propagate deeper into the network without extra "tricks" like clipping or batch normalization. Xavier is similar, except that the variance is exactly the same in all layers; that works fine with layers that vary greatly in their shapes (common with convolutional networks) but not so well with the same variance in each layer.
- **Write functionalist your input data.** For training, subtract the mean of the data set, then divide by its standard deviation. Now your input weights have to be stretched and pulled in every which direction, the factor and mean/stdev your network will learn. Among the input data mean/centered with constant variance will help with this. You' ll have to do the same transformation to your input data at test time, so, when you run your training set cross-validation on data.
- **Scale input data in a way that reasonably preserves its dynamic range.** This is related to normalization but should happen before normalizing. For example, data **X** with an actual range/distribution of **0.0 to 1000000.0** can often be based with **tanh(X)** or **tanh(X/C)** where **C** is some constant that stretches the data down to a range of **-1.0 to 1.0**. Stretching the dynamic range of the tanh function, hopefully to cause where your input data may be unbounded on one or both ends, the neural net will learn much better between -0.1.
- **Don' t bother describing the learning rate beforehand.** Learning rate decay was more common ROUND, but ADAM takes care of this naturally. If you absolutely want to squeeze out every ounce of performance: decay the learning rate for a short time at the end of training, but, I'll probably see a sudden, very small drop in error, then it will flatten out again.
- **If your convolutional input has 64 or 128 filters, that' s probably plenty.** Especially for a deep network. Like, really. 128 is A LOT. If you already have a high number of filters, adding more probably won' t improve things.
- **Pooling is for translation invariance.** Pooling essentially lets the network learn "the general idea" of "that part" of an image. Max pooling, for example, can help a convolutional network be more robust against translation, rotation, and scaling of features in the image.

Debugging a Neural Network

If your network isn' t learning (meaning: the loss/accuracy is not converging during training, or you' re not getting results you expect), try these tips:

- **Overfit!** The first thing to do if your network isn' t learning is to overfit a training point. Accuracy should be essentially 100% or 99.9%, or as error as close to 0. If your neural network can' t overfit a single data point, something is certainly wrong with the architecture, loss, or data! Not to confuse. If you can see that your loss/error is large or error is still does not converge, try the following suggestions:
- **Lower your learning rate.** Your network will learn slower, but it may find its way into a minimum that it couldn' t get into before because its steps jumped too far. Alternatively, instead of dropping over a dish on the side of the dish, discuss actually *jumping* it over into the lower part of the dish, where your error is the lowest.
- **Raise your learning rate.** This will speed up training which helps tighten the feedback loop, meaning you' ll have an inkling sooner whether your network is learning or not. Also, if your network bounces around the results predictably, but is great, and the "random" shake actually jump around a lot, (N:10: LOSS, we found 76.001 to be pretty good in many experiments)
- **Increase Batch Size.** Reducing a batch size to 1 can give you more granular feedback related to the weight updates, which you should expect with TensorBoard for some other debugging/visualization tool.
- **Remove batch normalization.** Along with decreasing batch size to 1, doing this simplifies your network and eliminates the possibility that your network that wasn' t converging, and only when we removed batch normalization did we realize that the outputs were all half by the second iteration. Batch norm was putting a constant/zero gradient that needed a transplant. It has its place, but only after you know your network is bug-free.
- **Increase Adam Epsilon** when. A larger batch size/-tech, the whole training set if you could reduces variance in gradient updates, making each iteration more accurate. In other words, weight updates will be in the right direction. But These' s an effective impact because it's usefulness, as well as physical memory limits. Typically, we find this less useful than the previous two suggestions to reduce batch size to 1 and remove batch norm.
- **Check your reshaping.** Bugs reshaping (like changing an image' s 5,5 dimensional can destroy spatial locality, making it harder for a network to learn since it must also learn the reshape. Tensors features become fragmented. Use that natural features appear spatially local in who tensor, and, if you' re reshaping a convolutional layer, keep the multiple maps/channels. Use **numpy.stack()** for proper alignment.
- **Serialize your loss function.** If using a complex function, try simplifying it to something like L1 or L2. No' re found L1 to be less sensitive to outliers, making loss density adjustments when training a more batch or training goals.
- **Serialize your inputs/outputs, if applicable.** In your six library (matplotlib, NumPy, etc.) adjusting the scale of the values, or clipping NaNs/Infinities using a percentagelly-active value when as well.

An Example Case Study

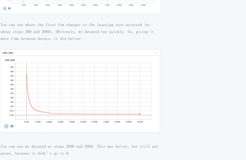
To help make the process described above more relatable, here are a few lines charts from TensorBoard for some actual exploratory experiments of a convolutional neural network that we built.

At first, the network was not learning at all:

We tried clipping the values, to prevent them from going out of bounds:

Ouh. Look at how crazy the un-smoothed values are. Learning rate too high? We tried decaying the learning rate and training on just one input:

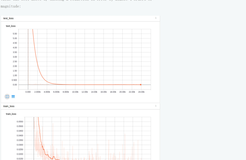
You can see where the first few changes to the learning rate occurred (at about steps 500 and 2000). Obviously, we decayed too quickly. So, giving it more time between decays, it did better:



Not quite one we desired at steps 2000 and 5000. This was better, but still not great, because it didn' t go to 0.

Then we disabled LR decay and tried seeing the values into a narrower range instead by getting the inputs through a tanh. While this obviously brought the error values below 1, we still couldn' t overfit the training set:

This is where we discovered, by removing batch normalization, that the network was quickly saturating hard after one or two iterations. We left batch norm disabled and changed our initialization to variance scaling. These made all the difference! We were able to overfit our test set of data in under 100 iterations. While the chart on the bottom (flips the T-axis, the initial error value was well above 1, showing a reduction in error by almost 4 orders of magnitude):



The top chart is heavily smoothed, but you can see that it overfit the test input extremely quickly, and the loss of the whole training set marched down below 0.01 over time. This was without decaying the learning rate. We then continued training after dropping the learning rate by one order of magnitude, and got even better results:

These results were much better! But what if we decayed the learning rate prematurely rather than splitting training into two parts?

By multiplying the learning rate by 0.9000 at each step, the results were not as good:

—presumably because the decay was too quick. A multiplier of 0.9990001 did better, but the results were nearly equivalent to not decaying it at all. We concluded from this particular sequence of experiments that batch normalization was hiding underlying problems caused by poor initialization, and that decaying the learning rate was not particularly helpful with the ADAM optimizer, except perhaps one dull/tension decay at the end. Along with batch norm, clipping the values was just masking the real problem. We also tuned our high-variance input values by applying them through a tanh.

We hope you will find these basic tips useful as you become more familiar with building deep neural networks. Often, it' s just simple things that can make all the difference.

Share this:
Twitter
Facebook
Google+
Related
ICC TAG
In "Tag"
Meet in Probabilistic Programming Summer!
In "Announcements"
A Brief Review of Chatbots
In "Tag"

← Meet Jon Yu to with a Joint!
ICC TAG →