Deep learning is tricky in several respects. Not only can the math and theory quickly lead to hairballs of gradient formulas and update equations, but deep learning models are also complex and often finicky pieces of software. Recently, we've seen promising projects like TDB for TensorFlow, which promises online visualization of neural networks and control flow interruption during training and inference, helping the developer diagnose the behavior of a neural network that isn't working.

In our adventures with Theano, Keras and neon, however, my colleagues and I often had problems verifying that we had represented the data correctly, figuring out if some bug or undesirable behavior was our fault or someone else's, and even getting our scripts to run at all. In my work on sunny-side-up, Lab41's exploration of deep learning for sentiment analysis, I often muttered some variant of the phrase "This bleeding convnet..." At most such times, I didn't even technically have a convolutional neural network, but instead a sad, broken piece of software trying valiantly to function as one. In this post I offer a few tips, tricks, and outright hacks for transforming your code into happy, "working" (no guarantees!) convolutional neural networks and other deep learning models. These tips are, in rough order:

- Start small
- Use funny numbers
- Debug with debuggers

**Start small**

The state-of-the-art deep learning architectures are only getting bigger and deeper. As a programmer trying to implement these architectures, this is not a problem, provided you know exactly what you are doing. If your knowledge is less than perfect, then things will go wrong—whether it's trying to replicate someone else's results or forging ahead beyond the limits of what has already been done. In particular, I have found two measures particularly help when you are just starting a new deep learning project: validate your data model with fake data, and proceed incrementally when building up architectures.

**Fake your data, fake your results**

My first piece of advice, which lands mostly in the "do as I say" category, is to start with the simplest possible architecture and decide on your representation of the data. Implement a small fully-connected feed-forward network (even just a logistic regression) for some version of your problem. Create a simulated data stream with the same shape and dimensionality of the data you plan on pushing through the net. Try data where the relationship between the input and the output is deterministic and easy to learn, and another where there is no relationship between input and output, and compare the performance of your algorithm:

```python
1 # Easily separable
2 a1 = np.random.normal(0.25, 0.1, 50)
3 b1 = np.random.normal(0.75, 0.1, 50)
4 # Not separable
```
These samples are plotted below. Which of these relationships would you expect to be easier to learn?

Call it a unit test of sorts. Fake data will set some sanity checks on the behavior of the algorithm you end up implementing. And you will need your sanity.

One thing at a time

For sunny-side-up, we implemented a nine-layer convolutional neural network from a paper by Xiang Zhang and collaborators at NYU. The reference implementation is in Torch. My colleagues and I initially set out to replicate this architecture in Theano, Keras and neon, and our efforts dragged on for days, even weeks.

In my own case, I eventually realized it was because I was updating the code for data ingest and for the classifier at the same time. Theano is one thing, but when it comes down to it, neon and Keras have fairly straightforward model specification idioms, so in theory it should be hard to screw them up too badly. But I had unwittingly introduced subtle changes in the way the data was represented when it arrived at the first model layer. For better or for worse, many architectures get very confused when you feed in a matrix whose dimensions don't match what that layer was expecting, leading to fun error messages like this:

```
5  a2 = np.random.normal(0.5, 0.3, 50)
6  b2 = np.random.normal(0.5, 0.3, 50)
```

So if your data model might be broken, fall back to your unit test and fix it first there. Then build back up to the network you’re implementing. In any event, what was that about array sizes...?

Diagnose array dimension mismatch with funny numbers

Sometimes the neural network framework you are using doesn't adequately document the expected shape of input to a layer. Or, it could be that maybe you didn't read the documentation, or implemented the layer
inconsistently. However it happens, it can be surprisingly hard to get the number of dimensions right going from one network layer to the next. Even in fairly simple architectures, it is not difficult to arrive at a point where you have \( F \times W \times I \) filters convolving over the third and fourth dimensions of a \( K \times I \times M \times N \) array. My trick here is to selectively edit the shape of the data, the batch size, or the input dimensions of particular layers in a way that makes them easier to identify if an error message comes up. Note that this very hacky technique (a very primitive cousin of debug-by-print) should go hand-in-hand with more rigorous dimensional analysis of the data and the model, and taking advantage of facilities such as neon’s for naming individual layers, which could circumvent the entire problem.

For instance, when you end up switching the dimension for the mini-batch size and the dimension for number of channels (like I did yesterday), your code will blow up, maybe with an error message if you’re lucky. The message will often (helpfully!) tell you which dimension just didn’t match up to expectations, but in my case it reported only the dimension size it encountered rather than the name I had been using using for it, or even which layer in particular was amiss. Just today I got the following very informative stack trace from Theano:

```
1 GpuCorrMM images and kernel must have the same stack size
2 3 Apply node that caused the error: GpuCorrMM{valid, (1, 1)}
4 (GpuContiguous.0, GpuContiguous.0)
5 Toposort index: 76
6 Inputs types: [CudaNdarrayType(float32, 4D),
7 CudaNdarrayType(float32, 4D)]
8 Inputs shapes: [(256, 1, 64, 64), (20, 32, 5, 5)]
9 Inputs strides: [(4096, 0, 64, 1), (800, 25, 5, 1)]
10 Inputs values: [not shown, not shown]
11 Outputs clients: [[GpuElemwiseAdd](0, 0)]
12 (GpuCorrMM{valid, (1, 1)}.0, GpuDimShuffle(x,0,x,x).0)]
```

I haven’t had any luck making use of the tantalizing Toposort index one often sees in such messages, and while the Inputs shapes line contains a lot of valuable information, it can be difficult to cross-reference it with the graph of your actual network. To be fair, at least Theano is nice enough to suggest ways to elicit more readable and verbose output, but don’t hold your breath for an easy fix. Confusingly, some frameworks also restrict the products of certain quantities, and these will often appear in errors as literal constants, e.g. 45, rather than \( K \times M \). Everyone loves AssertionErrors: 114688 > 2**16, right?

In general, the most thorough approach to fixing bad error messages would be to go into the code for your chosen framework, write helpful error messages, and push your improvements back to the maintainers. If you improve the software, everyone will be happier, wealthier and longer-lived. Such magnanimity can, however, be hard to muster in the midst of debugging this bleeping convnet. Barring that, try manipulating dimensions of interest: mini-batch size, width, number of channels, etc. to be prime (or just funny) numbers. If you see Error: Expected foo==26, you’re much more likely to see quickly that the problem involves your two dimensions measuring 2 and 13. I did this to debug my data model all the time, so that I knew if a problem arose involving the quantity 333 (1077) = 358641, it somehow involved my mini-batch size (333 records per mini-batch) or my data length (1077 characters). If this seems hacky, well, I don’t have much to say about that.

In most real-life neural network architectures, you don’t necessarily have the freedom to arbitrarily set the dimensionality of particular layers. For instance, one layer may accept as input the result of a convolution-and-pooling operation, which puts certain restrictions on possible dimension sizes. But in my experience, the interface between the data and the first layer is both the most problematic and the easiest to fiddle with. For
higher layers, meanwhile, you can often get into a ballpark that at least helps point the way toward the problem.

**Pdb might actually work**

As your neural network software slowly stops being a haphazard mess of stack dump-inducing spaghetti code and increasingly exasperated comments (# trying this againnn...) and flowers into a "working," possibly very quirky neural network training and testing system, tools like TDB, TensorBoard, neon's nvis, or your own bespoke logging and visualization systems will become increasingly useful in doing actual deep learning debugging: checking gradients, tweaking learning rates, and addressing other phenomena in the "this output is not what I wanted or expected" category—the fun stuff. But the squishy middle stage between broken and working can be frustrating, painful, and long.

My parting comment is that one of the most useful tools for debugging my neural networks in this stage was, strangely enough, the debugger—specifically IPython's ipdb, which provides IPythonified hooks to pdb. For certain kinds of errors, breaking on exception (as simple as issuing pdb in interactive IPython or a notebook) dumps you right into a place where you can print out array dimensions and start trying to fiddle and tinker. There are often several layers of logic hiding you from whatever is actually training your network, however, so pdb may not always be the right choice.

Of course, the right choice is always to know what you're doing and never to make any mistakes. This may involve sketching your network architecture out on paper, doing a masters in machine learning, or getting someone else do either one of those for you and going to the beach instead. If you are a fallible human being, though, or you've blown through your vacation budget already, I hope these nuggets of intermittently sound advice help you iterate faster and (eventually) get your bleeping convnet to work.