1. Verify that your code is bug free

There's a saying among writers that "All writing is re-writing" -- that is, the greater part of writing is revising. For programmers (or at least data scientists) the expression could be re-phrased as "All coding is debugging."

Any time you're writing code, you need to verify that it works as intended. The best method I've ever found for verifying correctness is to break your code into small segments, and verify that each segment works. This can be done by comparing the segment output to what you know to be the correct answer. This is called unit testing. Writing good unit tests is a key piece of becoming a good statistician/data scientist/machine learning expert/neural network practitioner. There is simply no substitute.

You have to check that your code is free of bugs before you can tune network performance! Otherwise, you might as well be re-arranging deck chairs on the *RMS Titanic*.

There are two features of neural networks that make verification even more important than for other types of machine learning or statistical models.

1. Neural networks are not "off-the-shelf" algorithms in the way that random forest or logistic regression are. Even for simple, feed-forward networks, the onus is largely on the user to make numerous decisions about how the network is configured, connected, initialized and optimized. This means writing code, and writing code means debugging.
2. *Even when a neural network code executes without raising an exception, the network can still have bugs!* These bugs might even be the insidious kind for which the network will train, but get stuck at a sub-optimal solution, or the resulting network does not have the desired architecture. (This is an example of the difference between a syntactic and semantic error.)

This *Medium* post, "How to unit test machine learning code," by Chase Roberts discusses unit-testing for machine learning models in more detail. I borrowed this example of buggy code from the article:

```
def make_convnet(input_image):
    net = slim.conv2d(input_image, 32, [11, 11], scope="conv1_11x11")
    net = slim.conv2d(input_image, 64, [5, 5], scope="conv2_5x5")
    net = slim.max_pool2d(net, [4, 4], stride=4, scope='pool1')
    net = slim.conv2d(input_image, 64, [5, 5], scope="conv3_5x5")
    net = slim.conv2d(input_image, 128, [3, 3], scope="conv4_3x3")
    net = slim.max_pool2d(net, [2, 2], scope='pool2')
    net = slim.conv2d(input_image, 128, [3, 3], scope="conv5_3x3")
    net = slim.max_pool2d(net, [2, 2], scope='pool3')
    net = slim.conv2d(input_image, 32, [1, 1], scope="conv6_1x1")
    return net
```

Do you see the error? Many of the different operations are not *actually used* because previous results are over-written with new variables. Using this block of code in a network will still train and the weights will update and the loss might even decrease -- but the code definitely isn't doing what was intended. (The author is also inconsistent about using single- or double-quotes but that's purely stylistic.)

The most common *programming* errors pertaining to neural networks are

- Variables are created but never used (usually because of copy-paste errors);
- Expressions for gradient updates are incorrect;
- Weight updates are not applied;
- Loss functions are not measured on the correct scale (for example, cross-entropy loss can be expressed in terms of probability or logits)
- The loss is not appropriate for the task (for example, using categorical cross-entropy loss for a regression task).
- Dropout is used during testing, instead of only being used for training.

- Make sure you're minimizing the loss function $L(x)$���(���), instead of minimizing $-L(x)$−���(���).
- Make sure your loss is computed correctly.

Unit testing is not just limited to the neural network itself. You need to test all of the steps that produce or transform data and feed into the network. Some common mistakes here are

- NA or NaN or Inf values in your data creating NA or NaN or Inf values in the output, and therefore in the loss function.
- Shuffling the labels independently from the samples (for instance, creating train/test splits for the labels and samples separately);
- Accidentally assigning the training data as the testing data;
- When using a train/test split, the model references the original, non-split data instead of the training partition or the testing partition.
- Forgetting to scale the testing data;
- Scaling the testing data using the statistics of the test partition instead of the train partition;
- Forgetting to un-scale the predictions (e.g. pixel values are in [0,1] instead of [0, 255]).
- Here's an example of a question where the problem appears to be one of model configuration or hyperparameter choice, but actually the problem was a subtle bug in how gradients were computed. Is this drop in training accuracy due to a statistical or programming error?

2. For the love of all that is good, ***scale your data***

The scale of the data can make an enormous difference on training. Sometimes, networks simply won't reduce the loss if the data isn't scaled. Other networks will decrease the loss, but only very slowly. Scaling the inputs (and certain times, the targets) can dramatically improve the network's training.

- Prior to presenting data to a neural network, **standardizing** the data to have 0 mean and unit variance, or to lie in a small interval like $[-0.5, 0.5]$$[-0.5, 0.5]$ can improve training. This amounts to pre-conditioning, and removes the effect that a choice in units has on network weights. For example, length in millimeters and length in kilometers both represent the same concept, but are on different scales. The exact details of how to standardize the data depend on what your data look like.

- Data normalization and standardization in neural networks
    - Why does $[0, 1]$$[0, 1]$ scaling dramatically increase training time for feed forward ANN (1 hidden layer)?

- **Batch or Layer normalization** can improve network training. Both seek to improve the network by keeping a running mean and standard deviation for neurons' activations as the network trains. It is not well-understood why this helps training, and remains an active area of research.
    - "Understanding Batch Normalization" by Johan Bjorck, Carla Gomes, Bart Selman
    - "Towards a Theoretical Understanding of Batch Normalization" by Jonas Kohler, Hadi Daneshmand, Aurelien Lucchi, Ming Zhou, Klaus Neymeyr, Thomas Hofmann
    - "How Does Batch Normalization Help Optimization? (No, It Is Not About Internal Covariate Shift)" by Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, Aleksander Madry

3. Crawl Before You Walk; Walk Before You Run

Wide and deep neural networks, and neural networks with exotic wiring, are the Hot Thing right now in machine learning. But these networks didn't spring fully-formed into existence; their designers built up to them from smaller units. First, build a small network with a single hidden layer and verify that it works correctly. Then incrementally add additional model complexity, and verify that each of those works as well.

- Too *few* **neurons** in a layer can restrict the representation that the network learns, causing under-fitting. Too many neurons can cause over-fitting because the network will "memorize" the training data.

  Even if you can prove that there is, mathematically, only a small number of neurons necessary to model a problem, it is often the case that having "a few more" neurons makes it *easier* for the optimizer to find a "good" configuration. (But I don't think anyone fully understands why this is the case.) I provide an example of this in the context of the XOR problem here: Aren't my iterations needed to train NN for XOR with MSE < 0.001 too high?.
- Choosing the number of **hidden layers** lets the network learn an abstraction from the raw data. Deep learning is all the rage these days, and networks with a large number of layers have shown impressive results. But adding too many hidden layers can make risk overfitting or make it very hard to optimize the network.
- Choosing a clever **network wiring** can do a lot of the work for you. Is your data source amenable to specialized network architectures? Convolutional neural networks can achieve impressive results on "structured" data sources, image or audio data. Recurrent neural networks can do well on sequential data types, such as natural language or time series data. Residual connections can improve deep feed-forward networks.

4. Neural Network Training Is Like Lock Picking

To achieve state of the art, or even merely good, results, you have to set up all of the parts configured to work well *together*. Setting up a neural network configuration that actually learns is a lot like picking a lock: all of the pieces have to be lined up *just right*. Just as it is not sufficient to have a single tumbler in the right place, neither is it sufficient to have only the architecture, or only the optimizer, set up correctly.

Tuning configuration choices is not really as simple as saying that one kind of configuration choice (e.g. learning rate) is more or less important than another (e.g. number of units), since all of these choices interact with all of the other choices, so one choice can do well *in combination with another choice made elsewhere*.

This is a non-exhaustive list of the configuration options which are not also regularization options or numerical optimization options.

All of these topics are active areas of research.

- The network **initialization** is often overlooked as a source of neural network bugs. Initialization over too-large an interval can set initial weights too large, meaning that single neurons have an outsize influence over the network behavior.
- The key difference between a neural network and a regression model is that a neural network is a composition of many nonlinear functions, called **activation functions**. (See: What is the essential difference between neural network and linear regression)

  Classical neural network results focused on sigmoidal activation functions (logistic or $\tanh$tanh functions). A recent result has found that ReLU (or similar) units tend to work better because the have steeper gradients, so updates can be applied quickly. (See: Why do we use ReLU in neural networks and how do we use it?) One caution about ReLUs is the "dead neuron" phenomenon, which can stymie learning; leaky relus and similar variants avoid this problem. See
- Why can't a single ReLU learn a ReLU?
- My ReLU network fails to launch

There are a number of other options. See: Comprehensive list of activation functions in neural networks with pros/cons

- Residual connections are a neat development that can make it easier to train neural networks. "Deep Residual Learning for Image Recognition" Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun In: CVPR. (2016). Additionally, changing the order of operations within the residual block can further improve the resulting network. "Identity Mappings in Deep Residual Networks" by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun.

5. Non-convex optimization is hard

The objective function of a neural network is only convex when there are no hidden units, all activations are linear, and the design matrix is full-rank -- because this configuration is identically an ordinary regression problem.

In all other cases, the optimization problem is non-convex, and non-convex optimization is hard. The challenges of training neural networks are well-known (see: Why is it hard to train deep neural networks?). Additionally, neural networks have a very large number of parameters, which restricts us to solely first-order methods (see: Why is Newton's method not widely used in machine learning?). **This is a very active area of research.**

- Setting the **learning rate** too large will cause the optimization to diverge, because you will leap from one side of the "canyon" to the other. Setting this too small will prevent you from making any real progress, and possibly allow the noise inherent in SGD to overwhelm your gradient estimates. See:
  - How can change in cost function be positive?
- **Gradient clipping** re-scales the norm of the gradient if it's above some threshold. I used to think that this was a set-and-forget parameter, typically at 1.0, but I found that I could make an LSTM language model dramatically better by setting it to 0.25. I don't know why that is.
- **Learning rate scheduling** can decrease the learning rate over the course of training. In my experience, trying to use scheduling is a lot like regex: it replaces one problem ("How do I get learning to continue after a certain epoch?") with two problems ("How do I get learning to continue after a certain epoch?" and "How do I choose a good schedule?"). Other people insist that scheduling is essential. I'll let you decide.
- Choosing a good **minibatch size** can influence the learning process indirectly, since a larger mini-batch will tend to have a smaller variance (law-of-large-numbers) than a smaller mini-batch. You want the mini-batch to be large enough to be informative about the direction of the gradient, but small enough that SGD can regularize your network.
- There are a number of variants on **stochastic gradient descent** which use momentum, adaptive learning rates, Nesterov updates and so on to improve upon vanilla SGD. Designing a better optimizer is very much an active area of research. Some examples:
  - No change in accuracy using Adam Optimizer when SGD works fine
  - How does the Adam method of stochastic gradient descent work?
  - Why does momentum escape from a saddle point in this famous image?
- When it first came out, the Adam optimizer generated a lot of interest. But some recent research has found that SGD with momentum can out-perform adaptive gradient methods for neural networks. "The Marginal Value of Adaptive Gradient Methods in Machine Learning" by Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, Benjamin Recht
- But on the other hand, this very recent paper proposes a new adaptive learning-rate optimizer which supposedly closes the gap between adaptive-rate methods and SGD with momentum. "Closing the Generalization Gap of Adaptive Gradient Methods in Training Deep Neural Networks" by Jinghui Chen, Quanquan Gu

  > Adaptive gradient methods, which adopt historical gradient information to automatically adjust the learning rate, have been observed to generalize worse than stochastic gradient descent (SGD) with momentum in training deep neural networks. This leaves how to close the generalization gap of adaptive gradient methods an open problem. In this work, we show that adaptive gradient methods such as Adam, Amsgrad, are sometimes "over adapted". We design a new algorithm, called Partially adaptive momentum estimation method (Padam), which unifies the Adam/Amsgrad with SGD to achieve the best from both worlds. Experiments on standard benchmarks show that Padam can maintain fast convergence rate as Adam/Amsgrad while generalizing as well as SGD in training deep neural networks. These results would suggest practitioners pick up adaptive gradient methods once again for faster training of deep neural networks.

- Specifically for triplet-loss models, there are a number of tricks which can improve training time and generalization. See: In training a triplet network, I first have a solid drop in loss, but eventually the loss slowly but consistently increases. What could cause this?

6. Regularization

Choosing and tuning network regularization is a key part of building a model that generalizes well (that is, a model that is not overfit to the training data). However, at the time that your network is struggling to decrease the loss on the training data -- when the network is not learning -- regularization can obscure what the problem is.

When my network doesn't learn, I turn off all regularization and verify that the non-regularized network works correctly. Then I add each regularization piece back, and verify that each of those works along the way.

This tactic can pinpoint where some regularization might be poorly set. Some examples are

- L^2���^{2} regularization (aka weight decay) or L^1���^{1} regularization is set too large, so the weights can't move.
- Two parts of regularization are in conflict. For example, it's widely observed that layer normalization and dropout are difficult to use together. Since either on its own is very useful, understanding how to use both is an active area of research.
  - "Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift" by Xiang Li, Shuo Chen, Xiaolin Hu, Jian Yang
  - "Adjusting for Dropout Variance in Batch Normalization and Weight Initialization" by Dan Hendrycks, Kevin Gimpel.
  - "Self-Normalizing Neural Networks" by Günter Klambauer, Thomas Unterthiner, Andreas Mayr and Sepp Hochreiter

7. Keep a Logbook of Experiments

When I set up a neural network, I don't hard-code any parameter settings. Instead, I do that in a configuration file (e.g., JSON) that is read and used to populate network configuration details at runtime. I keep all of these configuration files. If I make any parameter modification, I make a new configuration file. Finally, I append as comments all of the per-epoch losses for training and validation.

The reason that I'm so obsessive about retaining old results is that this makes it very easy to go back and review previous experiments. It also hedges against mistakenly repeating the same dead-end experiment. Psychologically, it also lets you look back and observe "Well, the project might not be where I want it to be today, but I am making progress compared to where I was k��� weeks ago."

As an example, I wanted to learn about LSTM language models, so I decided to make a Twitter bot that writes new tweets in response to other Twitter users. I worked on this in my free time, between grad school and my job. It took about a year, and I iterated over about 150 different models before getting to a model that did what I wanted: generate new English-language text that (sort of) makes sense. (One key sticking point, and part of the reason that it took so many attempts, is that it was not sufficient to simply get a low out-of-sample loss, since early low-loss models had managed to memorize the training data, so it was just reproducing germane blocks of text verbatim in reply to prompts -- it took some tweaking to make the model more spontaneous and still have low loss.)