# Unit Testing Is Your Friend

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](https://en.wikipedia.org/wiki/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](https://web.archive.org/web/20161201151434/https:/wci.llnl.gov/codes/basis/manual/node53.html).)

This Medium post, "[How to unit test machine learning code](https://medium.com/@keeper6928/how-to-unit-test-machine-learning-code-57cf6fd81765)," 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).

# 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?](https://stats.stackexchange.com/questions/351216/arent-my-iterations-needed-to-train-nn-for-xor-with-mse-0-001-too-high/351713#351713).

* 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.

# Neural Network Training Is Like Lock Picking

To achieve state of the art, or even merely good, results, you have to have to have 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](https://stats.stackexchange.com/questions/259950/what-is-the-essential-difference-between-neural-network-and-linear-regression))

Classical neural network results focused on sigmoidal activation functions (logistic or 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?](https://stats.stackexchange.com/questions/226923/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?](https://stats.stackexchange.com/questions/379884/why-cant-a-single-relu-learn-a-relu)
* [My ReLU network fails to launch](https://stats.stackexchange.com/questions/188040/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](https://stats.stackexchange.com/questions/115258/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"](https://arxiv.org/abs/1512.03385) 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](https://arxiv.org/pdf/1603.05027v3.pdf)" by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun.

# 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?](https://stats.stackexchange.com/questions/262750/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?](https://stats.stackexchange.com/questions/253632/why-is-newtons-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?](https://stats.stackexchange.com/questions/364360/how-can-change-in-cost-function-be-positive/364366#364366)
* **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](https://blog.codinghorror.com/regular-expressions-now-you-have-two-problems/): 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](https://stats.stackexchange.com/questions/tagged/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](https://stats.stackexchange.com/questions/313278/no-change-in-accuracy-using-adam-optimizer-when-sgd-works-fine)
  + [How does the Adam method of stochastic gradient descent work?](https://stats.stackexchange.com/questions/220494/how-does-the-adam-method-of-stochastic-gradient-descent-work/220563#comment661981_220563)
  + [Why does momentum escape from a saddle point in this famous image?](https://stats.stackexchange.com/questions/308835/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](https://arxiv.org/abs/1705.08292)" 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](https://arxiv.org/abs/1806.06763v1)" 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](https://stats.stackexchange.com/questions/tagged/triplet-loss) models, there are a number of tricks which can improve training time and generalization. See: [In training, I first have a solid drop in loss, but eventually the loss slowly but consistently increases. What could cause this?](https://stats.stackexchange.com/questions/475655/in-training-i-first-have-a-solid-drop-in-loss-but-eventually-the-loss-slowly-b)

# Normalization

The scale of the data can make a big difference on 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]
* 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](https://stats.stackexchange.com/questions/7757/data-normalization-and-standardization-in-neural-networks)
  + [Why does [0,1]](https://stats.stackexchange.com/questions/364735/why-does-0-1-scaling-dramatically-increase-training-time-for-feed-forward-an/364776#364776)
  + [scaling dramatically increase training time for feed forward ANN (1 hidden layer)?](https://stats.stackexchange.com/questions/364735/why-does-0-1-scaling-dramatically-increase-training-time-for-feed-forward-an/364776#364776)
* **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](https://arxiv.org/abs/1806.02375v1)" by Johan Bjorck, Carla Gomes, Bart Selman
  + "[Towards a Theoretical Understanding of Batch Normalization](https://arxiv.org/abs/1805.10694v1)" 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)](https://arxiv.org/abs/1805.11604v2)" by Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, Aleksander Madry

# 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

regularization (aka weight decay) or *L*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](https://arxiv.org/abs/1801.05134v1)" by Xiang Li, Shuo Chen, Xiaolin Hu, Jian Yang
  + "[Adjusting for Dropout Variance in Batch Normalization and Weight Initialization](https://arxiv.org/abs/1607.02488v2)" by Dan Hendrycks, Kevin Gimpel.
  + "[Self-Normalizing Neural Networks](https://arxiv.org/abs/1706.02515)" by Günter Klambauer, Thomas Unterthiner, Andreas Mayr and Sepp Hochreiter

# 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.)

# Sanity checks

1) **Train your model on a single data point. If this works, train it on two inputs with different outputs.**

This verifies a few things. First, it quickly shows you that your model is able to learn by checking if your model can overfit your data. In my case, I constantly make silly mistakes of doing Dense(1,activation='softmax') vs Dense(1,activation='sigmoid') for binary predictions, and the first one gives garbage results.

If your model is unable to overfit a few data points, then either it's too small (which is unlikely in today's age),or something is wrong in its structure or the learning algorithm.

2) **Pay attention to your initial loss.**

Continuing the binary example, if your data is 30% 0's and 70% 1's, then your intial expected loss around *L*=−0.3ln(0.5)−0.7ln(0.5)≈0.7

. This is because your model should start out close to randomly guessing.

A lot of times you'll see an initial loss of something ridiculous, like 6.5. Conceptually this means that your output is heavily saturated, for example toward 0. For example −0.3ln(0.99)−0.7ln(0.01)=3.2

, so if you're seeing a loss that's bigger than 1, it's likely your model is very skewed. This usually happens when your neural network weights aren't properly balanced, especially closer to the softmax/sigmoid. So this would tell you if your initialization is bad.

You can study this further by making your model predict on a few thousand examples, and then histogramming the outputs. This is especially useful for checking that your data is correctly normalized. As an example, if you expect your output to be heavily skewed toward 0, it might be a good idea to transform your expected outputs (your training data) by taking the square roots of the expected output. This will avoid gradient issues for saturated sigmoids, at the output.

3) **Generalize your model outputs to debug**

As an example, imagine you're using an LSTM to make predictions from time-series data. Maybe in your example, you only care about the latest prediction, so your LSTM outputs a single value and not a sequence. Switch the LSTM to return predictions at each step (in keras, this is return\_sequences=True). Then you can take a look at your hidden-state outputs after every step and make sure they are actually different. An application of this is to make sure that when you're masking your sequences (i.e. padding them with data to make them equal length), the LSTM is correctly ignoring your masked data. Without generalizing your model you will never find this issue.

4) **Look at individual layers**

Tensorboard provides a useful way of [visualizing your layer outputs](https://jhui.github.io/2017/03/12/TensorBoard-visualize-your-learning/). This can help make sure that inputs/outputs are properly normalized in each layer. It can also catch buggy activations. You can also query layer outputs in keras on a batch of predictions, and then look for layers which have suspiciously skewed activations (either all 0, or all nonzero).

5) **Build a simpler model first**

You've decided that the best approach to solve your problem is to use a CNN combined with a bounding box detector, that further processes image crops and then uses an LSTM to combine everything. It takes 10 minutes just for your GPU to initialize your model.

Instead, make a batch of fake data (same shape), and break your model down into components. Then make dummy models in place of each component (your "CNN" could just be a single 2x2 20-stride convolution, the LSTM with just 2 hidden units). This will help you make sure that your model structure is correct and that there are no extraneous issues. I struggled for a while with such a model, and when I tried a simpler version, I found out that one of the layers wasn't being masked properly due to a keras bug. You can easily (and quickly) query internal model layers and see if you've setup your graph correctly.

6) **Standardize your Preprocessing and Package Versions**

Neural networks in particular are extremely sensitive to small changes in your data. As an example, two popular image loading packages are cv2 and PIL. Just by virtue of opening a JPEG, both these packages will produce **slightly different** images. The differences are usually really small, but you'll occasionally see drops in model performance due to this kind of stuff. Also it makes debugging a nightmare: you got a validation score during training, and then later on you use a different loader and get different accuracy on the same darn dataset.

So if you're downloading someone's model from github, pay close attention to their preprocessing. What image loaders do they use? What image preprocessing routines do they use? When resizing an image, what interpolation do they use? Do they first resize and then normalize the image? Or the other way around? What's the channel order for RGB images?

The safest way of standardizing packages is to use a requirements.txt file that outlines all your packages just like on your training system setup, down to the keras==2.1.5 version numbers. In theory then, using Docker along with the same GPU as on your training system should then produce the same results.