Testing in Research, and Getting Neural Networks To Actually Learn Something

This document is split into two main parts: the first covers how you can trust your code, and the second is how you tweak and debug models once you trust your code.

Why bother?

Research code is, by nature, constantly shifting. We're often working out the exact hypothesis to be testing while simultaneously developing the techniques needed to verify it. It is often the case that, in pursuing a single hypothesis, we have to develop and discard a series of sub-hypotheses and techniques before we find the full set that works.

In that sense, we're writing code to reach a target that constantly changes, sometimes even before we're reached it. Parts of code become outdated, and functions you are writing change what they need to do, sometimes even before a previous rewrite is complete.

It's crucial that we can trust our code as we run experiments. A false-negative experiment at the wrong time will cost you time, and could cost you a paper, an entire discovery, or more.

Testing is the only good way we have to believe what your code tells you.

Cognitive Load

The limiting factor when debugging code is your cognitive load. When you are digging through code that isn't working, you generally want the set of states that the system could be in at each point to be as small as possible.

We already do this while writing code (abstraction! objects!), and writing good tests will help us reduce the cognitive load while debugging.

This notion of cognitive load was best expressed by Brian Kernighan as:

> "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."
>
> - Kernighan's law.

Testing techniques

For every bit of code that takes more than one sentence to explain, you should write some basic unit tests[1] to catch bugs. Here are some examples of unit tests you could write:

- Some basic values you can easily compute by hand or write down as an expression. (*general function* tests)
- More values that are close to existing functions.
- Values close to discontinuities or branches in the code. (*branch coverage* tests.)
- Illegal or disallowed input values, with a check to ensure that your function either throws an appropriate error or propagates the values.
- For bugs you have already caught and fixed (*regression* tests)

As a general rule, your test cases should be individually terse and cover the most common code paths. When writing tests, add a comment explaining what the test does, this will help you quickly figure out what the problem is.

And finally, remember that tests make it less likely that your code has bugs; they cannot guarantee they don't have any.

Example: *we have implemented a SmoothReLU function in PyTorch, that is:*
`lambda x: x**2 if 0 < x < 1 else torch.clamp(x, min=0)`
To test this, and its first derivative, we would write these tests:

- [ ] *Values in the three code-path ranges: (-1, 0.5, 2)*
- [ ] *Edge-case values: (0.0, 1e-13, 1e+13, 1.0, 1.0 + 1e-13, 1.0 - 1e-13)*
- [ ] *Weird values: (NaN, +Inf, -Inf), ensure NaNs are propagated to the output.*
- [ ] *Test gradient computation for all these (we're going to be using this for autodiff, and we want to be sure that the gradient is well-defined and correct around the discontinuous points 0.0, 1.0)*

Abstraction

Every research project, at one point or another, involves horribly written spaghetti code with unclear dependencies. We all know this is bad, but I imagine we all do it. I certainly do.

One of the benefits of writing tests is that it forces you to make clean(ish) abstractions right from the start. If you find you need to test a bit of code inside a function, consider making it a single-purpose function. (By convention, in Python, functions starting with _ are not meant for use outside the file they are defined in. Use that!)

Some additional tips:

[1] Here we're talking about unit tests. We'll cover integration and system/end-to-end tests later.

- A clunky abstraction is better than none.
- Wherever possible, make dependencies explicit.
- If you have hyperparameters or global parameters, wrap them in an immutable object and pass the object around.
- Parametrize your scripts as much as possible; use an argument parser like `argparse` or a configuration manager like FAIR's Hydra.

One of the benefits of abstraction is compatibility between modules. Designing *incompatible* abstractions is also a useful technique:

- if the meaning of a function changes, change the name! (example: when the function changes from computing total loss to mean loss, change the name accordingly!)
- If you have a function with multiple return values, use a dictionary with named keys instead of a tuple. You can then change or remove return values and have dependent code fail.

A word of caution for people using Jupyter Notebooks: it is possible for the global namespace to have functions that have been removed. Consider periodically clearing the output and restarting the kernel to prevent accidentally depending on removed code.

Types and Shapes

Type systems are your friend when it comes to preventing entire classes of bugs.

If your experiments involve structured data, define custom types and annotate your functions to enforce them. This prevents entire classes of bugs, especially when you have data that looks identical but means very different things.

> ***Example:*** *if you are computing a continuous-time ODE, you will often have some state* `x`*, and some time-derivative* `dx`*. These will be the same shape and likely the same type, but* `successor(x, dx)` *and* `successor(dx, x)` *have vastly different meanings! Wrap or annotate them with a custom type and you will eliminate this type of error.*

Runtime shape checking is also useful to prevent shape-related bugs. Even if a malformed shape would result in some error being thrown as the function executes, it is generally easier to test for this ahead of time.

You should enforce type- and shape-checking. A common (if repetitive) way to do it is to check types with `isinstance` and structure with `.shape` wherever needed. To automate this in Python, you could try using `mypy` for static checking and `pydantic` for dynamic type enforcement. I've found support for NumPy and PyTorch to still be a little spotty, but your mileage may vary. (Thinc.ai is a newer Python framework for handling neural networks that does this natively, but I have not yet tried it.)

As an aside, NaN poisoning is a common technique for catching computational errors: most functions that deal with floating-point input values will return NaN if any of the inputs are NaN. Any computation errors propagate up the computation graph (“poisoning” the computation). You should (at minimum) check if any of your loss terms are unexpectedly NaN so you can debug that.

Integration Testing and Oracles

When we pull together (individually unit-tested) parts into a script, we need to do some integration testing to ensure the code works as a whole.

A general way to do this is to *restrict the problem domain*: we either limit the input, the model, or structure of either so we restrict the output and can manually test this. You can do this by:

- Switching out your deep model for a linear model and comparing it to an existing implementation (like scipy.stats.linregress)
- Creating synthetic input data so you know what the output distribution should look like.
- Using a small problem and a reference implementation and checking the outputs match. (*Oracle* testing)
- Using an existing, known problem to check that your code performs as it should.

We’ll be discussing these approaches in the second part, where we look at failure analysis of neural networks.

Other people’s code

Be careful when using other people’s code. In my experience, the biggest time-waster is in the *double illusion of transparency*. This happens when the original developer makes one set of assumptions, you make a different set of assumptions, and the words used in the documentation and/or communication “make sense” to both people while meaning different things.

When confronted with some unexpected behavior and you suspect this is happening, the best way is to write a pair of tests: the simplest possible test that should succeed, and the simplest possible test that should fail. If you write these as executable code, you can then add this to the ensemble of tests (with comments explaining them).[2]

There are other risks, of course:

- Implementations may or may not be numerically stable around certain points.
- Floating point size and handling is slightly different between different languages, particularly languages that compile code down. For example, when using Julia code in Python, the floating point behavior is slightly different.

[2] I used the simplest-pair-of-tests technique when teaching *Intro to AI*, and it really helped make both students and me think critically about their understanding of concepts and code.

A general way to deal with these problems is a combination of simple functions tests (where you compare outputs between multiple implementations) and oracle tests (structured or limited input to an implementation evaluated by another script.)

Getting neural networks to actually learn something

Once you are *double-plus sure* your code is doing what you think it is doing, you need to get your neural network to actually learn whatever task you've given it. I assume your goal is to take a new component (network design, loss function, training method, etc.) and show it working on a big, difficult problem.

This is the part of research that I have most struggled with, and this is the synthesis of what I have learned:

Start small and work up to it

To get to your grand demonstration, the best way is to start with a task that is small and not difficult and to increase size and complexity. This is likely something we all already do, and is included for completeness. Smaller and simpler data means your training code completes more quickly, so you can iterate faster.

Consider using subsets of larger datasets to test tasks. A common trick is to use two classes of CIFAR-10 for initial testing, adding more classes, and eventually moving on to full ImageNet.

Try starting with shallow networks with large layers instead of deeper networks.

Use baselines and upper bounds

A neural network can be a universal function approximator, with emphasis on universal. Two useful implications of this:

1. With enough training and parameters, you should be able to find some neural network to over-fit your data.
2. The performance of *any* function on your task establishes an upper bound on the performance of some neural network on that task.

Deliberate overfitting

A good way to train a network when you don't know what network size and hyper-parameters you should use is to deliberately aim to overfit it. Pick a larger network than you imagine you need and experiment with hyperparameters until you observe very low loss on your training set but poorly in your held-out set. You can even reduce the size of your training set if necessary to find some network that overfits.

Once you have done that, you can apply your usual steps to reduce overfitting: reduce the network size, change the structure of your network, etc. You have a great starting point from which to set hyperparameters.

Upper-bound performance

Since neural networks can (in theory[3]) approximate any function, the performance of any function on your task can act as an upper bound on the performance of some neural network on the same task. The requirement here is that the loss is computed the same way.

This means that you can easily figure out what a "good" loss looks like using classic machine learning techniques: any kind of logistic/linear regression, boosted classifier, k-nearest-neighbors, etc. If you have a known closed-form solution, generate new parameterizations by replacing known parts with arbitrary ones.

> ***Example:*** *to lower-bound the performance of some learned function $f: R^4 \rightarrow R$ which was evaluated by the L2 distance of the output.*
>
> *I divided the input space into 2^{12} cells. For some training set, I grouped the output values by which cell the input corresponds to. The weighted sum over all cells of the variance in each cell is equal to the L2 loss of a function that partitions the input space and guesses the mean value for each cell. This was about 10^{-1}. This suggests that a network should be able to obtain an L2 loss of about10^{-1}on the training set.*
>
> *This also suggests that a good place to start looking for overfitting is a network that has more than 2^{12}parameters.*

Keep track of where you are on the bias-variance tradeoff

See these lecture notes from Cornell for an explanation. The Bias-variance tradeoff is a helpful way to conceptualize over- and under-fitting, and is the result of splitting the expected test error into three parts:

- *Noise*, the base rate of error inherent in the data. (Example: in MNIST some "1", "4" and "9" look like each other because of poor handwriting. Even a trained human couldn't consistently answer this.)
- *Bias*, the expected error from training your model on an infinite amount of data. This measures the inherent limitations of your model.
- *Variance*, the expected error from training your model on this particular dataset instead of an infinite amount of data sampled from the same distribution. This measures model overfitting to data.[4]

When training your model, you should log your loss on the training set and the validation set. Put these together to figure out where you are on the bias-variance tradeoff:

---

[3] Remember that in theory, theory and practice are the same; in practice they are not.

[4]This is confounded by the difference in distributions in the training and validation sets, so draw them from the same distribution.

Training Set Error	Close to noise	Close to noise	Higher than noise	Higher than noise
Validation Set Error	Close to training set error	Higher than training set error	Close to training set error	Higher than training set error
	You Win!	Overfitting. High Variance, low Bias regime.	Underfitting. High Bias, low Variance regime.	Train more, or change your model entirely.

Over- and under-fitting in a converged model can be fixed by:

- Overfitting
    - Reduce model “capacity” by making the model narrower and/or shallower.[5]
    - Data augmentation (discussed below)
    - Regularization (discussed below)
- Underfitting
    - Increase model “capacity” by making the model wider and/or deeper,
    - Increase the learning rate
        - If you see a marked improvement in performance at convergence, you were previously stuck in a local minima.

Regularization is king

Once you have overfit a model to the network (see upper-bound performance), the remaining work is to find a way to generalize. This is when your development or held-out validation set comes handy: your objective is to improve performance on that without losing too much performance on the training set.

Andrew Ng’s course covers this in detail. Watch that! If you have more time, take a look at the Overfitting and Regularization lectures from CalTech by Yaser Abu-Mostafa.

To regularize a network, you have some options:

- Classic techniques
    - Weight decay (an L2 norm penalty on parameter size)
    - Dropout
    - Early stopping.
- Adding domain-specific features
    - Example: Sine non-linear layers on periodic data
    - Example: Wavelet features on natural image data
- Gradient management, particularly for networks with exploding or vanishing gradients.
    - Adding skip connections (as in ResNets)

[5] This is the classic advice, but this 18-page summary of the research on neural network generalization suggests the notion of “capacity” is a little more complicated.

        - Normalizing activations (BatchNorm)
- More exotic techniques:
    - Perturbing/resampling network activations
        - Example: the reparameterization technique in Variational Autoencoders.[6]
    - Adding activation noise, particularly structured noise.
    - Training to be robust to adversarial noise. (Note: I've seen a lot of ink spilled on this, but I don't know if it actually works. *Caveat Emptor*.)

Data augmentation and synthetic data

A great way to test this is to create a purely synthetic dataset that lets you control the output distribution. A classic example of this is, for classification, to sample from different gaussian models and map this through some transformation[7]. This way you can control the separability of your synthetic data by carefully selecting the models and the transformation. You can slowly make the task more difficult to learn by changing the transformations.

You can also use this approach to either augment or filter a dataset to control the complexity of the task. Augmentation strategies add copies of existing data with transformations like rotations, small translations, blurring, etc. Filtering strategies (temporarily) remove data points by some problem-dependent criteria: typically excluding some difficult subset of the input space.

Example: *I was trying to optimize some function* $f: R^2 \rightarrow R^2$*, subject to additional constraints, which is composed over itself* $n(x)$ *times until the input* x*reaches zero. (i.e. manipulate* $f(x)$ *to minimize*$n(x)$*such that* $f^{(n(x))}(x) = 0.$*)*

I could easily get the network to overfit, but not to generalize well. I found, by plotting $f(x)$ and $n(x)$ *as many networks trained, that the networks were failing to generalize correctly between regions A and B that had a sharp boundary between them.*

Augmenting data around the A/B boundary did not help. Eventually, I solved it by filtering *the dataset depending on the solver state. That is, we hid points from B until the network overfit to A, and then presented points from both A and B until it learned a boundary.*

There are many, many interesting dataset remixes published online. For MNIST, take a look at colorMNIST (color transformed), Synthetic Digits (with exotic backgrounds) as examples of this.

Levers you can pull

Ensure you keep a separate development/holdout/validation set of data while evaluating these. This will prevent you from overfitting to your test set.

[6] I just like this somewhat-related paper.
[7] Tensorflow's demo does something like this to generate sample data: https://playground.tensorflow.org/

Once you get the network and training in order, you should expect your models to train well in a reasonably-sized set of hyperparameters. If training is overly sensitive to one parameter, check that and associated parameters.

Try manipulating all these:

- The solver:
    - Adam optimizers are the default, and they work great for most tasks.
        - They have a momentum that you can tweak.
    - Try classic SGD, RMSprop, Adadelta, and others. (PyTorch documentation.)
- Learning rate: look for a learning rate that makes the loss function look smooth and generally decreasing on both the training and validation sets.
    - If your learning rate is too high your loss function may oscillate
    - If it is too low:
        - training may get stuck at a loss value well above your upper-bound estimate even across a large range of hyperparameters.
        - the loss value may decrease very smoothly.
- Batch sizes:
    - Varying the batch size changes the gradients
        - If the batch is too large, the variance of the gradient updates between batches can be too low and you can get stuck in local minima.
        - If the batch size is too low, the variance of the gradient updates between batches can be too large. Steps taken for one batch may not be in a direction that will eventually improve the network's learning.
    - Shuffle the dataset to ensure that the gradient updates for any one batch are not biased.
        - **Example:** If your classification dataset contains all points for class A followed by all points for class B, and you do not shuffle your minibatches, your model will get gradient updates for minibatches with only A or B, not both.
    - For a fair comparison between experiments with different minibatch sizes, ensure the loss function is the mean, not the sum, of individual losses.
- Non-linearities:
    - ReLU (or Swish) is a popular start!
    - Try Tanh or Sigmoid as well.
    - If your task involves periodic behavior: the SIREN paper shows how periodic activations (Sine, Cosine) work great to fit periodic data.
    - If your task involves dealing with very deep networks, there are exploding- and vanishing-gradient problems to watch out for. Consider normalizing activations (BatchNorm), adding skip connections (a la ResNets), or something more exotic like spectral normalization.
- Parameter initialization
    - The default initializers are generally good, but this is something to keep in mind.
- Data transformations
    - Normalization

        - Especially for visual tasks, mean subtraction is a venerable trick.
        - PCA/ZCA is very common in medical tasks, and is used as a general pre-processing step.
    - Task-specific transformations
        - **Example:** If your data is a combination of distances and angles, such as `(x, theta)`, you can selectively transform the angles to get `(x, sin(theta), cos(theta))`.
        - **Example:** Some 3D reinforcement learning tasks work better with quaternion data instead of classic rotation-matrix data. Check publications from your field for such comparisons.