Checklist for debugging neural networks

Tangible steps you can take to identify and fix issues with training, generalization, and optimization for machine learning models

Cecelia Shao Follow

Mar 15 · 10 min read

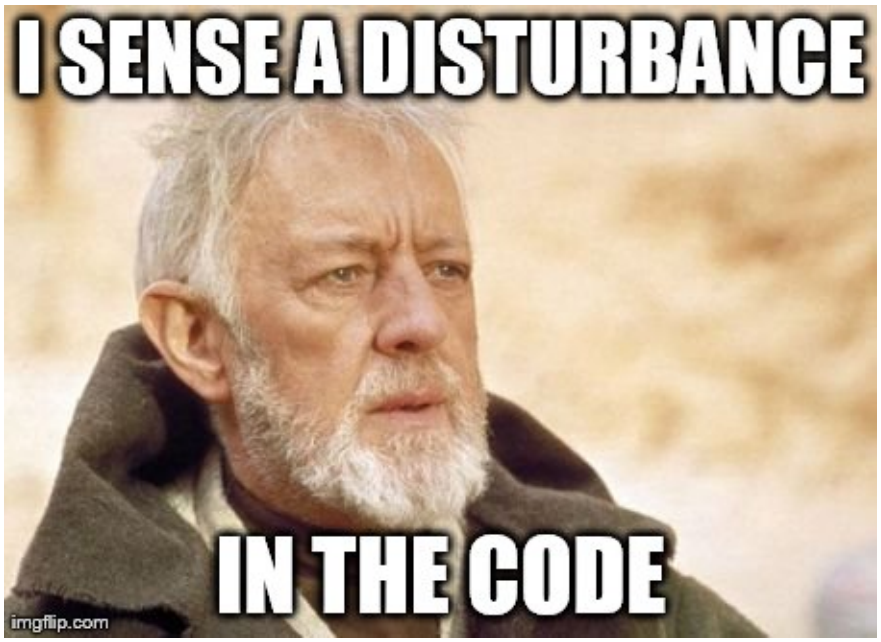
Photo by Glenn Carstens-Peters on Unsplash

Machine learning code can be notoriously difficult to debug with bugs that are expensive to chase. Even for simple, feedforward neural networks, you often have to make several decisions around network architecture , weight initialization, and network optimization—all of which can lead to insidious bugs in your machine learning code.

As Chase Roberts wrote in an excellent piece on ‘How to unit test machine learning code’, his frustrations stemmed from common traps like:

1. The code never crashes, raises an exception, or even slows down.
2. The network still trains and the loss will still go down.
3. The values converge after a few hours, but to really poor results

So what is to be done about it?



. . .

This article will provide a framework to help you debug your neural networks:

1. **Start simple**
2. **Confirm your loss**
3. **Check intermediate outputs and connections**
4. **Diagnose parameters**
5. **Tracking your work**

Feel free to skip to a particular section or read through below! Please note: we do not cover data preprocessing or specific model algorithm selection. There are plenty of great resources for those topics online (for example, check out 'Choosing the Right Machine Learning Algorithm').

1. Start simple

A neural network that has a complex architecture with regularization and a learning rate scheduler will be harder to debug than a simple network. We're kind of cheating with this first point since it's *not really* related to debugging a network you've already built, but it's still an important recommendation!

Start simple by:

- Building a simpler model first
- Train your model on a single data point

Build a simpler model first

To start off, build a small network with a single hidden layer and verify that everything is working correctly. Then gradually add model complexity while checking that each aspect of your model's structure (additional layer, parameter, etc..) works before moving on.

Train your model on a single data point

As a quick sanity check, you can use one or two training data points to confirm whether your model is able to overfit. The neural network should immediately overfit with a training accuracy of 100% and a validation accuracy that's commensurate to your model randomly guessing. If your model is unable to overfit just those data points, then either it's too small or there is a bug.

Even when you've verified that your model is working, try train for a single (or a few) epochs before progressing.

2. Confirm your loss

Your model's loss is the primary way to evaluate your model's performance and it's what the model is evaluating to set important

parameters, so you want to make sure that:

- The loss is appropriate for the task (using categorial cross-entropy loss for multi-classification problems or using focal loss to address class imbalance)
- Your loss functions are being measured on the correct scale. If you are using more than one type of loss in your network such as MSE, adversarial, L1, feature loss, then make sure all losses are scaled properly to be of same order

It's also important to pay attention to your initial loss. Check to see if that initial loss is close to your expected loss if your model started by guessing randomly. In the Stanford CS231n coursework, Andrej Karpathy suggests the following:

> *Look for correct loss at chance performance. Make sure you're getting the loss you expect when you initialize with small parameters. It's best to first check the data loss alone (so set regularization strength to zero). For example, for CIFAR-10 with a Softmax classifier we would expect the initial loss to be 2.302, because we expect a diffuse probability of 0.1 for each class (since there are 10 classes), and Softmax loss is the negative log probability of the correct class so: -ln(0.1) = 2.302.*

For a binary example, you would simply do a similar calculation for each of your classes. Let's say your data is 20% 0's and 80% 1's. Your expected initial loss would work out to $-0.2\ln(0.5)-0.8\ln(0.5) = 0.693147$. If your initial loss is much bigger than 1, it could indicate that your neural network weights are not balanced properly (i.e. your initialization was poor) or your data is not normalized.

3. Check intermediate outputs and connections

To debug a neural network, it can often be useful to understand the dynamics inside a neural network and the role played by the individual intermediate layers and how the layers are connected. You may be running into errors around:

- Incorrect expressions for gradient updates

- Weight updates not being applied
- Vanishing or exploding gradients

If your gradient values are zero, it could mean that the learning rate might be too small in the optimizer or that you're encountering Error #1 above with incorrect expressions for the gradient updates.

Aside from looking at the absolute values of the gradient updates, make sure to monitor the magnitudes of activations, weights, and updates of each layer match. For example, the magnitude of the updates to the parameters (weights and biases) should be 1-e3.

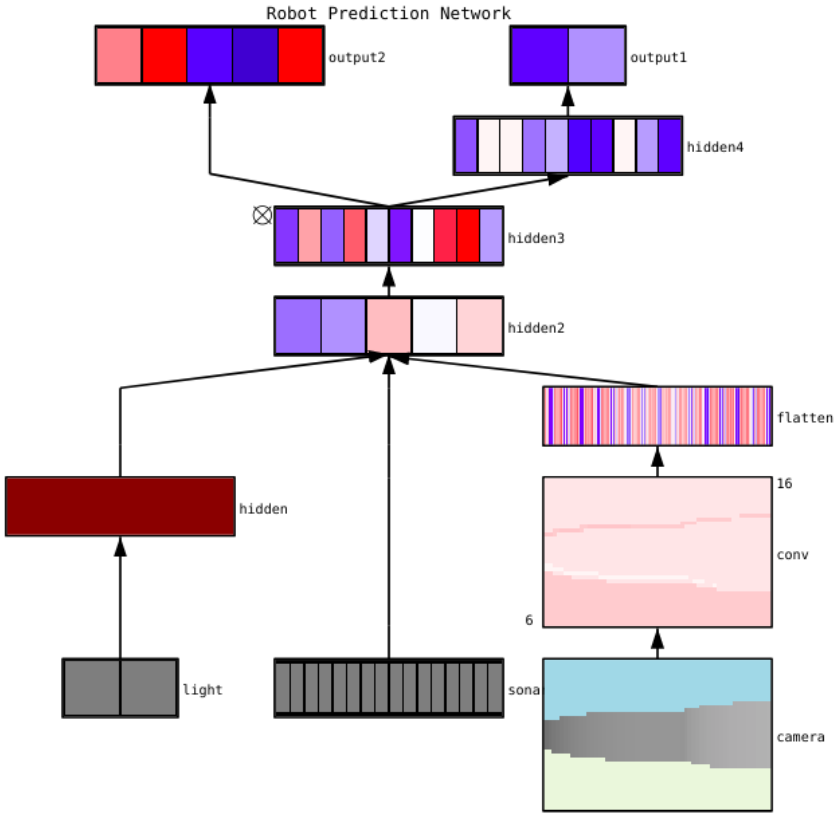
There is a phenomenon called the 'Dying ReLU' or 'vanishing gradient problem' where the ReLU neurons will output a zero after learning a large negative bias term for its weights. Those neurons will never activate on any datapoint again.

You can use gradient checking to check for these errors by approximating the gradient using a numerical approach. If it is close to the calculated gradients, then backpropagation was implemented correctly. To implement gradient checking, check out these great resources from CS231 here and here and Andrew Ng's specific lesson on the topic.

Faizan Shaikh writes about the three primary methods of visualizing your neural network:

- **Preliminary methods**—Simple methods which show us the overall structure of a trained model. These methods include printing out the shapes or filters of individual layers of neural network and the parameters in each layer.
- **Activation based methods**—In these methods, we decipher the activations of the individual neurons or a group of neurons to get an intuition of what they are doing
- **Gradient based methods—**These methods tend to manipulate the gradients that are formed from a forward and backward pass while training a model (includes saliency maps and class activation maps).

There are many useful tools for visualizing individual layers' activations and connections such as ConX and Tensorboard.



A sample dynamic, rendered visualization made with ConX

> *Working with image data? Erik Rippel has a great, colorful post on 'Visualizing parts of Convolutional Neural Networks using Keras and Cats'*

4. Diagnose parameters

Neural networks have large numbers of parameters that interact with each other, making optimization hard. Please note, this is an area of active research so the suggestions below are simply starting points.

- **Batch size** (technically called mini-batch) —You want the batch size to be large enough to have accurate estimates of the error gradient, but small enough that stochastic gradient descent (SGD) can regularize your network. Small batch sizes will result in a learning process that converges quickly at the cost of noise

in the training process and *might* lead to optimization difficulties. The paper 'On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima' describes how:

> *[It] has been observed in practice that when using a larger batch there is a degradation in the quality of the model, as measured by its ability to generalize. We investigate the cause for this generalization drop in the large-batch regime and present numerical evidence that supports the view that l****arge-batch methods tend to converge to sharp minimizers of the training and testing functions—and as is well known, sharp minima lead to poorer generalization****. In contrast, small-batch methods consistently converge to flat minimizers, and our experiments support a commonly held view that this is due to the inherent noise in the gradient estimation.*

- **Learning rate** —A learning rate that is too low will lead to slow convergence or the risk of getting stuck in a local minima, while a learning rate that too large will cause the optimization to diverge, because you risk jumping across a deeper, but narrower part of the loss function. Consider incorporating learning rate scheduling to decrease the learning rate as training progresses. The CS231n course has a great section on different techniques to implement annealing learning rates.

> *Machine learning frameworks such as Keras, Tensorflow, PyTorch, MXNet now all have documentation or examples around using learning rate schedulers/decay:*
>
> *Keras—https://keras.io/callbacks/#learningratescheduler*
>
> *Tensorflow—https://www.tensorflow.org/api_docs/python/tf/train/exponential_decay*
>
> *PyTorch—https://pytorch.org/docs/stable/_modules/torch/optim/lr_scheduler.html*

> *MXNet— https://mxnet.incubator.apache.org/versions/master/tutorials/gluon/learning_rate_schedules.html*

- **Gradient clipping**—This will clip parameters' gradients during backpropagation by a maximum value or maximum norm. Useful for addressing any exploding gradients that you might encounter in Step #3 above
- **Batch normalization** —Batch normalization is used to normalize the inputs of each layer, in order to fight the internal covariate shift problem. *Make sure to read the point below on Dropout if you're using Dropout and Batch Norma together.*

> *This article from Dishank Bansal 'Pitfalls of Batch Norm in TensorFlow and Sanity Checks for Training Networks' is a great resource for common errors with batch normalization.*

- **Stochastic Gradient Descent (SGD)**— There are several flavors of SGD that use momentum, adaptive learning rates, and Nesterov updates with no clear winner for both training performance and generalization(See Sebastian Ruder's excellent 'An overview of gradient descent optimization algorithms' and this interesting experiment 'SGD > Adam?') A recommended starting point is Adam or plain SGD with Nesterov momentum.
- **Regularization** —Regularization is crucial for building a generalizable model since it adds a penalty for model complexity or extreme parameter values. It significantly reduces the variance of the model, without substantial increase in its bias. As described in the CS231n course:

> *It is often the case that a loss function is a sum of the data loss and the regularization loss (e.g. L2 penalty on weights). One danger to be aware of is that the regularization loss may overwhelm the data loss, in which case the gradients will be primarily coming from the regularization term (which usually has a much simpler gradient expression). This can mask an incorrect implementation of the data loss gradient.*

To audit this, you should turn off regularization and check your data loss gradient independently.

- **Dropout**—Dropout is another technique to regularize your network to prevent overfitting. While training, dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise. As a result, the network has to use a different subset of parameters per training batch, which reduces the changes of specific parameters becoming dominant over others.
- The important note here is: if you're using both dropout and batch normalization (batch norm) together, be cautious of the order of these operations or even of using them together. This is still an active area of research, but you can see the latest discussions:

> ***From Stackoverflow user*** `MiloMinderBinder` *: "Dropout is meant to block information from certain neurons completely to make sure the neurons do not co-adapt. So, the batch normalization has to be after dropout otherwise you are passing information through normalization statistics."*

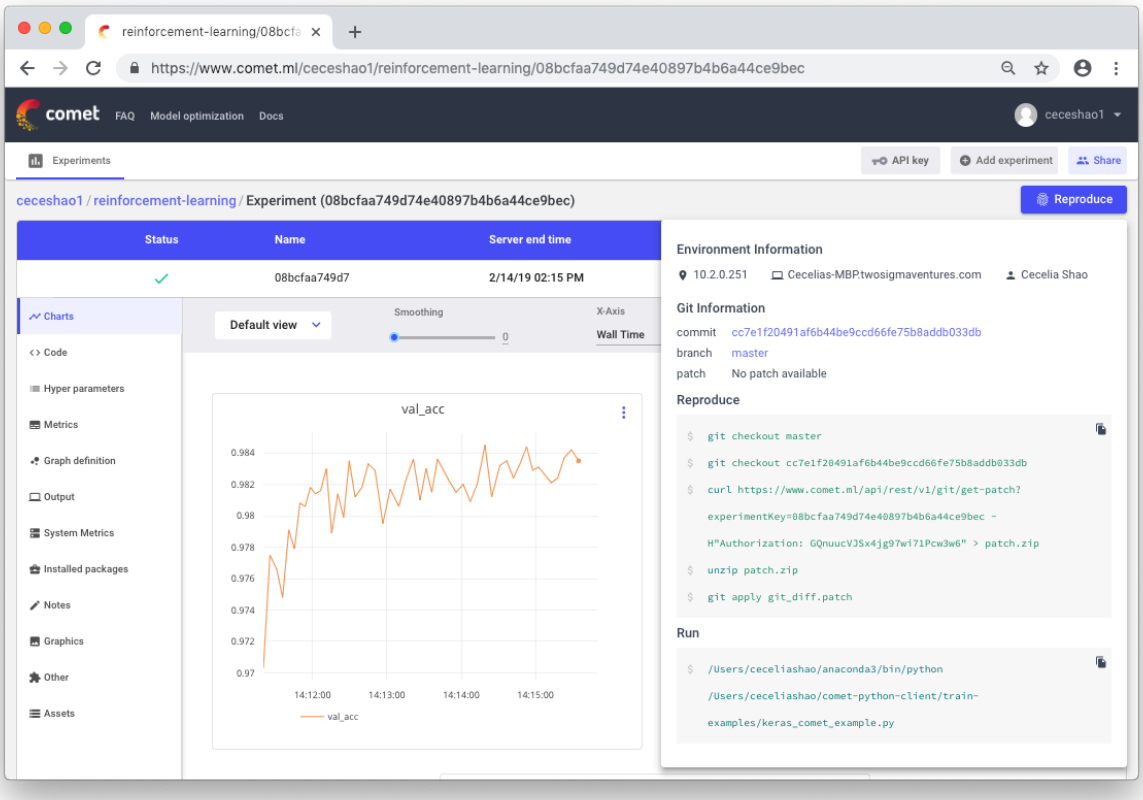
> ***From arXiv****:* Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift (*Xiang Li, Shuo Chen, Xiaolin Hu, Jian Yang)—"Theoretically, we find that Dropout would shift the variance of a specific neural unit when we transfer the state of that network from train to test. However, BN would maintain its statistical variance, which is accumulated from the entire learning procedure, in the test phase. The inconsistency of that variance (we name this scheme as "variance shift") causes the unstable numerical behavior in inference that leads to more erroneous predictions finally, when applying Dropout before BN."*

5. Track your work

It's easy to overlook the importance of documenting your experiments until you forget which learning rate or class weights you used. With better tracking, you can easily review and reproduce previous experiments to reduce duplicating work (aka running into the same errors).

However, manually documenting information can be difficult to do and scale for multiple experiments. Tools like Comet.ml can help automatically track datasets, code changes, experimentation history and production models—including key pieces of information about your model like hyperparameters, model performance metrics, and environment details.

Your neural network can be very sensitive to slight changes in both data, parameters, and even package versions—leading to drops in model performance that can build up. Tracking your work is the first step you can take to begin standardizing your environment and modeling workflow.



Check out model performance metrics and retrieve the code used to train the model from within Comet.ml. There's an example of Comet's automatic experiment tracking here.

• • •

Quick Recap

We hope this post serves a solid starting point for debugging your neural network. To summarize the highlights, you should:

1. **Start simple**—build a simpler model first and test by training on a few data points
2. **Confirm your loss**—check to see if you're using the correct loss and review your initial loss
3. **Check intermediate outputs and connections**—use gradient checking and visualization to check if your layers are properly connected and that your gradients are updating as expected
4. **Diagnose parameters**—from SGD to learning rates, identifying the right combination (or figuring out the wrong ones) 😅
5. **Tracking your work**— as a baseline, tracking your experimentation process and key modeling artifacts

Found this post useful? Think it's missing something? Comment below with your feedback and questions! 👩🏻‍🎓

> *Follow the discussion on HackerNews!*