Slav Ivanov Follow
Entrepreneur / Hacker
Jul 25 · 9 min read

# 37 Reasons why your Neural Network is not working

The network had been training for the last 12 hours. It all looked good: the gradients were flowing and the loss was decreasing. But then came the predictions: all zeroes, all background, nothing detected. “What did I do wrong?”—I asked my computer, who didn’t answer.

Where do you start checking if your model is outputting garbage (for example predicting the mean of all outputs, or it has really poor accuracy)?

A network might not be training for a number of reasons. Over the course of many debugging sessions, I would often find myself doing the same checks. I’ve compiled my experience along with the best ideas around in this handy list. I hope they would be of use to you, too.

## Table of Contents



## 0. How to use this guide?

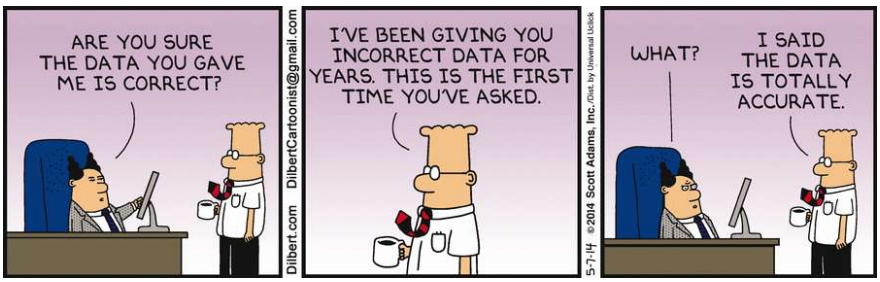
A lot of things can go wrong. But some of them are more likely to be broken than others. I usually start with this short list as an emergency first response:

1. Start with a simple model that is known to work for this type of data (for example, VGG for images). Use a standard loss if possible.
2. Turn off all bells and whistles, e.g. regularization and data augmentation.
3. If finetuning a model, double check the preprocessing, for it should be the same as the original model's training.
4. Verify that the input data is correct.
5. Start with a really small dataset (2–20 samples). Overfit on it and gradually add more data.
6. Start gradually adding back all the pieces that were omitted: augmentation/regularization, custom loss functions, try more complex models.

If the steps above don't do it, start going down the following big list and verify things one by one.

. . .

# I. Dataset issues




Source: http://dilbert.com/strip/2014-05-07

## 1. Check your input data

Check if the input data you are feeding the network makes sense. For example, I've more than once mixed the width and the height of an image. Sometimes, I would feed all zeroes by mistake. Or I would use the same batch over and over. So print/display a couple of batches of input and target output and make sure they are OK.

## 2. Try random input

Try passing random numbers instead of actual data and see if the error behaves the same way. If it does, it's a sure sign that your net is turning data into garbage at some point. Try debugging layer by layer /op by op/ and see where things go wrong.

## 3. Check the data loader

Your data might be fine but the code that passes the input to the net might be broken. Print the input of the first layer before any operations and check it.

## 4. Make sure input is connected to output

Check if a few input samples have the correct labels. Also make sure shuffling input samples works the same way for output labels.

## 5. Is the relationship between input and output too random?

Maybe the non-random part of the relationship between the input and output is too small compared to the random part (one could argue that stock prices are like this). I.e. the input are not sufficiently related to the output. There isn't an universal way to detect this as it depends on the nature of the data.

## 6. Is there too much noise in the dataset?

This happened to me once when I scraped an image dataset off a food site. There were so many bad labels that the network couldn't learn. Check a bunch of input samples manually and see if labels seem off.

The cutoff point is up for debate, as this paper got above 50% accuracy on MNIST using 50% corrupted labels.

## 7. Shuffle the dataset

If your dataset hasn't been shuffled and has a particular order to it (ordered by label) this could negatively impact the learning. Shuffle your dataset to avoid this. Make sure you are shuffling input and labels together.

## 8. Reduce class imbalance

Are there a 1000 class A images for every class B image? Then you might need to balance your loss function or try other class imbalance approaches.

## 9. Do you have enough training examples?

If you are training a net from scratch (i.e. not finetuning), you probably need lots of data. For image classification, people say you need a 1000 images per class or more.

### 10. Make sure your batches don't contain a single label

This can happen in a sorted dataset (i.e. the first 10k samples contain the same class). Easily fixable by shuffling the dataset.

### 11. Reduce batch size

This paper points out that having a very large batch can reduce the generalization ability of the model.

### Addition 1. Use standard dataset (e.g. mnist, cifar10)

Thanks to @hengcherkeng for this one:

> *When testing new network architecture or writing a new piece of code, use the standard datasets first, instead of your own data. This is because there are many reference results for these datasets and they are proved to be 'solvable'. There will be no issues of label noise, train/test distribution difference , too much difficulty in dataset, etc.*

. . .

## II. Data Normalization/Augmentation


## 12. Standardize the features

Did you standardize your input to have zero mean and unit variance?

## 13. Do you have too much data augmentation?

Augmentation has a regularizing effect. Too much of this combined with other forms of regularization (weight L2, dropout, etc.) can cause the net to underfit.

## 14. Check the preprocessing of your pretrained model

If you are using a pretrained model, make sure you are using the same normalization and preprocessing as the model was when training. For example, should an image pixel be in the range [0, 1], [-1, 1] or [0, 255]?

## 15. Check the preprocessing for train/validation/test set

CS231n points out a common pitfall:

> *"… any preprocessing statistics (e.g. the data mean) must only be computed on the training data, and then applied to the validation/test data. E.g. computing the mean and subtracting it from every image across the entire dataset and then splitting the data into train/val/test splits would be a mistake. "*

Also, check for different preprocessing in each sample or batch.

. . .

# III. Implementation issues


Credit: https://xkcd.com/1838/

## 16. Try solving a simpler version of the problem

This will help with finding where the issue is. For example, if the target output is an object class and coordinates, try limiting the prediction to object class only.

## 17. Look for correct loss “at chance”

Again from the excellent CS231n: *Initialize with small parameters, without regularization. For example, if we have 10 classes, at chance means we will get the correct class 10% of the time, and the Softmax loss is the negative log probability of the correct class so: -ln(0.1) = 2.302.*

After this, try increasing the regularization strength which should increase the loss.

## 18. Check your loss function

If you implemented your own loss function, check it for bugs and add unit tests. Often, my loss would be slightly incorrect and hurt the performance of the network in a subtle way.

## 19. Verify loss input

If you are using a loss function provided by your framework, make sure you are passing to it what it expects. For example, in PyTorch I would mix up the NLLLoss and CrossEntropyLoss as the former requires a softmax input and the latter doesn't.

## 20. Adjust loss weights

If your loss is composed of several smaller loss functions, make sure their magnitude relative to each is correct. This might involve testing different combinations of loss weights.

## 21. Monitor other metrics

Sometimes the loss is not the best predictor of whether your network is training properly. If you can, use other metrics like accuracy.

## 22. Test any custom layers

Did you implement any of the layers in the network yourself? Check and double-check to make sure they are working as intended.

## 23. Check for "frozen" layers or variables

Check if you unintentionally disabled gradient updates for some layers/variables that should be learnable.

## 24. Increase network size

Maybe the expressive power of your network is not enough to capture the target function. Try adding more layers or more hidden units in fully connected layers.

## 25. Check for hidden dimension errors

If your input looks like (k, H, W) = (64, 64, 64) it's easy to miss errors related to wrong dimensions. Use weird numbers for input dimensions (for example, different prime numbers for each dimension) and check how they propagate through the network.

## 26. Explore Gradient checking

If you implemented Gradient Descent by hand, gradient checking makes sure that your backpropagation works like it should. More info: 1 2 3.

. . .

# IV. Training issues


Credit: http://carlvondrick.com/ihog/

## 27. Solve for a really small dataset

**Overfit a small subset of the data and make sure it works.** For example, train with just 1 or 2 examples and see if your network can learn to differentiate these. Move on to more samples per class.

## 28. Check weights initialization

If unsure, use Xavier or He initialization. Also, your initialization might be leading you to a bad local minimum, so try a different initialization and see if it helps.

## 29. Change your hyperparameters

Maybe you using a particularly bad set of hyperparameters. If feasible, try a grid search.

## 30. Reduce regularization

Too much regularization can cause the network to underfit badly. Reduce regularization such as dropout, batch norm, weight/bias L2

regularization, etc. In the excellent "Practical Deep Learning for coders" course, Jeremy Howard advises getting rid of underfitting first. This means you overfit the training data sufficiently, and only then addressing overfitting.

## 31. Give it time

Maybe your network needs more time to train before it starts making meaningful predictions. If your loss is steadily decreasing, let it train some more.

## 32. Switch from Train to Test mode

Some frameworks have layers like Batch Norm, Dropout, and other layers behave differently during training and testing. Switching to the appropriate mode might help your network to predict properly.

## 33. Visualize the training

- Monitor the activations, weights, and updates of each layer. Make sure their magnitudes match. For example, the magnitude of the updates to the parameters (weights and biases) should be 1-e3.
- Consider a visualization library like Tensorboard and Crayon. In a pinch, you can also print weights/biases/activations.
- Be on the lookout for layer activations with a mean much larger than 0. Try Batch Norm or ELUs.
- Deeplearning4j points out what to expect in histograms of weights and biases:

> *"For weights, these histograms should have an **approximately Gaussian (normal)** distribution, after some time. For biases, these histograms will generally start at 0, and will usually end up being **approximately Gaussian** (One exception to this is for LSTM). Keep an eye out for parameters that are diverging to +/- infinity. Keep an eye out for biases that become very large. This can sometimes occur in the output layer for classification if the distribution of classes is very imbalanced."*

- Check layer updates, they should have a Gaussian distribution.

## 34. Try a different optimizer

Your choice of optimizer shouldn't prevent your network from training unless you have selected particularly bad hyperparameters. However, the proper optimizer for a task can be helpful in getting the most

training in the shortest amount of time. The paper which describes the algorithm you are using should specify the optimizer. If not, I tend to use Adam or plain SGD with momentum.

Check this excellent post by Sebastian Ruder to learn more about gradient descent optimizers.

## 35. Exploding / Vanishing gradients

- Check layer updates, as very large values can indicate exploding gradients. Gradient clipping may help.
- Check layer activations. From Deeplearning4j comes a great guideline: *"A good standard deviation for the activations is on the order of 0.5 to 2.0. Significantly outside of this range may indicate vanishing or exploding activations."*

## 36. Increase/Decrease Learning Rate

A low learning rate will cause your model to converge very slowly.

A high learning rate will quickly decrease the loss in the beginning but might have a hard time finding a good solution.

Play around with your current learning rate by multiplying it by 0.1 or 10.

## 37. Overcoming NaNs

Getting a NaN (Non-a-Number) is a much bigger issue when training RNNs (from what I hear). Some approaches to fix it:

- Decrease the learning rate, especially if you are getting NaNs in the first 100 iterations.
- NaNs can arise from division by zero or natural log of zero or negative number.
- Russell Stewart has great pointers on how to deal with NaNs.
- Try evaluating your network layer by layer and see where the NaNs appear.

. . .

Did I miss anything? Is anything wrong? Let me know by leaving a reply below.

> ***If you liked this article, please help others find it by clicking the little heart icon below. Thanks a lot!***



## Resources:

*http://cs231n.github.io/neural-networks-3/*
*http://russellsstewart.com/notes/0.html*
*https://stackoverflow.com/questions/41488279/neural-network-always-predicts-the-same-class*
*https://deeplearning4j.org/visualization*
*https://www.reddit.com/r/MachineLearning/comments/46b8dz/what_does_debugging_a_deep_net_look_like/*
*https://www.researchgate.net/post/why_the_prediction_or_the_output_of_neural_network_does_not_change_during_the_test_phase*
*http://book.caltech.edu/bookforum/showthread.php?t=4113*
*https://gab41.lab41.org/some-tips-for-debugging-deep-learning-3f69e56ea134*
*https://www.quora.com/How-do-I-debug-an-artificial-neural-network-algorithm*