On The Perils of Batch Norm

Apr 26, 2017

This post is written for deep learning practitioners, and assumes you know what batch norm is and how it works.

If you're new to batch norm, or want a refresher, a brief overview of batch norm can be found here.

* * *

Let's talk about batch norm. To be more specific, let's talk about why I'm starting to hate batch norm.

One day, I was training a neural network with reinforcement learning. I was trying to reproduce the results of a paper, and was having lots of trouble. (Par for the course in RL.) The first author recommended I add batch norm if I wasn't using it already, because it was key to solving some of the environments. I did so, but it still didn't work.

A few days later, I found out that when running my policy in the environment,

- I fed the current state in a batch of size 1.
- I ran the policy in train mode.

So I was normalizing my input to $\vec{0}$ all the time. Which sounds like a pretty obvious issue, but thanks to reinforcement learning's inherent randomness, it wasn't obvious my input was always $\vec{0}$.

I fixed it, and started getting the results I was supposed to get.

* * *

A few months later, an intern I was working with showed me a *fascinating* bug in his transfer learning experiments. He was using my code, which used TensorFlow's

MetaGraph tools. They let you take a model checkpoint and reconstruct the TF graph exactly the way it was at the time the checkpoint got saved. This makes it really, really easy to load an old model and add a few layers on top of it.

Unfortunately, MetaGraph ended up being our downfall. Turns out it doesn't play well with batch norm! Model checkpoints are saved while the model is training. Therefore, the model from the meta checkpoint is always stuck in train mode. Normally, that's fine. But batch norm turns it into a problem, because the train time code path differs from the test time code path. We couldn't do inference for the same reason as the previous bug - we'd always normalize the input to $\vec{0}$. (This is avoidable if you make the `is_training` flag a placeholder, but for structural reasons that wasn't doable for this project.)

I estimate we spent at least 6 hours tracking down the batch norm problem, and it ended with us concluding we needed to rerun all of the experiments we had done so far.

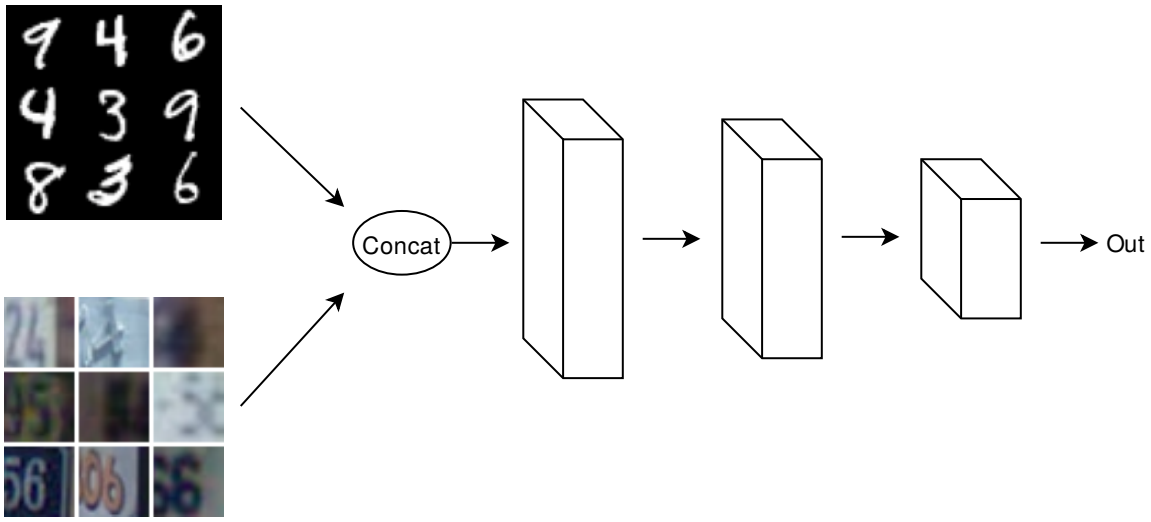
* * *

That same day (and I mean literally the same day), I was talking to my mentor about issues I was having in my own project. I had two implementations of a neural net. I was feeding the same input data every step. The networks had exactly the same loss, exactly the same hyperparameters, with exactly the same optimizer, trained with exactly the same number of GPUs, *and yet one version had 2% less classification accuracy, and consistently so.* It was clear that something had to be different between the two implementations, but what?

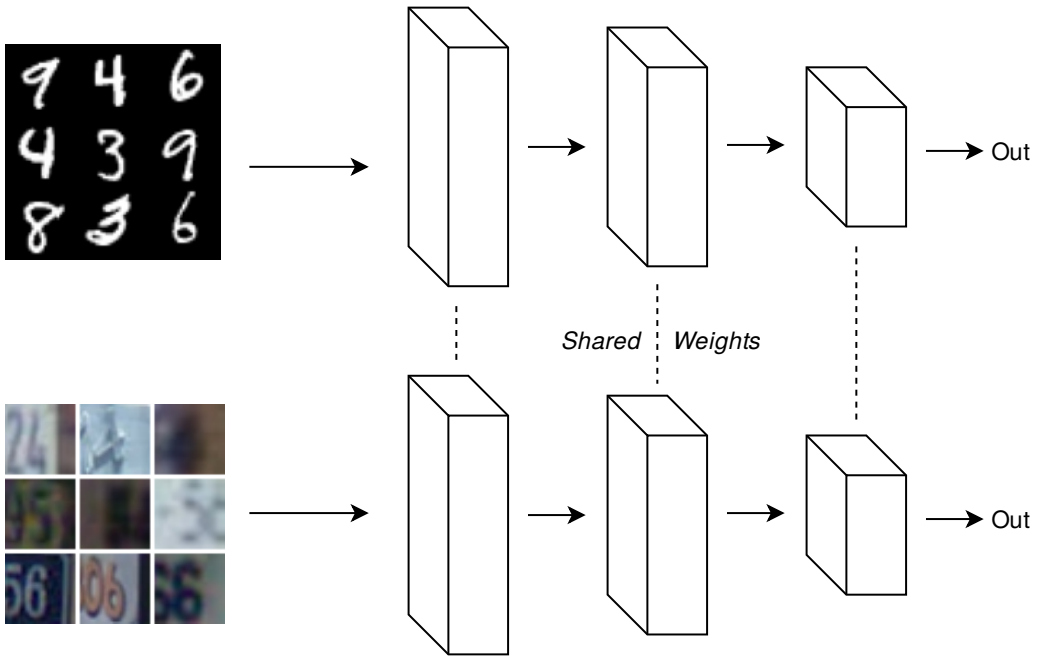
It was very lucky the MetaGraph issues got me thinking about batch norm. Who knows how long it would have taken me to figure it out otherwise?

Let's dig into this one a bit, because this problem was the inspiration for this blog post. I was training a model to classify two datasets. For the sake of an example, let's pretend I was classifying two digit datasets, MNIST and SVHN.

I had two implementations. In the first, I sample a batch of MNIST data and a batch of SVHN data, merge them into one big batch of twice the size, then feed it through the network.



In the second, I create two copies of the network with shared weights. One copy gets MNIST data, and the other copy gets SVHN data.

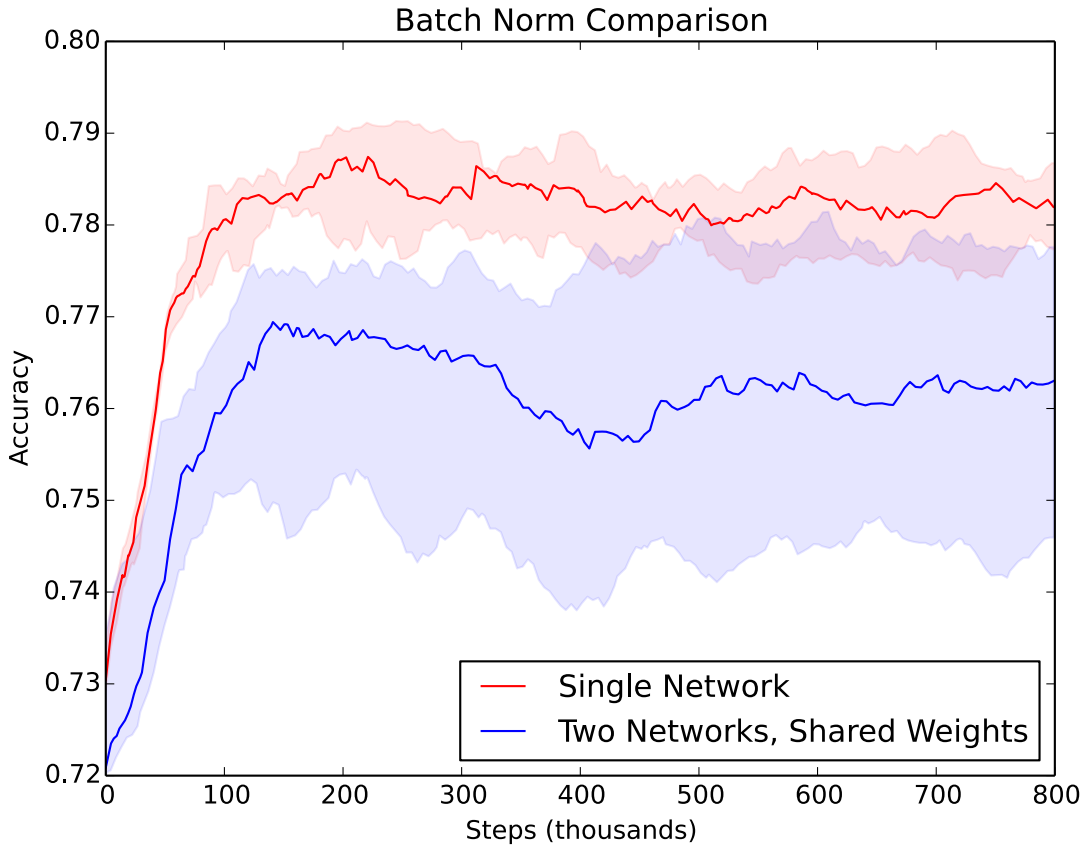


Note that in both cases, half the data is MNIST, half the data is SVHN, and thanks to shared weights, we have the same number of parameters and they're updated in the same way.

Naively, we'd expect the gradient to be the same in both versions of the model. And this is true - until batch norm comes into play. In the first approach, the model is trained on one batch of MNIST data and SVHN data. In the second approach, the model is trained on two batches, one of just MNIST data, and one of just SVHN data.

At training time, everything works fine. But you know how the two networks have shared weights? **The moving averages for dataset mean and variance were also shared, getting updated on both datasets.** In the second approach, the top network is trained with estimated mean and variance from MNIST data. The bottom network is traide with estimated mean and variance with SVHN data. But because the moving average was shared across the two networks, the moving average converged to the **average** of MNIST and SVHN data.

Thus, at test time, the scaling and shifting that we apply is different from the scaling and shifting the network expects. And when test-time normalization differs from train-time normalization, you get results like this.

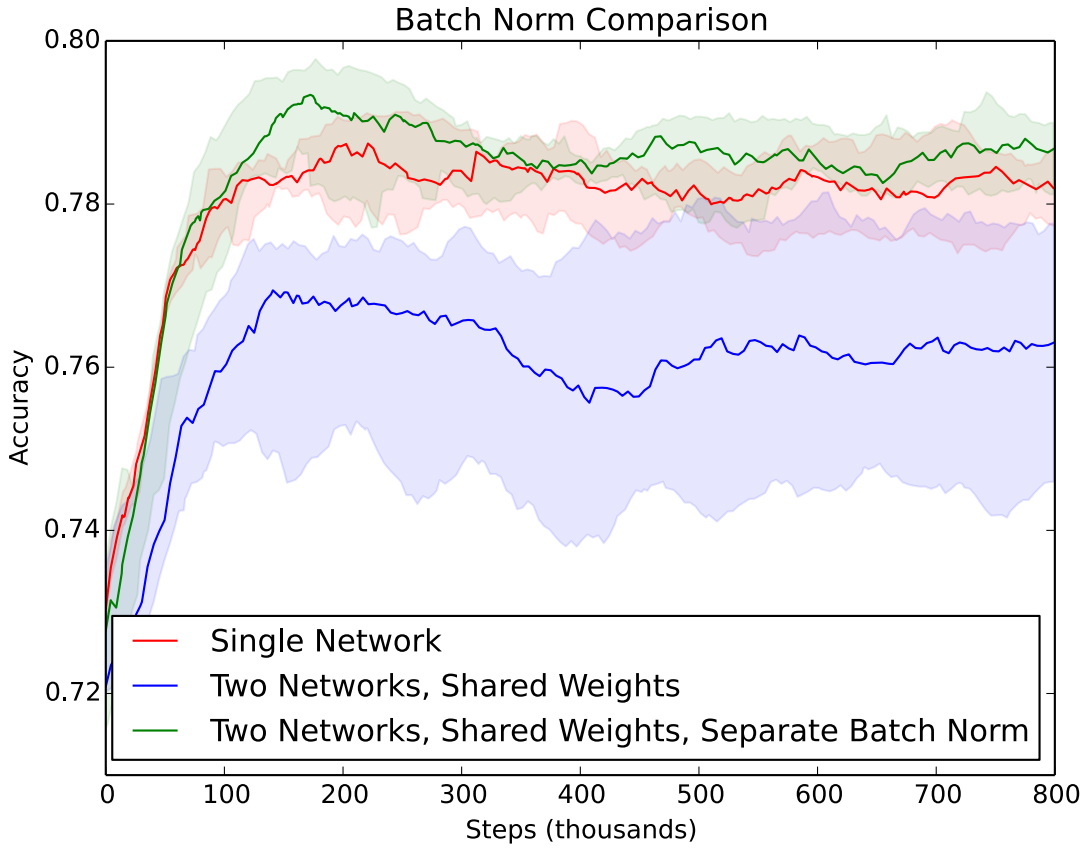


This plot is the top, median, and worst performance over 5 random seeds on one of my datasets. (This isn't with MNIST and SVHN anymore, it's with the two datasets I actually used.) When we do two networks with shared weights, not only was there a significant drop in performance, the variance of the output increased too.

Whenever individual minibatches aren't representative of your entire data distribution,

you can run into this problem. That means forgetting to randomize your input is especially bad with batch norm. It also plays a big role in GANs. The discriminator is usually trained on a mix of fake data and real data. If your discriminator uses batch norm, it's incorrect to alternate between batches of all fake or all real data. Each minibatch needs to be a 50-50 mix of both.

(Aside: in practice, we got the best results by using two networks with shared weights, with separate batch norm variables for each network. This was trickier to implement, but it did boost performance.)



Batch Norm: The Cause of, And Solution To, All of Life's Problems

You may have noticed a pattern in these stories.

I've thought about this quite a bit, and I've concluded that I'm never touching batch

norm again if I can get away with it.

My reasoning comes from the engineering side. Broadly, when code does the wrong thing, it happens for one of two reasons.

1. You make a mistake, and it's obvious once you see it. Something like a mistyped variable, or forgetting to call a function.
2. Your code has implicit assumptions about the behavior of other code it interacts with, and one of those assumptions breaks. These bugs are more pernicious, since it can take a while to figure out what assumption your code relied on.

Both mistakes are unavoidable. People make stupid mistakes, and people forget to check all the corner cases. However, the second class can be mitigated by favoring simpler solutions and reusing code that's known to work.

Alright. Now: batch norm. Batch norm changes models in two fundamental ways.

- At training time, the output for a single input x_i depends on the other x_j in the minibatch.
- At testing time, the model runs a different computation path, because now it normalizes with the moving average instead of the minibatch average.

Almost no other optimization trick has these properties. That makes it easier to write code that only works when inputs are minibatch independent, or only works when train time and test time do the same thing. The code's never been pushed that way. I mean, why would it? It's not like somebody's going to come up with a technique that breaks those assumptions, right?

Yes, you can treat batch norm as black box normalization magic, and it can even work out for a while. But in practice, the abstraction leaks, like all abstractions do, and batch norm's idiosyncrasies make it leak a lot more than it should.

Look, I just want things to work. So every time I run into Yet Another Batch Norm issue, I get disappointed. Every time I realize I have to make sure all my code is batch-norm proof, I get annoyed this is even a thing I have to do. Ever since the one network vs two network thing, I've been paranoid, because it is only by dumb luck that I implemented the same model twice. The difference is big enough that the whole project could have died.

So…Why Haven't People Ditched Batch Norm?

I'll admit I'm being unfair. Minibatch dependence is indefensible - no one is going to argue that it's a good quality for models to have. I've heard many people complain about batch norm, and for good reasons. Given all this, why is batch norm still so ubiquitous?

There's a famous letter in computer science: Dijkstra's Go To Statement Considered Harmful. In it, Dijkstra argues that the goto statement should be avoided, because it makes code harder to read, and any program that uses goto can be rewritten to avoid it.

I really, really wanted to title this post "Batch Norm Considered Harmful", but I couldn't justify it. Batch norm works too well.

Yes, it has issues, but when you do everything right, models train a lot faster. No contest. There's a reason the batch norm paper has over 1400 citations, as of this post.

There are alternatives to batch norm, but they have their own trade-offs. I've had some success with layer norm, and I hear it makes way more sense with RNNs. I've also heard it doesn't always work with convolutional layers.

Weight norm and cosine norm also sound interesting, and the weight norm paper said they were able to use it in a problem where batch norm didn't work. I haven't seen too much adoption of those methods though. Maybe it's a matter of time.

Layer norm, weight norm, and cosine norm all fix the contracts that batch norm breaks. If you're working on a new problem, and want to be brave, I'd try one of those instead of batch norm. Look, you'll need to do hyperparam tuning anyways. When tuned well, I'd expect the difference between various methods to be pretty low.

(If you want to be extra brave, you could try batch renormalization. Unfortunately it still has moving averages that are only used at test time. **EDIT (June 7, 2017): multiple people, including some of the paper authors, have told me this is incorrect. They're right, ignore this paragraph.**

In my case, I can't afford to switch from batch norm. Previous state of the art used batch norm, so I know it works, and I've already paid my dues of getting batch norm to work with my model. I imagine other researchers are in similar spots.

It's the Faustian bargain of deep learning. Faster training, in exchange for insanity. And I keep signing it. And so does everybody else.

Oh well. At least we didn't have to sacrifice any goats.

← How Computational Complexity Theory Crept Into A Cognitive Science Study
Read-through: Hyperparameter Optimization: A Spectral Approach →

f10w • 4 months ago

Hi Alex. Thank you for the post! Would you mind sharing how you implemented separate batch norm? Best.

• Reply • Share ›

Martin Görner • 5 months ago

Have you tried switching your activation functions from RELUs to Leaky-RELUs or SELUs ? I have heard people reporting that batch norm is no longer useful with those and intuitively it makes sense. Batch norm normalises logits before activation so as to distribute them over a useful part of the activation function. Sigmoid, tanh, RELUs all have useful and not-so-useful (i.e. flat) parts. L-RELUs or SELUs do not. But all this is handwaving conjecture so I was wondering if you had any relevant experimental results to share.

• Reply • Share ›

Christian Szegedy • 8 months ago

Batch-renormalization uses the moving averages both test and training time.

• Reply • Share ›

Erik Kruus • 8 months ago

... and in semi-supervised learning, class sampling may be extremely non-uniform. Ex. 1 label for ever 1000 unlabelled, but training using equal numbers of each. So mean and std devn represent essentially only the labelled data, while at inference time it's really more reasonable that data statistics look more like the unlabelled data, and may be far from the desired "global" values. (Don't know if any impls allow weighting according to inverse sample probability! At best, some may allow you to turn off maintaining the garbage stats). I've seen many Github codes that just ignore the issue. The batch renormalization paper talks about this issue but uses a different motivating example.

• Reply • Share ›

Sorta Insightful

Email: alexirpan [at] berkeley [dot] edu alexirpan