ANALYSIS

Stephen Mosher (2022)

Preliminaries

At first, I wanted to keep the networks across both of my implementations (TensorFlow and PyTorch) as close as possible. Of course, they have the same network architecture in terms of the number of layers and number of hidden units in each layer, and they both use ReLU activation functions in each layer. However, for whatever reason, I just couldn't get the network created in PyTorch to train properly with a cross-entropy loss function and softmax activation in the final layer. So, the PyTorch network uses a negative log-likelihood loss function instead, and a logsoftmax activation function in the final layer. Making these two changes helped get the PyTorch network into a semi-reasonable state.

Before getting to the questions raised by the challenge, I'll show the learning curves associated with each network's training process (yeah, I could have made the figures a little cleaner). In each of the two figures below, I plot the training loss, training accuracy, validation loss, and validation accuracy as a function of training epoch. Solid lines indicate training metrics, dashed lines indicate testing metrics.

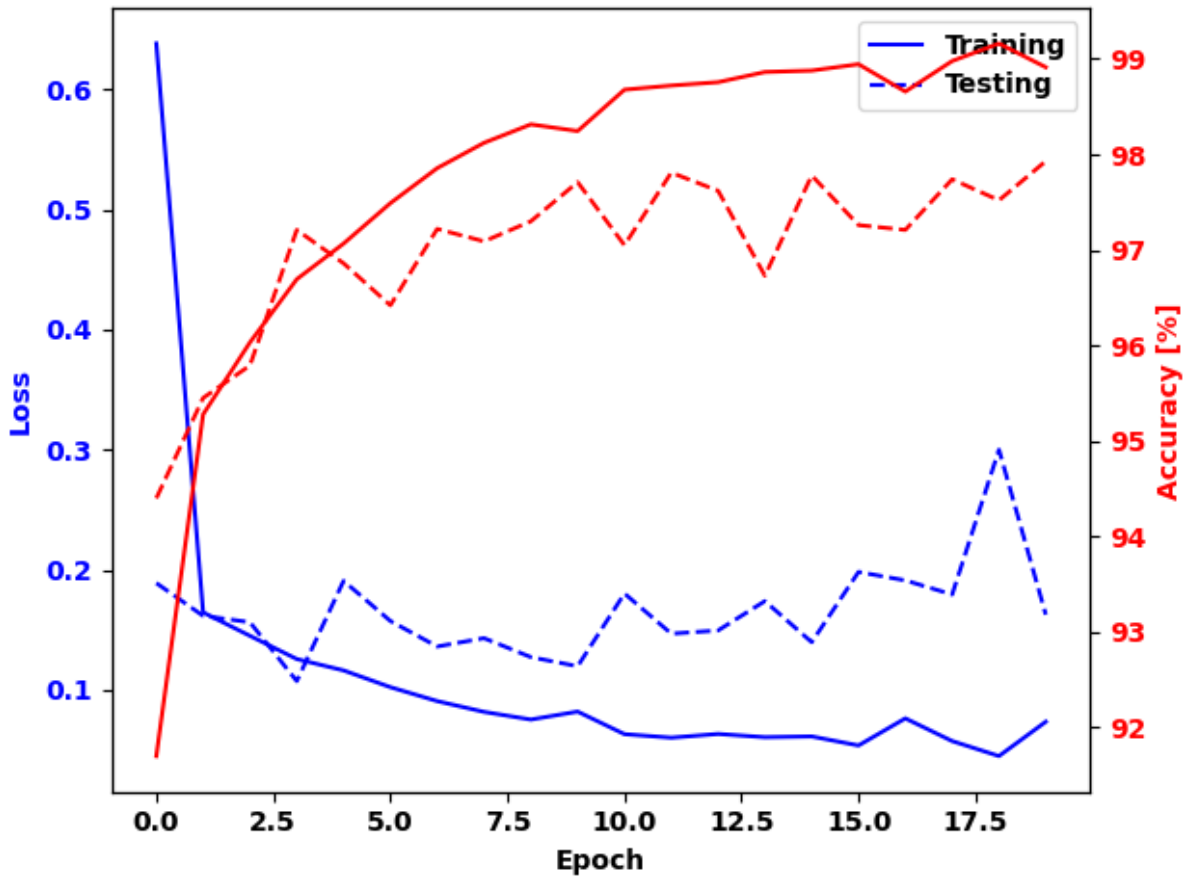

Figure 1: TensorFlow network's learning curves.

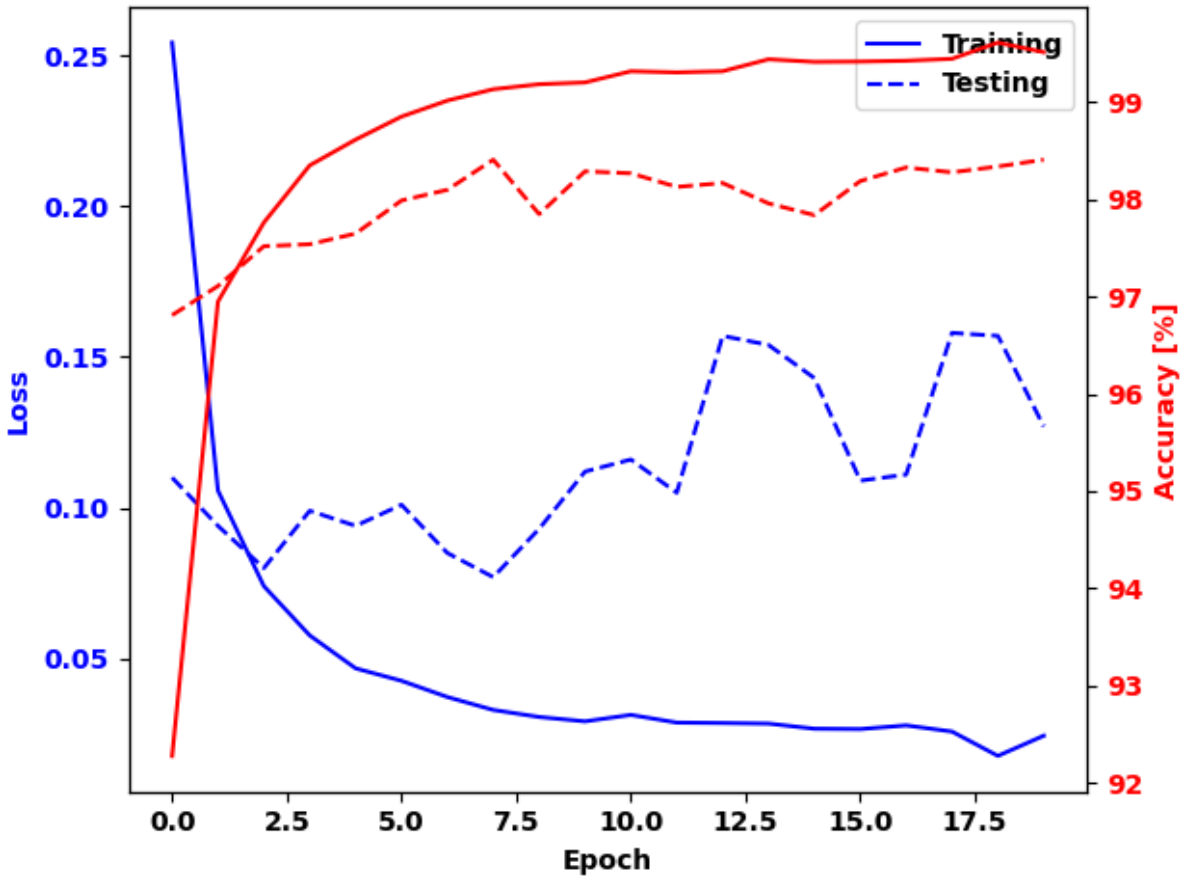

Figure 2: PyTorch network's learning curves.

Maybe it's because I'm more comfortable with TensorFlow, but the network trained with TensorFlow seems to perform a little bit better than the PyTorch network. They both overfit just a tad, but the TensorFlow network seems to overfit less than the PyTorch network. That being said, I might be splitting hairs because it's not like either network seems to perform ***that*** differently on the training and validation sets. All in all, given that I didn't do any data pre-processing, or fancy training, both of these networks seem to work just fine.

Next, I'll show the plots of the performance of the trained networks, both implementations, as function of sparsity, using both pruning methods (Weight-Pruning and Unit-Pruning). Then I'll get to the talking points raised in the challenge.

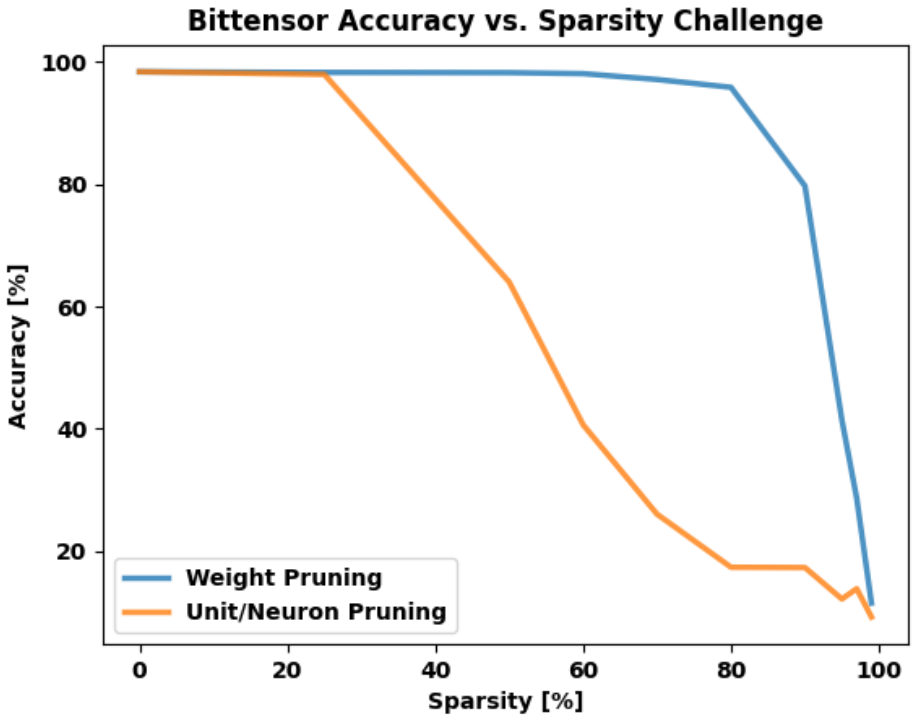

Figure 3: Performance vs. Sparsity for the PyTorch implementation

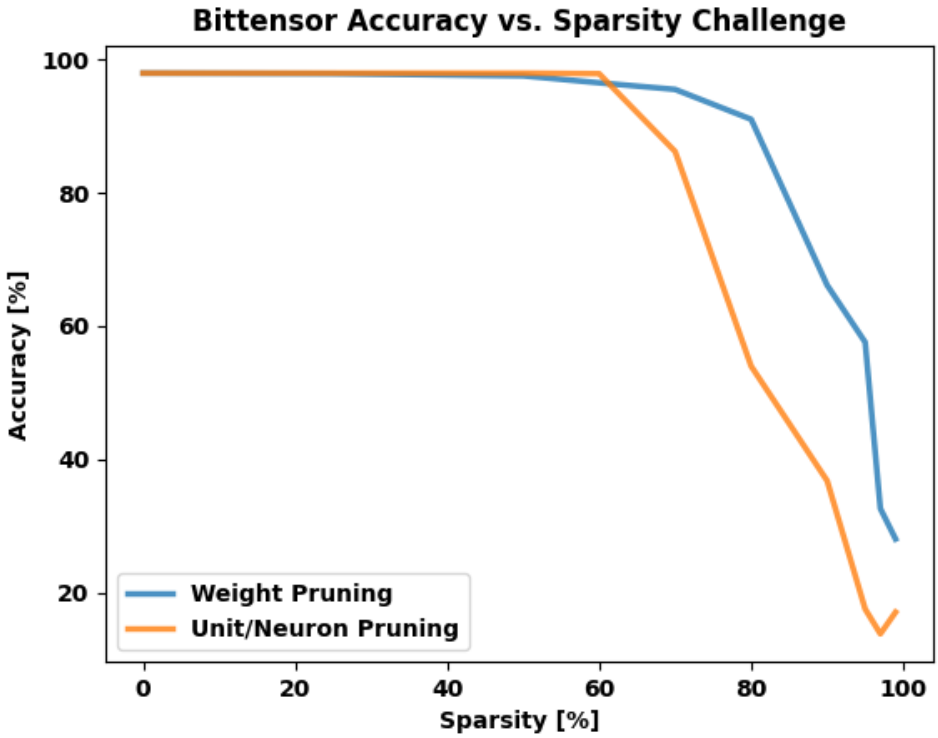

Figure 4: Performance vs. Sparsity for the TensorFlow implementation

Q1. What do you anticipate the degradation curve of sparsity vs. performance will be?

Truth be told, I wasn't too surprised by the shapes of the sparsity vs. performance curves. To be sure, before this challenge I wasn't aware of network pruning, but assessing the relative importance of weights or features in machine learning is a common theme. Going through this exercise made me think a lot about SVD, actually, another method that can be used to assess the importance of different factors and trim redundant/unnecessary data.

Q2. What interesting insights did you find?

After looking at all the cases I considered, it seems that a great deal of the network can be pruned without significantly affecting its performance, like 60% or up to 80% in one case! But maybe this is an argument for using a smaller network architecture in the first place?

One interesting application of pruning like this might be to modify someone else's trained network. Because at that point the network architecture isn't going to change, but perhaps you can prune it to make it more efficient to run on a more limited hardware/software platform…?

Q3. Do the curves differ? Why do you think that is/isn't?

Yes, the curves differ. Oddly enough, in the TensorFlow implementation they're much closer, and in the PyTorch implementation (assuming I don't have a bug somewhere) they differ more. In both cases, weight pruning seems to allow for a greater degree of pruning before the performance starts to degrade. This is only based on a few observations though, a more rigorous way to test this would be to keep track of the pruning results across several training runs, and across several different kinds of networks. Then you could compile statistics to obtain a more robust answer to this question.

Nonetheless, I think one reason weight-pruning seems to be able to go a little further before the performance starts to degrade is because it's less aggressive than unit-pruning.

Q4. Do you have any hypotheses as to why we are able to delete so much of the network without hurting performance?

Well, short answer, not exactly. As I sort-of touched upon in Q1, data compression is a common theme in, well, digital information, I guess. Consider something like Fourier analysis. We might look at a time-series sampled at 44,000 Hz and think that it's a very dense signal with a lot of information. And, in a way, it is. But one way to think about Fourier analysis, is that it offers up a way to find a sparse representation for such a signal. It might turn out after all, that the densely sampled time-series signal can be efficiently represented with just a handful of frequencies (or complex numbers, or whatever). And there are analogues of this in image processing as well. Using the Haar basis for 2D images, even if like 90% of an image is lost, it can be reconstructed quite well. I think the same sort of think is going on here for neural networks. We can build a neural network will millions of parameters, but really, there's no reason that most of the meaningful information in the network won't be contained in more than a small percentage of the network parameters…