7 PyTorch (Harsh)

While you were able to derive manual backpropogation rules for sigmoid and fully-connected layers, wouldn't it be nice if someone did that for lots of useful primatives and made it fast and easy to use for general computation? Meet automatic differentiation. Since we have high-dimensional inputs (images) and low-dimensional outputs (a scalar loss), it turns out **forward mode AD** is very efficient. Popular autodiff packages include pytorch (Facebook), tensorflow (Google), autograd (Boston-area academics). Autograd provides its own replacement for numpy operators and is a drop-in replacement for numpy, except you can ask for gradients now. The other two are able to act as shim layers for cuDNN, an implementation of auto-diff made by Nvidia for use on their GPUs. Since GPUs are able to perform large amounts of math much faster than CPUs, this makes the former two packages very popular for researchers who train large networks. Tensorflow asks you to build a computational graph using its API, and then is able to pass data through that graph. PyTorch builds a dynamic graph and allows you to mix autograd functions with normal python code much more smoothly, so it is currently more popular among CMU students.

For extra credit, we will use PyTorch as a framework. Many computer vision projects use neural networks as a basic building block, so familiarity with one of these frameworks is a good skill to develop. Here, we basically replicate and slightly expand our handwritten character recognition networks, but do it in PyTorch instead of doing it ourselves. Feel free to use any tutorial you like, but we like the offical one or this tutorial (in a jupyter notebook) or these slides (starting from number 35).

For this section, you're free to implement these however you like. All of the tasks required here are fairly small and don't require a GPU if you use small networks. Including 7.2.

7.1 Train a neural network in PyTorch

Q7.1.1 Code/Writeup [5 points] Re-write and re-train your fully-connected network on NIST36 in PyTorch. Plot training accuracy and loss over time.

Q7.1.2 Code/Writeup [2 points] Train a convolutional neural network with PyTorch on MNIST. Plot training accuracy and loss over time.

Q7.1.3 Code/Writeup [3 points] Train a convolutional neural network with PyTorch on the included NIST36 dataset.

Q7.1.4 Code/Writeup [10 points] Train a convolutional neural network with PyTorch on the EMNIST Balanced dataset and evaluate it on the findLetters bounded boxes from the images folder.

Q7.1.5 Code/Writeup Now that we have pyTorch to take care of gradients we can design really creative networks! Skip Connections and Residual Connections are some of the ideas that generally help. Do a small literature survey to understand these network designs [2] redesign your network to have:

1. Skip Connections [5 points]
2. Residual Connections [5 points]
3. A creative design that you want to try! [10 points]

After reading and running these experiments, in your write up try writing about:

1. The conceptual difference between residual connections and skip connections. Back it with equations and intuitions! [10 points]
2. For each network design explain your network design (Draw it!) and back it some reasoning, why you think this can work better? Feature visualization might help, think about gradients, use the ideas mentioned in the papers![5 + 5 + 5 points]

7.2 Fine Tuning

When training from scratch, a lot of epochs and data are often needed to learn anything meaningful. One way to avoid this is to instead initialize the weights more intelligently.

These days, it is most common to initialize a network with weights from another deep network that was trained for a different purpose. This is because, whether we are doing image classification, segmentation, recognition etc..., most real images share common properties. Simply copying the weights from the other network to yours gives your network a head start, so your network does not need to learn these common weights from scratch all over again. This is also referred to as fine tuning.

Q7.2.1 Code/Writeup [5 points] Fine-tune a single layer classifier using pytorch on the flowers 17 (or flowers 102!) dataset using squeezenet1_1, as well as an architecture you've designed yourself (*3 conv layers, followed 2 fc layers, it's standard slide 6*) and trained from scratch. How do they compare?

We include a script in `scripts/` to fetch the flowers dataset and extract it in a way that PyTorch ImageFolder can consume it, see an example, from `data/oxford-flowers17`. You should look at how SqueezeNet is defined, and just replace the classifier layer. There exists a pretty good example for fine-tuning in PyTorch.

References

[1] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. 2010. http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[3] P. J. Grother. Nist special database 19 – handprinted forms and characters database. https://www.nist.gov/srd/nist-special-database-19, 1995.