Neural Net Builder

Vikram Mullachery
mv333@nyu.edu

Advised by Prof. Dennis Shasha
shasha@cims.nyu.edu

Abstract

Neural Net Builder is a visual tool for designing and building neural networks. This tool intends to make the art of building neural nets less arcane to data scientists, data analysts, research scientists and non-programmers.

Machine learning and specifically deep neural networks have transformed our ability to predict and classify a variety of features and behaviors. However, this requires a well trained computer programmer and a deep learning expert to build neural networks. With this visual tool, we intend to make building a neural net easier, with much less code development time, and much more consistent and robust code for various machine learning libraries, namely Tensorflow and PyTorch.

The key use cases of this tool are: rapid, robust and error-free neural net construction, consistent reproduce-able neural network configurations, and a visual tool for code generation.

Related Work

Work in this field is minimal and the only notable one is by A. Karpathy, ConvNetJS. ConvNetJS is a machine learning framework in javascript; however it falls short of building a visual toolbox and generating code for robust production ready frameworks such as Tensorflow or PyTorch. Further, ConvNetJS is self-described as a toy two layer neural net for educational purposes.

A few other peripherally comparable works include Simbrain which attempts computer simulation of brain circuitry. This utility has a lot of the engineering concepts such as non-linearity and affine transformations built into it. However, Simbrain is more geared towards being a neuro-scientific education material than an utility for data analysts and statisticians.

Architecture

Architecturally speaking Neural Net Builder is composed of an AngularJS front end with a nodejs based server end for choreographing training and test runs. Inter-component communication utilizes HTTP calls and thus are highly decoupled and stateless. This allows for future scalability, robustness and high component-wise coherence.

Capabilities and Results

Current features include the availability of drag-and-droppable components of feed forward neural networks - Convolution, Pooling, FullyConnected and Softmax. Further, one can download a JSON based model metadata exchange Prototxt file. The target code is limited to PyTorch at this time, which can be downloaded for training and tweaking. Currently, the tool generates reliable PyTorch code (in Python using PyToch deep learning framework) consistently and reliably against MNIST dataset (of handwritten digits).

A short video demo of the utility is available here. The latest stable release of the product is available at the project github pages.

A few screenshots are attached here that showcase some of main features:

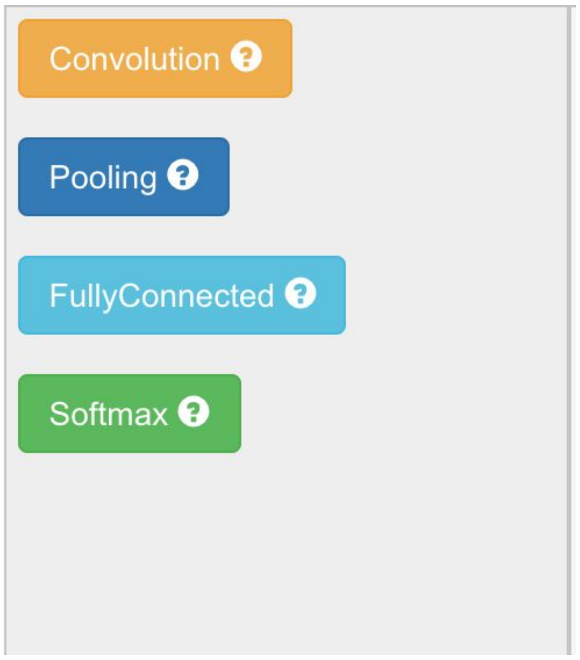


Figure 1: Feed-forward network components panel

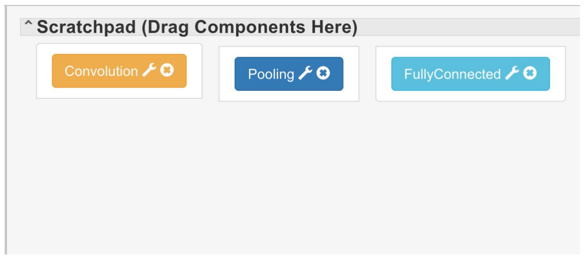


Figure 2: Main scratch pad panel

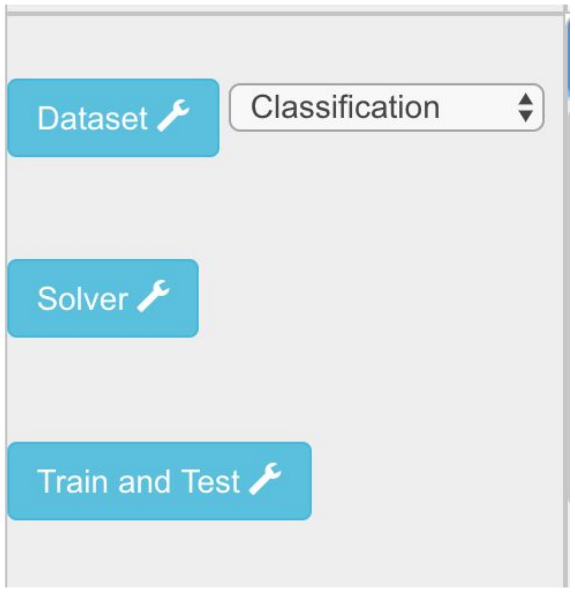


Figure 3: Neural network dataset and solver panel

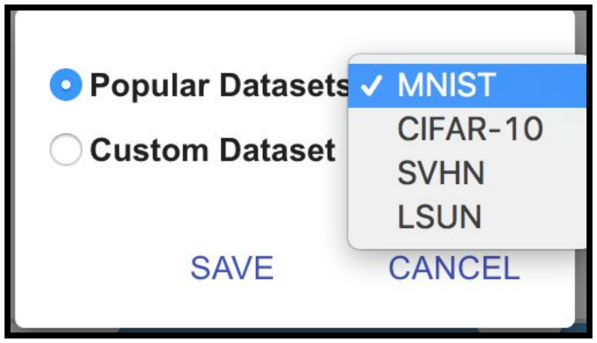
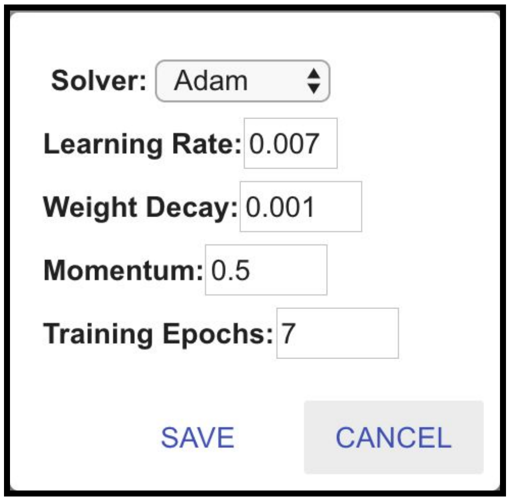


Figure 4: Dataset selection panel



Solver: Adam

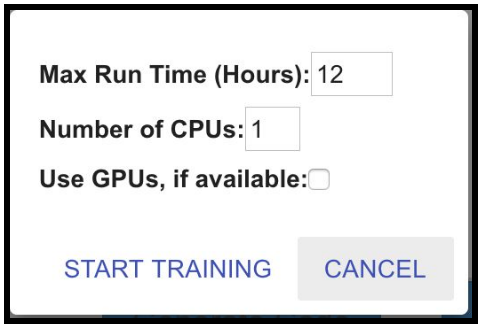
Learning Rate: 0.007

Weight Decay: 0.001

Momentum: 0.5

Training Epochs: 7

Figure 5: Solver selection



Max Run Time (Hours): 12

Number of CPUs: 1

Use GPUs, if available: ☐

Figure 6: One-click train and test

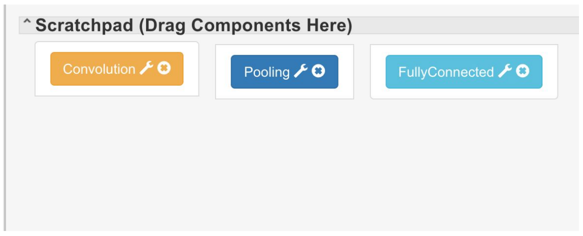
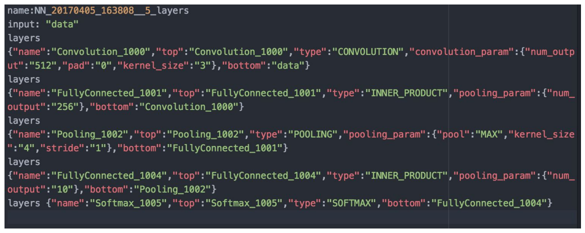
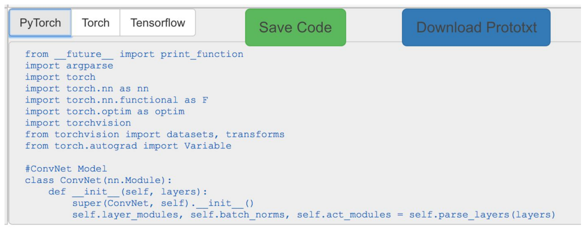


Figure 7: A simple feed-forward neural network with convolution for MNIST digit classification



```
name:NN_20170405_163808__5_layers
input: "data"
layers
{"name":"Convolution_1000","top":"Convolution_1000","type":"CONVOLUTION","convolution_param":{"num_outp
ut":"512","pad":"0","kernel_size":"3"},"bottom":"data"}
layers
{"name":"FullyConnected_1001","top":"FullyConnected_1001","type":"INNER_PRODUCT","pooling_param":{"num_
output":"256"},"bottom":"Convolution_1000"}
layers
{"name":"Pooling_1002","top":"Pooling_1002","type":"POOLING","pooling_param":{"pool":"MAX","kernel_size
":"4","stride":"1"},"bottom":"FullyConnected_1001"}
layers
{"name":"FullyConnected_1004","top":"FullyConnected_1004","type":"INNER_PRODUCT","pooling_param":{"num_
output":"10"},"bottom":"Pooling_1002"}
layers {"name":"Softmax_1005","top":"Softmax_1005","type":"SOFTMAX","bottom":"FullyConnected_1004"}
```

Figure 8: Generated Prototxt for MNIST digit classifier



```
from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
from torch.autograd import Variable

#ConvNet Model
class ConvNet(nn.Module):
    def __init__(self, layers):
        super(ConvNet, self).__init__()
        self.layer_modules, self.batch_norms, self.act_modules = self.parse_layers(layers)
```

Figure 9: Code panel (with generated code)

Future Work

We acknowledge that though a lot has been achieved in this short term, there is much more that we would like to incorporate into this tool.

A few feature additions of popular interest include: support for recurrent neural net (RNN), and ResNets, ability to graph components rather than the implicit sequential arrangement, support for predefined model templates akin to Caffe Model Zoo.

Finally, we plan to continue our wide set of user testing, with the assistance of students who have expressed interest in this effort. Given the already popular student demand, we are planning on incorporating this into a few of the classrooms that utilize neural networks.