Weekly Report(May 7 - May 13)

Liu Junnan

Abstract

This week I went on cs231n course and started to do assignment 2.

1 Work Done

1.1 Convolutional Network

Convolutional neural networks play a critical role in computer vision domain, because they improve the performance of almost all the CV tasks compared to traditional "shallow" machine learning methods. A typical convnet consists of a series of repeated conv-ReLU layers and fully connected layers in the end. Because the input of CV problems, which are mostly images, have three dimensions of width, height and depth(RGB channels), a convolutional layer takes a 3d matrix as input and also yields a 3d matrix with the same depth of input. A conv layer has multiple filters, each of which convolves the input image, that is, slides over image spatially and computes dot products(imaging the 3d matrices are unraveled as 1d vectors. Actually the dot product of unraveled vectors is mathematically equal to the convolution operation of 3d matrices). A filter convolving the images produces a 2d matrix, and all the output of filters stacked along depth dimension produces the output of this convolutional layer.

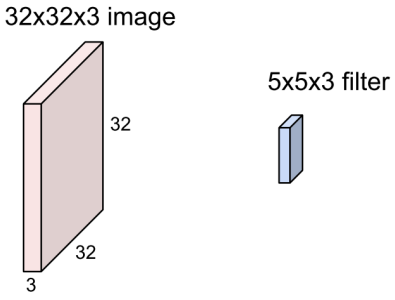

Figure 1: Example of convolutional Layer(with one filter)

Pooling layers are sometime used right after conv-relu layers to squeeze the spatial size(height and width) of flowing tensors to lower the number of learnable weights of the remaining layers.

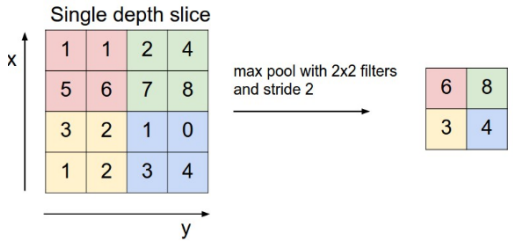

Figure 2: Example of pooling layer

There are plenty of details regarding to the implementation of convnet, and I will leave them to the assignment part of next report.

1.2 Batch Normalization

Batch Normalization was proposed by Ioffe et al in 2015. The intuition is that if the distribution of each layer is gaussian, the training will be much easier. The equation of normalization is given as

$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x]}}$$

Note that the expectation and variance are based on the mini-batch, which is why the method is named batch normalization.

Another issue is that simply normalizing each input of a layer may change what the layer can represent. This paper suggests that for each activation $x^{(k)}$, we add a pair of parameters $\gamma^{(k)}$, $\beta^{(k)}$, which scale and shift the normalized value:

$$y^{(k)} = \gamma^{(k)} * \hat{x}^{(k)} + \beta^{(k)}$$

These parameters are learned along with the original model parameters, and restore the representation power of the network.

Computing the gradients is not difficult since all the equations are differentiable. One thing needs be noticed that batch normalization layer behaves different during training and testing. During test time, the expectation and variance are estimated by all the training data instead of mini-batch.

1.3 Installing PyTorch

The instructor of cs231n introduced TensorFlow and PyTorch as preferred deep learning frameworks. TensorFlow provides a useful visualization tool called TensorBoard that can visualize your network architecture and monitor the process of training. But on the other hand the code is a little bit hard to write, since we have to first create a symbolic graph without actual computation, and then starts the computations that use another set of APIs, which I think will take quite a lot time to learn and get familiar to the APIs. So I decide to use PyTorch instead.

My laptop has Win10 OS and a CUDA compatible GPU, so I installed Win10-CUDA-enabled version of PyTorch via Anaconda following the installation instruction on PyTorch website, which goes quite smooth. Then I downloaded and installed CUDA9.1 from NVIDIA website. Enabling GPU computation on PyTorch is quite simple, you just have to move the variables to GPU and everything else is done automatically by PyTorch.

1.4 Assignment 2

I finished assignment 1 last week so I started assignment 2 this week since their contents are continuous. The implementation of 2- layered fully-connected neural network has done in assignment 1, but the code is imperative, non-modular-style, which is not scalable and extended to the case of multi-layer neural networks. Thus assignment 2 abstracts forward and backward process of layers and implement a feed forward neural network in a more modular approach. In my opinion, I prefer the modular philosophy.

1.4.1 FC Net

A fully-connected takes x(the input matrix), w(weights), b(bias) as input, and computes dot product as output. The input matrices are sometime multiple dimensional(> 2), which means each example has more than 1 dimension. For example the input x has shape $(N, d_1, ..., d_k)$ and contains a minibatch of N examples, where each example $x[i]$ has shape $(d_1, ..., d_k)$. So it is more efficient to reshape each input into a matrix of dimension $D = \prod_{i=1}^{k} d_i$ and then compute the dot products. Therefore FC layer is also called affine layer. The forward pass is simple:

```
x_flat = x.reshape((N, -1))
out = x_flat.dot(w) + b
cache = (x, w, b)
return out, cache
```

Notice that there is an extra variable cache, and it will be returned by the forward pass of all kinds of layers. The reason is that it will be more convenient for backward pass to compute gradients.

The backward pass is also naive:

```
x_flat = x.reshape((N, -1))
dw = x_flat.T.dot(dout)
dx = dout.dot(w.T).reshape(x.shape)
db = dout.sum(0)
```

The softmax loss is identical to the one we've implemented in the previous assignment.

Now all the components, except SGD-based optimizers, of a multi-layered fully-connected neural network are done. The test shows the implementation is correct.

1.4.2 Update Rules

Vanilla SGD has already been implemented in assignment 1, and this assignment requires us to implement SGD+Momentum, RMSProp and Adam. Although the math behind them is complicated, the code is not. Follow the equation and the implementation is straightforward.

1.4.3 BatchNorm

Assignment 2 uses another approach in test time compared to the original averaging method. During training we keep an exponentially decaying running mean of the mean and variance of each feature, and these averages are used to normalize data at test-time.

At each timestep we update the running averages for mean and variance using an exponential decay based on the momentum parameter:

running_mean = momentum * running_mean + (1 - momentum) * sample_mean

running_var = momentum * running_var + (1 - momentum) * sample_var

After the code of BatchNorm layer is finished, we insert them to the fully-connected network, and compare the training time compared to the net without BatchNorm. Here is the result

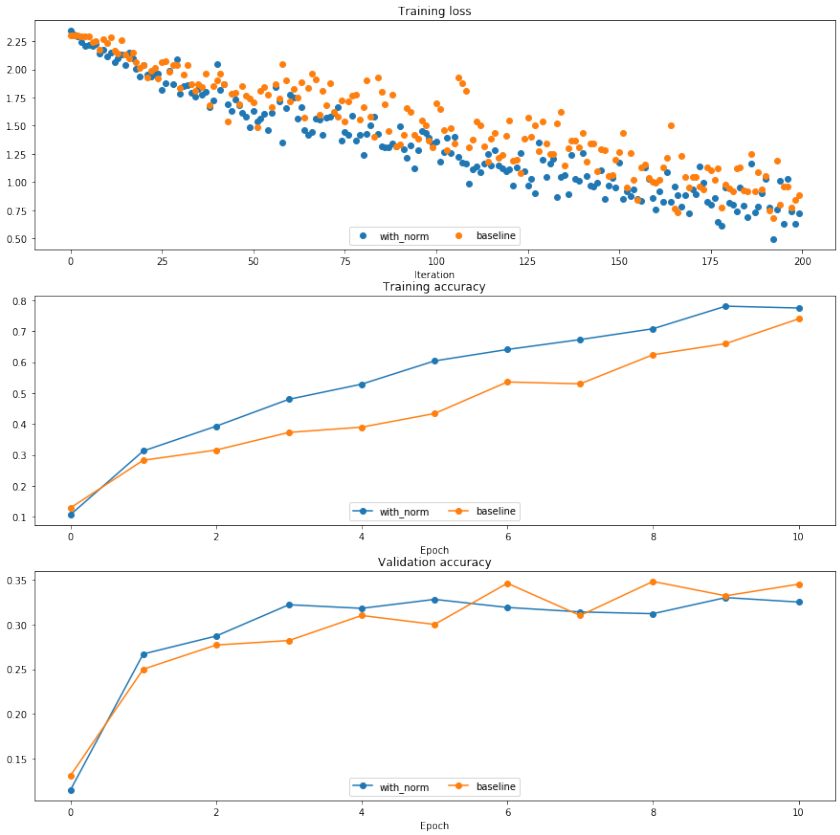

Figure 3: Training speedup

This simple experiment shows that the validation accuracies are comparable, but FC net with batch norm is much faster than baseline(without batch norm). It can be expected that FC with batch norm will exceed FC without batch norm given the same training time.

2 Plans

In the next week I plan to finish assignment 2 and continue to learn cs231n course.