TTIC 31230 Problem Set 3
Win 2017

Hao Jiang

January 26, 2017

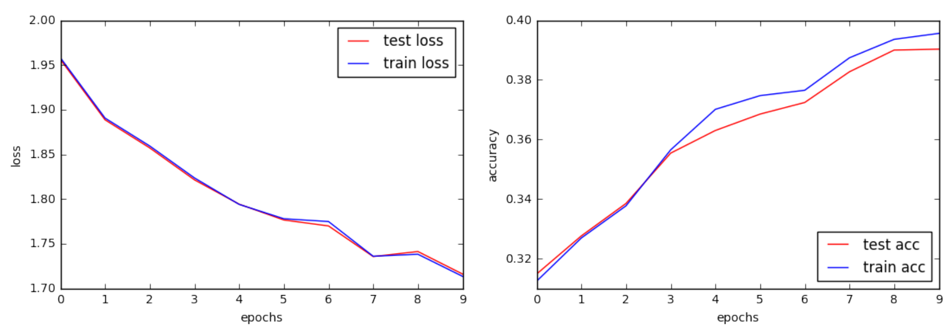
Problem 1

Implementation

Please see attached jupyter notebook for the implementation of Conv / MaxPool / AvePool class as well as the convolutional network architecture.

Experiment Result

The experiment result is demonstrated in Figure 1 and I get a test accuracy of `0.3904` after 10 epochs. The execution time on my test machine is around 400s per epoch.



(a) Training Loss

(b) Test Accuracy

Figure 1: Experiment Result

Performance Optimization

To speed up the convolution operation, I make heave use of numpy's ndarray operations instead of writing nested loops. All operations in my implementation only loops on two dimensions, namely the images' width and height.

In the `forward` function of `MaxPool` and `AvePool`, I use numpy's `max` and `average` operation. Let the shape of output value be (B, W, H, C), I loop on dimension W and H, which indicates a square region on all batch instances and channels. For each such square region, I calculate the max / average on axis 1,2, leaving axis 0 and 3 (which are the index in a batch and channel) unchanged. This operation is efficiently equivalent to compute max / average over each images in a batch and each channels in parallel. It avoids the loop on batch size and number of channels and speed up the operation.

In the `backward` function of `AvePool`, I again only loop on two dimensions. Let the shape of `y.grad` be (B, W, H, C). The loop runs on W and H and get a ndarray of shape (B, C) for each (W, H) pair. For each such ndarray, I use an all-1 square matrix of shape (k, k) to expand it to shape (B, k, k, C), where k is the size of the square region. Let S be the stride size, we have

$$\forall b, c, i, j, k_i, k_j, \texttt{expand}[b, i * S + k_i, j * S + k_j, c] = \texttt{ygrad}[b, i, j, c]$$

By simply adding all these expanding results together, we will have the gradient to be updated to `x.grad`.

In the `backward` function of `MaxPool`, I use the similar idea. But here in each square region, the gradient will only be backprop to locations holding the maximal value. To implement this, we create a mask by comparing `x.value` to a square matrix with every value equal to the maximal value.

An example is shown in Figure 2. Here the input value is $\begin{bmatrix} 3 & 6 \\ 7 & 7 \end{bmatrix}$, and the maximal value is 7. Comparing the input with an all-7 matrix gives us a mask indicating the location of maximal value. When we want backprop a gradient, e.g., 5 to X, we first expand the gradient to a square and element-wise multiply it with the mask, obtaining the backprop result.

For `Conv`'s `forward` and `backward` method, I follow the hint in problem description and use `numpy.einsum` to manipulate multi-dimensional array multiplications. Again this requires only explicit loop on the W and H dimension on the dataset.

$$X = \begin{bmatrix} 3 & 6 \\ 7 & 7 \end{bmatrix}$$

$$\left(\begin{bmatrix} 3 & 6 \\ 7 & 7 \end{bmatrix} == \begin{bmatrix} 7 & 7 \\ 7 & 7 \end{bmatrix} \right) \implies \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$$

$$\text{grad} = 5 \implies$$

$$\begin{bmatrix} 5 & 5 \\ 5 & 5 \end{bmatrix} \circ \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \implies \begin{bmatrix} 0 & 0 \\ 5 & 5 \end{bmatrix}$$

Figure 2: Example: Implementing MaxPool.backward