Author: Benjamin Smidt
Created: September 23, 2022
Last Updated: September 25, 2022

Assignment 3: Convolutional Networks and Batch Normalization

Note to reader.

This is my work for assignment three of Michigan's course EECS 498: Deep Learning for Computer Vision. The majority of explanations and understanding are derived from Justin Johnson's Lectures and Stanford's CS 231N Lecture Notes. This document is meant to be used as a reference, explanation, and resource for the assignment, not necessarily a comprehensive overview of Neural Networks. If there's a typo or a correction needs to be made, feel free to email me at benjamin.smidt@utexas.edu so I can fix it. Thank you! I hope you find this document helpful.

Contents

1 Convolutional Network Nuts and Bolts

1.1 Convolutional Layer Forward

Our first function is a naive implementation of a forward pass of a convolutional layer. We begin by grabbing our pad and stride parameters from the input and creating a new tensor, *x-pad*, that will be our padded version of the input tensors x. Using the stride and padding size, we then find the height and width of our new output and create it.

The formula for finding the new height and width is pretty intuitive. We have that *H-out* $= 1 + (H + 2 *$ *pad* - *H-filter*$)$ / *stride*. It should be fairly easy to see that we're finding the number of possible positions the filter can be place along the height dimension (so with stride 1), dividing by the stride, and then adding 1 because the first placement isn't naturally included in the previous calculation.

I chose to implement this function iteratively since it's a simple naive implementation but there are MUCH faster ways of doing this. We begin by iterating over each example in N and for each example we iterate over each filter f. Then for each filter (yeah I know this for-loop nest a little crazy) we iterate over each possible position in the input, take the dot product, add our bias, and add the final value to the proper position in our final output *out*.

While this method is certainly very slow, I do like the clarity of this code. I will note that for some reason it's particularly easy to forget the bias in this scenario (which had me debugging for literallly an hour), so don't forget that.

1.2 Convolutional Layer Backward

Our backward pass of our convolutional layer reuses a lot of my same code and has the same general format. I grab the inputs, form the padded input, create the outputs, and begin iterating through my crazy nested for-loop. The important decision here that I made is I decided to map my initial gradient onto *dx-pad* instead of *dx* and then just cut off the padding when I returned *dx* as the output. I think this made the code much simpler and easier to understand.

The other significant portion of this code was how I actually computed the gradient. Given that we've been doing gradients and backpropagation

for more than a few assignments now this should be pretty easy to get. Let's start with *dx-pad*. For any given position that we convolve with a filter, the gradient of that operation is simply the filter f. Hence, all we need to do is iterate over each possible position in *dx-pad* with the same stride used in x for our forward pass, and add f multiplied by the corresponding output in *d-out* to that position.

Since *dx-pad* and x have the same shape we can also iterate over x at the same time to compute *dw* (our tensor of filters). For a given filter, the gradient is the sum of all the visited positions in x multiplied by its corresponding output *d-out*. Finally, the gradient of *db* is simply $1 \cdot$ *d-out* for a given position in the input. Thus, we simply add all the values in *d-out* for a given filter.

1.3 Max Pooling Forward

Coding the convolutional layers was a little tough at first but this pooling operation is much easier. It's a max function, which is simply to compute both forward and backward. As in the Last two functions we begin our setup by grabbing our function inputs, defining the proper output shape and creating our output tensor *out*. We do our crazy for-loop in the same manner as usual and set the output of our max pooling operation to the maximum value in a given slice of our tensor. Yeah that's pretty much it. Nothing wild happening here to be honest.

1.4 Max Pooling Backward

By now you're familiar with the setup of my functions (I hope) so I'll skip right to finding the gradient. The gradient of the max values (my implementation allows multiple max values to pass through if they're equivalent but there are different implementations) is 1 (which we multiply be the coresponding *d-out* while every other value has a gradient of zero. Thus we can easily create a mask, *pool-mask*, and use to allow only the max values to be multiplied by *d-out*. Notice how similar this operation feels to a ReLU gradient, which makes sense since they both use a simply $max()$ function.

1.5 Fast Implementations

So in this section we'll not be implementing the parallelized and MUCH faster versions of convolutional and pooling operations (hashtag blessed). We're provided with PyTorches implementation using *torch.nn*. However, I do quickly want to point out how much faster it is. You can see my implementation took nearly 6 seconds (that's reallllly bad). The CPU computed fast implementation took only 0.000705 seconds while the GPU computed fast implementation took only 0.000478 seconds! This is a a speedup by factors of 4500x and 6600x! Even crazier, the speedup factor during backpropagation is over 11000! 11000! Amazing.

1.6 Convolutional Sandwich Layers

Again, no code written here but it's important to at least mention. We're provided with some functions that combine operations like we last assignment with the linear and ReLU operations. These functions actually use the classes and functions we created in the Fully Connected Networks assignment to implement Conv-ReLU (convolutional layer followed by ReLU) and Conv-ReLU-Pool (convolutional layer followed by max pooling)

2 End-to-End Convolutional Networks

2.1 3 Layer Convolutional Network

2.1.1 Initialization

2.1.2 Loss

2.1.3 Gradient

2.2 Deep Convolutional Network

3 Kaiming Initialization

4 Batch Normalization