

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	3
1.1	Convolutional Layer Forward	3
1.2	Convolutional Layer Backward	3
1.3	Max Pooling Forward	4
1.4	Max Pooling Backward	4
1.5	Fast Implementations	5
1.6	Convolutional Sandwich Layers	5
2	End-to-End Convolutional Networks	5
2.1	3 Layer Convolutional Network	5
2.1.1	Initialization	5
2.1.2	Loss	6
2.1.3	Gradient	7
2.2	Deep Convolutional Network	7
2.2.1	Initialization	7
2.2.2	Loss and Gradient	7

3	Kaiming Initialization	8
4	Batch Normalization	8

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 placed 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 is 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 literally 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 by the corresponding *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 PyTorch's 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 realllly 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 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

Since this is our 3rd or 4th network we've built and this one isn't particularly different from the fully connected network except there's a convolutional layer at the beginning, I'll mainly focus on finding the proper dimensions before and after our convolutional layer. We've already done the hard work of implementing our forward pass for the loss and backpropagation in a modular design. All that's needed now is to work out how to put it all together into one coherent network.

Since our architecture is Conv - ReLU - 2x2 Max Pool - Linear - ReLU - Linear - Softmax, we'll need three weight tensors and three bias tensors. I use "tensors" to be general (since some of these aren't matrices) but some are just regular matrices.

Our first layer is our convolutional layer followed by ReLU and max pooling. We create the weight tensor $W1$ with the following four dimensions: number of filters (*num-filters*), number of channels (C) in our input X , filter height (*filter-size*), and filter width (*filter-size*). Note that filter height and width are generally equal and this implementation actually doesn't allow them to be unequal.

Since the convolution, ReLU, and max pooling are treated as "1 layer" we need to find the output shape of our input after the max pooling operation (since this will be the input into the next layer). Our convolutional layer preserves the input spatial size by setting *padding* = (*filter-size* - 1). Since our input tensor shape is N (number of examples) $\times C \times H$ (input height) $\times W$ (input width), it's shape after the convolution input tensor will be of shape $N \times \text{num-filters} \times H \times W$. ReLU will not change these dimensions at all.

Now, our max pooling will not change N or *num-filters* but will change H and W such that $H\text{-post-pool} = 1 + (H - 2) / 2$ and $W\text{-post-pool} = 1 + (W - 2) / 2$. If you recall how we implemented our *Linear* class, it flattens everything after the first dimension N . Thus, our matrix after post pooling, which we're multiplying by $W2$, will effectively have shape $N \times (\text{num-filters} \cdot H\text{-post-pool} \cdot W\text{-post-pool})$. Thus we need to make $W2$ have dimensions $(\text{num-filters} \cdot H\text{-post-pool} \cdot W\text{-post-pool}) \times \text{hidden-dim}$ (dimensions of our hidden layer).

We then continue as normal with $W3$ having dimensions $\text{hidden-dim} \times \text{num-classes}$ to classify our output. The last thing to note here is that we initialize all our weight matrices according to a normal distribution with mean 0 and standard deviation equal to *weight-scale*.

2.1.2 Loss

We've already done all the hard work so I'll make this short. We use three different classes for the three different layers: *Conv-ReLU-Pool*, *Linear-ReLU*, and *Linear*. These capture our entire architecture and all we must do is simply pass our chain of inputs and outputs to receive our final *scores* matrix at the end. From there we pass our *scores* matrix off to our *softmax-loss* function to find our data loss. Finally, we add our regularization loss as usual (don't forget this!).

2.1.3 Gradient

Again, we've already done all the hard work. We have our gradient with respect to our *scores* matrix returned from our *softmax-loss* function. From here we just pass it down the line through our backwards implementations of the classes mentioned above in the "Loss" section. We find our gradients, add the additional gradient for regularization and place them in our dictionary. Done :).

2.2 Deep Convolutional Network

I will keep this section particularly short since it's just a generalization of the previous section.

2.2.1 Initialization

Our architecture allows for an arbitrary number of convolutional layers that all use a 3x3 filter size and preserve the spatial size of the input. For pooling we always use 2x2 max pooling. As a result, we initialize all our convolutional layers with a filter height of 3 and a filter weight of 3. We then just grab the number of filters specified by the input user for each layer.

The number of channels for a given weight tensor will be the number of filters used in the last operation. We just store this and iterate over each loop to update for the next loop. Things are different for the first, where the number of channels is just equal to C (input channel dimension), and the last layer, which doesn't directly take the number of channels as a dimension. Instead it uses the number of channels to calculate the flattened dimension size of $N \times \text{fully-connected-dimension}$.

2.2.2 Loss and Gradient

Honestly, this process is similar enough to Fully Connected assignment that I feel comfortable not explaining this at all really. Per usual, I'm just passing the inputs and outputs through the network and checking if a given layer has max pooling or not. The final layer is just a linear layer so it's outside the for loop and then we turn around and chug all the way back to the beginning. Seriously nothing new going on here.

3 Kaiming Initialization

4 Batch Normalization