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

Since this is our 3rd or 4th network we've built and this one isn't partcularly 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) x C x H (input height) x W (input width), it's shape after the convolution input tensor will be of shape N x *num-filters* x H x 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-post-pool* = 1 + (H - 2) / 2 and *W-post-pool* = 1 + (H - 2) / 2. If you recall how we implemented our *Linear* class, it flattens everythin after the first dimension N. Thus, our matrix after post pooling, which we're multiplying by $W2$, will effectively have shape N x (*num-filters* · *H-post-pool* · *W-post-pool*). Thus we need to make $W2$ have dimensions (*num-filters* · *H-post-pool* · *W-post-pool*) x *hidden-dim* (dimensions of our hidden layer).

We then continue as normal with $W3$ having dimensions *hidden-dim* x *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 x *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

What is Kaiming initialization? Good question. Johnson's discussion of Kaiming initialization was brief and the notes do not cover it at all. Although, it's a pretty simply idea at its core. Kaiming initialization applies specifically to using the ReLu function and aims to "avoid the reducing or magnifying of input signals exponentially". Essentially, it's often the case in deep networks that inputs become distorted such that they grow or shrink exponentially which becomes a real hindrance to network's ability to computationally learn.

If you want to read the derivation you can find it here. In practice, all you need to do is initialize your network as

$$w_l \sim \mathcal{N}(0, \sqrt{\frac{2}{n_l}})$$

where w_l is a weight matrix for a given layer and $n_l = k_l^2 c_l$. k_l is the kernel size for a convolutional layer and c_l is the input channel dimensions for that layer.

4 Batch Normalization

4.1 Conceptual Understanding

Finally, batch normalization. Batch normalization is one of the most useful (and intriguing) tactics used. As described https://arxiv.org/abs/1502.03167 in the original paper, batch normalization aims to fix "internal covariate shift". Essentially, because each layer depends on the output of the previous, there's constantly a change in the distribution of data being input (from the output of the previous layer).

This is an issue when learning because each gradient update is made for the entire network but the latter layers' updates may not apply as well (or at all) to the newly propagated outputs from the updates to earlier layers. This distributional shift slows down or even stops learning entirely. To counteract this we introduce batch normalization. We use the following equations to normalize a batch B of input x into $\hat{x}_i$, the normalized batch.

$$\mu_B = \frac{1}{m} \sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

For each training of mini-batch, we alter the data such that it has zero mean and unit variance. This allows the data to see the same distribution no matter its place in the network. However, there are cases where normalizing your data alters what the data can represent. The sigmoid function, for example, would be mostly constrained to its linear regime between -1 and 1. This is undesirable if the optimal parameters are outside of this regime. So we need some way to represent the original sigmoid function fully (in technical speak we make the the "transformation inserted in the network can represent the identity transform").

To fix this we add two parameters: γ (scale parameter) and β (shift parameter). Note that these parameters are unique to each layer. Thus, each layer where batch normalization is used will (likely) have a different γ or β. These scale and shift parameters can be learned, allowing the network to learn the optimal normalization for a given layer. Instead of our final output being $\hat{x}_i$, our final output is $\hat{y}_i$ where

$$\hat{y}_i = \gamma\hat{x}_i + \beta$$

There is one last kink we must remember using batch normalization:test time. Obviously, we want our predictions to be deterministic. No matter what other data we group with a given input, we need it to give the same output. Thus, we can't normalize our test time data with whatever batch we input it with because it isn't deterministic. Some fraction of the output depends on the other inputs in the batch. To fix this, we normalize our test inputs using a running mean and running average accrued during training. To show you what I mean concretely, we add the following code to the end of each training batch.

```
running_mean = momentum * running_mean + (1 - momentum) * batch_mean
running_var = momentum * running_var + (1 - momentum) * batch_var
```

The running mean and variance are updated for every single batch and stored in a dictionary to be used at test time. Usually we set the momentum to 0.9.

4.2 Forward

Since you just need to implement what I described above, I'll keep this short and sweet. Calculate the forward pass with the above equations. γ, β, ϵ, the mode (train or test), running mean, running variance, and momentum will all be passed into your forward pass function. In the cache for our backward pass we'll need $\beta, \gamma, x, \mu, \sigma, \hat{x}$, and ϵ.

4.3 Backward

Now we need to back propagate. Specifically we need to find the gradient of our loss function with respect to x_i. This backpropagation in particular is especially confusing (for me). We'll go back one function at a time beginning with $\hat{y}_i$. I'll put the original equations here for reference. Keep in mind that these gradients are for a (single) given example i.

$$\hat{y}_i = \gamma \hat{x}_i + \beta$$
$$\frac{\partial \hat{y}_i}{\partial \hat{x}_i} = \gamma$$

Now, we move backward to the next function.

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$
$$\frac{\partial \hat{x}_i}{\partial x_i} = \frac{1}{\sqrt{\sigma_B^2 + \epsilon}}$$
$$\frac{\partial \hat{x}_i}{\partial \sigma_B^2} = \frac{x_i - \mu_B}{(-2)\sqrt[3/2]{\sigma_B^2 + \epsilon}}$$
$$\frac{\partial \hat{x}_i}{\partial \mu_B} = \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}}$$

These are our three gradients with respect to $\hat{x}_i$ but remember that μ_B and σ_B^2 both depend on x_i so we must find their gradients with respect to x_i as well, not just $\frac{\partial \hat{x}_i}{\partial x_i}$. If that doesn't make sense to you right now just keep moving and it'll make sense here soon. We'll start with finding s$\frac{\partial \sigma_B^2}{\partial x_i}$.

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2$$

$$\frac{\partial \sigma_B^2}{\partial x_i} = \frac{2(x_i - \mu_B)}{m}$$
$$\frac{\partial \sigma_B^2}{\partial \mu_B} = \frac{-2(x_i - \mu_B)}{m}$$

Because σ_B^2 depends on μ_B we must add $\frac{\partial \sigma_B^2}{\partial \mu_B}$ to $\frac{\partial \hat{x}_i}{\partial \mu_B}$ to obtain the gradient with respect to μ_B. Now we're ready to find $\frac{\partial \mu_B}{\partial x_i}$.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$
$$\frac{\partial \mu_B}{\partial x_i} = \frac{1}{m}$$

Okay, this has been a lot so far so let's recap where we're at. We didn't talk about this earlier but remember that our backpropagation is with respect to our initial loss function, so we have

$$\frac{\partial l}{\partial x_i} = \frac{\partial l}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial \hat{x}_i}(\frac{\partial \hat{x}_i}{\partial x_i} + \frac{\partial \hat{x}_i}{\partial \sigma_B^2}(\frac{\partial \sigma_B^2}{\partial x_i} + \frac{\partial \sigma_B^2}{\partial \mu_i}\frac{\partial \mu_i}{\partial x_i}) + \frac{\partial \hat{x}_i}{\partial \mu_B}\frac{\partial \mu_B}{\partial x_i})$$

We can plug this in to get our final equation for $\frac{\partial l}{\partial x_i}$.

$$\frac{\partial l}{\partial x_i} = \frac{\partial l}{\partial \hat{y}_i}\cdot\gamma(\frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{x_i - \mu_B}{(-2)\sqrt[3/2]{\sigma_B^2 + \epsilon}}(\frac{2(x_i - \mu_B)}{m} + \frac{-2(x_i - \mu_B)}{m}\cdot\frac{1}{m}) + \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}}\cdot\frac{1}{m}$$

Again, please note that these gradients are with respect to a single example. Thus, when coding this for a given batch you may need to sum across the examples to get the correct gradient. For instance, the shape of $\frac{\partial}{\partial sigma_B^2}$ is the same as σ_B^2. However, there's a separate gradient with respect to each function so you'll need to sum across all the examples to get the correct gradient. Otherwise you'll end up with shape N x D even though σ_B^2 is shape D (vector).

This concludes all the functions implemented but the last thing to do is to actually go back and implement batch normalization in code. It's not "difficult" per se as much as it's just confusing to do. Here's two things to remember when using batch normalization: you need to initialize γ and β for every layer using batch normalization and the batch normalization classes accept and return more parameters. Just keep that in mind because if you

don't initialize those variables and pass everything through properly things can easily get messed up. One last thing to remember. For convolutional layers, remember that the number of channels in the next layer is equal to the number of filters in the current layer. This is important to keep in mind because γ and β are vectors that are the same size as the number of channels for a given layer.