Convolutional Neural Network Architecture in NumPy

Aaron Avram

June 14 2025

Introduction

In this writeup I am going to outline the design of my Convolutional Neural Network for image classification. However, my hope is to build a module that abstracts away individual model design and provides a framework for deploying any model type, drawing some inspiration from the archictecture of PyTorch.

High-Level Design

At the foundation of my model module is the Tensor class. This is a class with two NumPy arrays as attributes, one representing a value matrix and the other a gradient matrix. Since practically all of the components of a Neural network store value and gradient data it felt natural to couple them under a single class. The Tensor class will support gradient update and setting the gradient to zero.

On top of this class is the Layer class. This is meant to capture the core functionality of a layer in a neural network: it has an input, an (optional) list of parameters and an output, all stored as Tensor objects. The class supports a forward and a backward pass. I will create subclasses of the layer class for individual layer types, e.g. Convolutional, Dense or Activation. However, the superclass allows for polymorphic behaviour when implementing an entire model.

Finally, I am going to build the model class which has a list of layers, which chain together from input to output, using aliasing to conserve memory. My CNN model will be a particular instance of the model class with its layers structured in the standard way for image classifying CNNs.

Mathematical details

I will not go into the details of backpropagation for the dense layers in my model, as I have already derived these things in my NameNet project and will instead focus on the Convolutional, Batch Norm, and Pooling layers of my model.

Convolutions

The first layer in my archictecture takes in a batch of training examples $X^{(in)}$ which has dimensions (n, c_{in}, d, d), assuming square input data. Where n is the size of the batch. c_{in} is the number of channels in the input data, and d is the size of each training point. We then convolve this tensor with a kernel tensor K of dimension $(c_{out}, c_{in}, k, k$, where $c_{out})$ is the number of points in the next layer and K is the size of the kernel tensor. Now the Convolution operation on these tensors is defined by:

$$(X^{(in)} * K)_{i,j,k,l} = \sum_{r,s,t} X^{(in)}_{i,r,k+s,l+t} K_{r,s,t}$$

This is frankly a bit of a mess and inefficient as its time complexity scales linearly in each argument of $X^{(in)}$ and K getting out of hand quickly.

Luckily there is a more efficient way to perform this operation, as we can see that K slides along the each axis of X acting as a sort of slighting window. Thus, we can first copy out the necessary part of X then implement this operation as a matrix operation between the reshaped components. For this we will use a function provided by NumPy to make a copy of X and we will call this $X^{(col)}$, and if we let p be the size of the output points we see that $X^{(col)}$ will have dimension (n, c_{in}, p, p, k, k). So $X^{(col)}$ has all of the windows stored. Next we reorder the indices of $X^{(col)}$ with NumPy's transpose function to give it dimension (c_{in}, k, k, n, p, p) then we reshape it to have dimension $(c_{in}k^2, np^2)$. Next we reshape K to dimension $(c_{out}, c_{in}k^2)$. Then we can calculate their convolution by taking:

$$X^{(out)} = K(X^{(col)})$$

which is of dimension (c_{out}, np^2). If we reshape $X^{(out)}$ we can give it the dimension (n, c_{out}, p, p). This implementation is orders of magnitude faster than the previous implementation.

A note, often times these convolutions are done using strides, i.e. skipping certain windows when constructing $X^{(col)}$, this doesn't really change any derivations above or further down, only burdens notation.

Convolution Gradient

The most convenient way to reason about the convolution gradient is to consider the operation in its matrix form as outlined above, without $X^{(out)}$. Now, consider our setup earlier with $X^{(out)} = X^{(col)}K^T$. Then:

$$X^{(out)}_{ij} = \sum_{p=1}^{c_n k^2} K_{ip} X^{(col)}_{pj}$$

Then:

$$\frac{\partial X^{(out)}_{ij}}{\partial K_{kl}} = \begin{cases} 0 & i \neq k \\ X^{(col)}_{lj} & i = k \end{cases}$$

Since this information does not depend on i or k at all, the entire derivative information of $X^{(out)}$ with respect to K, which we will denote by $g(X^{(out)}, K) = (X^{col})$ which has dimension $(c_{in}k^2, np^2)$ Then if we are given $g(L, X^{(out)})$ which can be shaped as (c_{out}, np^2). Then it is clear that:

$$\begin{aligned} g(L, K) &= g(L, X^{(out)}) g(X^{(out)}, K)^T \\ &= g(L, X^{(out)})(X^{(col)})^T \end{aligned}$$

We can verify that this makes sense by considering each entry:

$$g(L, K)_{ij} = \sum_{k=1}^{np^2} g(L, X^{(out)})_{ik} g(X^{(out)}, K)_{jk}$$

As this gradient information for $X^{(out)}$ with respect to K is shared across each column:

$$= \sum_{k=0}^{np^2} \frac{\partial L}{\partial X_{ik}^{(out)}} \frac{X_{ik}^{(out)}}{K_{ij}}$$

Which is exactly what we want.

Next we look to find the gradient of the loss with respect to the input layer $X^{(in)}$. However our computations input $X^{(col)}$ and in fact it is easier to compute the gradient with respect to $X^{(col)}$ then use a sort of inverse function the sliding window function used to produce $X^{(col)}$ (a function easy to implement) to get the desired gradient. So consider again:

$$X_{ij}^{(out)} = \sum_{p=1}^{c_n k^2} K_{ip} X_{pj}^{(col)}$$

Then:

$$\frac{\partial X_{ij}^{(out)}}{\partial X_{kl}^{(col)}} = \begin{cases} 0 & j \neq l \\ K_{ik} & j = l \end{cases}$$

Again noticing that this derivative does not depend on j or l whatsoever, we can package the partial derivatives within $g(X^{(out)}, X^{(col)}) = K$. Then if we have $g(L, X^{(out)})$ with dimension (c_{out}, np^2), we find:

$$\begin{aligned} g(L, X^{(col)}) &= g(X^{(out)}, K)^T g(L, X^{(out)}) \\ &= K^T g(L, X^{(out)}) \end{aligned}$$

To verify this:

$$g(L, X^{(col)})_{ij} = \sum_{k=1}^{c_{out}} K_{ki} g(L, X^{(out)})_{kj}$$

As this gradient information for $X^{(out)}$ with respect to $X^{(col)}$ is shared across each row:

$$= \sum_{k=0}^{c_{out}} \frac{\partial X_{kj}^{(out)}}{\partial X_{ij}^{(col)}} \frac{\partial L}{\partial X_{kj}^{(out)}}$$

As required.

Batch Normalization

Before the output of the convolution layer is put through the Activation layer, we are going to apply batch normalization. This consists of two stages, but they will be part of the layer. Doing some relabeling, say the output of the convolution layer is X of dimension (c, np^2). Then we first produce $\hat{X}$ by normalizing X across each channel. More specifically, let μ be the mean across each channel and let ν be the variance per channel then we let $\sigma = \sqrt{\epsilon + \nu^2}$ and

$$\hat{X} = \frac{X - \mu}{\sigma}$$

The epsilon being included to avoid the undefined derivative of the square root function at 0.

Now the second step of the batch norm is to reparametrize the normalized output with a learnable mean β and standard deviation γ, specifically:

$$Y = \gamma \hat{X} + \beta$$

Where Y is the output of the layer.

Now for our gradients, if we are given $g(L, Y)$ of dimension (c, np^2), it is simple to find $g(L, \gamma)$ and $g(L, \beta)$. First consider:

$$\frac{\partial Y_{ij}}{\partial \gamma_k} = \begin{cases} 0 & i \neq k \\ \hat{X}_{ij} & i = k \end{cases}$$

So it is clear that $g(Y, \gamma) = \hat{X}$ and so $g(L, \gamma) = \sum_{k=1}^{np^2} \Big(g(L, Y) \odot g(Y, \gamma) \Big)_{:k}$. Which, with NumPy would look like a sum across the first axis.

We can figure out $g(L, \beta)$ from this by inspection as it is a constant term so $g(L, \beta) = \sum_{k=1}^{np^2} g(L, Y)_{:k}$