Data C182 Designing, Visualizing & Understanding DNN
Fall 2024 Eric Kim, Naveen Ashish

Discussion 04

This discussion will cover CNN and RNN.

1. Convolutional Neural Networks

Convolutional neural networks[1] (CNN) are a type of neural network architecture that have become the key ingredient for state of the art modern computer vision performance.

They perform operations similar to feed-forward neural networks that we have discussed, but explicitly account for spatial structure in the data, and so are very common for computer vision tasks where inputs are images. That said, CNNs can also be applied to non-image data with similar structure in the input, such as time series or text data (in which case they're taking advantage of temporal structure).

0.1 Convolution (Cross-Correlation) Operator

At the heart of CNNs is the convolution operator. In this discussion, what we refer to as a convolution is actually the **cross-correlation** operator here instead, which is the exact same but with the indexing of the weights in $\mathbf{w}$ inverted. For example, "convolutional" layers in the deep learning library Pytorch are also actually cross-correlations instead, and homework 1 will also similarly have you implement cross-correlation instead of the actual convolution.

To motivate the use of convolutions, we will work through an example of a 1-D convolution calculation to illustrate how convolutions work over a single spatial dimension. Suppose we have an input $\mathbf{x} \in \mathbb{R}^n$, and filter $\mathbf{w} \in \mathbb{R}^k$. We can compute the convolution of $\mathbf{x} \star \mathbf{w}$ as follows:

(a) Take your convolutional filter $\mathbf{w}$ and align it with the beginning of $\mathbf{x}$. Take the dot product of $\mathbf{w}$ and the $\mathbf{x}[0 : k-1]$ (using Python-style zero-indexing here) and assign that as the first entry of the output.

(b) Suppose we have stride s. Shift the filter down by s indices, and now take the dot product of $\mathbf{w}$ and $\mathbf{x}[s : k-1+s]$ and assign to the next entry of your output.

(c) Repeat until we run out of entries in $\mathbf{x}$.

Below, we illustrate a 1D convolution with stride 1.

$$\overbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_k \\ \vdots \\ x_n \end{bmatrix}}^{\text{Input vector } \mathbf{x} \in \mathbb{R}^n} \star \overbrace{\begin{bmatrix} w_1 \\ \vdots \\ w_k \end{bmatrix}}^{\text{Convolutional filter } \mathbf{w} \in \mathbb{R}^k} = \overbrace{\begin{bmatrix} \sum_{i=1}^k w_i x_i \\ \sum_{i=1}^k w_i x_{i+1} \\ \vdots \\ \sum_{i=1}^k w_i x_{i+n-k} \end{bmatrix}}^{\text{Output vector } \mathbf{y} \in \mathbb{R}^{n-k+1}}$$

[1] Recommended reading: http://cs231n.github.io/convolutional-networks/

 All Rights Reserved. This may not be publicly shared without explicit permission.

We see that the output vector is smaller than the input vector ($\mathbb{R}^{n-k-1}$ compared to $\mathbb{R}^n$). A common way to address this is **zero-padding**, in which we append zeros on both ends of the input vector before applying the convolution (note that there are other conventions for zero-padding as well).

Often, we'll be dealing with multiple spatial dimensions (2 spatial dimensions in the case of images). In this case, we would need to slide our filter along all spatial dimensions to construct the output.

Problem 1: Test your know knowledge of convolution dimensions

In this problem, we will run a series of convolution-related operations to better understand how dimensions are affected by convolutions.

(a) i. Suppose you have a $32 \times 32 \times 3$ image (a 32×32 image with 3 input channels). What are the resulting dimensions when you convolve with a $5 \times 5 \times 3$ filter with stride 1 and 0 padding?

ii. What if we zero-pad the input by 2?

iii. Suppose we now stack 10 of these $5 \times 5 \times 3$ filters and continue to zero pad the input by 2. What is the new shape of the output, and how many parameters are in our filters (not including any bias parameters)?

iv. What would be the spatial dimensions after applying a 1×1 convolution? Think about what this does.

Solution:

Solution 1: Test your know knowledge of convolution dimensions

i. The resulting spatial dimensions are 28×28 (with one output channel).

ii. The resulting spatial dimensions are 32×32, so we have preserved the same size as the input image.

iii. The resulting outputs are $32 \times 32 \times 10$, with 10 output channels. There are $5 \cdot 5 \cdot 3 = 75$ parameters per filter, so with 10 filters, we have 750 parameters in this layer. Note that, if we did choose to include a bias parameter, then there would be 76 parameters per filter, and so 760 in total.

iv. A 1×1 convolution does not change the spatial dimensions. For every spatial location, it performs a linear map of the the input channels pointwise over space. In practice, this is useful for changing the number of channels.

(b) (Convolutions as Matrix Multiplication) We note that convolutions are a linear operation. Recalling linear algebra, any linear map (between finite-dimensional spaces) can be expressed as a matrix, so we will see in this section how to write a convolution as a matrix multiplication.

 All Rights Reserved. This may not be publicly shared without explicit permission.

Problem 2: Expressing convolutions as matrix multiplication

We shall again consider a 1D convolution. Consider an input $\mathbf{x} \in \mathbb{R}^4$ and filter $\mathbf{w} \in \mathbb{R}^3$. Letting $\bar{\mathbf{x}}$ denote the result of zero-padding the input by 1 on each end, what is the matrix W such that

$$\overbrace{W}^{\mathbb{R}^{4\times 6}} \quad \overbrace{\begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix}}^{\text{Zero padded input } \bar{\mathbf{x}} \in \mathbb{R}^6} \quad = \bar{\mathbf{x}} * \mathbf{w}?$$

Solution:

Solution 2: Expressing convolutions as matrix multiplication

Computing the convolution, we see that

$$\bar{\mathbf{x}} * \mathbf{w} = \begin{bmatrix} x_1 w_2 + x_2 w_3 \\ x_1 w_1 + x_2 w_2 + x_3 w_3 \\ x_2 w_1 + x_3 w_2 + x_4 w_3 \\ x_3 w_1 + x_4 w_2 \end{bmatrix}.$$

Writing this out as a matrix multiplication, we obtain

$$W = \begin{bmatrix} w_1 & w_2 & w_3 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & w_3 \end{bmatrix}.$$

We can observe now that the resulting matrix will be very sparse (most entries are 0) if the filter size is much smaller than the input size, corresponding to the fact that such convolutions exploit spatial locality. We also observe that there is a lot of parameter reuse, as the convolutional filter weights are repeated many times throughout the explicit matrix.

This has several implications. First of all, this implies that convolutional layers are less expressive than fully-connected layers (as fully connected layers are represented by arbitrary matrices).

Another important implication stems from the fact that we have very optimized tools for computing matrix multiplications. While a naive implementation of a convolution will require looping over all the spatial dimensions, it will turn out that reformulating the convolution as a matrix multiplication will often be much faster due to these optimizations (for example, the Cythonized im2col function in part 4 of homework 1 essentially does this).

(c) (Backwards Pass for a Convolution) We'll consider the same 1D convolution as before, but without zero-padding for simplicity.

 © UCB Data C182, Fall 2024. All Rights Reserved. This may not be publicly shared without explicit permission.

Problem 3: Backwards pass for convolutions

Let $\mathbf{y} = \mathbf{x} * \mathbf{w} \in \mathbb{R}^2$, where $\mathbf{w} \in \mathbb{R}^3$, $\mathbf{x} \in \mathbb{R}^4$. Let $\nabla_{\mathbf{y}} L$ denote the gradient of the loss with respect to the output of the convolution. Compute the gradients of L with respect to $\mathbf{x}$ and $\mathbf{w}$. Can you express the gradients as convolutions themselves?

Solution:

Solution 3: Backwards pass for convolutions

Let $\delta_i = \frac{\partial L}{\partial y_i}$. We can explicitly write out the partial derivatives with respect to each entry of $\mathbf{x}$.

$$\frac{\partial L}{\partial x_1} = w_1 \delta_1$$
$$\frac{\partial L}{\partial x_2} = w_2 \delta_1 + w_1 \delta_2$$
$$\frac{\partial L}{\partial x_3} = w_3 \delta_1 + w_2 \delta_2$$
$$\frac{\partial L}{\partial x_4} = w_3 \delta_2$$

We recognize this as convolution where we zero pad δ by 2 on each end, and convolve with the filter $\tilde{\mathbf{w}}$, where $\tilde{\mathbf{w}}$ reverses the entries of the filter $\mathbf{w}$. (Draw this out for students, explain why sliding the filter along means that we should convolve the output derivative with $\tilde{\mathbf{w}}$ instead of $\mathbf{w}$).

Now, we can similarly compute the partial derivatives for $\mathbf{w}$

$$\frac{\partial L}{\partial w_1} = \delta_1 x_1 + \delta_2 x_2$$
$$\frac{\partial L}{\partial w_2} = \delta_1 x_2 + \delta_2 x_3$$
$$\frac{\partial L}{\partial w_3} = \delta_1 x_3 + \delta_2 x_4$$

We see that $\frac{\partial L}{\partial \mathbf{w}} = \mathbf{x} \star \nabla_{\mathbf{y}} L$ with no zero-padding.

(d) (**backpropagation, 2D case, through a convolution layer**) For the 2D convolution, we define the forward pass as

$$y_{i,j,c'} = \sum_{h,w,c} x_{i-h,j-w,c} w_{h,w,c,c'}$$

or more compactly as $Y = X * w$, where X is the input tensor, w is the kernel tensor, Y is the output tensor, and $*$ is the convolution operator.

The backward pass requires us to calculate the following two terms, $\frac{\partial Loss}{\partial w}$, $\frac{\partial Loss}{\partial X}$, which (by the chain

 All Rights Reserved. This may not be publicly shared without explicit permission.

rule) can be expressed as:

$$\frac{\partial L}{\partial w} = \frac{\partial Y}{\partial w} \cdot \frac{\partial L}{\partial Y}$$
$$\frac{\partial L}{\partial X} = \frac{\partial Y}{\partial X} \cdot \frac{\partial L}{\partial Y}$$

Where we are given $\frac{\partial L}{\partial Y}$ (eg the "dout" parameter passed into each "backwards()" function in HW01). First, for this question: derive the gradients for the input: $\frac{\partial L}{\partial X}$.
Hint: We can emulate the 1D case by calculating with all the indices, but we can also do this conceptually. Conceptually, if we pick a single pixel $x_{i,j,c}$ in the input, we can see that it contributes to multiple output pixels, and if we vary it by an infinitesimal amount $\delta x_{i,j,c}$, how does it vary the output tensor Y, and how does it vary the loss $\mathcal{L}$?

Solution: Since convolution is linear in both inputs and weights, we have $\delta Y = w * \delta X$. So, varying $\delta x_{i,j,c}$ will vary $y_{i+h,j+l,c'}$ by $w_{h,l,c,c'}\delta x_{i,j,c}$, where h, l are the indices of the filter, and c' is the output channel.
The loss $\mathcal{L}$ will vary by

$$\sum_{h,l,c'} \frac{\partial \mathcal{L}}{\partial y_{i+h,j+l,c'}} \frac{\partial y_{i+h,j+l,c'}}{\partial x_{i,j,c}} \delta x_{i,j,c} = \sum_{h,l,c'} \frac{\partial \mathcal{L}}{\partial y_{i+h,j+l,c'}} w_{h,l,c,c'} \delta x_{i,j,c}$$

giving

$$\frac{\partial \mathcal{L}}{\partial x_{i,j,c}} = \sum_{h,l,c'} \frac{\partial \mathcal{L}}{\partial y_{i+h,j+l,c'}} w_{h,l,c,c'}$$

where you might notice that the right hand side is a convolution of the gradient of the loss with respect to the output with the weights, except the weights are flipped. Concretely, we can define the following "transposed" (rotated 180 degrees) kernel w^T (note: this is not actually a transpose):

$$w^T_{h,l,c',c} := w_{k_h-h,k_w-l,c,c'}$$

where k_h, k_w denote the spatial height/width of the kernel.
then we can write the backward pass as $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} * w^T$.

(e) **(backpropagation, 2D case, kernel weights)**
Next, derive the gradients for the weights $\frac{\partial Loss}{\partial w}$.
Again consider the case of

$$y_{i,j,c'} = \sum_{h,w,c} x_{i-h,j-w,c} w_{h,w,c,c'}$$

If we vary the kernel weights by $\delta w_{h,w,c,c'}$, how does it vary the output tensor Y, and how does it vary the loss $\mathcal{L}$?

Solution: Varying $\delta w_{h,w,c,c'}$ will vary each of $y_{i,j,c'}$ by $\delta w_{h,w,c,c'} x_{i-h,j-w,c}$, where i, j ranges over the height and width of the output tensor. The loss $\mathcal{L}$ will vary by

$$\sum_{i,j} \frac{\partial \mathcal{L}}{\partial y_{i,j,c'}} x_{i-h,j-w,c} \delta w_{h,w,c,c'}$$

 © UCB Data C182, Fall 2024. All Rights Reserved. This may not be publicly shared without explicit permission.

giving

$$\frac{\partial \mathcal{L}}{\partial w_{h,w,c,c'}} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial y_{i,j,c'}} x_{i-h,j-w,c}$$

More compactly, we can write this as $(dw)_{h,w,c,c'} = \sum_{ij} (dY)_{i,j,c'} (X)_{i-h,j-w,c}$, which can be seen as a 2D convolution of dY with X reflected across its origin. That is, define X^T as the tensor with entries $X^T_{i,j,c} = X_{-i,-j,c}$, then we can write the backward pass as

$$dw_{\cdot,\cdot,c,c'} = (X^T)_{\cdot,\cdot,c} * (dY)_{\cdot,\cdot,c'}$$

Note: A good reference for backward pass in CNNs is Convolutions and Backpropagations.

(f) A **maxpooling** layer has 2 architectural hyperparameters: the stride step size(S) and the "filter size" (K). The maxpooling operation takes the maximum value in each $K \times K$ window of the input, and strides by S pixels each time. See Figure 1 for an example of maxpooling with $K = 2, S = 2$.

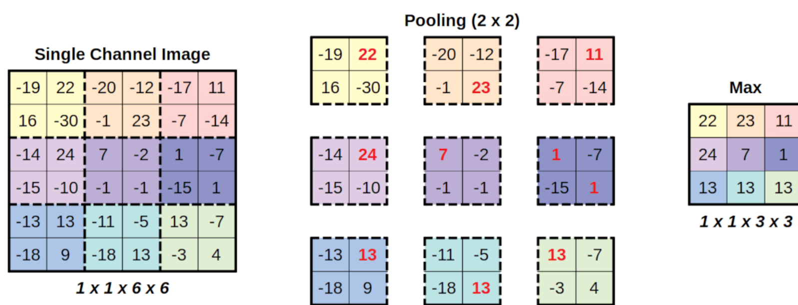

Figure 1: Example of maxpooling when $K = 2, S = 2$.

What is the output feature shape that this pooling layer produces?

Solution:
$W' = (W - K)/S + 1$
$H' = (H - K)/S + 1$
$C' = C$

(g) (**backpropagate through a 2D pool**) Consider a maxpooling layer $Y = \text{maxpool}(X)$. What is $\frac{\partial L}{\partial X}$ as an expression of X, $\frac{\partial L}{\partial Y}$? Similarly for average pooling. Note that since maxpooling is applied independently to each channel, we can without loss of generality consider the case of just a single channel. You can also try generalizing this to n-dimensional convolutions.

HINT: Start with the simplest case first, with just 2×2 maxpool, where $\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix}$. *Further assume that top left value is selected by the max operation. i.e.*

$$y_{1,1} = x_{1,1} = \max(x_{1,1}, x_{1,2}, x_{2,1}, x_{2,2}) \tag{1}$$

Then generalize to arbitrary max positions. Once you've done that, the average pooling case should be straightforward.

 All Rights Reserved. This may not be publicly shared without explicit permission.

Solution: In the simplest case, output $\mathbf{Y}$ has size 1x1. For the max pooling case,

$$\frac{\partial y_{11}}{\partial x_{i,j}} = \begin{cases} 1, & \text{if } (i,j) = 1,1 \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

Combining all four partial derivatives, we have

$$\frac{\partial L}{d\mathbf{X}} = \begin{bmatrix} dx_{11} & dx_{12} \\ dx_{21} & dx_{22} \end{bmatrix} = \begin{bmatrix} dy_{11} & 0 \\ 0 & 0 \end{bmatrix} \tag{3}$$

Where dy_{11} is $\frac{\partial L}{\partial y_1 1}$. For the average pooling case,

$$\frac{\partial y_{11}}{\partial x_{i,j}} = 1/4 \tag{4}$$

$$\frac{\partial L}{d\mathbf{X}} = \begin{bmatrix} dx_{11} & dx_{12} \\ dx_{21} & dx_{22} \end{bmatrix} = \begin{bmatrix} \frac{dy_{11}}{4} & \frac{dy_{11}}{4} \\ \frac{dy_{11}}{4} & \frac{dy_{11}}{4} \end{bmatrix} \tag{5}$$

Where dy_{ij} is $\frac{\partial L}{\partial y_i j}$.
In the general setting, for max pooling, we can notice that x and y have a one to one mapping. Each x value is involved in calculation of exactly one y value. Let $k = i//2$, $l = j//2$, where $//$ performs the floordiv operation, let

$$\delta_{i,j} = \frac{\partial y_{kl}}{\partial x_{i,j}} = \begin{cases} 1, & \text{if } x_{i,j} = \max(x_{i,j}, x_{i+1,j}, x_{i,j+1}, x_{i+1,j+1}) \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

, then

$$dx_{i,j} = \sum_{y_{mn}} dy_{mn} \frac{\partial y_{mn}}{\partial x_{i,j}} = dy_{kl}\delta_{i,j} \tag{7}$$

$\frac{\partial L}{d\mathbf{X}}$ is the matrix constructed by each $dx_{i,j}$. It will have similar pattern to the simplest case. For each 2x2 block, only one of the input pixel has gradient of magnitude one flowing back, and the rest three inputs have zero gradient.
This can be implemented in code with *argmax* in *numpy*. It is tricky to vectorize this, but simple (if slow) to implement this with nested loops.
For the average pooling general case,

$$\delta_{i,j} = \frac{\partial y_{kl}}{\partial x_{i,j}} = \frac{1}{4} \tag{8}$$

$$dx_{i,j} = \sum_{y_{mn}} dy_{mn} \frac{\partial y_{mn}}{\partial x_{i,j}} = dy_{kl}\delta_{i,j} = \frac{dy_{kl}}{4} \tag{9}$$

Average pooling distributes the gradient evenly across each 2x2 input blocks. Equivalently, $\frac{\partial L}{\partial X} = \text{AveragePool}(\frac{\partial L}{\partial Y})$, which is particularly elegant.

 All Rights Reserved. This may not be publicly shared without explicit permission.

(h) BatchNorm for CNNs is a bit different from BatchNorm for fully connected layers. The idea is that because CNN must treat a picture in the same way even if we shift the picture, we also should treat the neural activations inside the CNN the same way even if we shift the neurons. In other words, we treat the many outputs from a single convolutional kernel on a single picture as if they come from the same minibatch. That is, we apply BatchNorm per-channel, across all locations and all pictures in the minibatch.

```
import numpy as np

def batchnorm(x, gamma, beta, epsilon=1e-8):
    # Mean and variance of each feature
    mu = np.mean(x, axis=0)  # shape (N,)
    var = np.var(x, axis=0)  # shape (N,)

    # Normalize the activations
    x_hat = (x - mu) / np.sqrt(var + epsilon)  # shape (B, N)

    # Apply the linear transform
    y = gamma * x_hat + beta  # shape (B, N)

    return y

def batchnorm_cnn(x, gamma, beta, epsilon=1e-8):
    # Calculate the mean and variance for each channel.
    mean = np.mean(x, axis=(0, 1, 2), keepdims=True)
    var = np.var(x, axis=(0, 1, 2), keepdims=True)

    # Normalize the input tensor.
    x_hat = (x - mean) / np.sqrt(var + epsilon)

    # Scale and shift the normalized tensor.
    y = gamma * x_hat + beta

    return y

# Alternative implementation using reshape
# Since it is just a special case of batchnorm for cnn
def batchnorm_cnn(x, gamma, beta, epsilon=1e-8):
    B, H, W, C = x.shape
    x_reshaped = x.reshape(B * H * W, C)
    y = batchnorm(x_reshaped, gamma, beta, epsilon)
    y = y.reshape(B, H, W, C)
    return y
```

Given this, how do we implement the backward pass for BatchNorm in a CNN?

Solution: The backward pass for BatchNorm in a CNN is similar to the backward pass for BatchNorm in a fully connected layer. It suffices to reshape the input (B, H, W, C) tensor to (BHW, C), apply the BatchNorm backward pass, and then reshape the output back to the original shape.

© UCB Data C182, Fall 2024. All Rights Reserved. This may not be publicly shared without explicit permission.