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 **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 **w** and align it with the beginning of **x**. Take the dot product of **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 **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 **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/

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.

(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.

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

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

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.

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 $\boldsymbol{\nabla}_{\mathbf{y}} L$ 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?

(d) (**backpropagation through a kernel**) The above rewriting allows us to use the same method to backpropagate through a convolution layer as if it is an affine layer. However, this *as-if* fails for the parameters of the kernel itself, because of weight sharing. That is, in the equivalent affine layer, some entries are forced to always remain the same. This question derives the gradients of the loss with respect to the kernel weights.

Let's consider a convolution layer with input matrix $\mathbf{X} \in \mathbb{R}^{n \times n}$,

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,n} \end{bmatrix}, \tag{1}$$

weight matrix $\mathbf{w} \in \mathbb{R}^{k \times k}$,

$$\mathbf{w} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,k} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,1} & w_{k,2} & \cdots & w_{k,k} \end{bmatrix}, \tag{2}$$

and output matrix $\mathbf{Y} \in \mathbb{R}^{m \times m}$,

$$\mathbf{Y} = \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,m} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m,1} & y_{m,2} & \cdots & y_{m,m} \end{bmatrix}. \tag{3}$$

For simplicity, we assume the number of the input channel (of $\mathbf{X}$ is) and the number of the output channel (of output $\mathbf{Y}$) are both 1, and the convolutional layer has no padding and a stride of 1.

Then for all i, j,

$$y_{i,j} = \sum_{h=1}^{k}\sum_{l=1}^{k} x_{i+h-1,j+l-1} w_{h,l}, \tag{4}$$

or

$$\mathbf{Y} = \mathbf{X} * \mathbf{w}, \tag{5}$$

, where $*$ refers to the convolution operation. For simplicity, we omitted the bias term in this question. Suppose the final loss is $\mathcal{L}$, and the upstream gradient is $d\mathbf{Y} \in \mathbb{R}^{m,m}$,

$$d\mathbf{Y} = \begin{bmatrix} dy_{1,1} & dy_{1,2} & \cdots & dy_{1,m} \\ dy_{2,1} & dy_{2,2} & \cdots & dy_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ dy_{m,1} & dy_{m,2} & \cdots & dy_{m,m} \end{bmatrix}, \tag{6}$$

where $dy_{i,j}$ denotes $\dfrac{\partial \mathcal{L}}{\partial y_{i,j}}$.

Now, we **derive the gradient to the weight matrix** $d\mathbf{w} \in \mathbb{R}^{k,k}$,

$$d\mathbf{w} = \begin{bmatrix} dw_{1,1} & dw_{1,2} & \cdots & dw_{1,k} \\ dw_{2,1} & dw_{2,2} & \cdots & dw_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ dw_{k,1} & dw_{k,2} & \cdots & dw_{k,k} \end{bmatrix}, \tag{7}$$

where $dw_{h,l}$ denotes $\dfrac{\partial \mathcal{L}}{\partial w_{h,l}}$. Also, **derive the weight after one SGD step with a batch of a single image**.

(e) 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 $K = 2, S = 2$.

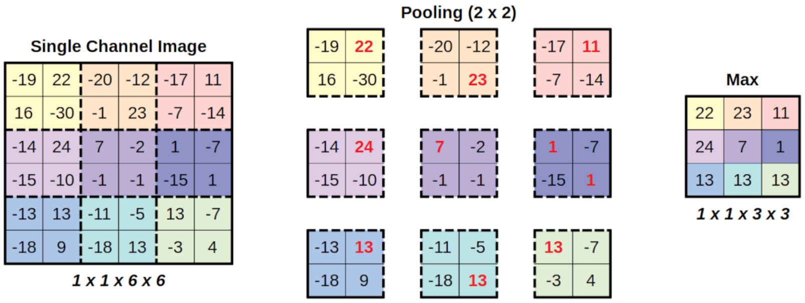

Figure 1: Example of maxpooling when $K = 2, S = 2$.

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

(f) **For a network with only 2x2 max-pooling layers (no convolution layers, no activations), what will be $d\mathbf{X} = [dx_{i,j}] = [\frac{\partial \mathcal{L}}{\partial x_{i,j}}]$? For a network with only 2x2 average-pooling layers (no convolution layers, no activations), what will be $d\mathbf{X}$?**

HINT: Start with the simplest case first, 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.*

$$y_{1,1} = x_{1,1} = \max(x_{1,1}, x_{1,2}, x_{2,1}, x_{2,2}) \tag{8}$$

Then generalize to higher dimension and arbitrary max positions.

(g) 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?