

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 networks¹ (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:

- 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.
- 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.
- Repeat until we run out of entries in \mathbf{x} .

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

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

¹Recommended 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)
- 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?
 - What if we zero-pad the input by 2?
 - 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)?
 - 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

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

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 $\nabla_{\mathbf{y}} L$ 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) 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 $K = 2$, $S = 2$.

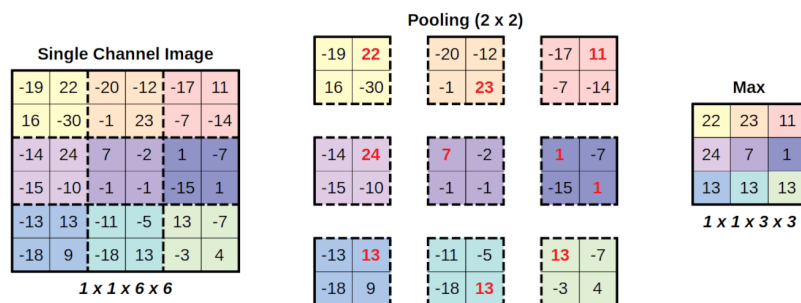


Figure 1: Example of maxpooling when $K = 2$, $S = 2$.

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

- (e) 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
```

2. Recurrent Neural Network The world is full of sequential information, from video to language modelling to time series data. In particular, we would like to model these sequences using neural networks, and solve some major types of tasks that we would like to solve with sequence models.

0.2 Types of Problems

- **One-to-one** problems take a single input x and produce a single output y . Problems like classification (takes an image as input, and produces a class label as output) and semantic segmentation (image as input, segmentation mask as output) fall under this category.
- **One-to-many** problems take a single input, and produce a sequence of output. Problems like image captioning (takes a single image as input, and produces a caption (a sequence of words) as output) fall under this category.
- **Many-to-many** problems take sequences of inputs and produce sequences of outputs. Problems like language translation (sequence of words in one language to sequence of words in another) fall under this category.

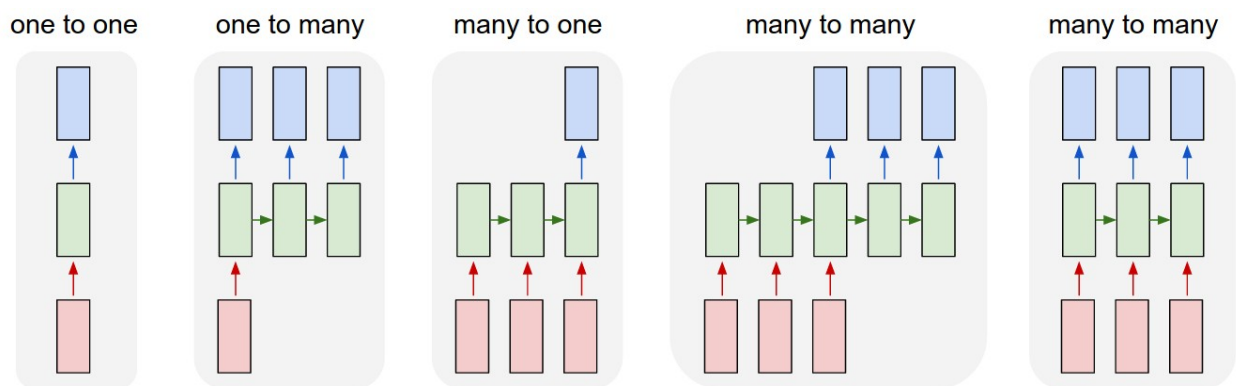


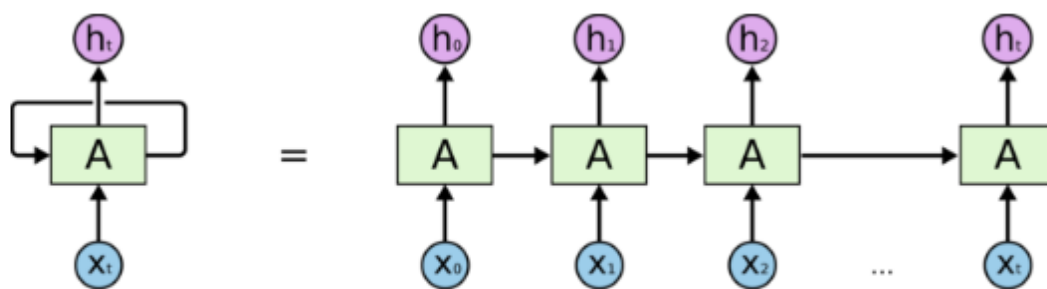
Figure 2: Types of problems we would like to solve using sequential models

0.3 Why the Recurrence?

As you read through this discussion worksheet, you don't process each word entirely on its own, but instead use your understanding from the previous words as well. Traditional neural networks do not have the

capability to use its reasoning about previous events to infer later ones. For example, if we would like to classify what is happening at every frame in a movie, this can be framed as an image classification task where the network is provided the current image. However, it is unclear how a traditional neural network should incorporate knowledge from the previous frames in the film to inform later ones.

Recurrent neural networks (RNNs) address this issue, by using the idea of “recurrent connections.” RNNs are networks with loops in them that allow information from previous inputs to persist as the network processes the future inputs. These recurrent connections allow information to propagate from “the past” (earlier in the sequence) to the future (later in the sequence).



An unrolled recurrent neural network.

Figure 3: An example of a generic recurrent neural network. This shows how to “unroll” a network through time - instead of thinking about sequence modeling as a single network with shared weights

In Figure 3, we illustrate the RNN computation as it is unrolled through time. Each $i \in \{0, \dots, t\}$ represents a new timestep in the network. By feeding in a state computed from earlier timesteps as an input together with the current input, information can persist throughout the time as the network “remembers” the past inputs it processed.

0.4 Vanilla RNN

In the following section, we will use the following notation. Denote the input sequence as $x_t \in \mathbb{R}^k$ for $t \in \{1, \dots, T\}$, and output of the network be $y_t \in \mathbb{R}^m$ for $t \in \{1, \dots, T\}$. In the following example, we construct a “vanilla” many-to-many RNN, consisting of a node that updates the hidden state h_t and produces an output y_t at each timestep with the following equations:

$$h_t = \tanh(W_{h,h}h_{t-1} + W_{x,h}x_t + B_h)$$

$$y_t = W_{h,y}h_t + B_y$$

where h_t is the time step of a hidden state (one can think of h_{t-1} as the previous hidden state), $W_{\cdot,\cdot}$ be the set of weights (for example, $W_{x,h}$ represents weight matrix that accepts an input vector and produce a new hidden state), y_t be the output at timestep t and B_h and B_y be the bias terms. We can also represent it as the diagram below,

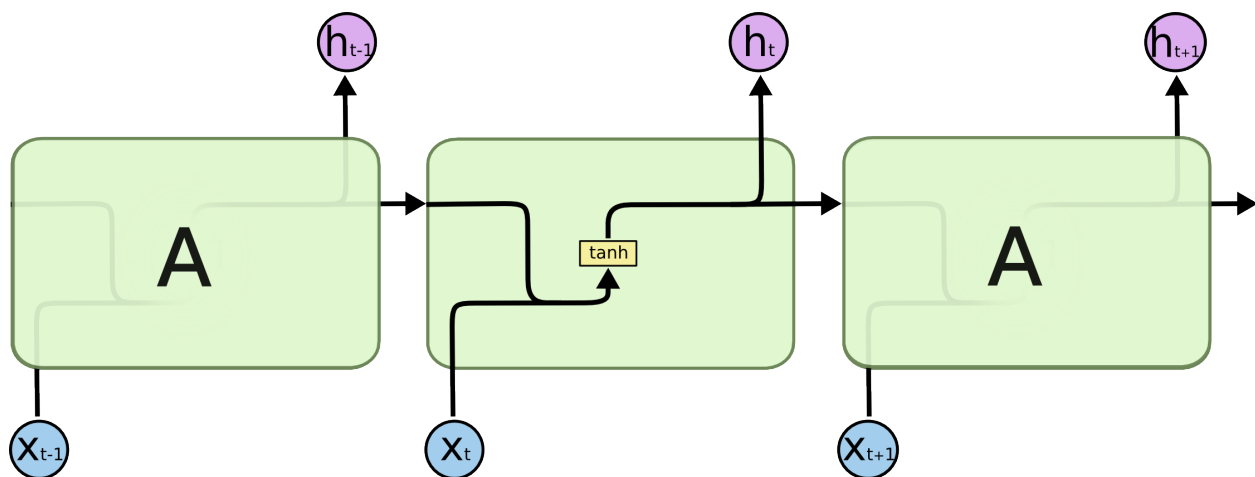


Figure 4: A simple RNN cell. As we can see by the arrows, we only pass a single hidden state from time $t - 1$ to time t

In this vanilla RNN, we update to a hidden state " h_t " based on the previous hidden state h_{t-1} and input at the current time x_t , and produce an output which that is a simple affine function of the hidden state. To compute the forward (and backward) passes of the network, we have to "unroll" the network, as shown in Figure 3. This "unrolling" process creates something that resembles a very deep feed forward network (with depth corresponding to the length of the input sequence), with shared affine parameters at each layer. Our gradient is computed by summing the losses from each time-step of the output.

Problem: Gradients in Vanilla RNN

Why are vanishing or exploding gradients an issue for RNNs?

Problem: Coding RNNs Up!

Complete the class definition, started for you below,

```
import numpy as np

class VanillaRNN:
    def __init__(self):
        self.hidden_state = np.zeros((3, 3))
        self.W_hh = np.random.randn(3, 3)
        self.W_xh = np.random.randn(3, 3)
        self.W_hy = np.random.randn(3, 3)
        self.Bh = np.random.randn(3)
        self.By = np.random.randn(3)
        self.hidden_state = np.zeros(3)

    def forward(self, x):
        # Processes the input at a single timestep and
        # updates the hidden state
        self.hidden_state = np.tanh(...)
        self.output = np.dot(...) + ...
        return self.output
```