1 Weight Sharing in CNNs

In this question, we will look at the mechanism of weight sharing in convolutions. Let's start with a 1-dimensional example. Suppose that we have a 9 dimensional input vector and compute a 1D convolution with the kernel filter that has 3 weights (parameters).

$$\mathbf{k} = [\, k_1 \quad k_2 \quad k_3 \,]^T$$
$$\mathbf{x} = [\, x_1 \quad x_2 \quad \dots \quad x_9 \,]^T$$

(a) What's the **output dimension** if we apply filter **k** with no padding and stride of 1? What's the **first element** of the output? What's the **last element**?

Solution: The output dimension is 9 - 3 + 1 = 7. The first element is $k_1x_1 + k_2x_2 + k_3x_3$ and the last element is $k_1x_7 + k_2x_8 + k_3x_9$. Expressed as a dot product, these are equivalent to $[\, x_1 \quad x_2 \quad x_3 \,]\mathbf{k}$ and $[\, x_7 \quad x_8 \quad x_9 \,]\mathbf{k}$

(b) What's the **output dimension** if we apply filter **k** with padding of size 1 and stride of 2? What's the **first element** of the output? What's the **second element**?

Solution: For the second example, the output dimension would be $(9 + 2 - 3)//2 + 1 = 5$. The first element is $k_1 0 + k_2x_1 + k_3x_2$ and the second element would be $k_1x_2 + k_2x_3 + k_3x_4$. Expressed as a dot product, these are equivalent to $[\, 0 \quad x_1 \quad x_2 \,]\mathbf{k}$ and $[\, x_2 \quad x_3 \quad x_4 \,]\mathbf{k}$

(c) Recall that CNN filters have the property of **weight sharing**, meaning that different portions of an image can share the same weight to extract the same set of features. Turns out convolution is a **linear operator** and we can express it in the form of linear layers, i.e. $\mathbf{x}' = \mathbf{K}\mathbf{x}$ (assumes that the bias term is zero).
Find K, the linear transformation matrix corresponding to the convolution applied in part (a). (***Hint***: *What is the dimension of* ***K***?)

Solution: This is a Toeplitz matrix corresponding to discrete convolution. The output size shrinks by k-1, which is 2 in this case, so K has a dimension of $\mathbb{R}^{7\times 9}$.

$$\begin{bmatrix} k_1 & k_2 & k_3 & 0 & 0 & \cdots & 0 \\ 0 & k_1 & k_2 & k_3 & 0 & \cdots & 0 \\ & & & \vdots & & & \\ 0 & \cdots & 0 & k_1 & k_2 & k_3 & 0 \\ 0 & \cdots & 0 & 0 & k_1 & k_2 & k_3 \end{bmatrix}$$

(d) Suppose that we no longer want to share weights spatially over the input, i.e. we go though the same mechanics as convolution "sliding window", but for different locations within the input, we apply different kernel. **How does this change our matrix? How many weights do we have now?**

Solution: We will still end up with a "sparse" weight matrix, but each row now represents a new kernel filter. In the case where no padding is applied, we end up with (9 - 3 + 1) * 3 = 21 non-zero weights as shown below.

$$\begin{bmatrix} k_1 & k_2 & k_3 & 0 & \cdots & 0 & 0 & 0 \\ 0 & k_4 & k_5 & k_6 & \cdots & 0 & 0 & 0 \\ & & & \vdots & & & & \\ 0 & \cdots & 0 & 0 & k_{16} & k_{17} & k_{18} & 0 \\ 0 & \cdots & 0 & 0 & 0 & k_{19} & k_{20} & k_{21} \end{bmatrix}$$

(e) Consider a 2-dimensional example with the following kernel filter and image. Using no padding and a stride of 1, compute the output, and describe the effect of this filter.

$$k = \frac{1}{4}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Solution: The output is

$$\begin{bmatrix} 3 & 4 \\ 6 & 7 \end{bmatrix}$$

This is a 2D moving average filter, and can be used to perform average pooling.

(f) We want to know the general formula for computing the output dimension of a convolution operation. Suppose we have a square input image of dimension $W \times W$ and a $K \times K$ kernel filter. If we assume stride of 1 and no padding, **what's the output dimension W'? What if we applied stride of s and padding of size p, how would the dimension change?**

Solution: For the first question, the output dimension will be $W' = W - K + 1$. This should follow from the example in part (a), since we have $W - K + 1$ contiguous sets of K indices from W total indices.

For the second one, the output dimension will be $\lfloor \frac{W+2p-K}{s} \rfloor + 1$. We include a $+2p$ because the padding is equivalent to adding extra terms on both the left and right side of the array. Dividing by s follows from the fact that we increment our counter by $+s$ instead of $+1$ when sliding along the image. The floor division operator ensures that the resulting output is an integer.

(g) Let's take what we've learnt into actual applications on image tasks. Suppose our input is a 256 by 256 RGB image. We are also given a set of 32 filters, each with kernel size of 5. Conventionally in frameworks such as PyTorch, images are 3D tensors arranged in the format of `[channels, height, width]`. In practice, it's more common to have an additional `batch_size` dimension at the front, but here we ignore that to simplify the math. What is the shape of the input tensor? What is the shape of each kernel filter? (Hint: Both your answers should have 3 dimensions.)

Solution: The input tensor is $3 \times 256 \times 256$. The kernel size is $3 \times 5 \times 5$.

(h) Now apply convolution on our image tensor with no padding and stride of 2. **What is the output tensor's dimension?** Considering all kernel filters, **how many weights do we have?** Had we not use CNN but MLP instead (with flattened image), **how many weights does that linear layer contain?** Feel free to use a calculator for this question.

Solution: Using the formula we derived in part (d), the output shape can be computed as $\lfloor \frac{256-5}{2} \rfloor + 1 = 126$, so the output dimension is $32 \times 126 \times 126$. In total, we have $32 * (3 * 5 * 5) = 2400$ weights.

Turning an input of $3 \times 256 \times 256$ to a $32 \times 126 \times 126$ tensor requires first flattening the image to a 196608 dimensional vector, and transforming it to a 508302 vector. The resulting transformation matrix will have $196608 * 508302 \approx 9.994 \times 10^{10}$.

 ©UCB CS 189/289A, Fall 2024. All Rights Reserved. This may not be publicly shared without explicit permission.

2 Self-Attention and Transformers

Recall the *attention mechanism* from sequence-to-sequence modeling, where "attention" values are computed for each input item in a sequence in order to determine how much an output should "attend" to the corresponding value at each input's position. In particular, we'll be focusing on *self-attention*, where attention values will be computed for each item in an input sequence of length n, pictorally represented by the following diagram from lecture:

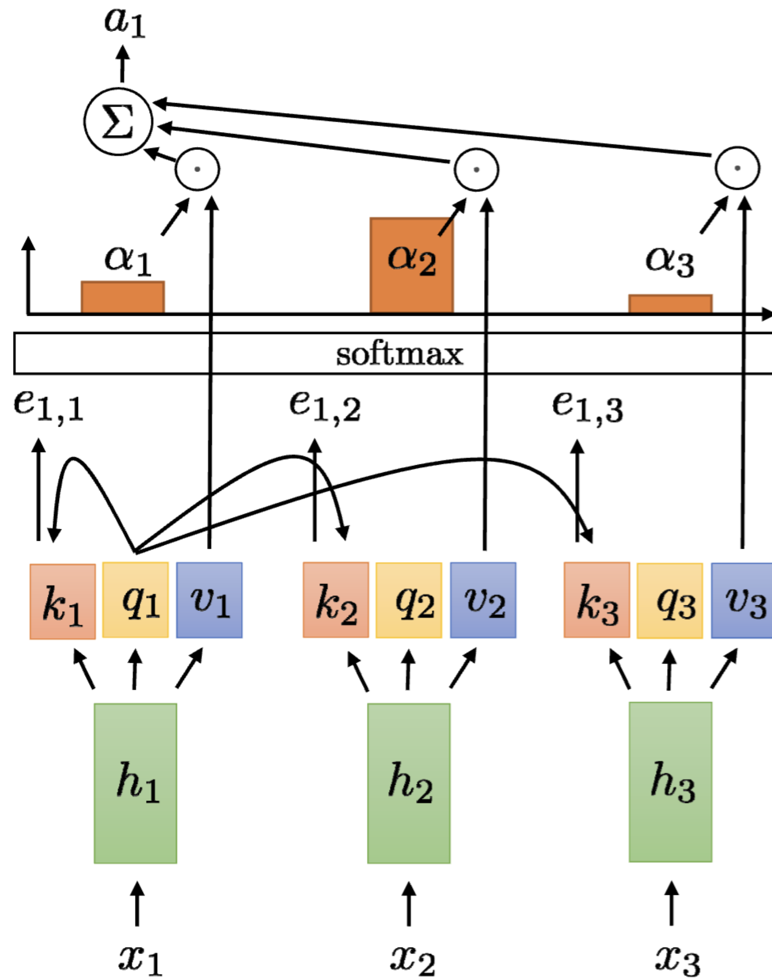

Figure 1: The self attention mechanism.

In self attention, we let the key, k, query q, and value v vectors be linear transformations of the input: $k_t = W_k h_t$, $q_t = W_q h_t$, and $v_t = W_v h_t$. For a given position in the input sequence, l, we compute the value $e_{l,t} = q_l \cdot k_t$ for every position in the input sequence. We then apply the softmax operation to each $e_{l,t}$ over all the n items in the sequence (where $t = 1 \ldots n$), which yields us values $\alpha_{l,t}$. These alpha values tell us how much to "attend" to each item in the sequence to compute our output, $a_l = \sum_t \alpha_{l,t} v_t$.

©UCB CS 189/289A, Fall 2024. All Rights Reserved. This may not be publicly shared without explicit permission.

(a) What is the runtime complexity of the aforementioned self-attention operation, in big-O? Briefly justify your answer. Assume that the values h_t have dimensionality d.

Solution: The runtime complexity is $O(d^2n + n^2d)$. First, for each input in our sequence of length n, we must compute the key, query, and value operations, which are each matrix multiplications on the order of d^2 operations, giving us a d^2n term. Then, for every input in our sequence, we compute "attention" values for the entire input sequence (n operations at each input value, of which there are n.) The dot products of our queries and keys are on the order of d operations. The softmax and output computation using the attention values are also on the order of d operations, giving us an n^2d term in our complexity.

(b) Consider the general version of the self-attention diagram, where we have multiple queries, $q_1 \dots q_n$. Write the computation for all the a values $a_1 \dots a_n$ in matrix notation.

Solution:

We can compute all of the a values by stacking our queries, keys, and values into matrices Q, K, and V, respectively. The computation then becomes:

$$a(Q, K, V) = \text{Softmax}(QK^T)V$$

(c) Next, let's consider the Transformer architecture, which applies multiple layers of self-attention to process sequential data. Recall from lecture that we need four things to get Transformers working in practice: (1) Positional Encodings, (2) Multi-Headed attention, (3) Adding non-linearities, and (4) masked decoding. In the following questions, we'll reason about different choices of positional encodings and the purpose of multi-headed attention.

Unlike Recurrent Neural Networks (RNNs), Self-attention mechanisms alone do not explicitly account for the relative position of each input in the sequence; that is, inputs far away from a given position are not treated any differently than inputs that are very close to a given position. In reality, we'd like to have some sort of encoding that allows us to take positions into account (often times, words closer to a given position are more relevant than words extremely far away, for example.)

Consider a positional encoding provided for each item in an input sequence that is *absolute*; that is, the encoding value assigned to each item in the sequence is dependent only on its absolute position in the sequence (first, second, third, etc.) Say that we use natural numbers as our absolute positional encoding: we assign the first item in the sequence a value of 1, the second item a value of 2, and so forth. What kind of issues might one anticipate with such an encoding? How might you fix this with a better absolute encoding?

Solution:

The main issue here is with scale: if we have an extremely long sequence, say of length 10000, that means that encoding values towards the end of the sequence will be orders of magnitude larger than values towards the beginning. This would make gradient-based training challenging, where different weights would have to be scaled arbitrarily differently based on the scale of the positional encodings, and may lead to unstable training. A possible fix is to

instead normalize all of the encoding values by dividing each natural number assignment by the largest number to get encodings all bounded between 0 and 1.

(d) In general, describe the potential downside that the absolute encoding approaches may have as positional encodings, and how we can improve on this with smarter approaches to positional encoding (*Hint*: think about the encodings you saw in lecture.)

Solution: The main issue with absolute encoding approaches is that they don't express *relative* relationships between inputs in a sequence. For example, consider the two semantically similar but syntactically different sentences "I went to the beach twice a week last year" and "Twice a week last year I went to the beach." In these cases, the absolute positional encodings of the phrases *I went* and *to the beach* are different, but their relative positions to one another didn't actually change. We'd like to have positional encodings that capture such relative relationships, such as the sinusoidal encoding seen in lecture.

(e) Explain the purpose and advantages of multi-head attention, or having multiple (key, query, value) pairs for every step in your input sequence. Give an example of structures in sequential problems that multi-headed attention could potentially serve useful for (*Hint*: think about structures that occur in natural language.)

Solution: Multi-Head attention allows for a single attention module to attend to multiple parts of the input sequence in different ways; that is, to have multiple different heads that can specialize in recognizing different types of structures in the input sequence. This is useful when the output is dependent on multiple inputs (such as in the case of the tense of a verb in translation). Multiple attention heads can be useful for finding multiple features in natural language, such as the start of sentences and paragraphs, relationships between subject and objects, pronouns that refer to specific nouns, and so on.

DIS5, ©UCB CS 189/289A, Fall 2024. All Rights Reserved. This may not be publicly shared without explicit permission.