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 $\mathbf{k}$ with no padding and stride of 1? What's the **first element** of the output? What's the **last element**?

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

(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* $\boldsymbol{K}$?)

(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?**

(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.

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

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

(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?**

(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.)

(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.

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:

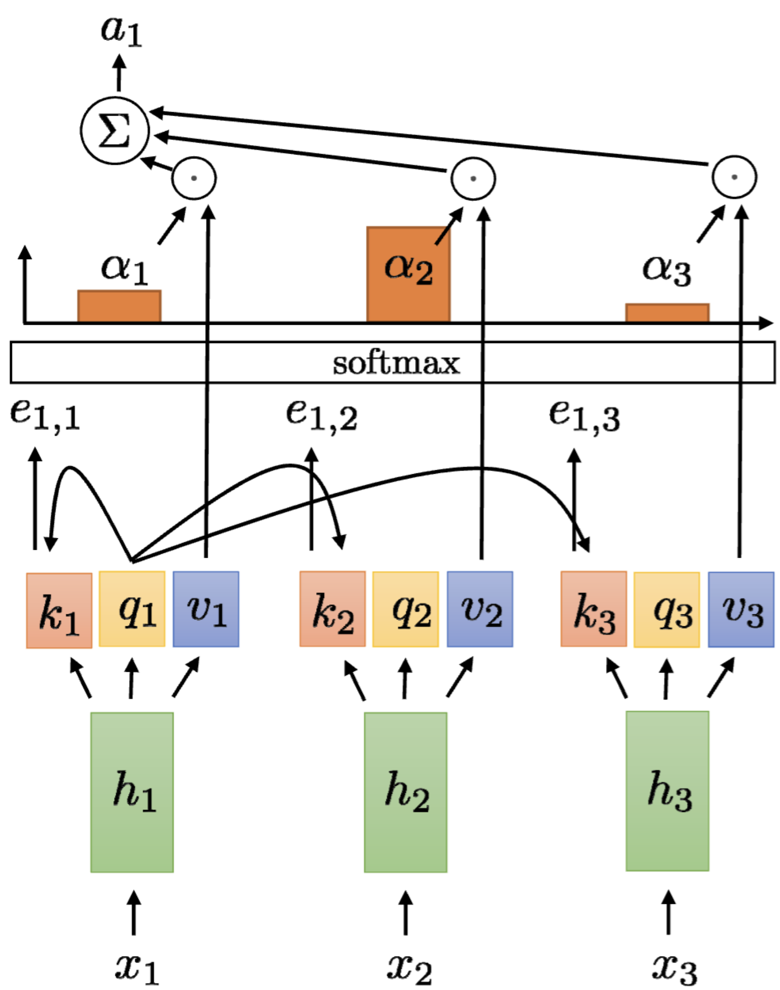

Figure 1: The self attention mechanism.

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, $a_l = \sum_t \alpha_{l,t} v_t$.

(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.

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

(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?

(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.)

(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.)

