CS 189/289A Introduction to Machine Learning
Fall 2023 Jennifer Listgarten, Jitendra Malik

DIS5

1 Convolution and Backprop

In this problem, we will walk through how discrete 2D convolution works and how we can use the backpropagation algorithm to compute derivatives through this operation.

(a) We have an image I that has three color channels I_r, I_g, I_b each of size $W \times H$ where W is the image width and H is the height. Each color channel represents the intensity of red, green, and blue for each pixel in the image. We also have a filter G with three color channels, G_r, G_g, G_b, and we represent these as a $w \times h$ matrix where w and h are the width and height of the mask. (Note that usually $w \ll W$ and $h \ll H$.) The output of the convolution operation $(I * G)[x, y]$ at point (x, y) is

$$(I * G)[x, y] = \sum_{a=0}^{w-1} \sum_{b=0}^{h-1} \sum_{c \in \{r,g,b\}} I_c[x + a, y + b] \cdot G_c[a, b]$$

For the purposes of this problem, we will assume no padding, that is, we will not concern ourselves with computing the convolution along the boundary of the image where the convolution filter would go "off image." In order to further reduce the dimension of the output, we can do a *strided* convolution in which we shift the convolutional filter by s positions instead of a single position along the image. So, in this example, if we had a stride s of 2, we would apply the convolutional filter every 2 pixels, effectively 'jumping' over one pixel and not applying the filter centered on that pixel that was strided over. In the case of no stride, we assume s to be 1.

What will the size of the output be from this convolutional operation with no striding? What will the size be if we perform a strided convolution with stride s?

Solution: In the case of no stride, the size of the output will be $(1 + W - w) \times (1 + H - h)$. In the case of a strided convolution with stride s, the resulting output will have size $\lfloor 1 + (W - w)/s \rfloor \times \lfloor 1 + (H - h)/s \rfloor$.

(b) Write pseudocode to compute the convolution of an image I with a filter G and a stride of s.

Solution:

Note that the weights in a filter are shared across all the pixels in the input. For a convolutional network, we always use weight sharing because the same filter is applied across multiple positions of an input and because it reduces model complexity, which allows for far fewer parameters than using a fully connected network.

```
for x in {0,s,2s,...,W-w}
 for y in {0,s,2s,...,H-h}
  total = 0
  for c in {r,g,b}
   window = I_c[x:x+w,y:y+h]
   conv = window * G_c // * is element-wise multiplication
   total = total + summation(conv)
  out[x/s,y/s] = total
```

The operator $*$ is element-wise multiplication of the two matrices, and summation() is the sum of all elements in the matrix.

(c) In many cases, we can handcraft convolutional filters to can produce edge detectors and other useful features. However, convolutional neural networks *learn* filters via backpropagation. These filters are often specific to the problem that we are solving. Learning these filters is a lot like learning weights in standard backpropagation, but because the same filter (with the same weights) is used in many different places, the chain rule is applied a little differently and we need to adjust the backpropagation algorithm accordingly. In short, during backpropagation each weight w in the filter has a partial derivative $\frac{\partial L}{\partial w}$ that receives contributions from every patch of image where w is applied.

Let L be the loss function or cost function our neural network is trying to minimize. Given the input image I, the convolution filter G, the convolution output $R = I * G$, and the partial derivative of the error with respect to each scalar in the output, $\frac{\partial L}{\partial R[i,j]}$, write an expression for the partial derivative of the loss with respect to a filter weight, $\frac{\partial L}{\partial G_c[x,y]}$, where $c \in \{r, g, b\}$.

Solution:

By the chain rule, we have

$$\frac{\partial L}{\partial G_c[x,y]} = \sum_{i,j} \frac{\partial L}{\partial R[i,j]} \frac{\partial R[i,j]}{\partial G_c[x,y]}.$$

From the equation for discrete convolution, the derivative for each entry $R[i, j]$ is

$$\begin{aligned}\frac{\partial R[i,j]}{\partial G_c[x,y]} &= \frac{\partial}{\partial G_c[x,y]} \sum_{c\in\{r,g,b\}} \sum_{a=0}^{w-1} \sum_{b=0}^{h-1} I_c[i+a, j+b] \cdot G_c[a,b] \\ &= I_c[i+x, j+y].\end{aligned}$$

When $i + x$ or $j + y$ go outside the boundary of the mask, we can treat the derivative as zero.

It follows that

$$\frac{\partial L}{\partial G_c[x,y]} = \sum_{i,j} \frac{\partial L}{\partial R[i,j]} I_c[i+x, j+y]. \tag{1}$$

(d) Another useful method to reduce the dimensions of a convolution operation is called max pooling. This method works similar to convolution in that we have a filter that moves around the image, but instead of multiplying the filter with a subsection of the image, we take the maximum value in the subimage captured by that filter. Max pooling can also be thought of as downsampling the image by only keeping the largest activations for each channel to 'represent' each subimage from the original input. We can also perform strided max pooling in which we shift the max pooling filter by s positions instead of a single position along the input, just like the striding operation described in part (a).

Let the output of a max pooling operation be an array R. Write a simple expression for element $R[i, j]$ of the output.

Solution:

$$R[i, j] = \max_{a=\{0,\ldots,w-1\};b=\{0,\ldots,h-1\}} (I * G)[s * i + a, s * j + b].$$

(e) Explain how we would compute derivatives through the max-pooling operation in order enable us to use backpropagation through max-pooling layers in our neural network. A plain English answer will suffice; equations are optional.

Solution: Similar to how we computed the derivatives through a convolution layer, we'll be given the derivative with respect to the output of the maxpool layer.

The gradient from the next layer is passed back only to the neuron which achieved the max. All other neurons get zero gradient.

Because maxpooling doesn't have any trainable parameters, we won't need to worry about calculating any derivatives for weights.

Once we have the derivative with respect to the input, the backprop algorithm can continue on to the layer before the maxpool by using this derivative.

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:

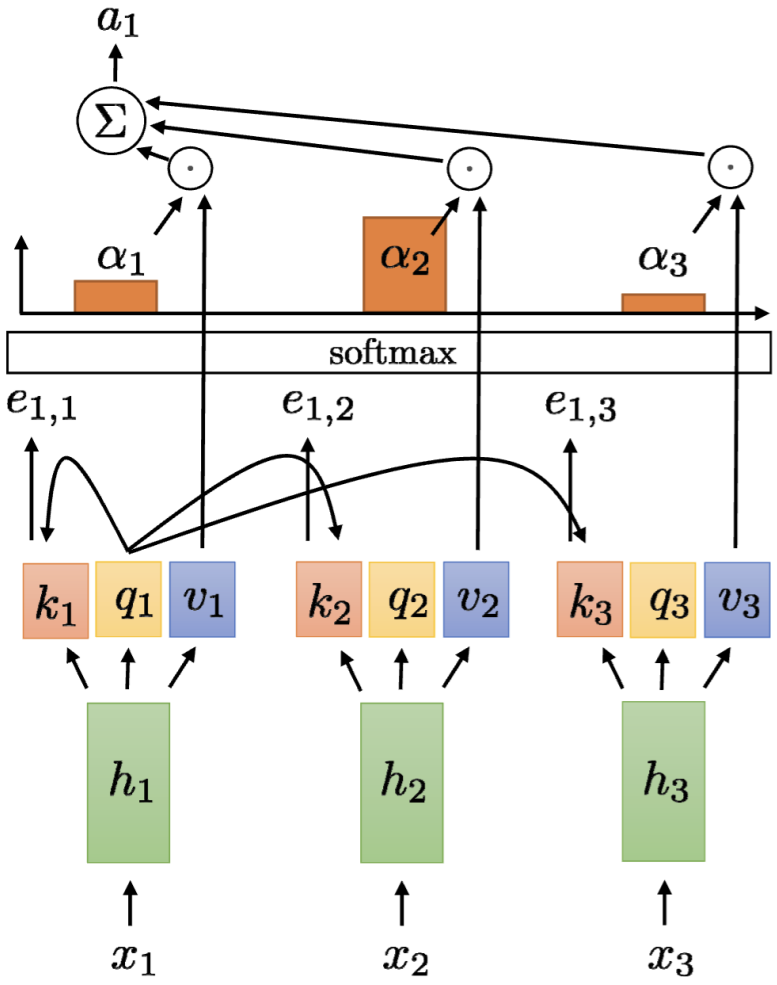

Figure 1: The self attention mechanism.

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.

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.