AUTORES: Alejandro Moya, Isaac Reyes & José Guzmán

11.6 Self-Attention and Positional Encoding

`sec_self-attention-and-positional-encoding`

In deep learning, we often use CNNs or RNNs to encode sequences. Now with attention mechanisms in mind, imagine feeding a sequence of tokens into an attention mechanism such that at every step, each token has its own query, keys, and values. Here, when computing the value of a token's representation at the next layer, the token can attend (via its query vector) to any other's token (matching based on their key vectors). Using the full set of query-key compatibility scores, we can compute, for each token, a representation by building the appropriate weighted sum over the other tokens. Because every token is attending to each other token (unlike the case where decoder steps attend to encoder steps), such architectures are typically described as *self-attention* models :cite: `Lin.Feng.Santos.ea.2017,Vaswani.Shazeer.Parmar.ea.2017` , and elsewhere described as *intra-attention* model :cite: `Cheng.Dong.Lapata.2016,Parikh.Tackstrom.Das.ea.2016,Paulus.Xiong.Socher.20`
In this section, we will discuss sequence encoding using self-attention, including using additional information for the sequence order.

Importamos las librerias necesarias

In [1]:
```
use strict;
use warnings;
use Data::Dump qw(dump);
use List::Util qw(zip min max sum);
use d2l;
IPerl->load_plugin('Chart::Plotly');
```

11.6.1. Self-Attention

Given a sequence of input tokens $\mathbf{x}_1, \ldots, \mathbf{x}_n$ where any $\mathbf{x}_i \in \mathbb{R}^d$ $(1 \leq i \leq n)$, its self-attention outputs a sequence of the same length $\mathbf{y}_1, \ldots, \mathbf{y}_n$, where

$$\mathbf{y}_i = f(\mathbf{x}_i, (\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_n, \mathbf{x}_n)) \in \mathbb{R}^d$$

according to the definition of attention pooling in :eqref: `eq_attention_pooling` . Using multi-head attention, the following code snippet computes the self-attention of a tensor with shape (batch size, number of time steps or sequence length in tokens, d). The output tensor has the same shape.

In [2]:
```
my ($num_hiddens, $num_heads) = (100, 5);
my $attention = new d2l::MultiHeadAttention($num_hiddens, $num_heads,
$attention->initialize();
my ($batch_size, $num_queries, $valid_lens) = (2, 4, mx->nd->array([3,
my $X = mx->nd->ones([$batch_size, $num_queries, $num_hiddens]);
d2l->check_shape($attention->forward($X, $X, $X, $valid_lens), [$batch_
```

Out[2]:
```
1
```

11.6.2. Comparing CNNs, RNNs, and Self-Attention

:label: `subsec_cnn-rnn-self-attention`

Let's compare architectures for mapping a sequence of n tokens to another one of equal length, where each input or output token is represented by a d-dimensional vector. Specifically, we will consider CNNs, RNNs, and self-attention. We will compare their computational complexity, sequential operations, and maximum path lengths. Note that sequential operations prevent parallel computation, while a shorter path between any combination of sequence positions makes it easier to learn long-range dependencies within the sequence :cite: `Hochreiter.Bengio.Frasconi.ea.2001` .

Comparing CNN (padding tokens are omitted), RNN, and self-attention architectures.

:label: `fig_cnn-rnn-self-attention`

Let's regard any text sequence as a "one-dimensional image". Similarly, one-dimensional CNNs can process local features such as n-grams in text. Given a sequence of length n, consider a convolutional layer whose kernel size is k, and whose numbers of input and output channels are both d. The computational complexity of the convolutional layer is $\mathcal{O}(knd^2)$. As :numref: `fig_cnn-rnn-self-attention` shows, CNNs are hierarchical, so there are $\mathcal{O}(1)$ sequential operations and the maximum path length is $\mathcal{O}(n/k)$. For example, $\mathbf{x}_1$ and $\mathbf{x}_5$ are within the receptive field of a two-layer CNN with kernel size 3 in :numref: `fig_cnn-rnn-self-attention` .

When updating the hidden state of RNNs, multiplication of the $d \times d$ weight matrix and the d-dimensional hidden state has a computational complexity of $\mathcal{O}(d^2)$. Since the sequence length is n, the computational complexity of the recurrent layer is $\mathcal{O}(nd^2)$. According to :numref: `fig_cnn-rnn-self-attention` , there are $\mathcal{O}(n)$ sequential operations that cannot be parallelized and the maximum path length is also $\mathcal{O}(n)$.

In self-attention, the queries, keys, and values are all $n \times d$ matrices. Consider the scaled dot product attention in :eqref: `eq_softmax_QK_V` , where an $n \times d$ matrix is multiplied by a $d \times n$ matrix, then the output $n \times n$ matrix is multiplied by an $n \times d$ matrix. As a result, the self-attention has a $\mathcal{O}(n^2 d)$ computational complexity. As we can see from :numref: `fig_cnn-rnn-self-attention` , each token is directly connected to any other token via self-attention. Therefore, computation can be parallel with $\mathcal{O}(1)$ sequential operations and the maximum path length is also $\mathcal{O}(1)$.

All in all, both CNNs and self-attention enjoy parallel computation and self-attention has the shortest maximum path length. However, the quadratic computational complexity with respect to

11.6.3 Positional Encoding

:label: `subsec_positional-encoding`

Unlike RNNs, which recurrently process tokens of a sequence one-by-one, self-attention ditches sequential operations in favor of parallel computation. Note that self-attention by itself does not preserve the order of the sequence. What do we do if it really matters that the model knows in which order the input sequence arrived?

The dominant approach for preserving information about the order of tokens is to represent this to the model as an additional input associated with each token. These inputs are called *positional encodings*, and they can either be learned or fixed *a priori*. We now describe a simple scheme for fixed positional encodings based on sine and cosine functions :cite: `Vaswani.Shazeer.Parmar.ea.2017` .

Suppose that the input representation $\mathbf{X} \in \mathbb{R}^{n\times d}$ contains the d-dimensional embeddings for n tokens of a sequence. The positional encoding outputs $\mathbf{X} + \mathbf{P}$ using a positional embedding matrix $\mathbf{P} \in \mathbb{R}^{n\times d}$ of the same shape, whose element on the i^th row and the $(2j)^\mathrm{th}$ or the $(2j + 1)^\mathrm{th}$ column is

$$\begin{aligned} p_{i, 2j} &= \sin\left(\frac{i}{10000^{2j/d}}\right),\\p_{i, 2j+1} &= \cos\left(\frac{i}{10000^{2j/d}}\right).\end{aligned}$$

:eqlabel: `eq_positional-encoding-def`

At first glance, this trigonometric function design looks weird. Before we give explanations of this design, let's first implement it in the following `PositionalEncoding` class.

In [3]:

```
package  PositionalEncoding{
      use base qw(AI::MXNet::Gluon::Block); #@save
      use List::Util qw(zip);
      sub new {
        my ($class, $num_hiddens,$dropout,%args) = (splice(@_,0,3), d2
        my  $self = $class->SUPER::new();
        $self->{dropout} = mx->gluon->nn->Dropout($dropout);
        map {$self->register_child($self->{$_})} ('dropout');
          my $X = mx->nd->arange(stop=>$args{max_len})->reshape([-1,1]
            (1000 ** (mx->nd->arange(start=>0, stop=>$num_hiddens, ste
          $self->{P} = mx->nd->concat(
            (map { $_->expand_dims(1) }
                map { @$_ }
                    zip \@{mx->nd->sin($X)->T}, \@{mx->nd->cos($X)->T}
            ),
            dim => 1
          )->expand_dims(0);

        return bless ($self, $class);
    }
    sub forward {
        my ($self, $X) = @_;
         $self->{P}->slice('X', [0, $X->shape->[1] -1], 'X')->as_in_co

        return $self->{dropout}->($X);
    }
1;
}
```

Out[3]: 1

In [4]:

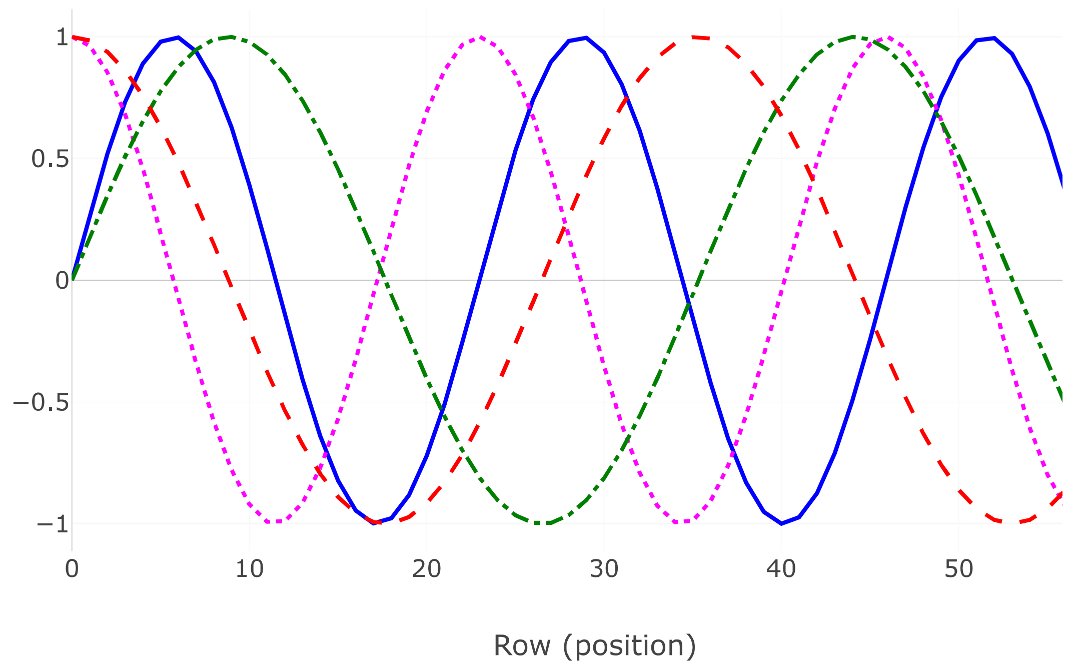
```
my ($encoding_dim ,$num_steps) = (32 , 60);
my $pos_encoding = new PositionalEncoding($encoding_dim, 0);
$pos_encoding->initialize();
my $X = $pos_encoding->forward(mx->nd->zeros([1, $num_steps,$encoding_
my $P = $pos_encoding->{P}->slice('X', [0, $X->shape->[1] -1], 'X');
```

Out[4]:

```
<AI::MXNet::NDArray 1x60x32 @cpu(0)>
```

In the positional embedding matrix $\mathbf{P}$, [**rows correspond to positions within a sequence and columns represent different positional encoding dimensions**]. In the example below, we can see that the 6^{th} and the 7^{th} columns of the positional embedding matrix have a higher frequency than the 8^{th} and the 9^{th} columns. The offset between the 6^{th} and the 7^{th} (same for the 8^{th} and the 9^{th}) columns is due to the alternation of sine and cosine functions.

In [5]:
```
#Graficamos:
my $squeeze_p = $P->slice(0, 'X', [6, 9])->squeeze(0);
d2l->plot(mx->nd->arange(stop => $num_steps), $squeeze_p->T,
          xlabel => 'Row (position)', figsize => [6, 2.5],
          legend => [map { "Col $_" } 6..9]);
```



11.6.3.1. Absolute Positional Information

To see how the monotonically decreased frequency along the encoding dimension relates to absolute positional information, let's print out [**the binary representations**] of $0, 1, \ldots, 7$. As we can see, the lowest bit, the second-lowest bit, and the third-lowest bit alternate on every number, every two numbers, and every four numbers, respectively.

In [6]:

```
for my $i (0..7) {
    printf "%d in binary is %03b\n", $i, $i;
}
my $P_expanded = $P->slice([0])->expand_dims(axis=>0);
```

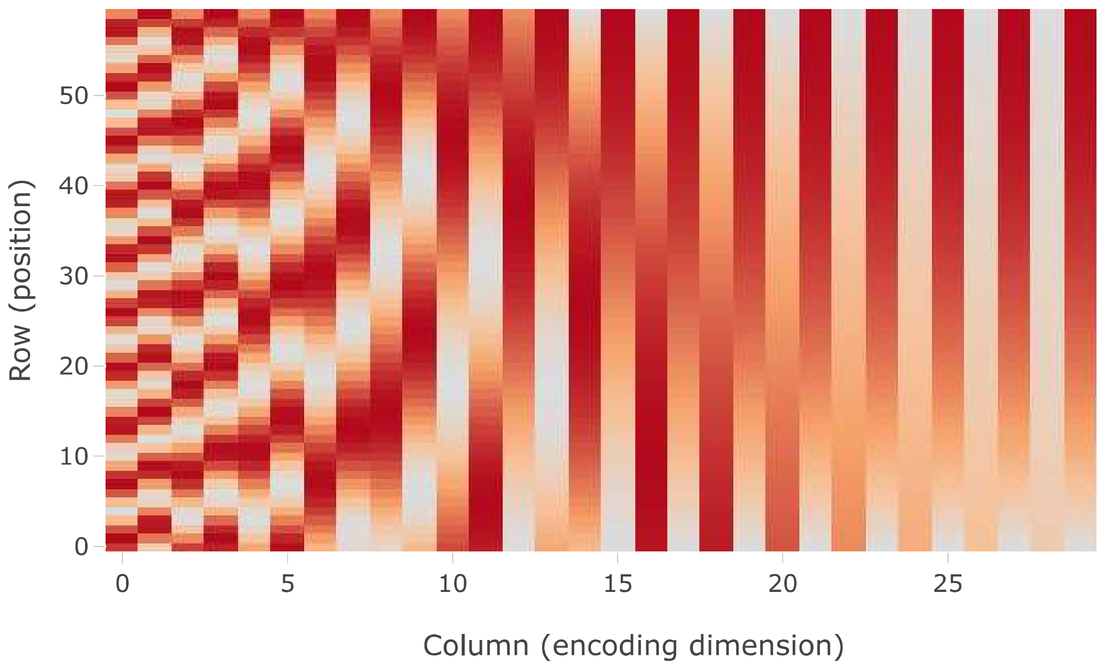
```
0 in binary is 000
1 in binary is 001
2 in binary is 010
3 in binary is 011
4 in binary is 100
5 in binary is 101
6 in binary is 110
7 in binary is 111
```

Out[6]:

```
<AI::MXNet::NDArray 1x1x60x32 @cpu(0)>
```

In binary representations, a higher bit has a lower frequency than a lower bit. Similarly, as demonstrated in the heat map below, [**the positional encoding decreases frequencies along the encoding dimension**] by using trigonometric functions. Since the outputs are float numbers, such continuous representations are more space-efficient than binary representations.

In [7]:
```
d2l->show_heatmaps(
    $P_expanded,
    xlabel => 'Column (encoding dimension)',
    ylabel => 'Row (position)',
    figsize => [3.5, 4],
    cmap => 'Reds'
);
```



11.6.3.2 Relative Positional Information

Besides capturing absolute positional information, the above positional encoding also allows a model to easily learn to attend by relative positions. This is because for any fixed position offset δ, the positional encoding at position $i + \delta$ can be represented by a linear projection of that at position i.

This projection can be explained mathematically. Denoting $\omega_j = 1/10000^{2j/d}$, any pair of $(p_{i,2j}, p_{i,2j+1})$ in :eqref: `eq_positional-encoding-def` can be linearly projected to $(p_{i+\delta,2j}, p_{i+\delta,2j+1})$ for any fixed offset δ:

$$\begin{bmatrix} \cos(\delta\omega_j) & \sin(\delta\omega_j) \\ -\sin(\delta\omega_j) & \cos(\delta\omega_j) \end{bmatrix} \begin{bmatrix} p_{i,2j} \\ p_{i,2j+1} \end{bmatrix} = \begin{bmatrix} \cos(\delta\omega_j)\sin(i\omega_j) + \sin(\delta\omega_j)\cos(i\omega_j) \\ -\sin(\delta\omega_j)\sin(i\omega_j) + \cos(\delta\omega_j)\cos(i\omega_j) \end{bmatrix}$$

Summary

In self-attention, the queries, keys, and values all come from the same place. Both CNNs and self-attention enjoy parallel computation and self-attention has the shortest maximum path length. However, the quadratic computational complexity with respect to the sequence length makes self-attention prohibitively slow for very long sequences. To use the sequence order information, we can inject absolute or relative positional information by adding positional encoding to the input representations.

Exercises

1. Suppose that we design a deep architecture to represent a sequence by stacking self-attention layers with positional encoding. What could the possible issues be?
2. Can you design a learnable positional encoding method?
3. Can we assign different learned embeddings according to different offsets between queries and keys that are compared in self-attention? Hint: you may refer to relative position embeddings :cite: `shaw2018self,huang2018music` .

In [ ]:

```

```