Video frame prediction using self-attention

Course: Advanced machine learning using neural networks.
Contact: Daniel Midtvedt.

---

1 Background

In many applications where deep learning is employed, the order of the input data matters for its interpretation. Examples of this includes natural language processing, where the order of the words is important in order to decipher the meaning of a sentence, and in time series analysis the order of the data is important to predict the future behaviour. In order to analyse such data, the deep learning models need a notion of time or sequence position. Traditionally, this has been solved by the use of recurrent neural networks (RNN) in which the data is fed sequentially through the network and the output from the network for a given sequence position depends on the network output at previous positions.

However, due to the sequential nature of the network analysis such approaches are typically slow, and long range dependencies in the data are difficult to capture. In order to overcome these limitations, *transformer networks* based on the concept of *self-attention* have emerged as an alternative to RNN in almost all areas of sequence analysis. In a transformer network, the entire data is fed to the network at once, allowing for parallelization and considerable speedup of both training and inference. At each step, an *attention gate* encodes how much weight should be placed at a certain input in the analysis of the sequence. However, since all steps in the sequence are analyzed in parallel, the notion of time has to be encoded into the data itself, through *time embedding*.

In this homework, you will build a auto-encoder with self-attention for predicting the next frame in a video sequence. The videos that you will analyze are simulated sequences of non-interacting spherical particles bouncing in a box.

2 Structure of the network

The structure of the network that you will implement is inspired by [1], where the concept of a transformer network was first introduced. Here, that network is adapted to the problem of video frame prediction.

First, each frame of the video sequence is downsampled through a series of convolutional steps to a one-dimensional representation of the original image of length n, such that the dimension of the output from the encoder when applied to a sequence of length t is (n, t). Next, these representations of the original frames are analyzed in a transformer encoder, including a time embedding layer and multiple attention gates. The output of this encoder is another one-dimensional array of length n. Finally, a convolutional decoder converts the output of the encoder to a new image of same shape as the input video frames.

2.1 Time embedding

To encode the notion of time into the data, we will make use of the idea introduced in [2]. The main idea is that a meaningful representation of time must be able to represent both a linear progression as well as periodically returning patterns. To accommodate this, it was suggested to represent time as a d-dimensional vector,

$$\vec{t_i}(\tau) = \mathcal{F}_i(\omega_i \tau + \varphi_i), \tag{1}$$

where τ is the input time series, the function $\mathcal{F}_i(x) = \sin(x)$ for $0 < i < d - 1$ and $\mathcal{F}_i(x) = x$ for $i = 0$. The frequencies ω_i and phases φ_i are learned parameters. In the current context, the time series τ is represented by the output of the convolutional decoder. Below is the code for implementing time2vec with a single time-periodic component using tensorflow.

```
class Time2Vector(Layer):
  def __init__(self, seq_len, **kwargs):
    super(Time2Vector, self).__init__()
    self.seq_len = seq_len

  def build(self, input_shape):
    self.weights_linear = self.add_weight(name='weight_linear',
                                    shape=(int(self.seq_len),),
                                    initializer='uniform',
                                    trainable=True)

    self.bias_linear = self.add_weight(name='bias_linear',
                                    shape=(int(self.seq_len),),
                                    initializer='uniform',
                                    trainable=True)

    self.weights_periodic = self.add_weight(name='weight_periodic',
                                    shape=(int(self.seq_len),),
                                    initializer='uniform',
                                    trainable=True)

    self.bias_periodic = self.add_weight(name='bias_periodic',
                                    shape=(int(self.seq_len),),
                                    initializer='uniform',
                                    trainable=True)

  def call(self, x):
    x = tf.math.reduce_mean(x[:,:,:], axis=-1)
    time_linear = self.weights_linear * x + self.bias_linear
    time_linear = tf.expand_dims(time_linear, axis=-1)

    time_periodic = tf.math.sin(tf.multiply(x, self.weights_periodic) + self.bias_period
    time_periodic = tf.expand_dims(time_periodic, axis=-1)
    return tf.concat([time_linear, time_periodic], axis=-1)
```

2.2 Attention gates

The output from the time embedding layer is appended to the outputs from the convolutional encoder to give a representation of the initial video sequence having dimension $(n + d, t)$. The next step is to apply *attention gates* to this representation. The purpose of the attention gates is to let the network learn how much emphasis should be placed on the data in a certain video frame when predicting the next frame. One way to achieve this is via the *scaled dot-product attention*, in which the $(n + d, t)$-dimensional representation of the video is transformed to three new representations q, k and v, of dimensions (d_k, t) (for q, k) and (d_v, t) (for v), through three densely connected layers. These three representaions are known as *query*, *key* and *value*, respectively, and form the inputs to the attention gate. The output of the attention gate is calculated as

$$Attention(q, k, v) = \text{softmax}(q \cdot k^{\mathrm{T}} / \sqrt{d_k})v, \tag{2}$$

to output the attention weights of a given input sequence. Often, multiple such attention gates work in parallel in order for the network to attend to multiple parts of the sequence simultaneously. This is called *multi-headed attention.* The outputs from the parallel attention gates are concatenated and passed through a densely connected layer to combine the information of all attention gates in a multi-headed attention layer.

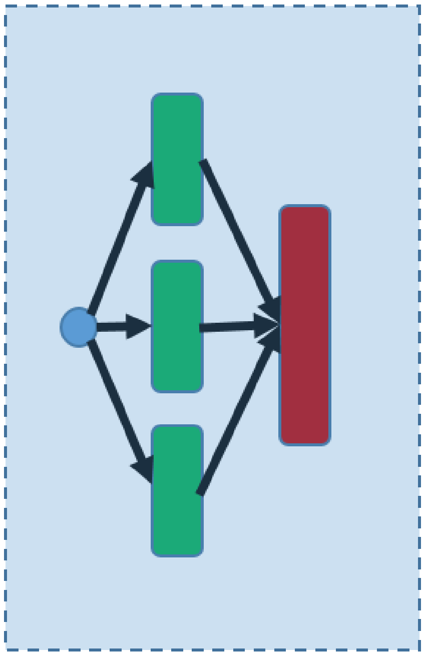

Figure 1: Attention gate. The input is passed through three parallel dense layers (green), and the output from these layers are passed through a scaled dot product attention layer (red).

Below is an example implementation of a single attention gate:

```
class SingleAttention(Layer):
  def __init__(self, d_k, d_v):
    super(SingleAttention, self).__init__()
    self.d_k = d_k
    self.d_v = d_v

  def build(self, input_shape):
    self.query = Dense(self.d_k, input_shape=input_shape,
    kernel_initializer='glorot_uniform', bias_initializer='glorot_uniform')
    self.key = Dense(self.d_k, input_shape=input_shape,
    kernel_initializer='glorot_uniform', bias_initializer='glorot_uniform')
    self.value = Dense(self.d_v, input_shape=input_shape,
    kernel_initializer='glorot_uniform', bias_initializer='glorot_uniform')

  def call(self, inputs): # inputs = (in_seq, in_seq, in_seq)
    q = self.query(inputs[0])
    k = self.key(inputs[1])

    attn_weights = tf.matmul(q, k, transpose_b=True)
    attn_weights = tf.map_fn(lambda x: x/np.sqrt(self.d_k), attn_weights)
    attn_weights = tf.nn.softmax(attn_weights, axis=-1)

    v = self.value(inputs[2])
    attn_out = tf.matmul(attn_weights, v)
    return attn_out
```

2.3 Transformer encoder

From the multi-headed attention gate, the transformer encoder is constructed as shown in the diagram below. Multiple such transformer encoders can be stacked to increase the depth of the network.

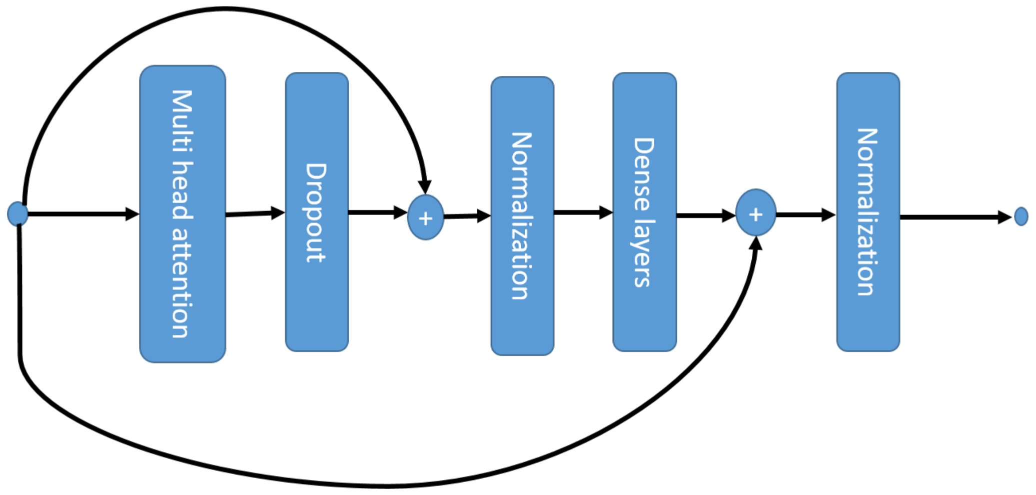

Figure 2: The transformer encoder

3 Task

Your task is to implement a transformer network as sketched above for the analysis of video sequences.

- 1: The first step is to pretrain the convolution encoder and decoder as an autoencoder. In other words, during this step of the training, frames are passed through the convolutional encoder individually, and the output from the encoder is passed directly to the decoder. The job of the decoder is to reconstruct the original video frame from the downsampled representation provided by the convolutional encoder. How does the performance of the autoencoder depend on the number of features extracted at the bottleneck? **3p**

- 2: After pretraining decoder/encoder pair, train the transformer encoder to predict the downsampled representation of the next video frame. Remember to keep the pretrained weights of the convolutional encoder/decoder fixed at this stage! Experiment with different number of transformer layers of different sizes. Do these parameters influence the performance of the network? How far ahead can the transformer predict the sequence of frames? **4p**

- 3: Benchmark the transformer encoder against a standard recurrent neural networks by replacing the transformer encoders by one or multiple long short-term memory (LSTM) layers, and train these layers in the same way as in task 2. How far ahead can an LSTM-based encoder predict the frames correctly? **3p**

4 Examination

For the examination you should present your own code and your result to the tasks above. You should be able to describe the functionality of transformers and how they differ from long-short term memory gates. Make sure to be well prepared as time is short.

References

[1] Vaswani, A. *et al.* Attention is all you need. *Advances in Neural Information Processing Systems* **2017-Decem**, 5999–6009 (2017). URL https://arxiv.org/abs/1706.03762v5. 1706.03762.

[2] Kazemi, S. M. *et al.* Time2Vec: Learning a Vector Representation of Time. *arXiv* (2019). URL http://arxiv.org/abs/1907.05321. 1907.05321.