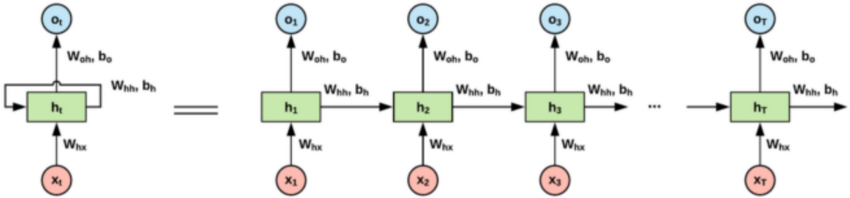
Introduction to Statistical Learning

Omid Safarzadeh

January 19, 2022

Table of contents

RNN Structure



This structure allows RNNs to link information from earlier steps to a present step. However, if the RNN is very deep, meaning that it has many layers, it will be prone to vanishing or exploding gradients.

RNN model

Given a sequence of $X = (x_1, x_2, .., x_T)$ inputs , RNN estimates a sequence of $Y = (y_1, y_2, .., y_T)$ outputs by iterating the following equation:

$$h_t = Sigmoid(W^{hx} x_t + W^{hh} h_{t-1})$$
$$y_t = W^{yh} h_t$$

Problem? T should be fixed in both input and output sequences. Solution? map input sequence to a fixed length vector and then map that vector to target sequence with another RNN.

Encoder - Decoder Architecture

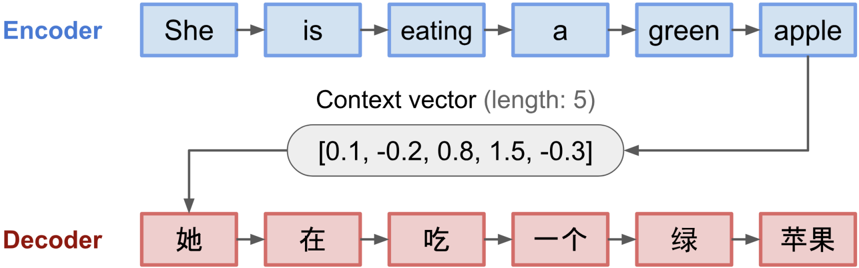
- proposed by (Cho et. al., 2014)
- The **encoder** processes each item in the input sequence, it compiles the information it captures into **context vector**. After processing the entire input sequence, the encoder sends the context vector to the **decoder**, which begins producing the output sequence item by item.

Example Example

Sequence to Sequence models

- A Sequence to Sequence model,, is a model that takes a sequence of items (words, letters, features , ...) and outputs another sequence of items (with any length for both input and output).
- Applications : machine translation between multiple languages in either text or audio, question-answer dialog generation, or even parsing sentences into grammar trees.
- seq2seq is introduced by (Sutskever, et al. 2014) replaced RNN with LSTM models. Example

NMT Architecture



Use RNN networks (LSTM, GRU,...) for Encoder and Decoder sides. Also we need a Context Vector. (here fixed length).
Context Vector is also called sentence embedding or "thought" vector which represents a good summary of the meaning of the whole source sequence.

Attention Mechanism

Attention

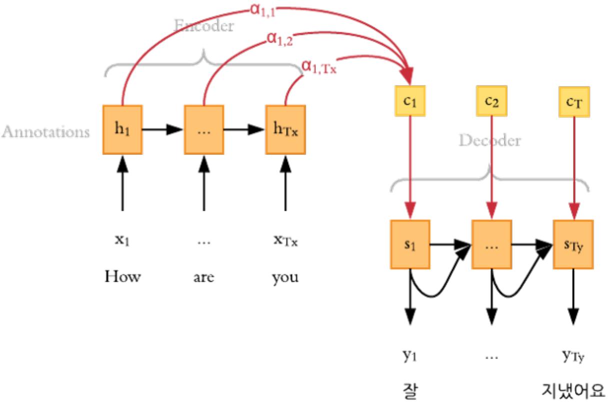
" Definition "

attention can be interpreted as a vector of importance weights: in order to predict or infer one element, such as a pixel in an image or a word in a sentence, we estimate using the attention vector how strongly it is correlated with (“attends to”) other elements and take the sum of their values weighted by the attention vector as the approximation of the target.

Ref: Lilian Weng

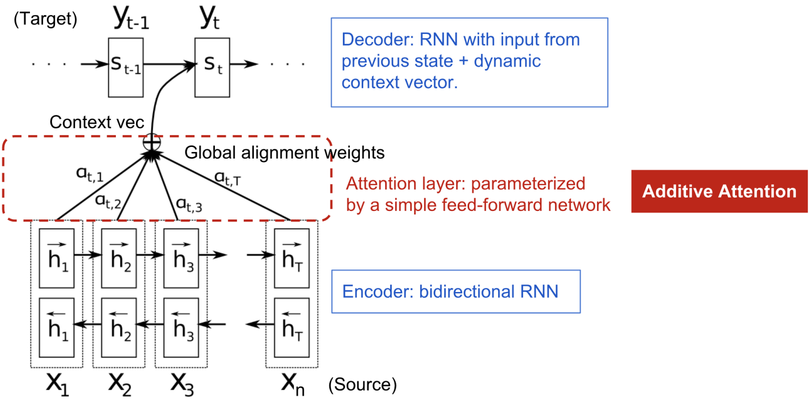
Bottleneck problem

The last hidden state is actually the context we pass along to the decoder and it is big challenge for the models to deal with long sentences.



Attention

Attention allows the model to focus on the relevant parts of the input sequence as needed. First, the encoder passes a lot more data to the decoder. Instead of passing the last hidden state of the encoding stage, the encoder passes all the hidden states to the decoder:



Bahdanau,et al. (2014).

Attention

$$p(y) = \prod_{t}^{T} x(p(y_t|y_1, ..., y_{t-1}), c)$$

$$p((y_t|y_1, ..., y_{t-1}), c) = g(y_{t-1}, s_t, c)$$

$$s_t = f(s_{i-1}, y_{i-1}, c_i)$$

$$c_i = \sum_{t}^{T} x(\alpha_{i_j} h_j)$$

$$\alpha_{i_j} = \frac{exp(e_{i_j})}{\sum_k^T x(e_{i_j})}$$

$$e_{i_j} = a(s_{i-1}, h_j)$$

C: context vector of output y_t
h: hidden state ; α_{i_j}: Softmax

attention-Encoder

First, the encoder passes a lot more data to the decoder. Instead of passing the last hidden state of the encoding stage, the encoder passes all the hidden states to the decoder Example

attention -Decoder

Second : the decoder does the following:

- Look at the set of encoder hidden states it received – each encoder hidden states is most associated with a certain word in the input sentence
- Give each hidden states a score (lets ignore how the scoring is done for now)
- Multiply each hidden states by its softmaxed score, thus amplifying hidden states with high scores, and drowning out hidden states with low scores

Example

Attention - all steps:

1. The attention decoder RNN takes in the embedding of the END token, and an initial decoder hidden state.
2. The RNN processes its inputs, producing an output and a new hidden state vector (h4). The output is discarded.
3. Attention Step: We use the encoder hidden states and the h4 vector to calculate a context vector (C4) for this time step.
4. We concatenate h4 and C4 into one vector.
5. We pass this vector through a feedforward neural network (one trained jointly with the model).
6. The output of the feedforward neural networks indicates the output word of this time step.
7. Repeat for the next time steps

Example

Categories of Attention Mechanisms

- Self – Attention
- Local vs Global Attention
- Hard vs Soft Attention

Self – attention **(intra-attention)**

- Machine reading
- Abstractive Summarization
- Image description generation

Ref:Loung 2015

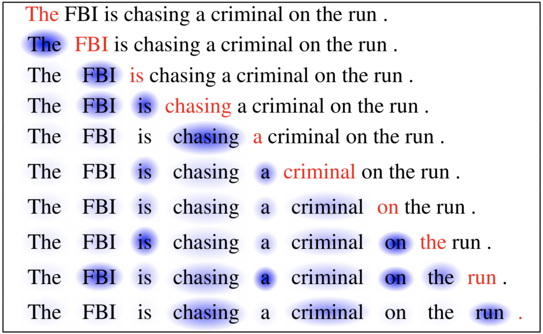


Image Captioning with Attention

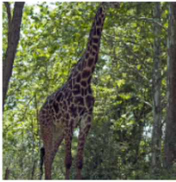
A woman is throwing a frisbee in a park.

A dog is standing on a hardwood floor.

A stop sign is
mountain in t

A little girl sitting on a bed with
a teddy bear.

A group of people sitting on a boat
in the water.

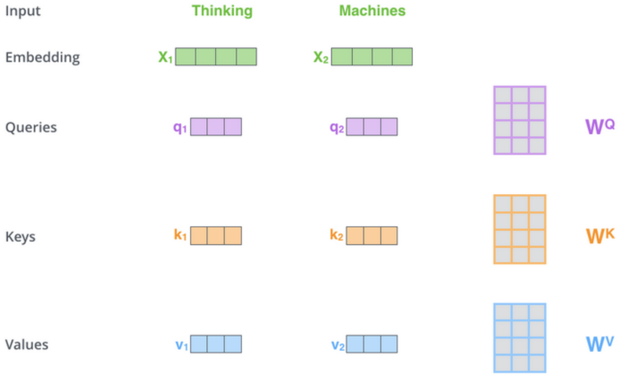


A giraffe standi
trees in the bac

Transformers

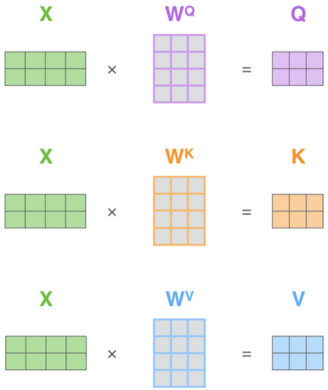
Self-attention mechanism

Embedding size: 512 , q_1 (1*64) , key(1*64) , value(1*64)



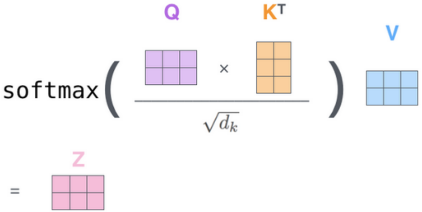
Matrix Notation

W^Q, W^K, W^V : have **512*64** weights to learn.
X dimensions is **(maxlen * 512)**. So X represents a sentence with two words.



Calculate Self Attention Score

- $q_1 * k_1$ score for position 1
- $q_1 * k_2$ score for position 2
- $d_k = 64$



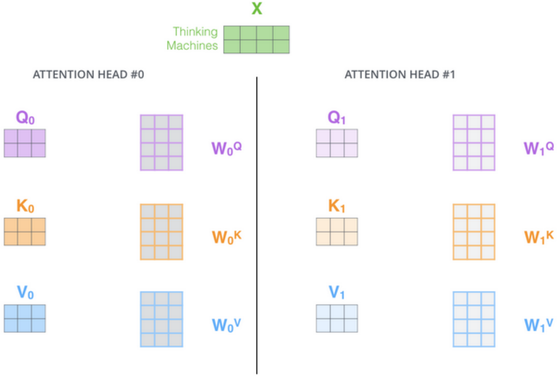
Tensorflow implementation

use tf.linalg.matmul for tensorflow 2

```
def scaled_dot_product_attention(q, k, v, mask):
  """Calculate the attention weights.
  q, k, v must have matching leading dimensions.
  k, v must have matching penultimate dimension, i.e.: seq_len_k = seq_len_v.
  The mask has different shapes depending on its type(padding or look ahead)
  but it must be broadcastable for addition.

  Args:
    q: query shape == (..., seq_len_q, depth)
    k: key shape == (..., seq_len_k, depth)
    v: value shape == (..., seq_len_v, depth_v)
    mask: Float tensor with shape broadcastable
          to (..., seq_len_q, seq_len_k). Defaults to None.

  Returns:
    output, attention_weights
  """

  matmul_qk = tf.matmul(q, k, transpose_b=True)  # (..., seq_len_q, seq_len_k)

  # scale matmul_qk
  dk = tf.cast(tf.shape(k)[-1], tf.float32)
  scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

  # add the mask to the scaled tensor.
  if mask is not None:
    scaled_attention_logits += (mask * -1e9)

  # softmax is normalized on the last axis (seq_len_k) so that the scores
  # add up to 1.
  attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)  # (..., seq_len_q, seq_len_k)

  output = tf.matmul(attention_weights, v)  # (..., seq_len_q, depth_v)

  return output, attention_weights
```

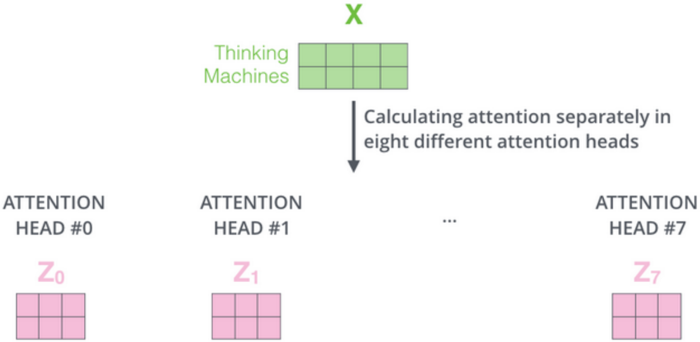
Multi-Head attention mechanism

we have 512 input dimension and we have 64 as W dimensions. so we need 512/64 =8 heads.
Note: 1. we need multi head to focus on different positions. z1 contains a little bit of every other encoding, but it could be dominated by the the actual word itself.
2. expand attention layer multiple "representation subspaces"

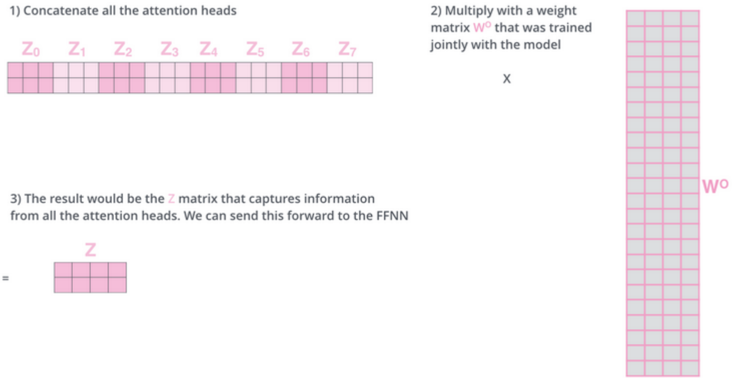


Multi-Head attention mechanism

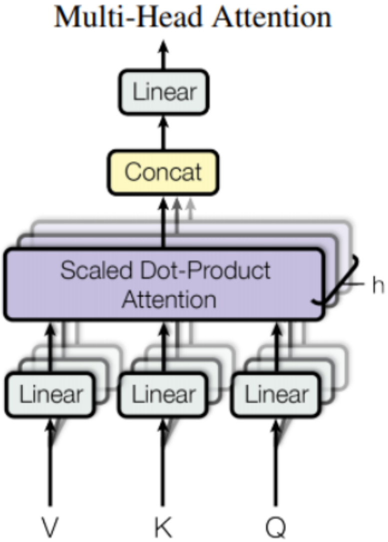
model dimension =512, number of heads=8



Multi-Head attention mechanism



Multi-Head Attention

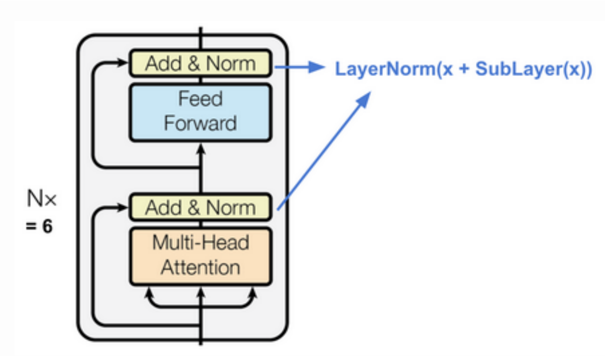


Multi-Head-Attention

```python
class MultiHeadAttention(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads):
    super(MultiHeadAttention, self).__init__()
    self.num_heads = num_heads
    self.d_model = d_model

    assert d_model % self.num_heads == 0

    self.depth = d_model // self.num_heads

    self.wq = tf.keras.layers.Dense(d_model)
    self.wk = tf.keras.layers.Dense(d_model)
    self.wv = tf.keras.layers.Dense(d_model)

    self.dense = tf.keras.layers.Dense(d_model)

  def split_heads(self, x, batch_size):
    """Split the last dimension into (num_heads, depth).
    Transpose the result such that the shape is (batch_size, nu
    """
    x = tf.reshape(x, (batch_size, -1, self.num_heads, self.de
    return tf.transpose(x, perm=[0, 2, 1, 3])

  def call(self, v, k, q, mask):
    batch_size = tf.shape(q)[0]

    q = self.wq(q)  # (batch_size, seq_len, d_model)
    k = self.wk(k)  # (batch_size, seq_len, d_model)
    v = self.wv(v)  # (batch_size, seq_len, d_model)

    q = self.split_heads(q, batch_size)  # (batch_size, num_heads, seq_len_q, depth)
    k = self.split_heads(k, batch_size)  # (batch_size, num_heads, seq_len_k, depth)
    v = self.split_heads(v, batch_size)  # (batch_size, num_heads, seq_len_v, depth)
```

```python
# scaled_attention.shape == (batch_size, num_heads, seq_len_q, depth)
# attention_weights.shape == (batch_size, num_heads, seq_len_q, seq_len_k)
scaled_attention, attention_weights = scaled_dot_product_attention(
    q, k, v, mask)

scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])  # (batch_siz

concat_attention = tf.reshape(scaled_attention,
                              (batch_size, -1, self.d_model))  # (batch_size, seq_

output = self.dense(concat_attention)  # (batch_size, seq_len_q, d_model)

return output, attention_weights
```

Encoder Layer

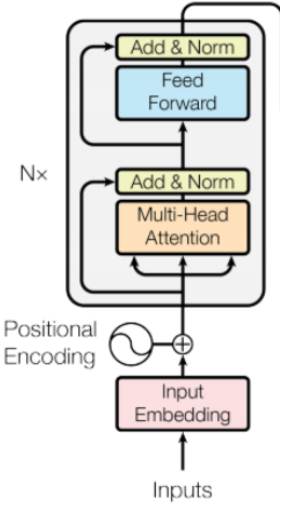


Encoder Layer Tensorflow Code

```
class EncoderLayer(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads, dff, rate=0.1):
    super(EncoderLayer, self).__init__()

    self.mha = MultiHeadAttention(d_model, num_heads)
    self.ffn = point_wise_feed_forward_network(d_model, dff)

    self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
    self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

    self.dropout1 = tf.keras.layers.Dropout(rate)
    self.dropout2 = tf.keras.layers.Dropout(rate)

  def call(self, x, training, mask):

    attn_output, _ = self.mha(x, x, x, mask)  # (batch_size, input_seq_len, d_model)
    attn_output = self.dropout1(attn_output, training=training)
    out1 = self.layernorm1(x + attn_output)  # (batch_size, input_seq_len, d_model)

    ffn_output = self.ffn(out1)  # (batch_size, input_seq_len, d_model)
    ffn_output = self.dropout2(ffn_output, training=training)
    out2 = self.layernorm2(out1 + ffn_output)  # (batch_size, input_seq_len, d_model)

    return out2
```

```
def point_wise_feed_forward_network(d_model, dff):
  return tf.keras.Sequential([
      tf.keras.layers.Dense(dff, activation='relu'),  # (batch_size, seq_len, dff)
      tf.keras.layers.Dense(d_model)  # (batch_size, seq_len, d_model)
  ])
```

Encoder Architecture

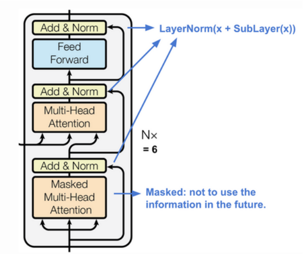


Encoder Tensorflow implementation

```
class Encoder(tf.keras.layers.Layer):
  def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
               maximum_position_encoding, rate=0.1):
    super(Encoder, self).__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.embedding = tf.keras.layers.Embedding(input_vocab_size, d_model)
    self.pos_encoding = positional_encoding(maximum_position_encoding,
                                            self.d_model)


    self.enc_layers = [EncoderLayer(d_model, num_heads, dff, rate)
                       for _ in range(num_layers)]

    self.dropout = tf.keras.layers.Dropout(rate)

  def call(self, x, training, mask):

    seq_len = tf.shape(x)[1]

    # adding embedding and position encoding.
    x = self.embedding(x)  # (batch_size, input_seq_len, d_model)
    x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
    x += self.pos_encoding[:, :seq_len, :]

    x = self.dropout(x, training=training)

    for i in range(self.num_layers):
      x = self.enc_layers[i](x, training, mask)

    return x  # (batch_size, input_seq_len, d_model)
```

Decoder Layer

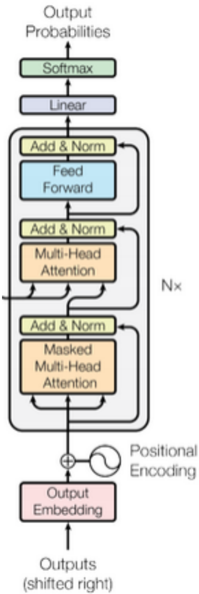
In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to -inf) before the softmax step in the self-attention calculation.
The "Encoder-Decoder Attention" layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.



Decoder Layer TF code

```python
class DecoderLayer(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads, dff, rate=0.1):
    super(DecoderLayer, self).__init__()

    self.mha1 = MultiHeadAttention(d_model, num_heads)
    self.mha2 = MultiHeadAttention(d_model, num_heads)

    self.ffn = point_wise_feed_forward_network(d_model, dff)

    self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
    self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
    self.layernorm3 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

    self.dropout1 = tf.keras.layers.Dropout(rate)
    self.dropout2 = tf.keras.layers.Dropout(rate)
    self.dropout3 = tf.keras.layers.Dropout(rate)


  def call(self, x, enc_output, training,
           look_ahead_mask, padding_mask):
    # enc_output.shape == (batch_size, input_seq_len, d_model)

    attn1, attn_weights_block1 = self.mha1(x, x, x, look_ahead_mask)  # (batch_size, target_seq_len, d_mode
    attn1 = self.dropout1(attn1, training=training)
    out1 = self.layernorm1(attn1 + x)

    attn2, attn_weights_block2 = self.mha2(
        enc_output, enc_output, out1, padding_mask)  # (batch_size, target_seq_len, d_model)
    attn2 = self.dropout2(attn2, training=training)
    out2 = self.layernorm2(attn2 + out1)  # (batch_size, target_seq_len, d_model)

    ffn_output = self.ffn(out2)  # (batch_size, target_seq_len, d_model)
    ffn_output = self.dropout3(ffn_output, training=training)
    out3 = self.layernorm3(ffn_output + out2)  # (batch_size, target_seq_len, d_model)

    return out3, attn_weights_block1, attn_weights_block2
```

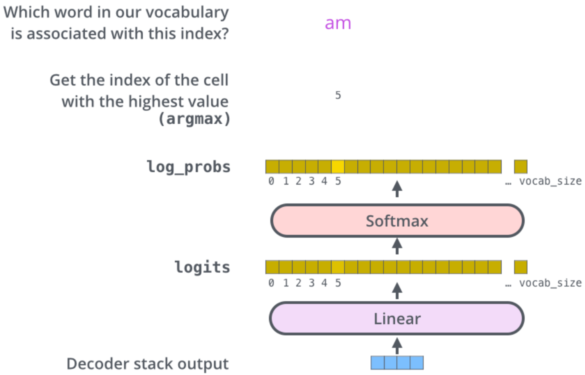
Decoder Architecture



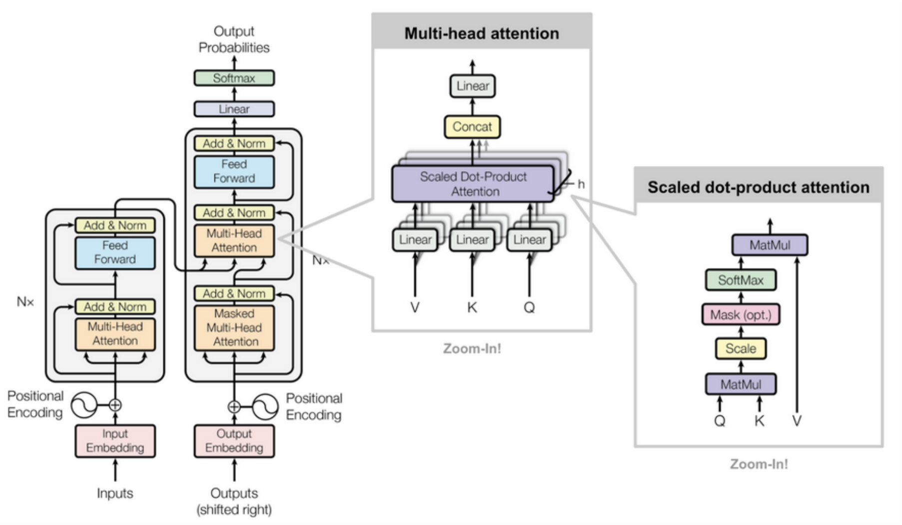
Decoder Architecture Tensorflow implementation

```
class Decoder(tf.keras.layers.Layer):
  def __init__(self, num_layers, d_model, num_heads, dff, target_vocab_size,
               maximum_position_encoding, rate=0.1):
    super(Decoder, self).__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.embedding = tf.keras.layers.Embedding(target_vocab_size, d_model)
    self.pos_encoding = positional_encoding(maximum_position_encoding, d_model)

    self.dec_layers = [DecoderLayer(d_model, num_heads, dff, rate)
                       for _ in range(num_layers)]
    self.dropout = tf.keras.layers.Dropout(rate)

  def call(self, x, enc_output, training,
           look_ahead_mask, padding_mask):

    seq_len = tf.shape(x)[1]
    attention_weights = {}

    x = self.embedding(x)  # (batch_size, target_seq_len, d_model)
    x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
    x += self.pos_encoding[:, :seq_len, :]

    x = self.dropout(x, training=training)

    for i in range(self.num_layers):
      x, block1, block2 = self.dec_layers[i](x, enc_output, training,
                                             look_ahead_mask, padding_mask)

      attention_weights['decoder_layer{}_block1'.format(i+1)] = block1
      attention_weights['decoder_layer{}_block2'.format(i+1)] = block2

    # x.shape == (batch_size, target_seq_len, d_model)
    return x, attention_weights
```

Transformer Decoder Output Softmax



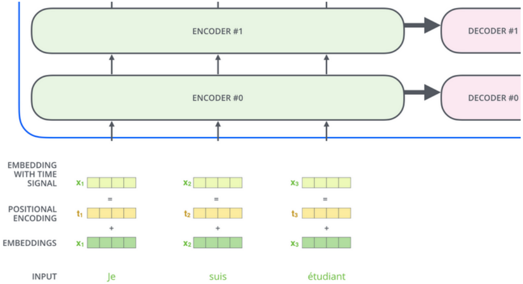
Full Architecture



Example

Positional Encoding

The intuition here is that adding these values to the embedding provides meaningful distances between the embedding vectors once they're projected into Q/K/V vectors and during dot-product attention.



Positional Encoding

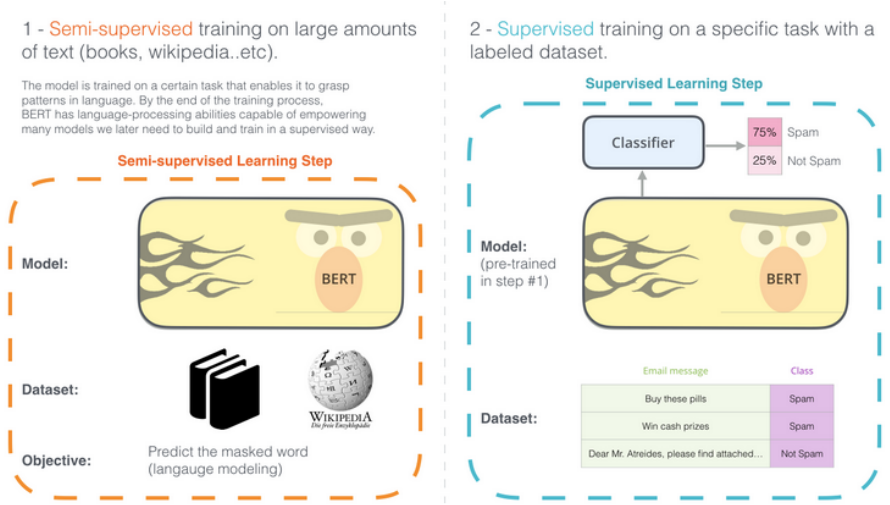
Positional Encoding

```
def get_angles(pos, i, d_model):
  angle_rates = 1 / np.power(10000, (2 * (i//2)) / np.float32(d_model))
  return pos * angle_rates
```

```
def positional_encoding(position, d_model):
  angle_rads = get_angles(np.arange(position)[:, np.newaxis],
                          np.arange(d_model)[np.newaxis, :],
                          d_model)

  # apply sin to even indices in the array; 2i
  angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])

  # apply cos to odd indices in the array; 2i+1
  angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])

  pos_encoding = angle_rads[np.newaxis, ...]

  return tf.cast(pos_encoding, dtype=tf.float32)
```

BERT

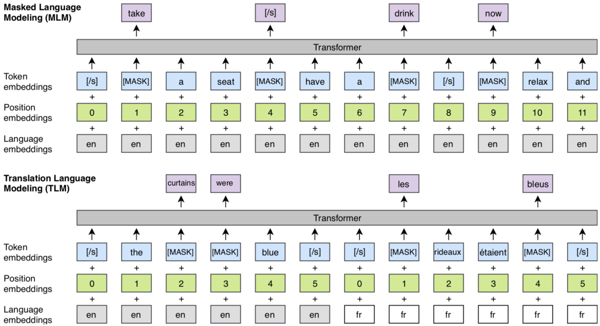
BERT



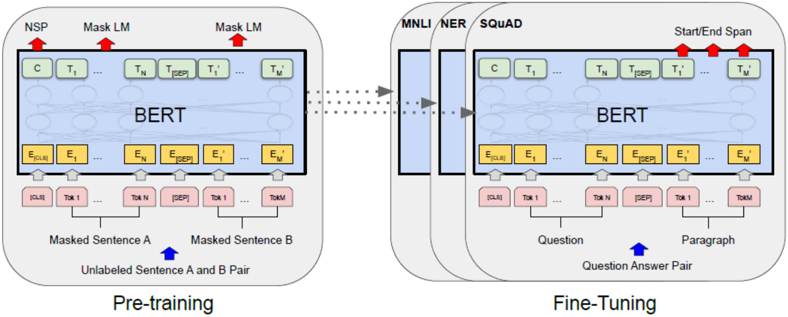
The two steps of how BERT is developed. You can download the model pre-trained in step 1 (trained on un-annotated data), and only worry about fine-tuning it for step 2. [Source for book icon].

Masked Language Modeling

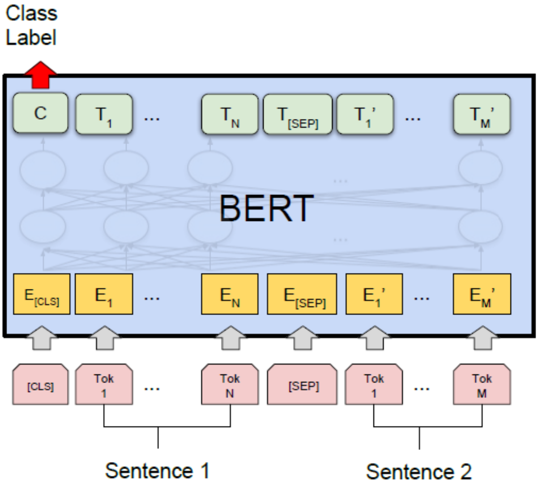
- A [CLS] token is inserted at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.
- A sentence embedding indicating Sentence A or Sentence B is added to each token.
- A positional embedding is added to each token to indicate its position in the sequence. The concept and implementation of positional embedding are presented in the Transformer paper.



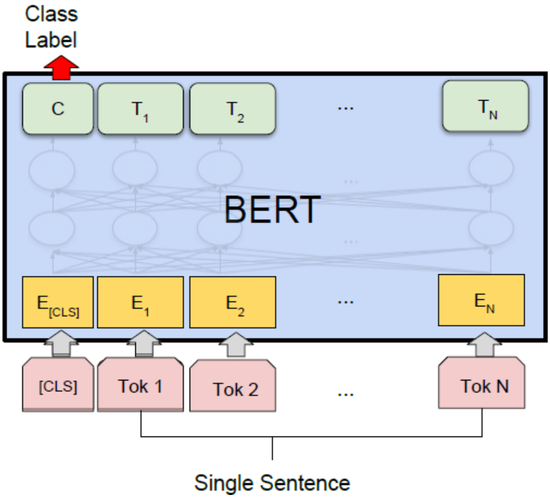
Sentence Classification with BERT



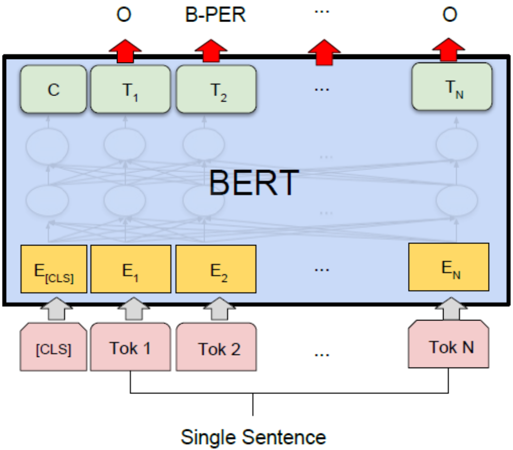
BERT Task: Sentence pair Classification



BERT Task: Single Sentence Classification

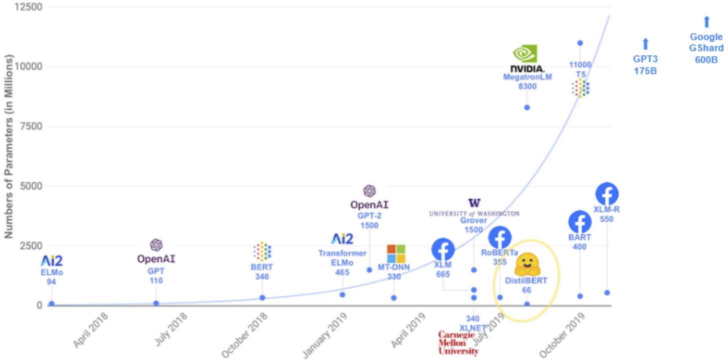


BERT Task: Name Entity Recognition



Model size and computational efficiency

- Recent trends
  - Going big on model sizes - over 1 billion parameters as become the norm for SOTA



References

1. http://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/
2. https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html
3. K. Cho, B. Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In Arxiv preprint arXiv:1406.1078, 2014
4. Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.
5. https://towardsdatascience.com/attention-and-its-different-forms-7fc3674d14dc
6. Luong, M.T., Pham, H. and Manning, C.D., 2015. Effective approaches to attention-based neural machine translation. arXiv preprint arXiv:1508.04025
7. Devlin, J., Chang, M.W., Lee, K. and Toutanova, K., 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.