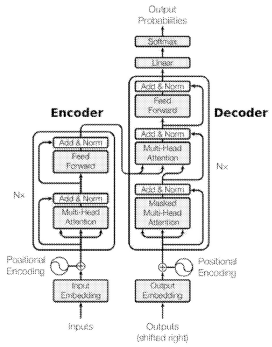
Introduction

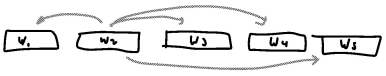
- Attention is all you need 2017
- Encoder-decoder architecture
- Auto-generative sequence to sequence mapping
- Supports attention
- Process entire sequence at once



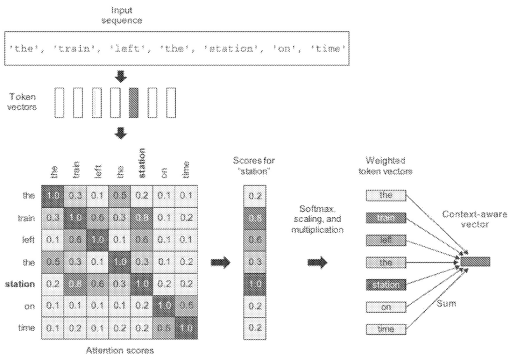
Encoder & Decoder - used to convert of a sequence of tokens to another seq of transformed tokens
Encoder alone - get i/p as a sq and then analyze it
Decoder alone - generative tasks to generate tokens

Self-attention

- Context-aware representation:
  - Attention can be used for more than just highlighting or erasing certain features.
  - Make features context-aware. Instead of using word embedding to assign a word semantic meaning, assign a meaning that is context-specific:
    - For example, "date" in "mark the date" does not have the same semantic meaning as the "date" fruit.
    - Pronouns like "he," "it," "in," are entirely sentence-specific and can change multiple times within a single sentence.
- Self attention:
  - Modulate the representation of a token by using the representations of related tokens in the sequence.
  - Produce context-aware token representations.



Self-attention



Self-attention

- Steps:
  - Compute relevance score between the "station" vector and every other word in the sentence (e.g. by dot product) -> attention scores.
  - Apply softmax to normalize to [0,1] and then scale.
  - Compute the sum of all word vectors in the sentence, weighted by the relevancy scores.
  - Words closely related to "station" will contribute more to the sum (including the word "station" itself).
  - The resulting vector is a new representation for "station" that incorporates surrounding context.
  - Repeat this process for every word in the sentence, producing a new sequence of vectors encoding the sentence.

$$G = AA^T = \begin{bmatrix} -a_1- \\ \vdots \\ -a_m- \end{bmatrix} \begin{bmatrix} | & & | \\ a_1 & \cdots & a_m \\ | & & | \end{bmatrix} = \begin{bmatrix} a_1 \cdot a_1 & \cdots & a_1 \cdot a_m \\ \vdots & & \\ a_m \cdot a_1 & \cdots & a_m \cdot a_m \end{bmatrix}$$

$$a_i^* = \sum_{j=1}^{m} \text{softmax}(a_i \cdot a_j)\, a_j$$

Dividing by sqrt() normalizes the scale of the dot products, keeping the values in a range where the softmax function operates more effectively, avoiding extreme probabilities and ensuring better gradient flow during training

Self-attention

```
def self_attention(input_sequence):
  output = np.zeros(shape=input_sequence.shape)
  # Iterate over each token in the input sequence
  for i, pivot_vector in enumerate(input_sequence):
    scores = np.zeros(shape=(len(input_sequence),))
    # Compute the dot product (attention) with other tokens
    for j, vector in enumerate(input_sequence):
        scores[j] = np.dot(pivot_vector, vector.T)
    # Scale the scores (average) and apply softmax
    scores /= np.sqrt(input_sequence.shape[1])
    scores = softmax(scores)
    new_pivot_representation = np.zeros(shape=pivot_vector.shape)
    # Take the weighted sum of all tokens (weighted by attention scores)
    for j, vector in enumerate(input_sequence):
        new_pivot_representation += vector * scores[j]
        # Assign the output
        output[i] = new_pivot_representation
  return output
```

Self-attention

- Keras code:
  - Keras layer Multi-headed attention: `MultiHeadAttention`:

```
num_heads = 4
embed_dim = 256
mha_layer = MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
outputs = mha_layer(inputs, inputs, inputs)
```

  - The input is passed three times to achieve self-attention as:

```
outputs = sum(inputs * pairwise_scores(inputs, inputs))
```

Embed dimen - dimension of each token
num heads - like paraellel processing in inception...trying to figure out different thing with diff path

Generalized cross-attention

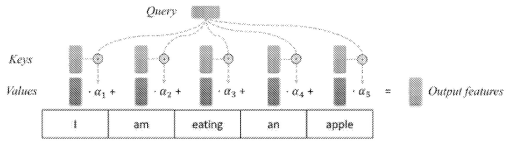
- Query-Key-Value model:
  - Consider the self attention

    `outputs = sum(inputs * pairwise_scores(inputs, inputs))`
  - The 3 inputs can be more generally different: query, keys, values

    `outputs = sum(values * pairwise_scores(query, keys))`
  - For each element in the `query` compute similarity to every `key` and use the similarity scores to weight the sum of `values`.
- Search engine/recommender system terminology: Use a `query` against keywords (`keys`) to compute relevance. Return top `N` matches.

Generalized cross-attention

- **Query** - A reference sequence. The query is a fixed length L_q sequence of d_q dimensional word embeddings. It can be represented by a 2D matrix with dimensions $L_q \times d_q$
- **Keys** - Summary of content of values (one key per value) . The key is a fixed length L_k sequence of d_q dimensional word embeddings. It can be represented by a 2D matrix with dimensions $L_k \times d_q$
- **Values** - Knowledge you are trying to relate to the query. The value is a fixed length L_k (same as key) sequence of d_v dimensional word embeddings. It can be represented by a 2D matrix with dimensions $L_k \times d_v$. In practice keys and values are often the same sequence.
- **Computation** - apply the query to the keys and return a weighted sum of values



Generalized cross-attention

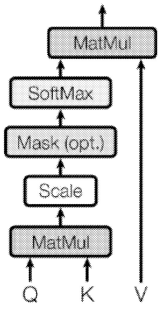
- Scaled dot-product attention:
  - The input consists of queries and keys of dimension d_k, and values of dimension d_v .
  - Compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.
  - In practice, the attention function is computed on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V.
  - The matrix of outputs is computed as: $\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^t}{\sqrt{d_k}})V$
  - The scaling (division by $\sqrt{d_k}$) is necessary to prevent saturation.
  - The optional mask block can be used to mask out specific entries in the attention matrix (e.g. when stacking multiple sequences of different length into a batch)
  - Illustration:

If the input values xi to softmax are very large or small, the exponential function can lead to
Extremely large values: e ^x i for large positive .
Extremely small values: ^ for large negative

This creates a situation where the softmax output is dominated by one value (close to 1 for the largest input, and close to 0 for the others), which leads to saturation. In this state, gradients become very small during backpropagation, making training inefficient or even ineffective (gradient vanishing problem).

Generalized cross-attention

- Scaled dot-product attention:



The optional mask in generalized cross-attention is used to control or restrict the interactions between elements in the input sequences when computing attention scores. Masks are a crucial part of attention mechanisms because they allow fine-grained control over which parts of the input can "attend" to which other parts.

Sequences in a batch often have different lengths. Padding tokens are added to shorter sequences for uniformity, but these padding tokens should not influence attention.

Masks can block attention to irrelevant or disallowed parts of the input.

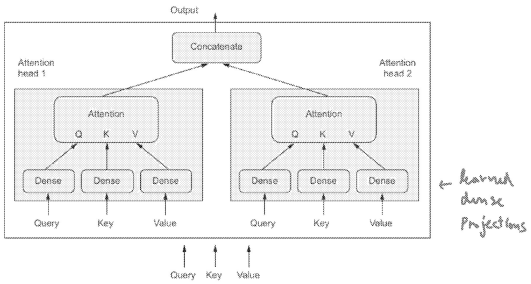
Generalized cross-attention

- Machine translation example:
  - To encode the source sequence, compute cross-similarity of each target token to source tokens and use the similarity measures to compute a weighted sum of the source tokens
  - **Query** - target sequence
  - **Keys** - source sequence
  - **Values** - source sequence

- Sequence classification example:
  - To encode the input sequence, compute self-similarity of each input token to input tokens and use the similarity measures to compute a weighted sum of the input tokens
  - **Query** - input sequence
  - **Keys** - input sequence
  - **Values** - input sequence

Multi-head attention

- Instead of a single attention encoding of inputs, produce multiple encodings that are learned separately -> "Multi-head attention"
- The query, key, and value are sent through k independent sets of **dense projections**, resulting in k separate vectors. Each vector is processed via neural attention, and the k outputs are concatenated back together into a single output sequence. Each such subspace is called a "head."



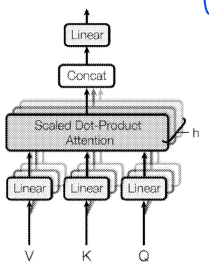
Multi-head attention

- Equations:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

final linear layer weight

initial linear layer weight



Multi-head attention

- Factoring feature spaces into independent subspaces:
  - Independent heads help the layer learn different groups of features (embeddings) for each token. While features within one group are correlated with each other, features between groups are mostly independent.
  - This is similar to inception blocks or depth-wise separable convolutions where the output space is factored into multiple subspaces that are learned independently.
- The Keras `MultiHeadAttention` layer implements multi-headed attention from "Attention is all you Need" (Vaswani et al., 2017).
- Multiple heads is similar to multiple convolution filters in convnets

Multi-head attention

```
tf.keras.layers.MultiHeadAttention(
    num_heads,
    key_dim,
    value_dim=None,
    dropout=0.0,
    use_bias=True,
    output_shape=None,
    attention_axes=None,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

Multi-head attention

- Arguments:
  - `num_heads`: Number of attention heads.
  - `key_dim`: Dimension of query and key tokens. The query and key are multiplied and so their dimensions must match.
  - `value_dim`: Dimension of value tokens.
  - `attention_axes`: Axes over which the attention is applied. `None` means attention over all axes, except for the following: batch, heads, and features.

- Tensor dimensions:
  - B = batch_size
  - S = source_sequence_length (key/value)
- T = target_sequence_length (query)

Multi-head attention

- Call arguments:
  - `query`: `(B, T, key_dim)`
  - `value`: `(B, S, value_dim)`
  - `key`: `(B, S, key_dim)`
  - `attention_mask`: `(B, T, S)`

t-target, s-source

- `return_attention_scores`: A boolean to indicate whether the output should be `attention_output` or `(attention_output, attention_scores)`
  - `training`: A boolean to indicate training mode (with dropout) or inference mode (no dropout).
  - `use_causal_mask`: A boolean to indicate whether to apply a causal mask to prevent tokens from attending to future tokens (e.g., used in a decoder Transformer).
  - Notes:
    - If `query, key, value` are the same, then this is self-attention.
    - `key is optional. Default:` key = value `
    - `attention_mask` is a boolean mask to mask out `query-key` pairs.

Multi-head attention

- Returned values:
  - `attention_output`: `(B, T, value_dim)`
  - `attention_scores`: [Optional] multi-head attention coefficients over attention axes.
  - Note: the `attention_output` dimension may be specified by `output_shape`

- Processing:
  - The query `(B, T, key_dim)` and key `(B, S, key_dim)` tensors are dot-producted and scaled, and then softmaxed to obtain attention probabilities. This produces a `(B, T, S)` attention probabilities tensor where each cell measures similarity between a query and a key.
  - The value tensors `(B, S, value_dim)` are then dot-producted by the attention probabilities tensor `(B, T, S)` to produce a `(B, T, value_dim)` tensor.
  - Note that `key` and `value` dimensions must be the same to allow producing an attention score for each `value`.
  - With multiple heads there is an additional `num_attention_heads` dimension.

Multi-head attention

- Examples:
  - Perform 1D cross-attention over two sequence inputs with an attention mask. Return attention score for each head:

```
layer = MultiHeadAttention(num_heads=2, key_dim=2) # 2D key vectors
target = tf.keras.Input(shape=[8, 16]) # 8 tokens of 16 bit
source = tf.keras.Input(shape=[4, 16]) # 4 tokens of 16 bit
output_tensor, weights = layer(target, source,
...      return_attention_scores=True) # query, value, [key=value]
print(output_tensor.shape)
  # (None, 8, 16)          >> (B, T, value_dim)
print(weights.shape)
  # (None, 2, 8, 4)        >> (B, num_heads, T, S)
```

  - Performs 2D self-attention over a 5D input tensor on axes 2 and 3:

```
layer = MultiHeadAttention(
...      num_heads=2, key_dim=2, attention_axes=(2, 3))
input_tensor = tf.keras.Input(shape=[5, 3, 4, 16])
output_tensor = layer(input_tensor, input_tensor)
print(output_tensor.shape)
  # (None, 5, 3, 4, 16)
```

attention axes - helps to say which elements of the input data are compared and attended to during the attention computation.

eg - we may exclude batch

Input embedding

- Sequence tokenization:
  - The input sentence is tokenized into distinct elements (tokens).
  - A tokenized sentence is a fixed-length sequence (truncated or padded as needed).
  - Example, with maximal length 200, every sentence will have 200 tokens with trailing paddings:

```
('Hello', 'there', <pad>, <pad>, ..., <pad>)
```

  - The tokens are integer indices of entries in a dictionary for the dataset

```
(8667, 1362, 0, 0, ..., 0)
```

  - The sequence normally terminates with a special token `<EOS>`

Input embedding

- Embedding:
  - To feed the tokens into the neural network, each token is converted into an embedding vector.
  - For example, use a 512-dimensional vector for the embedding of each token.
  - Given a sentence with a maximum length of 200, each sentence if represented by a 200×512 matrix.
  - The embedding could be learned ahead of time or learned during training.

Positional encoding

- Self-attention is a set-processing mechanism, focused on the relationships between pairs of sequence elements and is blind to the location of the elements.
- Transformers are a hybrid approach that is technically order-agnostic, but that manually injects order information in the representations it processes through positional encoding.

	Word order awareness	Context awareness (cross-words interactions)
Bag-of-unigrams	No	No
Bag-of-bigrams	Very limited	No
RNN	Yes	No
Self-attention	No	Yes
Transformer	Yes	Yes

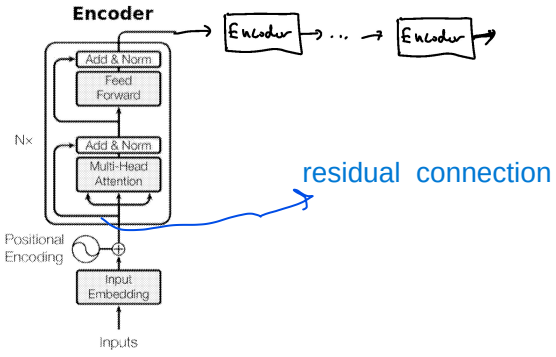
Positional encoding

- To give the model access to word- order information, add the word's position in the sentence to each word embedding.
- The input word embedding has two components: the usual word vector representing the word independently of any specific context, and a position vector representing the position of the word in the current sentence.
- Simple approach: concatenate the word's position to its embedding vector. Large positions will disrupt the range of values in the embedding vector.
- Cyclical position: add to word embeddings a vector containing values in the range [-1, 1] that vary cyclically depending on the position (using a sine/cosine function).

$$PE_{(pos,i)} = \begin{cases} \sin\left(\frac{pos}{10000^{i/d_{\text{model}}}}\right) & \text{if } i \bmod 2 = 0 \\ \cos\left(\frac{pos}{10000^{(i-1)/d_{\text{model}}}}\right) & \text{otherwise} \end{cases}$$

Refer image

Positional encoding



Positional embedding

- Positional embedding: learn position-embedding vectors in the same way that word embeddings are learned. Then add the position embedding to the word embedding to obtain a position-aware word embedding.
- A downside of position embeddings is that the sequence length needs to be known in advance.

Positional embedding

```
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, input_dim, output_dim, **kwargs):
        super().__init__(**kwargs)
        # Prepare an Embedding layer for token indices
        self.token_embeddings = layers.Embedding( input_dim=input_dim,
                                                  output_dim=output_dim)
        # And another one for the token positions
        self.position_embeddings = layers.Embedding(
            input_dim=sequence_length, output_dim=output_dim)
        self.sequence_length = sequence_length
        self.input_dim = input_dim self.output_dim = output_dim

    def call(self, inputs):
        length = tf.shape(inputs)[-1]
        positions = tf.range(start=0, limit=length, delta=1)
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        return embedded_tokens + embedded_positions

    def compute_mask(self, inputs, mask=None):
        return tf.math.not_equal(inputs, 0)
```

10000

256

256

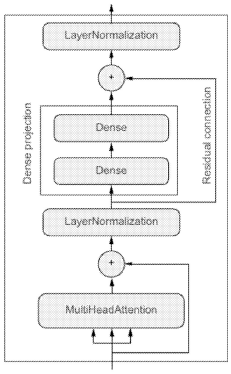
Positional embedding

```
    def get_config(self):
        config = super().get_config()
        config.update({ "output_dim": self.output_dim,
                        "sequence_length": self.sequence_length,
                        "input_dim": self.input_dim, })
        return config
```

Transformer encoder

- Transformer Encoder block:
  - The `TransformerEncoder` block Chains a `MultiHeadAttention` layer with a dense projection and adds normalization as well as residual connections.
  - The original Transformer architecture for sequence-to-sequence mapping (for machine translation) consists of: a Transformer encoder that processes the source sequence, and a Transformer decoder that uses the source sequence to generate a translated version.
  - The encoder part can be used for text classification by learning to turn a sequence into a more useful representation.
  - The skip connections are needed to prevent vanishing gradients in a deep network and maintaining information about the original sequence

Transformer encoder



Transformer encoder

- The `LayerNormalization` layer:
  - The Transformer block uses a `LayerNormalization` layer instead of a `BatchNormalization` layer because `BatchNormalization` doesn't work well for sequence data.
  - The `LayerNormalization` layer normalizes each sequence independently from other sequences in the batch, whereas the `BatchNormalization` layer collects information from many samples.
  - Pseudo code (during training):

```
# Layer normalization
# Input shape: (batch_size, sequence_length, embedding_dim)
def layer_normalization(batch_of_sequences):
    # To compute mean and variance, pool data over the last axis
    mean = np.mean(batch_of_sequences, keepdims=True, axis=-1)
    variance = np.var(batch_of_sequences, keepdims=True, axis=-1)
    return (batch_of_sequences - mean) / variance
# Batch normalization
# Input shape: (batch_size, height, width, channels)
def batch_normalization(batch_of_images):
    # Pool data over the batch axis (axis 0)
    # create interactions between samples in a batch
    mean = np.mean(batch_of_images, keepdims=True, axis=(0, 1, 2))
    variance = np.var(batch_of_images, keepdims=True, axis=(0, 1, 2))
    return (batch_of_images - mean) / variance
```

Transformer encoder

- Transformer encoder implementation:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

class TransformerEncoder(layers.Layer):

    # Define layers
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim # dimension of the input token vectors
        self.dense_dim = dense_dim # dimension of inner dense layer
        self.num_heads = num_heads # number of attention heads
        self.attention = layers.MultiHeadAttention( num_heads=num_heads,
            key_dim=embed_dim)     # create multi-head attention layer
        self.dense_proj = keras.Sequential( [layers.Dense(dense_dim,
            activation="relu"), layers.Dense(embed_dim),] )
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
```

Transformer encoder

```
    # Combine layers
    def call(self, inputs, mask=None): #
        if mask is not None:
            mask = mask[:, tf.newaxis, :]
        attention_output = self.attention( inputs, inputs,
            attention_mask=mask)
        proj_input = self.layernorm_1(inputs + attention_output)
        proj_output = self.dense_proj(proj_input)
        return self.layernorm_2(proj_input + proj_output)

    def get_config(self):
        config = super().get_config()
        config.update({
            "embed_dim": self.embed_dim,
            "num_heads": self.num_heads,
          "dense_dim": self.dense_dim, })
        return config
```

Transformer encoder

- Loading/saving custom layers:
  - The `get_config` method is necessary in custom layers to allow model saving and loading. The method should return a Python dictionary that contains the values of the constructor arguments used to create the layer. The config does not contain weight values.
  - Keras layers can be serialized and deserialized. Example:

```
layer = PositionalEmbedding(sequence_length, input_dim, output_dim)
config = layer.get_config()
new_layer = PositionalEmbedding.from_config(config)
model = keras.models.load_model( filename, custom_objects=
  {"PositionalEmbedding": PositionalEmbedding})
```

Transformer encoder

- Text classification example:
  - Create the model

```
vocab_size = 20000
embed_dim = 256
num_heads = 2
dense_dim = 32
inputs = keras.Input(shape=(None,), dtype="int64")
x = layers.Embedding(vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```

  - The `TransformerEncoder` returns full sequences, and so we need to reduce each sequence to a single vector for classification via a global pooling layer.
  - The `GlobalMaxPooling1D` layer is applied to 1D temporal data. It downsamples the input representation by taking the maximum value over the time dimension.

Transformer encoder

- Train and evaluate the model

```
callbacks = [
keras.callbacks.ModelCheckpoint("transformer_encoder.keras",
save_best_only=True) ]

model.fit(int_train_ds, validation_data=int_val_ds, epochs=20,
callbacks=callbacks)

model = keras.models.load_model( "transformer_encoder.keras",
custom_objects={"TransformerEncoder": TransformerEncoder})

print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

  - Note: You need to provide the custom TransformerEncoder class to the model loading function
  - The model gets to 87.5% test accuracy - slightly worse than the GRU model.

Transformer encoder

- Learning rate warm-up
  - Gradually increase the learning rate from 0 on to the desired learning rate in the first few iterations. Thus, slowly start learning instead of taking very large steps from the beginning.
  - Training a deep Transformer without learning rate warm-up can make the model diverge and achieve a much worse performance on training and testing.