Transformer Implementation with the High-Level Keras API

James Hirschorn

June 22, 2021

Contents

1 Transformer Implementation

There are a numerous blogs/tutorials demonstrating transformer implementations in TensorFlow from scratch, including the official TensorFlow transformer tutorial. However, we could not find a single example using the high-level Keras API for building and training the transformer. For example, the official tutorial does not use Keras' built-in APIs for training and validation. This created difficulties for us when we attempted to build a customized transformer based on existing examples.

The purpose of this article is to present a TensorFlow implementation of the transformer sequence-to-sequence architecture Vaswani et al. (2017) in Keras following the high-level API specifications. We use TensorFlow's built in implementation of the Keras API (see e.g. Guidance on High-level APIs in TensorFlow 2.0). Using a high-level API makes the learning process more straightforward and the code much briefer. It also avoids reinventing the wheel which can potentially introduce errors.

While the primary emphasis is on implementation, we also give our own in depth explanation of the transformer model.

1.1 Requirements

This library requires TensorFlow version 2.5.0. It *may* work on newer versions as well, and we have tested it on the version 2.6 development branch. The full requirements are listed in `inst/python/requirements.txt`.

Please note the root directory for the actual `python` code is `inst/python`.

2 Data

All of the data is obtained from the `tensorflow_datasets` library. We begin with the `ted_hrlr_translate` resource and the Portuguese to English language pair.

```
import tensorflow_datasets as tfds

resource = 'ted_hrlr_translate'
pair = 'pt_to_en'
examples, metadata = tfds.load(f'{resource}/{pair}', with_info=True,
                               as_supervised=True)

keys = metadata.supervised_keys
train_examples, eval_examples = examples['train'], examples['validation']

print(f'Keys: {metadata.supervised_keys}')
```

```
## Keys: ('pt', 'en')
```

The individual examples have the following format:

```
example1 = next(iter(train_examples))
print(example1)
```

```
## (<tf.Tensor: shape=(), dtype=string, numpy=b'e quando melhoramos a procura ,
## tiramos a \xc3\xbanica vantagem da impress\xc3\xa3o , que \xc3\xa9 a
## serendipidade .'>, <tf.Tensor: shape=(), dtype=string, numpy=b'and when you
## improve searchability , you actually take away the one advantage of print ,
## which is serendipity .'>)
```

2.1 Tokenizers

As usual for language modeling, sentences in some language must be converted to sequences of integers in order to serve as input for a neural network, in a process called *tokenization*. The input sentences are tokenized using the class `SubwordTokenizer` in the script `tokenizer/subword_tokenizer.py`. It is closely based on the `CustomTokenizer` class from the Subword tokenizer tutorial which is in turn based on the `BertTokenizer` from `tensorflow_text`.

The script `prepare_tokenizers.py` provides the `prepare_tokenizers` function which builds a pair of `SubwordTokenizers` from the input examples and saves them to disk for later reuse, as they take some time to build. The parameters below indicate that all text is converted to lowercase and that the maximum vocabulary size of both the inputs and targets is $2^{13} = 8192$.

```
from prepare_tokenizers import prepare_tokenizers

TRAIN_DIR = 'train'

tokenizers, _ = prepare_tokenizers(train_examples,
                                   lower_case=True,
                                   input_vocab_size=2 ** 13,
                                   target_vocab_size=2 ** 13,
                                   name=metadata.name + '-' + keys[0] + '_to_' + keys[1],
                                   tokenizer_dir=TRAIN_DIR,
                                   reuse=True)

input_vocab_size = tokenizers.inputs.get_vocab_size()
target_vocab_size = tokenizers.targets.get_vocab_size()
print("Number of input tokens: {}".format(input_vocab_size))
```

```
## Number of input tokens: 8318
```

```
print("Number of target tokens: {}".format(target_vocab_size))
```

```
## Number of target tokens: 7010
```

The tokenizer is demonstrated on the the English sentence from example 1 above.

```
example1_en_string = example1[1].numpy().decode('utf-8')
tokenizer = tokenizers.targets
print(f'Sentence: {example1_en_string}')
```

```
## Sentence: and when you improve searchability , you actually take away the one
## advantage of print , which is serendipity .
```

```
tokens = tokenizer.tokenize([example1_en_string])
print(f'Tokenized sentence: {tokens}')
```

```
## Tokenized sentence: <tf.RaggedTensor [[2, 72, 117, 79, 1259, 1491, 2362, 13,
## 79, 150, 184, 311, 71, 103, 2308, 74, 2679, 13, 148, 80, 55, 4840, 1434, 2423,
## 540, 15, 3]]>
```

```
text_tokens = tokenizer.lookup(tokens)
print(f'Text tokens: {text_tokens}')
```

```
## Text tokens: <tf.RaggedTensor [[b'[START]', b'and', b'when', b'you',
## b'improve', b'search', b'##ability', b',', b'you', b'actually', b'take',
## b'away', b'the', b'one', b'advantage', b'of', b'print', b',', b'which', b'is',
## b's', b'##ere', b'##nd', b'##ip', b'##ity', b'.', b'[END]']]>
```

```
round_trip = tokenizer.detokenize(tokens)
print(f"Convert tokens back to original sentence: " \
      f"{round_trip.numpy()[0][0].decode('utf-8')}")
```

```
## Convert tokens back to original sentence: and when you improve searchability ,
## you actually take away the one advantage of print , which is serendipity .
```

The `tokenize` method converts a sentence (or any block of text) into a sequence of tokens (i.e. integers). The `SubwordTokenizer` methods are intended for lists of sentences, corresponding to the batched inputs fed to the neural network, while in this example we use a batch of size one. The `lookup` method shows which subword each input token represents. Note that the tokenizer has added special start and end tokens accordingly to the tokenized sequence, which allows the model to understand about the start and end of each input. `detokenize` maps the tokens back to the original sentence.

2.2 Data Pipeline

The `tf.data.Dataset` API is used for the input pipeline, suitable for consumption by `TensorFlow/Keras` models. Since our data comes from `tensorflow_datasets` it is already a `tf.data` object to which we can apply the necessary transformations and then iterate as batches.

Our input pipeline tokenizes the sentences from both languages into sequences of integers, discards any examples where either the source or target has more than `MAX_LEN` tokens and collects them into batches of size `BATCH_SIZE`. The reason for limiting the length of the input sequences is that both the transformer run time and memory usage are quadratic in the input length, which is evident from the attention mechanism shown in equation (3) below.

The result is a `tf.data` dataset which return a tuple of `(inputs, targets)` for each batch. As is typical for Encoder–Decoder auto-regressive sequence-to-sequence architectures, the input is of the form `(encoder_inpout, decoder_input)` where `encoder_input` is the tokenized source sentence and `decoder_input` is tokenized target sentence with the last token dropped; while `targets` is the tokenized target sentence lagged by one for autoregression.

The input pipeline encapsulated in our `Dataset` class follows the TensorFlow Data Pipeline Performance Guide:

`transformer/dataset.py`

```
import tensorflow as tf

BUFFER_SIZE = 20000


class Dataset:
    """
    Provides a data pipeline suitable for use with transformers
    """
    def __init__(self, tokenizers, batch_size, input_seqlen, target_seqlen):
        self.tokenizers = tokenizers
        self.batch_size = batch_size
        self.input_seqlen = input_seqlen
        self.target_seqlen = target_seqlen

    def data_pipeline(self, examples, num_parallel_calls=None):
        return (
            examples
                .cache()
                .map(tokenize_pairs(self.tokenizers),
                     num_parallel_calls=num_parallel_calls)
                .filter(filter_max_length(max_x_length=self.input_seqlen,
                                          max_y_length=self.target_seqlen))
                .shuffle(BUFFER_SIZE)
                .padded_batch(self.batch_size)
                .prefetch(tf.data.AUTOTUNE)
        )


def filter_max_length(max_x_length, max_y_length):
    def filter(x, y):
        return tf.logical_and(tf.size(x['encoder_input']) <= max_x_length,
                              tf.size(y) < max_y_length)

    return filter


def tokenize_pairs(tokenizers):
    def tokenize(x, y):
        inputs = tokenizers.inputs.tokenize([x])[0]
        targets = tokenizers.targets.tokenize([y])[0]

        decoder_inputs = targets[:-1]
        decoder_targets = targets[1:]
        return dict(encoder_input=inputs, decoder_input=decoder_inputs), decoder_targets

    return tokenize
```

We extract the first batch from the data pipeline:

```
import tensorflow as tf
from transformer.dataset import Dataset

BATCH_SIZE = 64
MAX_LEN = 40

dataset = Dataset(tokenizers, batch_size=BATCH_SIZE,
                  input_seqlen=MAX_LEN, target_seqlen=MAX_LEN)
data_train = dataset.data_pipeline(train_examples,
                                   num_parallel_calls=tf.data.experimental.AUTOTUNE)
data_eval = dataset.data_pipeline(eval_examples,
                                  num_parallel_calls=tf.data.experimental.AUTOTUNE)
batch1 = next(iter(data_train))
print(batch1)
```

```
## ({'encoder_input': <tf.Tensor: shape=(64, 39), dtype=int64, numpy=
## array([[   2,  115,   54, ...,    0,    0,    0],
##        [   2, 1916,  101, ...,    0,    0,    0],
##        [   2,  228,   54, ...,    0,    0,    0],
##        ...,
##        [   2,  108,  457, ...,    0,    0,    0],
##        [   2,  329,   84, ...,    0,    0,    0],
##        [   2,  129, 1221, ...,    0,    0,    0]])>, 'decoder_input': <tf.Tensor: shape=(64, 35), dt
## array([[  2, 100,  76, ...,   0,   0,   0],
##        [  2,  76, 546, ...,   0,   0,   0],
##        [  2, 245, 963, ...,   0,   0,   0],
##        ...,
##        [  2, 110, 117, ...,   0,   0,   0],
##        [  2,  45, 133, ...,   0,   0,   0],
##        [  2, 476, 793, ...,   0,   0,   0]])>}, <tf.Tensor: shape=(64, 35), dtype=int64, numpy=
## array([[ 100,   76, 6072, ...,    0,    0,    0],
##        [  76,  546,  122, ...,    0,    0,    0],
##        [ 245,  963,  635, ...,    0,    0,    0],
##        ...,
##        [ 110,  117,   81, ...,    0,    0,    0],
##        [  45,  133,   78, ...,    0,    0,    0],
##        [ 476,  793,   73, ...,    0,    0,    0]])>)
```

3 Transformer Architecture

The transformer has an encoder-decoder structure as is typical for neural sequence transduction models, e.g. language translation. The encoder encodes a sequence of tokens $(x_1, \ldots, x_n)$ from the first language to a continuous representation $\mathbf{z} = (z_1, \ldots, z_n)$ in d_{model}-dimensional Euclidean space, i.e. each $z_i \in \mathbb{R}^{d_{\text{model}}}$. The decoder the maps this representation to a sequence of tokens $(y_1, \ldots, y_m)$ auto-regressively: each symbol y_j is predicted from $\mathbf{z}$ and the previously predicted symbols $y_1 \ldots, y_{j-1}$.

In all of the code snippets below, the imports are not shown but are included in the following:

```
import tensorflow as tf
from tensorflow.keras import Sequential, Model
from tensorflow.keras.layers import Layer, Embedding, Dropout, LayerNormalization, Input
```

Filenames in this section are relative to the `transformer` subdirectory of the source code.

3.1 Embeddings

The initial layer of the both the encoder and decoder map input and output symbols to d_{model}-dimensional space, respectively. This is the usual learned embedding, where for each symbol d_{model} weights are learned which map this symbol to d_{model}-space. These embeddings tend to have desirable properties which make them interesting in their own right. For example, when the symbols are tokens obtained from some language those with similar meanings have closer embeddings in d_{model}-space.

3.2 Masking

The Keras API has built in support for masking, which is used to ignore specified positions in sequential inputs. It is described for example in the TensorFlow Keras Guide on Masking and Padding. Layers that utilize masking are either mask consumers or producers, where the latter can also modify an existing mask. In the transformer architecture the attention layers are the mask consumers.

There are two uses for masking in the architecture. The first is the usual one for sequence models. The input sequences are padded at the end with zeros so that each batch has members of the same length. These are masked out so that, for example, the position pair (i, j) has zero attention weight whenever either the i^{th} or the j^{th} position of the input is padding. We define the `PaddingMask` layer to produce these masks. In the Embeddings section it is explained why we chose not to use the Keras built-in `Embedding` mask producing layer to produce these masks.

```python
_keras_mask_attr = '_keras_mask'

class PaddingMask(Layer):
    def __init__(self, mask_value=0):
        super().__init__()
        self.mask_value = mask_value

    def call(self, inputs):
        # Needed to ensure that mask gets recomputed
        if hasattr(inputs, _keras_mask_attr):
            delattr(inputs, _keras_mask_attr)
        return inputs

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

To implement a mask producing layer, the class member function `compute_mask` is implemented. It takes the inputs and an optional mask argument and uses them to produce a new mask. Keras stores a tensor's mask in the `_keras_mask` attribute. Notice that we remove this attribute in the `call` method. We found this to be necessary, because as we discovered TensorFlow has an efficiency "hack", where it does not produce a mask when a mask producing layer's output already has a mask. However, the `PaddingMask` simply passes its input forward which might have some preexisting mask which needs to be replaced.

The other use of masking is in auto-regression. During training the decoder layer uses what is called *teacher forcing* where the loss is determined by the model's output at position i, where it receives the decoder input sequence up to but not including position i. This achieved by masking in the decoder self-attention layer (see below), where positions pairs (i, j) with $i < j$ are masked out. This means that the i^{th} query can only do lookups with the first i keys (see Attention and Decoder Layer below). Without this masking there is a mismatch between the training, and what happens during inference which is necessarily auto-regressive. Indeed, we confirmed that the model outputs nonsense when trained without the auto-regressive mask.

```python
def create_look_ahead_mask(size):
    """
    Lower-triangular Boolean matrix
    """

    mask = tf.cast(tf.linalg.band_part(tf.ones((size, size)), -1, 0), tf.bool)
    return mask  # (size, size)


class AutoRegressiveMask(Layer):
    def __init__(self):
        super().__init__()

    def call(self, inputs, mask=None):
        # Needed to ensure that mask gets recomputed
        if hasattr(inputs, _keras_mask_attr):
            delattr(inputs, _keras_mask_attr)
        return inputs

    def compute_mask(self, inputs, mask=None):
        if mask is None:
            return None

        seq_length = tf.shape(inputs)[1]

        look_ahead_mask = create_look_ahead_mask(seq_length)
        mask = tf.logical_and(look_ahead_mask, tf.expand_dims(mask, axis=1))
        return mask
```

3.3 Positional Encodings

An effective sequence-to-sequence models takes into account the ordering of its input sequences. For example, in the case of language translation, it should be more than just a bag of words model. Positional encodings are used to inform the model of the position of each symbol in a given input sequence. Thus the integers, representing 0-based indices, are mapped into $\mathbb{R}^{d_{\text{model}}}$. The particular encoding used here is given by the Fourier basis functions with wavelengths varying with the dimensions: An integer p is mapped to $E(p) = (E(p)_0, \ldots, E(p)_{d_{\text{model}}-1})$ where

$$\begin{aligned} E_{2i}(p) &= \sin(p \cdot b^{-\frac{2i}{d_{\text{model}}}}), \\ E_{2i+1}(p) &= \cos(p \cdot b^{-\frac{2i}{d_{\text{model}}}}), \end{aligned} \tag{1}$$

where b is the base of the encoding. The wavelengths $\theta_i = 2\pi \cdot b^{\frac{2i}{d_{\text{model}}}}$ form a geometric progression from 2π to $2b\pi$ (noninclusive). The $2i^{\text{th}}$ dimension contains the sin Fourier basis functions at wavelength θ_i, evaluated at $x = 1$, and the $2i + 1^{\text{th}}$ dimension contains the cos basis functions at this wavelength. The use of varying wavelengths ensures that distant integers are dissimilar in d_{model}-space even though they might be very close in some dimensions. From basic properties of the Fourier basis, it follows that for each k there exists a vector $\lambda_k \in \mathbb{R}^{d_{\text{model}}}$ such that

$$E(p + k) = \lambda_k \cdot E(p) \quad \text{for all } p. \tag{2}$$

In Vaswani et al. (2017) it is hypothesized that this latter property allow the model to relative position as well as actual position.

The code for the `ScaledEmbedding` layer is displayed here, slightly simplified from the class in `embedding.py`. The `positional_encoding` function is from `positional_encoding.py` and was taken verbatim from the TensorFlow transformer tutorial.

```
class ScaledEmbedding(Layer):
    def __init__(self, input_dim, output_dim, dropout_rate, max_seqlen,
                 positional=True):
        super().__init__()
        self.embedding = Embedding(input_dim, output_dim, mask_zero=True)
        self.positional = positional
        if positional:
            self._positions_enc = positional_encoding(max_seqlen, output_dim)
        self.dropout = Dropout(dropout_rate)
        self._c = tf.math.sqrt(tf.cast(output_dim, dtype=tf.float32))
        self.supports_masking = True

    def call(self, inputs, training=None):
        x_enc = self.embedding(inputs) * self._c
        if self.positional:
            seq_len = tf.shape(inputs)[1]
            x_enc += self._positions_enc[:, :seq_len, :]
        return self.dropout(x_enc, training)
```

Custom layers that are not mask-producing, i.e. which do not modify the current input mask or create a new one, will destroy the current mask by default. Setting the `supports_masking` attribute to `True` allows the current mask to instead propagate through to the next layer unchanged. This is needed since we use `PaddingMask` before `ScaledEmbedding`, and this padding mask needs to be propagated to the encoder/decoder layers. Note that we could have used the built in `Embedding` layer to produce the padding mask, but this seemed less efficient because then we would need to add a `compute_mask` method to `ScaledEmbedding` which would then make a redundant second call to the `Embedding` just to get the mask.

The dropout layer only has an effect during training, and for this reason our `call` method takes the optional Boolean `training` parameter to inform the `Dropout` layer. When using the Keras API for training or inference, it automatically passes the correct `training` value to all of its layers (including `ScaledEmbedding`). In fact, we do not even have to use the `training` parameter because the API will automatically pass the correct value to the dropout layer. However, we include it so that our custom `ScaledEmbededing` can also be used properly independently of the Keras API.

3.4 Transformer Sublayers

The transformer architecture contains only attention sublayers and simple feed-forward sublayers: "Attention is all you need". Every sublayer in the transformer has a residual connection followed by Layer Normalization, which standardizes the output so that each sample has mean 0 and variance 1, as opposed to Batch Normalization which standardizes across the whole batch.

Residual connections avoid the problem of vanishing gradients, but more importantly avoid the *Degradation problem* where adding additional layers degrade accuracy even on the training set (so not due to overtraining). They are introduced in Deep Residual Learning for Image Recognition.

In general, mean/variance normalization speeds up stochastic gradient descent by creating a more symmetric error surface and also alleviates vanishing/exploding gradients in multilayer networks. *Batch Normalization* (BN) is a milestone technique, where the normalization is taken for each neuron over the input batch. BN is problematic for sequential models such as transformers, where the batch samples have differing sequence lengths. In Ba, Kiros, and Hinton (2016), the alternative *Layer Normalization* is introduced where the normalization is over each layer instead of neuron, but is taken one sample at a time avoiding the issue with differing sequence lengths. See also Group Normalization for more on generalizing Batch Normalization.

Given an input x, and possibly additional inputs $\ldots$, the output is then $\text{LayerNorm}(x + \text{SubLayer}(x, \ldots))$ where $\text{SubLayer}(x, \ldots)$ is the output of either an attention or feed-forward sublayer. In order to facilitate the residual connections, the output of each layer, including the embedding layers, must have the same dimension d_{model}. The original paper used $d_{\text{model}} = 512$ dimensions, while we used $d_{\text{model}} = 128$.

The transformer sublayer is implemented in `sublayer.py`.

```
class TransformerSubLayer(Layer):
    def __init__(self, input_sublayer, input_key=None, epsilon=1e-6,
                 dropout_rate=0.1):
        super().__init__()

        self.input_sublayer = input_sublayer
        self.input_key = input_key
        self.epsilon = epsilon
        self.dropout_rate = dropout_rate
        self.dropout = Dropout(dropout_rate)
        self.layernorm = LayerNormalization(epsilon=epsilon)

    def call(self, inputs, training=False, mask=None):
        if self.input_key is not None:
            x = inputs[self.input_key]
        else:
            x = inputs
        sublayer_outputs = self.input_sublayer(inputs=inputs, mask=mask)
        outputs = self.dropout(sublayer_outputs, training)
        outputs += x  # Loses the mask info
        return self.layernorm(outputs)

    def compute_mask(self, inputs, mask=None):
        if mask is None:
            return None

        if self.input_key:
            return mask[self.input_key]
        return mask
```

The `input_layer` parameter is an TensorFlow `Layer` (in its use for the transformer this is either an attention sublayer or a feed-forward sublayer). The `inputs` to the `call` function are passed to this `input_layer`, and can be either a TensorFlow tensor or a `dict` mapping names to tensors in the case of multiple inputs. In the latter case, `input_key` indicates which member of the `dict` is the primary input to be used in the residual connection; and also the input masks are modified so that only the mask relevant to the output is returned.

3.5 Attention

3.5.1 Scaled Dot-Product Attention

Given n_k keys of dimension d_k and $n_v = n_k$ values of dimension d_v, packed into an $n_k \times d_k$ matrix K and an $n_k \times d_v$ matrix V, respectively, along with n_q queries of dimension $d_q = d_k$ packed as Q, the attention function outputs the following $n_q \times d_v$ matrix:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V, \tag{3}$$

where the softmax is rowwise, so that the rows are probability distributions, forming an $n_q \times n_k$ matrix of *attention weights*. Thus the i^{th} row of the attention weight matrix gives a probability distribution (i.e. weights) over the values used to resolve the i^{th} query, where each weight is determined by the compatibility of the query with the corresponding key.

In the case of sequence-to-sequence models, the i^{th} query corresponds to the i^{th} position of the input sequence and the j^{th} key/value pair corresponds to the j^{th} position of the target sequence, so that the attention mechanism determines how much weight is given to this pair in the output.

In `attention.py`, we use nearly same code as in TensorFlow transformer tutorial to implement the scaled dot-product attention in (3), generalized to tensors and including masking, except that the `mask` parameter is a Boolean tensor (one Boolean value per timestep in the input) used to skip certain input timesteps when processing timeseries data.

```python
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.: n_k = n_v.
    q, k must have matching last dimension, i.e.: d_q = d_k.
    The mask has different shapes depending on its type(padding or look ahead)
    but it must be broadcastable for addition.

    Args:
      q: query shape == (..., n_q, d_q)
      k: key shape == (..., n_k, d_k)
      v: value shape == (..., n_v, d_v)
      mask: Boolean tensor with shape broadcastable
            to (..., n_q, n_k). Defaults to None.

    Returns:
      output, attention_weights
    """

    matmul_qk = tf.matmul(q, k, transpose_b=True)  # (..., n_q, n_k)

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

    # add -infinity to the masked out positions in the scaled tensor.
    if mask is not None:
        masked_out = tf.cast(tf.math.logical_not(mask), tf.float32)
        scaled_attention_logits += masked_out * -1e9

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

    output = tf.matmul(attention_weights, v)  # (..., n_q, d_v)

    return output, attention_weights
```

3.5.2 Attention Layer

For simplicity we present a single-head attention layer here. The actual transformer uses a multi-head attention layer in `attention.py`. It is implemented as a custom layer, using the Keras Layer subclassing API.

In the Keras Layer subclassing API, a custom layer is implemented as a `class` that inherits from `Layer`, and implements a `call` method with mandatory argument `inputs`, and "privileged" optional arguments `training` and `mask`. The API also specifies a `build` method for initialization steps which can only be performed after the input shape is known, but this does not apply to any of our layers.

The `SingleHeadAttention` layer is a mask consuming layer. Thus its `call` method exposes the `mask` parameter which gets passed on to `scaled_dot_product_attention`, to indicate which position pairs (i, j) in the inputs and targets to ignore.

single_head_attention_layer.py

```
class SingleHeadAttention(Layer):
    def __init__(self, d_model):
        super().__init__()
        self.d_model = d_model
        self.dense = Dense(d_model)

    def call(self, inputs, mask=None):
        q, k, v = inputs['q'], inputs['k'], inputs['v']

        scaled_attention, attention_weights = scaled_dot_product_attention(
            q, k, v, mask)

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

        return output, attention_weights
```

This layer performs the calculation:

$$\mathrm{SingleHeadAttention}(Q, K, V) = \mathrm{Attention}(Q, K, V)W^O, \tag{4}$$

where $W^O \in \mathbb{R}^{d_v \times d_{\text{model}}}$ is the learned linear projection of the scaled dot-product to d_{model}-space. Note that even though we shall always have $d_v = d_{\text{model}}$ the projection is still important to orient the output, since it is used in calculations such as $x + \mathrm{SingleHeadAttention}(x, x, x)$ where the orientation of the output becomes relevant. Indeed we verified that while the transformer still learns without the final projection, the results are substantially worse. (We did not make actual benchmarks, but simply inspected the model outputs after training for the same number of epochs in each case.)

3.5.2.1 Multi-Head Attention

The multi-head attention splits the attention mechanism into n separate "heads", each with their own representation of the queries, keys and values. They run in parallel, allowing the model to attend simultaneously to different position pairs in different representation spaces:

$$\mathrm{MultiHeadAttention}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_n)W^O \tag{5}$$

where Concat is row-wise concatenation and

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V), \tag{6}$$

and $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$ are projections.

In transformer architecture, the representation spaces are all of dimensions $d_q = d_k = d_v = d_{\text{model}} \,/\, h$. We use $h = 8$ heads as in the original paper, and saw an visible improvement of the single-head architecture. As noted in the paper, a single-head attention cannot simultaneously attend differently in different representation subspaces. If one tries to add projections W^Q, W^K and W^V in equation (4), they will simply be factored out into W^O and have no effect.

The code for the `MultiHeadAttention` class is in `attention.py` and is nearly the same as the corresponding class in the official transformer tutorial, except that the `call` method signature has been modified to conform with the Keras API.

3.5.2.2 Contextual Embeddings

In contrast to the embedding layers, which embed a symbol to the same vector regardless of its position in the sequence (besides the added positional encoding which encodes only its ordinal position), the attention layers are contextual embeddings meaning that the representation of each symbol is based on the entire input sequence—except when an auto-regressive mask is used so that it only depends on the preceding symbols in the sequence, as in the self-attention of the decoder layer described below.

3.6 Feed-Forward Networks

Each transformer sublayer has a fully connected feed-forward network consisting of two layers with a single ReLU activation in between. The first layer has dimension dff which is 2048 in the paper, while we use the smaller value of 512 in our example code. The second output layer has dimension d_{model}, the same as the input. Note that each position of the sequential input gets passed through the identical feed-forward network. This is the only place in the transformer architecture where there is nonlinearity. It is where the queries, keys and values are learned for the self-attention layers of the encoder and decoder, as described next.

These can be easily implemented using the Sequential API. A slightly simplified version of the code in `feed_forward` is as follows.

```python
def pointwise_feed_forward_network(d_model, dff):
    return Sequential([
        Dense(dff, activation='relu'),  # (batch_size, seq_len, dff)
        Dense(d_model)  # (batch_size, seq_len, d_model)
    ])
```

3.7 Encoder

The encoder is a stack of N identical encoder layers, where $N = 6$ in the paper while we use 4 layers in our example. The output of the encoder, which is the output of the last encoder layer, is a sequence of elements in d_{model}-dimensions space of the same length as the sequence input to the encoder. This output x is learned in the encoder layers to serve as a key-value lookup of the form (x, x) (from d_{model}-space into d_{model}-space). This lookup is the information that is passed from the encoder to the decoder, where the decoder queries the lookup from its own input.

3.7.1 Encoder Layer

Each encoder layer consists of a self-attention sublayer followed by a feed-forward sublayer. This is naturally implemented with the Sequential API. The following is (a slightly simplified version of) the function from `encoder.py` which creates the encoder layer.

```
def encoder_layer(d_model, num_heads, dff, dropout_rate):
    mha = MultiHeadSelfAttention(d_model, num_heads, mask_rank=2)
    ffn = pointwise_feed_forward_network(d_model, dff)

    return tf.keras.Sequential([
        TransformerSubLayer(mha, epsilon=1e-6, dropout_rate=dropout_rate),
        TransformerSubLayer(ffn, epsilon=1e-6, dropout_rate=dropout_rate)
    ])
```

The encoder itself can be simply implemented using the Sequential API. The `encoder` function from `encoder.py` implements the entire encoder stack including the input embeddings.

```
def encoder(num_layers, d_model, num_heads, dff, input_vocab_size,
            maximum_position_encoding, dropout_rate):
    layer_list = [
        PaddingMask(mask_value=0),
        ScaledEmbedding(input_vocab_size, d_model, dropout_rate,
                        maximum_position_encoding, positional=True)] + \
        [encoder_layer(d_model, num_heads, dff, dropout_rate)
         for i in range(num_layers)]

    return Sequential(layer_list)
```

3.8 Decoder

The decoder is a stack of N identical decoder layers. The output of the decoder is fed to the model "head" and determines the probability distribution over the target symbols for each element in the decoder input sequence.

3.8.1 Decoder Layer

Each decoder layer begins with a self-attention layer using the auto-regressive mask to prevent position i attending on (i.e. looking up) a position $j > i$. Note that since the target is shifted by one, position i should still looks up itself.
This is followed by the encoder-decoder attention sublayer which uses the output of the self-attention decoder sublayer as a query to the key-value lookup output by the encoder. The third sublayer is the feed-forward sublayer.

We cannot use the Sequential API for the decoder layer since it takes two inputs: the encoder output and the decoder input. It can naturally be implemented using the Functional API. This is (a slightly simplified version of) the decoder layer creation function from `decoder.py`.

```
def decoder_layer(d_model, num_heads, dff, dropout_rate):
    encoder_output = Input(shape=(None, d_model), name='encoder_output')
    decoder_input = Input(shape=(None, d_model), name='decoder_input')

    auto_regress = AutoRegressiveMask()
    mha_self = MultiHeadSelfAttention(d_model, num_heads, mask_rank=4)
    mha_auto_reg = Sequential([auto_regress, mha_self])
    mha_self_sublayer = TransformerSubLayer(mha_auto_reg, epsilon=1e-6,
                                            dropout_rate=dropout_rate)

    mha_2inp = MultiHeadTwoInputAttention(d_model, num_heads)
    mha_2inp_sublayer = TransformerSubLayer(mha_2inp, input_key='queries',
                                            epsilon=1e-6, dropout_rate=dropout_rate)
    ffn = pointwise_feed_forward_network(d_model, dff)
    ffn_sublayer = TransformerSubLayer(ffn, epsilon=1e-6, dropout_rate=dropout_rate)

    out1 = mha_self_sublayer(decoder_input)
    out2 = mha_2inp_sublayer(dict(queries=out1, lookups=encoder_output))
    outputs = ffn_sublayer(out2)

    return Model(inputs=[encoder_output, decoder_input], outputs=outputs)
```

The decoder itself including the inputs can then also be implemented with the Functional API. From `decoder.py`:

```
def decoder(num_layers, d_model, num_heads, dff, target_vocab_size,
            maximum_position_encoding, dropout_rate=0.1):

    encoder_output = Input(shape=(None, d_model), name='encoder_output')
    decoder_input = Input(shape=(None, ), name='decoder_input')

    embedding = Sequential([PaddingMask(),
                            ScaledEmbedding(target_vocab_size,
                                            d_model, dropout_rate,
                                            maximum_position_encoding,
                                            positional=True)])
    decoder_layers = [
        decoder_layer(d_model, num_heads, dff, dropout_rate=dropout_rate)
        for _ in range(num_layers)]

    x = embedding(decoder_input)
    for i in range(num_layers):
        x = decoder_layers[i](dict(decoder_input=x, encoder_output=encoder_output))

    return Model(inputs=[encoder_output, decoder_input], outputs=x)
```

Note that unlike the decoder layers the shape for the `decoder_input` is `(None, )` for the decoder itself, since the inputs are tokenized sequences which have not yet been embedded in d_{model}-space.

3.9 Transformer Model

The final transformer model has two inputs, the tokenized encoder and decoder sequences. First the encoder input is fed to the stack of encoder layers, and the resulting output is combined with the decoder input to feed to the stack of decoder layers. The decoder output is then passed to a final fully connected layer whose dimension is the size of the target vocabulary (i.e. number of symbols), so that the model output can predict target symbols. More precisely, as an auto-regressive model it is predicting the next-symbol probabilities.

In Vaswani et al. (2017 §3.4) it is stated that the encoder and decoder embedding layers and the final fully connected layer all share the same weight matrix, and thus all have the same symbol embedding. This method is introduced in Press and Wolf (2016), where is it claimed for example that using a standard subword tokenizer an English/French translation task show up to 90% of the same subwords. For this method the union of the subwords is taken so that the the source/target vocabularies are the same. In our implementation, the source and target vocabularies are not merged and these three weight matrices are all learned separately.

The transformer can be implemented naturally using the Functional API. The code in `transformer.py` is slightly simplified as follows.

```
def transformer(num_layers, d_model, num_heads, dff,
                input_vocab_size, target_vocab_size,
                pe_input_max, pe_target_max, dropout_rate):

    encoder_input = Input(shape=(None,), name='encoder_input')
    decoder_input = Input(shape=(None,), name='decoder_input')

    encoder_stack = encoder(num_layers, d_model, num_heads, dff,
                            input_vocab_size, pe_input_max, dropout_rate)
    decoder_stack = decoder(num_layers, d_model, num_heads, dff,
                            target_vocab_size, pe_target_max, dropout_rate)
    final_layer = Dense(target_vocab_size)

    encoder_output = encoder_stack(encoder_input)
    decoder_output = decoder_stack(
        dict(decoder_input=decoder_input, encoder_output=encoder_output))
    final_output = final_layer(decoder_output)

    return Model(inputs=[encoder_input, decoder_input], outputs=final_output)
```

It is crucial to note that the mask on the `decoder_output` is propagated through the `transformer` model. The is used by the loss function, which ensures that the padding at the end of the sequences is not included in the loss calculation (but see the caveat in the Loss section). Similarly the mask is used by the metrics, in our case the accuracy metric, ignoring the padding for the metric calculation.

4 Model Usage

Since the transformer model constructed here conforms to the Keras API guidelines, we can naturally use the built-in APIs for training and inference.

4.1 Training

4.1.1 Loss

One of the trickiest aspects of the implementation was getting the loss right. This is one place where the disadvantages of using the higher-level Keras API show up: Less control and less clarity about what is going on behind the scenes. It took some time to notice that losses compiled into the Keras model use the propagated mask to modify the loss calculation as wanted, but do not make the expected/desired reduction afterwards. We expected that simply compiling the built-in `SparseCategoricalCrossentropy` loss into the model would give the correct loss. The compiled losses use the mask on the model output to correctly mask out the losses for irrelevant sequence members, i.e. it zeros the losses corresponding to sequence padding; however, the average is then computed over the entire sequence. For example, if a batch has dimension `(64, 37)`, then while the `64 * 37` loss matrix will have `0`s where there is padding, the final loss is calculated by summing the loss matrix and then calculating the mean by dividing by `64 * 37`. However, to correctly calculate the summarized loss we want to divide by the number of non-masked elements in the batch. While the transformer still learns reasonably well with this built-in loss calculation, is does significantly better with the correct loss.

We could not see anyway to opt out of this behaviour, short of removing the mask from the final output which is a hack and causes the built-in metrics to give incorrect results. To overcome this we added the following “correction factor” to a custom loss, which is also a hack. From `transformer/loss.py`:

```
from tensorflow.keras.losses import Loss, sparse_categorical_crossentropy
```

```
class MaskedSparseCategoricalCrossentropy(Loss):
    def __init__(self, name='masked_sparse_categorical_cross_entropy'):
        super().__init__(name=name)

    def call(self, y_true, y_pred):
        loss = sparse_categorical_crossentropy(y_true, y_pred,
                                               from_logits=True)
        mask = getattr(y_pred, '_keras_mask')
        sw = tf.cast(mask, y_pred.dtype)
        # desired loss value
        reduced_loss = tf.reduce_sum(loss * sw) / tf.reduce_sum(sw)
        # cannot opt out of mask corrections in the API
        correction_factor = tf.reduce_sum(tf.ones(shape=tf.shape(y_true))) / \
                            tf.reduce_sum(sw)

        return reduced_loss * correction_factor
```

4.1.2 Optimization

`transformer/schedule.py`

```
from tensorflow.keras.optimizers.schedules import LearningRateSchedule


class CustomSchedule(LearningRateSchedule):
    def __init__(self, d_model, warmup_steps=4000):
        super(CustomSchedule, self).__init__()
        self.d_model = tf.cast(d_model, tf.float32)
        self.warmup_steps = warmup_steps

    def __call__(self, step):
        arg1 = tf.math.rsqrt(step)
        arg2 = step * (self.warmup_steps ** -1.5)

        return tf.math.rsqrt(self.d_model) * tf.math.minimum(arg1, arg2)
```

The `Adam` optimizer is used with the same settings as in the paper Vaswani et al. (2017).

```
from tensorflow.keras.optimizers import Adam

D_MODEL = 128

learning_rate = CustomSchedule(d_model=D_MODEL)
optimizer = Adam(learning_rate, beta_1=0.9, beta_2=0.98, epsilon=1e-9)
```

4.1.3 Learning

The actual code is in `program.py`. However, the following sequence illustrates how the Keras training API is called.

```
from transformer.transformer import transformer

model = transformer(num_layers=4, d_model=D_MODEL,
                    num_heads=8, dff=512,
                    input_vocab_size=input_vocab_size,
                    target_vocab_size=target_vocab_size,
                    pe_input_max=MAX_LEN,
                    pe_target_max=MAX_LEN,
                    dropout_rate=0.1)

model.compile(optimizer=optimizer,
              loss=MaskedSparseCategoricalCrossentropy(),
              metrics=['accuracy'])

model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    TRAIN_DIR + '/checkpoint.{epoch}.ckpt',
    save_weights_only=True)

model.fit(data_train, epochs=1, validation_data=data_eval,
          callbacks=model_checkpoint_callback)
```

4.2 Inference

Inference with the transformer, or any auto-regressive model, is not simply a matter of plugging a testing pipeline into the model and calling `predict`. The training process uses teacher forcing as previously discussed, which means the next symbol is predicted based on the given ground truth up to that point in the sequence. In contrast, during inference the sequence of predicted symbols is used to recursively predict the next symbol. The code for doing this is in `transformer/autoregression.py`:

```
def autoregress(model, input, delimiters, max_length):
    delimiters = delimiters[0]
    decoder_input = [delimiters[0]]

    output = tf.expand_dims(decoder_input, 0)

    done = False
    while not done:
        preds = model({'encoder_input': tf.expand_dims(input, 0), 'decoder_input': output})
        prediction = preds[:, -1, :]
        pred_id = tf.argmax(prediction, axis=-1) \
            if tf.shape(output)[1] < max_length - 1 else tf.expand_dims(delimiters[1], 0)

        done = pred_id == delimiters[1]
        output = tf.concat([output, tf.expand_dims(pred_id, 0)], axis=-1)

    return tf.squeeze(output, axis=0)


def translate(model, input, tokenizers, max_length):
    """
    Translate an input sentence to a target sentence using a model
    """
    input_encoded = tokenizers.inputs.tokenize([input])[0]

    if len(input_encoded) > max_length:
        return None

    prediction = autoregress(model,
                             input_encoded,
                            delimiters=tokenizers.targets.tokenize(['']),
                            max_length=max_length)
    prediction_decoded = tokenizers.targets.detokenize([prediction]).numpy()[0][0].decode('utf-8')

    return prediction_decoded
```

References

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. "Layer Normalization." https://arxiv.org/pdf/1607.06450.pdf.

Press, Ofir, and Lior Wolf. 2016. "Using the Output Embedding to Improve Language Models." *CoRR* abs/1608.05859. http://arxiv.org/abs/1608.05859.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. "Attention Is All You Need." *CoRR* abs/1706.03762. http://arxiv.org/abs/1706.03762.