Speech Recognition using Transformers

Introduction

Problem statement:

The task of automatic speech recognition (ASR) involves accurately transcribing spoken words into written text. This is a complex problem, as it requires mapping a sequence of audio features to a corresponding sequence of characters, words, or subword tokens. In addition, ASR must account for variations in pronunciation, accents, and speaking styles, making it a challenging task for both humans and machines.

For the screening test i have used the LJSpeech dataset from the [LibriVox](https://librivox.org) project. It consists of short audio clips of a single speaker reading passages from 7 non-fiction books. the model will be similar to the original Transformer (both encoder and decoder) as proposed in the paper, "Attention is All You Need".

```
import os
import random
from glob import glob
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

Define the Transformer Input Layer

When processing past target tokens for the decoder, we compute the sum of position embeddings and token embeddings.

When processing audio features, we apply convolutional layers to downsample them (via convolution stides) and process local relationships.

```
class TokenEmbedding(layers.Layer):
    """
    A Keras layer that combines token embeddings with positional embeddings.

    Args:
        num_vocab (int): The size of the vocabulary, i.e. the maximum integer index + 1.
        maxlen (int): The maximum length of input sequences.
        num_hid (int): The dimensionality of the embedding space.

    Input shape:
        2D tensor with shape `(batch_size, sequence_length)`.

    Output shape:
        3D tensor with shape `(batch_size, sequence_length, num_hid)`.

    """

    def __init__(self, num_vocab=1000, maxlen=100, num_hid=64):
```

```
        super().__init__()
        self.emb = tf.keras.layers.Embedding(num_vocab, num_hid)
        self.pos_emb = layers.Embedding(input_dim=maxlen, output_dim=num_hid)

    def call(self, x):
        """
        Compute token embeddings and positional embeddings, and add them together.

        Args:
            x (tf.Tensor): The input tensor, with shape `(batch_size, sequence_length)`.

        Returns:
            The output tensor, with shape `(batch_size, sequence_length, num_hid)`.

        """
        maxlen = tf.shape(x)[-1]
        x = self.emb(x)
        positions = tf.range(start=0, limit=maxlen, delta=1)
        positions = self.pos_emb(positions)
        return x + positions


class SpeechFeatureEmbedding(layers.Layer):
    """
    A Keras layer that processes speech features using convolutional neural networks.

    Args:
        num_hid (int): The number of filters in each convolutional layer.
        maxlen (int): The maximum length of input sequences.

    Input shape:
        3D tensor with shape `(batch_size, num_frames, num_features)`.

    Output shape:
        3D tensor with shape `(batch_size, sequence_length, num_hid)`.

    """

    def __init__(self, num_hid=64, maxlen=100):
        super().__init__()
        self.conv1 = tf.keras.layers.Conv1D(
            num_hid, 11, strides=2, padding="same", activation="relu"
        )
        self.conv2 = tf.keras.layers.Conv1D(
            num_hid, 11, strides=2, padding="same", activation="relu"
        )
        self.conv3 = tf.keras.layers.Conv1D(
            num_hid, 11, strides=2, padding="same", activation="relu"
        )
        self.pos_emb = layers.Embedding(input_dim=maxlen, output_dim=num_hid)

    def call(self, x):
        """
        Apply three convolutional layers to the input tensor, and return the result.

        Args:
```

```
            x (tf.Tensor): The input tensor, with shape `(batch_size, num_frames, num_features)`.

        Returns:
            The output tensor, with shape `(batch_size, sequence_length, num_hid)`.

        """
        x = self.conv1(x)
        x = self.conv2(x)
        return self.conv3(x)
```

Transformer Encoder Layer

The encoder consists of multiple identical layers, each of which contains a multi-head attention mechanism followed by a feed-forward neural network (FFN).

The TransformerEncoder takes as input a tensor inputs and applies three main operations:

Multi-head attention: The tensor is passed through a layers.MultiHeadAttention layer with num_heads heads and key_dim equal to embed_dim. The output is a tensor that contains information about how each position in the input sequence is related to all other positions.

Feed-forward network: The output of the multi-head attention layer is passed through a feed-forward network (FFN) composed of two dense layers with ReLU activation. The output of the FFN is a tensor with the same shape as the input tensor.

Residual connections and layer normalization: The output of the FFN is added to the input tensor (with an intermediate normalization step), and the resulting tensor is passed through another layer normalization step.

```
class TransformerEncoder(layers.Layer):
    """
    A Transformer encoder layer that consists of a multi-head self-attention mechanism
    and a feedforward neural network. Layer normalization and dropout are also applied
    before and after each sub-layer.

    Args:
        embed_dim (int): Dimensionality of the input and output embeddings.
        num_heads (int): Number of attention heads to use.
        feed_forward_dim (int): Dimensionality of the feedforward layer.
        rate (float): Dropout rate to apply.

    Returns:
        A tensor of the same shape as the input tensor, representing the output of the
        Transformer encoder layer.

    """
    def __init__(self, embed_dim, num_heads, feed_forward_dim, rate=0.1):
        super().__init__()
        self.att = layers.MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
        self.ffn = keras.Sequential(
            [
                layers.Dense(feed_forward_dim, activation="relu"),
```

```
                layers.Dense(embed_dim),
            ]
        )
        self.layernorm1 = layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = layers.LayerNormalization(epsilon=1e-6)
        self.dropout1 = layers.Dropout(rate)
        self.dropout2 = layers.Dropout(rate)

    def call(self, inputs, training):
        """
        Perform a forward pass through the Transformer encoder layer.

        Args:
            inputs (tensor): Input tensor of shape (batch_size, seq_len, embed_dim).
            training (bool): Whether the layer is in training mode or not.

        Returns:
            A tensor of the same shape as the input tensor, representing the output of the
            Transformer encoder layer.

        """
        attn_output = self.att(inputs, inputs)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output)
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        return self.layernorm2(out1 + ffn_output)
```

Transformer Decoder Layer

The Transformer Decoder layer has several components, including multi-head self-attention, multi-head attention with an encoder output, feed-forward network, and layer normalization.

The self-attention component allows the model to attend to different positions in the input sequence and the encoder output component allows the model to consider the context of the input sequence. The feed-forward network applies non-linear transformations to the output of the attention components.

Layer normalization is applied before and after each component to improve training stability.

```
class TransformerDecoder(layers.Layer):
    """
    TransformerDecoder layer of the Transformer model architecture.

    It consists of a self-attention mechanism and an encoder-decoder attention mechanism,
    followed by a feedforward neural network (FFN) layer. The layer also applies
    layer normalization and dropout regularization.

    Args:
        embed_dim (int): Dimensionality of the embedding space.
        num_heads (int): Number of attention heads to use.
```

```
        feed_forward_dim (int): Dimensionality of the FFN layer.
        dropout_rate (float): Dropout rate to use for regularization.

    Attributes:
        layernorm1 (LayerNormalization): Layer normalization for the self-attention output.
        layernorm2 (LayerNormalization): Layer normalization for the encoder-decoder attention output.
        layernorm3 (LayerNormalization): Layer normalization for the FFN output.
        self_att (MultiHeadAttention): Self-attention mechanism.
        enc_att (MultiHeadAttention): Encoder-decoder attention mechanism.
        self_dropout (Dropout): Dropout layer for the self-attention output.
        enc_dropout (Dropout): Dropout layer for the encoder-decoder attention output.
        ffn_dropout (Dropout): Dropout layer for the FFN output.
        ffn (Sequential): FFN layer.

    Methods:
        causal_attention_mask(batch_size, n_dest, n_src, dtype): Creates a causal attention mask.
        call(enc_out, target): Applies the TransformerDecoder layer to the input.

    """
    def __init__(self, embed_dim, num_heads, feed_forward_dim, dropout_rate=0.1):
        """
        Initializes a new instance of the TransformerDecoder layer.

        Args:
            embed_dim (int): Dimensionality of the embedding space.
            num_heads (int): Number of attention heads to use.
            feed_forward_dim (int): Dimensionality of the FFN layer.
            dropout_rate (float): Dropout rate to use for regularization.

        """
        super().__init__()

        # Layer normalization
        self.layernorm1 = layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = layers.LayerNormalization(epsilon=1e-6)
        self.layernorm3 = layers.LayerNormalization(epsilon=1e-6)

        # Self-attention and encoder-attention
        self.self_att = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim
        )
        self.enc_att = layers.MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)

        # Dropout layers
        self.self_dropout = layers.Dropout(dropout_rate)
        self.enc_dropout = layers.Dropout(dropout_rate)
        self.ffn_dropout = layers.Dropout(dropout_rate)

        # Feedforward layer
        self.ffn = keras.Sequential(
            [
                layers.Dense(feed_forward_dim, activation="relu"),
                layers.Dense(embed_dim),
            ]
        )
```

```
    def causal_attention_mask(self, batch_size, n_dest, n_src, dtype):
        """
        Creates a causal attention mask.

        The mask prevents flow of information from future tokens to current token by
        masking the upper half of the dot product matrix in self-attention.

        Args:
            batch_size (int): Size of the input batch.
            n_dest (int): Number of target tokens.
            n_src (int): Number of source tokens.
            dtype (dtype): Data type to use for the mask.

        Returns:
            Tensor: Causal attention mask tensor.

        """
        i = tf.range(n_dest)[:, None]
        j = tf.range(n_src)
        m = i >= j - n_src + n_dest
        mask = tf.cast(m, dtype)
        mask = tf.reshape(mask, [1, n_dest, n_src])
        mult = tf.concat(
            [tf.expand_dims(batch_size, -1), tf.constant([1, 1], dtype=tf.int32)], 0
        )
        return tf.tile(mask, mult)

    def call(self, enc_out, target):
        """
        Performs the forward pass of the decoder layer.

        Args:
            enc_out (Tensor): Output of the encoder layer with shape (batch_size, seq_len, embed_dim).
            target (Tensor): Input to the decoder layer with shape (batch_size, seq_len, embed_dim).

        Returns:
            Output of the decoder layer with shape (batch_size, seq_len, embed_dim).
        """
        input_shape = tf.shape(target)
        batch_size = input_shape[0]
        seq_len = input_shape[1]
        causal_mask = self.causal_attention_mask(batch_size, seq_len, seq_len, tf.bool)
        target_att = self.self_att(target, target, attention_mask=causal_mask)
        target_norm = self.layernorm1(target + self.self_dropout(target_att))
        enc_out = self.enc_att(target_norm, enc_out)
        enc_out_norm = self.layernorm2(self.enc_dropout(enc_out) + target_norm)
        ffn_out = self.ffn(enc_out_norm)
        ffn_out_norm = self.layernorm3(enc_out_norm + self.ffn_dropout(ffn_out))
        return ffn_out_norm
```

Complete the Transformer model

The model takes audio spectrograms as inputs and predicts a sequence of characters. During training, we give the decoder the target character sequence shifted to the left as input. During inference, the decoder uses its own past predictions to predict the next token.

```
class Transformer(keras.Model):
    """
    Transformer model for sequence-to-sequence tasks.

    Args:
        num_hid (int): Number of hidden units in each Transformer layer.
        num_head (int): Number of attention heads in each Transformer layer.
        num_feed_forward (int): Number of units in the feedforward network in each Transformer layer.
        source_maxlen (int): Maximum length of input sequences.
        target_maxlen (int): Maximum length of target sequences.
        num_layers_enc (int): Number of Transformer encoder layers.
        num_layers_dec (int): Number of Transformer decoder layers.
        num_classes (int): Number of classes in the output vocabulary.

    Attributes:
        loss_metric (keras.metrics.Mean): Mean loss metric for tracking loss during training.
        num_layers_enc (int): Number of Transformer encoder layers.
        num_layers_dec (int): Number of Transformer decoder layers.
        target_maxlen (int): Maximum length of target sequences.
        num_classes (int): Number of classes in the output vocabulary.
        enc_input (SpeechFeatureEmbedding): Speech feature embedding layer.
        dec_input (TokenEmbedding): Token embedding layer for decoder inputs.
        encoder (keras.Sequential): Transformer encoder.
        classifier (keras.layers.Dense): Final dense layer for classification.

    Methods:
        decode(enc_out, target):
            Decodes target sequences using the encoder output.
        call(inputs):
            Executes the forward pass of the Transformer model.
        train_step(batch):
            Processes one batch during training.
        test_step(batch):
            Processes one batch during testing.
        generate(source, target_start_token_idx):
            Performs inference over one batch of inputs using greedy decoding.
    """
    def __init__(
        self,
        num_hid=64,
        num_head=2,
        num_feed_forward=128,
        source_maxlen=100,
        target_maxlen=100,
        num_layers_enc=4,
        num_layers_dec=1,
        num_classes=10,
    ):
        super().__init__()
        self.loss_metric = keras.metrics.Mean(name="loss")
        self.num_layers_enc = num_layers_enc
```

```
        self.num_layers_dec = num_layers_dec
        self.target_maxlen = target_maxlen
        self.num_classes = num_classes

        self.enc_input = SpeechFeatureEmbedding(num_hid=num_hid, maxlen=source_maxlen)
        self.dec_input = TokenEmbedding(
            num_vocab=num_classes, maxlen=target_maxlen, num_hid=num_hid
        )

        self.encoder = keras.Sequential(
            [self.enc_input]
            + [
                TransformerEncoder(num_hid, num_head, num_feed_forward)
                for _ in range(num_layers_enc)
            ]
        )

        for i in range(num_layers_dec):
            setattr(
                self,
                f"dec_layer_{i}",
                TransformerDecoder(num_hid, num_head, num_feed_forward),
            )

        self.classifier = layers.Dense(num_classes)

    def decode(self, enc_out, target):
        """
        Decodes target sequences using the encoder output.

        Args:
            enc_out (tf.Tensor): Encoder output tensor.
            target (tf.Tensor): Target sequences tensor.

        Returns:
            Decoded target sequences tensor.
        """
        y = self.dec_input(target)
        for i in range(self.num_layers_dec):
            y = getattr(self, f"dec_layer_{i}")(enc_out, y)
        return y

    def call(self, inputs):
        """
        Executes the forward pass of the Transformer model.

        Args:
            inputs (tuple): Tuple of input sequences.

        Returns:
            Model output tensor.
        """
        source = inputs[0]
        target = inputs[1]
        x = self.encoder(source)
        y = self.decode(x, target)
```

```
        return self.classifier(y)

    @property
    def metrics(self):
        """
        Returns the metrics that the model should track during training and testing.

        Returns:
            List of metrics.
        """
        return [self.loss_metric]

    def train_step(self, batch):
        """
        Executes one training step on a batch of data.

        Args:
            batch (dict): Dictionary containing the batch data.

        Returns:
            Dictionary with the loss value.
        """
        source = batch["source"]
        target = batch["target"]
        dec_input = target[:, :-1]
        dec_target = target[:, 1:]
        with tf.GradientTape() as tape:
            preds = self([source, dec_input])
            one_hot = tf.one_hot(dec_target, depth=self.num_classes)
            mask = tf.math.logical_not(tf.math.equal(dec_target, 0))
            loss = self.compiled_loss(one_hot, preds, sample_weight=mask)
        trainable_vars = self.trainable_variables
        gradients = tape.gradient(loss, trainable_vars)
        self.optimizer.apply_gradients(zip(gradients, trainable_vars))
        self.loss_metric.update_state(loss)
        return {"loss": self.loss_metric.result()}

    def test_step(self, batch):
      """
      Executes one evaluation step on a batch of data.

      Args:
          batch (dict): Dictionary containing the batch data.

      Returns:
          Dictionary with the loss value.
      """
        source = batch["source"]
        target = batch["target"]
        dec_input = target[:, :-1]
        dec_target = target[:, 1:]
        preds = self([source, dec_input])
        one_hot = tf.one_hot(dec_target, depth=self.num_classes)
        mask = tf.math.logical_not(tf.math.equal(dec_target, 0))
        loss = self.compiled_loss(one_hot, preds, sample_weight=mask)
        self.loss_metric.update_state(loss)
```

```
        return {"loss": self.loss_metric.result()}

    def generate(self, source, target_start_token_idx):
        """
        Generates target sequences given a source sequence using greedy decoding.

        Args:
            source (tf.Tensor): Source sequence tensor.
            target_start_token_idx (int): Index of the start token in the target vocabulary.

        Returns:
            Generated target sequences tensor.
        """
        bs = tf.shape(source)[0]
        enc = self.encoder(source)
        dec_input = tf.ones((bs, 1), dtype=tf.int32) * target_start_token_idx
        dec_logits = []
        for i in range(self.target_maxlen - 1):
            dec_out = self.decode(enc, dec_input)
            logits = self.classifier(dec_out)
            logits = tf.argmax(logits, axis=-1, output_type=tf.int32)
            last_logit = tf.expand_dims(logits[:, -1], axis=-1)
            dec_logits.append(last_logit)
            dec_input = tf.concat([dec_input, last_logit], axis=-1)
        return dec_input
```

Download the dataset

Note: This requires ~3.6 GB of disk space and takes ~5 minutes for the extraction of files.

```
def download_and_extract_data(download_url, extract_to, archive_format="tar"):
    """
    Downloads a file from a given URL and extracts it to the specified location.

    Args:
        download_url (str): The URL to download the file from.
        extract_to (str): The path to extract the downloaded file to.
        archive_format (str, optional): The archive format of the downloaded file. Defaults to "tar".

    Returns:
        None
    """
    # Download and extract the file
    keras.utils.get_file(os.path.join(os.getcwd(), extract_to), download_url, extract=True, archive_format=archive_format, cache_dir=".")

def get_data(wavs, id_to_text, maxlen=50):
    """
    Returns a mapping of audio paths and transcription texts for the given audio files.

    Args:
        wavs (list): A list of paths to audio files.
        id_to_text (dict): A dictionary mapping audio file IDs to their transcriptions.
        maxlen (int, optional): The maximum length of the transcription text. Defaults to 50.
```

```
    Returns:
        list: A list of dictionaries with keys "audio" and "text", where "audio" is a path to an audio file
            and "text" is the corresponding transcription text.
    """
    data = []
    for w in wavs:
        id = w.split("/")[-1].split(".")[0]
        if len(id_to_text[id]) < maxlen:
            data.append({"audio": w, "text": id_to_text[id]})
    return data

def run_data_pipeline():
    """
    Downloads and extracts the LJSpeech-1.1 dataset, and returns a list of audio files and their corresponding transcription texts.

    Args:
        None

    Returns:
        list: A list of dictionaries with keys "audio" and "text", where "audio" is a path to an audio file
            and "text" is the corresponding transcription text.
    """
    # Download and extract the data
    download_url = "https://data.keithito.com/data/speech/LJSpeech-1.1.tar.bz2"
    extract_to = "./datasets/LJSpeech-1.1"
    download_and_extract_data(download_url, extract_to, archive_format="tar")

    # Load the transcription texts
    id_to_text = {}
    with open(os.path.join(extract_to, "metadata.csv"), encoding="utf-8") as f:
        for line in f:
            id = line.strip().split("|")[0]
            text = line.strip().split("|")[2]
            id_to_text[id] = text

    # Get the audio files and their transcriptions
    wavs = glob("{}/**/*.wav".format(extract_to), recursive=True)
    data = get_data(wavs, id_to_text, maxlen=50)

    return data

run_data_pipeline()
```

```
Downloading data from https://data.keithito.com/data/speech/LJSpeech-1.1.tar.bz2
2748572632/2748572632 [==============================] - 211s 0us/step
```

Preprocess the dataset

```
class VectorizeChar:
    """
    A class for vectorizing characters in a given text using a pre-defined vocabulary.
    Attributes:
```

```
      - vocab (list): A list of characters representing the vocabulary.
      - max_len (int): The maximum length of the input text after being pre-processed.
      - char_to_idx (dict): A dictionary mapping characters in the vocabulary to their corresponding indices.

    Methods:
      - __call__(text): Vectorizes the given text using the pre-defined vocabulary, padding it to the specified max_len.
      - get_vocabulary(): Returns the vocabulary used for vectorization.
    """
    def __init__(self, max_len=50):
        """
        Initializes the VectorizeChar class.

        Args:
        - max_len (int): The maximum length of the input text after being pre-processed.
        """
        self.vocab = (
            ["-", "#", "<", ">"]
            + [chr(i + 96) for i in range(1, 27)]
            + [" ", ".", ",", "?"]
        )
        self.max_len = max_len
        self.char_to_idx = {}
        for i, ch in enumerate(self.vocab):
            self.char_to_idx[ch] = i

    def __call__(self, text):
        """
        Vectorizes the given text using the pre-defined vocabulary and pads it to the specified max_len.

        Args:
        - text (str): The input text to be vectorized.

        Returns:
        - A list of integers representing the vectorized text with padding.
        """
        text = text.lower()
        text = text[: self.max_len - 2]
        text = "<" + text + ">"
        pad_len = self.max_len - len(text)
        return [self.char_to_idx.get(ch, 1) for ch in text] + [0] * pad_len

    def get_vocabulary(self):
        """
        Returns the vocabulary used for vectorization.

        Returns:
        - A list of characters representing the vocabulary.
        """
        return self.vocab


max_target_len = 200  # all transcripts in out data are < 200 characters
data = get_data(wavs, id_to_text, max_target_len)
vectorizer = VectorizeChar(max_target_len)
print("vocab size", len(vectorizer.get_vocabulary()))
```

```
def create_text_ds(data):
    """
    Creates a Tensorflow dataset of vectorized text data from the given data dictionary.

    Args:
      - data (list): A list of dictionaries containing the "text" key with text data.

    Returns:
      - A Tensorflow dataset of vectorized text data.
    """
    texts = [_["text"] for _ in data]
    text_ds = [vectorizer(t) for t in texts]
    text_ds = tf.data.Dataset.from_tensor_slices(text_ds)
    return text_ds


def path_to_audio(path):
    """
    Converts an audio file from the given path to a spectrogram tensor using short-time Fourier transform (STFT).
    Args:
      - path (str): The path to the audio file.

    Returns:
      - A Tensorflow tensor representing the spectrogram of the audio file.
    """
    # spectrogram using stft
    audio = tf.io.read_file(path)
    audio, _ = tf.audio.decode_wav(audio, 1)
    audio = tf.squeeze(audio, axis=-1)
    stfts = tf.signal.stft(audio, frame_length=200, frame_step=80, fft_length=256)
    x = tf.math.pow(tf.abs(stfts), 0.5)
    # normalisation
    means = tf.math.reduce_mean(x, 1, keepdims=True)
    stddevs = tf.math.reduce_std(x, 1, keepdims=True)
    x = (x - means) / stddevs
    audio_len = tf.shape(x)[0]
    # padding to 10 seconds
    pad_len = 2754
    paddings = tf.constant([[0, pad_len], [0, 0]])
    x = tf.pad(x, paddings, "CONSTANT")[:pad_len, :]
    return x


def create_audio_ds(data):
    """
    Creates a Tensorflow dataset of spectrogram data from the given data dictionary.
    Args:
      - data (list): A list of dictionaries containing the "audio" key with audio file paths.

    Returns:
      - A Tensorflow dataset of spectrogram data.
    """
    flist = [_["audio"] for _ in data]
    audio_ds = tf.data.Dataset.from_tensor_slices(flist)
    audio_ds = audio_ds.map(
```

```
        path_to_audio, num_parallel_calls=tf.data.AUTOTUNE
    )
    return audio_ds


def create_tf_dataset(data, bs=4):
    """
    Creates a Tensorflow dataset from the given data dictionary with batch size bs.
    Args:
    - data (list): A list of dictionaries containing the "audio" and "text" keys with audio file paths and text data respectively.
    - bs (int): The batch size for the Tensorflow dataset (default=4).

    Returns:
    - A Tensorflow dataset of audio and text pairs.
    """

    audio_ds = create_audio_ds(data)
    text_ds = create_text_ds(data)
    ds = tf.data.Dataset.zip((audio_ds, text_ds))
    ds = ds.map(lambda x, y: {"source": x, "target": y})
    ds = ds.batch(bs)
    ds = ds.prefetch(tf.data.AUTOTUNE)
    return ds


split = int(len(data) * 0.99)
train_data = data[:split]
test_data = data[split:]
ds = create_tf_dataset(train_data, bs=64)
val_ds = create_tf_dataset(test_data, bs=4)
```

```
vocab size 34
```

Callbacks to display predictions

```
class DisplayOutputs(keras.callbacks.Callback):
    def __init__(
        self, batch, idx_to_token, target_start_token_idx=27, target_end_token_idx=28
    ):
        """Displays a batch of outputs after every epoch

        Args:
            batch: A test batch containing the keys "source" and "target"
            idx_to_token: A List containing the vocabulary tokens corresponding to their indices
            target_start_token_idx: A start token index in the target vocabulary
            target_end_token_idx: An end token index in the target vocabulary
        """
        self.batch = batch
        self.target_start_token_idx = target_start_token_idx
        self.target_end_token_idx = target_end_token_idx
        self.idx_to_char = idx_to_token

    def on_epoch_end(self, epoch, logs=None):
```

```
        if epoch % 5 != 0:
            return
        source = self.batch["source"]
        target = self.batch["target"].numpy()
        bs = tf.shape(source)[0]
        preds = self.model.generate(source, self.target_start_token_idx)
        preds = preds.numpy()
        for i in range(bs):
            target_text = "".join([self.idx_to_char[_] for _ in target[i, :]])
            prediction = ""
            for idx in preds[i, :]:
                prediction += self.idx_to_char[idx]
                if idx == self.target_end_token_idx:
                    break
            print("\n")
            print(f"target:     {target_text.replace('-','')}")
            print(f"prediction: {prediction}\n")



batch = next(iter(val_ds))

# The vocabulary to convert predicted indices into characters
idx_to_char = vectorizer.get_vocabulary()
display_cb = DisplayOutputs(
    batch, idx_to_char, target_start_token_idx=2, target_end_token_idx=3
)  # set the arguments as per vocabulary index for '<' and '>'

model = Transformer(
    num_hid=200,
    num_head=2,
    num_feed_forward=400,
    target_maxlen=max_target_len,
    num_layers_enc=4,
    num_layers_dec=1,
    num_classes=34,
)
loss_fn = tf.keras.losses.CategoricalCrossentropy(
    from_logits=True, label_smoothing=0.1,
)

learning_rate = 0.0005
optimizer = keras.optimizers.Adam(learning_rate)
model.compile(optimizer=optimizer, loss=loss_fn)

history = model.fit(ds, validation_data=val_ds, callbacks=[display_cb], epochs= 50)
```

```
Epoch 1/50
203/203 [==============================] - ETA: 0s - loss: 1.3453

target:     <barnett estimated that approximately three minutes elapsed between the time he heard the last of the shots and the time he started guarding the front door.>
prediction: <the as and athe the athe the the athe the the the the are the athe athe the the athe the the are the the are are are the the the are the the the the the the the the tent tenned



target:     <was introduced as early as seventeen ninety by mr. blackburn>
prediction: <the as the athe the athe the the the athe the the the are the as as the athe the the the the are the the the are are the the the are the the the.>
```

```
target:     <the five hundred block of north beckley is five blocks south of the roominghouse.>
prediction: <the as the athe the athe the the as are the athe are the athe the the the athe the the the are the the the the the the the the the the the the ale.>



target:     <the scaffold hung with black# and the inhabitants of the neighborhood, having petitioned the sheriffs to remove the scene of execution to the old place,>
prediction: <the as and athe the athe the the the athe the the the are the athe as the the athe the the the are are the an the the as the the are the the the the the the the the the tent tenned

203/203 [==============================] - 240s 995ms/step - loss: 1.3453 - val_loss: 1.3817
Epoch 2/50
203/203 [==============================] - 196s 965ms/step - loss: 1.3043 - val_loss: 1.3723
Epoch 3/50
203/203 [==============================] - 194s 953ms/step - loss: 1.2933 - val_loss: 1.3570
Epoch 4/50
203/203 [==============================] - 196s 964ms/step - loss: 1.2599 - val_loss: 1.2944
Epoch 5/50
203/203 [==============================] - 195s 961ms/step - loss: 1.1770 - val_loss: 1.2156
Epoch 6/50
203/203 [==============================] - ETA: 0s - loss: 1.1202

target:     <barnett estimated that approximately three minutes elapsed between the time he heard the last of the shots and the time he started guarding the front door.>
prediction: <the commission of the the sarried the and the sand the sand the the sarding the the prison the presiden the the the the the the the the thend the thend s te thenthendententhe the t



target:     <was introduced as early as seventeen ninety by mr. blackburn>
prediction: <the secret the secret the secret the secret the secret the sevent the sight.>



target:     <the five hundred block of north beckley is five blocks south of the roominghouse.>
prediction: <the commission of the the sarding the and the sand the sand the the sight the sand the sand.>



target:     <the scaffold hung with black# and the inhabitants of the neighborhood, having petitioned the sheriffs to remove the scene of execution to the old place,>
prediction: <the secret the secret the servent the secret the sand the sand the sight the sand the secret the secret the the the the the the the thenthend thentend tente.>

203/203 [==============================] - 200s 986ms/step - loss: 1.1202 - val_loss: 1.1715
Epoch 7/50
203/203 [==============================] - 194s 954ms/step - loss: 1.0885 - val_loss: 1.1470
Epoch 8/50
```

References:

- [Attention is All You Need]()
- [Very Deep Self-Attention Networks for End-to-End Speech Recognition]()
- [Speech Transformers]()
- [LJSpeech Dataset]()