Automatic Speech Recognition - Improving Whisper's Robustness

Alex Adams and Jake Walper

April 23, 2024

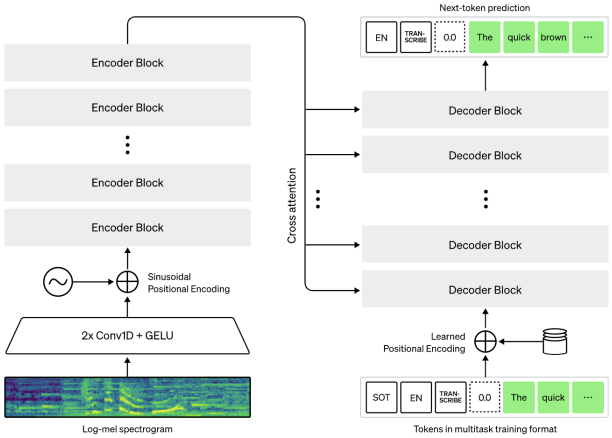
Outline

Topic and Relevance

- The focus of our project will be on improving the open source ASR model Whisper to account for accented speech.
- ASR is the process of using machine learning to transform spoken words into readable text. Common, every day examples include Siri and Alexa.
- Current projections show 90% of technical support helplines for S&P 500 companies such as Apple and Microsoft will be employing ASR models to field calls by as early as 2030.

Interest and Approach

- Whisper is one of the most accurate and cutting-edge ASR models today with close to human level accuracy, but still has room for improvement.
- We used the Whisper model for transfer learning, and fine-tuned the model to improve accuracy on accented English.
- The model success was determined by the word error rate (WER) of a test data set relative to the original model.

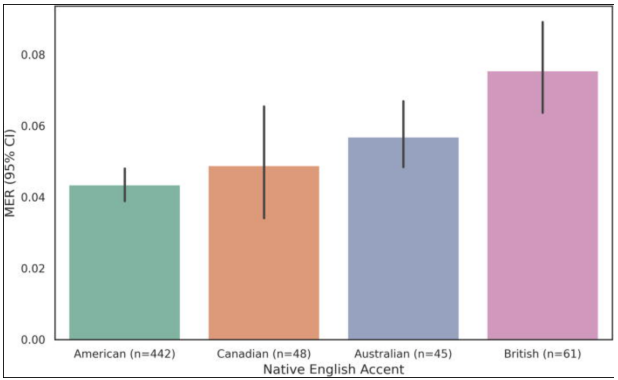


Whisper Origins

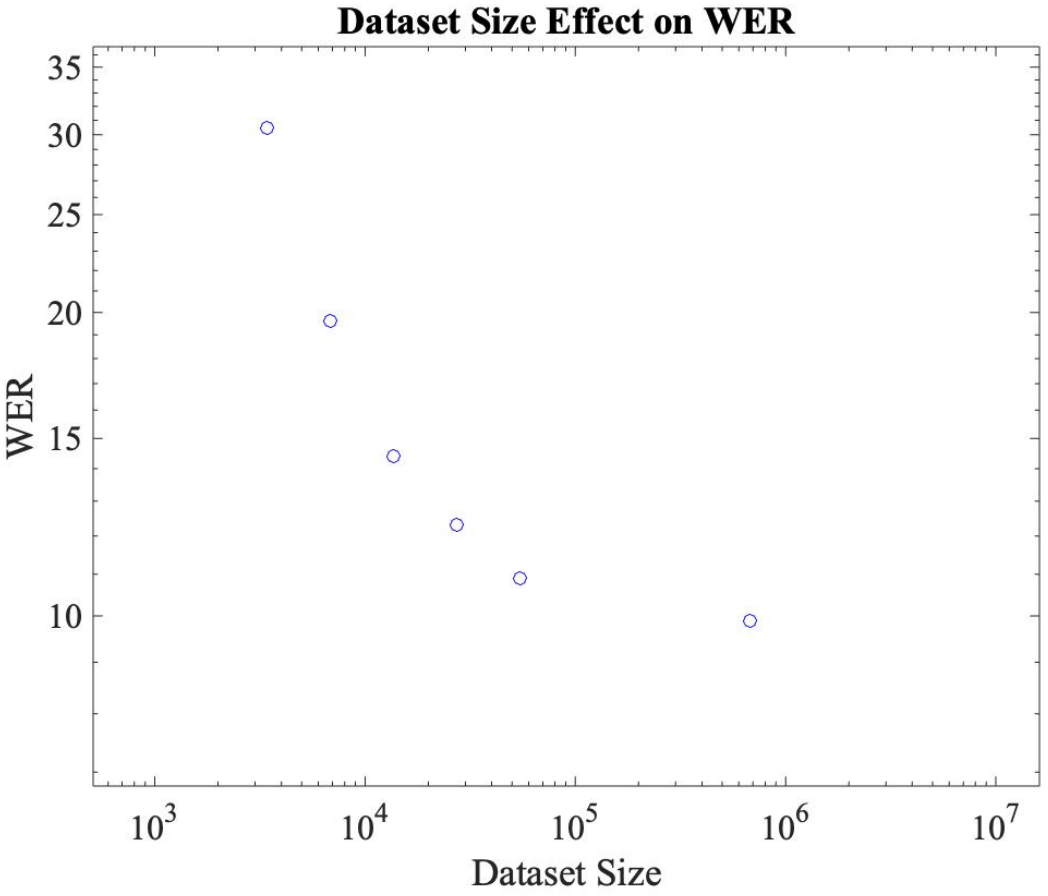
- Radford et al. published a paper titled ”Robust Speech Recognition via Large-Scale Weak Supervision” which introduced Whisper as an alternative to current ASR models.
  - The model was trained with 680,000 hours of multilingual and multitask supervised data.
- The primary motivation for the creation of Whisper was to implement supervised pre-training techniques on a wide range of different data sets instead of the commonly used unsupervised approach that had become popular with the embedding technique Wav2vec.
  - While the data could be trained without reference to human labeling, the researchers found that the models using this technique still made many basic labeling mistakes.

Limitations

- A number of limitations to Whisper were discussed in the publication, inclusing the following:
  - Room for additional high-quality supervised datasets for long-form transcription.
  - Poor Performance on languages that are not English.
  - Additional fine tuning of model could improve performance.
- A separate paper published by the Acoustical Society of America found another limitation to Whisper was its inability to recognize accented language.
  - British, Australian, and even Canadian English accents were found to perform significantly worse than American accents.



Limitations



Accented Speech

- Whisper's inability to capture accented English speech effectively is due to the training datasets used.
  - The Radford group estimated they used less than 100 hours of speech with accents that differed from standard, Midwestern American accent.
  - Additionally, the group used less than 1000 hours of speech with a language other than English.
- While the problem of other language recognition is one that deserves attention, the problem we will focus on is finding a way to improve the WER for accented speech by fine-tuning the Whisper model with specific datasets that have accents different from American English.
- The accuracy of our fine-tuned model will be compared against the traditional Whisper model using WER.

Dataset

- The dataset to be used for the project comes from Hugging Face
  - 20 Hours of Accented speech
  - Each speech segment is between 5 and 10 seconds in length
  - Over 173 different accents accounted for
  - 2 hours of testing data with all 173 different accents

General Architecture and Training Procedure

- We will start by preprocessing the data into usable embeddings for our model.
- For model fine tuning, we loaded the parameters from the existing whisper model ”openai/whisper-small.”
  - It should be noted the Whisper family has 5 model sizes, varying in number of layers. The small size is the third largest model in the family, with 12 layers and 244 M parameters.
- We additionally used the same transformer architecture as the whisper model for our training due to the rigors of developing our own transformer. This is the same general transformer architecture outlined by Vaswani et al. in the famous paper ”Attention is all you Need.”

Loading and Printing Dataset

```
1 from datasets import load_dataset, DatasetDict
2
3 common_voice = DatasetDict()
4
5 common_voice["train"] = load_dataset("DTU54DL/common-accent", split="train",
      use_auth_token=True)
6 common_voice["test"] = load_dataset("DTU54DL/common-accent", split="test",
      use_auth_token=True)
7
8 print(common_voice)
```

Waveform to Mel-Spectogram

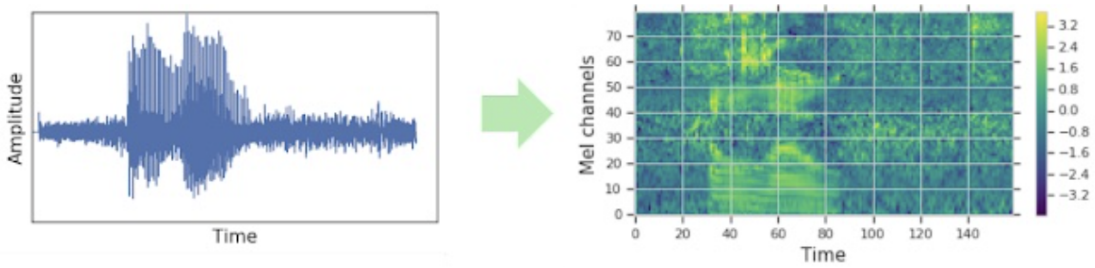
- One of the unique aspects of ASR is converting raw audio data into applicable embeddings.
- The most common approach takes a waveform (.wav) audio file, and converts the data into a spectogram using a Fourier transform to convert the aplitudes of the waveform into separate frequency values, and then scaling this data for human recognition, using a mel-scale.

$$\mathscr{F}(f(t)) = F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt$$

$$m = 2595 log_{10}(1 + \frac{\omega}{700})$$

Waveform to Mel-Spectogram

- The Whisper feature extractor performs two operations:
  - Pads / truncates the audio inputs to 30s: any audio inputs shorter than 30s are padded to 30s with silence (zeros), and those longer that 30s are truncated to 30s
  - Converts the audio inputs to log-Mel spectrogram input features, a visual representation of the audio and the form of the input expected by the Whisper model
- This is a little misleading, as the feature extractor used in Whisper does not actually pass the raw audio through a CNN layer to extract the necessary features from the graph. This is performed when calling the main model itself.



Waveform to Mel-Spectogram

```
# Import the WhisperFeatureExtractor class from the transformers library
from transformers import WhisperFeatureExtractor

# Initialize a WhisperFeatureExtractor object by loading pre-trained weights from the "openai/whisper-small" model
feature_extractor = WhisperFeatureExtractor.from_pretrained("openai/whisper-small")
```

Tokenizer

- Similar to the feature extractor, the Whisper tokenizer takes the text input strings and converts them into label IDs that can be processed by the model. The tokenizer was pre-trained for Whisper, and we use the pre-trained tokenizer for our model as well.
  - In other words, it takes the target text and converts it into a format that the model can understand.
- The tokenizer takes the input words and makes a vector of values corresponding to each word

```
# Import the WhisperTokenizer class from the transformers library
from transformers import WhisperTokenizer

# Initialize a WhisperTokenizer object by loading pre-trained weights from the "openai/whisper-small" model
# Set the language to "english" and the task to "transcribe"
tokenizer = WhisperTokenizer.from_pretrained("openai/whisper-small", language="english", task="transcribe")
```

Processor

- The processor combines the feature extractor and the tokenizer into a single function that can handle preprocessing both the raw audio files and the corresponding transcriptions.

```
# Import the WhisperProcessor class from the transformers library
from transformers import WhisperProcessor

# Initialize a WhisperProcessor object by loading pre-trained weights from the "openai/whisper-small" model
# Set the language to "english" and the task to "transcribe"
processor = WhisperProcessor.from_pretrained("openai/whisper-small", language="english", task="transcribe")
```

Prepare Data for Model

```python
def prepare_dataset(batch):
    # Load and resample audio data from 48 to 16 kHz
    audio = batch["audio"]

    # Compute log-Mel input features from the input audio array using the feature extractor and extract log-Mel features from the audio array at the specified sampling rate
    batch["input_features"] = feature_extractor(audio["array"], sampling_rate=audio["sampling_rate"]).input_features[0]

    # Encode target text sentences to label IDs using the tokenizer and Convert the sentences to numerical token IDs suitable for model training
    batch["labels"] = tokenizer(batch["sentence"]).input_ids

    # Return the processed batch containing input features and label IDs
    return batch
```

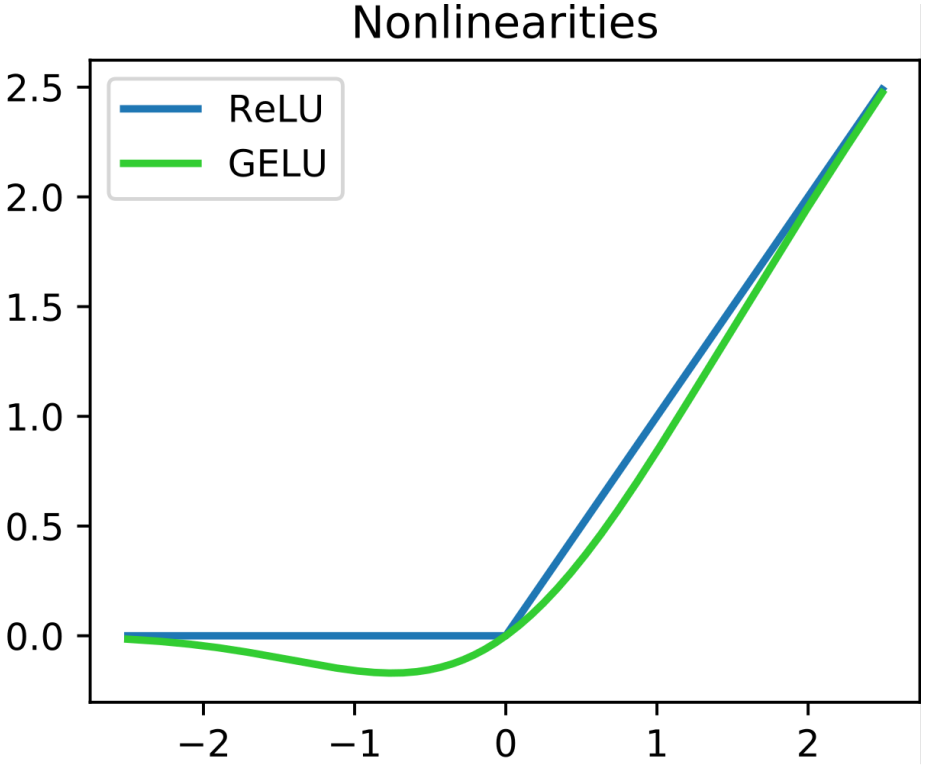
The Model - Convolutions and GELU

- The raw audio files, now in the form of Mel-spectograms, is passed through 2 CNN layers and activated using the GELU (Gaussian Error Linear Unit) activation.
  - The GELU activation has quickly become the standard for most transformer architectures due to its observed superior smoothness, performance, and normalization as compared with RELU, and other activation functions (Hendrycks et al.).
  - Φ(x) is known as the standard Gaussian cumulative distribution function

$$f(x) = x\Phi(x)$$

$$f(x) = \frac{x}{2}\left(1 + \tanh\left(\frac{\sqrt{2}}{\pi}\left(x + 0.044715x^3\right)\right)\right)$$

The Model - ReLU vs GELU



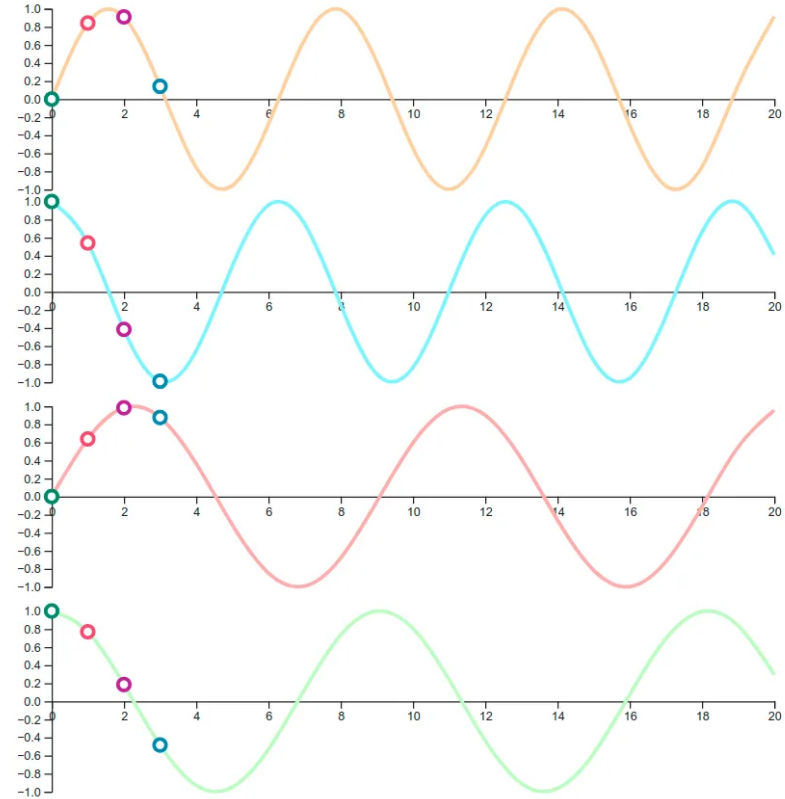
The Model - Positional Encoding

- The Whisper architecture makes use sinusoidal encoding in the encoder (different from learned positional embeddings for the decoder), which is used instead of the more popular learned positional embeddings given that the two types of embeddings were found to be nearly identical in performance, and sinusoidal encodings have less parameters.
- Like word embedding, positional encoding takes in the position of the token, is added to the inital embeddings, and outputs an encoded embedding that represents the relative position of the token within the sequence. This way, the individual tokens themselves know their relative position even when extracted from the sequence.

$$\mathrm{PE}(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

$$\mathrm{PE}(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

The Model - Positional Encoding



	p0	p1	p2	p3	
i=0	0.000	0.841	0.909	0.141	
i=1	1.000	0.540	-0.416	-0.990	
i=2	0.000	0.638	0.983	0.875	
i=3	1.000	0.770	0.186	-0.484	

Positional Encoding

$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

Settings: d = 50

The value of each positional encoding depends on the *position* (*pos*) and *dimension* (*d*). We calculate result for every *index* (*i*) to get the whole vector.

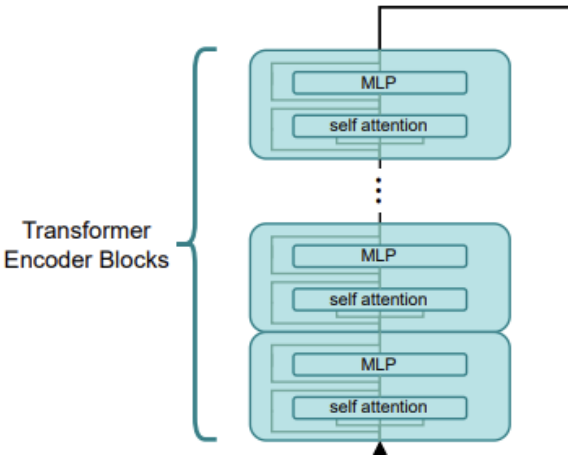
Pytorch for CNN and Positional Encoding

```
1
2  class Conv1d(nn.Conv1d):
3      # Customized 1D convolution class inheriting from nn.Conv1d
4      def _conv_forward(
5          self, x: Tensor, weight: Tensor, bias: Optional[Tensor]
6      ) -> Tensor:
7          # Overriding the forward function of the Conv1d class
8          return super()._conv_forward(
9              x, weight.to(x.dtype), None if bias is None else bias.to(x.dtype)
10         )
11
12 def sinusoids(length, channels, max_timescale=10000):
13     """Returns sinusoids for positional embedding"""
14     # Function to generate sinusoidal positional encodings
15     assert channels % 2 == 0
16     # Ensure the number of channels is even
17     log_timescale_increment = np.log(max_timescale) / (channels // 2 - 1)
18     # Calculate the increment of the logarithm of the timescale
19     inv_timescales = torch.exp(-log_timescale_increment * torch.arange(channels
           // 2))
20     # Calculate the inverse of the timescales
21     scaled_time = torch.arange(length)[:, np.newaxis] * inv_timescales[np.newaxis
           , :]
22     # Calculate scaled time
23     return torch.cat([torch.sin(scaled_time), torch.cos(scaled_time)], dim=1)
24     # Return concatenated sinusoids
```

The Model - Encoder Blocks

- Each encoder block in Whisper contains 2 sub-layers; multi-head self-attention, and a positionwise feed forward network.
  - Multi-head self-attention has various queries, keys, and values, unlike normal self-attention which limits tokens to only one Q, K, and V.
  - In Whisper, the feed forward network is just an MLP.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$



The Model - Multi-head Attention Code

```
1
2  class MultiHeadAttention(d2l.Module):  #@save
3      """Multi-head attention."""
4      def __init__(self, num_hiddens, num_heads, dropout, bias=False, **kwargs):
5          super().__init__()
6          self.num_heads = num_heads
7          self.attention = d2l.DotProductAttention(dropout)
8          self.W_q = nn.LazyLinear(num_hiddens, bias=bias)
9          self.W_k = nn.LazyLinear(num_hiddens, bias=bias)
10         self.W_v = nn.LazyLinear(num_hiddens, bias=bias)
11         self.W_o = nn.LazyLinear(num_hiddens, bias=bias)
12
13     def forward(self, queries, keys, values, valid_lens):
14         # Shape of queries, keys, or values:
15         # (batch_size, no. of queries or key-value pairs, num_hiddens)
16         # Shape of valid_lens: (batch_size,) or (batch_size, no. of queries)
17         # After transposing, shape of output queries, keys, or values:
18         # (batch_size * num_heads, no. of queries or key-value pairs,
19         # num_hiddens / num_heads)
20         queries = self.transpose_qkv(self.W_q(queries))
21         keys = self.transpose_qkv(self.W_k(keys))
22         values = self.transpose_qkv(self.W_v(values))
```

The Model - Multi-head Attention Code

```
        if valid_lens is not None:
            # On axis 0, copy the first item (scalar or vector) for num_heads
            # times, then copy the next item, and so on
            valid_lens = torch.repeat_interleave(
                valid_lens, repeats=self.num_heads, dim=0)

        # Shape of output: (batch_size * num_heads, no. of queries,
        # num_hiddens / num_heads)
        output = self.attention(queries, keys, values, valid_lens)
        # Shape of output_concat: (batch_size, no. of queries, num_hiddens)
        output_concat = self.transpose_output(output)
        return self.W_o(output_concat)
```

The Model - MLP Code

```
class PositionWiseFFN(nn.Module):  #@save
    """The positionwise feed-forward network."""
    def __init__(self, ffn_num_hiddens, ffn_num_outputs):
        super().__init__()
        self.dense1 = nn.LazyLinear(ffn_num_hiddens)
        self.relu = nn.ReLU()
        self.dense2 = nn.LazyLinear(ffn_num_outputs)

    def forward(self, X):
        return self.dense2(self.relu(self.dense1(X)))
```

The Model - Layer Normalization

```python
class AddNorm(nn.Module):  #@save
    """The residual connection followed by layer normalization."""
    def __init__(self, norm_shape, dropout):
        super().__init__()
        self.dropout = nn.Dropout(dropout)
        self.ln = nn.LayerNorm(norm_shape)

    def forward(self, X, Y):
        return self.ln(self.dropout(Y) + X)
```

The Model - Encoder Block

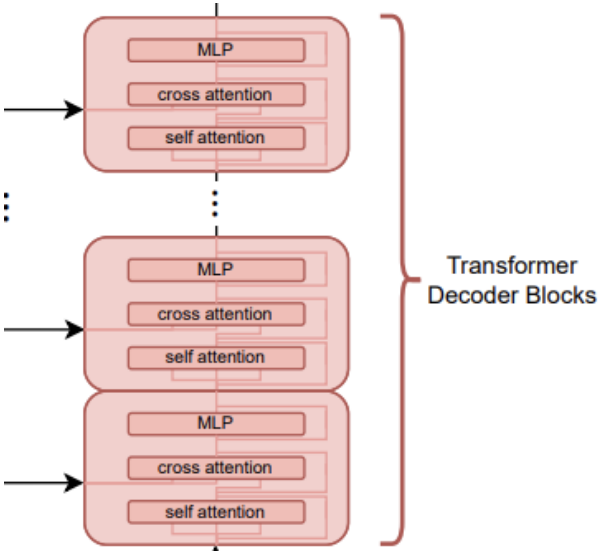
```
1
2  class TransformerEncoderBlock(nn.Module):  #@save
3      """The Transformer encoder block."""
4      def __init__(self, num_hiddens, ffn_num_hiddens, num_heads, dropout,
5                   use_bias=False):
6          super().__init__()
7          self.attention = d2l.MultiHeadAttention(num_hiddens, num_heads,
8                                                  dropout, use_bias)
9          self.addnorm1 = AddNorm(num_hiddens, dropout)
10         self.ffn = PositionWiseFFN(ffn_num_hiddens, num_hiddens)
11         self.addnorm2 = AddNorm(num_hiddens, dropout)
12
13     def forward(self, X, valid_lens):
14         Y = self.addnorm1(X, self.attention(X, X, X, valid_lens))
15         return self.addnorm2(Y, self.ffn(Y))
```

The Model - Learned Positional Embeddings

- The decoder makes use of learned positional embeddings, which are trainable and updateable as the model progresses, unlike sinusoidal positional embeddings.
- Despite taking up more computing time and space in an architecture, they often provide a more accurate embedding, as they are optimized to fit the specific task the model is being trained for.

The Model - Decoder Blocks

- The amount and structure of decoder blocks is identical to that of encoder blocks, with the addition of encoder-decoder attention.
  - Cross-attention enables the decoder to help keep track of the significant words in the input, and works identically to the other attention mechanisms used in the model.



The Model - Decoder Code

```
1
2  class TransformerDecoderBlock(nn.Module):
3      # The i-th block in the Transformer decoder
4      def __init__(self, num_hiddens, ffn_num_hiddens, num_heads, dropout, i):
5          super().__init__()
6          self.i = i
7          self.attention1 = d2l.MultiHeadAttention(num_hiddens, num_heads,
8                                                   dropout)
9          self.addnorm1 = AddNorm(num_hiddens, dropout)
10         self.attention2 = d2l.MultiHeadAttention(num_hiddens, num_heads,
11                                                  dropout)
12         self.addnorm2 = AddNorm(num_hiddens, dropout)
13         self.ffn = PositionWiseFFN(ffn_num_hiddens, num_hiddens)
14         self.addnorm3 = AddNorm(num_hiddens, dropout)
```

The Model - Decoder Code

```
    def forward(self, X, state):
        enc_outputs, enc_valid_lens = state[0], state[1]
        # During training, all the tokens of any output sequence are processed
        # at the same time, so state[2][self.i] is None as initialized. When
        # decoding any output sequence token by token during prediction,
        # state[2][self.i] contains representations of the decoded output at
        # the i-th block up to the current time step
        if state[2][self.i] is None:
            key_values = X
        else:
            key_values = torch.cat((state[2][self.i], X), dim=1)
        state[2][self.i] = key_values
        if self.training:
            batch_size, num_steps, _ = X.shape
            # Shape of dec_valid_lens: (batch_size, num_steps), where every
            # row is [1, 2, ..., num_steps]
            dec_valid_lens = torch.arange(
                1, num_steps + 1, device=X.device).repeat(batch_size, 1)
        else:
            dec_valid_lens = None
        # Self-attention
        X2 = self.attention1(X, key_values, key_values, dec_valid_lens)
        Y = self.addnorm1(X, X2)
        # Encoder-decoder attention. Shape of enc_outputs:
        # (batch_size, num_steps, num_hiddens)
        Y2 = self.attention2(Y, enc_outputs, enc_outputs, enc_valid_lens)
        Z = self.addnorm2(Y, Y2)
        return self.addnorm3(Z, self.ffn(Z)), state
```

The Model - Putting it Together

```
1
2  class TransformerDecoder(d2l.AttentionDecoder):
3      def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens, num_heads,
4                   num_blks, dropout):
5          super().__init__()
6          self.num_hiddens = num_hiddens
7          self.num_blks = num_blks
8          self.embedding = nn.Embedding(vocab_size, num_hiddens)
9          self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
10         self.blks = nn.Sequential()
11         for i in range(num_blks):
12             self.blks.add_module("block"+str(i), TransformerDecoderBlock(
13                 num_hiddens, ffn_num_hiddens, num_heads, dropout, i))
14         self.dense = nn.LazyLinear(vocab_size)
```

The Model - Putting it Together

```
    def init_state(self, enc_outputs, enc_valid_lens):
        return [enc_outputs, enc_valid_lens, [None] * self.num_blks]

    def forward(self, X, state):
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self._attention_weights = [[None] * len(self.blks) for _ in range (2)]
        for i, blk in enumerate(self.blks):
            X, state = blk(X, state)
            # Decoder self-attention weights
            self._attention_weights[0][
                i] = blk.attention1.attention.attention_weights
            # Encoder-decoder attention weights
            self._attention_weights[1][
                i] = blk.attention2.attention.attention_weights
        return self.dense(X), state

    @property
    def attention_weights(self):
        return self._attention_weights
```

Load the Pre-Trained Whisper Model

- This is where we fetch the pre-trained Whisper model parameters from Hugging Face for fine-tuning.

```
1
2 from transformers import WhisperForConditionalGeneration
3
4 model = WhisperForConditionalGeneration.from_pretrained("openai/whisper-small")
```

Define a Data Collator

```python
import torch
from dataclasses import dataclass
from typing import Any, Dict, List, Union

@dataclass
class DataCollatorSpeechSeq2SeqWithPadding:
    processor: Any
    decoder_start_token_id: int

    def __call__(self, features: List[Dict[str, Union[List[int], torch.Tensor]]])
          -> Dict[str, torch.Tensor]:

        # split inputs and labels since they have to be of different lengths and
            need different padding methods
        # first treat the audio inputs by simply returning torch tensors

        input_features = [{"input_features": feature["input_features"]} for
            feature in features]
        batch = self.processor.feature_extractor.pad(input_features,
            return_tensors="pt")
```

Define a Data Collator Cont.

```
        # get the tokenized label sequences
        label_features = [{"input_ids": feature["labels"]} for feature in
             features]

        # pad the labels to max length
        labels_batch = self.processor.tokenizer.pad(label_features,
             return_tensors="pt")

          # replace padding with -100 to ignore loss correctly

        labels = labels_batch["input_ids"].masked_fill(labels_batch.
             attention_mask.ne(1), -100)

        # if bos token is appended in previous tokenization step,
        # cut bos token here as it's append later anyways
        if (labels[:, 0] == self.decoder_start_token_id).all().cpu().item():
            labels = labels[:, 1:]

        batch["labels"] = labels

        return batch
```

Initialize Collator and Define Evaluation Metric

```python
import evaluate


data_collator = DataCollatorSpeechSeq2SeqWithPadding(
    processor=processor,
    decoder_start_token_id=model.config.decoder_start_token_id,
)

metric = evaluate.load("wer")

def compute_metrics(pred):
    pred_ids = pred.predictions
    label_ids = pred.label_ids

    # replace -100 with the pad_token_id
    label_ids[label_ids == -100] = tokenizer.pad_token_id

    # we do not want to group tokens when computing the metrics
    pred_str = tokenizer.batch_decode(pred_ids, skip_special_tokens=True)
    label_str = tokenizer.batch_decode(label_ids, skip_special_tokens=True)

    wer = 100 * metric.compute(predictions=pred_str, references=label_str)

    return {"wer": wer}
```

Define Training Configuration

```python
from transformers import Seq2SeqTrainingArguments

training_args = Seq2SeqTrainingArguments(
    output_dir="./whisper-small-aajw",  # change to a repo name of your choice
    per_device_train_batch_size=16,
    gradient_accumulation_steps=1,  # increase by 2x for every 2x decrease in batch size
    learning_rate=1e-5,
    warmup_steps=500,
    max_steps=4000,
    gradient_checkpointing=True,
    fp16=True,
    evaluation_strategy="steps",
    per_device_eval_batch_size=8,
    predict_with_generate=True,
    generation_max_length=225,
    save_steps=1000,
    eval_steps=1000,
    logging_steps=25,
    report_to=["tensorboard"],
    load_best_model_at_end=True,
    metric_for_best_model="wer",
    greater_is_better=False,
    push_to_hub=False,
)
```

Define Training Parameters and Train

```
1
2  from transformers import Seq2SeqTrainer
3
4  trainer = Seq2SeqTrainer(
5      args=training_args,
6      model=model,
7      train_dataset=common_voice["train"],
8      eval_dataset=common_voice["test"],
9      data_collator=data_collator,
10     compute_metrics=compute_metrics,
11     tokenizer=processor.feature_extractor,
12 )
13
14 processor.save_pretrained(training_args.output_dir)
15
16 trainer.train()
```

- Total Training time was 2:24.57.
- The Nvidia A100 GPU was used for training.

Results From Testing - Accented Speech

- Both the fine-tuned and unchanged model were tested with the same, accented speech dataset, comprising 2 hours of speech with over 173 accents.
- Using the WER as the evaluation metric, our model performed quite well with respect to the unchanged model. We had a WER of 21.77% at the final step checkpoint, and was as small as 20.74%. The unchanged model had a WER of 25.08%.
- For reference, when Whisper was trained, moving from 54486 hours of supervised speech to 681070 hours of supervised speech decreased WER from 10.9% to 9.9%.

Step	Training Loss	Validation Loss	Wer
1000	0.165700	0.467473	20.735708
2000	0.018500	0.534014	21.579194
3000	0.006400	0.574676	21.274602
4000	0.002700	0.595369	21.766635

Figure 1: Whisper small fine-tuned on accented speech.

```
Average Word Error Rate (WER): 0.25083456208733385
```

Figure 2: Whisper small not fine-tuned.

Project Limitations

- The testing data set was not very large, and consisted of a variety of accents. This could implicate our model needing only to recognize a few accents much better than the original model to exhibit enhanced performance.
  - It is of note that the original model struggled the most with transcribing Chinese and Hindu accents.
- It is also important to note that these findings likely cannot be scaled to the larger Whisper models, as larger Whisper models will have inherently smaller WER given their larger architecture.
  - Additionally, changes in WER from fine-tuning with small speech data sets will not be near as drastic for the same reasons.

Conclusions and Future Considerations

- Overall, transformer architecture has radically reduced the WER among ASR models that make use of the its architecture and provides the basis for additional modifications for enhancing a myriad of other models.
- It should not be discounted that training on supervised, accented data may help pose a solution for continuing to enhance model accuracy.
- Future iterations of the project would scale our design to the large-whisper model, and use more data than what was used for our project.
    - Additionally, testing the larger models on American accented speech to see if training a variety of accents would help the model decipher American speech better would be an interesting study as well.