Assignment 3 Part 2 - Speech Transcription

Introduction to Deep Learning
Emeritus & Carnegie Mellon University

1 Overview

In this assignment, we'll be developing a model to transcribe audio directly to text.

In the first assignment, you did a similar task that involved identifying phonemes: units of speech. By contrast, your model in this assignment should directly produce readable text, complete with punctuation.

Objective: Given audio of human speech in the form of a spectrogram, produce a text transcription of what was said.

Warning: This assignment is much more difficult than previous ones. Not only are the pipeline and model complicated, but the training process can be unstable. The code will be surprisingly straightforward, but the concepts are difficult.

But completing this should be rewarding. You'll have a line-by-line understanding of a recent state-of-the-art model in speech transcription, and you'll also learn key concepts that appear in transformers and many other recent architectures.

2 Pipeline Overview

The model and pipeline we'll use comes from the Listen, Attend, Spell (Chan, Jaitly, Le, Vinyals 2015) paper.

It's an older architecture, but we selected it because it accurately depicts how difficult state-of-the-art model design is, and also introduces covers many key concepts. Concepts include **attention, seq2seq problems, encoder-decoder architectures, and teacher forcing**.

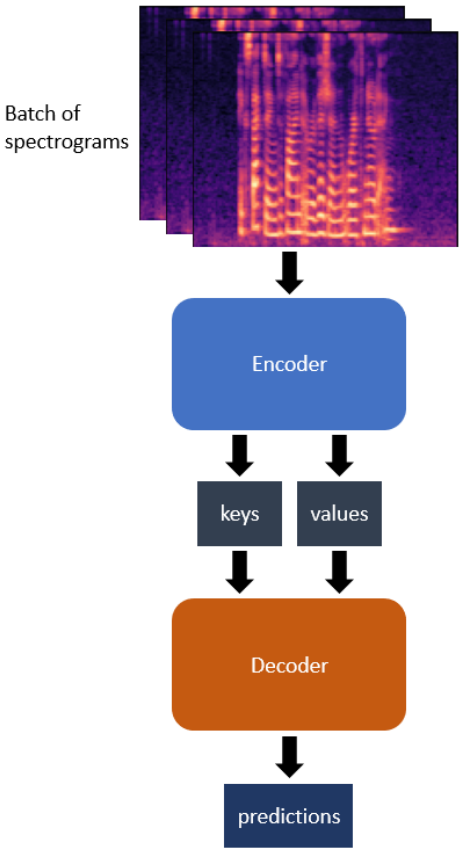


Figure 1: Broad overview of LAS

Above is high-level view of LAS's architecture.

The main idea is pretty simple; we input a batch of spectrograms into the **encoder**. The goal of the encoder is to generate a condensed 'summary' of the input that contains useful information for the decoder. The encoder returns two tensors (**keys** and **values**) that are given to the **decoder**.

The decoder's goal is to generate predictions of what's being said, token by token. Its output will be a **predictions** tensor shaped `(batch_size, vocab_size, max_len)`. Basically, for each audio clip in the batch, we produce a sequence of logits. Each logit in the sequence indicates the probability that some character is at that position of the sequence.

The big picture is genuinely this simple. Of course, the devil is in the details, which we'll now cover.

2.1 Batching differently-sized elements

Before we get to the encoder, an important question: how do you batch together multiple observations and/or labels that may differ in length?

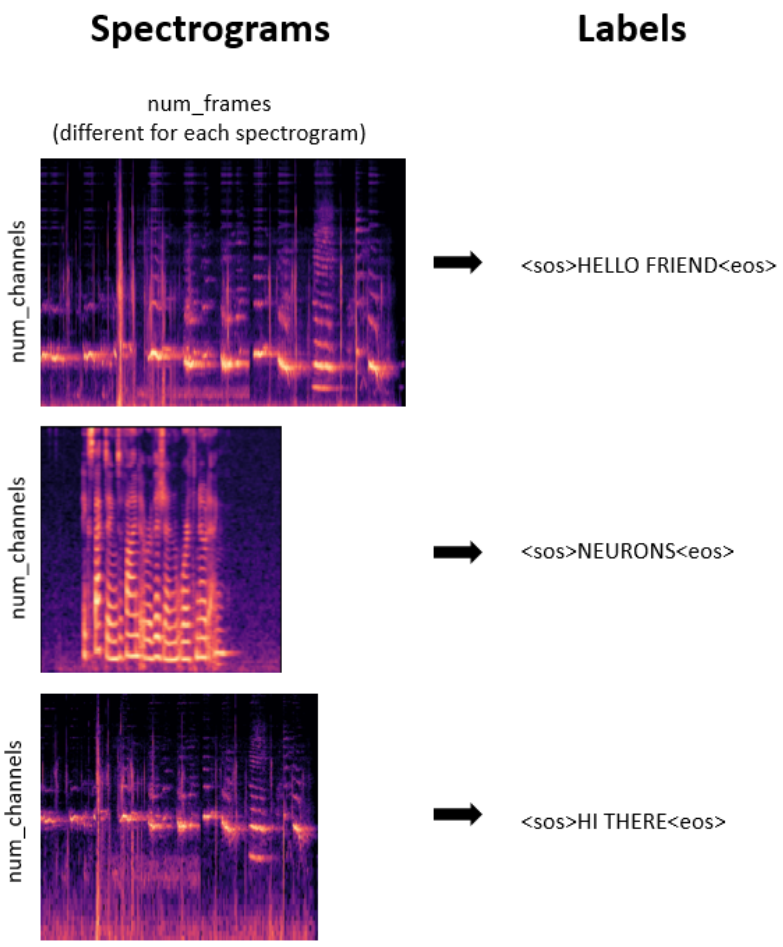


Figure 2: Multiple spectrograms and their labels.

Every spectrogram will have the same number of channels, but may differ in length. The spectrograms are each a tensor of shape `(num_channels, num_frames)`, where `num_frames` varies.

The labels also vary in length. The labels begin and end with special tokens, marking the start and end of the sequence. Each label is converted to a 1D tensor, where each token is mapped to an integer index. So the first label in Figure 2 would be something like `[37, 8, 5, 12, 12, 15, 36, 6, 18, 9, 5, 14, 4, 38]`.

In the first two assignments of this course, we never had to worry about batches containing differently-sized elements; everything was the same size. All we had to do was stack each element along a new dimension, and we had a finished batch.

But these varying lengths make things harder.

The simple solution is **padding**.

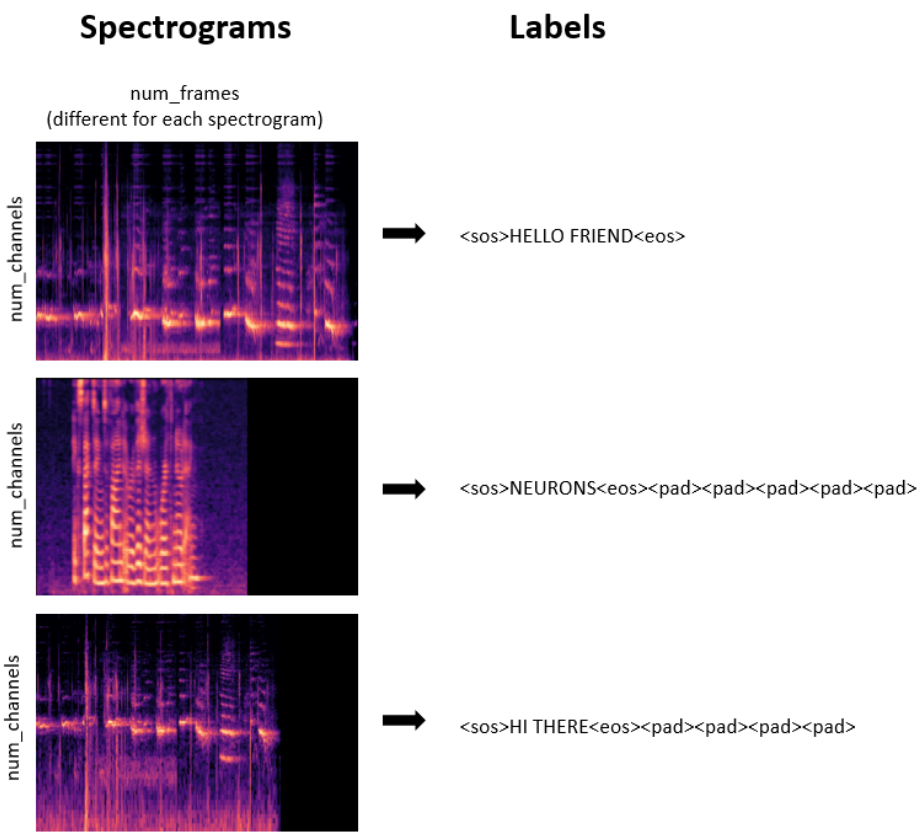


Figure 3: Spectrograms and labels after padding. Note that `<sos>`, `<eos>`, and `<pad>` are single tokens.

While initializing your dataloader, you'll give it a `collate_fn` ("collate function") that we wrote for you.

- The `collate_fn` tells the dataloader how to convert a list of observations into a single batch.
- In previous homeworks, we never had to write a custom `collate_fn`, and so it used a default one that assumes everything is the same size, and just concatenates along a new axis.
- Our method pads the audio and labels to the same length, then concatenates and returns each of them.
- It also creates and returns `data_lens` and `label_lens` tensors, which will be needed throughout the pipeline.

Hopefully not too bad so far!

3 Encoder ("Listener")

Now onto the actual model.

The goal of the encoder in this paper is to extract useful information from the audio input. For this reason, this paper sometimes calls the encoder the "Listener"[1].

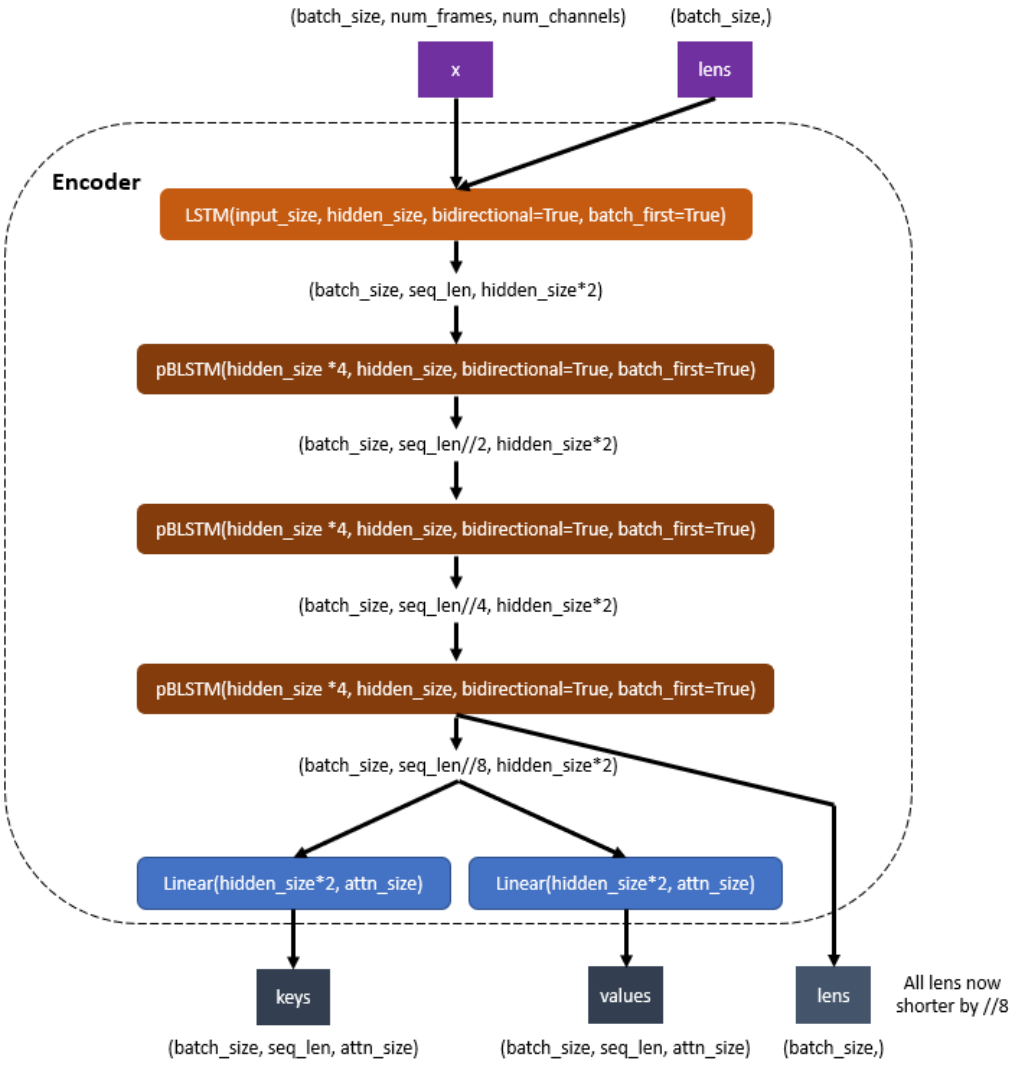


Figure 4: Encoder architecture

First, take a look at the initial plain LSTM layer of the encoder. Some notes:

1. The `batch_first=True` parameter tells the layer to input/output tensors with the `batch_size` dimension first[2].

2. The output has `hidden_size*2` because `bidirectional=True`. Read the docs for `nn.LSTM` for why.

[1] "Listener" is not a general nickname for encoders; it's specific to this paper

[2] For complicated CUDA-related reasons, RNN-based modules have the `seq_len` dimension first, but nowadays most components expect `batch_size` first, so we do this for consistency with previous assignments.

3.1 pBLSTMs

The encoder uses custom layers called **pBLSTM**s ("Pyramidal Bidirectional LSTMs").

The main goal of each `pBLSTM` is to **downsample the length of a given input by half**. The encoder has 3 `pBLSTM` layers total, so after passing through all three layers, the data will have `seq_len//8` (because $\frac{1}{2}^3 = \frac{1}{8}$).

But note that they do more than just compression; they'll also be manipulating the values of the data.

3.1.1 Downsampling?

Question: Why downsample the input?

Answer: we downsample to make the information denser and more accessible to the decoder.

Reasons:

1. Not every frame and not all parts of every frame will be useful
    - E.g. silent frames or silent frequencies in frames
2. The number of frames is likely much larger than the length of the final text output
    - Without downsampling, the decoder would need to look at a huge number of frames to generate just a single character

To illustrate, let's say we had a spectrogram of 200 frames of someone saying "HELLO". Roughly speaking, this means there are around `200/5=40` frames we need to look at in order to generate a single character. But if we downsample to `200/8=25` frames, we only look at around `25/5=5` frames per character.

3.1.2 Implementation

Implementing the `pBLSTM` is surprisingly simple. It's just a wrapper around a bidirectional LSTM that handles downsampling.

Here's the pseudocode:

1. If there are currently an odd number of frames, throw away the last frame to end up with an even number[3]. This is needed for the next step.
2. `reshape()` the input from `(batch_size, seq_len, hidden_size)` to `(batch_size, seq_len//2, hidden_size*2)`
3. Give the reshaped input to the BiLSTM, return the output

That's it. Not too bad.

[3]We can safely throw away this frame because it's unlikely to carry critical information, especially because it's at the end.

3.1.3 Optional Note: Reshaping?

You can interpret the reshaping as moving some frames onto the `hidden_size` dimension.

```
In [11]: a = np.arange(20,dtype=float).reshape(4,5)

In [12]: a
Out[12]:
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.],
       [10., 11., 12., 13., 14.],
       [15., 16., 17., 18., 19.]])

In [13]: a.reshape(2,10)
Out[13]:
array([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
       [10., 11., 12., 13., 14., 15., 16., 17., 18., 19.]])
```

Figure 5: Downsampling via reshaping example

Here's a simple visual example of this, assuming `(batch_size=1, seq_len=4, hidden_size=5)`. After reshaping, every second "frame" got moved onto the hidden dim of its predecessor.

The Bi-LSTM in the `pBLSTM` then "mixes up" the information. And because the output size is smaller than the input size, it's forced to "compress" it.

3.1.4 Optional Note: Layer sizes?

Hopefully you noticed a few odd details about the `pBLSTM` layer sizes. Here is their derivation.

- Why the input size of `hidden_size*4`?
  - Recall from the encoder diagram the initial plain `BLSTM` layer that outputs a tensor with `hidden_size*2`.
  - Our first `pBLSTM` layer then performs the reshaping, which again multiplies that dimension by 2, resulting in `hidden_size*4`, the exact size the layer is expecting.
- Why the output size of `hidden_size`?
  - Because the `pBLSTM` is bidirectional, specifying an output size of `hidden_size` will actually yield `hidden_size*2` features, just like we said on page 5.
  - This means that, each `pBLSTM` layer outputs the same number of hidden features as it received. However, the sequence length got halved.
- As a result, all layers can have the same sizes, yet still manage to compress the input sequence.

It's definitely possible to do downsampling in other ways too!

3.2 Key Network and Value Network

The final part of the encoder is two **separate** linear layers: the `key_network` and `value_network`. You'll need to give the output of the last `pBLSTM` to each of these layers in order to get the `keys` and `values`.

Notice that the `keys` and `values` are essentially two different projections of the input. We make these to use for attention, which we'll explain on the next page.

4 Attention

Attention is actually calculated during the decoder, but we'll introduce it now, so you don't have to juggle learning both the decoder and attention later on.

4.1 Introduction

Attention has become a very important concept in deep learning in the last few years. Most notably, transformers use it extensively in the form of self-attention and multi-head attention (which we won't cover, but we encourage you to look up after completing this assignment).

Attention in deep learning is meant to loosely mimic how attention works in organic brains. When humans 'pay attention' to something, they limit the amount of information they process and focus in on certain stimuli. This is believed to help cognition by allowing them to better allocate their finite computational resources and extract signal from noise.

A computational model of this is very useful in LAS.

The decoder works by predicting one token of the output at a time, using the encoder's summary of the input. But some parts of the input are more relevant to certain tokens than others.

For instance, if our decoder is near the end of a transcription, information near the very beginning of the input is probably not as important anymore. So we should ideally deprioritize it and focus more on the end of the input. This is what attention will help us do.

4.2 Intuition: Queries, Keys, and Values

Attention works by using three tensors called `query`, `keys`, and `values`. They're so named because attention kinda works like querying a database.

For example, say we're searching YouTube for a specific video:

- You search for a video by typing out a string, which is your **query** to the YouTube database.
- The query gets compared to a set of **keys**, which are things like video titles, tags, description text, etc.
- If any keys match well with our query, the database returns the **values** (videos) that correspond to those keys.

The main idea is that this process allows us to filter out parts of the `values` that weren't relevant to our `query`.

Here's that same metaphor above, but in LAS terms:

- The decoder will produce a `query` every time it wants to predict a token.
  - This `query` is attempting to find parts of the input (from `values`, which is the encoder's summary of the input) that are relevant to its next prediction.
- The `query` gets compared against the `keys` that the encoder had generated.
- We calculate `attention`, which represents how well each `key` matched our `query`. We then compare `attention` with `values`, which creates a tensor `context`.
  - `context` is like the list of relevant videos YouTube would give you. It's the subset of information from `values` that we're paying attention to.

We encourage you to review this description after seeing the next page, if needed.

4.3 Implementing Dot Product Attention

Now that we've covered the 'why' of attention, let's discuss the 'how'.

For LAS, we'll be implementing one of the simplest forms of attention in deep learning, called **dot product attention**.

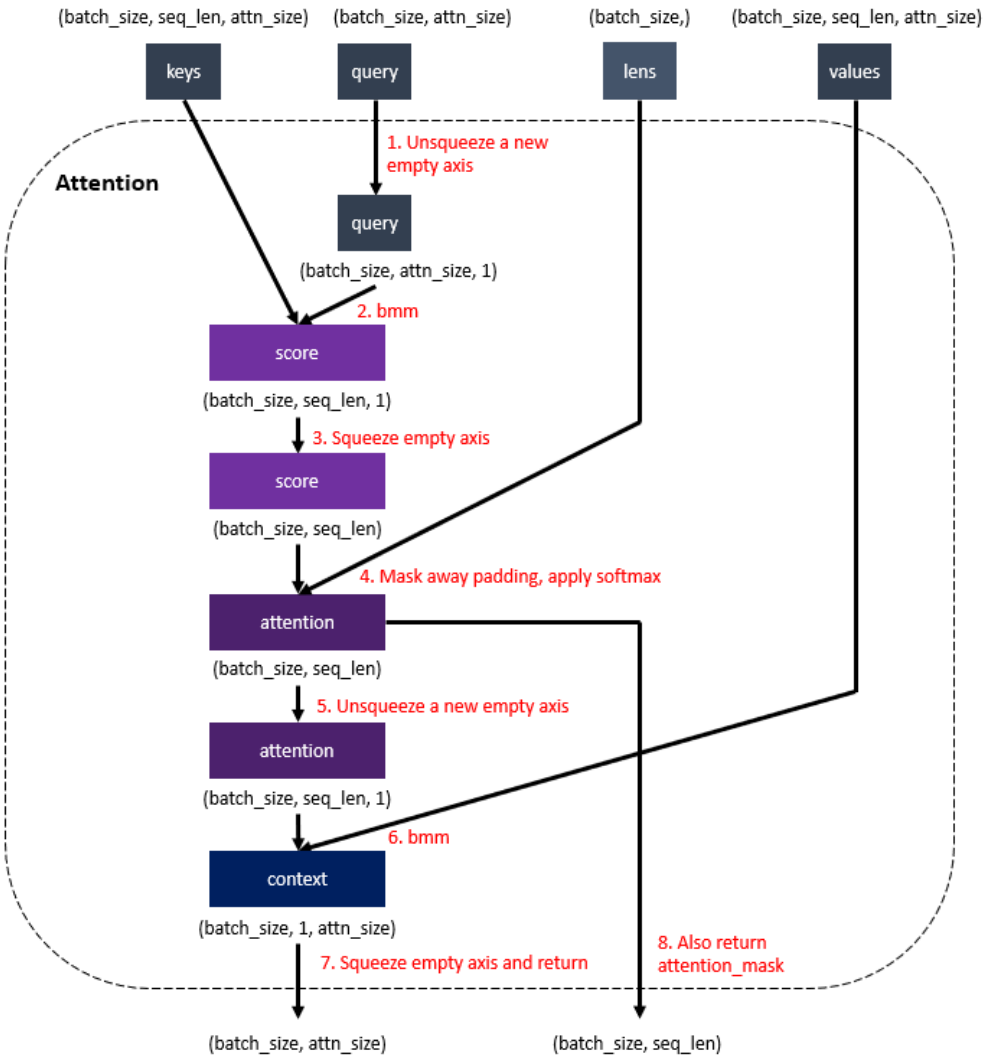


Figure 6: Attention pipeline

In colloquial terms, we first compare the `keys` to the `query` to get `scores` of how well they match. We mask away any padding tokens, then scale the `scores` between 0 and 1 with a softmax, giving us our `attention`. Finally, we compare the `attention` with the `values` to get our final `context`. We also return the `attention` mask we generated, which we can use to monitor the network's training progress.

The diagram may look a little complex, but it's actually all you'll need to implement the code. We marked the steps of the pseudocode in red. Look up the `torch` methods for `unsqueeze`, `squeeze`, and `bmm`.

The code for step 4 is difficult to explain and not very insightful, so we'll just give it to you in the notebook. The goal of this step is to prevent attending to padding. We set the pre-softmax values of the padding to $-\infty$, which makes the post-softmax values close to 0.

5 Decoder ("Speller")

The decoder is unfortunately much more complicated than the encoder - the encoder is a linear pipeline, but the decoder is **recurrent**.

Also, note that the **decoder works differently depending on if you're in training or eval mode, and whether you have teacher forcing or not**. For now, we'll introduce the decoder in training mode and discuss how eval and teacher forcing modify it.

5.1 Decoder Overview

The decoder's main job is to predict the appropriate output sequence using the encoder's summary of the input.

It does this by **predicting one token at a time, in order**. In other words, at `t=0`, given the start token, the decoder calculates `context` and predict the first token of the label. At `t=1`, the decoder uses the token it predicted previously to calculate `context` and predict the next token, and so on.

Notice how this process resembles how humans write; we write one character at a time, while keeping in mind what we've written previously.

But at what `t` should it stop predicting new tokens? It stops when it predicts enough tokens to match the length of the longest label in the batch. In other words, it stops when `t=max_len`, where `max_len = seq_len - 1`[4].

Summary:

- The main idea of the decoder is a `for` loop, that runs from `t=[0, ..., max_len-1]`.
- At each `t`, the decoder will try to predict the next token using information about the previous token it generated and attended input.

5.2 How the Decoder Makes Predictions

Now for a high-level conceptual overview of what happens at each timestep of the `for` loop:

1. Create an embedding of the previous token we generated
2. Pass this embedding through two `LSTMCell`s
3. Figure out which parts of the input to pay attention to (`context`), by using the output of the `LSTMCell`s as your `query` to the attention layer
4. Predict the next character by concatenating the attended input and the output of the cells, and passing that through a linear layer to get logits.
5. Save your prediction logit by appending it to a list.

In short, at each timestep, we predict the next token given our previous predictions and the parts of the input that we're currently paying attention to.

Now for the implementation details.

[4]The `-1` is because the start token is already given

5.3 Implementing the Decoder

5.3.1 Initialization

We first initialize the following layers:

- `nn.Embedding`
  - This layer is basically a learned lookup table, mapping some index to a meaningful embedding vector. Essentially, it makes indices richer and more interpretable for deep learning.
- Two `nn.LSTMCell`s
  - Just like we did in part A of this assignment, we're going to be passing inputs and hidden states between cells and between time steps.
- `Attention` layer to attend to input
- `nn.Linear` to predict the next token

5.3.2 Preparing for the `for` Loop

The decoder's forward pass receives the `keys`, `values`, and `lens` tensors from the encoder.

It then does the following to prepare for the `for` loop:

1. Set `max_len = seq_len - 1`
2. Initialize `prediction` tensor, shaped `(batch_size, vocab_size)`.
   - During the `for` loop, this will be our logits that represent our prediction of the next char at each iter.
   - However, since we haven't predicted anything yet, we set it as if it is predicting `<sos>`, since we always know the `<sos>` token comes first.
3. Initialize `context` tensor, shaped `(batch_size, attn_size)`, by taking it from the first timestep of the `values` tensor.
   - During the `for` loop, this tensor will abstractly store information about previous characters we've predicted.
4. Initialize some empty lists:
   - `predictions`, where we'll append the logit for each token we predict. At the end of the `for` loop, it should contain `max_len` tensors. Later, we'll concatenate this list into a single tensor and give that to our loss function.
   - `hidden_states`, the hidden states of the two `LSTMCells` that create the attention query for each timestep
   - `attentions`, to store the attention mask produced at each time step. Saving these is optional but recommended; we can visualize them later to make sure our network is learning properly.

It's a lot to take in, so don't worry if it's not 100% clear yet. The next section should help explain why we declared all of those variables.

5.3.3 The `for` Loop

This is the prediction algorithm we described in 5.2, but with technical details.

1. Create embedding of previous token `x`, that's the input tensor for the `LSTMCell`s
   (a) Identify the indices of the tokens we predicted in the previous timestep by taking the `prediction` vector and applying `argmax` along its `vocab_size` dim. This creates a batch of indices shaped `(batch_size,)`. Pass these to your embedding layer, to get a `char_embed` tensor
   (b) Concatenate `char_embed (batch_size, hidden_size)` and `context (batch_size, attn_size)` along the appropriate dimension to get `x (batch_size, hidden_size+attn_size)`
2. Pass `x` through the `LSTMCell`s
   (a) Give the first cell `x` and `hidden_states[0]`. Store the output in hidden_states[0]. Set `x` to be `hidden_states[0][0]`
   (b) Give the second cell `x` and `hidden_states[1]`. Store the output in hidden_states[1]. Set `x` to be `hidden_states[1][0]`
3. Give appropriate arguments to attention, using `x` as the `query`. It will return `context` and `attention`
4. Concatenate `x` and `context` along the `hidden_size` dimension. Give this to your linear layer to get `predictions`
5. Store your results by appending `prediction` to `predictions` and appending `attention[0, :]` to `attentions`
   - The reason for the `[0, :]` indexing on `attention` is to grab only the first item in the batch. We only need one per timestep for visualization purposes

That's it for the `for` loop.

Make sure to look up the official documentation for each method you use. Otherwise, try not to overthink each individual step; we streamlined this process to make it less ambiguous.

5.3.4 Returning Results

To wrap things up, we need to combine our `predictions` and `attentions` and `return` them.

`torch.stack` your `predictions` along a new `dim=2` axis to get `(batch_size, vocab_size, max_len)`. Also, `torch.stack` the `attentions` along a new axis `dim=0` to get `(max_len, seq_len)`. Return both.

You're almost done with the decoder, but there's two last implementation details we need to cover.

5.4 Teacher Forcing

We mentioned earlier that **teacher forcing** affects the implementation of the decoder.

Teacher forcing is a technique used during training to dramatically improve performance. It's basically mandatory for this assignment; if you don't implement it, your decoder may never even converge.

The main idea is **to embed the label token instead of the token it predicted previously some percentage of the time**. In other words, if `tf_prob=0.9`, with 90% probability we will provide the label instead of our previous prediction at each timestep.

The reason this helps is because the decoder will be awful when you first begin training. Because each prediction relies on the quality of previous predictions, having very poor quality predictions means it'll take ages to improve, if it ever does.

5.4.1 Implementing Teacher Forcing

Fortunately, the implementation is simple, with minimal changes.

- (Given) While preparing for the `for` loop, generate embeddings for the labels in advance to save on computation time
  - Pass `labels` through the `embedding_layer` to get `label_embeddings` shaped `(batch_size, max_len+1, hidden_size)`
- In the `for` loop pseudocode, modify step 1a by checking if some random float between 0 and 1 is lower than the `tf_prob`
  - If so, set `char_embed` as `label_embeddings[:,t,:]`.
  - Else, embed your previous prediction, just like normal
- When training, make sure you give the model a non-zero `tf_prob`. Conversely, during eval, make sure `tf_prob` is its default value, zero.

5.4.2 Rate and Scheduling of Teacher Forcing

Now for a critical question: **what `tf_prob` should you use?**

The answer is to **around 90 to 95%**. You can optionally begin to lower this number by 0.5% or 1% per epoch after attention begins to converge.

The reason this high rate helps is because it lets your decoder initially focus on learning how to spell. And gradually decreasing it after it begins to converge coaxes the model into relying on its own previous predictions. After it becomes a decent speller, it'll start taking into account the encoder information in order to further lower the loss.

5.5 Eval Mode

Eval also affects the implementation of the decoder.

It's fortunately very simple:

- (Given) `max_len` should be some reasonable length given typical labels in the dataset (we set it as 600)
- `labels` should be None (default value)
- `tf_prob` must be 0 (default value)

No changes to your model code are necessary, as we already gave you the first item and the other two only matter in your training and eval loops. Just be aware of these differences.

6 Training and Evaluating LAS

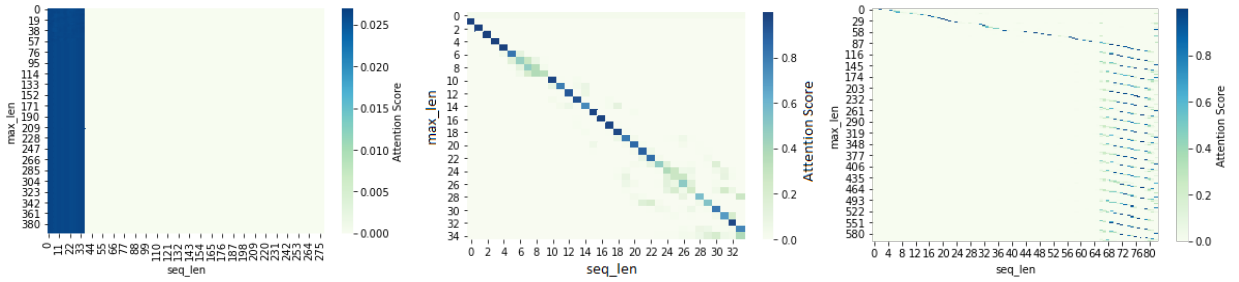
Some final important notes.

6.1 Training

- Follow the teacher forcing scheduling in 5.4.2. It will be critical to passing the assignment.
- Mixed precision from the previous assignment probably won't cause much speedup; the time bottleneck is in the sequential nature of the LSTMs and decoder.
- We recommend you use gradient clipping to prevent exploding gradients from harming training.
- If you choose to use LR scheduling, don't activate it until attention starts to work. This may take anywhere between 10 to 30 epochs.
- Each epoch will probably take between 10 to 25 minutes.
  - If your code is much slower, make sure that you're using a GPU and that your dataloader settings are adequate (particularly `num_workers`). If those two are correct, try to identify and address the speed bottleneck in your code.
- Training will be unstable.
  - It's common to observe slow decreases in loss and poor eval performance for 10-20 epochs, then a sudden improvement.
  - Unfortunately, this can make it hard to tell if your network is bugfree and actually working.
  - One way to tell your network is working is by monitoring **attention plots**, which we'll introduce on the next page

6.1.1 Attention Plots

Visualizing attention of a single utterance once per epoch is a good way to gauge training progress.



(a) Untrained model's attention plot

(b) Trained model's attention plot (short example for clarity, no padding)

(c) Trained model's attention plot (realistic example, padding)

Figure 7: Examples of attention plots

- The leftmost image is a bad attention plot of an untrained model. The center and right image are good plots of trained models.
- Interpretation: "during each timestep, which part of the summarized input did the decoder pay attention to?"
  - `seq_len` is the length of the padded spectrogram after being shortened by the encoder by a factor of 8.
  - `max_len` is the number of timesteps in the decoder.
- A decent model's attention plot should have a distinct diagonal line that starts at the top left corner.
  - As seen in the last image, padding causes weird behavior, but this is perfectly fine because it doesn't matter anyway.
  - It also doesn't have to go to the bottom right corner, nor be be perfectly diagonal, nor be perfectly clear. This is also fine.
- Notice the values range between 0 and 1. This is due to the softmax used to calculate attention. The higher the value, the more attention was paid.

Hopefully you can see why having a diagonal line is good. The beginning of your audio is likely relevant to the first few characters you generate, same for the middle, same for the end. Assuming a model is using the audio input properly, the attention will form this pattern.

For the first 5 to 20 epochs, the plots will probably be pretty poor, despite train loss decreasing. This is normal; your model is likely learning how to spell first, without paying much attention to the input. After it gets decent and `tf_prob` starts to gradually decrease, it'll start learning to interpret the input, which is when the performance will suddenly improve.

We wrote a method that creates and saves an attention plot for you in `utils.py` called `plot_attentions`. We recommend you save an attention plot once per epoch, to get a gauge on how it's doing.

6.2 Evaluation

Last section.

- We recommend you use a metric called Levenshtein distance to gauge your model's performance.
  - In short, it measures how different your predicted string is from the true label. The higher the distance, the worse your prediction is.
  - Make sure to run the cell in the beginning of the notebook that installs the `python-Levenshtein` library.

You're done with the writeup! See you in the notebook!