COGS 402 Summary

Intro:

The goal for this project is to experiment with the use of IPA in machine learning models instead of training with the English alphabet as the target. Using IPA as a target can enable more a more robust model as it removes a layer of inference and projection the model has to make. To put it simply, mouth movements produce sounds and sounds correspond to words in English, we are attempting to simplify the model so it does not have to map sounds to English characters which can cause a layer of confusion, as there is sometimes a one to many mapping of sounds to English characters (the phoneme /sh/ can be represented by "sh", "ch", "ti", "ci" in the English language)

Furthermore, using IPA can yield in a more generalized model that can be use across different language since we are focusing on the sounds being made not the words spoken (although training the model on the English language results in ignoring some phonemes that are common in other languages)

One of the more apparent ways to experiment with IPA representations in machine learning is to simply build a lip reading model, as there is a stronger correlation between mouth movements and phonemes rather than mouth movements and English Characters.

Below I will highlight the important parts of the code and explain the reasoning behind them

Transforming to IPA

CMUDICT

The following code is based of the code provided below:

http://www.speech.cs.cmu.edu/cgi-bin/cmudict

https://github.com/arianshamei/phonemic-parser/blob/main/phonemic_parser.py

https://github.com/nicknochnack/LipNet

I used CMUDICT to convert English words to their equivalent phonetic translation.

“CMUDICT pronunciation dictionary for North American English
that contains over
134,000 words and their pronunciation” as noted here

Since CMUDICT provides three versions of the vowel phonemes based on their stress level (0,1,2) I decided to remove stress levels for vowel phonemes in order to gauge a better accuracy metric since the stress level of the vowel is not of much importance. (AH0, AH1, AH2 all become AH)

`simplify_phonemes` and `remove_stress_from_phonemes` achieve very similar things.

`remove_stress_from_phonemes` is used when converting English words to IPA and `simplify_phonemes` is used to remove the stress phonemes from the original CMUDICT list of phonemes that will later be used as target for this model. We will go from 68 phonemes when counting the stress levels of the vowel phonemes to 39 without.

```
def simplify_phonemes(phonemes):
    simplified_set = set()
    for phoneme in phonemes:
        # Check if the last character is a digit, if so, remo
        if phoneme[-1].isdigit():
            simplified_set.add(phoneme[:-1])
        else:
            simplified_set.add(phoneme)
    return sorted(simplified_set)

def remove_stress_from_phonemes(phonemes):
    return [phoneme[:-1] if phoneme[-1].isdigit()
    else phoneme for phoneme in phonemes]
```

Since CMUDICT has a dictionary of 134,000 words and their pronunciations as key value pairs

`english_to_phonetic(text)` then simply translates English text to its phonetics representation, removing the stress phonemes in the process, and if the word pronounciation cannot be found then it returns and empty space.

`ipa_to_num` maps each phoneme to an integer so we using the IPA vocab initialized earlier on

and `num_to_ipa` does the reverse process converting integers into phonemes.

```
ipa_to_num = tf.keras.layers.StringLookup(vocabulary=vocab_ip
num_to_ipa = tf.keras.layers.StringLookup(vocabulary=ipa_to_n
oov_token="", invert=True)
```

Loading the Data:

The loading functions are based off the followling code:

https://github.com/nicknochnack/LipNet

`def load_video(path:str) -> List[float]:`

reads goes through all 75 frames of the video, isolates the region around the mouth and transforms them into grayscale. Then each frame is normalized by subtracting the mean and dividing by the standard deviation of all 75 frames

```
# take data path and returns a list of floats that represent
def load_video(path:str) -> List[float]:

    # loop thru each frame of the video and store it in
    # an array called frames
    cap = cv2.VideoCapture(path)
    frames = []
    for _ in range(int(cap.get(cv2.CAP_PROP_FRAME_COUNT))):
        ret, frame = cap.read()
        # testing
        if not ret:
            continue  # Skip frames that can't be captured.
        frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)  # Ens
        #testing
        frame = tf.image.rgb_to_grayscale(frame)
        # isolating the mouth region (static slice function)
        # more advanced way to do it by extracting the lips
        # using DLIB to islate the mouth
        frames.append(frame[190:236,80:220,:])
    cap.release()

    #calculate the mean and std and scale
    # subtract the mean and divide by std
    mean = tf.math.reduce_mean(frames)
    std = tf.math.reduce_std(tf.cast(frames, tf.float32))
    return tf.cast((frames - mean), tf.float32) / std
```

`def load_alignments(path:str) -> List[str]:`

reads the data that is stored as “alignments” which are transcriptions of the words being spoken in the corresponding videos. Each word is translated to phonemes using the `english_to_phonetic` mentioned above.

```
# take in paths and load up aligments .align
def load_alignments(path:str) -> List[str]:
    with open(path, 'r') as f:
        lines = f.readlines()
    # tokens = []
    tokens_ipa = []

    # print(lines)

    for line in lines:
        line = line.split()
        if line[2] != 'sil':
            e_to_ipa = english_to_phonetic(line[2])
            tokens_ipa = [*tokens_ipa, ' ']
            for sublist in e_to_ipa:
              tokens_ipa.extend(sublist)
    return ipa_to_num(tokens_ipa)[1:]
```

`def load_data(path: str):`

loads the frames and their corresponding alignments when given a path

The data is then loaded from the path (only the data for speaker 1), shuffles, and the `load_data` function is mapped onto each piece of data so it can be stores as frames and alignments.

the data are then batches as groups of two, and padded out, specifically for the alignments since alignment lengths differ depending on the data point. `data.prefetch(tf.data.AUTOTUNE)` improves the training efficiency by pre-loading the data.

Then data is split in training and validation sets, using 85% of the data for training and 15% for validation. In hindsight I should have set apart some of the data for testing only and not just validation, but I thought I would need as much data for training as possible. If I could have a redo I would do a split of 75% training, 15% validation and 10% testing.

```
# Get data from speaker 1 (s1)
data = tf.data.Dataset.list_files('/content/drive/MyDrive/COG
# shuffle and load the data as frames and alignments
data = data.shuffle(500, reshuffle_each_iteration=False)
data = data.map(mappable_function)
# pad it out because alignmnets have different lengths
# padded batch group sizes of two, each one two sets of vids
data = data.padded_batch(2, padded_shapes=([75, 46, 140, 1],[
# pre loading to speed up training process
data = data.prefetch(tf.data.AUTOTUNE)
# split into train and validation sets
train = data.take(425)
test = data.skip(425)
```

Model Design:

The models used are based of the following code:

https://github.com/nicknochnack/LipNet

https://github.com/rizkiarm/LipNet

Dummy Model:

Since I ran out of compute units with Colab pro, I did not get to get any meaningful training done with my dummy model, however I kept it for future reference.

The dummy model closely follow the design of the main model, but more simplified (I could have simplified it more, but I did not get a chance to experiment much with the model parameter, layers and so on, I used it to gain a better understanding the main model design and its layers) It is a sequential model with an input layer of (75, 46, 140,1). 75 for the frames of the video, 46 × 140 for the dimension of the video and 1 for the colour channel the frames are in grayscale. Then a Convolution layer is added to handle the processing the data from the image. With 128 filters (should probably lower it to 32 as the output space of 128 is too high for a dummy model), kernel size of 3×3×3, and padding to preserve dimension of the input. And the activation function is set to 'relu' to introduce non linearity. The output from the Conv3D is then flattened in the "TimeDistribution" layer, which converts the output of the Conv3D from 75×46×140×128 to a 75×824320. Basically condenses all the data for each frame into a single dimension, hence 46×140×128 == 824,320. The final layer, the Dense layer, determines the final output of the model given the features that were learned by the previous layers as input, with `ipa_to_num.vocabulary_size()+1` being the initial phoneme vocab size (the +1 is for handling any potential unknown values that the model might predict that is outside of the set vocab) 'activation='softmax' enforces that the output layer will give a probability for each possible phoneme given in the vocab, equalling to 1 for each frame of the video.

```
#dummy model
from keras.models import Sequential
from keras.layers import Input, Conv3D, Flatten, Dense, TimeD
9
dummy_model = Sequential()
dummy_model.add(Input(shape=(75, 46, 140, 1))) # Define the i
dummy_model.add(Conv3D(128, kernel_size=3, padding='same', ac
dummy_model.add(TimeDistributed(Flatten()))
dummy_model.add(Dense(ipa_to_num.vocabulary_size()+1,kernel_i
dummy_model.summary()
```

Model:

I didn't have much many Colab Pro compute units to experiment with the parameters of the different layers of mode, so I followed the same design that I saw here and here to ensure that I get reliable results. If I had infinite compute units, I would probably reduce some of the layers in the model (for example removing one of the Conv3D and MaxPooling layers) or reduced the filter number from the Conv3D (maybe halve it to 64) or the halve the units the LSTM layer and increase the dropout. Depending how the model performs, I would adjust these layers and parameters in order to minimize over-fitting and see how they generalize with other speaker data.

The main model follows the some of the features of the dummy model. However there are now two extra Conv3D layers each followed by a MaxPooling layer. The MaxPooling layer reduces the dimensions of the input features by half as it reduces each frame to a 2×2 square and takes the max value inside, reducing the computational load by taking in the most prominent features aand keeps the temporal resolution since each of the 75 frames are accounted for. The increase from 128 to 256 filters ensures that the model can learn the more complex features in the data and the decrease to 75 helps the model focus on the more relevant features as all the complexity has already been determined by the previous layers. Also 75 filters can decrease the likelihood of over fitting. The other difference between the two models is the Bidirectional LSTM layers tjat are added here. LSTMs are used to handle sequential data and it is bidirectional so each video of a speaker can gain information from the previous frames as well as the upcoming frames. Thus each frame has the context of the entire video that it is a part of. The kernel_initializer determines how the weights of are initialized before training. Setting return_sequences to true ensures the output of the LSTM is the entire sequence instead of just the last frame, thus keeping temporal resolution. And the dropout rate of 0.5 drops half of the inputs units as it sets them to 0 at each update of training, this help reduce the chances of overfitting.

The rest of the model is explained in the dummy portion

```
# from keras.initializers import Orthogonal
from keras.models import Sequential
from keras.layers import Input, Conv3D, Flatten, Dense, TimeD

model = Sequential()
model.add(Conv3D(128, 3, input_shape=(75,46,140,1), padding='
model.add(MaxPool3D((1,2,2)))

model.add(Conv3D(256, 3, padding='same',  activation='relu'))
model.add(MaxPool3D((1,2,2)))

model.add(Conv3D(75, 3, padding='same',  activation='relu'))
model.add(MaxPool3D((1,2,2)))

model.add(TimeDistributed(Flatten()))

model.add(Bidirectional(LSTM(128, kernel_initializer='Orthogo
model.add(Dropout(.5))

model.add(Bidirectional(LSTM(128, kernel_initializer='Orthogo
model.add(Dropout(.5))

model.add(Dense(ipa_to_num.vocabulary_size()+1, kernel_initia
model.summary()
```

CTC Loss Function:

The code for CTCLoss is taken from this tutorial

Connectionist Temporal Classification well equipped to handle temporal inputs and the labelled outputs are not perfectly aligned and there are transition frames where the input could have multiple labels, and a label is responsible for a sequence of time frames. In this example of there are frames that capture transitory mouth movements of two phonemes and CTC can help predict that sequence. In simple terms, CTC provides a probability of all the labels for each time steps, and used a decoding function to predict the most likely output for each time step and thus predicting the sequence of the labels.

CTC takes in the target value (y_true) and the predicted value (y_pred), determines the length of the batch that is being processed at once, determines how long each predicted sequence is len(y_pred), determines how long the true sequence is len(y_true), standardizes the sequence lengths across the batch and uses then uses the built it ctc_batch_cost function to calculate the loss.

```
def CTCLoss(y_true, y_pred):
    batch_len = tf.cast(tf.shape(y_true)[0], dtype="int64")
    input_length = tf.cast(tf.shape(y_pred)[1], dtype="int64"
    label_length = tf.cast(tf.shape(y_true)[1], dtype="int64"

    input_length = input_length * tf.ones(shape=(batch_len, 1
    label_length = label_length * tf.ones(shape=(batch_len, 1

    loss = tf.keras.backend.ctc_batch_cost(y_true, y_pred, in
    return loss
```

TensorBoard and Accuracy metric:

Code here references this tutorial

Setup TensorBoard in the notebook and add a tensorboard_callback to create logs for the model that is being trained

```
# Set up TensorBoard logging
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%
tensorboard_callback = TensorBoard(log_dir=log_dir, histogram
```

CharacterAccuracy function calculates the percentage of phonemes the model accurately calculates. Using CTC's decoder, it decodes the predicted values and compares them to the true values. y_true and y_pred are both standardized and padded to make sure the are the same length and the model won't throw errors during training. The accuracy of each batch is then calculated by counting how many predicted values and true values match, and then divided by the total to get the correct percentage and returns it as the result, and reset_state is used to clear the prediction score.

```

class CharacterAccuracy(tf.keras.metrics.Metric):
    def __init__(self, name='character_accuracy', **kwargs):
        super(CharacterAccuracy, self).__init__(name=name, **
        self.correct_predictions = self.add_weight(name='corr
        self.total_predictions = self.add_weight(name='total'

    def update_state(self, y_true, y_pred, sample_weight=None
        input_length = tf.math.reduce_sum(tf.cast(tf.not_equa
        decoded, log_prob = tf.nn.ctc_greedy_decoder(tf.trans
        y_pred_decoded = tf.sparse.to_dense(decoded[0], defau
        y_true_decoded = tf.cast(y_true, 'int32')
        y_pred_decoded = tf.cast(y_pred_decoded, 'int32')

        # calculate the max length for padding (was getting e
        max_length = tf.maximum(tf.shape(y_true_decoded)[1],
        y_true_decoded = tf.pad(y_true_decoded, [[0, 0], [0, 
        y_pred_decoded = tf.pad(y_pred_decoded, [[0, 0], [0, 

        # calculating the accuracy per batch
        correct = tf.reduce_sum(tf.cast(tf.equal(y_pred_decod
        total = tf.cast(tf.size(y_true_decoded), tf.float32)

        self.correct_predictions.assign_add(correct)
        self.total_predictions.assign_add(total)

    def result(self):
        return self.correct_predictions / self.total_predicti

    def reset_state(self):
        self.correct_predictions.assign(0.0)
        self.total_predictions.assign(0.0)
```

References

CTC:

https://www.cs.toronto.edu/~graves/icml_2006.pdf
CTC Loss function:

https://keras.io/examples/audio/ctc_asr/
ARPABET:

http://www.speech.cs.cmu.edu/cgi-bin/cmudict

https://github.com/arianshamei/phonemic-parser/blob/main/phonemic_parser.py

TensorBoard:

https://www.tensorflow.org/tensorboard/get_started
Data:

https://spandh.dcs.shef.ac.uk/gridcorpus/
Model Design Data Loading Training and Notebook structure:

https://github.com/rizkiarm/LipNet

https://github.com/nicknochnack/LipNet