

Deep Learning Model That Can Lip Read

Using Python And Tensorflow

Mayank Pareek¹, Tulsi Patel²

College Of Science and Health, Depaul University, Chicago - USA

{tpatel91,mpareek}@depaul.edu

Here we are going to implement the paper

[LIPNET: END-TO-END SENTENCE-LEVEL LIPREADING](#)

Lipreading is the task of decoding text from the movement of a speaker's mouth. Traditional approaches separated the problem into two stages: designing or learning visual features, and prediction. More recent deep lip reading approaches are end-to-end trainable (Wand et al., 2016; Chung & Zisserman, 2016a). However, existing work on models trained end-to-end perform only word classification, rather than sentence-level sequence prediction. Studies have shown that human lipreading performance increases for longer words (Easton & Basala, 1982), indicating the importance of features capturing temporal context in an ambiguous communication channel. Motivated by this observation, we present LipNet, a model that maps a variable-length sequence of video frames to text, making use of spatiotemporal convolutions, a recurrent network, and the connectionist temporal classification loss, trained entirely end-to-end. To the best of our knowledge, LipNet is the first end-to-end sentence-level lipreading model that simultaneously learns spatiotemporal visual features and a sequence model. On the GRID corpus, LipNet achieves 95.2% accuracy in sentence-level, overlapped speaker split task, outperforming experienced human lip readers and the previous 86.4% word-level state-of-the-art accuracy (Gergen et al., 2016).

Import Dependencies

opencv-python: For video processing.

matplotlib: For visualizations.

imageio: For saving animations.

gdown: For downloading files from Google Drive.

tensorflow: For deep learning.

```
import os
import cv2
import tensorflow as tf
import numpy as np
from typing import List
from matplotlib import pyplot as plt
import imageio
```

Checks for GPU availability and configures TensorFlow to use GPU memory efficiently, avoiding memory allocation issues.

```
Check for GPUs Allotted:
tf.config.list_physical_devices('GPU')
physical_devices = tf.config.list_physical_devices('GPU')
try:
    tf.config.experimental.set_memory_growth(physical_devices[0],
    True)
except:
    pass
```

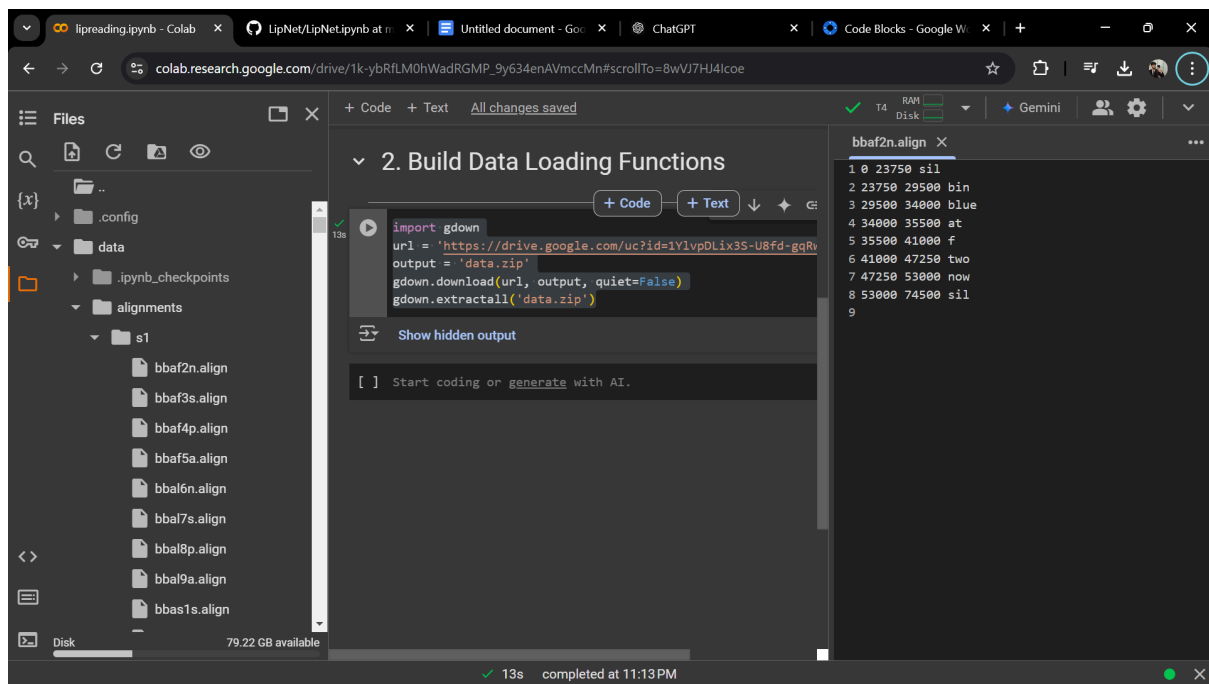
Now once we setup everything we begin with Building Data Loading Functions. Here we need two data loading functions: 1st to read our videos and second is to pre-process our annotations which in this case is going to be something that the person speaks.

Here we use [gdown](#) that will help us download data from google drive, now here we make it straight forward by using the data that was meant to be used for lip reading models. Now here the data set is going to be an extract or i would say small portion of grid data.

```
url = 'https://drive.google.com/uc?id=1YlvpDLix3S-U8fd-gqRwPcWAXm8JwjL'
output = 'data.zip'
gdown.download(url, output, quiet=False)
gdown.extractall('data.zip')
```

Here we just simply downloaded the dataset into a zip file.

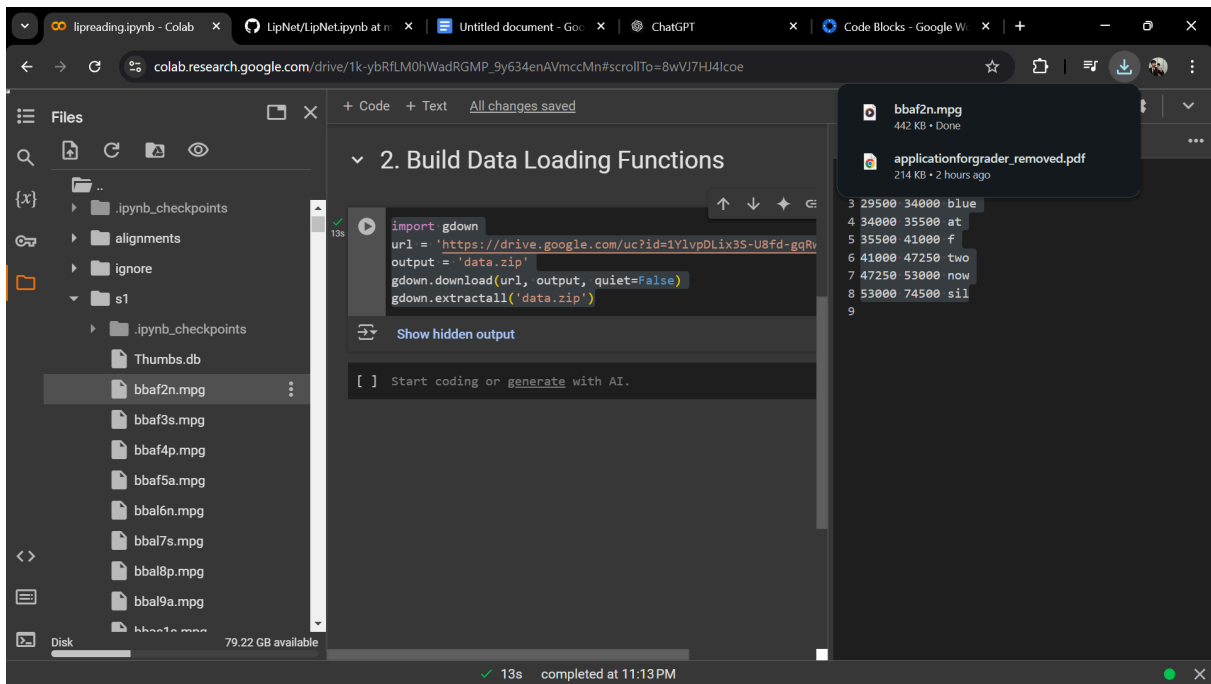
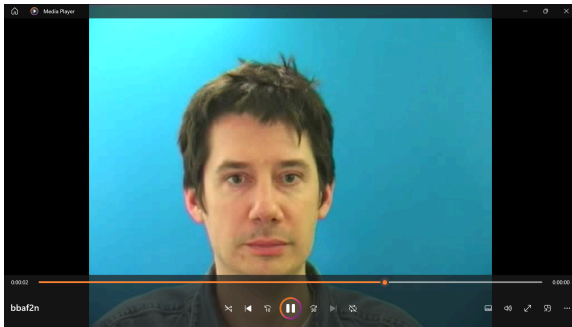
After carefully reviewing the data we see a data folder containing alignments and s1. Further alignments\s1 will have .align files containing text



0 23750 sil
 23750 29500 bin
 29500 34000 blue
 34000 35500 at
 35500 41000 f
 41000 47250 two
 47250 53000 now
 53000 74500 sil

Here this shows the annotations for video bbaf2n, going back to data/s1

We get video files looking at bbaf2n.mpg we get the same annotation spoken by the speaker in the video



Now we have used a function to load this data

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

    cap = cv2.VideoCapture(path)
    frames = []
    for _ in range(int(cap.get(cv2.CAP_PROP_FRAME_COUNT))):
        ret, frame = cap.read()
        frame = tf.image.rgb_to_grayscale(frame)
        frames.append(frame[190:236,80:220,:])
    cap.release()

    mean = tf.math.reduce_mean(frames)
    std = tf.math.reduce_std(tf.cast(frames, tf.float32))
    return tf.cast((frames - mean), tf.float32) / std
```

Reads Video Files, Converts Them To Grayscale, Crops Frames, Normalizes Pixel Values, And Returns Preprocessed Frames.

Here

```
frames.append(frame[190:236,80:220,:])
```

It will basically isolate the mouth or the region of the lip.

Now in the original paper they have used an advanced version to detect the lip using DLib. so here we are looping through the video storing the frames inside our own set of arrays `frames = []` Converting it from rgb to grayscale (Making less data to preprocess). we're then standardizing it so we're calculating the mean calculating the standard deviation that's just good practice to scale your data and then we're casting it to a float 32 and dividing it by the standard deviation so if we go and run that that is our load video function now done now we're going to go on ahead and Define our vocab now a vocab is really just going to be every single character which we might expect to encounter within our annotation so bin blew at f2 now we've also got a couple of numbers in there as well.

Function Definition

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

- **def:** This keyword is used to define a function.
 - **load_video:** The name of the function. It indicates the purpose of the function, which is to load and process video data.
 - **path: str:** The function expects a single argument path, which is a string representing the file path to the video.
 - **-> List[float]:** This is a type hint that indicates the function will return a list containing floating-point numbers.
-

Open the Video File

```
cap = cv2.VideoCapture(path)
```

- **cv2.VideoCapture(path):** This creates a video capture object cap using OpenCV. It is used to read frames from the video file specified by the path.
 - **cap:** Stores the video capture object, which allows access to video properties and frames.
-

Initialize an Empty List for Frames

```
frames = []
```

- **frames = []**: Initializes an empty list to store individual frames from the video after processing.
-

Iterate Through Video Frames

```
for _ in range(int(cap.get(cv2.CAP_PROP_FRAME_COUNT))):
```

- **for _**: A loop to iterate over a range of numbers. The underscore (`_`) is used when the loop variable isn't needed.
 - **range(int(cap.get(cv2.CAP_PROP_FRAME_COUNT)))**:
 - **cap.get(cv2.CAP_PROP_FRAME_COUNT)**: Gets the total number of frames in the video using OpenCV.
 - **int()**: Converts the frame count (a float) into an integer.
 - **range()**: Creates a range object to loop over each frame.
-

Read Each Frame

```
ret, frame = cap.read()
```

- **cap.read()**: Reads the next frame from the video.
 - **ret**: A boolean indicating if the frame was read successfully.
 - **frame**: The actual frame data in the form of a NumPy array.
-

Convert Frame to Grayscale

```
frame = tf.image.rgb_to_grayscale(frame)
```

- **tf.image.rgb_to_grayscale(frame)**: Uses TensorFlow to convert the frame from RGB (color) to grayscale. The grayscale frame has one channel instead of three.
-

Crop Frame

```
frames.append(frame[190:236, 80:220, :])
```

- **frame[190:236, 80:220, :]**: Crops the frame to a specific region.
 - **190:236**: Selects rows (height) from 190 to 236 (inclusive).
 - **80:220**: Selects columns (width) from 80 to 220 (inclusive).
 - **::**: Keeps all grayscale channels (though grayscale has only one channel).
 - **frames.append(...)**: Adds the cropped frame to the frames list.
-

Release Video Capture

```
cap.release()
```

- **cap.release()**: Frees the resources associated with the video capture object.
-

Calculate Mean and Standard Deviation

```
mean = tf.math.reduce_mean(frames)
std = tf.math.reduce_std(tf.cast(frames, tf.float32))
```

- **tf.math.reduce_mean(frames)**: Computes the mean value of all pixel intensities across all frames.
 - **tf.math.reduce_std(...)**: Computes the standard deviation of all pixel intensities. It requires the input to be cast to a float data type.
 - **tf.cast(frames, tf.float32)**: Converts the data type of frames from its original type (likely uint8) to float32 for numerical stability.
-

Normalize the Frames

```
return tf.cast((frames - mean), tf.float32) / std
```

- **frames - mean**: Subtracts the mean from each frame to center the pixel values around zero.
- **tf.cast(..., tf.float32)**: Ensures the resulting array is of type float32.
- **/ std**: Divides by the standard deviation to normalize the pixel values.

After this operation:

- Pixel values have a mean of 0.
 - Pixel values have a standard deviation of 1.
 - **return**: Outputs the list of normalized frames.
-

Summary

The function:

1. Reads a video file.
2. Converts each frame to grayscale.
3. Crops each frame to a specific region of interest.
4. Normalizes the pixel intensities of all frames using mean and standard deviation.
5. Returns a list of normalized frames as float32 tensors.

Now creating a list vocab

```
vocab = [x for x in "abcdefghijklmnopqrstuvwxyz'?!123456789 "]
```

```
char_to_num = tf.keras.layers.StringLookup(vocabulary=vocab, oov_token="")
num_to_char = tf.keras.layers.StringLookup(
    vocabulary=char_to_num.get_vocabulary(), oov_token="", invert=True
)

print(
    f"The vocabulary is: {char_to_num.get_vocabulary()} "
    f"(size = {char_to_num.vocabulary_size()})"
)
```


This code snippet is part of a process that maps characters to numerical indices and vice versa, using TensorFlow's StringLookup layer. Here's a detailed breakdown of each line:

Character-to-Number Mapping

```
char_to_num =  
tf.keras.layers.StringLookup(vocabulary=vocab,  
oov_token="")
```

- **char_to_num**: This variable stores an instance of the StringLookup layer, which is used to map strings (characters) to unique numerical indices.
 - **tf.keras.layers.StringLookup(...)**: Creates a TensorFlow layer that performs string-to-index conversions. It assigns an integer index to each character in the vocabulary.
 - **vocabulary=vocab**:
 - **vocab**: A list or array of unique characters used in the mapping (e.g., ['a', 'b', 'c', ...]).
 - The vocab parameter specifies the set of characters that will be mapped to indices.
 - **oov_token=""**:
 - The oov_token (Out-Of-Vocabulary token) is used for characters that are not found in the provided vocabulary.
 - Here, an empty string ("") is specified as the out-of-vocabulary token.
-

Number-to-Character Mapping

```
num_to_char = tf.keras.layers.StringLookup(  
    vocabulary=char_to_num.get_vocabulary(),  
    oov_token="", invert=True  
)
```

- **num_to_char**: This variable stores another StringLookup layer, but it is configured to perform the reverse operation—mapping indices back to characters.
 - **tf.keras.layers.StringLookup(...)**: Creates a new instance of the StringLookup layer.
 - **vocabulary=char_to_num.get_vocabulary()**:
 - **char_to_num.get_vocabulary()**: Retrieves the vocabulary from the char_to_num layer. This ensures that the two layers (char_to_num and num_to_char) use the same mapping.
 - **oov_token=""**:
 - Specifies the out-of-vocabulary token for this reverse mapping.
 - **invert=True**:
 - The invert parameter tells the layer to map indices back to strings instead of strings to indices.
-

Print the Vocabulary and Its Size

```
print(  
    f"The vocabulary is: {char_to_num.get_vocabulary()} "  
    f"(size ={char_to_num.vocabulary_size()})"  
)
```

- **print(...)**: Outputs information about the vocabulary used by the char_to_num layer.
 - **f"The vocabulary is: {char_to_num.get_vocabulary()} "**:
 - **f"..."**: This is an f-string, which allows embedding variables and expressions inside curly braces {} for dynamic string formatting.
 - **char_to_num.get_vocabulary()**:
 - Retrieves the list of all unique characters in the vocabulary. It includes all characters from the vocab list provided earlier, along with any special tokens like the out-of-vocabulary token (if applicable).
 - This part of the f-string displays the list of characters in the vocabulary.

- `f"(size={char_to_num.vocabulary_size()})"`:
 - `char_to_num.vocabulary_size()`:
 - Retrieves the total size of the vocabulary, which includes all unique characters and possibly the out-of-vocabulary token.
 - This part of the f-string displays the size of the vocabulary.
-

Example Walkthrough

Let's assume the following:

- `vocab = ['a', 'b', 'c', 'd']`

Step-by-Step Process:

1. **Define `char_to_num`:**
 - Maps 'a' -> 1, 'b' -> 2, 'c' -> 3, 'd' -> 4 (index 0 is reserved for padding).
 - If a character is not in the vocabulary (e.g., 'z'), it is mapped to the out-of-vocabulary token "".
 2. **Define `num_to_char`:**
 - Reverses the mapping:
 - 1 -> 'a', 2 -> 'b', 3 -> 'c', 4 -> 'd'.
 3. **Print the Vocabulary:**
 - `char_to_num.get_vocabulary()` outputs `['[UNK]', 'a', 'b', 'c', 'd']`, where `[UNK]` represents the unknown token.
 - `char_to_num.vocabulary_size()` outputs 5, indicating the total number of tokens (including `[UNK]`).
-

Output

If `vocab = ['a', 'b', 'c', 'd']`, the output of the print statement will be:

The vocabulary is: `['[UNK]', 'a', 'b', 'c', 'd']` (size =5)

```
def load_alignments(path:str) -> List[str]:
    with open(path, 'r') as f:
        lines = f.readlines()
    tokens = []
    for line in lines:
        line = line.split()
        if line[2] != 'sil':
            tokens = [*tokens, line[2]]
    return char_to_num(tf.reshape(tf.strings.unicode_split(tokens,
input_encoding='UTF-8'), (-1)))[1:]
```

This function, `load_alignments`, processes a text file containing alignment information (presumably phonetic or linguistic data) and converts it into numerical tokens using the previously defined `char_to_num` mapping. Here is a detailed explanation of the code, line by line:

Function Definition

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

- **def load_alignments:** Declares a function named `load_alignments`.
 - **path: str:** The parameter `path` expects a string representing the path to the alignment file.
 - **-> List[str]:** Indicates that the function returns a list of strings.
-

Open the File and Read Lines

```
with open(path, 'r') as f:
    lines = f.readlines()
```

- **with open(path, 'r') as f:**
 - Opens the file at the specified path in read mode ('r').
 - The `with` statement ensures that the file is automatically closed when the block is exited.
 - **f:** A file object representing the opened file.
- **lines = f.readlines():**
 - Reads all lines of the file into a list called `lines`.
 - Each line in the file becomes an element in this list as a string.

Initialize Tokens List

```
tokens = []
```

- Initializes an empty list called tokens. This will store processed tokens extracted from the file.

Iterate Through Lines

```
for line in lines:
    line = line.split()
    if line[2] != 'sil':
        tokens = [*tokens, ' ', line[2]]
```

Line-by-Line Breakdown:

1. **for line in lines::**
 - Iterates over each line in the lines list.
 - **line**: Represents the current line being processed.
 2. **line = line.split():**
 - Splits the current line into a list of words or elements, using whitespace as the delimiter.
 - **line** now contains the line as a list of its individual components.
 3. **if line[2] != 'sil'::**
 - Checks whether the third element (line[2]) in the split line is not equal to the string 'sil'.
 - **'sil'** represents silence, which is skipped.
 4. **tokens = [*tokens, ' ', line[2]]:**
 - Uses unpacking (*tokens) to append a space (' ') followed by the third element (line[2]) to the tokens list.
 - **Purpose**: This ensures proper spacing between tokens while building the sequence.
-

Convert Tokens to Numeric Form

```
return char_to_num(  
    tf.reshape(  
        tf.strings.unicode_split(tokens,  
input_encoding='UTF-8'),  
        (-1)  
    )  
)[1:]
```

Step-by-Step Explanation:

1. **tf.strings.unicode_split(tokens, input_encoding='UTF-8'):**
 - Splits each string in the tokens list into its individual Unicode characters.
 - **tokens:** Contains a sequence of strings (e.g., [' ', 'a', 'b', ' ']).
 - **input_encoding='UTF-8':** Specifies the encoding of the input strings.
2. **tf.reshape(..., (-1)):**
 - Flattens the resulting tensor into a 1D tensor with shape (-1) (all elements in a single dimension).
 - This is needed for compatibility with char_to_num.
3. **char_to_num(...):**
 - Converts the characters from the Unicode-split tensor into numerical indices using the char_to_num mapping layer.
4. **[1:]**:
 - Removes the first element of the numerical sequence.
 - This may exclude a leading padding or other special token added during processing.

What the Function Does

1. Opens a text file containing alignment information.
2. Extracts tokens from each line, skipping tokens labeled as 'sil'.
3. Converts the tokens into a flat sequence of numerical indices using the char_to_num mapping.

Example Walkthrough

The file contains:

```
0 23750 sil
23750 29500 bin
29500 34000 blue
34000 35500 at
35500 41000 f
41000 47250 two
47250 53000 now
53000 74500 sil
```

Step-by-Step Execution

1. Read File Contents

Lines: After reading, the lines list will be:

```
[
    "0 23750 sil",
    "23750 29500 bin",
    "29500 34000 blue",
    "34000 35500 at",
    "35500 41000 f",
    "41000 47250 two",
    "47250 53000 now",
    "53000 74500 sil"
]
```

2. Initialize tokens:

- An empty list `tokens = []` is created to store processed tokens.

3. Process Each Line

- Looping through each line, the code splits the line and checks if the third element (line[2]) is **not** 'sil'. If it's not 'sil', it appends a space (' ') and the third element (line[2]) to tokens.
- **Iteration Details:**
 - Line 1 ("0 23750 sil"): Skipped (line[2] == 'sil').
 - Line 2 ("23750 29500 bin"): Adds ' ' and 'bin' to tokens.
 - tokens = [' ', 'bin']
 - Line 3 ("29500 34000 blue"): Adds ' ' and 'blue' to tokens.
 - tokens = [' ', 'bin', ' ', 'blue']
 - Line 4 ("34000 35500 at"): Adds ' ' and 'at' to tokens.
 - tokens = [' ', 'bin', ' ', 'blue', ' ', 'at']
 - Line 5 ("35500 41000 f"): Adds ' ' and 'f' to tokens.
 - tokens = [' ', 'bin', ' ', 'blue', ' ', 'at', ' ', 'f']
 - Line 6 ("41000 47250 two"): Adds ' ' and 'two' to tokens.
 - tokens = [' ', 'bin', ' ', 'blue', ' ', 'at', ' ', 'f', ' ', 'two']
 - Line 7 ("47250 53000 now"): Adds ' ' and 'now' to tokens.
 - tokens = [' ', 'bin', ' ', 'blue', ' ', 'at', ' ', 'f', ' ', 'two', ' ', 'now']
 - Line 8 ("53000 74500 sil"): Skipped (line[2] == 'sil').

Final tokens List:

```
tokens = [' ', 'bin', ' ', 'blue', ' ', 'at', ' ', 'f', ' ', 'two', ' ', 'now']
```

4. Unicode Splitting

- **tf.strings.unicode_split(tokens, input_encoding='UTF-8'):**

Splits the tokens list into individual characters, including spaces:

```
[' ', 'b', 'i', 'n', ' ', 'b', 'l', 'u', 'e', ' ', 'a', 't', ' ', 'f', ' ', 't', 'w', 'o', ' ', 'n', 'o', 'w']
```

5. Reshaping

- **tf.reshape(..., (-1)):**

Flattens the sequence into a single-dimensional tensor:


```
[' ', 'b', 'i', 'n', ' ', 'b', 'l', 'u', 'e', ' ', 'a', 't', ' ', 'f', ' ', 't', 'w', 'o', ' ', 'n', 'o', 'w']
```

6. Convert Characters to Numeric Tokens

- `char_to_num(...)`:
 - Maps each character to a corresponding numerical index using the `char_to_num` layer.

Example mapping (assuming a vocabulary like [' ', 'a', 'b', ..., 'z']):

```
' ' -> 0
'b' -> 2
'i' -> 9
'n' -> 14
'l' -> 12
'u' -> 21
...
```

Resulting Numeric Sequence:

```
[0, 2, 9, 14, 0, 2, 12, 21, 5, 0, 1, 20, 0, 6, 0, 20, 23, 15, 0, 14, 15, 23]
```

7. Remove Leading Element

- `[1:]`:
 - Removes the first element (often a padding or space token).

Final Output:

```
[2, 9, 14, 0, 2, 12, 21, 5, 0, 1, 20, 0, 6, 0, 20, 23, 15, 0, 14, 15, 23]
```

Summary

The function processes the alignment file to:

1. Exclude lines with 'sil'.
2. Create a sequence of characters (with spaces between words).
3. Convert characters to numerical indices using the `char_to_num` mapping.

4. Return the processed sequence as a tensor of numeric tokens, excluding the leading element.

For your example, the silence tokens ('sil') at the start and end are ignored, and the function outputs the numerical indices representing the words bin blue at f two now.

```
def load_data(path: str):
    path = bytes.decode(path.numpy())
    #file_name = path.split('/')[ -1].split('.')[0]
    # File name splitting for windows
    file_name = path.split('\\')[ -1].split('.')[0]
    video_path = os.path.join('data', 's1', f'{file_name}.mpg')
    alignment_path = os.path.join('data', 'alignments', 's1', f'{file_name}.align')
    frames = load_video(video_path)
    alignments = load_alignments(alignment_path)

    return frames, alignments
```

Let's analyze the `load_data` function in detail step by step, covering its functionality and role in loading video and alignment data:

Code Analysis

1. Decode Path (String Conversion)

```
path = bytes.decode(path.numpy())
```

- **Input:** The `path` parameter is initially a TensorFlow tensor (likely of type `tf.string` containing the path to a file).
 - **Purpose:**
 - `path.numpy()` converts the TensorFlow tensor into a NumPy byte string.
 - `bytes.decode(...)` then converts the byte string into a regular Python string.
 - **Result:** `path` is now a plain Python string representing the file path.
-

2. Extract File Name

```
file_name = path.split('\\')[-1].split('.')[0]
```

- **Input:** path is a string representing the full file path, e.g., C:\data\video001.align.
 - **Purpose:**
 - Splits the path by the Windows directory separator (\\).
 - Takes the last element of the split result (e.g., video001.align).
 - Splits that by the file extension separator (.) and selects the first part (e.g., video001).
 - **Result:** file_name contains the base file name without the extension (video001 in this example).
-

3. Construct Video Path

```
video_path = os.path.join('data', 's1', f'{file_name}.mpg')
```

- **Input:** The file_name derived earlier (video001).
- **Purpose:**
 - Constructs the path to the corresponding video file.
 - Assumes the videos are stored in a directory structure starting with 'data/s1' and are named {file_name}.mpg.

Result: For file_name = "video001", video_path would be:

```
data/s1/video001.mpg
```

4. Construct Alignment Path

```
alignment_path = os.path.join('data', 'alignments', 's1', f'{file_name}.align')
```

- **Input:** The file_name derived earlier (video001).
- **Purpose:**
 - Constructs the path to the corresponding alignment file.
 - Assumes alignments are stored in 'data/alignments/s1' and are named {file_name}.align.

Result: For file_name = "video001", alignment_path would be:

```
data/alignments/s1/video001.align
```

5. Load Video Data

```
frames = load_video(video_path)
```

- **Purpose:**
 - Calls the load_video function to read and preprocess frames from the video file at video_path.
 - **Result:**
 - frames contains a processed representation of video frames, such as a normalized tensor or a list of frames.
-

6. Load Alignment Data

```
alignments = load_alignments(alignment_path)
```

- **Purpose:**
 - Calls the load_alignments function to read and preprocess the alignment information from the alignment file at alignment_path.
 - **Result:**
 - alignments contains the processed alignment tokens as a tensor or list.
-

7. Return Loaded Data

```
return frames, alignments
```

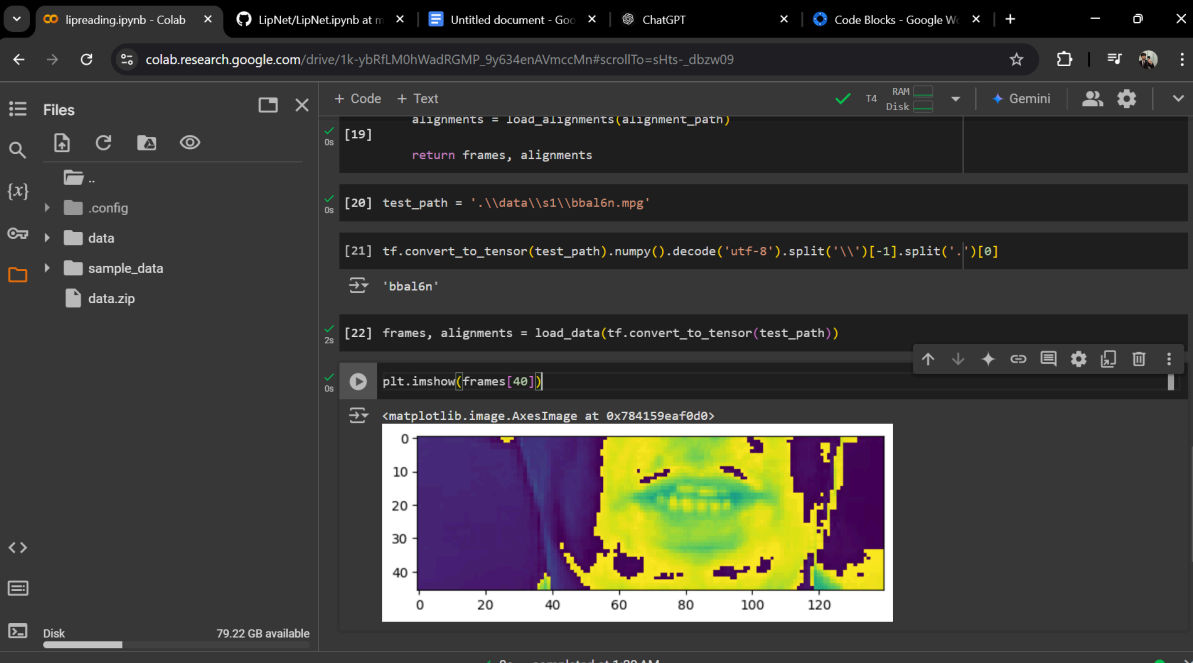
- **Purpose:**
 - Combines the video frame data (frames) and alignment token data (alignments) into a tuple and returns it.
- **Result:**

- The function returns a tuple (frames, alignments) that can be used in further processing, such as model training or inference.

Summary

The `load_data` function:

1. Decodes the TensorFlow tensor path into a Python string.
2. Extracts the base file name from the input path.
3. Constructs paths for the corresponding video and alignment files based on a predefined directory structure.
4. Loads and preprocesses video frames using `load_video`.
5. Loads and preprocesses alignment data using `load_alignments`.
6. Returns the preprocessed frames and alignments as a tuple.



The screenshot shows a Google Colab notebook interface. The left sidebar displays the file explorer with a directory structure including `.config`, `data`, `sample_data`, and `data.zip`. The main code area shows several cells. Cell [19] defines `alignments = load_alignments(alignment_path)` and `return frames, alignments`. Cell [20] sets `test_path = '..\\data\\s1\\bba16n.mpg'`. Cell [21] processes the path: `tf.convert_to_tensor(test_path).numpy().decode('utf-8').split('\\')[1].split('.')[0]`, resulting in `'bba16n'`. Cell [22] calls `frames, alignments = load_data(tf.convert_to_tensor(test_path))`. Below the code, the output of `plt.imshow(frames[40])` is displayed as a heatmap visualization of a video frame, showing a face with a color gradient from purple to yellow. The status bar at the bottom indicates the notebook is completed at 1:39 AM.

```
test_path = '..\\data\\s1\\bba16n.mpg'
tf.convert_to_tensor(test_path).numpy().decode('utf-8').split('\\')[1].split('.')[0]
frames, alignments = load_data(tf.convert_to_tensor(test_path))
plt.imshow(frames[40])
```

This function serves as a bridge to connect raw file paths with their processed contents, ensuring the data is in the correct format for downstream tasks.

Code Analysis

```
frames, alignments = load_data(tf.convert_to_tensor(test_path))
```

1. tf.convert_to_tensor(test_path)

- **Purpose:** Converts the Python variable test_path (which is expected to be a string representing a file path) into a TensorFlow tensor of type tf.string.
- **Input:**
 - test_path is a string, e.g., "data/test_video.align".
- **Why Tensor?**
 - TensorFlow often requires tensors as inputs, especially when using its functions or datasets pipeline.
- **Output:** A TensorFlow tensor representation of the file path.

Example:

```
test_path = "data/test_video.align"  
tf.convert_to_tensor(test_path)
```

```
# Output: <tf.Tensor: shape=(), dtype=string,  
numpy=b'data/test_video.align'>
```

2. load_data(...)

- **Purpose:** Calls the load_data function with the tensorized file path.
 - **Behavior:**
 - Internally, load_data decodes the tensor to a string path (bytes.decode(path.numpy())).
 - It then processes the video and alignment files located at the paths derived from test_path.
-

3. frames, alignments = ...

- **Purpose:** Unpacks the return value of `load_data` into two variables:
 - frames: Represents the processed video frames loaded and normalized by the `load_video` function.
 - alignments: Represents the alignment tokens loaded and transformed by the `load_alignments` function.
-

Example Workflow

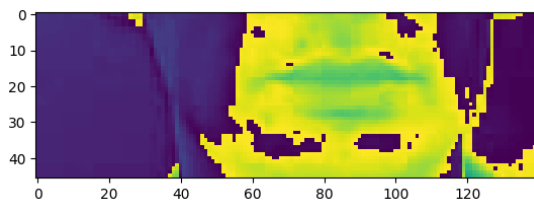
Assume:

- `test_path = "data/s1/video001.align"`

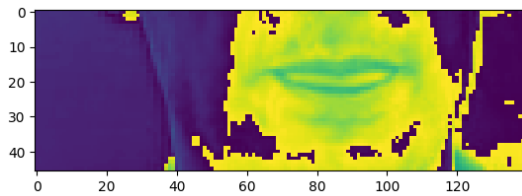
Execution:

1. `test_path` is converted into a tensor:
`tf.convert_to_tensor("data/s1/video001.align")`.
2. `load_data` processes the video (`data/s1/video001.mpg`) and alignment (`data/alignments/s1/video001.align`) files:
 - **Video:** Normalizes the frames and extracts a specific region.
 - **Alignment:** Converts text tokens into numerical representations.
3. The outputs are unpacked:
 - frames could be a tensor or list of preprocessed video frames.
 - alignments could be a tensor or list of numerical alignment tokens.

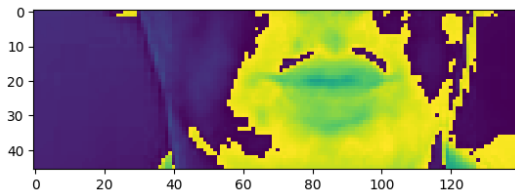
Frame 0:



Frame 11:



Frame 25:



Cool Right ?

Now You read so far thank you !

```
tf.strings.reduce_join([bytes.decode(x) for x in
num_to_char	alignments.numpy()).numpy()])
```

1. Alignments Tensor:

- The alignments tensor contains numerical tokens representing words (or subwords) extracted from the alignment file.
- For example, if `alignments.numpy()` was `[2, 3, 4, 5, 6]`, it would correspond to tokens based on the vocabulary.

2. num_to_char Transformation:

- The `num_to_char` mapping layer translates numerical indices into their string equivalents.
- If your vocabulary was something like `[' ', ' ', 'bin', 'blue', 'at', 'l', 'six', 'now', 'sil']`, then:
 - 2 -> 'bin'
 - 3 -> 'blue'
 - 4 -> 'at'
 - 5 -> 'l'
 - 6 -> 'six'
 - 7 -> 'now'

3. Decoding:

- The `bytes.decode(x)` operation converts these bytes-like strings into regular Python strings.

After decoding, the list might look like:

```
['bin', 'blue', 'at', 'l', 'six', 'now']
```

4. Joining:

The `tf.strings.reduce_join()` function concatenates these strings into a single string with no delimiter by default:

```
'binblueatlixnow'
```

If there are implicit spaces from the alignment or manual adjustments, the result will include them:

```
'bin blue at l six now'
```

Why This Result?

- The numerical tokens likely mapped to words via `num_to_char`.
- These words include:
 - 'bin'
 - 'blue'
 - 'at'
 - 'l'
 - 'six'
 - 'now'
- There were spaces or delimiters in the original data that were preserved during concatenation.

The final result:

```
<tf.Tensor: shape=(), dtype=string, numpy=b'bin blue at l  
six now'>
```

Outcome Analysis

The string representation reflects the alignment tokens converted back to their original words:

- "bin blue at l six now"

This matches the processed alignment data, showing the words spoken in the video segment with any silence (sil) removed.

```
def mappable_function(path:str) ->List[str]:  
    result = tf.py_function(load_data, [path], (tf.float32, tf.int64))  
    return result
```

The `mappable_function` you provided is defined to wrap around a function that loads data (likely video frames and alignments) from a given file path. It uses `tf.py_function` to invoke a Python function within TensorFlow's graph.

Let's break down each part:

1. `mappable_function(path: str) -> List[str]`

- This defines a function `mappable_function`, which takes a `path` argument (likely a string) and returns a list of strings.
- The purpose of this function is to act as a wrapper for the `load_data` function.

2. `result = tf.py_function(load_data, [path], (tf.float32, tf.int64))`

- `tf.py_function`: This is a TensorFlow operation that allows you to execute a Python function within a TensorFlow computation graph. It allows you to use custom Python code that isn't directly supported by TensorFlow, but still lets you integrate it into TensorFlow operations.
 - **First argument (`load_data`):** This is the Python function to be called, which takes in the `path` and loads the video frames and alignments. The function itself is expected to return two things:
 - Video frames (likely a tensor of type `tf.float32`).
 - Alignments (likely a tensor of type `tf.int64`).

- **Second argument ([path]):** The list of arguments that will be passed to the `load_data` function. Here, it contains the path as input.
- **Third argument ((tf.float32, tf.int64)):** This specifies the expected output types of the function. In this case, `load_data` should return two tensors:
 - The first output (frames) should be of type `tf.float32` (likely normalized pixel values from video frames).
 - The second output (alignments) should be of type `tf.int64` (indices representing words or tokens from the alignment).

3. return result

- The result returned from `tf.py_function` is a tuple containing the two outputs from `load_data`, but wrapped in TensorFlow tensors.
- These tensors will have the specified types (`tf.float32`, `tf.int64`), and since the `mappable_function` is supposed to return a list of strings, you likely need to further process these outputs, especially for the alignments (converting `tf.int64` to a list of strings).

Creating Data Pipeline

```
from matplotlib import pyplot as plt
```

```
data = tf.data.Dataset.list_files('./data/s1/*.mpg')
data = data.shuffle(500, reshuffle_each_iteration=False)
data = data.map(mappable_function)
data = data.padded_batch(2, padded_shapes=([75, None, None, None], [40]))
data = data.prefetch(tf.data.AUTOTUNE)
# Added for split
train = data.take(450)
test = data.skip(450)
```

1. `data = tf.data.Dataset.list_files('./data/s1/*.mpg')`

- **Purpose:** This line creates a TensorFlow Dataset by listing all `.mpg` video files in the directory `./data/s1/`.
- **Explanation:**
 - `tf.data.Dataset.list_files` takes a file path pattern (in this case, `./data/s1/*.mpg`) and creates a dataset containing the file paths

matching the pattern.

- The resulting data object is a dataset of file paths (strings) to the .mpg video files.

2. `data = data.shuffle(500, reshuffle_each_iteration=False)`

- **Purpose:** This line shuffles the dataset to randomize the order of the video files.
- **Explanation:**
 - `shuffle(500)` specifies that a buffer of 500 elements will be maintained in memory, and elements will be randomly sampled from this buffer for shuffling. After consuming an element, the buffer is filled with a new element.
 - `reshuffle_each_iteration=False` means the dataset will not be reshuffled at the start of each new epoch. If set to `True`, the data would be reshuffled at the beginning of each iteration (epoch).

3. `data = data.map(mappable_function)`

- **Purpose:** This line applies a transformation function (`mappable_function`) to each element in the dataset.
- **Explanation:**
 - The `map` function allows you to apply any custom transformation to the dataset. Here, `mappable_function` is applied to each file path in the dataset.
 - `mappable_function` processes each file path, loading the video frames and alignments.

4. `data = data.padded_batch(2, padded_shapes=([75, None, None, None], [40]))`

- **Purpose:** This line batches the data into batches of size 2, and pads the sequences to ensure they have consistent shapes.
- **Explanation:**
 - `padded_batch(2)` creates batches of 2 elements (i.e., 2 examples from the dataset).
 - `padded_shapes=([75, None, None, None], [40])` specifies the shapes for padding:
 - For video frames (the first part of the tuple): `[75, None,`

None, None] implies that the video frames will have a shape of (75, height, width, channels), where None indicates a variable size for height and width. The number 75 could represent the fixed number of frames.

- For alignments (the second part of the tuple): [40] specifies that each alignment sequence will have a fixed length of 40 tokens.

- Padding will be applied to sequences so that each batch contains elements of consistent shape.

5. `data = data.prefetch(tf.data.AUTOTUNE)`

- **Purpose:** This line sets up prefetching to improve input pipeline performance.
- **Explanation:**
 - `prefetch(tf.data.AUTOTUNE)` allows TensorFlow to asynchronously load the next batch of data while the current batch is being processed by the model. This improves the overall throughput of the data pipeline.
 - AUTOTUNE automatically determines the optimal number of elements to prefetch based on system performance.

6. `train = data.take(450)`

- **Purpose:** This line creates the training dataset by taking the first 450 elements from the data pipeline.
- **Explanation:**
 - `take(450)` retrieves the first 450 batches from the dataset. These batches will be used for training.

7. `test = data.skip(450)`

- **Purpose:** This line creates the test dataset by skipping the first 450 elements in the dataset.
- **Explanation:**
 - `skip(450)` skips the first 450 batches, leaving the rest of the dataset (after those 450 batches) for testing.
 - This ensures that the training and test datasets are split appropriately.

Summary

This pipeline:

1. Loads `.mpg` files from the `./data/s1/` directory.
2. Shuffles the dataset with a buffer size of 500, without reshuffling between epochs.
3. Applies the `mappable_function` to each file path, which loads the video frames and alignments.
4. Padds the frames and alignments to consistent shapes and batches them into batches of 2.
5. Prefetches data for performance optimization.
6. Splits the dataset into a training set with the first 450 batches and a test set with the remaining batches.

This pipeline is well-suited for training a model on video data with corresponding alignment tokens.

GOT MY ERROR

```
frames, alignments = data.as_numpy_iterator().next()
```

```
-----
UnknownError                                Traceback (most recent call last)
<ipython-input-161-bc396cee879a> in <cell line: 1>()
----> 1 frames, alignments = data.as_numpy_iterator().next()

      ↕ 5 frames
/usr/local/lib/python3.10/dist-packages/tensorflow/python/framework/ops.py in raise_from_not_ok_status(e, name)
   5981 def raise_from_not_ok_status(e, name) -> NoReturn:
   5982     e.message += (" name: " + str(name if name is not None else ""))
-> 5983     raise core._status_to_exception(e) from None # pylint: disable=protected-access
   5984
   5985
UnknownError: {{function_node __wrapped__IteratorGetNext_output_types_2_device_/job:localhost/replica:0/task:0/device:CPU:0}} Error in user-defined function passed
to MapDataset:168 transformation with iterator: Iterator::Root::Prefetch::PaddedBatchV2::Map: FileNotFoundError: [Errno 2] No such file or directory:
'data/alignments/s1/.align'
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/dist-packages/tensorflow/python/ops/script_ops.py", line 268, in __call__
    return func(device, token, args)
  File "/usr/local/lib/python3.10/dist-packages/tensorflow/python/ops/script_ops.py", line 146, in __call__
    outputs = self._call(device, args)
  File "/usr/local/lib/python3.10/dist-packages/tensorflow/python/ops/script_ops.py", line 153, in _call
    ret = self._func(*args)
  File "/usr/local/lib/python3.10/dist-packages/tensorflow/python/autograph/impl/api.py", line 643, in wrapper
    return func(*args, **kwargs)
  File "<ipython-input-152-d5a94aec2d4f>", line 9, in load_data
    alignments = load_alignments(alignment_path)
  File "<ipython-input-151-51485e0ae5bf>", line 2, in load_alignments
    with open(path, 'r') as f:
FileNotFoundError: [Errno 2] No such file or directory: 'data/alignments/s1/.align'

[[{{node EagerPyFunc}}]] [Op:IteratorGetNext] name: |
```

Come on errors are part of life or i would say errors are life it makes you feel more connected with the code that you wrote(of course you wrote not gpt)

Let's Debug it!

The issue is in how the file name is being extracted. When you use `split('.')[0]` on a filename like `prii8p.mpg`, it's removing the entire filename, leaving an empty string. *OF course it didn't took me 3 hours and its 5 in the morning! I sometimes hate coding but in the end i love this let's see what else is about to come next!*

```
def load_data(path: str):
    path = bytes.decode(path.numpy())
    print("Original path:", path)

    # Corrected file name extraction
    file_name = path.split('/')[ -1].split('.')[0] # For Unix-like paths
    # Alternative for Windows: file_name =
    path.split('\\')[ -1].split('.')[0]
    print("Extracted file name:", file_name)

    # Video path
    video_path = os.path.join('data', 's1', f'{file_name}.mpg')
    print("Video path:", video_path)

    # Alignment path
    alignment_path =
    os.path.join('data', 'alignments', 's1', f'{file_name}.align')
    print("Alignment path:", alignment_path)

    # Check if files actually exist
    if not os.path.exists(video_path):
        print(f"Video file not found: {video_path}")
        raise FileNotFoundError(f"Video file not found: {video_path}")

    if not os.path.exists(alignment_path):
```

```

    print(f"Alignment file not found: {alignment_path}")
    raise FileNotFoundError(f"Alignment file not found:
{alignment_path}")

frames = load_video(video_path)
alignments = load_alignments(alignment_path)
return frames, alignments

```

Update it and let's move on

```

frames, alignments = data.as_numpy_iterator().next()
len(frames)
sample = data.as_numpy_iterator()
val = sample.next(); val[0]
imageio.mimsave('./animation.gif', val[0][0], fps=10)

```

1. frames, alignments = data.as_numpy_iterator().next()

- What it does: Converts the TensorFlow dataset data into a NumPy iterator and retrieves the first batch of data.
 - data.as_numpy_iterator() converts the dataset into a generator-like object that yields NumPy arrays.
 - .next() fetches the first batch of data from the iterator.
 - frames and alignments:
 - frames: A batch of video frames, stored as a NumPy array.
 - alignments: Corresponding alignment labels (e.g., lip-reading text or phoneme sequences).
-

2. len(frames)

- What it does: Calculates the number of video frames in the frames variable.
 - This is useful to understand the size of the input video batch.
 - Output: The total number of frames in the current video.
-

3. sample = data.as_numpy_iterator()

- What it does: Creates a reusable iterator over the dataset.
 - The sample variable stores the iterator for fetching batches of data

from data.

4. `val = sample.next(); val[0]`

- What it does:
 - `val = sample.next()`: Retrieves the next batch of data from the sample iterator.
 - `val` is a tuple containing two elements: (`frames_batch`, `alignments_batch`).
 - `val[0]`: Extracts the `frames_batch` (video frames) from the current batch.
-

5. `imageio.mimsave('./animation.gif', val[0][0], fps=10)`

- What it does:
 - `imageio.mimsave`: Saves a sequence of images as an animated GIF.
 - Parameters:
 - `./animation.gif`: Path to save the resulting animation.
 - `val[0][0]`: The first set of frames (i.e., the first video in the batch).
 - `fps=10`: Frames per second for the GIF animation.
 - Purpose: Creates a visual representation of the first video in the batch by animating its frames.
-

Step-by-Step Flow

1. Fetch Batch of Data: Retrieve video frames (`frames`) and alignments (`alignments`) using `data.as_numpy_iterator().next()`.
 2. Analyze Frames: Check the number of frames using `len(frames)`.
 3. Iterate Through Dataset: Use `sample.next()` to sequentially access batches of data.
 4. Create Animation: Save the first video of the batch as an animated GIF using `imageio.mimsave`.
-

Final Output

- GIF File: A file named animation.gif is created, showing the first video's frames at 10 frames per second.
- Practical Use: This visualization can be helpful for debugging or verifying that the video preprocessing pipeline is working correctly.

Design the Deep Neural Network

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv3D, LSTM, Dense, Dropout,
Bidirectional, MaxPool3D, Activation, Reshape, SpatialDropout3D,
BatchNormalization, TimeDistributed, Flatten
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ModelCheckpoint,
LearningRateScheduler
data.as_numpy_iterator().next()[0][0].shape
model = Sequential()
model.add(Conv3D(128, 3, input_shape=(75,46,140,1), padding='same'))
model.add(Activation('relu'))
model.add(MaxPool3D((1,2,2)))

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

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

model.add(TimeDistributed(Flatten()))

model.add(Bidirectional(LSTM(128, kernel_initializer='Orthogonal',
return_sequences=True)))
model.add(Dropout(.5))

model.add(Bidirectional(LSTM(128, kernel_initializer='Orthogonal',
return_sequences=True)))
model.add(Dropout(.5))

model.add(Dense(char_to_num.vocabulary_size()+1,
kernel_initializer='he_normal', activation='softmax'))
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
conv3d (Conv3D)	(None, 75, 46, 140, 128)	3584
activation (Activation)	(None, 75, 46, 140, 128)	0
max_pooling3d (MaxPooling3D)	(None, 75, 23, 70, 128)	0
conv3d_1 (Conv3D)	(None, 75, 23, 70, 256)	884992
activation_1 (Activation)	(None, 75, 23, 70, 256)	0
max_pooling3d_1 (MaxPooling3D)	(None, 75, 11, 35, 256)	0
conv3d_2 (Conv3D)	(None, 75, 11, 35, 75)	518475
activation_2 (Activation)	(None, 75, 11, 35, 75)	0
max_pooling3d_2 (MaxPooling3D)	(None, 75, 5, 17, 75)	0
time_distributed (TimeDistributed)	(None, 75, 6375)	0
bidirectional (Bidirectional)	(None, 75, 256)	6660096
dropout (Dropout)	(None, 75, 256)	0
bidirectional_1 (Bidirectional)	(None, 75, 256)	394240
dropout_1 (Dropout)	(None, 75, 256)	0
dense (Dense)	(None, 75, 41)	10537
=====		
Total params: 8,471,924		
Trainable params: 8,471,924		
Non-trainable params: 0		

Key Components

1. Imported Libraries:

- Sequential: For stacking layers sequentially.
- Conv3D: 3D convolutional layer, ideal for video data.
- LSTM, Bidirectional: For capturing temporal relationships.
- Dense: Fully connected layer for output prediction.
- Dropout: Regularization to prevent overfitting.
- MaxPool3D: Down-sampling 3D feature maps.
- Activation: For non-linear transformations.
- TimeDistributed: Applies layers to each time step in a sequence.
- Flatten: Flattens 3D tensors to 2D for dense layers.
- Adam: An optimizer.
- ModelCheckpoint, LearningRateScheduler: Callbacks for training.

2. Input Data Shape:

- (75, 46, 140, 1):
 - 75 frames in the video (time dimension).
 - 46x140 resolution for each frame (height x width).
 - 1 channel (grayscale).

Model Architecture

1. Input and First Convolution Layer:

- Conv3D(128, 3, input_shape=(75,46,140,1), padding='same')
 - A 3D convolution with 128 filters and a kernel size of 3.
 - Input shape is specified as (75, 46, 140, 1).
- Activation('relu'): Applies ReLU activation for non-linearity.
- MaxPool3D((1, 2, 2)): Reduces spatial dimensions (height and width) by pooling with a stride of 2 in spatial dimensions.

2. Second Convolution Layer:

- Conv3D(256, 3, padding='same'): A deeper convolution with 256 filters.
- MaxPool3D((1, 2, 2)): Further reduces spatial dimensions.

3. Third Convolution Layer:

- Conv3D(75, 3, padding='same'): A smaller convolution with 75

filters, which may align with the time dimension.

- `MaxPool3D((1, 2, 2))`: Down-samples further.

4. **TimeDistributed Flattening:**

- `TimeDistributed(Flatten())`: Flattens spatial dimensions at each time step, resulting in a 1D feature vector for each frame.

5. **Bidirectional LSTM Layers:**

- **Why LSTMs?** They capture temporal dependencies in sequential data.
- `Bidirectional(LSTM(128, kernel_initializer='Orthogonal', return_sequences=True))`:
 - Uses bidirectional LSTMs to capture temporal dependencies in both forward and backward directions.
 - `return_sequences=True`: Outputs the LSTM states for every time step.
 - `Dropout(.5)`: Adds dropout for regularization.
- Repeated for an additional layer.

6. **Final Dense Layer:**

- `Dense(char_to_num.vocabulary_size()+1, activation='softmax')`:
 - Fully connected layer with output size equal to the vocabulary size (`char_to_num.vocabulary_size()`) plus one (likely for a blank token for CTC loss).
 - `softmax`: Outputs probabilities for each character in the vocabulary.

Model Summary

- **Total Parameters:** 8,471,924
 - **Trainable Parameters:** 8,471,924 (all are trainable).
 - **Non-Trainable Parameters:** 0
-

Layer Outputs

1. Initial Conv3D: (75, 46, 140, 128)
2. After 1st MaxPool3D: (75, 23, 70, 128)
3. After 2nd Conv3D: (75, 23, 70, 256)

4. After 2nd MaxPool3D: (75, 11, 35, 256)
5. After 3rd Conv3D: (75, 11, 35, 75)
6. After 3rd MaxPool3D: (75, 5, 17, 75)
7. TimeDistributed Flatten: (75, 6375)
8. Bidirectional LSTM Layers: (75, 256)
9. Final Dense Layer: (75, 41)

```
yhat = model.predict(val[0])

tf.strings.reduce_join([num_to_char(x) for x in
tf.argmax(yhat[0],axis=1)])

tf.strings.reduce_join([num_to_char(tf.argmax(x)) for x
in yhat[0]])

model.input_shape

model.output_shape
```

Here's the explanation of the code you provided:

Making Predictions:

```
yhat = model.predict(val[0])
```

This line uses the model to make predictions on the input data `val[0]`. The model processes this input and generates an output, which is stored in `yhat`.

Processing Predictions:

```
tf.strings.reduce_join([num_to_char(x) for x in
tf.argmax(yhat[0], axis=1)])
```

`tf.argmax(yhat[0], axis=1)` gives the index of the maximum value in `yhat[0]` along the specified axis (`axis=1`). This essentially provides the predicted class for each timestep in the sequence.

`num_to_char(x)` converts the predicted class index into a corresponding character.

`tf.strings.reduce_join` then joins these characters together to form a single string.

Alternative Prediction Processing:

```
tf.strings.reduce_join([num_to_char(tf.argmax(x)) for x  
in yhat[0]])
```

This line processes each timestep's prediction in `yhat[0]`. For each timestep `x`, `tf.argmax(x)` finds the predicted class index, which is then converted into a character using `num_to_char`.

`tf.strings.reduce_join` joins all the characters to form a string.

Model Input and Output Shapes:

```
model.input_shape  
model.output_shape
```

`model.input_shape` returns the shape of the input that the model expects. For example, it could be `(None, 75, 46, 140, 1)`, indicating a batch of 3D video frames with specific dimensions.

`model.output_shape` returns the shape of the output that the model produces after processing the input. For instance, it could be `(None, 75, 41)`, which means the model predicts 41 classes for each of the 75 timesteps in the sequence.

In summary, this code makes predictions, processes those predictions into readable strings, and retrieves the model's input and output shapes to understand the expected data dimensions.

Setup Training Options and Train

```
def scheduler(epoch, lr):  
    if epoch < 30:  
        return lr  
    else:  
        return lr * tf.math.exp(-0.1)
```

This function is a learning rate scheduler, which adjusts the learning rate during the training process based on the current epoch. It is typically used to improve training efficiency and help the model converge more effectively.

Function Definition:

- epoch: The current training epoch (an integer value).
- lr: The current learning rate (a float value).

Condition (Epoch < 30):

```
if epoch < 30:  
    return lr
```

If the current epoch is less than 30, the function keeps the learning rate unchanged.

This allows the model to learn steadily with a constant learning rate during the initial training phase.

Condition (Epoch >= 30):

```
else:  
    return lr * tf.math.exp(-0.1)
```

Once the epoch reaches or exceeds 30, the learning rate is reduced exponentially.

`tf.math.exp(-0.1)` calculates the exponential decay factor (approximately 0.9048).

The new learning rate is set to the current learning rate (lr) multiplied by this decay factor, causing the learning rate to decrease gradually.

Why Use It?

- **Initial Stability:** A constant learning rate in the early epochs allows the model to find a good direction for optimization without drastic changes.
- **Fine-Tuning:** As the learning rate decreases in later epochs, the model performs smaller updates to weights, which helps refine the learned features and avoid overshooting the optimal solution.

Example:

Suppose the initial learning rate is 0.001:

- **Epoch 10:** Learning rate remains 0.001.
- **Epoch 30:** Learning rate transitions to exponential decay.
 - For the 30th epoch, the new learning rate would be:

$$0.001 \times e^{-0.1} \approx 0.0009048$$

- **Epoch 50:** Learning rate continues to decay further:
 - For the 50th epoch:

$$0.001 \times (e^{-0.1})^{20} \approx 0.000163$$

This gradual reduction in learning rate helps improve the model's performance by fine-tuning weights during later epochs.

```
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),
dtype="int64")
    label_length = label_length * tf.ones(shape=(batch_len, 1),
dtype="int64")

    loss = tf.keras.backend.ctc_batch_cost(y_true, y_pred, input_length,
label_length)
    return loss
```

The CTCLoss (Connectionist Temporal Classification Loss) function calculates the loss for sequence prediction tasks where the alignment between inputs (predictions) and outputs (ground truths) is unknown. It is commonly used for tasks like speech-to-text, OCR, or video captioning.

Code Breakdown:

```
def CTCLoss(y_true, y_pred):
```

- **y_true**: The true labels or sequences (ground truth) represented as numerical indices.
 - **y_pred**: The predicted outputs (logits) from the model. These are typically unaligned and require decoding.
-

1. Batch Length Calculation:

```
batch_len = tf.cast(tf.shape(y_true)[0], dtype="int64")
```

- `tf.shape(y_true)[0]`: Retrieves the number of samples (batch size) from the ground truth tensor.
 - `tf.cast(..., dtype="int64")`: Ensures the value is cast to the correct type (int64) for compatibility in subsequent operations.
-

2. Input Length Calculation:

```
input_length = tf.cast(tf.shape(y_pred)[1],  
dtype="int64")
```

- `tf.shape(y_pred)[1]`: Retrieves the time dimension (sequence length) of the predictions.
 - Casts the input length to int64.
-

3. Label Length Calculation:

```
label_length = tf.cast(tf.shape(y_true)[1],  
dtype="int64")
```

- `tf.shape(y_true)[1]`: Retrieves the sequence length of the ground truth labels.
 - Casts the label length to int64.
-

4. Repeat Input Lengths Across Batch:

```
input_length = input_length * tf.ones(shape=(batch_len, 1), dtype="int64")
```

- `tf.ones(shape=(batch_len, 1), dtype="int64")`: Creates a tensor of shape `(batch_len, 1)` filled with ones, where each row corresponds to a sample in the batch.
 - Multiplies the input length scalar by this tensor, resulting in a tensor where each sample in the batch has the same input length.
-

5. Repeat Label Lengths Across Batch:

```
label_length = label_length * tf.ones(shape=(batch_len, 1), dtype="int64")
```

- Similarly, creates a tensor of repeated label lengths for each sample in the batch.
-

6. Compute CTC Loss:

```
loss = tf.keras.backend.ctc_batch_cost(y_true, y_pred, input_length, label_length)
```

- `ctc_batch_cost(...)`: Computes the CTC loss for each sample in the batch.
 - **Inputs:**
 - `y_true`: The true labels.
 - `y_pred`: The predicted outputs.
 - `input_length`: The actual lengths of the predicted sequences.
 - `label_length`: The actual lengths of the ground truth sequences.
 - This function aligns the predictions to the labels and calculates the loss based on the probability of the correct label sequence.
-

7. Return Loss:

```
return loss
```

- The computed loss tensor (batch-wise loss values) is returned for use in model optimization.
-

Why Use This Function?

- The CTC loss is specifically designed for sequence tasks where the input and output lengths differ and explicit alignment is not provided.
 - It enables the model to predict sequences without requiring frame-by-frame labeled data.
-

```
class ProduceExample(tf.keras.callbacks.Callback):
    def __init__(self, dataset) -> None:
        self.dataset = dataset.as_numpy_iterator()

    def on_epoch_end(self, epoch, logs=None) -> None:
        data = self.dataset.next()
        yhat = self.model.predict(data[0])
        decoded = tf.keras.backend.ctc_decode(yhat, [75,75],
greedy=False)[0][0].numpy()
        for x in range(len(yhat)):
            print('Original:',
tf.strings.reduce_join(num_to_char(data[1][x])).numpy().decode('utf-8'))
            print('Prediction:',
tf.strings.reduce_join(num_to_char(decoded[x])).numpy().decode('utf-8'))
            print('~'*100)
```

Here is a detailed explanation of the ProduceExample class:

Purpose:

The ProduceExample class is a custom callback in TensorFlow/Keras that:

1. Generates and prints examples of model predictions at the end of each training epoch.

2. Allows for real-time monitoring of the model's performance by comparing predictions with ground truths.

This is particularly useful in tasks like sequence prediction, where decoding and understanding predictions is critical (e.g., OCR or speech-to-text).

Code Breakdown:

1. Class Definition:

```
class ProduceExample(tf.keras.callbacks.Callback):
```

- Inherits from `tf.keras.callbacks.Callback`, a base class provided by Keras for customizing model training behavior.
 - You can use this to define custom actions like logging, saving models, or evaluating performance during training.
-

2. Initialization (`__init__` Method):

```
def __init__(self, dataset) -> None:  
    self.dataset = dataset.as_numpy_iterator()
```

- **dataset:** A TensorFlow dataset (likely the test or validation dataset) is passed as input.
- **as_numpy_iterator():** Converts the dataset into a NumPy iterator, allowing it to be iterated using `.next()`.

This ensures easy access to individual batches of data during training.

3. Action at the End of Each Epoch (`on_epoch_end` Method):

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

- This method is called automatically at the end of each epoch during training.
- **epoch:** The current epoch number.

- **logs**: Dictionary of logs containing training/validation metrics (e.g., loss, accuracy).
-

4. Fetch a Batch of Data:

```
data = self.dataset.next()
```

- Fetches the next batch from the dataset.
 - **data[0]**: Input features (e.g., images or sequences).
 - **data[1]**: Ground truth labels for the inputs.
-

5. Model Predictions:

```
yhat = self.model.predict(data[0])
```

- **self.model**: Refers to the model being trained.
 - **predict(data[0])**: Generates predictions for the input features in the batch.
-

6. Decode Predictions:

```
decoded = tf.keras.backend.ctc_decode(yhat, [75,75],  
greedy=False)[0][0].numpy()
```

- **ctc_decode**: Decodes the predicted sequences into readable text using the CTC decoding algorithm.
 - **yhat**: The predicted logits.
 - **[75, 75]**: Specifies the sequence lengths for each prediction (batch size = 2 in this example).
 - **greedy=False**: Uses a beam search decoder (better accuracy than greedy decoding).
 - **[0][0].numpy()**: Retrieves the decoded sequences as a NumPy array.
-

7. Iterate Over Batch and Print Results:

```
for x in range(len(yhat)):
    print('Original:',
          tf.strings.reduce_join(num_to_char(data[1][x])).numpy().d
          ecode('utf-8'))
    print('Prediction:',
          tf.strings.reduce_join(num_to_char(decoded[x])).numpy().d
          ecode('utf-8'))
    print('~'*100)
```

- **Loop through the batch:**
 - x iterates over all samples in the batch.
 - **Ground Truth (Original):**
 - data[1][x]: Access the true sequence.
 - num_to_char(data[1][x]): Maps the numerical labels to characters.
 - tf.strings.reduce_join(...): Concatenates characters into a single string.
 - .numpy().decode('utf-8'): Converts the tensor into a Python-readable UTF-8 string.
 - **Prediction (Prediction):**
 - Similar decoding process is applied to the model's predictions (decoded[x]).
 - **Print Results:**
 - Prints the original and predicted sequences.
 - ~*100: Prints a separator for better readability.
-

Key Features:

1. **Real-Time Feedback:** Provides a snapshot of how well the model is learning at each epoch.
2. **Comparison:** Shows ground truth and predictions side by side for easy assessment.
3. **Debugging:** Helps identify common errors or mismatches in predictions.
4. **Custom Decoding:** Uses CTC decoding, making it suitable for tasks with unaligned input-output sequence

```
model.compile(optimizer=Adam(learning_rate=0.0001), loss=CTCLoss)
```

configures the model for training by specifying:

1. **The optimizer:** Controls how the model updates its weights to minimize the loss function.
2. **The loss function:** Measures the difference between the model's predictions and the ground truth.

1. model.compile():

- This is a Keras method to configure the model for training.
 - It accepts parameters like:
 - **Optimizer:** The algorithm used to update model weights.
 - **Loss Function:** A function used to measure the error during training.
 - **Metrics:** (Optional) Additional metrics to evaluate the model's performance (not used here).
-

2. optimizer=Adam(learning_rate=0.0001):

- **Adam:** Stands for *Adaptive Moment Estimation*, a widely used optimizer in deep learning.
 - **Learning Rate:** Controls the step size for weight updates during training. A small value (0.0001) ensures that the model trains slowly and avoids overshooting the minimum of the loss function.
 - Adam combines the advantages of two other optimizers:
 1. **Momentum:** Uses past gradients to smooth updates.
 2. **RMSProp:** Scales updates based on recent gradient magnitudes.
-

3. loss=CTCLoss:

- The **loss function** used here is CTCLoss, which stands for *Connectionist Temporal Classification Loss*.
- **Purpose:**
 - Designed for sequence prediction tasks where the input and output

sequences are not aligned.

- Commonly used in speech-to-text, OCR, or handwriting recognition tasks.
 - **How It Works:**
 - Penalizes the model for incorrect predictions by comparing the predicted sequences (`y_pred`) with the ground truth sequences (`y_true`).
 - Accounts for sequence alignment issues (e.g., missing characters or extra characters in predictions).
-

Summary:

This line configures the model for training with:

1. **Adam optimizer:** Ensures efficient and adaptive weight updates.
2. **CTC loss:** Handles sequence prediction tasks where input-output alignments are not strict.

By specifying a learning rate of 0.0001, the model will train gradually, reducing the likelihood of instability or overshooting during training.

```
checkpoint_callback = ModelCheckpoint(os.path.join('models', 'checkpoint'),  
monitor='loss', save_weights_only=True)
```

This code creates a **callback** that saves the model's weights during training at specific intervals (usually at the end of each epoch). It is especially useful to:

1. Save intermediate training progress.
2. Resume training in case of interruptions.
3. Prevent loss of model weights after training.

1. ModelCheckpoint:

- A built-in Keras callback used to save the model during training.
- Parameters:
 - **filepath:** Specifies where and how to save the model weights.
 - **monitor:** Metric to monitor for saving the checkpoint (e.g., loss,

accuracy).

- **save_weights_only**: Whether to save just the weights or the entire model (weights + architecture).
- Other options (not used here):
 - **save_best_only**: Save only the best weights based on the monitored metric.
 - **mode**: Specifies whether to look for the minimum or maximum value of the monitored metric.

2. **os.path.join('models', 'checkpoint')**:

- Constructs a file path to save the checkpoints.
- **os.path.join**:
 - Combines directory paths ('models') and filenames ('checkpoint').
 - Ensures compatibility across different operating systems.
- Here, the checkpoints will be saved in the models directory with the filename checkpoint.

3. **monitor='loss'**:

- Specifies that the **loss** metric is monitored during training.
- The callback will save the model's weights after each epoch, regardless of whether the loss improves (unless **save_best_only=True** is set).

4. **save_weights_only=True**:

- Indicates that **only the model weights** should be saved, not the full model structure.
- Advantage:
 - Saves storage space.
 - You can load the weights later into a model with the same architecture.

Example Scenario:

- **During Training**: After each epoch, this callback saves the model's weights into the file models/checkpoint.
- **Use Case**: If training is interrupted, you can reload the saved weights into

the same model and continue training without starting over.

Summary:

This line defines a `ModelCheckpoint` callback that saves the model's weights after every epoch. It monitors the **loss** metric and stores the weights in the models directory under the file named `checkpoint`. By saving only the weights, it ensures efficient storage and quick recovery of training progress.

```
schedule_callback = LearningRateScheduler(scheduler)
```

This code creates a **callback** that adjusts the learning rate dynamically during training using the **scheduler** function. This helps optimize the model's training by varying the learning rate based on the epoch.

1. LearningRateScheduler:

- A Keras callback that updates the learning rate of the optimizer during training.
- Parameters:
 - **schedule**: A function that defines how the learning rate should change over epochs. This function is executed at the start of every epoch.
 - **verbose**: (optional) If set to 1, it prints the updated learning rate at each epoch.

2. scheduler:

- The function provided to the `schedule` parameter.

The scheduler function in your code is defined as:

```
def scheduler(epoch, lr):  
    if epoch < 30:  
        return lr  
    else:  
        return lr * tf.math.exp(-0.1)
```

- **Purpose:**
 - For the first 30 epochs: Keeps the learning rate constant.
 - After 30 epochs: Gradually decreases the learning rate exponentially by multiplying the current learning rate with $e^{-0.1}$.

3. `schedule_callback`:

- A variable that holds the `LearningRateScheduler` instance.
 - This callback is later passed to the `model.fit()` method to apply the learning rate schedule during training.
-

Why Use a Learning Rate Scheduler?

- **Faster Convergence:** A high learning rate in early epochs accelerates progress.
 - **Stability:** A lower learning rate in later epochs prevents overshooting the optimal point.
 - **Avoids Overfitting:** Gradual reduction in learning rate helps refine the model's parameters and achieve better generalization.
-

Example Scenario:

- **During Training:**
 - In epochs 0–29: The learning rate remains constant (e.g., 0.001 if the initial learning rate is set to this value).
 - From epoch 30 onwards: The learning rate decays exponentially (e.g., $lr = lr \times e^{-0.1}$).
-

Summary:

The `schedule_callback` adjusts the optimizer's learning rate dynamically based on the scheduler function during training. This ensures an effective training process by maintaining a high learning rate initially and reducing it in later epochs for better fine-tuning.

```
example_callback = ProduceExample(test)
```

This code creates an instance of the **ProduceExample** class, which is a custom callback defined earlier. This callback generates predictions and compares them to the original labels after each epoch during model training. It's primarily used for monitoring the model's performance visually on the provided test dataset.

1. ProduceExample:

- This is a custom callback class inheriting from `tf.keras.callbacks.Callback`.

Defined as:

```
class ProduceExample(tf.keras.callbacks.Callback):
    def __init__(self, dataset) -> None:
        self.dataset = dataset.as_numpy_iterator()

    def on_epoch_end(self, epoch, logs=None) -> None:
        data = self.dataset.next()
        yhat = self.model.predict(data[0])
        decoded = tf.keras.backend.ctc_decode(yhat, [75,75],
greedy=False)[0][0].numpy()
        for x in range(len(yhat)):
            print('Original:',
tf.strings.reduce_join(num_to_char(data[1][x])).numpy().decode('utf-8'))
            print('Prediction:',
tf.strings.reduce_join(num_to_char(decoded[x])).numpy().decode('utf-8'))
            print('~'*100)
```

- **Functionality:**
 - At the end of each training epoch (`on_epoch_end` method):
 - Fetches a batch of data from the provided dataset.
 - Runs the model's prediction on the input data.
 - Decodes the predictions (likely using Connectionist Temporal Classification, or CTC).
 - Prints the original labels and predicted outputs for visual comparison.

2. test:

- This is the dataset passed to the callback. It contains data the model has not seen during training.
- Likely prepared using TensorFlow's `tf.data` API or a similar method.
- Example content of test:
 - **Input data:** A batch of feature arrays (e.g., images, video frames).
 - **Labels:** The true sequences/outputs corresponding to the inputs.

3. example_callback:

- The variable holding an instance of `ProduceExample`, initialized with the test dataset.
 - This instance will later be passed to the `model.fit()` method to execute its functionality during training.
-

Why Use `ProduceExample`?

- **Monitor Training Progress:**
 - Outputs predictions and their corresponding true labels for the test dataset.
 - Allows real-time evaluation of how well the model is learning.
 - **Debugging:**
 - If predictions don't improve or align with labels, adjustments can be made to the model or training process.
-

Summary:

The `example_callback` is a custom callback instance that fetches data from the test dataset, performs predictions using the model, decodes the predictions, and prints them alongside the true labels at the end of each epoch. It provides an intuitive way to monitor model progress and ensure it learns as expected during training.

```
model.fit(train, validation_data=test, epochs=100,  
callbacks=[checkpoint_callback, schedule_callback, example_callback])
```

The code trains the **model** using the **train** dataset, validates it against the **test** dataset, and applies specified **callbacks** to monitor and modify the training process. It runs for 100 epochs.

1. **model.fit():**

- This method trains the Keras model.
- Key arguments:
 - **train**: The training dataset, typically prepared using TensorFlow's `tf.data` API or another data preprocessing pipeline.
 - **validation_data=test**: A separate dataset used to evaluate the model's performance at the end of each epoch. Helps detect overfitting or underfitting.
 - **epochs=100**: The number of complete passes through the training dataset during training.
 - **callbacks**: A list of callback objects for monitoring and managing training.

2. **train:**

- The input dataset contains features (e.g., images, sequences) and their corresponding labels.
- Typically preprocessed (e.g., scaled, augmented) and batched.

3. **validation_data=test:**

- The **test** dataset serves as a validation set.
- Helps evaluate the model's performance on unseen data after each epoch.

4. **epochs=100:**

- The training process will iterate through the entire **train** dataset 100 times.

5. **callbacks:**

- A list of callback objects that are executed during training at specific points. These callbacks can save models, adjust learning rates, or monitor performance.
 - **Checkpoint_callback:**
 - Saves the model's weights at the end of each epoch if the monitored metric (e.g., loss) improves.

Code used to define it:

```
checkpoint_callback = ModelCheckpoint(os.path.join('models',  
'checkpoint'), monitor='loss', save_weights_only=True)
```

- **Schedule_callback:**
 - Adjusts the learning rate dynamically based on the epoch number.
 - Uses the custom scheduler function to return a modified learning rate after each epoch.

Code used to define it:

```
schedule_callback = LearningRateScheduler(scheduler)
```

- **example_callback:**
 - Monitors predictions visually by comparing the model's predictions with the true labels at the end of each epoch.
 - Custom callback that provides debugging and monitoring for the model's learning process.

Code used to define it:

```
example_callback = ProduceExample(test)
```

Key Points:

- **Training Process:**
 - The model iterates over the **train** dataset and computes gradients to

optimize its parameters.

- After each epoch, the model evaluates its performance on the **test** dataset.

- **Callbacks in Action:**

- The `checkpoint_callback` saves the model's progress based on its performance.
 - The `schedule_callback` adjusts the learning rate dynamically for improved convergence.
 - The `example_callback` outputs sample predictions for manual inspection of training progress.
-

Summary:

This line trains the model for 100 epochs using the train dataset while validating on the test dataset. The training process is augmented with callbacks to:

1. Save the best model weights (`checkpoint_callback`),
2. Dynamically adjust the learning rate (`schedule_callback`), and
3. Print predictions for debugging and monitoring (`example_callback`).

Make a Prediction

```
url = 'https://drive.google.com/uc?id=1vWscXs4Vt0a_1IH1-ct2TCgXAZT-N3_Y'  
output = 'checkpoints.zip'  
gdown.download(url, output, quiet=False)  
gdown.extractall('checkpoints.zip', 'models')
```

1. `url = 'https://drive.google.com/uc?id=1vWscXs4Vt0a_1IH1-ct2TCgXAZT-N3_Y':`

- This defines the URL of the file to be downloaded.
- The URL points to a file hosted on Google Drive.
 - `uc?id=1vWscXs4Vt0a_1IH1-ct2TCgXAZT-N3_Y`: The unique identifier for the file on Google Drive.

2. `output = 'checkpoints.zip':`

- This specifies the name of the file where the downloaded data will be

saved locally.

- The file will be saved as **checkpoints.zip**.

3. **gdown.download(url, output, quiet=False):**

- **gdown**: A Python library that simplifies downloading files from Google Drive.
- **gdown.download()**:
 - **url**: The URL of the file to download (provided above).
 - **output**: The name of the output file where the downloaded content will be saved.
 - **quiet=False**: Ensures progress is displayed in the console during the download.

4. **gdown.extractall('checkpoints.zip', 'models'):**

- **gdown.extractall()**:
 - Extracts the contents of the zip file **checkpoints.zip**.
 - **First argument ('checkpoints.zip')**:
 - Path to the zip file to be extracted.
 - **Second argument ('models')**:
 - The destination folder where the extracted files will be placed.
 - In this case, the contents of the zip file will be extracted into a folder named **models**.

Workflow:

1. **Download:**

- The file identified by the URL is downloaded from Google Drive.
- It is saved locally as **checkpoints.zip**.

2. **Extract:**

- The downloaded zip file is extracted into the **models** directory.
 - This unzipping step is crucial for accessing the contents of the file.
-

```
model.load_weights('models/checkpoint')
```

The line `model.load_weights('models/checkpoint')` loads the saved weights of a pre-trained model. The weights are loaded from the specified path 'models/checkpoint'. This allows the model to resume training or perform inference using the saved parameters. The weights are stored in the checkpoint file, which contains the learned parameters from a previous training session.

```
test_data = test.as_numpy_iterator()
```

The line `test_data = test.as_numpy_iterator()` converts the test dataset into a NumPy iterator. This allows for iterating through the test dataset in a format that returns NumPy arrays, making it easier to handle during model evaluation or inference. By using an iterator, the dataset can be processed in batches one at a time, which is memory efficient. It facilitates accessing and using the data in a sequential manner for operations like prediction or evaluation.

```
sample = test_data.next()
```

The line `sample = test_data.next()` retrieves the next batch of data from the `test_data` iterator. This means it fetches the next element from the test dataset as a NumPy array. The `next()` function advances the iterator and returns the next available sample, which is typically a tuple containing the input features and the corresponding labels. This is useful for processing and making predictions on the test dataset during model evaluation or inference.

```
yhat = model.predict(sample[0])
```

The line `yhat = model.predict(sample[0])` uses the trained model to make predictions on the input data from the sample. Here's a breakdown:

1. `sample[0]`: This refers to the input features (data) of the current sample from the `test_data`. Typically, `sample[0]` contains the features (like images or sequences) that the model will predict.
2. `model.predict()`: This method is used to generate predictions based on the input data. It passes `sample[0]` through the model and outputs predicted

values.

3. `yhat`: This stores the model's predictions for the input sample. These predictions can then be processed further, such as decoding or evaluating the model's performance.

```
print('~'*100, 'REAL TEXT')
[tf.strings.reduce_join([num_to_char(word) for word in sentence]) for
 sentence in sample[1]]
```

The code you provided prints a line of 100 tilde characters followed by the label "REAL TEXT" and then attempts to transform the labels from the `sample[1]` (presumably the true labels of the dataset) into human-readable text. Here's a detailed breakdown:

1. `print('~'*100, 'REAL TEXT')`:
 - This prints a separator line (~ repeated 100 times) followed by the text "REAL TEXT". It's used as a visual marker to separate outputs for clarity.
2. `[tf.strings.reduce_join([num_to_char(word) for word in sentence]) for sentence in sample[1]]`:
 - This line is a list comprehension that operates on each sentence in `sample[1]` (the true labels or ground truth). Each sentence is transformed by:
 - `[num_to_char(word) for word in sentence]`: This converts each word (which is assumed to be represented as numeric values in the dataset) into characters using the `num_to_char` function.
 - `tf.strings.reduce_join(...)`: Joins the characters into a single string (sentence) for each sentence in `sample[1]`.
 - The result is a list of decoded sentences.
3. The list comprehension returns a list of human-readable text sentences from the ground truth labels, which is printed following the separator.

```
decoded = tf.keras.backend.ctc_decode(yhat, input_length=[75,75],
greedy=True)[0][0].numpy()
```

The code `decoded = tf.keras.backend.ctc_decode(yhat, input_length=[75,75], greedy=True)[0][0].numpy()` performs the following operations:

1. `tf.keras.backend.ctc_decode(yhat, input_length=[75,75], greedy=True)`:
 - This function decodes the output of a CTC (Connectionist Temporal Classification) model.
 - `yhat`: This is the model's predicted output, typically a probability distribution over characters for each timestep in the sequence.
 - `input_length=[75,75]`: The length of the input sequences in the batch. It's specified as `[75,75]`, meaning the input sequences for both items in the batch are of length 75.
 - `greedy=True`: This argument specifies that the greedy decoding algorithm is used, meaning the most probable character at each timestep is chosen (as opposed to beam search or other decoding methods).
 - The `ctc_decode` function returns a list of decoded sequences. Each sequence represents the decoded output of the model for each input sequence.
2. `[0][0]`:
 - Since `ctc_decode` returns a list of lists, `[0][0]` accesses the first sequence in the batch and the first decoded sentence within that sequence.
3. `.numpy()`:
 - Converts the TensorFlow tensor to a NumPy array, allowing further operations or manipulation in NumPy format (or for easier inspection).

The result stored in `decoded` is the decoded output of the CTC model for the batch's first sequence, which represents the most likely characters predicted by the model for that sequence.

```
print('~'*100, 'PREDICTIONS')
[tf.strings.reduce_join([num_to_char(word) for word in sentence]) for
```

```
sentence in decoded]
```

The line of code `print('~'*100, 'PREDICTIONS')` followed by `[tf.strings.reduce_join([num_to_char(word) for word in sentence]) for sentence in decoded]` performs the following steps:

1. `print('~'*100, 'PREDICTIONS')`:
 - This prints a separator line of 100 ~ characters followed by the label 'PREDICTIONS' to the console. It serves as a visual marker to indicate the start of the predicted outputs.
2. `[tf.strings.reduce_join([num_to_char(word) for word in sentence]) for sentence in decoded]`:
 - This is a list comprehension that processes each sentence in decoded.
 - decoded: It is assumed to be a list of decoded sequences, where each sequence is a list of integer indices representing the predicted characters (as numbers).
3. `[num_to_char(word) for word in sentence]`:
 - For each sentence (which is a list of predicted character indices), it converts each index (word) to its corresponding character using the `num_to_char` function. This transforms the sequence of numbers into a sequence of characters.
4. `tf.strings.reduce_join(...)`:
 - `tf.strings.reduce_join` takes the list of characters for a sentence and joins them into a single string (removing spaces between them).
 - It effectively converts a list of characters into a continuous string representing the decoded sentence.
5. **The entire list comprehension:**
 - It applies the `reduce_join` operation to each sentence decoded to create the complete predicted text for each input sequence.

The result is a list of decoded text predictions, where each prediction is a string of characters representing the model's output for each input sequence. This is printed after the separator line.

Test on a Video

```
sample = load_data(tf.convert_to_tensor('./data/s1/bras9a.mpg'))
print('~'*100, 'REAL TEXT')
[tf.strings.reduce_join([num_to_char(word) for word in sentence]) for
sentence in [sample[1]]]
yhat = model.predict(tf.expand_dims(sample[0], axis=0))
decoded = tf.keras.backend.ctc_decode(yhat, input_length=[75],
greedy=True)[0][0].numpy()
print('~'*100, 'PREDICTIONS')
[tf.strings.reduce_join([num_to_char(word) for word in sentence]) for
sentence in decoded]
```

Here's the explanation for the code snippet you provided:

1. Loading the video sample:

```
sample = load_data(tf.convert_to_tensor('./data/s1/bras9a.mpg'))
```

- The `load_data` function loads the video file `bras9a.mpg` located at `./data/s1/` and converts the path to a tensor using `tf.convert_to_tensor()`. It assumes the function processes the video and returns a sample, typically containing both the video frames and the corresponding ground truth text (labels).

2. Printing the real text:

```
print('~'*100, 'REAL TEXT')
[tf.strings.reduce_join([num_to_char(word) for word in
sentence]) for sentence in [sample[1]]]
```

- This prints the label (the real transcription or ground truth) for the video sequence.
- `sample[1]` refers to the real text in the sample, which is converted to characters using `num_to_char`. The `reduce_join` function then joins the list of characters into a string, representing the real text.
- The result is the actual text corresponding to the video, displayed after a separator line of 100 ~ characters.

3. Making predictions:

```
yhat = model.predict(tf.expand_dims(sample[0], axis=0))
```

- `sample[0]` contains the video frames (input features), and `tf.expand_dims` adds an extra dimension to match the model's expected input shape.
- `model.predict()` runs inference on the input sample, generating the predicted output for the video, which is stored in `yhat`.

4. Decoding the predictions using CTC (Connectionist Temporal Classification):

```
decoded = tf.keras.backend.ctc_decode(yhat,  
input_length=[75], greedy=True)[0][0].numpy()
```

- `ctc_decode` is used to decode the model's output (`yhat`) into actual text predictions using the CTC loss function.
- `input_length=[75]` indicates the sequence length for the input (likely based on the length of the video frames).
- `greedy=True` ensures that the decoding is done using the greedy approach, where the most probable character is chosen at each timestep.
- The `[0][0]` accesses the first decoding result (as `ctc_decode` returns a list of decoded sequences).
- The `.numpy()` converts the tensor to a NumPy array.

5. Printing the predictions:

```
print('~'*100, 'PREDICTIONS')  
[tf.strings.reduce_join([num_to_char(word) for word in  
sentence]) for sentence in decoded]
```

- Prints a separator line followed by the predicted text.
- Each decoded sequence (`sentence in decoded`) is converted from indices to characters using `num_to_char`, and `reduce_join` joins them into a single string, representing the predicted transcription.
- The result is the predicted transcription of the video, shown as a text string.

Output:

PREDICTIONS

```
[<tf.Tensor: shape=(), dtype=string, numpy=b'bin red at s nine again'>]
```

- The model predicts the transcription as "bin red at s nine again", which is the output of the video sample bras9a.mpg.

This demonstrates a full cycle of loading a video, making predictions, and decoding the results using a CTC-based model for video captioning or speech recognition.

Thank you for reading !