fingerspelling_v2

May 13, 2024

[ ]:
```python
import pandas as pd
import pyarrow.parquet as pq
import math
import torch
import torch.nn as nn
import json
import torch._dynamo
from torchdata.dataloader2 import DataLoader2, MultiProcessingReadingService
import matplotlib.pyplot as plt
from torchdata.datapipes.iter import (
    FileLister,
    FileOpener,
    TFRecordLoader,
    Mapper,
    Batcher,
    Collator,
    Shuffler,
)
import torch.nn.functional as F
import multiprocessing as mp
import tensorflow as tf
import numpy as np
from tqdm.notebook import (
    tqdm_notebook,
)  # Assuming you only need one tqdm implementation
from sklearn.preprocessing import StandardScaler
from scipy.interpolate import interp1d
import pytorch_lightning as pl
from pytorch_lightning import Trainer
from pytorch_lightning.loggers import TensorBoardLogger
from pytorch_lightning.callbacks import (
    ModelCheckpoint,
    LearningRateMonitor,
    EarlyStopping,
    RichModelSummary,
)
from torchmetrics.text import EditDistance
```

```python
# Set the default matmul precision to medium, or high/highest?
torch.backends.cuda.matmul.allow_tf32 = True
torch.backends.cudnn.allow_tf32 = True
torch.set_float32_matmul_precision("medium")
# Read the first CSV file
dataset_train_df = pd.read_csv("train.csv")

# Read the second CSV file
dataset_supplemental_df = pd.read_csv("supplemental_metadata.csv")

# Concatenate the two dataframes
dataset_df = pd.concat([dataset_train_df, dataset_supplemental_df],␣
↪ignore_index=True)

# Save the combined CSV file
# dataset_df.to_csv("train_full.csv", index=False)
dataset_df = dataset_df
```

[ ]:
```python
# Read the first row of the DataFrame
path, sequence_id, file_id, phrase = dataset_df.iloc[0][
    ["path", "sequence_id", "file_id", "phrase"]
]
print(f"path: {path}, sequence_id: {sequence_id}, file_id: {file_id}, phrase:␣
↪{phrase}")

sample_sequence_df = pq.read_table(
    f"{str(path)}",
    filters=[
        [("sequence_id", "=", sequence_id)],
    ],
).to_pandas()
print("Full sequence dataset shape is {}".format(sample_sequence_df.shape))

# Calculate the length of each phrase by characters
df['phrase_length_chars'] = df['phrase'].apply(len)

# Create a histogram
ax = df['phrase_length_chars'].plot.hist(bins=30, color='grey', alpha=0.7)
plt.title('Distribution of Phrase Lengths in Characters')
plt.xlabel('Length of Phrases (characters)')
plt.ylabel('Frequency')
plt.grid(True)

# Calculate and display mean and median
mean_val = df['phrase_length_chars'].mean()
```

```
median_val = df['phrase_length_chars'].median()
plt.axvline(mean_val, color='k', linestyle='dashed', linewidth=1)
plt.axvline(median_val, color='c', linestyle='dashed', linewidth=1)
plt.legend({'Mean':mean_val, 'Median':median_val})

plt.text(mean_val + 10, 1000, f'Mean: {mean_val:.2f}', rotation=0)
plt.text(median_val + 10, 45, f'  Median: {median_val:.2f}', rotation=0)

plt.show()
```

[ ]:

```
import os

def process_directory(directory):
  data = []
  for filename in os.listdir(directory):
    if filename.endswith(".parquet"):
      filepath = os.path.join(directory, filename)
      # Read the Parquet file
      df = pd.read_parquet(filepath)
      # Group by sequence_id and count the number of frames
      grouped_data = df.groupby('sequence_id')['frame'].count().reset_index()
      # Append the group data to the main list
      data.append(grouped_data)
  # Combine all group data into a single DataFrame
  result_df = pd.concat(data, ignore_index=True)
  return result_df

directory = "/home/jpinn/asl-fingerspelling-recognition/src/train_landmarks"

# Process the directory and get the DataFrame
result_df = process_directory(directory)
```

[ ]:

```
# Create a histogram
plt.figure(figsize=(8, 6))  # Adjust figure size as needed
plt.hist(result_df['frame'], color='grey', bins=100)  # Adjust the number of␣
 ↪bins as needed
plt.xlabel('Number of Frames')
plt.ylabel('Number of Sequences')
plt.title('Distribution of Frames per Sequence')
plt.grid(True)  # Add gridlines for better readability

# Calculate and display mean and median
mean_val = result_df['frame'].mean()
median_val = result_df['frame'].median()
plt.axvline(mean_val, color='k', linestyle='dashed', linewidth=1)
plt.axvline(median_val, color='c', linestyle='dashed', linewidth=1)
plt.legend({'Mean':mean_val, 'Median':median_val})
```

```
plt.text(mean_val + 150, 1000, f'Mean: {mean_val:.2f}', rotation=0)
plt.text(median_val + 200, 500, f'  Median: {median_val:.2f}', rotation=0)

plt.tight_layout()
plt.show()
```

[ ]:

```
import pandas as pd
import matplotlib.pyplot as plt

# Function to load and concatenate CSV files
def load_and_concatenate_csv(file_paths):
    dataframes = []
    step_offset = 0  # Initialize step offset
    for file_path in file_paths:
        # Use delimiter='\t' to specify that the values are separated by tabs
        df = pd.read_csv(file_path)
        print(df.columns)  # This will print the column names to check them

        # Ensure 'Step' column exists
        if 'Step' not in df.columns:
            raise ValueError(f"Column 'Step' not found in {file_path}. Columns␣
↪found: {df.columns}")

        if step_offset > 0:  # Adjust 'Step' if it's not the first file
            df['Step'] += step_offset

        dataframes.append(df)
        step_offset = df['Step'].iloc[-1]  # Update step offset to the last␣
↪step of the current df

    concatenated_df = pd.concat(dataframes)
    return concatenated_df

# File paths for train and validation CSV files
train_csv_files = ['run-transformer_version_218-tag-train_loss_epoch.csv',␣
↪'run-transformer_version_220-tag-train_loss_epoch.csv']
val_csv_files = ['run-transformer_version_218-tag-val_loss_epoch.csv',␣
↪'run-transformer_version_220-tag-val_loss_epoch.csv']

# Load and concatenate data
train_data = load_and_concatenate_csv(train_csv_files)
val_data = load_and_concatenate_csv(val_csv_files)

# Plotting the training and validation loss
plt.figure(figsize=(10, 5))
plt.plot(train_data['Step'], train_data['Value'], label='Train Loss')
```

```
plt.plot(val_data['Step'], val_data['Value'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Step')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

[ ]:
```
import pandas as pd

def parse_line(line):
    # Split each part of the line
    parts = line.split(', ')
    predicted = parts[0].split(': ')[1]
    target = parts[1].split(': ')[1]
    edit_distance = float(parts[2].split(': ')[1])
    return {'Predicted': predicted, 'Target': target, 'Edit Distance':␣
 ↪edit_distance}

def read_data(filepath):
    data_list = []
    with open(filepath, 'r') as file:
        for line in file:
            if line.strip():
                data_list.append(parse_line(line))
    return pd.DataFrame(data_list)

# Replace 'data.txt' with the path to your text file
df = read_data('edit_dists.txt')

total_edit_distance = df['Edit Distance'].mean()
print(total_edit_distance)

mean_target_length = df['Target'].apply(len).mean()
print(mean_target_length)
```

[ ]:
```
# Read the total amount unique files
unique_paths = dataset_df["path"].unique()

sum = unique_paths.shape[0]

print("Total number of files: {}".format(sum))
```

[ ]:
```
LIP = [
    61,
    185,
    40,
```

```
    39,
    37,
    267,
    269,
    270,
    409,
    291,
    146,
    91,
    181,
    84,
    17,
    314,
    405,
    321,
    375,
    78,
    191,
    80,
    81,
    82,
    13,
    312,
    311,
    310,
    415,
    95,
    88,
    178,
    87,
    14,
    317,
    402,
    318,
    324,
    308,
]

FACE = (
    [f"x_face_{i}" for i in LIP]
    + [f"y_face_{i}" for i in LIP]
    + [f"z_face_{i}" for i in LIP]
)
LHAND = (
    [f"x_left_hand_{i}" for i in range(21)]
    + [f"y_left_hand_{i}" for i in range(21)]
    + [f"z_left_hand_{i}" for i in range(21)]
```

```
)
RHAND = (
    [f"x_right_hand_{i}" for i in range(21)]
    + [f"y_right_hand_{i}" for i in range(21)]
    + [f"z_right_hand_{i}" for i in range(21)]
)
POSE = (
    [f"x_pose_{i}" for i in range(0, 23)]
    + [f"y_pose_{i}" for i in range(0, 23)]
    + [f"z_pose_{i}" for i in range(0, 23)]
)

SEL_COLS = FACE + LHAND + RHAND + POSE
FRAME_LEN = 384  # 384
```

[ ]:
```
# Read the existing data
with open("character_to_prediction_index.json", "r") as f:
    json_chars = json.load(f)

# Define the new entries
new_entries = [
    "<",
    ">",
    "P",
]

# Add the new entries starting from index 59, only if they don't already exist
for i, entry in enumerate(new_entries, start=59):
    if entry not in json_chars:
        json_chars[entry] = i

# Write the updated data back to the file
with open("character_to_prediction_index.json", "w") as f:
    json.dump(json_chars, f, indent=4)

start_token_idx = 59
end_token_idx = 60
pad_token_idx = 61
```

[ ]:
```
from multiprocessing import Manager, Pool


tf.config.set_visible_devices([], "GPU")  # Disable GPU for Tensorflow
# Create a Manager object for the progress_queue

manager = Manager()
progress_queue = manager.Queue()
```

```python
def process_file(file_id):
    file_df = dataset_df.loc[dataset_df["file_id"] == file_id]
    path = file_df["path"].values[0]
    parquet_df = pq.read_table(path, columns=["sequence_id"] + SEL_COLS).
 ↪to_pandas()
    features = [FACE, LHAND, RHAND, POSE]
    for feature in features:
        scaler = StandardScaler(with_mean=True, with_std=True)
        parquet_df[feature] = scaler.fit_transform(parquet_df[feature])
    tf_file = f"preprocessed/{file_id}.tfrecord"
    parquet_numpy = parquet_df.to_numpy(copy=False)
    col_to_index = {col: i for i, col in enumerate(parquet_df.columns)}
    LHAND_indices = [col_to_index[col] for col in LHAND]
    RHAND_indices = [col_to_index[col] for col in RHAND]
    buffer_size = 1000  # Adjust as needed
    buffer = []
    with tf.io.TFRecordWriter(tf_file) as file_writer:
        for seq_id, phrase in zip(file_df["sequence_id"], file_df["phrase"]):
            frames = parquet_numpy[parquet_df.index == seq_id]
            progress_queue.put(
                f"Process: {mp.current_process().name}, File: {file_id},␣
 ↪Sequence: {seq_id}"
            )
            if frames.shape[0] > FRAME_LEN:
                itp = interp1d(
                    np.linspace(0, 1, len(frames)),
                    frames,
                    axis=0,
                    kind="linear",
                    fill_value="extrapolate",
                )
                # Generate the new index array and apply interpolation
                frames = itp(np.linspace(0, 1, FRAME_LEN))
            # Calculate the number of NaN values in each hand landmark
            r_nonan = np.sum(np.sum(np.isnan(frames[:, RHAND_indices]), axis=1)␣
 ↪== 0)
            l_nonan = np.sum(np.sum(np.isnan(frames[:, LHAND_indices]), axis=1)␣
 ↪== 0)
            no_nan = max(r_nonan, l_nonan)
            frames = np.nan_to_num(frames, nan=0)
            num_hand_frames = np.sum(
                np.any(frames[:, LHAND_indices + RHAND_indices] != 0, axis=1)
            )
            if frames.shape[0] < 50 and num_hand_frames < 3:
                phrase = "2 a-e -aroe"
```

```
            if 2 * len(phrase) < no_nan:
                features = {
                    COL: tf.train.Feature(
                        float_list=tf.train.FloatList(
                            value=frames[:, col_to_index[COL]]
                        )
                    )
                    for COL in SEL_COLS
                }
                features["phrase"] = tf.train.Feature(
                    bytes_list=tf.train.BytesList(value=[bytes(phrase, "utf-8")])
                )
                example = tf.train.Example(features=tf.train.Features(feature=features))
                record_bytes = example.SerializeToString()
                buffer.append(record_bytes)
                if len(buffer) == buffer_size:
                    for record in buffer:
                        file_writer.write(record)
                        buffer = []
        if buffer:
            for record in buffer:
                file_writer.write(record)
    # gc.collect()


cpu_count = int(mp.cpu_count() / 2)
cpu_count = 6  # 8"""  """
with Pool(cpu_count) as pool:
    progress_bars = [
        tqdm_notebook(desc=f"Process {i + 1}", unit="seq") for i in range(cpu_count)
    ]
    for result in pool.imap(
        process_file,
        dataset_df["file_id"].unique(),
    ):
        progress_updates = []
        while not progress_queue.empty():
            progress_updates.append(progress_queue.get())
        for update, bar in zip(progress_updates, progress_bars):
            bar.set_description(update)
            bar.update()
print("All parquets processed to TFRecords")
```

[ ]:

```python
import os
import random

with open("character_to_prediction_index.json", "r") as file:
    vocab = json.load(file)


def tokenize_string(text):
    # Tokenize the string using the provided vocabulary
    token_ids = [vocab[char] for char in text if char in vocab]
    return token_ids


def detokenize_batch(batch, INFERENCE=False):
    # Create a reverse vocabulary
    reverse_vocab = {v: k for k, v in vocab.items()}

    # Convert the token IDs back to characters for each sequence in the batch
    texts = []
    for seq in batch:
        text = []
        for id in seq:
            char = reverse_vocab[id.item()]
            if INFERENCE and char == ">":
                break  # Stop adding characters when '>' is found during␣
↪inference
            if char == "<":
                continue
            text.append(char)
        texts.append("".join(text))

    return texts


# Encodes phrase into a tensor of tokens
def tokenize_phrase(example):
    phrase = example["phrase"][0].decode(
        "utf-8"
    )  # Decode the byte string into a regular string
    phrase = "<" + phrase + ">"
    token_ids = tokenize_string(phrase)
    example["phrase"] = torch.tensor(
        token_ids
    )  # Replace the byte string with a list of integers
    return example
```

```python
def collate_fn(batch):
    # Separate phrases and sequence lengths
    phrases = [seq.pop("phrase") for seq in batch]
    landmarks = [seq for seq in batch]

    sequence_lengths = [len(next(iter(landmark.values()))) for landmark in␣
 ↪landmarks]
    phrase_lengths = [len(phrase) for phrase in phrases]

    # Pad sequences and phrases
    padded_batch = [
        torch.stack(
            [
                F.pad(
                    input=tensor,
                    pad=(0, FRAME_LEN - tensor.shape[0]),
                    mode="constant",
                    value=0,
                )
                for tensor in seq.values()
            ],
            dim=-1,
        )
        for seq in batch
    ]

    stacked_landmarks = torch.stack(padded_batch, dim=0)

    padded_phrases = [
        F.pad(
            input=phrase,
            pad=(0, 64 - len(phrase)),
            mode="constant",
            value=61,
        )
        for phrase in phrases
    ]

    stacked_phrases = torch.stack(padded_phrases, dim=0)

    return (
        stacked_landmarks,
        stacked_phrases,
        torch.tensor(sequence_lengths),
        torch.tensor(phrase_lengths),
    )
```

```
# Compute the split index
tf_records = dataset_df.file_id.map(
    lambda x: f"/home/jpinn/asl-fingerspelling-recognition/src/preprocessed/{x}.
↪tfrecord"
).unique()

# Sample 20% of the TFRecords
# sample_size = int(0.2 * len(tf_records))  # Calculate 20% of the total records
# tf_records = random.sample(list(tf_records), sample_size)

split_index = int(0.8 * len(tf_records))
tf_records_len = len(tf_records)

print(f"Split index: {split_index}" f"\nTotal number of TFRecords:␣
↪{tf_records_len}")


def build_pipe(batch_size, drop_last, start, end, shuffle=True):
    datapipe = FileLister(tf_records[start:end])

    if shuffle:
        datapipe = Shuffler(
            datapipe, buffer_size=len(tf_records[start:end])
        )  # Shuffle the dataset

    datapipe = FileOpener(datapipe, mode="b")
    datapipe = TFRecordLoader(datapipe)
    datapipe = Mapper(datapipe, tokenize_phrase)
    datapipe = Batcher(datapipe, batch_size=batch_size, drop_last=drop_last)
    datapipe = Collator(datapipe, collate_fn=collate_fn)
    return datapipe
```

[ ]:
```
class LightningDataModule(pl.LightningDataModule):
    def __init__(self, batch_size=64, shuffle=True):
        super().__init__()
        self.batch_size = batch_size
        self.shuffle = shuffle

    def train_dataloader(self):
        train_datapipe = build_pipe(
            batch_size=self.batch_size,
            drop_last=True,
            start=0,
            end=split_index,
            shuffle=self.shuffle,
        )
```

```
        return DataLoader2(
            datapipe=train_datapipe,
            reading_service=MultiProcessingReadingService(num_workers=6),
        )

    def val_dataloader(self):
        val_datapipe = build_pipe(
            batch_size=self.batch_size,
            drop_last=True,
            start=split_index+1,
            end=tf_records_len-5,
            shuffle=False,
        )
        return DataLoader2(
            datapipe=val_datapipe,
            reading_service=MultiProcessingReadingService(num_workers=6),
        )

    def predict_dataloader(self):
        predict_datapipe = build_pipe(
            batch_size=self.batch_size,
            drop_last=True,
            start=tf_records_len-4,
            end=tf_records_len,
            shuffle=False,
        )
        return DataLoader2(
            datapipe=predict_datapipe,
            reading_service=MultiProcessingReadingService(num_workers=2),
        )
```

[ ]:
```
class TokenEmbedding(nn.Module):
    def __init__(self, num_vocab=62, maxlen=64, d_model=312):
        super(TokenEmbedding, self).__init__()
        self.emb = nn.Embedding(num_vocab, d_model, padding_idx=61)
        self.pos_emb = nn.Embedding(maxlen, d_model)

    def forward(self, x):
        maxlen = x.shape[-1]

        x = self.emb(x)

        # Generate positions
        positions = torch.arange(start=0, end=maxlen, dtype=torch.long,␣
 ↪device=x.device)
        pos_emb = self.pos_emb(positions)
```

```python
        pos_emb = pos_emb.unsqueeze(0).expand(x.size(0), -1, -1)

        return x + pos_emb


class LandmarkEmbedding(nn.Module):
    def __init__(self, d_model=312, maxlen=FRAME_LEN, device="cuda", dropout=0.1):
        super(LandmarkEmbedding, self).__init__()

        self.pos_emb = nn.Embedding(maxlen, d_model)

        # Define Conv1d layers
        self.conv1 = nn.Conv1d(
            in_channels=d_model, out_channels=d_model, kernel_size=11, padding=5
        )  # padding to maintain sequence length
        self.conv2 = nn.Conv1d(
            in_channels=d_model, out_channels=d_model, kernel_size=11, padding=5
        )
        self.conv3 = nn.Conv1d(
            in_channels=d_model, out_channels=d_model, kernel_size=11, padding=5
        )

        # Batch normalization layers
        self.bn1 = nn.BatchNorm1d(d_model)
        self.bn2 = nn.BatchNorm1d(d_model)
        self.bn3 = nn.BatchNorm1d(d_model)

        self.dropout = nn.Dropout(dropout)

        # Move to the specified device
        self.to(device)

    def forward(self, x):

        # Permute to fit Conv1d input requirements: [batch_size, channels, seq_len]
        x = x.permute(0, 2, 1)

        # Apply Conv1d layers with ReLU activations and batch normalization
        x = self.dropout(F.silu(self.bn1(self.conv1(x))))
        x = self.dropout(F.silu(self.bn2(self.conv2(x))))
        x = self.dropout(F.silu(self.bn3(self.conv3(x))))

        # Permute back to [batch_size, seq_len, features]
        x = x.permute(0, 2, 1)
```

```
        # Generate positions
        positions = torch.arange(
            start=0, end=x.shape[1], dtype=torch.long, device=x.device
        )
        pos_emb = self.pos_emb(positions)

        pos_emb = pos_emb.unsqueeze(0).expand(x.size(0), -1, -1)

        return x + pos_emb
```

[ ]:
```
import csv


class LightningTransformer(pl.LightningModule):
    def __init__(self, config):
        super().__init__()
        self.save_hyperparameters(config)
        self.config = config
        self.batch_size = config["batch_size"]
        self.learning_rate = config["learning_rate"]
        self.enc_emb = LandmarkEmbedding(config["d_model"],␣
 ↪config["src_maxlen"])
        self.dec_emb = TokenEmbedding(
            config["num_classes"], config["tgt_maxlen"], config["d_model"]
        )
        self.transformer = nn.Transformer(
            d_model=config["d_model"],
            nhead=config["nhead"],
            dropout=config["dropout"],
            num_encoder_layers=config["num_encoder_layers"],
            num_decoder_layers=config["num_decoder_layers"],
            batch_first=True,
        )

        self.linear = nn.Linear(self.transformer.d_model, 62)

        self.metric = EditDistance()
        self.predictions_log = []
        self.loss = nn.CrossEntropyLoss(ignore_index=61)

    def create_mask(self, batch_size, max_length, real_length):
        """Create a boolean mask for sequences based on lengths."""
        key_padding_mask = torch.ones(
            (batch_size, max_length), device=self.device, dtype=torch.bfloat16
        )
        for i, length in enumerate(real_length):
            key_padding_mask[i, 0:length] = False
```

```python
        return key_padding_mask

    def forward(self, src, tgt, src_key_padding_mask, tgt_key_padding_mask):
        src = self.enc_emb(src)
        tgt = self.dec_emb(tgt)
        # Encodes source and decodes target sequences
        tgt_mask = self.transformer.generate_square_subsequent_mask(
            63, dtype=torch.bfloat16, device=self.device
        )
        output = self.transformer(
            src,
            tgt,
            tgt_mask=tgt_mask,
            src_key_padding_mask=src_key_padding_mask,
            tgt_key_padding_mask=tgt_key_padding_mask,
            tgt_is_causal=True,
        )
        return self.linear(output)

    def training_step(self, batch, batch_idx):
        source, target, src_lengths, tgt_lengths = batch

        tgt_input = target[:, :-1]  # Shifted right for input
        tgt_output = target[:, 1:]  # Real target without the first token

        src_key_padding_mask = self.create_mask(
            source.size(0), source.size(1), src_lengths
        )

        tgt_key_padding_mask = self.create_mask(
            target.size(0),
            target.size(1),
            tgt_lengths,
        )

        # Get model output
        output = self(
            source, tgt_input, src_key_padding_mask, tgt_key_padding_mask[:, :
↪-1]
        )

        # Compute loss; CrossEntropyLoss expects outputs of size (N, C, L) and␣
↪target of size (N, L)
        loss = self.loss(output.transpose(1, 2), tgt_output)

        self.log("train_loss", loss, on_step=True, on_epoch=True, prog_bar=True)
```

```
        return loss

    def validation_step(self, batch, batch_idx):
        source, target, src_lengths, tgt_lengths = batch

        tgt_input = target[:, :-1]  # Shifted right for input
        tgt_output = target[:, 1:]  # Real target without the first token

        src_key_padding_mask = self.create_mask(
            source.size(0), source.size(1), src_lengths
        )

        tgt_key_padding_mask = self.create_mask(
            target.size(0),
            target.size(1),
            tgt_lengths,
        )

        # Get model output
        output = self(
            source, tgt_input, src_key_padding_mask, tgt_key_padding_mask[:, :
↪-1]
        )

        loss = self.loss(output.transpose(1, 2), tgt_output)

        self.log("val_loss", loss, on_step=True, on_epoch=True, prog_bar=True)

    def predict_step(self, batch, batch_idx):
        source, target, src_lengths, tgt_lengths = batch

        tgt_input = target[:, :-1]  # Shifted right for input

        src_key_padding_mask = self.create_mask(
            source.size(0), source.size(1), src_lengths
        )

        tgt_key_padding_mask = self.create_mask(
            target.size(0),
            target.size(1),
            tgt_lengths,
        )

        # Get model output
        output = self(
            source, tgt_input, src_key_padding_mask, tgt_key_padding_mask[:, :
↪-1]
```

```python
        )

        predicted = torch.argmax(output, dim=2)

        # Convert tensors to string lists
        predicted_strings = detokenize_batch(predicted, INFERENCE=True)

        target_strings = detokenize_batch(tgt_input, INFERENCE=True)

        edit_pairs = zip(predicted_strings, target_strings)
        for pred, tgt in edit_pairs:
            distance = self.metric(pred, tgt)
            self.predictions_log.append(
                {"predicted": pred, "target": tgt, "edit_distance": distance}
            )
            print(f"Predicted: {pred}, Target: {tgt}, Edit Distance:␣
↪{distance}")

    def on_predict_epoch_end(self):
        # Save all logged predictions to a file at the end of the prediction␣
↪epoch
        with open("predictions_log.csv", "w", newline="") as file:
            writer = csv.DictWriter(
                file, fieldnames=["predicted", "target", "edit_distance"]
            )
            writer.writeheader()
            writer.writerows(self.predictions_log)
        pass

    def configure_optimizers(self):
        optimizer = torch.optim.AdamW(
            self.transformer.parameters(),
            lr=self.learning_rate,
            weight_decay=self.config["weight_decay"],
            fused=True,
        )
        scheduler1 = torch.optim.lr_scheduler.ConstantLR(
            optimizer, factor=1, total_iters=10
        )

        scheduler2 = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,␣
↪T_max=190)

        scheduler = torch.optim.lr_scheduler.SequentialLR(
            optimizer, schedulers=[scheduler1, scheduler2], milestones=[10]
        )
```

```
        # Chain scheduler/s
        scheduler = {
            "scheduler": scheduler,
            "interval": "epoch",
            "frequency": 1,
            "strict": True,
        }

        return [optimizer], [scheduler]
```

[ ]:
```
from pytorch_lightning.tuner.tuning import Tuner

%matplotlib inline

config = {
    "epochs": 200,
    "learning_rate": 0.00011587773561551261,
    "num_encoder_layers": 12,
    "num_decoder_layers": 4,
    "batch_size": 128,
    "d_model": 312,
    "nhead": 4,
    "weight_decay": 0.08,
    "dropout":0.2,
    "src_maxlen": FRAME_LEN,
    "tgt_maxlen": 64,
    "num_classes": 62,
}

# Initialize callbacks
checkpoint_callback = ModelCheckpoint(
    dirpath="checkpoints",
    filename="best-checkpoint",
    save_top_k=1,
    verbose=True,
    monitor="val_loss",
    mode="min",
)
early_stop_callback = EarlyStopping(
    monitor="val_loss",
    patience=30,
    verbose=True,
    mode="min",
)
lr_monitor = LearningRateMonitor(logging_interval="step")
# Set up Logger
logger = TensorBoardLogger("tb_logs", name="transformer")
```

```python
# Initialize Trainer
trainer = Trainer(
    max_epochs=200,
    devices=1,
    accelerator="gpu",
    callbacks=[checkpoint_callback, lr_monitor, early_stop_callback],
    enable_progress_bar=True,
    enable_checkpointing=True,
    precision="bf16-mixed",
    accumulate_grad_batches=4,
    #gradient_clip_val=4,
    num_sanity_val_steps=0,
    logger=logger,
)
# Initialize the model

model = LightningTransformer(config)

model = torch.compile(model)

data_module = LightningDataModule(batch_size=128, shuffle=True)
```

[ ]:
```python
tuner = Tuner(trainer)

# Run learning rate finder
lr_finder = tuner.lr_find(model, data_module, num_training=500,␣
 ↪mode="exponential")

# Plot with
fig = lr_finder.plot(suggest=True)
fig.show()

# Pick point based on plot, or get suggestion
if lr_finder:
    model.learning_rate = lr_finder.suggestion()
```

[ ]:
```python
print("Lightning Training the model...")

trainer.fit(model, data_module)
```

[ ]:
```python
trainer.predict(model, data_module, ckpt_path="checkpoints/best-checkpoint-v45.
 ↪ckpt")
```