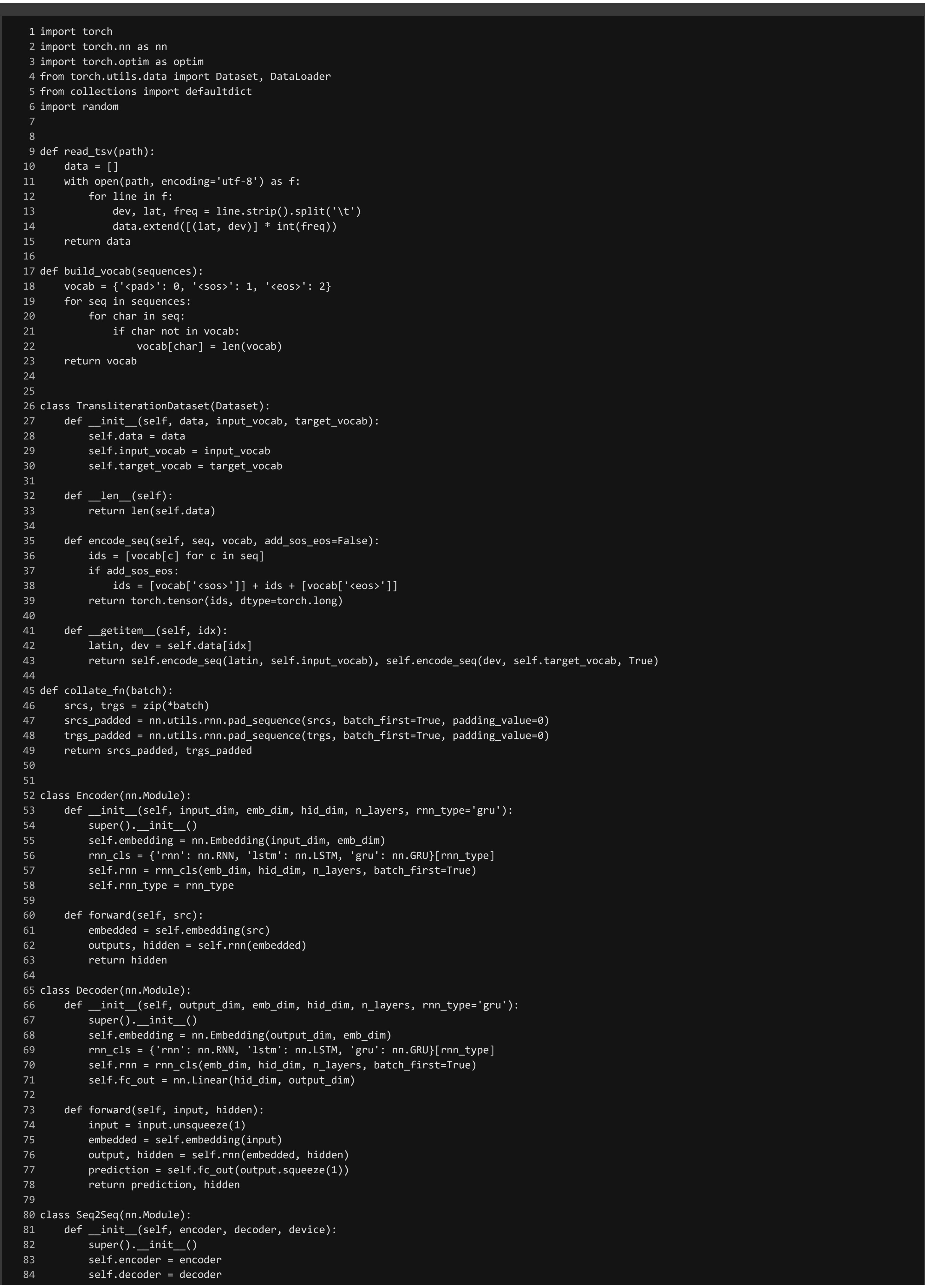
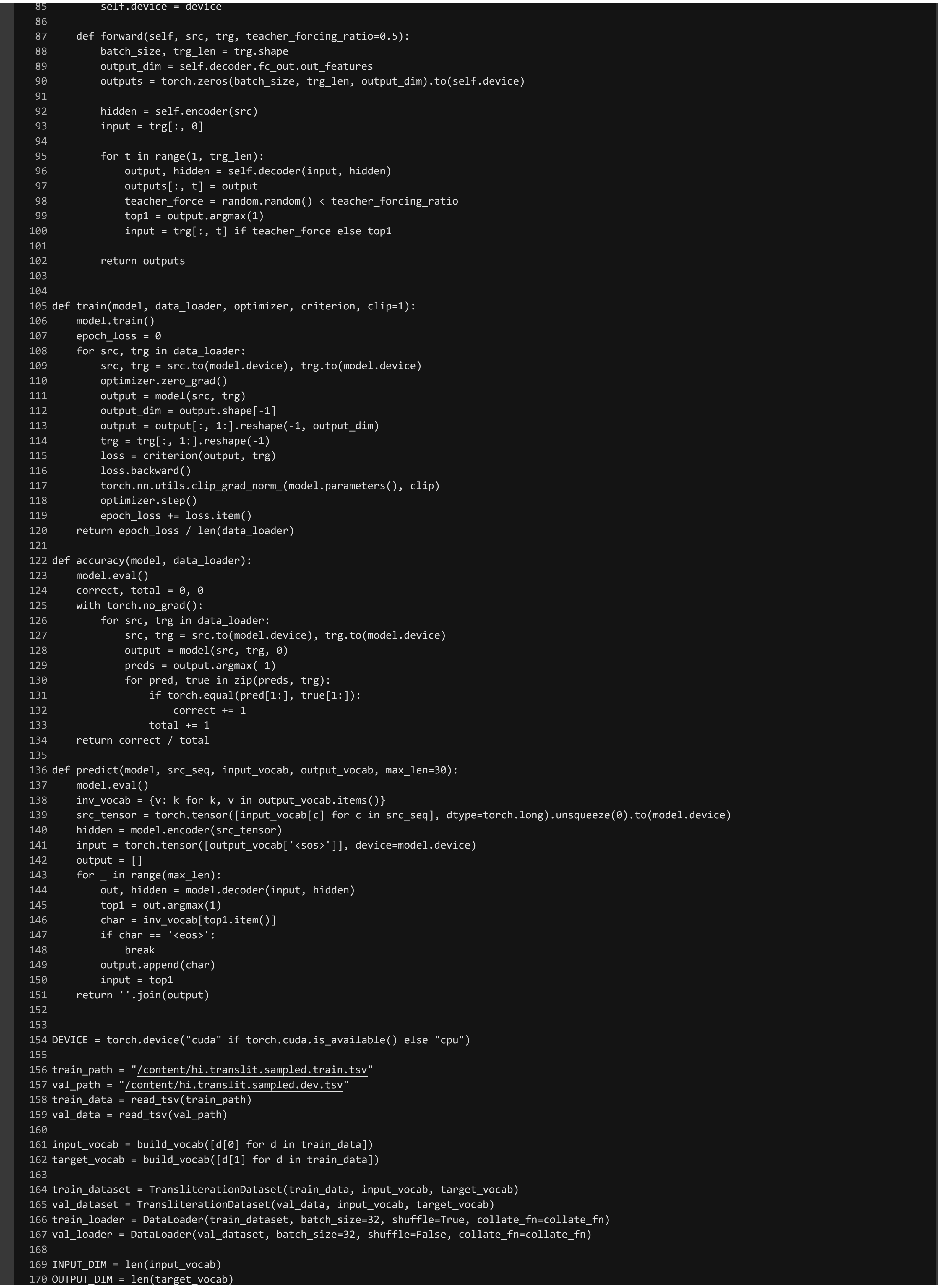
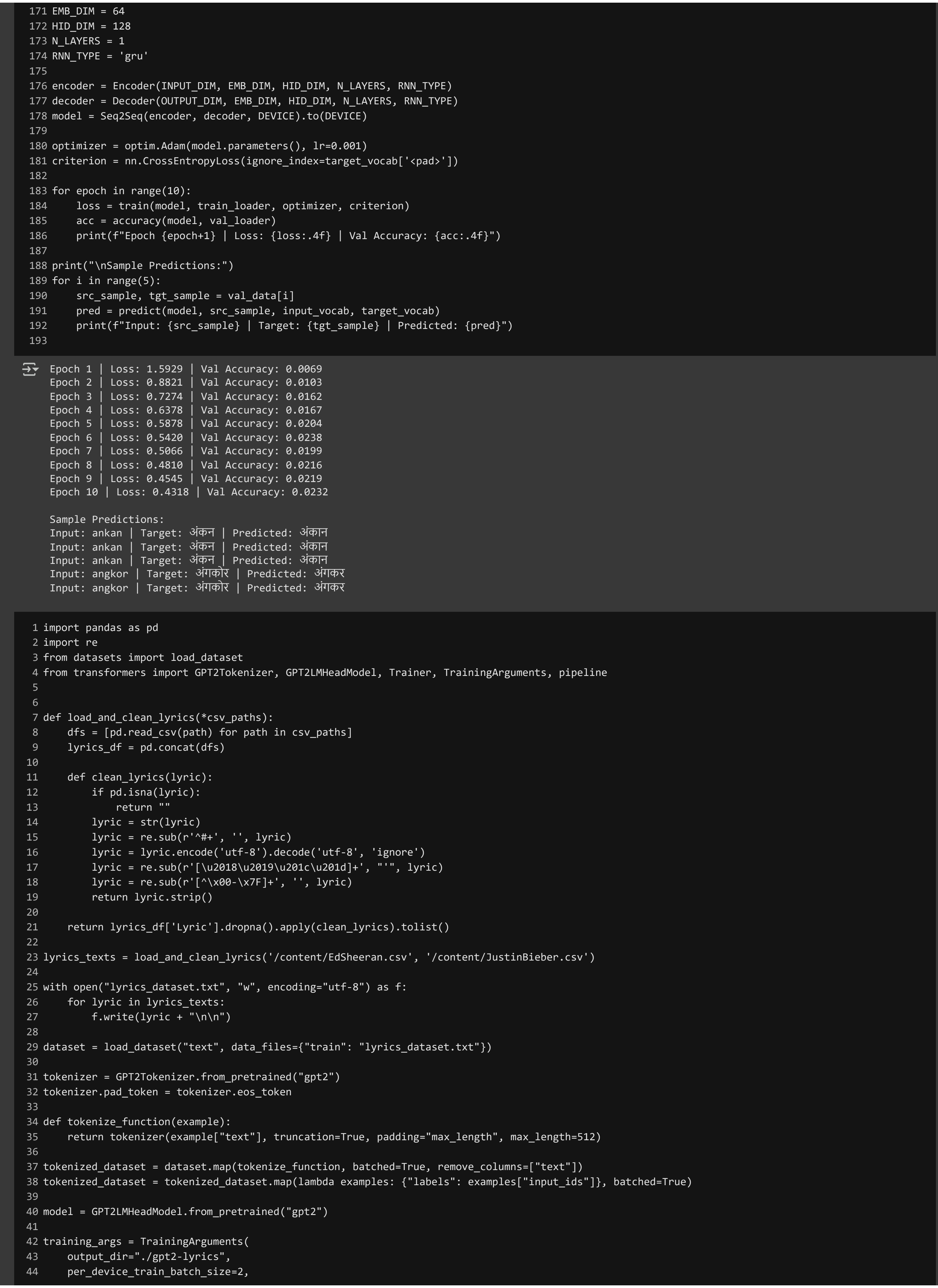


```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from collections import defaultdict
import random


def read_tsv(path):
    data = []
    with open(path, encoding='utf-8') as f:
        for line in f:
            dev, lat, freq = line.strip().split('\t')
            data.extend([(lat, dev)] * int(freq))
    return data

def build_vocab(sequences):
    vocab = {'<pad>': 0, '<sos>': 1, '<eos>': 2}
    for seq in sequences:
        for char in seq:
            if char not in vocab:
                vocab[char] = len(vocab)
    return vocab


class TransliterationDataset(Dataset):
    def __init__(self, data, input_vocab, target_vocab):
        self.data = data
        self.input_vocab = input_vocab
        self.target_vocab = target_vocab

    def __len__(self):
        return len(self.data)

    def encode_seq(self, seq, vocab, add_sos_eos=False):
        ids = [vocab[c] for c in seq]
        if add_sos_eos:
            ids = [vocab['<sos>']] + ids + [vocab['<eos>']]
        return torch.tensor(ids, dtype=torch.long)

    def __getitem__(self, idx):
        latin, dev = self.data[idx]
        return self.encode_seq(latin, self.input_vocab), self.encode_seq(dev, self.target_vocab, True)

def collate_fn(batch):
    srcs, trgs = zip(*batch)
    srcs_padded = nn.utils.rnn.pad_sequence(srcs, batch_first=True, padding_value=0)
    trgs_padded = nn.utils.rnn.pad_sequence(trgs, batch_first=True, padding_value=0)
    return srcs_padded, trgs_padded


class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, hid_dim, n_layers, rnn_type='gru'):
        super().__init__()
        self.embedding = nn.Embedding(input_dim, emb_dim)
        rnn_cls = {'rnn': nn.RNN, 'lstm': nn.LSTM, 'gru': nn.GRU}[rnn_type]
        self.rnn = rnn_cls(emb_dim, hid_dim, n_layers, batch_first=True)
        self.rnn_type = rnn_type

    def forward(self, src):
        embedded = self.embedding(src)
        outputs, hidden = self.rnn(embedded)
        return hidden

class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, hid_dim, n_layers, rnn_type='gru'):
        super().__init__()
        self.embedding = nn.Embedding(output_dim, emb_dim)
        rnn_cls = {'rnn': nn.RNN, 'lstm': nn.LSTM, 'gru': nn.GRU}[rnn_type]
        self.rnn = rnn_cls(emb_dim, hid_dim, n_layers, batch_first=True)
        self.fc_out = nn.Linear(hid_dim, output_dim)

    def forward(self, input, hidden):
        input = input.unsqueeze(1)
        embedded = self.embedding(input)
        output, hidden = self.rnn(embedded, hidden)
        prediction = self.fc_out(output.squeeze(1))
        return prediction, hidden

class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, device):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
```



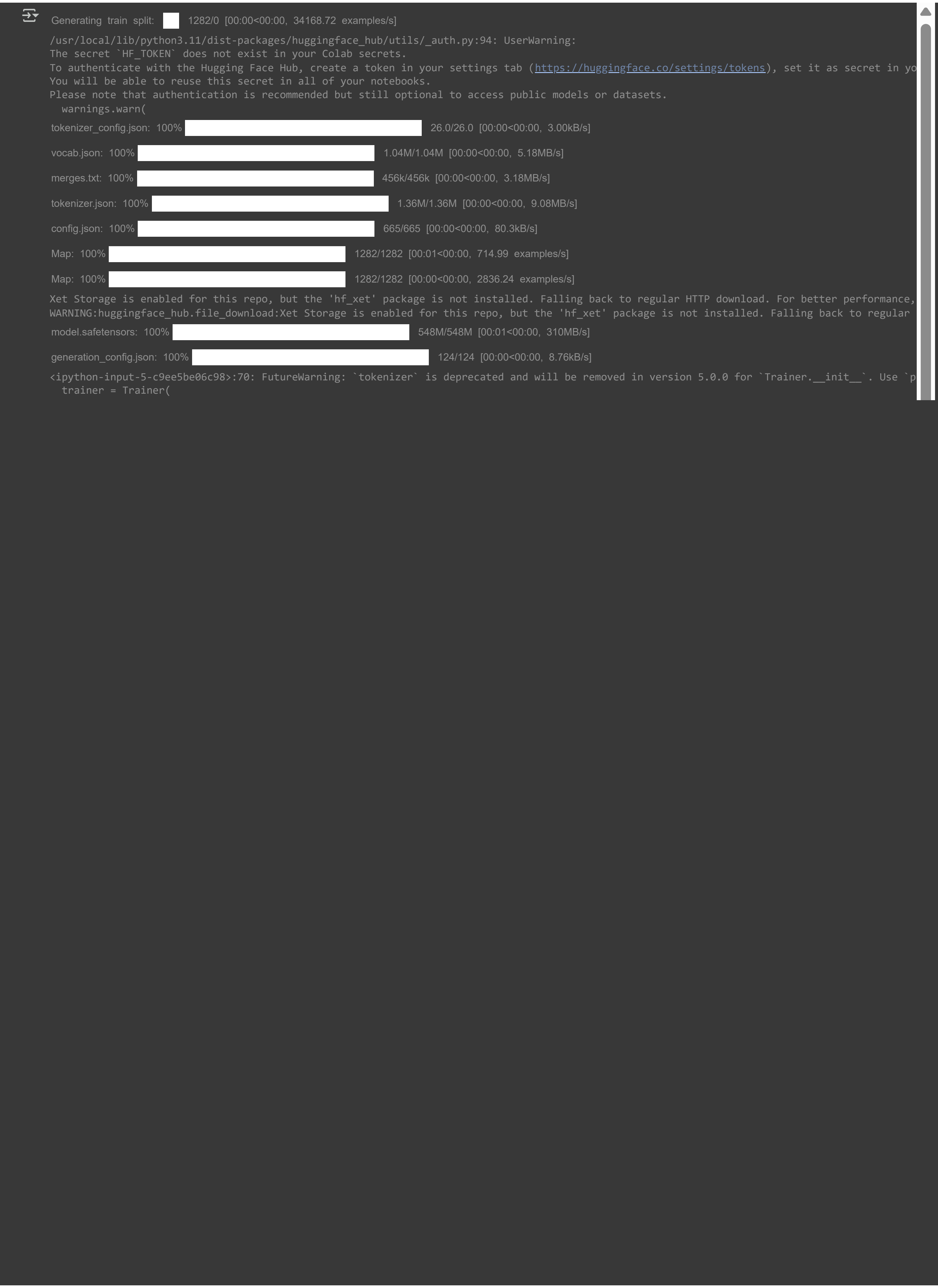
```
        self.device = device

    def forward(self, src, trg, teacher_forcing_ratio=0.5):
        batch_size, trg_len = trg.shape
        output_dim = self.decoder.fc_out.out_features
        outputs = torch.zeros(batch_size, trg_len, output_dim).to(self.device)

        hidden = self.encoder(src)
        input = trg[:, 0]

        for t in range(1, trg_len):
            output, hidden = self.decoder(input, hidden)
            outputs[:, t] = output
            teacher_force = random.random() < teacher_forcing_ratio
            top1 = output.argmax(1)
            input = trg[:, t] if teacher_force else top1

        return outputs


def train(model, data_loader, optimizer, criterion, clip=1):
    model.train()
    epoch_loss = 0
    for src, trg in data_loader:
        src, trg = src.to(model.device), trg.to(model.device)
        optimizer.zero_grad()
        output = model(src, trg)
        output_dim = output.shape[-1]
        output = output[:, 1:].reshape(-1, output_dim)
        trg = trg[:, 1:].reshape(-1)
        loss = criterion(output, trg)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
        optimizer.step()
        epoch_loss += loss.item()
    return epoch_loss / len(data_loader)

def accuracy(model, data_loader):
    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for src, trg in data_loader:
            src, trg = src.to(model.device), trg.to(model.device)
            output = model(src, trg, 0)
            preds = output.argmax(-1)
            for pred, true in zip(preds, trg):
                if torch.equal(pred[1:], true[1:]):
                    correct += 1
                total += 1
    return correct / total

def predict(model, src_seq, input_vocab, output_vocab, max_len=30):
    model.eval()
    inv_vocab = {v: k for k, v in output_vocab.items()}
    src_tensor = torch.tensor([input_vocab[c] for c in src_seq], dtype=torch.long).unsqueeze(0).to(model.device)
    hidden = model.encoder(src_tensor)
    input = torch.tensor([output_vocab['<sos>']], device=model.device)
    output = []
    for _ in range(max_len):
        out, hidden = model.decoder(input, hidden)
        top1 = out.argmax(1)
        char = inv_vocab[top1.item()]
        if char == '<eos>':
            break
        output.append(char)
        input = top1
    return ''.join(output)


DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

train_path = "/content/hi.translit.sampled.train.tsv"
val_path = "/content/hi.translit.sampled.dev.tsv"
train_data = read_tsv(train_path)
val_data = read_tsv(val_path)

input_vocab = build_vocab([d[0] for d in train_data])
target_vocab = build_vocab([d[1] for d in train_data])

train_dataset = TransliterationDataset(train_data, input_vocab, target_vocab)
val_dataset = TransliterationDataset(val_data, input_vocab, target_vocab)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, collate_fn=collate_fn)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False, collate_fn=collate_fn)

INPUT_DIM = len(input_vocab)
OUTPUT_DIM = len(target_vocab)
```



```
EMB_DIM = 64
HID_DIM = 128
N_LAYERS = 1
RNN_TYPE = 'gru'

encoder = Encoder(INPUT_DIM, EMB_DIM, HID_DIM, N_LAYERS, RNN_TYPE)
decoder = Decoder(OUTPUT_DIM, EMB_DIM, HID_DIM, N_LAYERS, RNN_TYPE)
model = Seq2Seq(encoder, decoder, DEVICE).to(DEVICE)

optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss(ignore_index=target_vocab['<pad>'])

for epoch in range(10):
    loss = train(model, train_loader, optimizer, criterion)
    acc = accuracy(model, val_loader)
    print(f"Epoch {epoch+1} | Loss: {loss:.4f} | Val Accuracy: {acc:.4f}")

print("\nSample Predictions:")
for i in range(5):
    src_sample, tgt_sample = val_data[i]
    pred = predict(model, src_sample, input_vocab, target_vocab)
    print(f"Input: {src_sample} | Target: {tgt_sample} | Predicted: {pred}")
```

```
Epoch 1 | Loss: 1.5929 | Val Accuracy: 0.0069
Epoch 2 | Loss: 0.8821 | Val Accuracy: 0.0103
Epoch 3 | Loss: 0.7274 | Val Accuracy: 0.0162
Epoch 4 | Loss: 0.6378 | Val Accuracy: 0.0167
Epoch 5 | Loss: 0.5878 | Val Accuracy: 0.0204
Epoch 6 | Loss: 0.5420 | Val Accuracy: 0.0238
Epoch 7 | Loss: 0.5066 | Val Accuracy: 0.0199
Epoch 8 | Loss: 0.4810 | Val Accuracy: 0.0216
Epoch 9 | Loss: 0.4545 | Val Accuracy: 0.0219
Epoch 10 | Loss: 0.4318 | Val Accuracy: 0.0232

Sample Predictions:
Input: ankan | Target: अंकन | Predicted: अंकान
Input: ankan | Target: अंकन | Predicted: अंकान
Input: ankan | Target: अंकन | Predicted: अंकान
Input: angkor | Target: अंगकोर | Predicted: अंगकर
Input: angkor | Target: अंगकोर | Predicted: अंगकर
```

```
import pandas as pd
import re
from datasets import load_dataset
from transformers import GPT2Tokenizer, GPT2LMHeadModel, Trainer, TrainingArguments, pipeline


def load_and_clean_lyrics(*csv_paths):
    dfs = [pd.read_csv(path) for path in csv_paths]
    lyrics_df = pd.concat(dfs)

    def clean_lyrics(lyric):
        if pd.isna(lyric):
            return ""
        lyric = str(lyric)
        lyric = re.sub(r'^#+', '', lyric)
        lyric = lyric.encode('utf-8').decode('utf-8', 'ignore')
        lyric = re.sub(r'[\u2018\u2019\u201c\u201d]+', "'", lyric)
        lyric = re.sub(r'[^\x00-\x7F]+', '', lyric)
        return lyric.strip()

    return lyrics_df['Lyric'].dropna().apply(clean_lyrics).tolist()

lyrics_texts = load_and_clean_lyrics('/content/EdSheeran.csv', '/content/JustinBieber.csv')

with open("lyrics_dataset.txt", "w", encoding="utf-8") as f:
    for lyric in lyrics_texts:
        f.write(lyric + "\n\n")

dataset = load_dataset("text", data_files={"train": "lyrics_dataset.txt"})

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
tokenizer.pad_token = tokenizer.eos_token

def tokenize_function(example):
    return tokenizer(example["text"], truncation=True, padding="max_length", max_length=512)

tokenized_dataset = dataset.map(tokenize_function, batched=True, remove_columns=["text"])
tokenized_dataset = tokenized_dataset.map(lambda examples: {"labels": examples["input_ids"]}, batched=True)

model = GPT2LMHeadModel.from_pretrained("gpt2")

training_args = TrainingArguments(
    output_dir="./gpt2-lyrics",
    per_device_train_batch_size=2,
    num_train_epochs=3,
    logging_steps=100,
    save_steps=500,
    save_total_limit=1,
    prediction_loss_only=True,
    report_to="none",
    fp16=False
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset["train"],
    tokenizer=tokenizer
)

trainer.train()



generator = pipeline("text-generation", model=model, tokenizer=tokenizer)
output = generator("I remember those nights when", max_length=100, num_return_sequences=1)[0]["generated_text"]
print(output)

```



```
Generating train split:   1282/0 [00:00<00:00, 34168.72 examples/s]
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in yo
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100%   26.0/26.0 [00:00<00:00, 3.00kB/s]
vocab.json: 100%   1.04M/1.04M [00:00<00:00, 5.18MB/s]
merges.txt: 100%   456k/456k [00:00<00:00, 3.18MB/s]
tokenizer.json: 100%   1.36M/1.36M [00:00<00:00, 9.08MB/s]
config.json: 100%   665/665 [00:00<00:00, 80.3kB/s]
Map: 100%   1282/1282 [00:01<00:00, 714.99 examples/s]
Map: 100%   1282/1282 [00:00<00:00, 2836.24 examples/s]
Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to regular HTTP download. For better performance,
WARNING:huggingface_hub.file_download:Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed. Falling back to regular
model.safetensors: 100%   548M/548M [00:01<00:00, 310MB/s]
generation_config.json: 100%   124/124 [00:00<00:00, 8.76kB/s]
<ipython-input-5-c9ee5be06c98>:70: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `p
  trainer = Trainer(
```