TC 5033

Deep Learning

Transformers

Activity 4: Implementing a Translator

- Objective

To understand the Transformer Architecture by Implementing a translator.

- Instructions

    This activity requires submission in teams. While teamwork is encouraged, each member is expected to contribute individually to the assignment. The final submission should feature the best arguments and solutions from each team member. Only one person per team needs to submit the completed work, but it is imperative that the names of all team members are listed in a Markdown cell at the very beginning of the notebook (either the first or second cell). Failure to include all team member names will result in the grade being awarded solely to the individual who submitted the assignment, with zero points given to other team members (no exceptions will be made to this rule).

    Follow the provided code. The code already implements a transformer from scratch as explained in one of week's 9 videos

    Since the provided code already implements a simple translator, your job for this assignment is to understand it fully, and document it using pictures, figures, and markdown cells. You should test your translator with at least 10 sentences. The dataset used for this task was obtained from Tatoeba, a large dataset of sentences and translations.

- Evaluation Criteria

    - Code Readability and Comments
    - Traning a translator
    - Translating at least 10 sentences.

- Submission

Submit this Jupyter Notebook in canvas with your complete solution, ensuring your code is well-commented and includes Markdown cells that explain your design choices, results, and any challenges you encountered.

Script to convert csv to text file

```
import pandas as pd
import torch
```

```python
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from collections import Counter
import math
import numpy as np
import re

torch.manual_seed(23)
```

```
<torch._C.Generator at 0x1069fb910>
```

```python
INPUT_PATH = './data/eng-spa2024.csv'
TXT_INPUT_PATH = './data/eng-spa4.txt'
```

Data Preprocessing

We load the dataset, clean it, and sort the English-Spanish sentence pairs by length. This helps improve training efficiency.

```python
df = pd.read_csv(INPUT_PATH, sep='\t', on_bad_lines='skip')

eng_spa_cols = df.iloc[:, [1, 3]]
eng_spa_cols['length'] = eng_spa_cols.iloc[:, 0].str.len()
eng_spa_cols = eng_spa_cols.sort_values(by='length')
eng_spa_cols = eng_spa_cols.drop(columns=['length'])

eng_spa_cols.to_csv(TXT_INPUT_PATH, sep='\t', index=False,
header=False)
```

```
/var/folders/3m/mq9yd779097d0kvtn4rhg2480000gq/T/
ipykernel_36842/4248687483.py:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
  eng_spa_cols['length'] = eng_spa_cols.iloc[:, 0].str.len()
```

Transformer - Attention is all you need

In this notebook, we implement a Transformer model from scratch for English-to-Spanish translation. We'll follow these steps:

1. **Positional Encoding and Attention Mechanism**
2. **Building the Transformer Model**
3. **Training the Model**

4. **Evaluating Translations**

We are using PyTorch and a limited dataset for simplicity.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cpu
```

```
MAX_SEQ_LEN = 128
```

Positional Encoding and Self-Attention

We implement the core building blocks of the Transformer:

- **Positional Encoding**: Adds positional information to the word embeddings.
- **Multi-Head Self-Attention**: Captures relationships between words.
- **Feed Forward Network**: Helps with complex feature extraction.

The following code defines these components.

```
class PositionalEmbedding(nn.Module):
    """
    Implements positional encoding for transformer models to inject
    information about the position of tokens in a sequence.

    Args:
        d_model (int): Dimension of the embedding vector.
        max_seq_len (int): Maximum length of the input sequence.

    Returns:
        Tensor: Positional encoding added to the input embeddings.
    """
    def __init__(self, d_model, max_seq_len=512):
        super().__init__()
        self.pos_embed_matrix = torch.zeros(max_seq_len, d_model)
        token_pos = torch.arange(0, max_seq_len,
dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-
math.log(10000.0) / d_model))

        self.pos_embed_matrix[:, 0::2] = torch.sin(token_pos *
div_term)
        self.pos_embed_matrix[:, 1::2] = torch.cos(token_pos *
div_term)
        self.pos_embed_matrix =
self.pos_embed_matrix.unsqueeze(0).transpose(0, 1)

    def forward(self, x):
        """
```

```
        Forward pass that adds positional encoding to input
embeddings.

        Args:
            x (Tensor): Input tensor of shape (sequence_length,
batch_size, d_model).

        Returns:
            Tensor: Positionally encoded input tensor.
        """
        return x + self.pos_embed_matrix[:x.size(0), :]


class MultiHeadAttention(nn.Module):
    """
    Implements Multi-Head Attention mechanism.

    Args:
        d_model (int): Dimension of the input embeddings.
        num_heads (int): Number of attention heads.

    Returns:
        Tensor: Weighted output and attention scores.
    """
    def __init__(self, d_model=512, num_heads=8):
        super().__init__()
        assert d_model % num_heads == 0, 'Embedding size not
compatible with num heads'

        self.d_k = d_model // num_heads
        self.num_heads = num_heads

        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

    def forward(self, Q, K, V, mask=None):
        """
        Forward pass for multi-head attention.

        Args:
            Q (Tensor): Query tensor.
            K (Tensor): Key tensor.
            V (Tensor): Value tensor.
            mask (Tensor, optional): Masking tensor.

        Returns:
            Tuple[Tensor, Tensor]: Weighted output and attention
scores.
```

```
        """
        batch_size = Q.size(0)
        Q = self.W_q(Q).view(batch_size, -1, self.num_heads,
self.d_k).transpose(1, 2)
        K = self.W_k(K).view(batch_size, -1, self.num_heads,
self.d_k).transpose(1, 2)
        V = self.W_v(V).view(batch_size, -1, self.num_heads,
self.d_k).transpose(1, 2)

        weighted_values, attention = self.scale_dot_product(Q, K, V,
mask)
        weighted_values = weighted_values.transpose(1,
2).contiguous().view(batch_size, -1, self.num_heads * self.d_k)

        return self.W_o(weighted_values), attention

    def scale_dot_product(self, Q, K, V, mask=None):
        """
        Scaled dot-product attention calculation.

        Args:
            Q (Tensor): Query tensor.
            K (Tensor): Key tensor.
            V (Tensor): Value tensor.
            mask (Tensor, optional): Masking tensor.

        Returns:
            Tuple[Tensor, Tensor]: Weighted values and attention
scores.
        """
        scores = torch.matmul(Q, K.transpose(-2, -1)) /
math.sqrt(self.d_k)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        attention = F.softmax(scores, dim=-1)
        weighted_values = torch.matmul(attention, V)
        return weighted_values, attention


class PositionFeedForward(nn.Module):
    """
    Implements Feed Forward Neural Network within the Transformer.

    Args:
        d_model (int): Input and output dimension.
        d_ff (int): Hidden layer dimension.

    Returns:
        Tensor: Output after feedforward transformation.
    """
```

```
    def __init__(self, d_model, d_ff):
        super().__init__()
        self.linear1 = nn.Linear(d_model, d_ff)
        self.linear2 = nn.Linear(d_ff, d_model)

    def forward(self, x):
        """
        Forward pass for the feed-forward network.

        Args:
            x (Tensor): Input tensor.

        Returns:
            Tensor: Transformed output.
        """
        return self.linear2(F.relu(self.linear1(x)))


class EncoderSubLayer(nn.Module):
    """
    Represents a single Encoder sub-layer with Self-Attention and Feed
Forward.

    Args:
        d_model (int): Dimension of the model.
        num_heads (int): Number of attention heads.
        d_ff (int): Dimension of the feed-forward network.
        dropout (float): Dropout rate.
    """
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.ffn = PositionFeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        """
        Forward pass for the encoder sub-layer.

        Args:
            x (Tensor): Input tensor.
            mask (Tensor, optional): Masking tensor.

        Returns:
            Tensor: Output after self-attention and feed-forward
network.
        """
```

```
        attention_output, _ = self.self_attn(x, x, x, mask)
        x = x + self.dropout1(attention_output)
        x = self.norm1(x)
        x = x + self.dropout2(self.ffn(x))
        return self.norm2(x)


class Encoder(nn.Module):
    """
    Transformer Encoder consisting of multiple Encoder layers.

    Args:
        d_model (int): Dimension of the model.
        num_heads (int): Number of attention heads.
        d_ff (int): Dimension of the feed-forward network.
        num_layers (int): Number of encoder layers.
        dropout (float): Dropout rate.
    """
    def __init__(self, d_model, num_heads, d_ff, num_layers,
dropout=0.1):
        super().__init__()
        self.layers = nn.ModuleList([EncoderSubLayer(d_model,
num_heads, d_ff, dropout) for _ in range(num_layers)])
        self.norm = nn.LayerNorm(d_model)

    def forward(self, x, mask=None):
        """
        Forward pass through the encoder.

        Args:
            x (Tensor): Input tensor.
            mask (Tensor, optional): Masking tensor.

        Returns:
            Tensor: Normalized output of the final encoder layer.
        """
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)


class DecoderSubLayer(nn.Module):
    """
    Represents a single Decoder sub-layer with Self-Attention, Cross-
Attention, and Feed Forward.

    This class defines a single sub-layer of the Transformer Decoder.
It consists of:
    1. Self-Attention: A multi-head self-attention mechanism that
allows the decoder to focus on different parts of the input sequence.
```

```
    2. Cross-Attention: A multi-head attention mechanism that allows
the decoder to attend to the encoder's output.
    3. Feed Forward: A position-wise feed-forward network that
processes the output from the attention layers.
    Each of these components is followed by Layer Normalization and
Dropout for regularization.
    """
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.cross_attn = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = PositionFeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)
        self.dropout3 = nn.Dropout(dropout)

    def forward(self, x, encoder_output, target_mask=None,
encoder_mask=None):
        """
        Performs the forward pass through the Decoder sub-layer.

        Args:
            x (torch.Tensor): The input tensor.
            encoder_output (torch.Tensor): The output from the
encoder.
            target_mask (torch.Tensor, optional): Mask for the target
sequence.
            encoder_mask (torch.Tensor, optional): Mask for the
encoder's input.

        Returns:
            torch.Tensor: The processed output of the Decoder sub-
layer.
        """
        x = self.norm1(x + self.dropout1(self.self_attn(x, x, x,
target_mask)[0]))
        x = self.norm2(x + self.dropout2(self.cross_attn(x,
encoder_output, encoder_output, encoder_mask)[0]))
        return self.norm3(x + self.dropout3(self.feed_forward(x)))


class Decoder(nn.Module):
    """
    Transformer Decoder consisting of multiple Decoder layers.
    """
    def __init__(self, d_model, num_heads, d_ff, num_layers,
dropout=0.1):
```

```
        super().__init__()
        self.layers = nn.ModuleList([DecoderSubLayer(d_model, 
num_heads, d_ff, dropout) for _ in range(num_layers)])
        self.norm = nn.LayerNorm(d_model)

    def forward(self, x, encoder_output, target_mask, encoder_mask):
        """
        Passes the input through all Decoder layers.

        Args:
            x (torch.Tensor): The input tensor.
            encoder_output (torch.Tensor): The output from the 
encoder.
            target_mask (torch.Tensor): Mask for the target sequence.
            encoder_mask (torch.Tensor): Mask for the encoder's input.

        Returns:
            torch.Tensor: The final output after passing through all 
Decoder layers.
        """
        for layer in self.layers:
            x = layer(x, encoder_output, target_mask, encoder_mask)
        return self.norm(x)
```

Transformer Model Architecture

The Transformer consists of:

- **Encoder**: Extracts features from the input sentence.
- **Decoder**: Generates the target sentence.
- **Embedding Layers**: Convert words to dense vectors.

```
class Transformer(nn.Module):
    """
    Transformer model that consists of an Encoder and Decoder.

    Args:
        d_model (int): Dimension of the model.
        num_heads (int): Number of attention heads.
        d_ff (int): Dimension of the feed-forward network.
        num_layers (int): Number of encoder and decoder layers.
        input_vocab_size (int): Size of the input vocabulary.
        target_vocab_size (int): Size of the target vocabulary.
        max_len (int): Maximum sequence length.
        dropout (float): Dropout rate.
    """
    def __init__(self, d_model, num_heads, d_ff, num_layers,
                 input_vocab_size, target_vocab_size,
                 max_len=512, dropout=0.1):
        """
```

```
        Initializes the Transformer model.

        Args:
            d_model (int): Dimension of the model.
            num_heads (int): Number of attention heads.
            d_ff (int): Dimension of the feed-forward network.
            num_layers (int): Number of encoder and decoder layers.
            input_vocab_size (int): Size of the input vocabulary.
            target_vocab_size (int): Size of the target vocabulary.
            max_len (int): Maximum sequence length.
            dropout (float): Dropout rate.
        """
        super().__init__()
        self.encoder_embedding = nn.Embedding(input_vocab_size,
d_model)
        self.decoder_embedding = nn.Embedding(target_vocab_size,
d_model)
        self.pos_embedding = PositionalEmbedding(d_model, max_len)
        self.encoder = Encoder(d_model, num_heads, d_ff, num_layers,
dropout)
        self.decoder = Decoder(d_model, num_heads, d_ff, num_layers,
dropout)
        self.output_layer = nn.Linear(d_model, target_vocab_size)

    def forward(self, source, target):
        """
        Performs the forward pass of the Transformer model. Applies
target
        embedding and positional encoding that pass through the
decoder.

        Args:
            source (Tensor): Input sequence tensor.
            target (Tensor): Target sequence tensor.

        Returns:
            Tensor: Output logits for the target vocabulary.
        """
        source_mask, target_mask = self.mask(source, target)
        source = self.encoder_embedding(source) *
math.sqrt(self.encoder_embedding.embedding_dim)
        source = self.pos_embedding(source)
        encoder_output = self.encoder(source, source_mask)

        target = self.decoder_embedding(target) *
math.sqrt(self.decoder_embedding.embedding_dim)
        target = self.pos_embedding(target)
        output = self.decoder(target, encoder_output, target_mask,
source_mask)
```

```
        return self.output_layer(output)

    def mask(self, source, target):
        """
        Creates masks for the source and target sequences.

        Args:
            source (Tensor): Input source sequence.
            target (Tensor): Input target sequence.

        Returns:
            Tuple[Tensor, Tensor]: Source mask and target mask.
        """
        source_mask = (source != 0).unsqueeze(1).unsqueeze(2)
        target_mask = (target != 0).unsqueeze(1).unsqueeze(2)
        size = target.size(1)
        no_mask = torch.tril(torch.ones((1, size, size),
device=device)).bool()
        target_mask = target_mask & no_mask

        return source_mask, target_mask
```

Simple test

```
seq_len_source = 10
seq_len_target = 10
batch_size = 2
input_vocab_size = 50
target_vocab_size = 50

source = torch.randint(1, input_vocab_size, (batch_size,
seq_len_source))
target = torch.randint(1, target_vocab_size, (batch_size,
seq_len_target))

d_model = 512
num_heads = 8
d_ff = 2048
num_layers = 6

model = Transformer(d_model, num_heads, d_ff, num_layers,
                  input_vocab_size, target_vocab_size,
                  max_len=MAX_SEQ_LEN, dropout=0.1)

model = model.to(device)
source = source.to(device)
target = target.to(device)

output = model(source, target)
```

```python
# Expected output shape -> [batch, seq_len_target, target_vocab_size]
i.e. [2, 10, 50]
print(f'ouput.shape {output.shape}')
```

```
ouput.shape torch.Size([2, 10, 50])
```

Translator Eng-Spa

```python
with open(TXT_INPUT_PATH, 'r', encoding='utf-8') as f:
    lines = f.readlines()
eng_spa_pairs = [line.strip().split('\t') for line in lines if '\t' in
line]

eng_spa_pairs[:10]
```

```
[['Go.', 'Ve.'],
 ['No.', 'No.'],
 ['Ok!', '¡OK!'],
 ['Hi.', 'Hola.'],
 ['Ah!', '¡Anda!'],
 ['Hi.', '¡Hola!'],
 ['Go!', '¡Ve!'],
 ['Go!', '¡Sal!'],
 ['So?', '¿Y?'],
 ['Go!', '¡Ya!']]
```

```python
eng_sentences = [pair[0] for pair in eng_spa_pairs]
spa_sentences = [pair[1] for pair in eng_spa_pairs]

print(eng_sentences[:10])
print(spa_sentences[:10])
```

```
['Go.', 'No.', 'Ok!', 'Hi.', 'Ah!', 'Hi.', 'Go!', 'Go!', 'So?', 'Go!']
['Ve.', 'No.', '¡OK!', 'Hola.', '¡Anda!', '¡Hola!', '¡Ve!', '¡Sal!',
'¿Y?', '¡Ya!']
```

```python
def preprocess_sentence(sentence):
    """
    Preprocesses the input sentence by:
    1. Converting it to lowercase.
    2. Removing extra spaces and accented characters.
    3. Keeping only lowercase letters.
    4. Adding <sos> and <eos> tokens to the start and end.

    Args:
        sentence (str): The input sentence.

    Returns:
        str: The preprocessed sentence with <sos> and <eos> tokens.
    """
    sentence = sentence.lower().strip()
```

```
    sentence = re.sub(r'[" "]+', " ", sentence)
    sentence = re.sub(r"[á]+", "a", sentence)
    sentence = re.sub(r"[é]+", "e", sentence)
    sentence = re.sub(r"[í]+", "i", sentence)
    sentence = re.sub(r"[ó]+", "o", sentence)
    sentence = re.sub(r"[ú]+", "u", sentence)
    sentence = re.sub(r"[^a-z]+", " ", sentence)
    sentence = sentence.strip()
    sentence = '<sos> ' + sentence + ' <eos>'
    return sentence

s1 = '¿Hola @ cómo estás? 123'

print(s1)
print(preprocess_sentence(s1))
```

```
¿Hola @ cómo estás? 123
<sos> hola como estas <eos>
```

```
eng_sentences = [preprocess_sentence(sentence) for sentence in
eng_sentences]
spa_sentences = [preprocess_sentence(sentence) for sentence in
spa_sentences]

spa_sentences[:10]
```

```
['<sos> ve <eos>',
 '<sos> no <eos>',
 '<sos> ok <eos>',
 '<sos> hola <eos>',
 '<sos> anda <eos>',
 '<sos> hola <eos>',
 '<sos> ve <eos>',
 '<sos> sal <eos>',
 '<sos> y <eos>',
 '<sos> ya <eos>']
```

```
def build_vocab(sentences):
    """
    Builds vocabulary from a list of sentences.

    This function creates two mappings:
    1. `word2idx`: A dictionary mapping words to unique indices
(starting from 2).
    2. `idx2word`: A dictionary mapping indices back to words.

    Special tokens `<pad>` (0) and `<unk>` (1) are added to the
vocabulary.

    Args:
        sentences (list of str): A list of sentences (each sentence is
```

```
a string).

    Returns:
        tuple: A tuple containing two dictionaries:
            - word2idx (dict): Mapping from words to indices.
            - idx2word (dict): Mapping from indices to words.
    """
    words = [word for sentence in sentences for word in
sentence.split()]
    word_count = Counter(words)
    sorted_word_counts = sorted(word_count.items(), key=lambda x:x[1],
reverse=True)
    word2idx = {word: idx for idx, (word, _) in
enumerate(sorted_word_counts, 2)}
    word2idx['<pad>'] = 0
    word2idx['<unk>'] = 1
    idx2word = {idx: word for word, idx in word2idx.items()}
    return word2idx, idx2word

eng_word2idx, eng_idx2word = build_vocab(eng_sentences)
spa_word2idx, spa_idx2word = build_vocab(spa_sentences)
eng_vocab_size = len(eng_word2idx)
spa_vocab_size = len(spa_word2idx)

print(eng_vocab_size, spa_vocab_size)
```

```
27934 47343
```

```
class EngSpaDataset(Dataset):
    """
    A dataset class for English-Spanish sentence pairs.

    This class is used to handle a collection of English and Spanish
sentence pairs,
    and provides methods to retrieve the sentences as index sequences
based on the
    provided word-to-index mappings for both languages.

    Args:
        eng_sentences (list of str): A list of English sentences.
        spa_sentences (list of str): A list of Spanish sentences.
        eng_word2idx (dict): A dictionary mapping English words to
indices.
        spa_word2idx (dict): A dictionary mapping Spanish words to
indices.
    """
    def __init__(self, eng_sentences, spa_sentences, eng_word2idx,
spa_word2idx):
        self.eng_sentences = eng_sentences
        self.spa_sentences = spa_sentences
```

```
        self.eng_word2idx = eng_word2idx
        self.spa_word2idx = spa_word2idx

    def __len__(self):
        """
        Returns the total number of sentence pairs in the dataset.

        Returns:
            int: The number of English-Spanish sentence pairs.
        """
        return len(self.eng_sentences)

    def __getitem__(self, idx):
        """
        Retrieves the English and Spanish sentences at the specified
index as token indices.

        This method splits the sentences into words, looks up their
corresponding indices in
        the word-to-index mappings, and returns the indices as
tensors.

        Args:
            idx (int): The index of the sentence pair to retrieve.

        Returns:
            tuple: A tuple of two tensors:
                - The first tensor contains the English sentence as
token indices.
                - The second tensor contains the Spanish sentence as
token indices.
        """
        eng_sentence = self.eng_sentences[idx]
        spa_sentence = self.spa_sentences[idx]

        eng_idxs = [self.eng_word2idx.get(word,
self.eng_word2idx['<unk>']) for word in eng_sentence.split()]
        spa_idxs = [self.spa_word2idx.get(word,
self.spa_word2idx['<unk>']) for word in spa_sentence.split()]

        return torch.tensor(eng_idxs), torch.tensor(spa_idxs)

def collate_fn(batch):
    """
    Pads and prepares a batch of English-Spanish sentence pairs.

    Args:
        batch (list of tuples): A list of sentence pairs, where each
tuple contains:
            - English sentence as tensor of word indices.
```

```
            - Spanish sentence as tensor of word indices.

    Returns:
        tuple: Two tensors containing padded English and Spanish
sentences.
    """
    eng_batch, spa_batch = zip(*batch)
    eng_batch = [seq[:MAX_SEQ_LEN].clone().detach() for seq in
eng_batch]
    spa_batch = [seq[:MAX_SEQ_LEN].clone().detach() for seq in
spa_batch]
    eng_batch = torch.nn.utils.rnn.pad_sequence(eng_batch,
batch_first=True, padding_value=0)
    spa_batch = torch.nn.utils.rnn.pad_sequence(spa_batch,
batch_first=True, padding_value=0)
    return eng_batch, spa_batch
```

Training the Transformer

We train the model using Cross-Entropy Loss and Adam Optimizer. The model will learn to translate from English to Spanish.

```
def train(model, dataloader, loss_function, optimiser, epochs):
    """
    Trains the model for a specified number of epochs.

    Args:
        model (nn.Module): The model to be trained.
        dataloader (DataLoader): The DataLoader providing batches of
data.
        loss_function (callable): The loss function used to calculate
loss.
        optimiser (Optimizer): The optimizer used to update model
parameters.
        epochs (int): The number of epochs to train the model.

    Prints:
        The average loss for each epoch.
    """
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for i, (eng_batch, spa_batch) in enumerate(dataloader):
            eng_batch = eng_batch.to(device)
            spa_batch = spa_batch.to(device)

            target_input = spa_batch[:, :-1]
            target_output = spa_batch[:, 1:].contiguous().view(-1)
```

```
            optimiser.zero_grad()

            output = model(eng_batch, target_input)
            output = output.view(-1, output.size(-1))

            loss = loss_function(output, target_output)

            loss.backward()
            optimiser.step()
            total_loss += loss.item()

        avg_loss = total_loss/len(dataloader)
        print(f'Epoch: {epoch + 1}/{epochs}, Loss: {avg_loss:.4f}')

BATCH_SIZE = 64
dataset = EngSpaDataset(eng_sentences, spa_sentences, eng_word2idx,
spa_word2idx)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True,
collate_fn=collate_fn)

model = Transformer(d_model=512, num_heads=8, d_ff=2048, num_layers=6,
                    input_vocab_size=eng_vocab_size,
target_vocab_size=spa_vocab_size,
                    max_len=MAX_SEQ_LEN, dropout=0.1)

model = model.to(device)
loss_function = nn.CrossEntropyLoss(ignore_index=0)
optimiser = optim.Adam(model.parameters(), lr=0.0001)

train(model, dataloader, loss_function, optimiser, epochs = 10)
```

```
Epoch: 1/10, Loss: 3.5813
Epoch: 2/10, Loss: 2.1919
Epoch: 3/10, Loss: 1.6944
Epoch: 4/10, Loss: 1.3691
Epoch: 5/10, Loss: 1.1197
Epoch: 6/10, Loss: 0.9188
Epoch: 7/10, Loss: 0.7543
Epoch: 8/10, Loss: 0.6278
Epoch: 9/10, Loss: 0.5335
Epoch: 10/10, Loss: 0.4657
```

Evaluating Translations

We test the model with a few English sentences and check the quality of the translations.

```
def sentence_to_indices(sentence, word2idx):
    """
    Converts a sentence into a list of word indices.
```

```
    Args:
        sentence (str): The input sentence.
        word2idx (dict): A dictionary mapping words to indices.

    Returns:
        list: A list of indices corresponding to the words in the
sentence.
    """
    return [word2idx.get(word, word2idx['<unk>']) for word in
sentence.split()]

def indices_to_sentence(indices, idx2word):
    """
    Converts a list of indices back into a sentence.

    Args:
        indices (list): A list of word indices.
        idx2word (dict): A dictionary mapping indices to words.

    Returns:
        str: The reconstructed sentence from the indices.
    """
    return ' '.join([idx2word[idx] for idx in indices if idx in
idx2word and idx2word[idx] != '<pad>'])

def translate_sentence(model, sentence, eng_word2idx, spa_idx2word,
max_len=MAX_SEQ_LEN, device='cpu'):
    """
    Translates a sentence from English to Spanish using the trained
model.

    Args:
        model (nn.Module): The trained translation model.
        sentence (str): The English sentence to translate.
        eng_word2idx (dict): A dictionary mapping English words to
indices.
        spa_idx2word (dict): A dictionary mapping Spanish indices to
words.
        max_len (int, optional): The maximum length of the translated
sentence. Default is MAX_SEQ_LEN.
        device (str, optional): The device to run the model on (e.g.,
'cpu' or 'cuda'). Default is 'cpu'.

    Returns:
        str: The translated Spanish sentence.
    """
    model.eval()
    sentence = preprocess_sentence(sentence)
    input_indices = sentence_to_indices(sentence, eng_word2idx)
```

```
    input_tensor = torch.tensor(input_indices).unsqueeze(0).to(device)

    # Initialize the target tensor with <sos> token
    tgt_indices = [spa_word2idx['<sos>']]
    tgt_tensor = torch.tensor(tgt_indices).unsqueeze(0).to(device)

    with torch.no_grad():
        for _ in range(max_len):
            output = model(input_tensor, tgt_tensor)
            output = output.squeeze(0)
            next_token = output.argmax(dim=-1)[-1].item()
            tgt_indices.append(next_token)
            tgt_tensor =
torch.tensor(tgt_indices).unsqueeze(0).to(device)
            if next_token == spa_word2idx['<eos>']:
                break

    return indices_to_sentence(tgt_indices, spa_idx2word)

def evaluate_translations(model, sentences, eng_word2idx,
spa_idx2word, max_len=MAX_SEQ_LEN, device='cpu'):
  """
  Translates and prints sentences.

  Args:
      model (nn.Module): The trained model.
      sentences (list of str): List of sentences to translate.
      eng_word2idx (dict): English word-to-index dictionary.
      spa_idx2word (dict): Spanish index-to-word dictionary.
      max_len (int, optional): Max length of translated sentences.
Default is MAX_SEQ_LEN.
      device (str, optional): Device to run the model on. Default is
'cpu'.
  """
  for sentence in sentences:
    translation = translate_sentence(model, sentence, eng_word2idx,
spa_idx2word, max_len, device)
    print(f'Input sentence: {sentence}')
    print(f'Traducción: {translation}')
    print()

# Example sentences to test the translator
test_sentences = [
    "What happns with words tht are not properly writtn?",
    "She plays the piano very well.",
    "We are going to the beach tomorrow.",
    "Can you help me, please?",
    "This book is really interesting.",
    "They traveled to Spain last year.",
    "I have never been to Japan.",
```

```
    "What time is the meeting?",
    "My favorite color is blue.",
    "The cat is sleeping on the couch.",
]

model = model.to(device)
evaluate_translations(model, test_sentences, eng_word2idx,
spa_idx2word, max_len=MAX_SEQ_LEN, device=device)
```

```
Input sentence: What happns with words tht are not properly writtn?
Traducción: <sos> las palabras no se ven con que mira las palabras
<eos>

Input sentence: She plays the piano very well.
Traducción: <sos> ella toca el piano muy bien <eos>

Input sentence: We are going to the beach tomorrow.
Traducción: <sos> ma ana nos vamos a la playa <eos>

Input sentence: Can you help me, please?
Traducción: <sos> puede ayudarme por favor <eos>

Input sentence: This book is really interesting.
Traducción: <sos> este libro es realmente interesante <eos>

Input sentence: They traveled to Spain last year.
Traducción: <sos> el a o pasado viajo a espa a <eos>

Input sentence: I have never been to Japan.
Traducción: <sos> no he estado nunca en japon <eos>

Input sentence: What time is the meeting?
Traducción: <sos> a que hora es la reunion <eos>

Input sentence: My favorite color is blue.
Traducción: <sos> el azul que me encanta es mi color favorito <eos>

Input sentence: The cat is sleeping on the couch.
Traducción: <sos> el gato esta durmiendo en el sofa <eos>
```

Conclusion

The Transformer model for English-to-Spanish translation demonstrated promising potential, albeit with notable limitations due to the constrained dataset and compute power. The extensive training duration of 778 minutes underscored the computational demands of the architecture, showcasing that the model has aptitude for basic translations but struggled with complex structures, revealing signs of overfitting. Additionally, the model displayed difficulties in handling improperly written input, as seen in the following example:

Input sentence: What happens with words that are not properly written?
Traducción: las palabras no se ven con que mira las palabras

This translation is not proper, illustrating the model's challenges when encountering misspelled or poorly structured input, leading to incorrect or incomplete outputs.

Some important points to consider:

- The importance of efficient hardware to mitigate prolonged training times.
- The value of diverse datasets to prevent overfitting.
- The potential of utilizing pre-trained models and fine-tuning on more extensive datasets.
- The benefits of harnessing hardware accelerators to enhance translation quality and reduce training time.

This highlights the importance of efficient hardware to mitigate prolonged training times and the value of diverse datasets to prevent overfitting. Strategies such as utilizing pre-trained models, fine-tuning on more extensive datasets, and harnessing hardware accelerators could significantly enhance translation quality, reduce training time, and enable the model to generalize better across various linguistic contexts, including cases of misspelled or incomplete words.