Sequence-to-Sequence Model with LSTM for English-Spanish Translation

Aryan Gahlaut (2K22/CO/102)

April 13, 2025

1 Introduction

This document provides a detailed explanation of the code used to implement a sequence-to-sequence (seq2seq) model for English-Spanish translation using Long Short-Term Memory (LSTM) layers. The model is trained using teacher forcing, with word embeddings for input tokens and BLEU score for evaluation.

The model achieved a BLEU score of 72.17 after training for 80 epochs, with the final training loss of 0.4022.

2 Libraries Used

The following libraries were used in the implementation:

- **PyTorch** - For building and training the neural network model.
- **Torchtext** - For handling text data, tokenization, and vocabulary creation.
- **NLTK** - For calculating the BLEU score during evaluation.
- **NumPy** - For numerical operations.
- **Random** - For random operations, particularly teacher forcing.
- **sklearn.model_selection.train_test_split** - For splitting the dataset into training and validation sets.

3 Code Explanation

The following sections provide a detailed explanation of the code, broken down into key parts.

3.1 Data Preparation

The dataset contains parallel sentences in English and Spanish. The data is read, cleaned, tokenized, and split into training and validation sets.

```
def read_data(file_path):
    with open(file_path, encoding='utf-8') as f:
        lines = f.read().strip().split('\n')
    pairs = [line.split('\t') for line in lines]
    return pairs

def tokenize(text):
    text = text.lower()
    text = re.sub(r"[^a-zA-Zñáéíóúüç¿¡ ]+", "", text)
    return text.strip().split()
```

The `read_data` function reads the dataset from a tab-separated file, where each line consists of an English-Spanish sentence pair. The `tokenize` function converts the text to lowercase, removes punctuation, and splits it into individual words (tokens).

3.2 Vocabulary Creation

The vocabulary class `Vocab` is used to create a mapping from words to indices and vice versa. It also handles padding, start-of-sequence (`<sos>`), end-of-sequence (`<eos>`), and unknown words (`<unk>`).

```
class Vocab:
    def __init__(self, sentences, min_freq=2):
        self.freq = Counter()
        for sentence in sentences:
            self.freq.update(sentence)

        self.pad = '<pad>'
```

```
        self.sos = '<sos>'
        self.eos = '<eos>'
        self.unk = '<unk>'

        self.itos = [self.pad, self.sos, self.eos, self.unk] + [w for w, c in self
        self.stoi = {w: i for i, w in enumerate(self.itos)}

    def numericalize(self, tokens):
        return [self.stoi.get(token, self.stoi[self.unk]) for token in tokens]

    def denumericalize(self, indices):
        return [self.itos[i] for i in indices if i not in (self.stoi[self.pad],)]
```

The `Vocab` class constructs the vocabulary from the input sentences, assigns each word an index, and provides methods to convert between words and indices.

3.3 Dataset and DataLoader

The `TranslationDataset` class prepares the data for training by converting sentences into numerical tokens. The sentences are padded to the same length using `collate_fn`.

```
class TranslationDataset(Dataset):
    def __init__(self, pairs, src_vocab, trg_vocab):
        self.data = []
        for src, trg in pairs:
            src_tokens = tokenize(src)
            trg_tokens = tokenize(trg)
            src_ids = src_vocab.numericalize(src_tokens)
            trg_ids = [trg_vocab.stoi['<sos>']] + trg_vocab.numericalize(trg_token
            self.data.append((src_ids, trg_ids))

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

    def __getitem__(self, idx):
        return self.data[idx]
```

```
def collate_fn(batch):
    src_batch, trg_batch = zip(*batch)
    src_max_len = max(len(x) for x in src_batch)
    trg_max_len = max(len(x) for x in trg_batch)

    src_batch_padded = [x + [src_vocab.stoi['<pad>']] * (src_max_len - len(x)) for
    trg_batch_padded = [x + [trg_vocab.stoi['<pad>']] * (trg_max_len - len(x)) for

    return torch.tensor(src_batch_padded), torch.tensor(trg_batch_padded)
```

The `TranslationDataset` class converts the English and Spanish sentences into numerical indices. The `collate_fn` function handles padding for sentences of different lengths.

3.4 Seq2Seq Model

The sequence-to-sequence model is composed of an encoder and a decoder. Both components use LSTM layers for encoding and generating the translations.

```
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, hid_dim):
        super().__init__()
        self.embedding = nn.Embedding(input_dim, emb_dim)
        self.lstm = nn.LSTM(emb_dim, hid_dim, batch_first=True)

    def forward(self, src):
        embedded = self.embedding(src)
        outputs, (hidden, cell) = self.lstm(embedded)
        return hidden, cell
```

The `Encoder` class embeds the input tokens and processes them through an LSTM to produce hidden states and cell states, which are passed to the decoder.

```
class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, hid_dim):
        super().__init__()
```

```
        self.embedding = nn.Embedding(output_dim, emb_dim)
        self.lstm = nn.LSTM(emb_dim, hid_dim, batch_first=True)
        self.fc = nn.Linear(hid_dim, output_dim)

    def forward(self, input, hidden, cell):
        input = input.unsqueeze(1)
        embedded = self.embedding(input)
        output, (hidden, cell) = self.lstm(embedded, (hidden, cell))
        prediction = self.fc(output.squeeze(1))
        return prediction, hidden, cell
```

The `Decoder` class generates the output tokens, feeding the predicted token back into the LSTM for the next timestep.

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

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

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

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

        return outputs
```

The `Seq2Seq` class coordinates the interaction between the encoder and

decoder. During training, it uses teacher forcing, where the true target token is passed to the decoder, and the predicted token is used at inference time.

3.5 Training Function

The training function updates the model parameters by minimizing the loss (cross-entropy loss) using backpropagation.

```
def train(model, iterator, optimizer, criterion, clip=1):
    model.train()
    total_loss = 0

    for src, trg in iterator:
        src, trg = src.to(device), trg.to(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()

        total_loss += loss.item()

    return total_loss / len(iterator)
```

3.6 Evaluation with BLEU Score

After training, the model is evaluated using the BLEU score, which is calculated by comparing the predicted translations with the reference translations.

```
def evaluate_bleu(model, dataset, src_vocab, trg_vocab, max_len=20):
```

```
model.eval()
smoothie = SmoothingFunction().method4
total_score = 0
count = 0

with torch.no_grad():
    for src, trg in dataset:
        src_tensor = torch.tensor([src]).to(device)
        hidden, cell = model.encoder(src_tensor)

        input_token = torch.tensor([trg_vocab.stoi['<sos>']]).to(device)
        result = []

        for _ in range(max_len):
            output, hidden, cell = model.decoder(input_token, hidden, cell)
            top1 = output.argmax(1)
            if top1.item() == trg_vocab.stoi['<eos>']:
                break
            result.append(top1.item())
            input_token = top1

        pred_tokens = trg_vocab.denumericalize(result)
        reference_tokens = trg_vocab.denumericalize(trg[1:-1])

        score = sentence_bleu([reference_tokens], pred_tokens, smoothing_funct
        total_score += score
        count += 1

return total_score / count
```

3.7 Results

After training the model for 80 epochs, the following results were obtained:

- **BLEU Score:** 72.17
- **Final Training Loss:** 0.4022
- **Example Translation:** The translation of `"hello!"` is `"hola"`.

4 Conclusion

This document explains the implementation of a sequence-to-sequence translation model using LSTM. The model was trained on English-Spanish sentence pairs and achieved a BLEU score of 72.17 after 80 epochs. The final training loss was 0.4022. An example translation produced by the model is: *"hello!"* $\rightarrow$ *"hola"*. Future improvements could include using pre-trained embeddings, incorporating attention mechanisms, and training on a larger dataset.