Sequence-to-Sequence Modeling with Attention Mechanism

1.Installing Dependencies

```
!pip install torch torchvision
```

This command installs PyTorch and torchvision, which are essential for building and training deep learning models in this notebook.

2. Imports and Device Setup

```
import torch

import torch.nn as nn

import torch.optim as optim

import random

import numpy as np

from torch.utils.data import Dataset, DataLoader
```

- **torch, torch.nn, torch.optim**: Core PyTorch libraries for building and training neural networks.
- **random, numpy**: Used for random operations and numerical computations.
- **torch.utils.data.Dataset, DataLoader**: Classes to handle datasets and batch data efficiently during training.

The device variable is set up to use a GPU if available, improving the efficiency of model training.

3. Data Generation

```
def generate_data(num_samples, seq_len, vocab_size):

  data = []

  for _ in range(num_samples):

    src = [random.randint(1, vocab_size-1) for _ in range(seq_len)]

    tgt = src[::-1]  # Reverse the source sequence for the target

    data.append((src, tgt))

  return data
```

- **generate_data()**: Creates synthetic data for training by generating random sequences of integers (representing tokens) and reversing each sequence to create a target.
- **src and tgt**: Each src sequence (input) is paired with tgt, which is the reverse of src.

4. Dataset and DataLoader Classes

```
class Seq2SeqDataset(Dataset):
  def __init__(self, data):
    self.data = data

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

  def __getitem__(self, idx):
    src, tgt = self.data[idx]
    return torch.tensor(src, dtype=torch.long), torch.tensor(tgt, dtype=torch.long)
```

- **Seq2SeqDataset**: Custom dataset class that holds the (src, tgt) pairs, providing methods to return the length and individual samples as tensors, which are required for efficient data handling in PyTorch.

5. Attention Mechanism

```
class Attention(nn.Module):
  def __init__(self, hidden_dim):
    super(Attention, self).__init__()
    self.attention = nn.Linear(hidden_dim * 2, hidden_dim)
    self.v = nn.Parameter(torch.rand(hidden_dim))

  def forward(self, hidden, encoder_outputs):
    ...
```

- Defines an **Attention** layer, calculating attention weights based on the similarity between the decoder's hidden state and the encoder's outputs.
- The **forward method** computes the attention weights, guiding the decoder to focus on relevant parts of the encoder's output at each decoding step.

6. Encoder-Decoder Architecture

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

```
    self.rnn = nn.GRU(emb_dim, hid_dim, num_layers, batch_first=True)

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

- **Encoder**: Encodes input sequences into a context vector.
- **nn.Embedding**: Maps input tokens to dense vectors.
- **nn.GRU**: Processes the embeddings and returns hidden states that summarize the input sequence.

```
class Decoder(nn.Module):
  def __init__(self, output_dim, emb_dim, hid_dim, num_layers, attention):
    super(Decoder, self).__init__()
    self.attention = attention
    self.embedding = nn.Embedding(output_dim, emb_dim)
    self.rnn = nn.GRU(emb_dim + hid_dim, hid_dim, num_layers, batch_first=True)
    self.fc_out = nn.Linear(hid_dim * 2, output_dim)

  def forward(self, x, hidden, encoder_outputs):
    embedded = self.embedding(x).unsqueeze(1)
    attn_weights = self.attention(hidden[-1], encoder_outputs)
    rnn_input = torch.cat((embedded, attn_weights), dim=2)
    output, hidden = self.rnn(rnn_input, hidden.unsqueeze(0))
    prediction = self.fc_out(torch.cat((output, attn_weights), dim=2))
    return prediction, hidden
```

- **Decoder**: Uses encoder outputs and hidden states to generate the output sequence, focusing on relevant encoder outputs based on the attention weights.

7. Seq2Seq Model (Combining Encoder and Decoder)

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

```
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, src, trg):
        encoder_outputs, hidden = self.encoder(src)
        outputs = []
        for t in range(trg.size(1)):
            output, hidden = self.decoder(trg[:, t], hidden, encoder_outputs)
            outputs.append(output)
        return torch.stack(outputs, dim=1)
```

- Combines the encoder and decoder, passing the encoder's hidden states to initialize the decoder, then iterating over each token in the target sequence for prediction.

8. Training Process

```
def train(model, data_loader, optimizer, criterion):
    model.train()
    epoch_loss = 0
    for src, trg in data_loader:
        optimizer.zero_grad()
        output = model(src, trg)
        loss = criterion(output, trg)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
    return epoch_loss / len(data_loader)
```

- **train()**: Handles model training, computing the loss, and updating weights through backpropagation.
- **optimizer.zero_grad()**: Resets gradients.
- **loss.backward()**: Backpropagation.
- **optimizer.step()**: Updates weights.

9. Evaluation

```
def evaluate(model, data_loader, criterion):
    model.eval()
```

```
    epoch_loss = 0
    with torch.no_grad():
        for src, trg in data_loader:
            output = model(src, trg)
            loss = criterion(output, trg)
            epoch_loss += loss.item()
    return epoch_loss / len(data_loader)
```

- **evaluate()**: Calculates the model's performance on validation/test data by running predictions without gradients to save memory and speed up computations.

10. Inference (Making Predictions)

```
def translate_sentence(model, sentence, tokenizer):
    tokens = preprocess(sentence)
    output = model(tokens, trg=None)
    return output.argmax(dim=-1)
```

- **translate_sentence()**: Makes predictions by translating input sequences into output sequences based on model output. output.argmax(dim=-1) converts predictions into predicted tokens.

11. Saving and Loading the Model

```
torch.save(model.state_dict(), 'seq2seq_model.pt')
model.load_state_dict(torch.load('seq2seq_model.pt'))
```

- Saves the trained model’s state, allowing it to be reloaded later for further inference or evaluation.