Name: **B. SAI CHARAN**

Roll No: **2203A51L72**

Batch No: **21CSBTB12**

ASSIGNMENT – 9

Question:

Translate simple sentences by training neural machine models

Answer:

Training a Neural Machine Translation (NMT) model for translating simple sentences involves several key steps. Here's an outline of the process, focusing on building a basic sequence-to-sequence (seq2seq) model with attention, using encoder-decoder architecture.

1. Prepare the Dataset

- **Collect Parallel Data**: The first step is to gather parallel sentences in the source and target languages. For instance, a simple English-to-French dataset might contain sentences like "Hello" → "Bonjour" and "Thank you" → "Merci."
- **Preprocess the Data**: Tokenize and clean the text (remove punctuation, convert to lowercase). You may also use subword tokenization (like Byte-Pair Encoding or SentencePiece) to handle out-of-vocabulary words and create a shared vocabulary.

2. Define the Model Architecture

- **Encoder**: Processes the input sentence from the source language and generates a context vector (a set of hidden states). For example, in translating "How are you?" to French, the encoder will create a representation of this sentence.
- **Decoder**: Takes the context from the encoder and generates the target sentence word by word. It predicts each word based on previous words in the target sentence and the context from the encoder.
- **Attention Mechanism**: Allows the model to focus on relevant parts of the input sentence for each word in the output. For instance, when translating "How are you?" to "Comment ça va?", the model should focus on "How" while generating "Comment."

3. Training the Model

- **Data Loading**: Divide your data into training, validation, and test sets.
- **Define the Loss Function**: Cross-entropy loss is typically used for sequence-to-sequence models, as it penalizes incorrect predictions at each timestep.
- **Optimization**: Use optimizers like Adam with learning rate scheduling to adjust the learning rate over time.
- **Teacher Forcing**: During training, the model is given the correct previous word as input when predicting the next word, which speeds up convergence and helps avoid getting stuck in repetitive sequences.

4. Inference and Translation

During inference (or testing), the model generates translations without the actual target sentence. It generates words one at a time, feeding the previously generated word back into the model until it generates an end-of-sequence token.

- **Greedy Search**: Selects the word with the highest probability at each step.
- **Beam Search**: Explores multiple translation paths, keeping the top-scoring translations and selecting the one with the highest probability. This improves translation quality but requires more computational resources.

5. Evaluate the Model

Use metrics like BLEU (Bilingual Evaluation Understudy) to compare generated translations with reference translations. BLEU measures how many words or n-grams in the generated translation match those in the reference translation.

6. Improving the Model

- **Data Augmentation**: Use back-translation, where sentences in the target language are translated to the source language, then used as additional training data.
- **Hyperparameter Tuning**: Adjust the number of layers, hidden size, dropout, and learning rate to optimize performance.

- **Transformer Models**: Although seq2seq with attention works well, Transformer-based models (like BERT or T5) can achieve better performance with parallel processing and self-attention mechanisms.

CODE:

```python
import torch
import torch.nn as nn

# Encoder definition
class Encoder(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim):
        super(Encoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.gru = nn.GRU(embedding_dim, hidden_dim)

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

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

    def forward(self, hidden, encoder_outputs):
        # Compute attention scores
        attn_scores = torch.tanh(self.attn(torch.cat((hidden, encoder_outputs), dim=2)))
        attn_weights = torch.softmax(attn_scores, dim=1)
        return attn_weights

# Decoder definition
class Decoder(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, attention):
        super(Decoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.gru = nn.GRU(embedding_dim + hidden_dim, hidden_dim)
        self.fc_out = nn.Linear(hidden_dim * 2, vocab_size)
        self.attention = attention

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

```python
# Decoder definition
class Decoder(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, attention):
        super(Decoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.gru = nn.GRU(embedding_dim + hidden_dim, hidden_dim)
        self.fc_out = nn.Linear(hidden_dim * 2, vocab_size)
        self.attention = attention

    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input)
        attn_weights = self.attention(hidden, encoder_outputs)
        context = attn_weights * encoder_outputs
        gru_input = torch.cat((embedded, context), dim=2)
        output, hidden = self.gru(gru_input, hidden)
        output = self.fc_out(output)
        return output, hidden
```