homework4

March 21, 2025

[3]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.colors import LinearSegmentedColormap
# Device configuration
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

N_EPOCHS = 45
```

```
cuda
```

1. Develop a GRU-based encoder-decoder architecture for English to French Translation. Train the model on the entire dataset and evaluate it on the entire dataset. Report training loss, validation loss, and validation accuracy. Also, try some qualitative validation as well, asking the network to generate French translations for some English sentences.

[4]:
```
# Eglish to French translation

# English to French translation dataset
english_to_french = [
    ("I am cold", "J'ai froid"),
    ("You are tired", "Tu es fatigué"),
    ("He is hungry", "Il a faim"),
    ("She is happy", "Elle est heureuse"),
    ("We are friends", "Nous sommes amis"),
    ("They are students", "Ils sont étudiants"),
    ("The cat is sleeping", "Le chat dort"),
    ("The sun is shining", "Le soleil brille"),
    ("We love music", "Nous aimons la musique"),
    ("She speaks French fluently", "Elle parle français couramment"),
    ("He enjoys reading books", "Il aime lire des livres"),
    ("They play soccer every weekend", "Ils jouent au football chaque␣
 ↪week-end"),
    ("The movie starts at 7 PM", "Le film commence à 19 heures"),
```

```
    ("She wears a red dress", "Elle porte une robe rouge"),
    ("We cook dinner together", "Nous cuisinons le dîner ensemble"),
    ("He drives a blue car", "Il conduit une voiture bleue"),
    ("They visit museums often", "Ils visitent souvent des musées"),
    ("The restaurant serves delicious food", "Le restaurant sert une délicieuse␣
↪cuisine"),
    ("She studies mathematics at university", "Elle étudie les mathématiques à␣
↪l'université"),
    ("We watch movies on Fridays", "Nous regardons des films le vendredi"),
    ("He listens to music while jogging", "Il écoute de la musique en faisant␣
↪du jogging"),
    ("They travel around the world", "Ils voyagent autour du monde"),
    ("The book is on the table", "Le livre est sur la table"),
    ("She dances gracefully", "Elle danse avec grâce"),
    ("We celebrate birthdays with cake", "Nous célébrons les anniversaires avec␣
↪un gâteau"),
    ("He works hard every day", "Il travaille dur tous les jours"),
    ("They speak different languages", "Ils parlent différentes langues"),
    ("The flowers bloom in spring", "Les fleurs fleurissent au printemps"),
    ("She writes poetry in her free time", "Elle écrit de la poésie pendant son␣
↪temps libre"),
    ("We learn something new every day", "Nous apprenons quelque chose de␣
↪nouveau chaque jour"),
    ("The dog barks loudly", "Le chien aboie bruyamment"),
    ("He sings beautifully", "Il chante magnifiquement"),
    ("They swim in the pool", "Ils nagent dans la piscine"),
    ("The birds chirp in the morning", "Les oiseaux gazouillent le matin"),
    ("She teaches English at school", "Elle enseigne l'anglais à l'école"),
    ("We eat breakfast together", "Nous prenons le petit déjeuner ensemble"),
    ("He paints landscapes", "Il peint des paysages"),
    ("They laugh at the joke", "Ils rient de la blague"),
    ("The clock ticks loudly", "L'horloge tic-tac bruyamment"),
    ("She runs in the park", "Elle court dans le parc"),
    ("We travel by train", "Nous voyageons en train"),
    ("He writes a letter", "Il écrit une lettre"),
    ("They read books at the library", "Ils lisent des livres à la␣
↪bibliothèque"),
    ("The baby cries", "Le bébé pleure"),
    ("She studies hard for exams", "Elle étudie dur pour les examens"),
    ("We plant flowers in the garden", "Nous plantons des fleurs dans le␣
↪jardin"),
    ("He fixes the car", "Il répare la voiture"),
    ("They drink coffee in the morning", "Ils boivent du café le matin"),
    ("The sun sets in the evening", "Le soleil se couche le soir"),
    ("She dances at the party", "Elle danse à la fête"),
    ("We play music at the concert", "Nous jouons de la musique au concert"),
```

```
    ("He cooks dinner for his family", "Il cuisine le dîner pour sa famille"),
    ("They study French grammar", "Ils étudient la grammaire française"),
    ("The rain falls gently", "La pluie tombe doucement"),
    ("She sings a song", "Elle chante une chanson"),
    ("We watch a movie together", "Nous regardons un film ensemble"),
    ("He sleeps deeply", "Il dort profondément"),
    ("They travel to Paris", "Ils voyagent à Paris"),
    ("The children play in the park", "Les enfants jouent dans le parc"),
    ("She walks along the beach", "Elle se promène le long de la plage"),
    ("We talk on the phone", "Nous parlons au téléphone"),
    ("He waits for the bus", "Il attend le bus"),
    ("They visit the Eiffel Tower", "Ils visitent la tour Eiffel"),
    ("The stars twinkle at night", "Les étoiles scintillent la nuit"),
    ("She dreams of flying", "Elle rêve de voler"),
    ("We work in the office", "Nous travaillons au bureau"),
    ("He studies history", "Il étudie l'histoire"),
    ("They listen to the radio", "Ils écoutent la radio"),
    ("The wind blows gently", "Le vent souffle doucement"),
    ("She swims in the ocean", "Elle nage dans l'océan"),
    ("We dance at the wedding", "Nous dansons au mariage"),
    ("He climbs the mountain", "Il gravit la montagne"),
    ("They hike in the forest", "Ils font de la randonnée dans la forêt"),
    ("The cat meows loudly", "Le chat miaule bruyamment"),
    ("She paints a picture", "Elle peint un tableau"),
    ("We build a sandcastle", "Nous construisons un château de sable"),
    ("He sings in the choir", "Il chante dans le chœur")
]


# Special tokens for the start and end of sequences
SOS_token = 0  # Start Of Sequence Token
EOS_token = 1  # End Of Sequence Token

# Preparing the word to index mapping and vice versa
word_to_index = {"SOS": SOS_token, "EOS": EOS_token}
for pair in english_to_french:
    for word in pair[0].split() + pair[1].split():
        if word not in word_to_index:
            word_to_index[word] = len(word_to_index)

index_to_word = {i: word for word, i in word_to_index.items()}

class TranslationDataset(Dataset):
    """Custom Dataset class for handling translation pairs."""
    def __init__(self, dataset, word_to_index):
        self.dataset = dataset
        self.word_to_index = word_to_index
```

```
    def __len__(self):
        # Returns the total number of translation pairs in the dataset
        return len(self.dataset)

    def __getitem__(self, idx):
        # Retrieves a translation pair by index, converts words to indices,
        # and adds the EOS token at the end of each sentence.
        input_sentence, target_sentence = self.dataset[idx]
        input_indices = [self.word_to_index[word] for word in input_sentence.
 ↪split()] + [EOS_token]
        target_indices = [self.word_to_index[word] for word in target_sentence.
 ↪split()] + [EOS_token]
        return torch.tensor(input_indices, dtype=torch.long), torch.
 ↪tensor(target_indices, dtype=torch.long)

# Creating a DataLoader to batch and shuffle the dataset
translation_dataset = TranslationDataset(english_to_french, word_to_index)
dataloader = DataLoader(translation_dataset, batch_size=1, shuffle=True)

class Encoder(nn.Module):
    """The Encoder part of the seq2seq model."""
    def __init__(self, input_size, hidden_size):
        super(Encoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)  # Embedding␣
 ↪layer
        self.gru = nn.GRU(hidden_size, hidden_size)  # GRU layer

    def forward(self, input, hidden):
        # Forward pass for the encoder
        embedded = self.embedding(input).view(1, 1, -1)
        output, hidden = self.gru(embedded, hidden)
        return output, hidden

    def initHidden(self):
        # Initializes hidden state
        return torch.zeros(1, 1, self.hidden_size, device=device)

class Decoder(nn.Module):
    """The Decoder part of the seq2seq model."""
    def __init__(self, hidden_size, output_size):
        super(Decoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(output_size, hidden_size)  # Embedding␣
 ↪layer
        self.gru = nn.GRU(hidden_size, hidden_size)  # GRU layer
```

```
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output, hidden = self.gru(embedded, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)

# Assuming all words in the dataset + 'SOS' and 'EOS' tokens are included in␣
 ↪word_to_index
input_size = len(word_to_index)
hidden_size = 256  # Adjust according to your preference
output_size = len(word_to_index)

encoder = Encoder(input_size=input_size, hidden_size=hidden_size).to(device)
decoder = Decoder(hidden_size=hidden_size, output_size=output_size).to(device)

# Set the learning rate for optimization
learning_rate = 0.008

# Initializing optimizers for both encoder and decoder with SGD optimizer
encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)

def train(input_tensor, target_tensor, encoder, decoder, encoder_optimizer,␣
 ↪decoder_optimizer, criterion):
    # Initialize encoder hidden state
    encoder_hidden = encoder.initHidden()

    # Clear gradients for optimizers
    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    # Calculate the length of input and target tensors
    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)

    # Initialize loss
    loss = 0

    # Encoding each word in the input
    for ei in range(input_length):
```

```
        encoder_output, encoder_hidden = encoder(input_tensor[ei].unsqueeze(0),␣
↪encoder_hidden)

    # Decoder's first input is the SOS token
    decoder_input = torch.tensor([[SOS_token]], device=device)

    # Decoder starts with the encoder's last hidden state
    decoder_hidden = encoder_hidden

    # Decoding loop
    for di in range(target_length):
        decoder_output, decoder_hidden = decoder(decoder_input, decoder_hidden)
        # Choose top1 word from decoder's output
        topv, topi = decoder_output.topk(1)
        decoder_input = topi.squeeze().detach()  # Detach from history as input

        # Calculate loss
        loss += criterion(decoder_output, target_tensor[di].unsqueeze(0))
        if decoder_input.item() == EOS_token:  # Stop if EOS token is generated
            break

    # Backpropagation
    loss.backward()

    # Update encoder and decoder parameters
    encoder_optimizer.step()
    decoder_optimizer.step()

    # Return average loss
    return loss.item() / target_length

# Negative Log Likelihood Loss function for calculating loss
criterion = nn.NLLLoss()

# Set number of epochs for training
n_epochs = N_EPOCHS

# Lists to store training and validation metrics
train_losses = []
val_losses = []
val_accuracies = []

# Training loop
for epoch in range(n_epochs):
    # Training phase
    encoder.train()
    decoder.train()
```

```
    total_train_loss = 0

    for input_tensor, target_tensor in dataloader:
        # Move tensors to the correct device
        input_tensor = input_tensor[0].to(device)
        target_tensor = target_tensor[0].to(device)

        # Perform a single training step and update total loss
        loss = train(input_tensor, target_tensor, encoder, decoder,␣
↪encoder_optimizer, decoder_optimizer, criterion)
        total_train_loss += loss

    # Calculate average training loss for this epoch
    avg_train_loss = total_train_loss / len(dataloader)
    train_losses.append(avg_train_loss)

    # Validation phase - using the same dataset
    encoder.eval()
    decoder.eval()
    total_val_loss = 0
    correct_predictions = 0

    with torch.no_grad():
        for input_tensor, target_tensor in dataloader:  # Using the same␣
↪dataloader
            # Move tensors to the correct device
            input_tensor = input_tensor[0].to(device)
            target_tensor = target_tensor[0].to(device)

            # Initialize encoder hidden state
            encoder_hidden = encoder.initHidden()

            input_length = input_tensor.size(0)
            target_length = target_tensor.size(0)

            # Encoding step
            for ei in range(input_length):
                encoder_output, encoder_hidden = encoder(input_tensor[ei].
↪unsqueeze(0), encoder_hidden)

            # Decoding step
            decoder_input = torch.tensor([[SOS_token]], device=device)
            decoder_hidden = encoder_hidden

            loss = 0
            predicted_indices = []
```

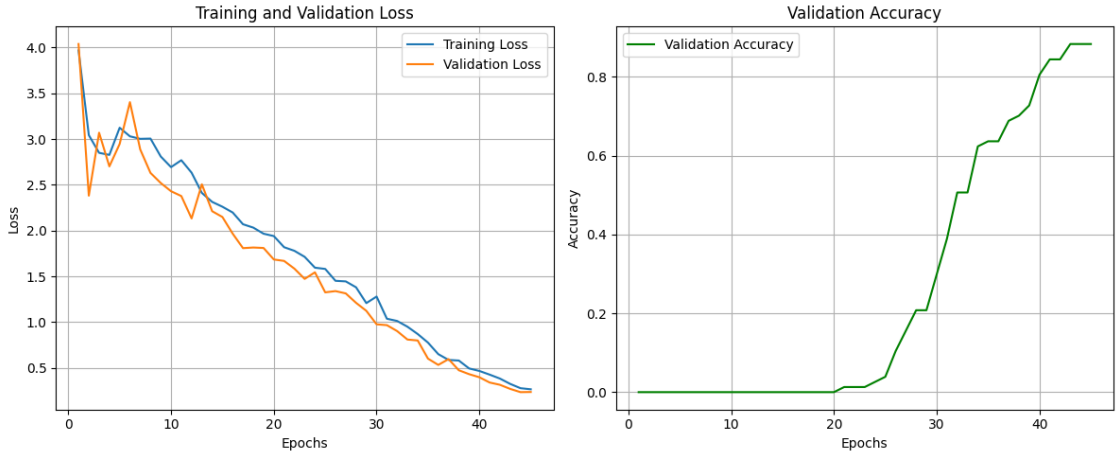
```
            for di in range(target_length):
                decoder_output, decoder_hidden = decoder(decoder_input,␣
 ↪decoder_hidden)
                topv, topi = decoder_output.topk(1)
                predicted_indices.append(topi.item())
                decoder_input = topi.squeeze().detach()

                loss += criterion(decoder_output, target_tensor[di].
 ↪unsqueeze(0))
                if decoder_input.item() == EOS_token:
                    break

            # Calculate validation loss
            total_val_loss += loss.item() / target_length

            # Check if prediction is correct (exact match)
            if predicted_indices == target_tensor.tolist():
                correct_predictions += 1

    # Calculate average validation loss and accuracy for this epoch
    avg_val_loss = total_val_loss / len(dataloader)
    val_losses.append(avg_val_loss)

    val_accuracy = correct_predictions / len(dataloader)
    val_accuracies.append(val_accuracy)

    # Print metrics every 10 epochs
    if epoch % 5 == 0 or epoch == n_epochs - 1:
        print(f'Epoch {epoch+1}/{n_epochs}, Train Loss: {avg_train_loss:.4f},␣
 ↪Val Loss: {avg_val_loss:.4f}, Val Accuracy: {val_accuracy:.4f}')

def evaluate_and_show_examples(encoder, decoder, dataloader, criterion,␣
 ↪n_examples=10):
    # Switch model to evaluation mode
    encoder.eval()
    decoder.eval()

    total_loss = 0
    correct_predictions = 0

    # No gradient calculation
    with torch.no_grad():
        for i, (input_tensor, target_tensor) in enumerate(dataloader):
            # Move tensors to the correct device
            input_tensor = input_tensor[0].to(device)
            target_tensor = target_tensor[0].to(device)
```

```
            encoder_hidden = encoder.initHidden()

            input_length = input_tensor.size(0)
            target_length = target_tensor.size(0)

            loss = 0

            # Encoding step
            for ei in range(input_length):
                encoder_output, encoder_hidden = encoder(input_tensor[ei].
↪unsqueeze(0), encoder_hidden)

            # Decoding step
            decoder_input = torch.tensor([[SOS_token]], device=device)
            decoder_hidden = encoder_hidden

            predicted_indices = []

            for di in range(target_length):
                decoder_output, decoder_hidden = decoder(decoder_input,␣
↪decoder_hidden)
                topv, topi = decoder_output.topk(1)
                predicted_indices.append(topi.item())
                decoder_input = topi.squeeze().detach()

                loss += criterion(decoder_output, target_tensor[di].
↪unsqueeze(0))
                if decoder_input.item() == EOS_token:
                    break

            # Calculate and print loss and accuracy for the evaluation
            total_loss += loss.item() / target_length
            if predicted_indices == target_tensor.tolist():
                correct_predictions += 1

            # Optionally, print some examples
            if i < n_examples:
                predicted_sentence = ' '.join([index_to_word[index] for index␣
↪in predicted_indices if index not in (SOS_token, EOS_token)])
                target_sentence = ' '.join([index_to_word[index.item()] for␣
↪index in target_tensor if index.item() not in (SOS_token, EOS_token)])
                input_sentence = ' '.join([index_to_word[index.item()] for␣
↪index in input_tensor if index.item() not in (SOS_token, EOS_token)])

                print(f'Input: {input_sentence}, Target: {target_sentence},␣
↪Predicted: {predicted_sentence}')
```

```
        # Print overall evaluation results
        average_loss = total_loss / len(dataloader)
        accuracy = correct_predictions / len(dataloader)
        print(f'Evaluation Loss: {average_loss}, Accuracy: {accuracy}')

# Visualize training and validation loss
plt.figure(figsize=(12, 5))

# Plot losses
plt.subplot(1, 2, 1)
plt.plot(range(1, n_epochs + 1), train_losses, label='Training Loss')
plt.plot(range(1, n_epochs + 1), val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.grid(True)

# Plot validation accuracy
plt.subplot(1, 2, 2)
plt.plot(range(1, n_epochs + 1), val_accuracies, label='Validation Accuracy',␣
 ↪color='green')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Validation Accuracy')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.savefig('english2french_training_metrics.png')
plt.show()

# Print final validation metrics
final_val_loss = val_losses[-1]
final_val_accuracy = val_accuracies[-1]
print(f'\nFinal Validation Results:')
print(f'Loss: {final_val_loss:.4f}')
print(f'Accuracy: {final_val_accuracy:.4f}')

# Perform evaluation with examples
print(f'\nExample translations:')
evaluate_and_show_examples(encoder, decoder, dataloader, criterion,␣
 ↪n_examples=5)
```

```
Epoch 1/45, Train Loss: 3.9691, Val Loss: 4.0389, Val Accuracy: 0.0000
Epoch 6/45, Train Loss: 3.0305, Val Loss: 3.4040, Val Accuracy: 0.0000
Epoch 11/45, Train Loss: 2.7695, Val Loss: 2.3778, Val Accuracy: 0.0000
```

```
Epoch 16/45, Train Loss: 2.1981, Val Loss: 1.9667, Val Accuracy: 0.0000
Epoch 21/45, Train Loss: 1.8191, Val Loss: 1.6693, Val Accuracy: 0.0130
Epoch 26/45, Train Loss: 1.4520, Val Loss: 1.3396, Val Accuracy: 0.1039
Epoch 31/45, Train Loss: 1.0376, Val Loss: 0.9675, Val Accuracy: 0.3896
Epoch 36/45, Train Loss: 0.6509, Val Loss: 0.5332, Val Accuracy: 0.6364
Epoch 41/45, Train Loss: 0.4269, Val Loss: 0.3407, Val Accuracy: 0.8442
Epoch 45/45, Train Loss: 0.2668, Val Loss: 0.2373, Val Accuracy: 0.8831
```




```
Final Validation Results:
Loss: 0.2373
Accuracy: 0.8831

Example translations:
Input: He sings beautifully, Target: Il chante magnifiquement, Predicted: Il
chante magnifiquement
Input: We build a sandcastle, Target: Nous construisons un château de sable,
Predicted: Nous construisons un château de sable
Input: He writes a letter, Target: Il écrit une lettre, Predicted: Il écrit une
lettre
Input: We are friends, Target: Nous sommes amis, Predicted: Nous sommes amis
Input: The stars twinkle at night, Target: Les étoiles scintillent la nuit,
Predicted: Les étoiles scintillent la nuit
Evaluation Loss: 0.2372823596476796, Accuracy: 0.8831168831168831
```

2. Repeat problem 1, this time extend the network with attention. Train the model on the entire dataset and evaluate it on the entire dataset. Report training loss, validation loss, and validation accuracy. Also, try some qualitative validation as well, asking the network to generate French translations for some English sentences. Also, compare the results against problem 1.

[5]:
```
# English to French translation with attention.

class TranslationDataset(Dataset):
    """Custom Dataset class for handling translation pairs."""
    def __init__(self, dataset, word_to_index):
        self.dataset = dataset
        self.word_to_index = word_to_index

    def __len__(self):
        # Returns the total number of translation pairs in the dataset
        return len(self.dataset)

    def __getitem__(self, idx):
        # Retrieves a translation pair by index, converts words to indices,
        # and adds the EOS token at the end of each sentence.
        input_sentence, target_sentence = self.dataset[idx]
        input_indices = [self.word_to_index[word] for word in input_sentence.
 ↪split()] + [EOS_token]
        target_indices = [self.word_to_index[word] for word in target_sentence.
 ↪split()] + [EOS_token]
        return torch.tensor(input_indices, dtype=torch.long), torch.
 ↪tensor(target_indices, dtype=torch.long)

# Creating a DataLoader to batch and shuffle the dataset
max_length = 14
translation_dataset = TranslationDataset(english_to_french, word_to_index)
dataloader = DataLoader(translation_dataset, batch_size=1, shuffle=True)

class Encoder(nn.Module):
    """The Encoder part of the seq2seq model."""
    def __init__(self, input_size, hidden_size):
        super(Encoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)  # Embedding␣
 ↪layer
        self.gru = nn.GRU(hidden_size, hidden_size)  # GRU layer

    def forward(self, input, hidden):
        # Forward pass for the encoder
        embedded = self.embedding(input).view(1, 1, -1)
        output, hidden = self.gru(embedded, hidden)
        return output, hidden

    def initHidden(self):
        # Initializes hidden state
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

```
class AttentionDecoder(nn.Module):
    """The Decoder part of the seq2seq model with attention mechanism."""
    def __init__(self, hidden_size, output_size, max_length=14, dropout_p=0.1):
        super(AttentionDecoder, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.dropout_p = dropout_p
        self.max_length = max_length
        self.embedding = nn.Embedding(self.output_size, self.hidden_size)  #␣
 ↪Embedding layer
        self.attn = nn.Linear(self.hidden_size * 2, self.max_length)  #␣
 ↪Attention layer
        self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_size) ␣
 ↪# Combining layer
        self.dropout = nn.Dropout(self.dropout_p)
        self.gru = nn.GRU(self.hidden_size, self.hidden_size)  # GRU layer
        self.out = nn.Linear(self.hidden_size, output_size)  # Output layer

    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)

        # Calculating attention weights
        attn_weights = torch.softmax(
            self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))

        # Combining embedded input with attention output
        output = torch.cat((embedded[0], attn_applied[0]), 1)
        output = self.attn_combine(output).unsqueeze(0)

        output = torch.relu(output)
        output, hidden = self.gru(output, hidden)

        output = torch.log_softmax(self.out(output[0]), dim=1)
        return output, hidden, attn_weights

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)

# Assuming all words in the dataset + 'SOS' and 'EOS' tokens are included in␣
 ↪word_to_index
input_size = len(word_to_index)
hidden_size = 256  # Adjust according to your preference
output_size = len(word_to_index)
```

```
encoder = Encoder(input_size=input_size, hidden_size=hidden_size).to(device)
decoder = AttentionDecoder(hidden_size=hidden_size, output_size=output_size).
 ↪to(device)

# Set the learning rate for optimization
learning_rate = 0.008

# Initializing optimizers for both encoder and decoder with SGD optimizer
encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)

def train(input_tensor, target_tensor, encoder, decoder, encoder_optimizer,␣
 ↪decoder_optimizer, criterion, max_length=14):
    # Initialize encoder hidden state
    encoder_hidden = encoder.initHidden()

    # Clear gradients for optimizers
    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    # Calculate the length of input and target tensors
    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)

    # Initialize loss
    loss = 0

    # Encoding each word in the input
    encoder_outputs = torch.zeros(max_length, encoder.hidden_size,␣
 ↪device=device)
    for ei in range(input_length):
        encoder_output, encoder_hidden = encoder(input_tensor[ei].unsqueeze(0),␣
 ↪encoder_hidden)
        encoder_outputs[ei] = encoder_output[0, 0]

    # Decoder's first input is the SOS token
    decoder_input = torch.tensor([[SOS_token]], device=device)

    # Decoder starts with the encoder's last hidden state
    decoder_hidden = encoder_hidden

    # Decoding loop with attention
    for di in range(target_length):
        decoder_output, decoder_hidden, decoder_attention =␣
 ↪decoder(decoder_input, decoder_hidden, encoder_outputs)
        # Choose top1 word from decoder's output
        topv, topi = decoder_output.topk(1)
```

```
        decoder_input = topi.squeeze().detach()  # Detach from history as input

        # Calculate loss
        loss += criterion(decoder_output, target_tensor[di].unsqueeze(0))
        if decoder_input.item() == EOS_token:  # Stop if EOS token is generated
            break

    # Backpropagation
    loss.backward()

    # Update encoder and decoder parameters
    encoder_optimizer.step()
    decoder_optimizer.step()

    # Return average loss
    return loss.item() / target_length

# Negative Log Likelihood Loss function for calculating loss
criterion = nn.NLLLoss()

def evaluate(encoder, decoder, dataloader, criterion):
    # Switch model to evaluation mode
    encoder.eval()
    decoder.eval()

    total_loss = 0
    correct_predictions = 0

    # No gradient calculation
    with torch.no_grad():
        for input_tensor, target_tensor in dataloader:
            # Move tensors to the correct device
            input_tensor = input_tensor[0].to(device)
            target_tensor = target_tensor[0].to(device)

            encoder_hidden = encoder.initHidden()

            input_length = input_tensor.size(0)
            target_length = target_tensor.size(0)

            loss = 0

            # Encoding step
            encoder_outputs = torch.zeros(max_length, encoder.hidden_size,␣
 ↪device=device)
            for ei in range(input_length):
```

```
                encoder_output, encoder_hidden = encoder(input_tensor[ei].
↪unsqueeze(0), encoder_hidden)
                encoder_outputs[ei] = encoder_output[0, 0]

            # Decoding step
            decoder_input = torch.tensor([[SOS_token]], device=device)
            decoder_hidden = encoder_hidden

            predicted_indices = []

            for di in range(target_length):
                decoder_output, decoder_hidden, decoder_attention =␣
↪decoder(decoder_input, decoder_hidden, encoder_outputs)
                topv, topi = decoder_output.topk(1)
                predicted_indices.append(topi.item())
                decoder_input = topi.squeeze().detach()

                loss += criterion(decoder_output, target_tensor[di].
↪unsqueeze(0))
                if decoder_input.item() == EOS_token:
                    break

            # Calculate loss and check if prediction is correct
            total_loss += loss.item() / target_length
            if predicted_indices == target_tensor.tolist():
                correct_predictions += 1

        # Calculate average loss and accuracy
        average_loss = total_loss / len(dataloader)
        accuracy = correct_predictions / len(dataloader)

        return average_loss, accuracy

# Set number of epochs for training
n_epochs = N_EPOCHS

# Lists to store training and validation losses
train_losses = []
val_losses = []
val_accuracies = []

# Training loop
for epoch in range(n_epochs):
    # Set models to training mode
    encoder.train()
    decoder.train()
```

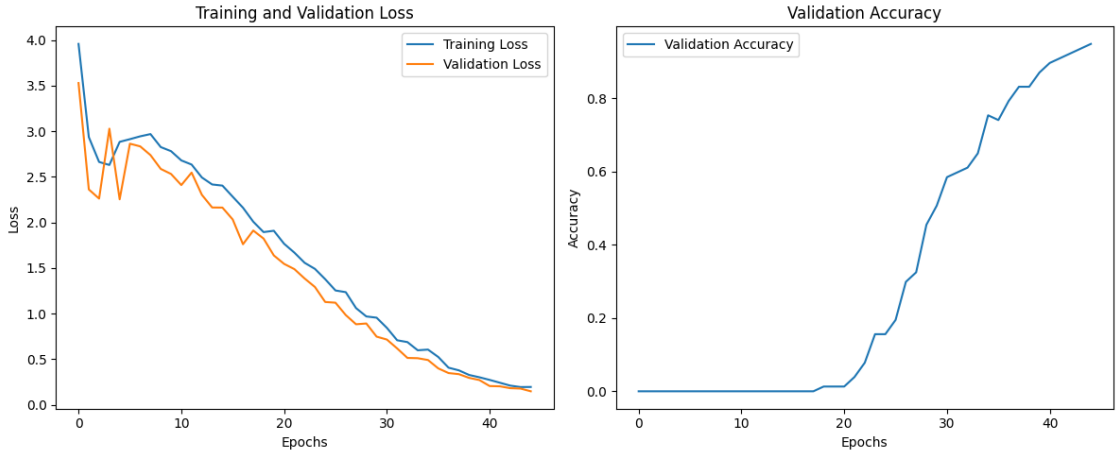
```
    total_loss = 0
    for input_tensor, target_tensor in dataloader:
        # Move tensors to the correct device
        input_tensor = input_tensor[0].to(device)
        target_tensor = target_tensor[0].to(device)

        # Perform a single training step and update total loss
        loss = train(input_tensor, target_tensor, encoder, decoder,␣
↪encoder_optimizer, decoder_optimizer, criterion)
        total_loss += loss

    # Calculate average training loss for this epoch
    avg_train_loss = total_loss / len(dataloader)
    train_losses.append(avg_train_loss)

    # Evaluate on validation set (using the same data for simplicity)
    val_loss, val_accuracy = evaluate(encoder, decoder, dataloader, criterion)
    val_losses.append(val_loss)
    val_accuracies.append(val_accuracy)

    # Print loss every 10 epochs
    if epoch % 5 == 0 or epoch == n_epochs - 1:
       print(f'Epoch {epoch}, Train Loss: {avg_train_loss}, Val Loss:␣
↪{val_loss}, Val Accuracy: {val_accuracy}')



def evaluate_and_show_examples(encoder, decoder, dataloader, criterion,␣
↪n_examples=10):
    # Switch model to evaluation mode
    encoder.eval()
    decoder.eval()

    total_loss = 0
    correct_predictions = 0

    # No gradient calculation
    with torch.no_grad():
        for i, (input_tensor, target_tensor) in enumerate(dataloader):
            # Move tensors to the correct device
            input_tensor = input_tensor[0].to(device)
            target_tensor = target_tensor[0].to(device)

            encoder_hidden = encoder.initHidden()

            input_length = input_tensor.size(0)
            target_length = target_tensor.size(0)
```

```
            loss = 0

            # Encoding step
            encoder_outputs = torch.zeros(max_length, encoder.hidden_size,␣
↪device=device)
            for ei in range(input_length):
                encoder_output, encoder_hidden = encoder(input_tensor[ei].
↪unsqueeze(0), encoder_hidden)
                encoder_outputs[ei] = encoder_output[0, 0]

            # Decoding step
            decoder_input = torch.tensor([[SOS_token]], device=device)
            decoder_hidden = encoder_hidden

            predicted_indices = []

            for di in range(target_length):
                decoder_output, decoder_hidden, decoder_attention =␣
↪decoder(decoder_input, decoder_hidden, encoder_outputs)
                topv, topi = decoder_output.topk(1)
                predicted_indices.append(topi.item())
                decoder_input = topi.squeeze().detach()

                loss += criterion(decoder_output, target_tensor[di].
↪unsqueeze(0))
                if decoder_input.item() == EOS_token:
                    break

            # Calculate and print loss and accuracy for the evaluation
            total_loss += loss.item() / target_length
            if predicted_indices == target_tensor.tolist():
                correct_predictions += 1

            # Optionally, print some examples
            if i < n_examples:
                predicted_sentence = ' '.join([index_to_word[index] for index␣
↪in predicted_indices if index not in (SOS_token, EOS_token)])
                target_sentence = ' '.join([index_to_word[index.item()] for␣
↪index in target_tensor if index.item() not in (SOS_token, EOS_token)])
                input_sentence = ' '.join([index_to_word[index.item()] for␣
↪index in input_tensor if index.item() not in (SOS_token, EOS_token)])

                print(f'Input: {input_sentence}, Target: {target_sentence},␣
↪Predicted: {predicted_sentence}')
```

```
        # Print overall evaluation results
        average_loss = total_loss / len(dataloader)
        accuracy = correct_predictions / len(dataloader)
        print(f'Evaluation Loss: {average_loss}, Accuracy: {accuracy}')

# Visualize training and validation loss
plt.figure(figsize=(12, 5))

# Plot training and validation loss
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()

# Plot validation accuracy
plt.subplot(1, 2, 2)
plt.plot(val_accuracies, label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Validation Accuracy')
plt.legend()

plt.tight_layout()
plt.savefig('training_validation_metrics.png')
plt.show()

# Report final validation accuracy
final_val_loss, final_val_accuracy = val_losses[-1], val_accuracies[-1]
print(f'\nFinal Validation Loss: {final_val_loss:.4f}')
print(f'Final Validation Accuracy: {final_val_accuracy:.4f}␣
 ↪({final_val_accuracy*100:.2f}%)')

# Perform evaluation with examples
print('\nExample translations:')
evaluate_and_show_examples(encoder, decoder, dataloader, criterion)
```

```
Epoch 0, Train Loss: 3.95748347620747, Val Loss: 3.528450883095891, Val
Accuracy: 0.0
Epoch 5, Train Loss: 2.9127448197272217, Val Loss: 2.8629542666792847, Val
Accuracy: 0.0
Epoch 10, Train Loss: 2.6795212492700005, Val Loss: 2.4090172813248816, Val
Accuracy: 0.0
Epoch 15, Train Loss: 2.281403148289785, Val Loss: 2.031969985596251, Val
Accuracy: 0.0
Epoch 20, Train Loss: 1.7654055902378465, Val Loss: 1.5447591183431812, Val
```

```
Accuracy: 0.012987012987012988
Epoch 25, Train Loss: 1.2529798253618103, Val Loss: 1.1189077896264332, Val
Accuracy: 0.19480519480519481
Epoch 30, Train Loss: 0.8429811672233254, Val Loss: 0.7146987077889306, Val
Accuracy: 0.5844155844155844
Epoch 35, Train Loss: 0.5234715990456472, Val Loss: 0.39825946636984866, Val
Accuracy: 0.7402597402597403
Epoch 40, Train Loss: 0.27205875707892774, Val Loss: 0.204722956987453, Val
Accuracy: 0.8961038961038961
Epoch 44, Train Loss: 0.19465378785340393, Val Loss: 0.14717081917533922, Val
Accuracy: 0.948051948051948
```




```
Final Validation Loss: 0.1472
Final Validation Accuracy: 0.9481 (94.81%)

Example translations:
Input: He enjoys reading books, Target: Il aime lire des livres, Predicted: Il
aime lire des livres
Input: The book is on the table, Target: Le livre est sur la table, Predicted:
Le livre est sur la table
Input: They read books at the library, Target: Ils lisent des livres à la
bibliothèque, Predicted: Ils lisent des livres à la bibliothèque
Input: She writes poetry in her free time, Target: Elle écrit de la poésie
pendant son temps libre, Predicted: Elle écrit de la poésie pendant
Input: He climbs the mountain, Target: Il gravit la montagne, Predicted: Il
gravit la montagne
Input: She dances gracefully, Target: Elle danse avec grâce, Predicted: Elle
danse avec grâce
Input: She paints a picture, Target: Elle peint un tableau, Predicted: Elle
peint un tableau
Input: We dance at the wedding, Target: Nous dansons au mariage, Predicted: Nous
```

```
dansons au mariage
Input: We play music at the concert, Target: Nous jouons de la musique au
concert, Predicted: Nous jouons de la musique au concert
Input: They are students, Target: Ils sont étudiants, Predicted: Ils sont
étudiants
Evaluation Loss: 0.14717081917533925, Accuracy: 0.948051948051948
```

3. Repeat problems 1 and 2, this time try to translate from French to English. Train the model on the entire dataset and evaluate it on the entire dataset. Report training loss, validation loss, and validation accuracy. Also, try some qualitative validation as well, asking the network to generate English translations for some French sentences. Which one seems to be more effective, French-to-English or English-to-French?

[6]:
```
#French to English

# French to English translation dataset
french_to_english = [
    ("J'ai froid", "I am cold"),
    ("Tu es fatigué", "You are tired"),
    ("Il a faim", "He is hungry"),
    ("Elle est heureuse", "She is happy"),
    ("Nous sommes amis", "We are friends"),
    ("Ils sont étudiants", "They are students"),
    ("Le chat dort", "The cat is sleeping"),
    ("Le soleil brille", "The sun is shining"),
    ("Nous aimons la musique", "We love music"),
    ("Elle parle français couramment", "She speaks French fluently"),
    ("Il aime lire des livres", "He enjoys reading books"),
    ("Ils jouent au football chaque week-end", "They play soccer every␣
 ↪weekend"),
    ("Le film commence à 19 heures", "The movie starts at 7 PM"),
    ("Elle porte une robe rouge", "She wears a red dress"),
    ("Nous cuisinons le dîner ensemble", "We cook dinner together"),
    ("Il conduit une voiture bleue", "He drives a blue car"),
    ("Ils visitent souvent des musées", "They visit museums often"),
    ("Le restaurant sert une délicieuse cuisine", "The restaurant serves␣
 ↪delicious food"),
    ("Elle étudie les mathématiques à l'université", "She studies mathematics␣
 ↪at university"),
    ("Nous regardons des films le vendredi", "We watch movies on Fridays"),
    ("Il écoute de la musique en faisant du jogging", "He listens to music␣
 ↪while jogging"),
    ("Ils voyagent autour du monde", "They travel around the world"),
    ("Le livre est sur la table", "The book is on the table"),
    ("Elle danse avec grâce", "She dances gracefully"),
    ("Nous célébrons les anniversaires avec un gâteau", "We celebrate birthdays␣
 ↪with cake"),
```

```
    ("Il travaille dur tous les jours", "He works hard every day"),
    ("Ils parlent différentes langues", "They speak different languages"),
    ("Les fleurs fleurissent au printemps", "The flowers bloom in spring"),
    ("Elle écrit de la poésie pendant son temps libre", "She writes poetry in␣
↪her free time"),
    ("Nous apprenons quelque chose de nouveau chaque jour", "We learn something␣
↪new every day"),
    ("Le chien aboie bruyamment", "The dog barks loudly"),
    ("Il chante magnifiquement", "He sings beautifully"),
    ("Ils nagent dans la piscine", "They swim in the pool"),
    ("Les oiseaux gazouillent le matin", "The birds chirp in the morning"),
    ("Elle enseigne l'anglais à l'école", "She teaches English at school"),
    ("Nous prenons le petit déjeuner ensemble", "We eat breakfast together"),
    ("Il peint des paysages", "He paints landscapes"),
    ("Ils rient de la blague", "They laugh at the joke"),
    ("L'horloge tic-tac bruyamment", "The clock ticks loudly"),
    ("Elle court dans le parc", "She runs in the park"),
    ("Nous voyageons en train", "We travel by train"),
    ("Il écrit une lettre", "He writes a letter"),
    ("Ils lisent des livres à la bibliothèque", "They read books at the␣
↪library"),
    ("Le bébé pleure", "The baby cries"),
    ("Elle étudie dur pour les examens", "She studies hard for exams"),
    ("Nous plantons des fleurs dans le jardin", "We plant flowers in the␣
↪garden"),
    ("Il répare la voiture", "He fixes the car"),
    ("Ils boivent du café le matin", "They drink coffee in the morning"),
    ("Le soleil se couche le soir", "The sun sets in the evening"),
    ("Elle danse à la fête", "She dances at the party"),
    ("Nous jouons de la musique au concert", "We play music at the concert"),
    ("Il cuisine le dîner pour sa famille", "He cooks dinner for his family"),
    ("Ils étudient la grammaire française", "They study French grammar"),
    ("La pluie tombe doucement", "The rain falls gently"),
    ("Elle chante une chanson", "She sings a song"),
    ("Nous regardons un film ensemble", "We watch a movie together"),
    ("Il dort profondément", "He sleeps deeply"),
    ("Ils voyagent à Paris", "They travel to Paris"),
    ("Les enfants jouent dans le parc", "The children play in the park"),
    ("Elle se promène le long de la plage", "She walks along the beach"),
    ("Nous parlons au téléphone", "We talk on the phone"),
    ("Il attend le bus", "He waits for the bus"),
    ("Ils visitent la tour Eiffel", "They visit the Eiffel Tower"),
    ("Les étoiles scintillent la nuit", "The stars twinkle at night"),
    ("Elle rêve de voler", "She dreams of flying"),
    ("Nous travaillons au bureau", "We work in the office"),
    ("Il étudie l'histoire", "He studies history"),
    ("Ils écoutent la radio", "They listen to the radio"),
```

```
    ("Le vent souffle doucement", "The wind blows gently"),
    ("Elle nage dans l'océan", "She swims in the ocean"),
    ("Nous dansons au mariage", "We dance at the wedding"),
    ("Il gravit la montagne", "He climbs the mountain"),
    ("Ils font de la randonnée dans la forêt", "They hike in the forest"),
    ("Le chat miaule bruyamment", "The cat meows loudly"),
    ("Elle peint un tableau", "She paints a picture"),
    ("Nous construisons un château de sable", "We build a sandcastle"),
    ("Il chante dans le chœur", "He sings in the choir")
]

# Preparing the word to index mapping and vice versa
word_to_index = {"SOS": SOS_token, "EOS": EOS_token}
for pair in french_to_english:
    for word in pair[0].split() + pair[1].split():
        if word not in word_to_index:
            word_to_index[word] = len(word_to_index)

index_to_word = {i: word for word, i in word_to_index.items()}

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

class TranslationDataset(Dataset):
    """Custom Dataset class for handling translation pairs."""
    def __init__(self, dataset, word_to_index):
        self.dataset = dataset
        self.word_to_index = word_to_index

    def __len__(self):
        # Returns the total number of translation pairs in the dataset
        return len(self.dataset)

    def __getitem__(self, idx):
        # Retrieves a translation pair by index, converts words to indices,
        # and adds the EOS token at the end of each sentence.
        input_sentence, target_sentence = self.dataset[idx]
        input_indices = [self.word_to_index[word] for word in input_sentence.
 ↪split()] + [EOS_token]
        target_indices = [self.word_to_index[word] for word in target_sentence.
 ↪split()] + [EOS_token]
        return torch.tensor(input_indices, dtype=torch.long), torch.
 ↪tensor(target_indices, dtype=torch.long)

# Creating a DataLoader to batch and shuffle the dataset
translation_dataset = TranslationDataset(french_to_english, word_to_index)
dataloader = DataLoader(translation_dataset, batch_size=1, shuffle=True)
```

```
class Encoder(nn.Module):
    """The Encoder part of the seq2seq model."""
    def __init__(self, input_size, hidden_size):
        super(Encoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)  # Embedding␣
 ↪layer
        self.gru = nn.GRU(hidden_size, hidden_size)  # GRU layer

    def forward(self, input, hidden):
        # Forward pass for the encoder
        embedded = self.embedding(input).view(1, 1, -1)
        output, hidden = self.gru(embedded, hidden)
        return output, hidden

    def initHidden(self):
        # Initializes hidden state
        return torch.zeros(1, 1, self.hidden_size, device=device)

class Decoder(nn.Module):
    """The Decoder part of the seq2seq model."""
    def __init__(self, hidden_size, output_size):
        super(Decoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(output_size, hidden_size)  # Embedding␣
 ↪layer
        self.gru = nn.GRU(hidden_size, hidden_size)  # GRU layer
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output, hidden = self.gru(embedded, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)

# Assuming all words in the dataset + 'SOS' and 'EOS' tokens are included in␣
 ↪word_to_index
input_size = len(word_to_index)
hidden_size = 256  # Adjust according to your preference
output_size = len(word_to_index)

encoder = Encoder(input_size=input_size, hidden_size=hidden_size).to(device)
decoder = Decoder(hidden_size=hidden_size, output_size=output_size).to(device)
```

```
# Set the learning rate for optimization
learning_rate = 0.008

# Initializing optimizers for both encoder and decoder with SGD optimizer
encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)

def train(input_tensor, target_tensor, encoder, decoder, encoder_optimizer,␣
 ↪decoder_optimizer, criterion):
    # Initialize encoder hidden state
    encoder_hidden = encoder.initHidden()

    # Clear gradients for optimizers
    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    # Calculate the length of input and target tensors
    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)

    # Initialize loss
    loss = 0

    # Encoding each word in the input
    for ei in range(input_length):
        encoder_output, encoder_hidden = encoder(input_tensor[ei].unsqueeze(0),␣
 ↪encoder_hidden)

    # Decoder's first input is the SOS token
    decoder_input = torch.tensor([[SOS_token]], device=device)

    # Decoder starts with the encoder's last hidden state
    decoder_hidden = encoder_hidden

    # Decoding loop
    for di in range(target_length):
        decoder_output, decoder_hidden = decoder(decoder_input, decoder_hidden)
        # Choose top1 word from decoder's output
        topv, topi = decoder_output.topk(1)
        decoder_input = topi.squeeze().detach()  # Detach from history as input

        # Calculate loss
        loss += criterion(decoder_output, target_tensor[di].unsqueeze(0))
        if decoder_input.item() == EOS_token:  # Stop if EOS token is generated
            break
```

```
    # Backpropagation
    loss.backward()

    # Update encoder and decoder parameters
    encoder_optimizer.step()
    decoder_optimizer.step()

    # Return average loss
    return loss.item() / target_length

# Negative Log Likelihood Loss function for calculating loss
criterion = nn.NLLLoss()

# Set number of epochs for training
n_epochs = N_EPOCHS

# Lists to store training and validation losses for visualization
train_losses = []
val_losses = []

# Training loop
for epoch in range(n_epochs):
    # Training phase
    encoder.train()
    decoder.train()
    total_train_loss = 0

    for input_tensor, target_tensor in dataloader:
        # Move tensors to the correct device
        input_tensor = input_tensor[0].to(device)
        target_tensor = target_tensor[0].to(device)

        # Perform a single training step and update total loss
        loss = train(input_tensor, target_tensor, encoder, decoder,␣
 ↪encoder_optimizer, decoder_optimizer, criterion)
        total_train_loss += loss

    # Calculate average training loss for this epoch
    avg_train_loss = total_train_loss / len(dataloader)
    train_losses.append(avg_train_loss)

    # Validation phase (using the same dataset for simplicity)
    encoder.eval()
    decoder.eval()
    total_val_loss = 0
    correct_predictions = 0
```

```
    with torch.no_grad():
        for input_tensor, target_tensor in dataloader:
            # Move tensors to the correct device
            input_tensor = input_tensor[0].to(device)
            target_tensor = target_tensor[0].to(device)

            encoder_hidden = encoder.initHidden()
            input_length = input_tensor.size(0)
            target_length = target_tensor.size(0)
            loss = 0

            # Encoding step
            for ei in range(input_length):
                encoder_output, encoder_hidden = encoder(input_tensor[ei].
↪unsqueeze(0), encoder_hidden)

            # Decoding step
            decoder_input = torch.tensor([[SOS_token]], device=device)
            decoder_hidden = encoder_hidden
            predicted_indices = []

            for di in range(target_length):
                decoder_output, decoder_hidden = decoder(decoder_input,␣
↪decoder_hidden)
                topv, topi = decoder_output.topk(1)
                predicted_indices.append(topi.item())
                decoder_input = topi.squeeze().detach()

                loss += criterion(decoder_output, target_tensor[di].
↪unsqueeze(0))
                if decoder_input.item() == EOS_token:
                    break

            # Track validation loss and accuracy
            total_val_loss += loss.item() / target_length
            if predicted_indices == target_tensor.tolist():
                correct_predictions += 1

    # Calculate average validation loss and accuracy for this epoch
    avg_val_loss = total_val_loss / len(dataloader)
    val_accuracy = correct_predictions / len(dataloader)
    val_losses.append(avg_val_loss)

    # Print progress every 10 epochs
    if epoch % 5 == 0 or epoch == n_epochs - 1:
        print(f'Epoch {epoch}, Train Loss: {avg_train_loss:.4f}, Val Loss:␣
↪{avg_val_loss:.4f}, Val Accuracy: {val_accuracy:.4f}')
```

```
# Print final validation accuracy
print(f'\nFinal Validation Accuracy: {val_accuracy:.4f}')

# Visualize training and validation losses
plt.figure(figsize=(10, 6))
plt.plot(range(1, n_epochs + 1), train_losses, label='Training Loss')
plt.plot(range(1, n_epochs + 1), val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.grid(True)
plt.savefig('french2english_loss.png')
plt.show()

def evaluate_and_show_examples(encoder, decoder, dataloader, criterion,␣
 ↪n_examples=10):
    # Switch model to evaluation mode
    encoder.eval()
    decoder.eval()

    total_loss = 0
    correct_predictions = 0

    # No gradient calculation
    with torch.no_grad():
        for i, (input_tensor, target_tensor) in enumerate(dataloader):
            # Move tensors to the correct device
            input_tensor = input_tensor[0].to(device)
            target_tensor = target_tensor[0].to(device)

            encoder_hidden = encoder.initHidden()

            input_length = input_tensor.size(0)
            target_length = target_tensor.size(0)

            loss = 0

            # Encoding step
            for ei in range(input_length):
                encoder_output, encoder_hidden = encoder(input_tensor[ei].
 ↪unsqueeze(0), encoder_hidden)

            # Decoding step
            decoder_input = torch.tensor([[SOS_token]], device=device)
            decoder_hidden = encoder_hidden
```

```
            predicted_indices = []

            for di in range(target_length):
                decoder_output, decoder_hidden = decoder(decoder_input,␣
↪decoder_hidden)
                topv, topi = decoder_output.topk(1)
                predicted_indices.append(topi.item())
                decoder_input = topi.squeeze().detach()

                loss += criterion(decoder_output, target_tensor[di].
↪unsqueeze(0))
                if decoder_input.item() == EOS_token:
                    break

            # Calculate and print loss and accuracy for the evaluation
            total_loss += loss.item() / target_length
            if predicted_indices == target_tensor.tolist():
                correct_predictions += 1

            # Optionally, print some examples
            if i < n_examples:
                predicted_sentence = ' '.join([index_to_word[index] for index␣
↪in predicted_indices if index not in (SOS_token, EOS_token)])
                target_sentence = ' '.join([index_to_word[index.item()] for␣
↪index in target_tensor if index.item() not in (SOS_token, EOS_token)])
                input_sentence = ' '.join([index_to_word[index.item()] for␣
↪index in input_tensor if index.item() not in (SOS_token, EOS_token)])

                print(f'Input: {input_sentence}, Target: {target_sentence},␣
↪Predicted: {predicted_sentence}')

        # Print overall evaluation results
        average_loss = total_loss / len(dataloader)
        accuracy = correct_predictions / len(dataloader)
        print(f'Evaluation Loss: {average_loss}, Accuracy: {accuracy}')

# Perform evaluation with examples
evaluate_and_show_examples(encoder, decoder, dataloader, criterion)
```

```
Epoch 0, Train Loss: 4.1198, Val Loss: 2.5558, Val Accuracy: 0.0000
Epoch 5, Train Loss: 3.1443, Val Loss: 3.4980, Val Accuracy: 0.0000
Epoch 10, Train Loss: 2.7152, Val Loss: 3.0072, Val Accuracy: 0.0000
Epoch 15, Train Loss: 2.2012, Val Loss: 1.9004, Val Accuracy: 0.0000
Epoch 20, Train Loss: 1.8719, Val Loss: 1.5652, Val Accuracy: 0.0130
Epoch 25, Train Loss: 1.3061, Val Loss: 1.1781, Val Accuracy: 0.2208
Epoch 30, Train Loss: 0.7598, Val Loss: 0.6115, Val Accuracy: 0.6494
```

```
Epoch 35, Train Loss: 0.3808, Val Loss: 0.3218, Val Accuracy: 0.9091
Epoch 40, Train Loss: 0.2329, Val Loss: 0.1891, Val Accuracy: 0.9610
Epoch 44, Train Loss: 0.1526, Val Loss: 0.1291, Val Accuracy: 0.9870

Final Validation Accuracy: 0.9870
```


```
Input: Il aime lire des livres, Target: He enjoys reading books, Predicted: He
enjoys reading books
Input: Le chien aboie bruyamment, Target: The dog barks loudly, Predicted: The
dog barks loudly
Input: Ils rient de la blague, Target: They laugh at the joke, Predicted: They
laugh at the joke
Input: J'ai froid, Target: I am cold, Predicted: I am cold
Input: Le soleil brille, Target: The sun is shining, Predicted: The sun is
shining
Input: Nous regardons des films le vendredi, Target: We watch movies on Fridays,
Predicted: We watch movies on Fridays
Input: Nous travaillons au bureau, Target: We work in the office, Predicted: We
work in the office
Input: Elle étudie dur pour les examens, Target: She studies hard for exams,
Predicted: She studies hard for exams
Input: Le bébé pleure, Target: The baby cries, Predicted: The baby cries
Input: Elle enseigne l'anglais à l'école, Target: She teaches English at school,
Predicted: She teaches English at school
Evaluation Loss: 0.129089000442825, Accuracy: 0.987012987012987
```

With Attention:

[10]:
```
# French to English translation with attention.
class TranslationDataset(Dataset):
    """Custom Dataset class for handling translation pairs."""
    def __init__(self, dataset, word_to_index):
        self.dataset = dataset
        self.word_to_index = word_to_index

    def __len__(self):
        # Returns the total number of translation pairs in the dataset
        return len(self.dataset)

    def __getitem__(self, idx):
        # Retrieves a translation pair by index, converts words to indices,
        # and adds the EOS token at the end of each sentence.
        input_sentence, target_sentence = self.dataset[idx]
        input_indices = [self.word_to_index[word] for word in input_sentence.
 ↪split()] + [EOS_token]
        target_indices = [self.word_to_index[word] for word in target_sentence.
 ↪split()] + [EOS_token]
        return torch.tensor(input_indices, dtype=torch.long), torch.
 ↪tensor(target_indices, dtype=torch.long)

# Creating DataLoaders for training and validation
max_length = 14
translation_dataset = TranslationDataset(french_to_english, word_to_index)

# Use the entire dataset for both training and validation
train_dataloader = DataLoader(translation_dataset, batch_size=1, shuffle=True)
val_dataloader = DataLoader(translation_dataset, batch_size=1, shuffle=False)

class Encoder(nn.Module):
    """The Encoder part of the seq2seq model."""
    def __init__(self, input_size, hidden_size):
        super(Encoder, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)  # Embedding␣
 ↪layer
        self.gru = nn.GRU(hidden_size, hidden_size)  # GRU layer

    def forward(self, input, hidden):
        # Forward pass for the encoder
        embedded = self.embedding(input).view(1, 1, -1)
        output, hidden = self.gru(embedded, hidden)
        return output, hidden
```

```
    def initHidden(self):
        # Initializes hidden state
        return torch.zeros(1, 1, self.hidden_size, device=device)

class AttentionDecoder(nn.Module):
    """The Decoder part of the seq2seq model with attention mechanism."""
    def __init__(self, hidden_size, output_size, max_length=14, dropout_p=0.1):
        super(AttentionDecoder, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.dropout_p = dropout_p
        self.max_length = max_length
        self.embedding = nn.Embedding(self.output_size, self.hidden_size)  #␣
 ↪Embedding layer
        self.attn = nn.Linear(self.hidden_size * 2, self.max_length)  #␣
 ↪Attention layer
        self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_size) ␣
 ↪# Combining layer
        self.dropout = nn.Dropout(self.dropout_p)
        self.gru = nn.GRU(self.hidden_size, self.hidden_size)  # GRU layer
        self.out = nn.Linear(self.hidden_size, output_size)  # Output layer

    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)

        # Calculating attention weights
        attn_weights = torch.softmax(
            self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))

        # Combining embedded input with attention output
        output = torch.cat((embedded[0], attn_applied[0]), 1)
        output = self.attn_combine(output).unsqueeze(0)

        output = torch.relu(output)
        output, hidden = self.gru(output, hidden)

        output = torch.log_softmax(self.out(output[0]), dim=1)
        return output, hidden, attn_weights

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)

# Assuming all words in the dataset + 'SOS' and 'EOS' tokens are included in␣
 ↪word_to_index
```

```
input_size = len(word_to_index)
hidden_size = 256  # Adjust according to your preference
output_size = len(word_to_index)

encoder = Encoder(input_size=input_size, hidden_size=hidden_size).to(device)
decoder = AttentionDecoder(hidden_size=hidden_size, output_size=output_size).
 ↪to(device)

# Set the learning rate for optimization
learning_rate = 0.008

# Initializing optimizers for both encoder and decoder with SGD optimizer
encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)

def train(input_tensor, target_tensor, encoder, decoder, encoder_optimizer,␣
 ↪decoder_optimizer, criterion, max_length=14):
    # Initialize encoder hidden state
    encoder_hidden = encoder.initHidden()

    # Clear gradients for optimizers
    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    # Calculate the length of input and target tensors
    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)

    # Initialize loss
    loss = 0

    # Encoding each word in the input
    encoder_outputs = torch.zeros(max_length, encoder.hidden_size,␣
 ↪device=device)
    for ei in range(input_length):
        encoder_output, encoder_hidden = encoder(input_tensor[ei].unsqueeze(0),␣
 ↪encoder_hidden)
        encoder_outputs[ei] = encoder_output[0, 0]

    # Decoder's first input is the SOS token
    decoder_input = torch.tensor([[SOS_token]], device=device)

    # Decoder starts with the encoder's last hidden state
    decoder_hidden = encoder_hidden

    # Decoding loop with attention
    for di in range(target_length):
```

```
        decoder_output, decoder_hidden, decoder_attention =␣
 ↪decoder(decoder_input, decoder_hidden, encoder_outputs)
        # Choose top1 word from decoder's output
        topv, topi = decoder_output.topk(1)
        decoder_input = topi.squeeze().detach()  # Detach from history as input

        # Calculate loss
        loss += criterion(decoder_output, target_tensor[di].unsqueeze(0))
        if decoder_input.item() == EOS_token:  # Stop if EOS token is generated
            break

    # Backpropagation
    loss.backward()

    # Update encoder and decoder parameters
    encoder_optimizer.step()
    decoder_optimizer.step()

    # Return average loss
    return loss.item() / target_length

# Negative Log Likelihood Loss function for calculating loss
criterion = nn.NLLLoss()

# Set number of epochs for training
n_epochs = N_EPOCHS

# Lists to store training and validation losses
train_losses = []
val_losses = []

# Training loop
for epoch in range(n_epochs):
    # Training phase
    encoder.train()
    decoder.train()
    total_train_loss = 0

    for input_tensor, target_tensor in train_dataloader:
        # Move tensors to the correct device
        input_tensor = input_tensor[0].to(device)
        target_tensor = target_tensor[0].to(device)

        # Perform a single training step and update total loss
        loss = train(input_tensor, target_tensor, encoder, decoder,␣
 ↪encoder_optimizer, decoder_optimizer, criterion)
        total_train_loss += loss
```

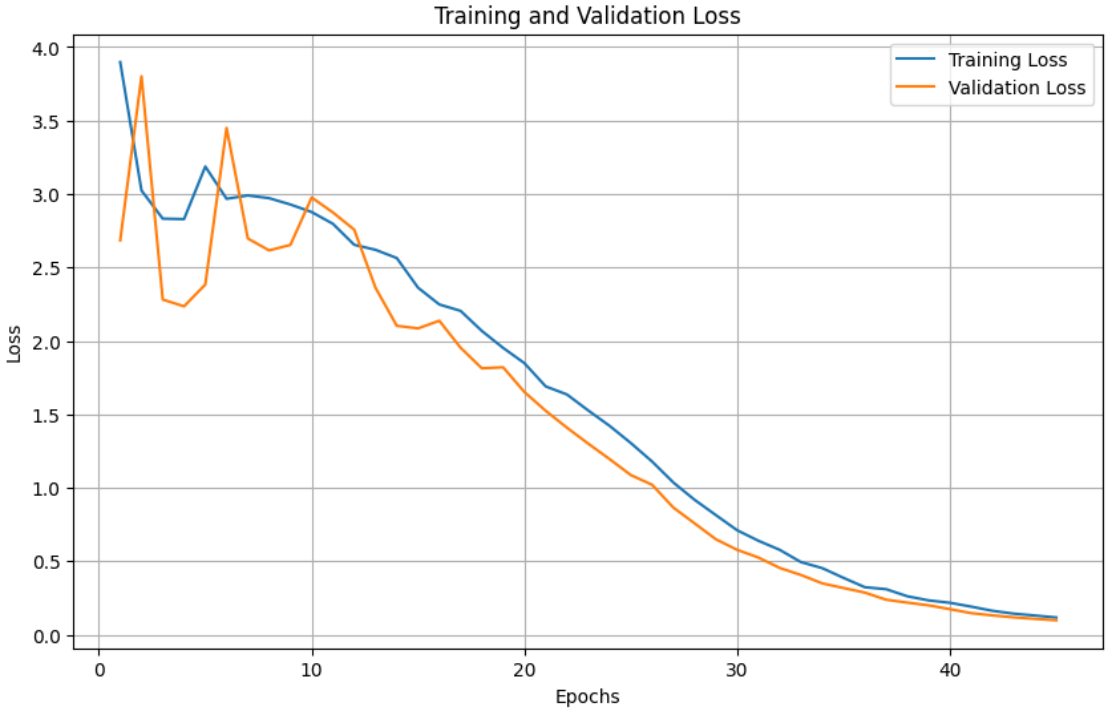
```
    # Calculate average training loss for this epoch
    avg_train_loss = total_train_loss / len(train_dataloader)
    train_losses.append(avg_train_loss)

    # Validation phase
    encoder.eval()
    decoder.eval()
    total_val_loss = 0

    with torch.no_grad():
        for input_tensor, target_tensor in val_dataloader:
            # Move tensors to the correct device
            input_tensor = input_tensor[0].to(device)
            target_tensor = target_tensor[0].to(device)

            # Calculate validation loss
            encoder_hidden = encoder.initHidden()
            input_length = input_tensor.size(0)
            target_length = target_tensor.size(0)

            # Encoding step
            encoder_outputs = torch.zeros(max_length, encoder.hidden_size,␣
↪device=device)
            for ei in range(input_length):
                encoder_output, encoder_hidden = encoder(input_tensor[ei].
↪unsqueeze(0), encoder_hidden)
                encoder_outputs[ei] = encoder_output[0, 0]

            # Decoding step
            decoder_input = torch.tensor([[SOS_token]], device=device)
            decoder_hidden = encoder_hidden
            loss = 0

            for di in range(target_length):
                decoder_output, decoder_hidden, decoder_attention =␣
↪decoder(decoder_input, decoder_hidden, encoder_outputs)
                topv, topi = decoder_output.topk(1)
                decoder_input = topi.squeeze().detach()

                loss += criterion(decoder_output, target_tensor[di].
↪unsqueeze(0))
                if decoder_input.item() == EOS_token:
                    break

            total_val_loss += loss.item() / target_length
```

```
    # Calculate average validation loss for this epoch
    avg_val_loss = total_val_loss / len(val_dataloader)
    val_losses.append(avg_val_loss)

    # Print progress every 5 epochs
    if epoch % 5 == 0 or epoch == n_epochs - 1:
        print(f'Epoch {epoch}, Train Loss: {avg_train_loss:.4f}, Val Loss:␣
↪{avg_val_loss:.4f}')

def evaluate_and_show_examples(encoder, decoder, dataloader, criterion,␣
↪n_examples=10):
    # Switch model to evaluation mode
    encoder.eval()
    decoder.eval()

    total_loss = 0
    correct_predictions = 0

    # No gradient calculation
    with torch.no_grad():
        for i, (input_tensor, target_tensor) in enumerate(dataloader):
            # Move tensors to the correct device
            input_tensor = input_tensor[0].to(device)
            target_tensor = target_tensor[0].to(device)

            encoder_hidden = encoder.initHidden()

            input_length = input_tensor.size(0)
            target_length = target_tensor.size(0)

            loss = 0

            # Encoding step
            encoder_outputs = torch.zeros(max_length, encoder.hidden_size,␣
↪device=device)
            for ei in range(input_length):
                encoder_output, encoder_hidden = encoder(input_tensor[ei].
↪unsqueeze(0), encoder_hidden)
                encoder_outputs[ei] = encoder_output[0, 0]

            # Decoding step
            decoder_input = torch.tensor([[SOS_token]], device=device)
            decoder_hidden = encoder_hidden

            predicted_indices = []

            for di in range(target_length):
```

```
                decoder_output, decoder_hidden, decoder_attention =␣
↪decoder(decoder_input, decoder_hidden, encoder_outputs)
                topv, topi = decoder_output.topk(1)
                predicted_indices.append(topi.item())
                decoder_input = topi.squeeze().detach()

                loss += criterion(decoder_output, target_tensor[di].
↪unsqueeze(0))
                if decoder_input.item() == EOS_token:
                    break

            # Calculate and print loss and accuracy for the evaluation
            total_loss += loss.item() / target_length
            if predicted_indices == target_tensor.tolist():
                correct_predictions += 1

            # Optionally, print some examples
            if i < n_examples:
                predicted_sentence = ' '.join([index_to_word[index] for index␣
↪in predicted_indices if index not in (SOS_token, EOS_token)])
                target_sentence = ' '.join([index_to_word[index.item()] for␣
↪index in target_tensor if index.item() not in (SOS_token, EOS_token)])
                input_sentence = ' '.join([index_to_word[index.item()] for␣
↪index in input_tensor if index.item() not in (SOS_token, EOS_token)])

                print(f'Input: {input_sentence}, Target: {target_sentence},␣
↪Predicted: {predicted_sentence}')

        # Print overall evaluation results
        average_loss = total_loss / len(dataloader)
        accuracy = correct_predictions / len(dataloader)
        print(f'Evaluation Loss: {average_loss}, Accuracy: {accuracy}')

# Visualize training and validation loss
plt.figure(figsize=(10, 6))
plt.plot(range(1, n_epochs + 1), train_losses, label='Training Loss')
plt.plot(range(1, n_epochs + 1), val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.grid(True)
plt.savefig('french2english_attention_loss.png')
plt.show()

# Perform final evaluation with examples and report validation accuracy
```

```
print("\nFinal Evaluation Results:")
evaluate_and_show_examples(encoder, decoder, val_dataloader, criterion)
```

```
Epoch 0, Train Loss: 3.8980, Val Loss: 2.6844
Epoch 5, Train Loss: 2.9676, Val Loss: 3.4497
Epoch 10, Train Loss: 2.7971, Val Loss: 2.8745
Epoch 15, Train Loss: 2.2483, Val Loss: 2.1375
Epoch 20, Train Loss: 1.6908, Val Loss: 1.5247
Epoch 25, Train Loss: 1.1797, Val Loss: 1.0214
Epoch 30, Train Loss: 0.6401, Val Loss: 0.5260
Epoch 35, Train Loss: 0.3241, Val Loss: 0.2875
Epoch 40, Train Loss: 0.1920, Val Loss: 0.1476
```




```
Final Evaluation Results:
Input: J'ai froid, Target: I am cold, Predicted: I am cold
Input: Tu es fatigué, Target: You are tired, Predicted: You are tired
Input: Il a faim, Target: He is hungry, Predicted: He is hungry
Input: Elle est heureuse, Target: She is happy, Predicted: She is happy
Input: Nous sommes amis, Target: We are friends, Predicted: We are friends
Input: Ils sont étudiants, Target: They are students, Predicted: They are
students
Input: Le chat dort, Target: The cat is sleeping, Predicted: The cat is sleeping
Input: Le soleil brille, Target: The sun is shining, Predicted: The sun is
shining
```

```
Input: Nous aimons la musique, Target: We love music, Predicted: We love music
Input: Elle parle français couramment, Target: She speaks French fluently,
Predicted: She speaks French fluently
Evaluation Loss: 0.09898607425884691, Accuracy: 1.0
```

[ ]: