Problem-3-RNN-Attention

March 12, 2024

[ ]:
```
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import random
import time
import numpy as np

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda:0
```

[ ]:
```
PRINT_EVERY = 5000
PLOT_EVERY = 100

language = 'ta'
```

[ ]:
```
# Language Model
SOS_token = 0
EOS_token = 1

class Language:
    def __init__(self, name):
        self.name = name
        self.word2index = {}
        self.word2count = {}
        self.index2word = {SOS_token: "<", EOS_token: ">"}
        self.n_chars = 2  # Count SOS and EOS

    def addWord(self, word):
        for char in str(word):
            self.addChar(char)

    def addChar(self, char):
        if char not in self.word2index:
            self.word2index[char] = self.n_chars
```

```
                self.word2count[char] = 1
                self.index2word[self.n_chars] = char
                self.n_chars += 1
            else:
                self.word2count[char] += 1
```

[ ]:
```
def get_data(lang: str, type: str) -> list[list[str]]:
    """
    Returns: 'pairs': list of [input_word, target_word] pairs
    """
    path = f"../data/dakshina_dataset_v1.0/{lang}/lexicons/{lang}.translit.
 ↪sampled.{type}.tsv"
    df = pd.read_csv(path, sep='\t', header=None)
    pairs = df.values.tolist()
    return pairs
```

[ ]:
```
def get_languages(lang: str):
    """
    Returns
    1. input_lang: input language - English
    2. output_lang: output language - Given language
    3. pairs: list of [input_word, target_word] pairs
    """
    input_lang = Language('eng')
    output_lang = Language(lang)
    pairs = get_data(lang, 'train')
    for pair in pairs:
        input_lang.addWord(pair[1])
        output_lang.addWord(pair[0])
    return input_lang, output_lang, pairs
```

[ ]:
```
def get_cell(cell_type: str):
    if cell_type == "LSTM":
        return nn.LSTM
    elif cell_type == "GRU":
        return nn.GRU
    elif cell_type == "RNN":
        return nn.RNN
    else:
        raise Exception("Invalid cell type")

def get_optimizer(optimizer: str):
    if optimizer == "SGD":
        return optim.SGD
    elif optimizer == "ADAM":
        return optim.Adam
    else:
```

```
        raise Exception("Invalid optimizer")
```

[ ]:
```
def indexesFromWord(lang:Language, word:str):
    return [lang.word2index[char] for char in str(word)]

def tensorFromWord(lang:Language, word:str):
    indexes = indexesFromWord(lang, word)
    indexes.append(EOS_token)
    return torch.tensor(indexes, dtype=torch.long, device=device).view(-1, 1)

def tensorsFromPair(input_lang:Language, output_lang:Language, pair:list[str]):
    input_tensor = tensorFromWord(input_lang, pair[1])
    target_tensor = tensorFromWord(output_lang, pair[0])
    return (input_tensor, target_tensor)
```

[ ]:
```
class Encoder(nn.Module):
    def __init__(self,
                 in_sz: int,
                 embed_sz: int,
                 hidden_sz: int,
                 cell_type: str,
                 n_layers: int,
                 dropout: float):

        super(Encoder, self).__init__()
        self.hidden_sz = hidden_sz
        self.n_layers = n_layers
        self.dropout = dropout
        self.cell_type = cell_type
        self.embedding = nn.Embedding(in_sz, embed_sz)

        self.rnn = get_cell(cell_type)(input_size = embed_sz,
                                       hidden_size = hidden_sz,
                                       num_layers = n_layers,
                                       dropout = dropout)

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

        if(self.cell_type == "LSTM"):
            output, (hidden, cell) = self.rnn(embedded, (hidden, cell))
        else:
            output, hidden = self.rnn(embedded, hidden)

        return output, hidden, cell

    def initHidden(self):
```

```
        return torch.zeros(self.n_layers, 1, self.hidden_sz, device=device)
```

[ ]:
```
class AttentionDecoder(nn.Module):
    def __init__(self,
                 out_sz: int,
                 embed_sz: int,
                 hidden_sz: int,
                 cell_type: str,
                 n_layers: int,
                 dropout: float):

        super(AttentionDecoder, self).__init__()
        self.hidden_sz = hidden_sz
        self.n_layers = n_layers
        self.dropout = dropout
        self.cell_type = cell_type
        self.embedding = nn.Embedding(out_sz, embed_sz)

        self.attn = nn.Linear(hidden_sz + embed_sz, 50)
        self.attn_combine = nn.Linear(hidden_sz + embed_sz, hidden_sz)

        self.rnn = get_cell(cell_type)(input_size = hidden_sz,
                                       hidden_size = hidden_sz,
                                       num_layers = n_layers,
                                       dropout = dropout)

        self.out = nn.Linear(hidden_sz, out_sz)
        self.softmax = nn.LogSoftmax(dim=1)

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

        attn_weights = F.softmax(self.attn(torch.cat((embedding[0], hidden[0]),␣
↪1)), dim=1)
        attn_applied = torch.bmm(attn_weights.unsqueeze(0), encoder_outputs.
↪unsqueeze(0))

        output = torch.cat((embedding[0], attn_applied[0]), 1)
        output = self.attn_combine(output).unsqueeze(0)

        if(self.cell_type == "LSTM"):
            output, (hidden, cell) = self.rnn(output, (hidden, cell))
        else:
            output, hidden = self.rnn(output, hidden)

        output = self.softmax(self.out(output[0]))
        return output, hidden, cell, attn_weights
```

```python
    def initHidden(self):
        return torch.zeros(self.n_layers, 1, self.hidden_sz, device=device)
```

[ ]:
```python
class Translator:
    def __init__(self, lang: str, params: dict):
        self.lang = lang
        self.input_lang, self.output_lang, self.pairs = get_languages(self.lang)
        self.input_size = self.input_lang.n_chars
        self.output_size = self.output_lang.n_chars

        self.training_pairs = [tensorsFromPair(self.input_lang, self.
↪output_lang, pair) for pair in self.pairs]

        self.encoder = Encoder(in_sz = self.input_size,
                               embed_sz = params["embed_size"],
                               hidden_sz = params["hidden_size"],
                               cell_type = params["cell_type"],
                               n_layers = params["num_layers"],
                               dropout = params["dropout"]).to(device)

        self.decoder = AttentionDecoder(out_sz = self.output_size,
                               embed_sz = params["embed_size"],
                               hidden_sz = params["hidden_size"],
                               cell_type = params["cell_type"],
                               n_layers = params["num_layers"],
                               dropout = params["dropout"]).to(device)

        self.encoder_optimizer = get_optimizer(params["optimizer"])(self.
↪encoder.parameters(), lr=params["learning_rate"],␣
↪weight_decay=params["weight_decay"])
        self.decoder_optimizer = get_optimizer(params["optimizer"])(self.
↪decoder.parameters(), lr=params["learning_rate"],␣
↪weight_decay=params["weight_decay"])

        self.criterion = nn.NLLLoss()

        self.teacher_forcing_ratio = params["teacher_forcing_ratio"]
        self.max_length = params["max_length"]

    def train_single(self, input_tensor, target_tensor):
        encoder_hidden = self.encoder.initHidden()
        encoder_cell = self.encoder.initHidden()

        self.encoder_optimizer.zero_grad()
        self.decoder_optimizer.zero_grad()
```

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

        encoder_outputs = torch.zeros(self.max_length, self.encoder.hidden_sz,␣
↪device=device)

        loss = 0

        for ei in range(input_length):
            encoder_output, encoder_hidden, encoder_cell = self.
↪encoder(input_tensor[ei], encoder_hidden, encoder_cell)
            encoder_outputs[ei] = encoder_output[0, 0]

        decoder_input = torch.tensor([[SOS_token]], device=device)
        decoder_hidden, decoder_cell = encoder_hidden, encoder_cell

        use_teacher_forcing = True if random.random() < self.
↪teacher_forcing_ratio else False

        if use_teacher_forcing:
            for di in range(target_length):
                decoder_output, decoder_hidden, decoder_cell, decoder_attention␣
↪= self.decoder(decoder_input, decoder_hidden, decoder_cell, encoder_outputs)
                loss += self.criterion(decoder_output, target_tensor[di])

                decoder_input = target_tensor[di]
        else:
            for di in range(target_length):
                decoder_output, decoder_hidden, decoder_cell, decoder_attention␣
↪= self.decoder(decoder_input, decoder_hidden, decoder_cell, encoder_outputs)
                loss += self.criterion(decoder_output, target_tensor[di])

                topv, topi = decoder_output.topk(1)
                decoder_input = topi.squeeze().detach()
                if decoder_input.item() == EOS_token:
                    break

        loss.backward()
        self.encoder_optimizer.step()
        self.decoder_optimizer.step()

        return loss.item() / target_length

    def train(self, iters=-1):
        start_time = time.time()
        plot_losses = []
        print_loss_total = 0
```

```
        plot_loss_total = 0

        random.shuffle(self.training_pairs)
        iters = len(self.training_pairs) if iters == -1 else iters

        for iter in range(1, iters):
            training_pair = self.training_pairs[iter - 1]
            input_tensor = training_pair[0]
            target_tensor = training_pair[1]

            loss = self.train_single(input_tensor, target_tensor)
            print_loss_total += loss
            plot_loss_total += loss

            if iter % PRINT_EVERY == 0:
                print_loss_avg = print_loss_total / PRINT_EVERY
                print_loss_total = 0
                current_time = time.time()
                print("Loss: {:.4f} | Iterations: {} | Time: {:.3f}".
↪format(print_loss_avg, iter, current_time - start_time))

            if iter % PLOT_EVERY == 0:
                plot_loss_avg = plot_loss_total / PLOT_EVERY
                plot_losses.append(plot_loss_avg)
                plot_loss_total = 0

        return plot_losses

    def evaluate(self, word):
        with torch.no_grad():
            input_tensor = tensorFromWord(self.input_lang, word)
            input_length = input_tensor.size()[0]
            encoder_hidden = self.encoder.initHidden()
            encoder_cell = self.encoder.initHidden()

            encoder_outputs = torch.zeros(self.max_length, self.encoder.
↪hidden_sz, device=device)

            for ei in range(input_length):
                encoder_output, encoder_hidden, encoder_cell = self.
↪encoder(input_tensor[ei], encoder_hidden, encoder_cell)
                encoder_outputs[ei] += encoder_output[0, 0]

            decoder_input = torch.tensor([[SOS_token]], device=device)
            decoder_hidden, decoder_cell = encoder_hidden, encoder_cell

            decoded_chars = ""
```

```
        decoder_attentions = torch.zeros(self.max_length, self.max_length)

        for di in range(self.max_length):
            decoder_output, decoder_hidden, decoder_cell, decoder_attention␣
↪= self.decoder(decoder_input, decoder_hidden, decoder_cell, encoder_outputs)
            decoder_attentions[di] = decoder_attention.data
            topv, topi = decoder_output.topk(1)

            if topi.item() == EOS_token:
                break
            else:
                decoded_chars += self.output_lang.index2word[topi.item()]

            decoder_input = topi.squeeze().detach()

        return decoded_chars, decoder_attentions[:di + 1]

    def test_validate(self, type:str):
        pairs = get_data(self.lang, type)
        accuracy = 0
        for pair in pairs:
            output, _ = self.evaluate(pair[1])
            if output == pair[0]:
                accuracy += 1
        return accuracy / len(pairs)
```

[ ]:
```
params = {
    "embed_size": 32,
    "hidden_size": 256,
    "cell_type": "RNN",
    "num_layers": 2,
    "dropout": 0,
    "learning_rate": 0.001,
    "optimizer": "SGD",
    "teacher_forcing_ratio": 0.5,
    "max_length": 50,
    "weight_decay": 0.001
}
```

[ ]:
```
model = Translator(language, params)
```

[ ]:
```
epochs = 10
old_validation_accuracy = 0

train_losses = []
train_accuracies, val_accuracies = [], []
```

```python
for epoch in range(epochs):
    print("Epoch: {}".format(epoch + 1))
    plot_losses = model.train()

    # take average of plot losses as training loss
    training_loss = sum(plot_losses) / len(plot_losses)
    train_losses.append(training_loss)

    print("Training Loss: {:.4f}".format(training_loss))

    training_accuracy = model.test_validate('train')
    print("Training Accuracy: {:.4f}".format(training_accuracy))
    train_accuracies.append(training_accuracy)

    validation_accuracy = model.test_validate('dev')
    print("Validation Accuracy: {:.4f}".format(validation_accuracy))
    val_accuracies.append(validation_accuracy)

    if epoch > 0:
        if validation_accuracy < 0.0001:
            print("Validation Accuracy is too low. Stopping training.")
            break

        if validation_accuracy < 0.95 * old_validation_accuracy:
            print("Validation Accuracy is decreasing. Stopping training.")
            break

    old_validation_accuracy = validation_accuracy
print("Training Complete")

print("Testing Model")
test_accuracy = model.test_validate('test')
print("Test Accuracy: {:.4f}".format(test_accuracy))
print("Testing Complete")
```

```
Epoch: 1
Loss: 2.5687 | Iterations: 5000 | Time: 43.384
Loss: 2.4377 | Iterations: 10000 | Time: 95.929
Loss: 2.3213 | Iterations: 15000 | Time: 162.978
Loss: 1.7871 | Iterations: 20000 | Time: 234.288
Loss: 1.3276 | Iterations: 25000 | Time: 306.692
Loss: 1.0786 | Iterations: 30000 | Time: 380.043
Loss: 0.9721 | Iterations: 35000 | Time: 452.405
Loss: 0.8827 | Iterations: 40000 | Time: 527.827
Loss: 0.8217 | Iterations: 45000 | Time: 599.997
Loss: 0.7804 | Iterations: 50000 | Time: 672.937
Loss: 0.7558 | Iterations: 55000 | Time: 748.516
Loss: 0.7182 | Iterations: 60000 | Time: 823.095
```

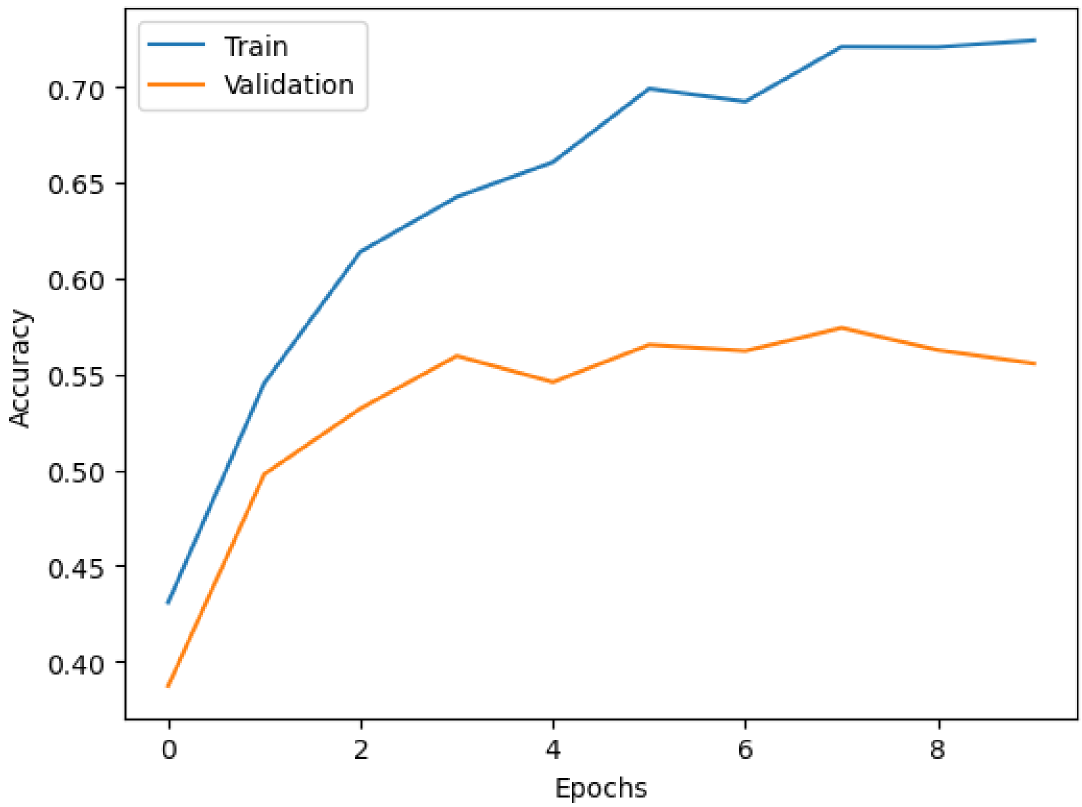
```
Loss: 0.6967 | Iterations: 65000 | Time: 897.271
Training Loss: 1.2888
Training Accuracy: 0.4309
Validation Accuracy: 0.3873
Epoch: 2
Loss: 0.6478 | Iterations: 5000 | Time: 74.271
Loss: 0.6342 | Iterations: 10000 | Time: 149.762
Loss: 0.5884 | Iterations: 15000 | Time: 235.037
Loss: 0.6335 | Iterations: 20000 | Time: 318.919
Loss: 0.6009 | Iterations: 25000 | Time: 401.862
Loss: 0.5723 | Iterations: 30000 | Time: 484.142
Loss: 0.5993 | Iterations: 35000 | Time: 565.919
Loss: 0.5828 | Iterations: 40000 | Time: 647.159
Loss: 0.5880 | Iterations: 45000 | Time: 729.762
Loss: 0.5773 | Iterations: 50000 | Time: 812.198
Loss: 0.5468 | Iterations: 55000 | Time: 893.582
Loss: 0.5763 | Iterations: 60000 | Time: 970.691
Loss: 0.5500 | Iterations: 65000 | Time: 1047.062
Training Loss: 0.5899
Training Accuracy: 0.5454
Validation Accuracy: 0.4979
Epoch: 3
Loss: 0.5259 | Iterations: 5000 | Time: 74.859
Loss: 0.4945 | Iterations: 10000 | Time: 149.271
Loss: 0.5354 | Iterations: 15000 | Time: 224.545
Loss: 0.5192 | Iterations: 20000 | Time: 299.779
Loss: 0.5077 | Iterations: 25000 | Time: 374.641
Loss: 0.5059 | Iterations: 30000 | Time: 453.810
Loss: 0.4760 | Iterations: 35000 | Time: 532.342
Loss: 0.5068 | Iterations: 40000 | Time: 606.857
Loss: 0.5008 | Iterations: 45000 | Time: 682.950
Loss: 0.4871 | Iterations: 50000 | Time: 756.878
Loss: 0.5246 | Iterations: 55000 | Time: 831.683
Loss: 0.4597 | Iterations: 60000 | Time: 906.690
Loss: 0.4833 | Iterations: 65000 | Time: 985.941
Training Loss: 0.4993
Training Accuracy: 0.6141
Validation Accuracy: 0.5322
Epoch: 4
Loss: 0.4742 | Iterations: 5000 | Time: 75.448
Loss: 0.4447 | Iterations: 10000 | Time: 148.141
Loss: 0.4650 | Iterations: 15000 | Time: 219.083
Loss: 0.4535 | Iterations: 20000 | Time: 287.889
Loss: 0.4651 | Iterations: 25000 | Time: 350.972
Loss: 0.4399 | Iterations: 30000 | Time: 415.016
Loss: 0.4293 | Iterations: 35000 | Time: 478.990
Loss: 0.4492 | Iterations: 40000 | Time: 543.519
Loss: 0.4641 | Iterations: 45000 | Time: 608.867
```

```
Loss: 0.4412 | Iterations: 50000 | Time: 678.899
Loss: 0.4562 | Iterations: 55000 | Time: 751.783
Loss: 0.4734 | Iterations: 60000 | Time: 824.210
Loss: 0.4547 | Iterations: 65000 | Time: 897.664
Training Loss: 0.4539
Training Accuracy: 0.6426
Validation Accuracy: 0.5595
Epoch: 5
Loss: 0.4263 | Iterations: 5000 | Time: 64.759
Loss: 0.4270 | Iterations: 10000 | Time: 134.601
Loss: 0.4368 | Iterations: 15000 | Time: 204.149
Loss: 0.4130 | Iterations: 20000 | Time: 275.321
Loss: 0.3975 | Iterations: 25000 | Time: 351.159
Loss: 0.4288 | Iterations: 30000 | Time: 426.540
Loss: 0.4045 | Iterations: 35000 | Time: 500.401
Loss: 0.4167 | Iterations: 40000 | Time: 573.826
Loss: 0.4048 | Iterations: 45000 | Time: 647.725
Loss: 0.4077 | Iterations: 50000 | Time: 721.949
Loss: 0.4143 | Iterations: 55000 | Time: 796.224
Loss: 0.4201 | Iterations: 60000 | Time: 871.987
Loss: 0.4087 | Iterations: 65000 | Time: 946.028
Training Loss: 0.4152
Training Accuracy: 0.6608
Validation Accuracy: 0.5459
Epoch: 6
Loss: 0.3899 | Iterations: 5000 | Time: 70.318
Loss: 0.3717 | Iterations: 10000 | Time: 142.172
Loss: 0.3754 | Iterations: 15000 | Time: 216.522
Loss: 0.4078 | Iterations: 20000 | Time: 289.566
Loss: 0.4064 | Iterations: 25000 | Time: 365.519
Loss: 0.3899 | Iterations: 30000 | Time: 442.102
Loss: 0.3817 | Iterations: 35000 | Time: 519.230
Loss: 0.3671 | Iterations: 40000 | Time: 592.892
Loss: 0.3893 | Iterations: 45000 | Time: 667.274
Loss: 0.4023 | Iterations: 50000 | Time: 743.081
Loss: 0.3762 | Iterations: 55000 | Time: 817.869
Loss: 0.3996 | Iterations: 60000 | Time: 894.173
Loss: 0.3859 | Iterations: 65000 | Time: 972.227
Training Loss: 0.3884
Training Accuracy: 0.6991
Validation Accuracy: 0.5654
Epoch: 7
Loss: 0.3610 | Iterations: 5000 | Time: 77.543
Loss: 0.3736 | Iterations: 10000 | Time: 157.138
Loss: 0.3762 | Iterations: 15000 | Time: 235.739
Loss: 0.3578 | Iterations: 20000 | Time: 316.097
Loss: 0.3652 | Iterations: 25000 | Time: 393.181
Loss: 0.3633 | Iterations: 30000 | Time: 472.511
```

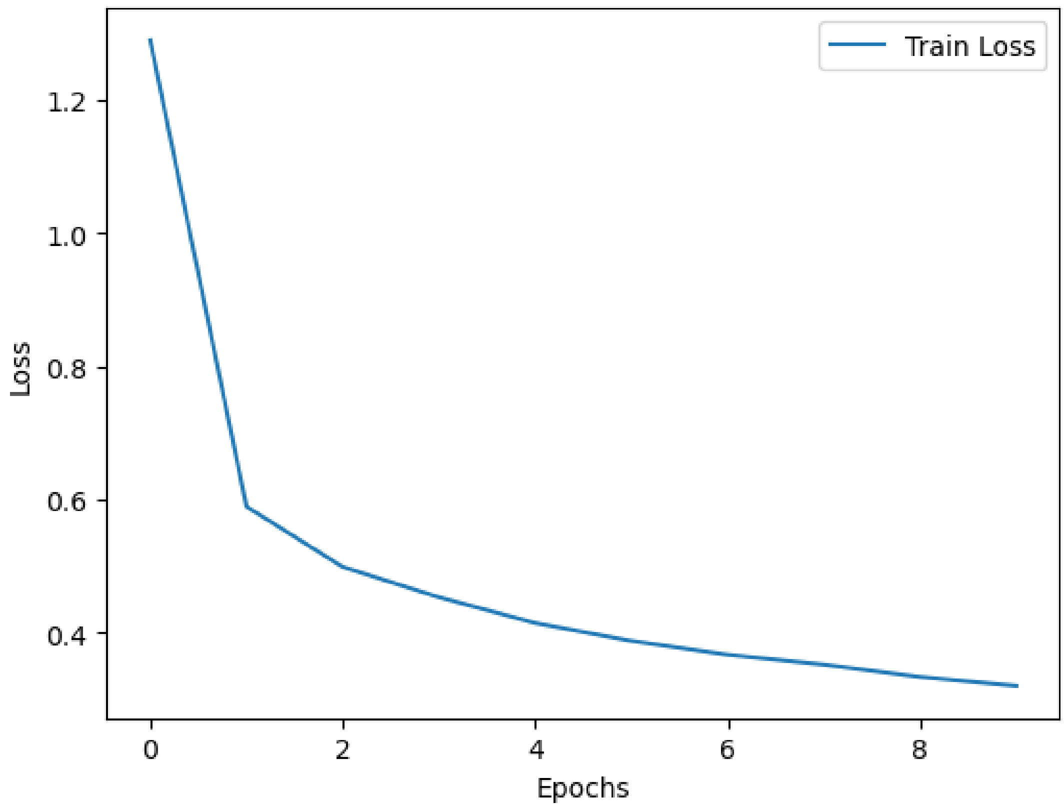
```
Loss: 0.3681 | Iterations: 35000 | Time: 555.723
Loss: 0.3781 | Iterations: 40000 | Time: 633.270
Loss: 0.3782 | Iterations: 45000 | Time: 714.262
Loss: 0.3680 | Iterations: 50000 | Time: 792.417
Loss: 0.3577 | Iterations: 55000 | Time: 870.202
Loss: 0.3830 | Iterations: 60000 | Time: 946.142
Loss: 0.3593 | Iterations: 65000 | Time: 1021.624
Training Loss: 0.3678
Training Accuracy: 0.6924
Validation Accuracy: 0.5622
Epoch: 8
Loss: 0.3424 | Iterations: 5000 | Time: 81.970
Loss: 0.3504 | Iterations: 10000 | Time: 160.628
Loss: 0.3366 | Iterations: 15000 | Time: 239.138
Loss: 0.3653 | Iterations: 20000 | Time: 316.910
Loss: 0.3381 | Iterations: 25000 | Time: 394.848
Loss: 0.3650 | Iterations: 30000 | Time: 473.577
Loss: 0.3255 | Iterations: 35000 | Time: 550.536
Loss: 0.3553 | Iterations: 40000 | Time: 630.661
Loss: 0.3744 | Iterations: 45000 | Time: 707.624
Loss: 0.3515 | Iterations: 50000 | Time: 783.059
Loss: 0.3708 | Iterations: 55000 | Time: 858.600
Loss: 0.3572 | Iterations: 60000 | Time: 932.349
Loss: 0.3631 | Iterations: 65000 | Time: 1004.714
Training Loss: 0.3529
Training Accuracy: 0.7210
Validation Accuracy: 0.5742
Epoch: 9
Loss: 0.3250 | Iterations: 5000 | Time: 75.163
Loss: 0.3286 | Iterations: 10000 | Time: 150.939
Loss: 0.3234 | Iterations: 15000 | Time: 225.936
Loss: 0.3229 | Iterations: 20000 | Time: 302.095
Loss: 0.3385 | Iterations: 25000 | Time: 373.813
Loss: 0.3323 | Iterations: 30000 | Time: 445.833
Loss: 0.3474 | Iterations: 35000 | Time: 517.990
Loss: 0.3213 | Iterations: 40000 | Time: 589.402
Loss: 0.3442 | Iterations: 45000 | Time: 659.495
Loss: 0.3420 | Iterations: 50000 | Time: 728.784
Loss: 0.3345 | Iterations: 55000 | Time: 790.503
Loss: 0.3461 | Iterations: 60000 | Time: 845.181
Loss: 0.3276 | Iterations: 65000 | Time: 913.235
Training Loss: 0.3344
Training Accuracy: 0.7208
Validation Accuracy: 0.5626
Epoch: 10
Loss: 0.3002 | Iterations: 5000 | Time: 44.593
Loss: 0.3042 | Iterations: 10000 | Time: 88.928
Loss: 0.3210 | Iterations: 15000 | Time: 133.665
```

```
Loss: 0.3079 | Iterations: 20000 | Time: 183.219
Loss: 0.3316 | Iterations: 25000 | Time: 268.201
Loss: 0.3080 | Iterations: 30000 | Time: 354.579
Loss: 0.3110 | Iterations: 35000 | Time: 441.995
Loss: 0.3309 | Iterations: 40000 | Time: 527.984
Loss: 0.3233 | Iterations: 45000 | Time: 614.574
Loss: 0.3154 | Iterations: 50000 | Time: 702.614
Loss: 0.3227 | Iterations: 55000 | Time: 788.615
Loss: 0.3429 | Iterations: 60000 | Time: 862.546
Loss: 0.3291 | Iterations: 65000 | Time: 913.468
Training Loss: 0.3210
Training Accuracy: 0.7244
Validation Accuracy: 0.5556
Training Complete
Testing Model
Test Accuracy: 0.5345
Testing Complete
```

[ ]:
```python
import matplotlib.pyplot as plt

def plot_accuracy(train_accuracies, val_accuracies):
    plt.plot(train_accuracies, label="Train")
    plt.plot(val_accuracies, label="Validation")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.show()

plot_accuracy(train_accuracies, val_accuracies)
```




```
[ ]: def plot_loss(train_losses):
         plt.plot(train_losses, label="Train Loss")
         plt.xlabel("Epochs")
         plt.ylabel("Loss")
         plt.legend()
         plt.show()

     plot_loss(train_losses)
```




```
[ ]: # Store the model and results

import pickle

with open("../models/model_attn.pkl", "wb") as f:
    pickle.dump(model, f)

with open("../models/results_attn.pkl", "wb") as f:
    results = {
        "train_losses": train_losses,
        "train_accuracies": train_accuracies,
        "val_accuracies": val_accuracies,
        "test_accuracy": test_accuracy
    }
    pickle.dump(results, f)
```

```
[ ]: # Generate predictions

def generate_predictions(lang: str, model: Translator, type: str):
    pairs = get_data(lang, type)
    data = [pair[1] for pair in pairs]
```

```python
    predictions = [model.evaluate(pair[1]) for pair in pairs]
    return data, predictions

# Store predictions

def store_predictions(lang: str, model: Translator, type: str):
    data, predictions = generate_predictions(lang, model, type)
    df = pd.DataFrame({'data': data, 'predictions': predictions})
    df.to_csv(f'../predictions/{lang}_{type}_predictions_attn.csv', index=False)

store_predictions(language, model, 'train')
store_predictions(language, model, 'dev')
store_predictions(language, model, 'test')
```

[ ]:

```python
# Load the model and results

model = None
results = None

with open("../models/model_attn.pkl", "rb") as f:
    model = pickle.load(f)

with open("../models/results_attn.pkl", "rb") as f:
    results = pickle.load(f)

print(results)
```