Problem-3-RNN

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, device: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],␣
      ↪device:str):
         input_tensor = tensorFromWord(input_lang, pair[1], device)
         target_tensor = tensorFromWord(output_lang, pair[0], device)
         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,
                      device: str):

             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.device = device

             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=self.device)
```

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

        super(Decoder, 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.device = device

        self.rnn = get_cell(cell_type)(input_size = embed_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):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)

        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

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

```
[ ]: class Translator:
         def __init__(self, lang: str, params: dict, device: str):
             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.device = device

             self.training_pairs = [tensorsFromPair(self.input_lang, self.
      ↪output_lang, pair, self.device) 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"],
                                    device=self.device).to(self.device)

             self.decoder = Decoder(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"],
                                    device=self.device).to(self.device)

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

             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=self.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=self.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 = self.
↪decoder(decoder_input, decoder_hidden, decoder_cell)
                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 = self.
↪decoder(decoder_input, decoder_hidden, decoder_cell)
                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+1):
            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, self.device)
            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=self.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=self.device)
            decoder_hidden, decoder_cell = encoder_hidden, encoder_cell

            decoded_chars = ""

            for di in range(self.max_length):
                decoder_output, decoder_hidden, decoder_cell = self.
↪decoder(decoder_input, decoder_hidden, decoder_cell)
                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

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

[ ]:
```
params = {
    "embed_size": 8,
    "hidden_size": 256,
    "cell_type": "LSTM",
    "num_layers": 3,
    "dropout": 0.2,
    "learning_rate": 0.005,
    "optimizer": "SGD",
    "teacher_forcing_ratio": 0.5,
    "max_length": 50
}
```

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

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

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

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: 2.8841 | Iterations: 5000 | Time: 64.635
Loss: 2.6962 | Iterations: 10000 | Time: 131.616
Loss: 2.6588 | Iterations: 15000 | Time: 199.169
Loss: 2.6402 | Iterations: 20000 | Time: 266.864
Loss: 2.6112 | Iterations: 25000 | Time: 334.496
Loss: 2.5487 | Iterations: 30000 | Time: 403.456
Loss: 2.4545 | Iterations: 35000 | Time: 471.179
Loss: 2.2946 | Iterations: 40000 | Time: 538.222
Loss: 2.1806 | Iterations: 45000 | Time: 606.455
Loss: 2.0885 | Iterations: 50000 | Time: 676.911
Loss: 2.0331 | Iterations: 55000 | Time: 745.801
Loss: 1.9481 | Iterations: 60000 | Time: 814.212
Loss: 1.8541 | Iterations: 65000 | Time: 882.523
Training Loss: 2.3470
Training Accuracy: 0.0095
Validation Accuracy: 0.0144
Epoch: 2
Loss: 1.6591 | Iterations: 5000 | Time: 68.932
Loss: 1.5533 | Iterations: 10000 | Time: 137.578
Loss: 1.4129 | Iterations: 15000 | Time: 205.870
Loss: 1.3510 | Iterations: 20000 | Time: 279.407
```

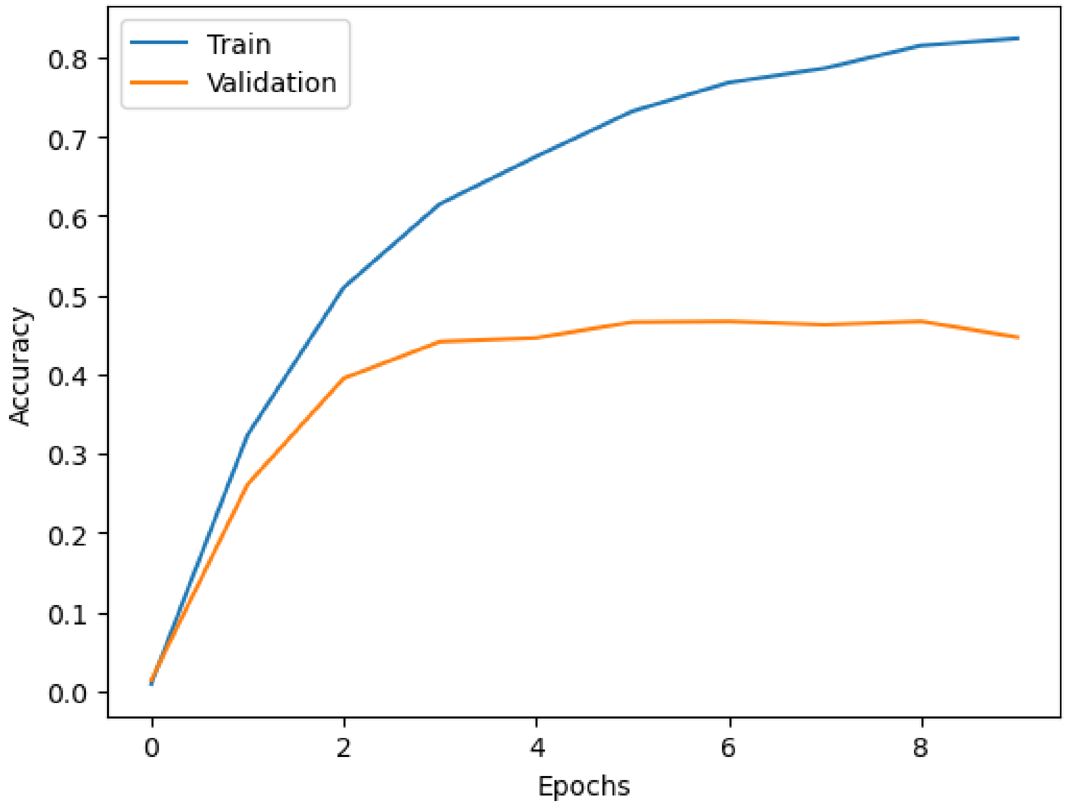
```
Loss: 1.2734 | Iterations: 25000 | Time: 355.935
Loss: 1.1833 | Iterations: 30000 | Time: 430.812
Loss: 1.1081 | Iterations: 35000 | Time: 505.340
Loss: 1.0521 | Iterations: 40000 | Time: 579.302
Loss: 0.9937 | Iterations: 45000 | Time: 653.348
Loss: 0.9575 | Iterations: 50000 | Time: 727.240
Loss: 0.9245 | Iterations: 55000 | Time: 802.384
Loss: 0.8868 | Iterations: 60000 | Time: 877.215
Loss: 0.8832 | Iterations: 65000 | Time: 951.421
Training Loss: 1.1561
Training Accuracy: 0.3234
Validation Accuracy: 0.2610
Epoch: 3
Loss: 0.7862 | Iterations: 5000 | Time: 67.838
Loss: 0.7473 | Iterations: 10000 | Time: 133.613
Loss: 0.7460 | Iterations: 15000 | Time: 199.569
Loss: 0.7081 | Iterations: 20000 | Time: 265.998
Loss: 0.7018 | Iterations: 25000 | Time: 333.657
Loss: 0.7027 | Iterations: 30000 | Time: 400.682
Loss: 0.6822 | Iterations: 35000 | Time: 469.765
Loss: 0.6277 | Iterations: 40000 | Time: 539.638
Loss: 0.6668 | Iterations: 45000 | Time: 609.418
Loss: 0.6133 | Iterations: 50000 | Time: 679.772
Loss: 0.6219 | Iterations: 55000 | Time: 750.811
Loss: 0.6042 | Iterations: 60000 | Time: 822.436
Loss: 0.5937 | Iterations: 65000 | Time: 895.374
Training Loss: 0.6724
Training Accuracy: 0.5097
Validation Accuracy: 0.3950
Epoch: 4
Loss: 0.5543 | Iterations: 5000 | Time: 74.924
Loss: 0.5249 | Iterations: 10000 | Time: 148.160
Loss: 0.5505 | Iterations: 15000 | Time: 219.842
Loss: 0.5303 | Iterations: 20000 | Time: 289.522
Loss: 0.5051 | Iterations: 25000 | Time: 360.734
Loss: 0.5095 | Iterations: 30000 | Time: 432.505
Loss: 0.5001 | Iterations: 35000 | Time: 506.002
Loss: 0.5398 | Iterations: 40000 | Time: 577.962
Loss: 0.5193 | Iterations: 45000 | Time: 646.905
Loss: 0.4943 | Iterations: 50000 | Time: 716.285
Loss: 0.4739 | Iterations: 55000 | Time: 786.774
Loss: 0.4661 | Iterations: 60000 | Time: 856.964
Loss: 0.4618 | Iterations: 65000 | Time: 926.777
Training Loss: 0.5086
Training Accuracy: 0.6152
Validation Accuracy: 0.4413
Epoch: 5
Loss: 0.4034 | Iterations: 5000 | Time: 66.382
```

```
Loss: 0.4215 | Iterations: 10000 | Time: 134.385
Loss: 0.4182 | Iterations: 15000 | Time: 203.856
Loss: 0.4258 | Iterations: 20000 | Time: 271.774
Loss: 0.4171 | Iterations: 25000 | Time: 341.327
Loss: 0.4260 | Iterations: 30000 | Time: 408.939
Loss: 0.4194 | Iterations: 35000 | Time: 475.203
Loss: 0.4039 | Iterations: 40000 | Time: 543.228
Loss: 0.4218 | Iterations: 45000 | Time: 608.980
Loss: 0.4049 | Iterations: 50000 | Time: 674.463
Loss: 0.3917 | Iterations: 55000 | Time: 742.140
Loss: 0.4137 | Iterations: 60000 | Time: 807.648
Loss: 0.4226 | Iterations: 65000 | Time: 873.393
Training Loss: 0.4143
Training Accuracy: 0.6750
Validation Accuracy: 0.4462
Epoch: 6
Loss: 0.3535 | Iterations: 5000 | Time: 70.955
Loss: 0.3561 | Iterations: 10000 | Time: 141.332
Loss: 0.3530 | Iterations: 15000 | Time: 210.727
Loss: 0.3470 | Iterations: 20000 | Time: 279.939
Loss: 0.3653 | Iterations: 25000 | Time: 349.451
Loss: 0.3511 | Iterations: 30000 | Time: 419.522
Loss: 0.3614 | Iterations: 35000 | Time: 489.426
Loss: 0.3432 | Iterations: 40000 | Time: 561.367
Loss: 0.3504 | Iterations: 45000 | Time: 630.619
Loss: 0.3392 | Iterations: 50000 | Time: 701.133
Loss: 0.3485 | Iterations: 55000 | Time: 766.848
Loss: 0.3469 | Iterations: 60000 | Time: 833.770
Loss: 0.3258 | Iterations: 65000 | Time: 900.215
Training Loss: 0.3488
Training Accuracy: 0.7322
Validation Accuracy: 0.4659
Epoch: 7
Loss: 0.3086 | Iterations: 5000 | Time: 68.678
Loss: 0.2964 | Iterations: 10000 | Time: 138.494
Loss: 0.2871 | Iterations: 15000 | Time: 207.756
Loss: 0.3069 | Iterations: 20000 | Time: 277.667
Loss: 0.3088 | Iterations: 25000 | Time: 349.355
Loss: 0.3086 | Iterations: 30000 | Time: 420.816
Loss: 0.3084 | Iterations: 35000 | Time: 492.012
Loss: 0.3068 | Iterations: 40000 | Time: 561.412
Loss: 0.3083 | Iterations: 45000 | Time: 633.633
Loss: 0.3011 | Iterations: 50000 | Time: 704.236
Loss: 0.3099 | Iterations: 55000 | Time: 774.678
Loss: 0.2941 | Iterations: 60000 | Time: 846.557
Loss: 0.2984 | Iterations: 65000 | Time: 919.248
Training Loss: 0.3028
Training Accuracy: 0.7682
```

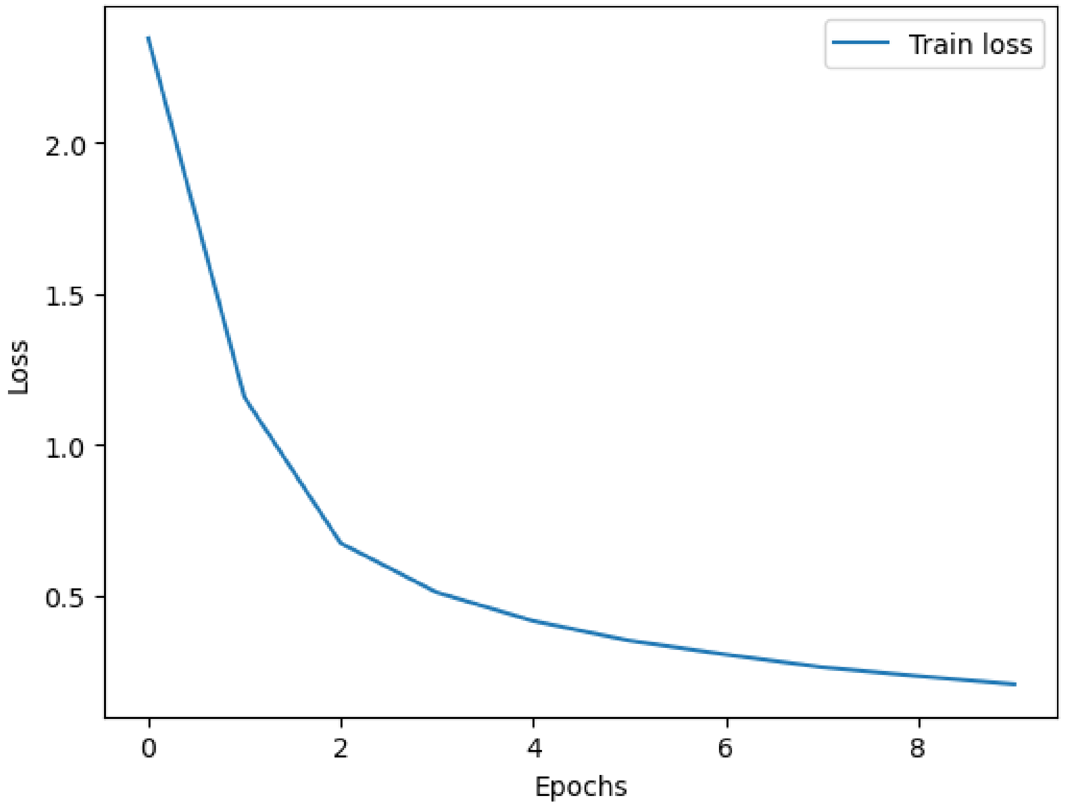
```
Validation Accuracy: 0.4670
Epoch: 8
Loss: 0.2381 | Iterations: 5000 | Time: 70.976
Loss: 0.2535 | Iterations: 10000 | Time: 142.201
Loss: 0.2541 | Iterations: 15000 | Time: 214.768
Loss: 0.2455 | Iterations: 20000 | Time: 289.409
Loss: 0.2619 | Iterations: 25000 | Time: 362.886
Loss: 0.2844 | Iterations: 30000 | Time: 437.327
Loss: 0.2692 | Iterations: 35000 | Time: 510.346
Loss: 0.2637 | Iterations: 40000 | Time: 583.203
Loss: 0.2656 | Iterations: 45000 | Time: 658.213
Loss: 0.2697 | Iterations: 50000 | Time: 730.779
Loss: 0.2607 | Iterations: 55000 | Time: 805.356
Loss: 0.2652 | Iterations: 60000 | Time: 878.202
Loss: 0.2532 | Iterations: 65000 | Time: 951.508
Training Loss: 0.2608
Training Accuracy: 0.7865
Validation Accuracy: 0.4627
Epoch: 9
Loss: 0.2218 | Iterations: 5000 | Time: 72.184
Loss: 0.2206 | Iterations: 10000 | Time: 143.360
Loss: 0.2215 | Iterations: 15000 | Time: 216.000
Loss: 0.2243 | Iterations: 20000 | Time: 289.040
Loss: 0.2333 | Iterations: 25000 | Time: 362.066
Loss: 0.2260 | Iterations: 30000 | Time: 436.978
Loss: 0.2411 | Iterations: 35000 | Time: 510.349
Loss: 0.2465 | Iterations: 40000 | Time: 583.791
Loss: 0.2160 | Iterations: 45000 | Time: 656.920
Loss: 0.2289 | Iterations: 50000 | Time: 730.024
Loss: 0.2310 | Iterations: 55000 | Time: 803.585
Loss: 0.2403 | Iterations: 60000 | Time: 875.832
Loss: 0.2382 | Iterations: 65000 | Time: 950.775
Training Loss: 0.2312
Training Accuracy: 0.8150
Validation Accuracy: 0.4670
Epoch: 10
Loss: 0.1971 | Iterations: 5000 | Time: 68.289
Loss: 0.1992 | Iterations: 10000 | Time: 136.357
Loss: 0.1997 | Iterations: 15000 | Time: 206.243
Loss: 0.2048 | Iterations: 20000 | Time: 275.213
Loss: 0.2109 | Iterations: 25000 | Time: 344.570
Loss: 0.1971 | Iterations: 30000 | Time: 413.992
Loss: 0.1923 | Iterations: 35000 | Time: 483.569
Loss: 0.2026 | Iterations: 40000 | Time: 553.177
Loss: 0.2099 | Iterations: 45000 | Time: 623.511
Loss: 0.2186 | Iterations: 50000 | Time: 694.236
Loss: 0.2038 | Iterations: 55000 | Time: 765.466
Loss: 0.2035 | Iterations: 60000 | Time: 836.808
```

```
Loss: 0.2210 | Iterations: 65000 | Time: 908.134
Training Loss: 0.2041
Training Accuracy: 0.8242
Validation Accuracy: 0.4470
Training Complete
Testing Model
Test Accuracy: 0.4496
Testing Complete
```

[ ]:
```
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_rnn.pkl", "wb") as f:
    pickle.dump(model, f)

with open("../models/results_rnn.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]
    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.csv', index=False)

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

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

model = None
results = None

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

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

print(results)
```