Uppsala University

Deep learning

Hand-in assignment (3)

Tong You

June 10, 2021

Contents

1 Introduction

Whereas in the previous two assignments we were working with image data (MNIST and Warwick data), in this final assignment we will implement a language model. We will be using the PennTreebank (PTB) as our dataset. This is a collection of articles from the Wall Street Journal. We will implement the language model in several stages. First, we need to preprocess the data. Next we will implement a (simple) Elmann RNN. Finally, we will look at several different ways to improve the model. We will show generated example phrases at each stage in order to show the differences. The model will be implemented in Pytorch using an RTX 2080 Ti in order to do GPU training.

2 The dataset

As mentioned previously, we will be using the PTB dataset. This is split up into three splits: a training split, validation split, and testing split. Before we can feed this data into our language model we need to preprocess the data, generate a dictionary with our keys being the unique words in the three splits and the values being an unique integer corresponding to a particular unique word. Before we proceed, we will be showing some example sentences. We pick two random sentences from each split.

- Example training set phrase: in addition the cray-3 will contain N processors twice as many as the largest current supercomputer
- Example training set phrase: what 's more the test and learning materials are both produced by the same company <unk> a joint venture of mcgraw-hill inc. and macmillan 's parent britain 's maxwell communication corp
- Example validation set phrase: the ghost of the soviet <unk> discovered in cuba back in the <unk> costs just a few hundred million the price of the caribbean command in key west that president carter created in N
- Example validation set phrase: mr. wolf has <unk> merger advice from a major wall street securities firm relying instead only on a takeover lawyer peter <unk> of <unk> <unk> slate <unk> & flom

- Example test set phrase: the executive denied speculation that saatchi was bringing in the new chief executive officer only to clean up the company financially so that the brothers could lead a buy-back
- Example test set phrase: quantum 's lot is mostly tied to polyethylene <unk> used to make garbage bags milk <unk> <unk> toys and meat packaging among other items

The next step is to add an additional term <eos> that denotes the end of a sentence. All code will be attached in the Appendix. Next we want to check the size of the three splits. For this count we consider the special characters <eos> and <unk>. The size of the splits:

- Number of training words: 929589
- Number of validation words: 73760
- Number of testing words: 82430

From the above we want to generate a dictionary as mentioned earlier. In order to generate the dictionary we first flatten the three splits and combine the splits into one big list. Then we use the set function in Python. This will generate a set with the unique words. From this we can easily see how many unique words there are. There are 10000 unique words in the entire PTB dataset. Finally, we use this dictionary to map our words to a unique integer. To generate sample phrases we would look up the word corresponding to any unique integer. Our input to the RNN will therefore consist of a vector of integers. Next we will look at the language model.

3 A simple language model

In the previous section we did some preprocessing on the PTB dataset. Now we are ready to implement a simple RNN using PyTorch. For implementation details we refer to the Appendix, however code that is deemed important is explained in the following text. After some initial tweaking, an "ideal" set of hyperparameters was chosen. The test perplexity still doesn't reach the value of a highly-optimized Elman RNN. However, we will implement some methods to try to lower the perplexity. In order to make an one-to-one comparison between the sections we will use the same input sentence(s) when generating sample phrases.

3.1 Results

To start with, we will show results of a vanilla RNN without any gradient clipping. In order to highlight differences between different methods we will try to use the same parameters in order to really show the gains that were made. With normal SGD it was very difficult get a low perplexity that starts to approach a well-tuned Elmann RNN. The following loss curves correspond to a training batch size of 32 and validation/testing batch sizes of 512. The sequence length for all splits was 50. Both the hidden and embedding dimensions were set to 500. The RNN was implemented with a layer number of 1 first. The loss was averaged each epoch to yield less noisier training/validation curves. We will also use dropout since that really improved the training for the simple RNN (value of 0.5). To start with, we will show results of a vanilla RNN without any gradient clipping. In order to highlight differences between different methods we will try to use the same parameters in order to really show the gains that were made. With normal SGD it was very difficult get a low perplexity that starts to approach a well-tuned Elmann RNN. The following loss curves correspond to a training batch size of 32 and validation/testing batch sizes of 512. The sequence length for all splits was 50. Both the hidden and embedding dimensions were set to 500. The RNN was implemented with a layer number of 1 first. The loss was averaged each epoch to yield less noisier training/validation curves. We will also use dropout since that really improved the training for the simple RNN (value of 0.5). The learning rate was 0.5. The loss curve below shows the results for the simple RNN. This model was then evaluated on the test split and we got a perplexity of 214.15121520493514. This is still quite high, but not as high as initial testing of parameters.

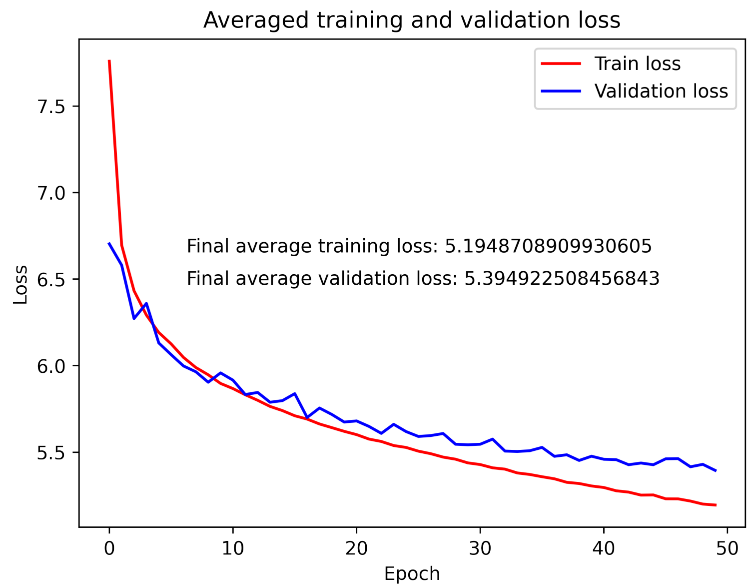

Figure 1: Training and validation loss averaged over all iterations each epoch. The final losses are annotated in the figure. The RNN has one layer.

We will see in the next section how we can improve this.

4 Improving the recurrent model

As mentioned earlier, there are several ways to improve upon the previous result. In addition we will also show examples of generated phrases by providing a seeding input to the trained model.

4.1 Results - gradient clipping

We will first show some results of gradient clipping applied to a two-layer RNN. We will use the same parameters, but after some preliminary testing we found that we can use quite high learning rates, so we chose 10. We clipped the gradients at 0.5 using the L-2 norm. The one-layer RNN got higher perplexity than the one-layer network with perplexity in testing so we decided to show results for a two-layer network with gradient clipping.

This network gets higher test perplexity than the above network. The test perplexity after 50 epochs was 247.3608380295578.

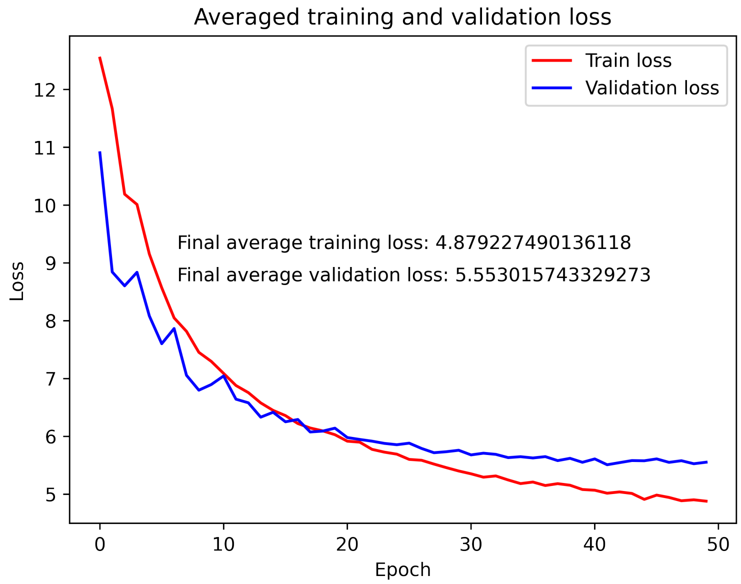

Figure 2: Training and validation loss averaged over all iterations each epoch. Training was done with gradient clipping. The final losses are annotated in the figure. The RNN has two layers.

Next we will show a generated phrase giving an input sequence. We will use a greedy method. However, the current model outputs the same word when we specify how many words we want to generate. Several attempts to fix this didn't work, so only one generated phrase is shown. However, it seems that the model outputs the most frequent word. The seeding sequence is 2 words long and is created by using uniformly generated random integers up to the number of unique words. We started with "there" and "unrelated".

- Generated phrase: "there" "unrelated" "with" "with" "with"

4.2 Results - LSTM

Since the Elmann RNN is quite "simple", we will next use an LSTM. These are much less prone to exploding gradients, although we didn't encounter such problems for our implementation. We will use the exact same parameters as the results in the subsection above, only changing the RNN to an LSTM. For ease of reading, we will give them again. For the training we used a batch size of 32 and for validation/testing we evaluated on data with batch sizes of 512. The sequences were of length 50 for all splits. We also used the same embedding and hidden dimensions of 500. And the LSTM in this case has two layers. We also found that a dropout of 0.8 yielded quite good results. The learning rate was set to 10.0 with gradient clipping with a max L2-norm of 0.5. We also trained for 50 epochs using standard SGD. The final test perplexity was 159.23738342645728.

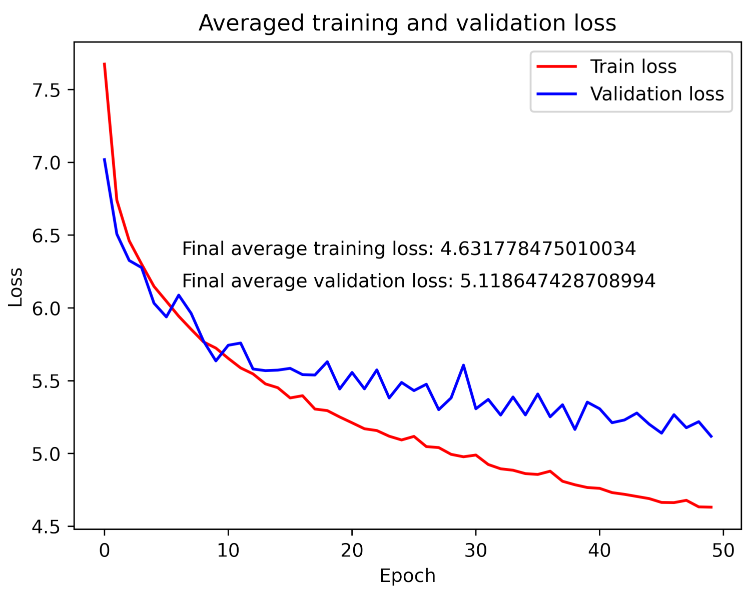

Figure 3: Training and validation loss averaged over all iterations each epoch. Training was done with gradient clipping. The final losses are annotated in the figure. Instead of a two-layer RNN, a two-layer LSTM was trained.

Next we show an example of a generated phrase. We used as our input sequence: ”there” and ”competitiveness”. We wanted to generate three words.

- Generated phrase: ”there” ”competitiveness” ”is” ”<unk>” ”<unk>”.

4.3 Results - additional modifications

There are different changes we can apply to our current model and see if we can spot any improvements. We will use the same settings as used above, but swap the LSTM with an GRU. Training will be done in the same way.

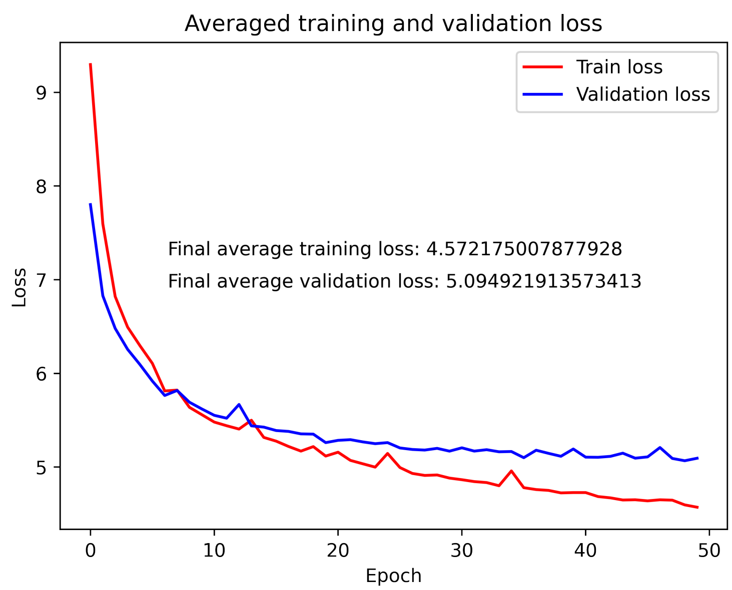

Figure 4: Training and validation loss averaged over all iterations each epoch. Training was done with gradient clipping. The final losses are annotated in the figure. A two-layer GRU was used to train the PTB dataset.

The final test perplexity was slightly lower than the LSTM: 157.50157610436733. An example of a generated phrase:

- Generated phrase: "there" "unknown" "is" "a" "<unk>".

The trained model was seeded with a sequence in a similar way as above. The model generates much more promising results, however it still outputs the same word after a few generations. We will finally try to improve on the previous model by using a custom learning rate scheduler. We will still use the GRU but instead use a cosine learning rate scheduler and train for 50 epochs again. We will use SGD again with a learning rate of 10, but this will be modified by the cosine learning rate scheduler. The scheduler was implemented as follows:

```
cosine_sched = optim.lr_scheduler.CosineAnnealingLR(optimizer,
↪   T_max = 500, eta_min = 0.1, last_epoch = -1, verbose =
↪   True)
```

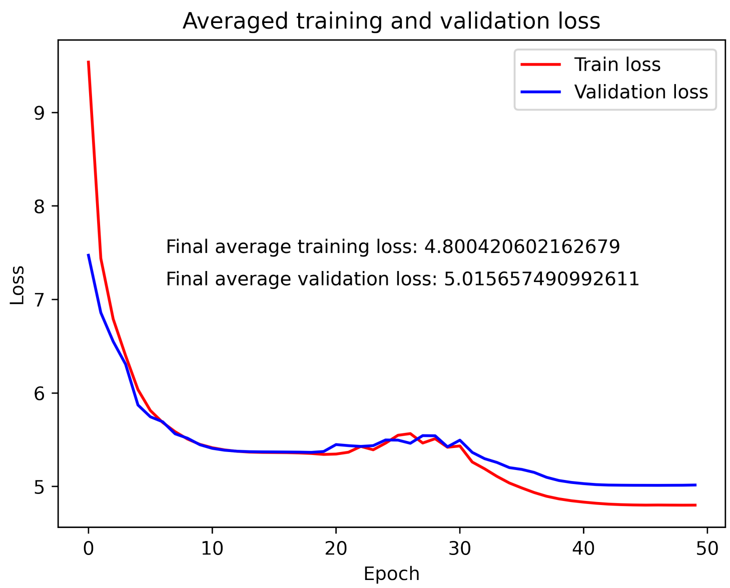

Figure 5: Training and validation loss averaged over all iterations each epoch. Training was done with gradient clipping. The final losses are annotated in the figure. A two-layer GRU was used to train the PTB dataset. This time a cosine learning rate scheduler was used.

With the same parameters as the GRU with SGD and no learning rate scheduling, we now get a final perplexity of 146.02419039272354. This is about an 11 point difference compared to the GRU without rate scheduling. We can see in the loss curves that the loss peaked a bit after about 25 epochs, but decreased again afterwards. Better results might be expected by combining an exponential decay scheduler with a cosine learning rate scheduler. Despite the improvements in perplexity using the same seeding mechanism as we had previously the generated phrases still repeat the same word after only a few words:

- Generated phrase: "there" "brewing" "is" "a" "<unk>".

5 Appendix

Since the code was implemented in a single Jupyter Notebook, the code in the different cells will be attached here as a single file.

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

import functools
import operator
import random
import pickle

import numpy as np

from sklearn.metrics import confusion_matrix
from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader
from matplotlib import pyplot as plt

# Exercise 1
# Training data
keepNew = True
with open('PTB/ptb.train.txt', mode = 'r', newline = None) as train_f:
    train_dat = train_f.read().splitlines(keepNew)

# Validation data
with open('PTB/ptb.valid.txt', mode = 'r', newline = None) as valid_f:
    valid_dat = valid_f.read().splitlines(keepNew)
```

```

# Testing data
with open('PTB/ptb.test.txt', mode = 'r', newline = None) as
↪  test_f:
    test_dat = test_f.read().splitlines(keepNew)

# Printing examples phrases
n_examp = np.random.choice(1000, size = 6, replace = False)

print("Example training set phrase:", train_dat[n_examp[0]])
print("Example training set phrase:", train_dat[n_examp[1]])
print("Example validation set phrase:", valid_dat[n_examp[2]])
print("Example validation set phrase:", valid_dat[n_examp[3]])
print("Example test set phrase:", test_dat[n_examp[4]])
print("Example test set phrase:", test_dat[n_examp[5]])

# Adding <eos>
train_proc, valid_proc, test_proc = [], [], []
for l_train in train_dat:
    l_mod = l_train.replace('\n', '<eos>')
    train_proc.append(l_mod)

for l_valid in valid_dat:
    l_mod = l_valid.replace('\n', '<eos>')
    valid_proc.append(l_mod)

for l_train in test_dat:
    l_mod = l_train.replace('\n', '<eos>')
    test_proc.append(l_mod)

# Split up each line in individual words
train_words, valid_words, test_words = [], [], []
for tp in train_proc:
    train_words.append(tp.split())

for tp in valid_proc:
    valid_words.append(tp.split())

```

```python
for tp in test_proc:
    test_words.append(tp.split())

# Flatten list of lists into a single list
train_words = functools.reduce(operator.iconcat, train_words,
↪   [])
valid_words = functools.reduce(operator.iconcat, valid_words,
↪   [])
test_words = functools.reduce(operator.iconcat, test_words,
↪   [])

num_train = len(train_words)
num_valid = len(valid_words)
num_test = len(test_words)

print("Number of training words:", num_train)
print("Number of validation words:", num_valid)
print("Number of testing words:", num_test)

# Building a dictionary
all_words = train_words + valid_words + test_words
set_words = set(all_words)
num_unique = len(set_words)

print("Number of unique words in training + validation +
↪   testing splits:", num_unique)

num_id = np.random.choice(num_unique, size = num_unique,
↪   replace = False)

n = 0
unique_dict = {}
for uw in set_words:
    unique_dict.update({uw : num_id[n]})
    n += 1

# Replacing all words in the training/validation/testing
↪   splits with their integer representation
```

```python
train_ints, valid_ints, test_ints = [], [], []

for word in train_words:
    int_rep = unique_dict[word]
    train_ints.append(int_rep)

for word in valid_words:
    int_rep = unique_dict[word]
    valid_ints.append(int_rep)

for word in test_words:
    int_rep = unique_dict[word]
    test_ints.append(int_rep)

# Check if CUDA is available
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
print("Device:", torch.cuda.get_device_name(device))

# Resetting model function
# Credits: https://discuss.pytorch.org/t/reset-model-weights/19180/4
def reset_model(model):
    for layer in model.children():
       if hasattr(layer, 'reset_parameters'):
           layer.reset_parameters()

# Exercise 2
# Convert our training/validation/testing splits to Torch tensors
train_dat = torch.tensor(train_ints)
valid_dat = torch.tensor(valid_ints)
test_dat = torch.tensor(test_ints)

### Data loading
batch_size = 32
batch_eval = 512
batch_test = 512
```

```
seq_train = 50
seq_valid = 50
seq_test = 50

s_train_l = num_train // seq_train
s_valid_l = num_valid // seq_valid
s_test_l = num_test // seq_test

# Trim training/validation/testing data and reshape into tensor of num_sequences by sequence_length
# Training data
train_seq = torch.narrow(train_dat, 0, 0, seq_train * s_train_l)
train_lab = torch.roll(train_seq, shifts = -1, dims = 0)

train_seq = train_seq.reshape(s_train_l, seq_train)
train_lab_seq = train_lab.reshape(s_train_l, seq_train)

# Validation data
valid_seq = torch.narrow(valid_dat, 0, 0, seq_valid * s_valid_l)
valid_lab = torch.roll(valid_seq, shifts = -1, dims = 0)

valid_seq = valid_seq.reshape(s_valid_l, seq_valid)
valid_lab_seq = valid_lab.reshape(s_valid_l, seq_valid)

# Testing data
test_seq = torch.narrow(test_dat, 0, 0, seq_test * s_test_l)
test_lab = torch.roll(test_seq, shifts = -1, dims = 0)

test_seq = test_seq.reshape(s_test_l, seq_test)
test_lab_seq = test_lab.reshape(s_test_l, seq_test)

# Divide training and validation data into correct mini-batches
num_batches = train_seq.shape[0] // batch_size
valid_batches = valid_seq.shape[0] // batch_eval
test_batches = test_seq.shape[0] // batch_test
```

```
training_set = TensorDataset(train_seq.to(device),
 ↪  train_lab_seq.to(device))
training_loader = DataLoader(training_set, shuffle = False,
 ↪  batch_size = num_batches)

valid_set = TensorDataset(valid_seq.to(device),
 ↪  valid_lab_seq.to(device))
valid_loader = DataLoader(valid_set, shuffle = False,
 ↪  batch_size = valid_batches)

test_set = TensorDataset(test_seq.to(device),
 ↪  test_lab_seq.to(device))
test_loader = DataLoader(test_set, shuffle = False, batch_size
 ↪  = test_batches)

### RNN code
# Embedding parameter
embed_dim = 500

# RNN parameters
hidden_dim = 500
in_size = embed_dim
n_layers = 2

# Vanilla RNN
class ElmanRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers,
     ↪  num_embeddings, embedding_dim, hidden_dim, num_unique,
     ↪  drop_out):
        super(ElmanRNN, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layer = num_layers
        self.drop_out = drop_out
        self.num_embeddings = num_embeddings
        self.embedding_dim = embedding_dim
```

```python

        self.embed = nn.Embedding(num_embeddings, embedding_dim)
        self.elman = nn.RNN(input_size, hidden_size, num_layers, dropout = drop_out, batch_first = True)
        self.linear = nn.Linear(hidden_dim, num_unique)

    def forward(self, mod_input):
        word_embed = self.embed(mod_input)
        rnn_out, hidden_out = self.elman(word_embed)
        rnn_out = self.linear(rnn_out)
        rnn_out = rnn_out.view(-1, num_unique)

        return rnn_out, hidden_out

# LSTM
class LSTMRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_embeddings, embedding_dim, hidden_dim, num_unique, drop_out):
        super(LSTMRNN, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layer = num_layers
        self.drop_out = drop_out
        self.num_embeddings = num_embeddings
        self.embedding_dim = embedding_dim

        self.embed = nn.Embedding(num_embeddings, embedding_dim)
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, dropout = drop_out, batch_first = True, bidirectional = False)
        self.linear = nn.Linear(hidden_dim, num_unique)

    def forward(self, mod_input):
```

```
        word_embed = self.embed(mod_input)
        lstm_out, hidden_out = self.lstm(word_embed)
        lstm_out = self.linear(lstm_out)
        lstm_out = lstm_out.view(-1, num_unique)

        return lstm_out, hidden_out

# GRU
class GRURNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers,
     ↪  num_embeddings, embedding_dim, hidden_dim, num_unique,
     ↪  drop_out):
        super(GRURNN, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layer = num_layers
        self.drop_out = drop_out
        self.num_embeddings = num_embeddings
        self.embedding_dim = embedding_dim

        self.embed = nn.Embedding(num_embeddings,
         ↪  embedding_dim)
        self.gru = nn.GRU(input_size, hidden_size, num_layers,
         ↪  dropout = drop_out, batch_first = True,
         ↪  bidirectional = False)
        self.linear = nn.Linear(hidden_dim, num_unique)

    def forward(self, mod_input):
        word_embed = self.embed(mod_input)
        gru_out, hidden_out = self.gru(word_embed)
        gru_out = self.linear(gru_out)
        gru_out = gru_out.view(-1, num_unique)

        return gru_out, hidden_out

elman_rnn = ElmanRNN(in_size, hidden_dim, n_layers,
 ↪  num_unique, embed_dim, hidden_dim, num_unique, 0.5)
```

```
lstm_rnn = LSTMRNN(in_size, hidden_dim, n_layers, num_unique,
    embed_dim, hidden_dim, num_unique, 0.8)
gru_rnn = GRURNN(in_size, hidden_dim, n_layers, num_unique,
    embed_dim, hidden_dim, num_unique, 0.8)

# Selecting model
model = gru_rnn

reset_model(model)
model.to(device)

# Optimizer
l_rate = 10.0
sgd = optim.SGD(model.parameters(), lr = l_rate, weight_decay
    = 0, momentum = 0.0)
adam = optim.Adam(model.parameters(), lr = l_rate, betas =
    (0.9, 0.999), eps = 1e-08, weight_decay = 0.0, amsgrad =
    False)

optimizer = sgd

# Cross entropy loss
loss = nn.CrossEntropyLoss()

# Training parameters
num_epochs = 50
train_epoch = np.zeros(num_epochs)
valid_epoch = np.zeros(num_epochs)

train_loss = []
valid_loss = []
test_loss = []

# Gradient clipping
clipGrad = True

# Learning rate scheduler
```

```
exp_sched = optim.lr_scheduler.ExponentialLR(optimizer, gamma = 0.5,last_epoch = -1, verbose = True)
cosine_sched = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max = 500, eta_min = 0.1, last_epoch = -1, verbose = True)
#plat_sched = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode = 'min', factor = 0.1, patience = 10, threshold = 0.0001, threshold_mode = 'rel', cooldown = 0, min_lr = 0, eps = 1e-08, verbose = True)

sched = cosine_sched
useScheduler = True

# Training and evaluation on validation data
for epoch in range(num_epochs):
    train_losss = []
    valid_losss = []

    for data, lab_train in training_loader:
        model.train()
        optimizer.zero_grad()

        pred_out, hid_out = model(data)

        ce_loss = loss(pred_out, lab_train.flatten())

        train_loss.append(ce_loss.item())
        train_losss.append(ce_loss.item())

        ce_loss.backward()

        # Using gradient clipping
        if clipGrad:
            nn.utils.clip_grad_norm_(model.parameters(), max_norm = 0.5, norm_type = 2.0)

        optimizer.step()
```

```
        # Using learning rate scheduler
        if useScheduler:
            sched.step()

    train_epoch[epoch] = np.mean(train_losss)

    for data_eval, lab_eval in valid_loader:
        model.eval()
        with torch.no_grad():
            valid_preds, valid_hid = model(data_eval)

            ce_valid = loss(valid_preds, lab_eval.flatten())
            valid_loss.append(ce_valid.item())
            valid_losss.append(ce_valid.item())

    valid_epoch[epoch] = np.mean(valid_losss)

    print("Epoch: %s" % (epoch + 1))

# Evaluation on test data
model.eval()
for data_test, lab_test in test_loader:
    with torch.no_grad():
            test_preds, test_hid = model(data_test)

            ce_test = loss(test_preds, lab_test.flatten())
            test_loss.append(ce_test.item())

plt.figure(1, figsize = (6.4, 4.8))
train, = plt.plot(train_epoch, 'r')
valid, = plt.plot(valid_epoch, 'b')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Averaged training and validation loss")
plt.legend([train, valid], ['Train loss', 'Validation loss'])
```

```
plt.annotate("Final average training loss: %s" % (train_epoch[-1]) ,xycoords = 'figure fraction', xy = (0.25,0.55))
plt.annotate("Final average validation loss: %s" % (valid_epoch[-1]), xycoords = 'figure fraction', xy = (0.25,0.50))
plt.savefig("train_valid_epoch", dpi = 500)
print("Final average training loss: %s." % train_epoch[-1])
print("Final average validation loss: %s." % valid_epoch[-1])
print()

plt.figure(2, figsize = (6.4, 4.8))
train_it, = plt.plot(train_loss, 'r')
valid_it, = plt.plot(valid_loss, 'b')
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Training and validation loss")
plt.legend([train_it, valid_it], ['Train loss', 'Validation loss'])
plt.annotate("Final training loss: %s" % (train_loss[-1]) ,xycoords = 'figure fraction', xy = (0.25,0.55))
plt.annotate("Final validation loss: %s" % (valid_loss[-1]), xycoords = 'figure fraction', xy = (0.25,0.50))
#plt.savefig("train_valid_loss_iter", dpi = 500)
print("Final training loss: %s." % train_loss[-1])
print("Final validation loss: %s." % valid_loss[-1])
print()

plt.figure(3, figsize = (6.4, 4.8))
train_fin_ep, = plt.plot(train_losss, 'r')
valid_fin_ep, = plt.plot(valid_losss, 'b')
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Training and validation loss final epoch")
plt.legend([train_fin_ep, valid_fin_ep], ['Train loss', 'Validation loss'])
```

```
plt.annotate("Training loss final epoch: %s" % (train_losss[-1]) ,xycoords = 'figure fraction', xy = (0.25,0.55))
plt.annotate("Validation loss final epoch: %s" % (valid_losss[-1]), xycoords = 'figure fraction', xy = (0.25,0.50))
#plt.savefig("train_valid_final_epoch", dpi = 500)

perplexity = np.exp(np.mean(test_loss))
print("Test perplexity: %s." % (perplexity))

# Generate sample phrases
dictionary = unique_dict # number of unique words/tokens is 10000
dict_keys = list(dictionary.keys())
dict_values = list(dictionary.values())

# Words
w_1 = "there"
i_1 = dictionary[w_1]
w_2 = "is"
i_2 = dictionary[w_2]
w_3 = "a"
i_3 = dictionary[w_3]
w_4 = "why"
i_4 = dictionary[w_4]
w_5 = "on"
i_5 = dictionary[w_5]

# Input sequence
input_tens = torch.randint(num_unique, (2, 1)).long().to(device)
input_tens[0] = i_1
print("Input", input_tens)

inputs_list = input_tens.tolist()
print("List input", inputs_list)

```

```
softmax_layer = nn.Softmax(dim = 1)
num_words = 5 # how many words to predict
model.eval()
with torch.no_grad():
    for n in range(num_words):
        out, hidden = model(input_tens)
        probs = softmax_layer(out)

        maxv, maxi = torch.max(probs, dim = 1)
        sortedv, sortedi = -np.sort(-maxv.cpu().numpy()), -np.sort(-maxi.cpu().numpy())

        next_word = sortedi[0]

        inputs_list.append([next_word])

        input_tens = input_tens.tolist()
        input_tens.append([next_word])
        input_tens = torch.tensor(input_tens).long().to(device)


print(inputs_list)
output_sentence = []

for i in inputs_list:
    word_i = dict_keys[dict_values.index(i[0])]
    output_sentence.append(word_i)

print(output_sentence)
```