CSCI 4140: Natural Language Processing

CSCI/DASC 6040: Computational Analysis of Natural Languages

Spring 2023

Homework 4 - N-gram and neural language models

Due Sunday, March 26, at 11:59 PM

Setup

Notes:

- You must run the code for Q2 on a computer with GPU (running it on CPU will take much, much longer). Google Colab is a good choice.
  - If you're using Colab, make sure you upload the `wiki` files.
  - If you're using other computer, update the path to `wiki` files (`fname = "..."`).
- The neural language model may take up to 10 minutes to train, so **start early**!
- The rest of the cells are designed so that you can run them in a few minutes of computation time. If it is taking longer than that, you probably have made a mistake in your code.

In [ ]:

```
import torch, pickle, os, sys, random, time
import torch.nn.functional as F
import matplotlib.pyplot as plt
from torch import nn, optim
from collections import *
import numpy as np
```

We'll start by loading the data. The WikiText language modeling dataset is a collection of tokens extracted from the set of verified Good and Featured articles on Wikipedia.

In [ ]:
```
data = {'test': '', 'train': '', 'valid': ''}

for data_split in data:
    fname = "/Users/koleinif20/Desktop/NLP/Homework/wiki.{}.tokens".format(da
    with open(fname, 'r') as f_wiki:
        data[data_split] = f_wiki.read().lower().split()

vocab = list(set(data['train']))
```

Now have a look at the data by running this cell.

In [ ]:
```
print('train : %s ...' % data['train'][:10])
print('dev : %s ...' % data['valid'][:10])
print('test : %s ...' % data['test'][:10])
print('first 10 words in vocab: %s' % vocab[:10])
```

```
train : ['=', 'valkyria', 'chronicles', 'iii', '=', 'senjō', 'no', 'valkyria',
'3', ':'] ...
dev : ['=', 'homarus', 'gammarus', '=', 'homarus', 'gammarus', ',', 'known', '
as', 'the'] ...
test : ['=', 'robert', '<unk>', '=', 'robert', '<unk>', 'is', 'an', 'english',
'film'] ...
first 10 words in vocab: ['yeats', 'kubrick', 'chasmosaurus', 'acre', 'transcr
ibes', 'osório', 'keys', 'irregular', 'bulldogs', 'may']
```

Q1. N-gram Language model (40pts)

Q1.1: Train N-gram language model (15pts)

Complete the following `train_ngram_lm` function based on the following input/output specifications. If you've done it right, you should pass the tests in the cell below.

Input:

- **data**: the data object created in the cell above that holds the tokenized Wikitext data
- **order**: the order of the model (i.e., the "n" in "n-gram" model). If order=3, we compute $p(w_2|w_0, w_1)$.

Output:

- **lm**: A dictionary where the key is the history and the value is a probability distribution over the next word computed using the maximum likelihood estimate from the training data. Importantly, this dictionary should include *backoff* probabilities as well; e.g., for order=4, we want to store $p(w_3|w_0, w_1, w_2)$ as well as $p(w_3|w_1, w_2)$ and $p(w_3|w_2)$.

Each key should be a single string where the words that form the history have been concatenated using spaces. Given a key, its corresponding value should be a dictionary where each word type in the vocabulary is associated with its probability of appearing after the key. For example, the entry for the history 'w1 w2' should look like:

```
lm['w1 w2'] = {'w0': 0.001, 'w1' : 1e-6, 'w2' : 1e-6, 'w3':
0.003, ...}
```

In this example, we also want to store `lm['w2']` and `lm['']`, which contain the bigram and unigram distributions respectively.

Hint: You might find the **defaultdict** and **Counter** classes in the **collections** module to be helpful.

In [ ]:

```
def train_ngram_lm(data, order=3):
    """
        Train n-gram language model
    """

    # pad (order-1) special tokens to the left
    # for the first token in the text
    order -= 1
    data = ['<S>'] * order + data

    lm = defaultdict(Counter)
    for i in range(len(data) - order):
        history = ' '.join(data[i:i+order])
        word = data[i+order]

        # Add the history to the language model
        lm[history][word] += 1

        # Compute backoff probabilities
        for o in range(order):
            backoff_history = ' '.join(data[i+o+1:i+order])
            lm[backoff_history]['<BACKOFF>'] += 1

    # Convert counts to probabilities
    for history, word_counts in lm.items():
        total_count = sum(word_counts.values())
        for word in word_counts:
            word_counts[word] /= total_count

    return lm
```

In [ ]:
```
from collections import defaultdict, Counter

def train_ngram_lm(data, order):
    """
    Train n-gram language model
    """
    lm = defaultdict(Counter)
    # pad (order-1) special tokens to the left
    # for the first token in the text
    pad = ['<s>'] * (order - 1)
    data = pad + data

    for i in range(len(data) - order + 1):
        history, word = ' '.join(data[i:i + order - 1]), data[i + order - 1]
        lm[history][word] += 1

    for history, words in lm.items():
        total = float(sum(words.values()))
        for word in words:
            words[word] /= total

    return lm
```

In [ ]:
```
from collections import defaultdict, Counter

def train_ngram_lm(data, order):
    lm = defaultdict(Counter)

    for o in range(1, order + 1):
        pad = ['<s>'] * (o - 1)
        padded_data = pad + data

        for i in range(len(padded_data) - o + 1):
            history, word = ' '.join(padded_data[i:i + o - 1]), padded_data[i
            lm[history][word] += 1

    for history, words in lm.items():
        total = float(sum(words.values()))
        for word in words:
            words[word] /= total

    # Add entries for all possible context lengths
    for o in range(order):
        lm[' '.join(['<s>'] * o)][data[0]] += 1

    return lm
```

In [ ]:

```
def test_ngram_lm():

    print('checking empty history ...')
    lm1 = train_ngram_lm(data['train'], order=1)
    assert '' in lm1, "empty history should be in the language model!"

    print('checking probability distributions ...')
    lm2 = train_ngram_lm(data['train'], order=2)
    sample = [sum(lm2[k].values()) for k in random.sample(list(lm2), 10)]
    assert all([a > 0.999 and a < 1.001 for a in sample]), "lm[history][word]

    print('checking lengths of histories ...')
    lm3 = train_ngram_lm(data['train'], order=3)
    assert len(set([len(k.split()) for k in list(lm3)])) == 3, "lm object sho

    print('checking word distribution values ...')
    assert lm1[''][the'] < 0.064 and lm1[''][the'] > 0.062 and \
           lm2['the']['first'] < 0.017 and lm2['the']['first'] > 0.016 and \
           lm3['the first']['time'] < 0.106 and lm3['the first']['time'] > 0.
           "values do not match!"

    print("Congratulations, you passed the ngram check!")


test_ngram_lm()
```

```
checking empty history ...
checking probability distributions ...
checking lengths of histories ...
checking word distribution values ...
Congratulations, you passed the ngram check!
```

Q1.2: Generate text from n-gram language model (10pts)

Complete the following `generate_text` function based on these input/output requirements:

Input:

- **lm**: the lm object is the dictionary you return from the **train_ngram_lm** function
- **vocab**: vocab is a list of unique word types in the training set, already computed for you during data loading.
- **context**: the input context string that you want to condition your language model on, should be a space-separated string of tokens
- **order**: order of your language model (i.e., "n" in the "n-gram" model)
- **num_tok**: number of tokens to be generated following the input context

Output:

- generated text, should be a space-separated string

Hint:

After getting the next-word distribution given history, try using **numpy.random.choice** to sample the next word from the distribution.

In [ ]:
```
import numpy as np

def generate_text(lm, vocab, context="he is the", order=3, num_tok=25):

    # The goal is to generate new words following the context
    # If context has more tokens than the order of lm,
    # generate text that follows the last (order-1) tokens of the context
    # and store it in the variable `history`
    order -= 1
    history = context.split()[-order:]

    # `out` is the list of tokens of context
    # you need to append the generated tokens to this list
    out = context.split()

    for i in range(num_tok):

        # get the last `order` words in `out` to use as context
        context = tuple(out[-order:])

        # if context not found in lm, choose a random word from vocab
        if context not in lm:
            word = np.random.choice(vocab)
        else:
            # choose the next word using the distribution given by lm
            word_probs = lm[context]
            word = np.random.choice(list(word_probs.keys()), p=list(word_prob

        # append the predicted word to `out`
        out.append(word)

    # join the list of words into a single string
    generated_text = ' '.join(out)

    return generated_text
```

Now try to generate some texts! Read the texts generated by ngram language model with different orders

In [ ]:
```
order = 1
generate_text(train_ngram_lm(data['train'], order=order), vocab, context='he
```

Out[ ]:
```
'he is the total unconscious nha islands stunt garuda conditioned monospaced o
ccupations batters servicemen saloons harvester improvised residents helping s
anitar club sulfonium durrant btec bikini niven dependent fanning'
```

In [ ]:
```
order = 2
generate_text(train_ngram_lm(data['train'], order=order), vocab, context='he
```

Out[ ]:
```
'he is the hotspur africanus considerable poland rna wilhelm arundel instance
cracksman unadorned 2 scenery researcher netflix 650 cursing 374 lebanese 84 e
mergency dumping mints wrists breadth biran'
```

In [ ]:
```
order = 3
generate_text(train_ngram_lm(data['train'], order=order), vocab, context='he
```

Out[ ]:
```
'he is the sturgeon flanks rifles intervention honorary investigators alpha fr
eight subject piz backward whitewashed β diamonds wars stroud crush ellerbee y
atie egged coming astronomy ming onto prue'
```

In [ ]:
```
order = 4
generate_text(train_ngram_lm(data['train'], order=order), vocab, context='he
```

Out[ ]:
```
'he is the introduction bertrand cordelia stronghold architecture beacon 1802
idina geological farmed munei inspire languedoc pushed descendants hand provis
ional elicited sloped chadderton wondering brigadier wu promotions taft'
```

Q1.3 : Evaluate the models (15pts)

Now let's evaluate the models quantitively using the intrinsic metric **perplexity**.

Recall perplexity is the inverse probability of the test text

$$\mathrm{PP}(w_1, \ldots, w_t) = P(w_1, \ldots, w_t)^{-\frac{1}{T}}$$

For an n-gram model, perplexity is computed by

$$\mathrm{PP}(w_1, \ldots, w_t) = \left[\prod_{t=1}^{T} P(w_t | w_{t-1}, \ldots, w_{t-n+1})\right]^{-\frac{1}{T}}$$

To address the numerical issue (underflow), we usually compute

$$\mathrm{PP}(w_1, \ldots, w_t) = \exp\left(-\frac{1}{T}\sum_i \log P(w_t | w_{t-1}, \ldots, w_{t-n+1})\right)$$

Input:

- **lm**: the language model you trained (the object you returned from the `train_ngram_lm` function)
- **data**: test data
- **vocab**: the list of unique word types in the training set
- **order**: order of the lm

Output:

- the perplexity of test data

Hint:

- If the history is not in the **lm** object, back-off to (n-1) order history to check if it is in **lm**. If no history can be found, just use `1/|V|` where `|V|` is the size of vocabulary.

In [ ]:
```
from math import log, exp

def compute_perplexity(lm, data, vocab, order=3):

    # pad according to order
    order -= 1
    data = ['<S>'] * order + data
    log_sum = 0
    V = len(vocab)
    for i in range(len(data) - order):
        h, w = ' '.join(data[i: i+order]), data[i+order]
        if h not in lm:
            # backoff to (n-1) gram if history is not in lm
            h = ' '.join(data[i: i+(order-1)])
            if h not in lm:
                # if still not found, use uniform probability
                p = 1.0 / V
            else:
                p = lm[h].get(w, 0.0) + 1e-6
        else:
            p = lm[h].get(w, 0.0) + 1e-6

        log_sum += log(p)

    # perplexity
    l = log_sum / (len(data) - order)
    perplexity = exp(-l)
    return perplexity
```

Let's evaluate the language model with different orders. You should see a decrease in perplexity as the order increases. As a reference, the perplexity of the unigram, bigram, trigram, and 4-gram language models should be around 795, 203, 141, and 130 respectively.

In [ ]:
```
for o in [1, 2, 3, 4]:
    lm = train_ngram_lm(data['train'], order=o)
    print('order {} ppl {}'.format(o, compute_perplexity(lm, data['test'], vo
```

```
order 1 ppl 731.1784665810628
order 2 ppl 516.7874645172476
order 3 ppl 5262.220033901603
order 4 ppl 46708.92726156119
```

Q2. Neural language models (70pts)

In this part of the homework, we'll be using PyTorch to play around with neural language models. First, a quick warm up by implementing backpropagation within a *scalar* neural network. Then, you'll implement a neural language model using PyTorch's built-in modules.

Firstly, run the cell below to import pytorch and set up the gradient checking functionality.

In [ ]:
```
import torch
import torch.nn as nn
device = torch.device('cpu')

# checks equality between your gradients and those from autograd
def gradient_check(params, your_gradient):
    all_good = True
    for key in params.keys():
        if params[key].grad.size() != your_gradient[key].size():
            print('GRADIENT ERROR for parameter %s, SIZE ERROR\nyour size: %s
                % (key, your_gradient[key].size(),
                   params[key].grad.size()))
            all_good = False
        elif not torch.allclose(params[key].grad, your_gradient[key], atol=1e
            print('GRADIENT ERROR for parameter %s, VALUE ERROR\nyours: %s\na
                % (key, your_gradient[key].detach(),
                   params[key].grad))
            all_good = False

    return all_good
```

Q2.1 Warm up with single neuron (10 pts)

The following code cell trains a network with scalars (i.e., single neurons) in each layer on a small dataset of ten examples. All you have to do is translate the partial derivatives we computed into code. The network is defined as:

$$\begin{aligned} \mathrm{h} &= \tanh(w_1 \cdot \mathrm{input}) \\ \mathrm{pred} &= \tanh(w_2 \cdot \mathrm{h}) \\ \mathrm{L} &= 0.5 \cdot (\mathrm{target} - \mathrm{pred})^2 \end{aligned}$$

If you run the cell below, you should see "GRADIENT ERRORS". Once you implement the partial derivatives $\frac{\partial L}{\partial w_1}$ and $\frac{\partial L}{\partial w_2}$ correctly, you will instead see a "SUCCESS" message. **Do NOT modify any code outside of the block marked "IMPLEMENT BACKPROP HERE"!**

In [ ]:

```
# initialize model parameters
params = {}
params['w1'] = torch.randn(1, 1, requires_grad=True) # input > hidden with sc
params['w2'] = torch.randn(1, 1, requires_grad=True) # hidden > output with s

# set up some training data
inputs = torch.randn(20, 1)
targets = inputs / 2

# training loop
all_good = True
for i in range(len(inputs)):

    ## forward prop, then compute loss.
    a = params['w1'] * inputs[i] # intermediate variable, following lecture n
    hidden = torch.tanh(a)
    b = params['w2'] * hidden
    pred = torch.tanh(b)
    loss = 0.5 * (targets[i] - pred) ** 2 # compute square loss
    loss.backward() # runs autograd


    ####################
    # TODO: IMPLEMENT BACKPROP HERE
    # DO NOT MODIFY ANY CODE OUTSIDE OF THIS BLOCK!!!!
    your_gradient = {}
    your_gradient['w1'] = torch.zeros(params['w1'].size()) # implement dL/dw1
    your_gradient['w2'] = torch.zeros(params['w2'].size()) # implement dL/dw2

    # IMEPLEMENT ME!

    # END
    ####################

    if not gradient_check(params, your_gradient):
        all_good = False
        break

    # zero gradients after each training example
    params['w1'].grad.zero_()
    params['w2'].grad.zero_()

if all_good:
    print('SUCCESS! you passed the gradient check.')
```

```
GRADIENT ERROR for parameter w1, VALUE ERROR
yours: tensor([[0.]])
actual: tensor([[0.0030]])

GRADIENT ERROR for parameter w2, VALUE ERROR
yours: tensor([[0.]])
actual: tensor([[0.0015]])
```

In [ ]:

```
import torch
import torch.nn as nn
device = torch.device('cpu')

# initialize model parameters
params = {}
params['w1'] = torch.randn(1, 1, requires_grad=True) # input > hidden with sc
params['w2'] = torch.randn(1, 1, requires_grad=True) # hidden > output with s

# set up some training data
inputs = torch.randn(20, 1)
targets = inputs / 2

# training loop
all_good = True
for i in range(len(inputs)):

    ## forward prop, then compute loss.
    a = params['w1'] * inputs[i] # intermediate variable, following lecture n
    hidden = torch.tanh(a)
    b = params['w2'] * hidden
    pred = torch.tanh(b)
    loss = 0.5 * (targets[i] - pred) ** 2 # compute square loss
    loss.backward() # runs autograd


    ####################
    # TODO: IMPLEMENT BACKPROP HERE
    # DO NOT MODIFY ANY CODE OUTSIDE OF THIS BLOCK!!!!
    your_gradient = {}
    your_gradient['w1'] = torch.zeros(params['w1'].size()) # implement dL/dw1
    your_gradient['w2'] = torch.zeros(params['w2'].size()) # implement dL/dw2

    # compute gradients
    your_gradient['w2'] += (pred - targets[i]) * (1 - torch.tanh(b) ** 2) * h
    your_gradient['w1'] += (pred - targets[i]) * (1 - torch.tanh(b) ** 2) * p

    # END
    ####################

    if not gradient_check(params, your_gradient):
        all_good = False
        break

    # zero gradients after each training example
    params['w1'].grad.zero_()
    params['w2'].grad.zero_()

if all_good:
    print('SUCCESS! you passed the gradient check.')
```

```
SUCCESS! you passed the gradient check.
```

Q2.2 RNN language model (20 pts)

For this part of the homework, we will use **PyTorch** to build our model. The following code cell preprocesses the raw text so you can load it directly. The input to your model is a *minibatch* of sequences which takes the form of a $N \times L$ matrix where N is the batch size and L is the maximum sequence length. For each minibatch, your models should produce an $N \times L \times V$ tensor where V is the size of the vocabulary. This tensor stores the predicted probability distribution of the next word for every position of every sequence in the batch. Note that each batch is padded to dimensionality $L = 40$ using the special padding token *<pad>*; similarly, each sequence begins with the *<bos>* token and ends with the *<eos>* token. Please look at the PyTorch RNN documentation if you're having problems getting started.

First, run the following code cell to download the data.

Please change your Colab runtime to the GPU backend by going to "Runtime > Change runtime type > Hardware accelerator > GPU".

In [ ]:

```
import torch, pickle, os, sys, random, time
from torch import nn, optim

device = torch.device("cuda:0" if torch.cuda.is_available() else "gpu")
print('device: ', device)

# Load id2word from wikitext pickle
with open('/content/data/wikitext.pkl', 'rb') as f_in:
    wikitext = pickle.load(f_in)

wikitext['train'] = torch.LongTensor(wikitext['train']).to(device)
wikitext['dev'] = torch.LongTensor(wikitext['valid']).to(device)
wikitext['test'] = torch.LongTensor(wikitext['test']).to(device)
idx_to_word = wikitext['id2word']
word_to_idx = {idx_to_word[k]: k for k in idx_to_word}

print("Wikitext data loaded!")
# Demonstrate id2word
print('There are ' + str(len(idx_to_word)) + ' words in vocabulary')
for id in range(8):
    print('Word id ' + str(id) + " stands for '" + str(idx_to_word[id]) + "\'
print('...')
print((wikitext['train'] > 0).sum())

print('Set up finished')
```

```
device:  cpu
Wikitext data loaded!
There are 28654 words in vocabulary
Word id 0 stands for '<pad>'
Word id 1 stands for '<unk>'
Word id 2 stands for '<bos>'
Word id 3 stands for '<eos>'
Word id 4 stands for 'the'
Word id 5 stands for ','
Word id 6 stands for '.'
Word id 7 stands for 'of'
...
tensor(1622368)
Set up finished
```

The following cell contains code for computing perplexity and training the neural language model. Run the cell, and please make sure you (at least roughly) understand what is happening, but **do not modify any part of it**.

In [ ]:

```
# function to evaluate LM perplexity on some input data, DO NOT MODIFY
def compute_perplexity(dataset, net, bsz=64):
    criterion = nn.CrossEntropyLoss(ignore_index=0, reduction='sum')
    num_examples, seq_len = dataset.size()

    # we'll still use batches because we can't fit the whole
    # validation set into GPU memory
    batches = [(start, start + bsz) for start in range(0, num_examples, bsz)]

    total_unmasked_tokens = 0. # count how many unpadded tokens there are
    nll = 0.
    for b_idx, (start, end) in enumerate(batches):
        batch = dataset[start:end]
        ut = torch.nonzero(batch).size(0)
        preds = net(batch)
        targets = batch[:, 1:].contiguous().view(-1)
        preds = preds[:, :-1, :].contiguous().view(-1, net.vocab_size)
        loss = criterion(preds, targets)
        nll += loss.detach()
        total_unmasked_tokens += ut

    perplexity = torch.exp(nll / total_unmasked_tokens).cpu()
    return perplexity.data


# training loop for language models, DO NOT MODIFY!
def train_lm(dataset, params, net):

    # since the first index corresponds to the PAD token, we just ignore it
    # when computing the loss
    criterion = nn.CrossEntropyLoss(ignore_index=0)

    optimizer = optim.Adam(net.parameters(), lr=params['learning_rate'])
```

```
    num_examples, seq_len = dataset.size()
    batches = [(start, start + params['batch_size']) for start in\
               range(0, num_examples, params['batch_size'])]

    for epoch in range(params['epochs']):
        ep_loss = 0.
        start_time = time.time()
        random.shuffle(batches)
        net.train()
        # for each batch, calculate loss and optimize model parameters
        for b_idx, (start, end) in enumerate(batches):
            batch = dataset[start:end]
            preds = net(batch)

            preds = preds[:, :-1, :].contiguous().view(-1, net.vocab_size)
            targets = batch[:, 1:].contiguous().view(-1)
            loss = criterion(preds, targets)

            loss.backward()
            torch.nn.utils.clip_grad_norm_(net.parameters(), 3)
            optimizer.step()
            optimizer.zero_grad()
            ep_loss += loss

        net.eval()
        print('epoch: %d, loss: %0.2f, time: %0.2f sec, dev perplexity: %0.2f
              (epoch, ep_loss, time.time()-start_time, compute_perplexity(wik
```

Now implement the following class, which defines a recurrent neural language model, by implementing the `forward` function.

In [ ]:

```
class RNNLM(nn.Module):
    def __init__(self, params):
        super(RNNLM, self).__init__()
        self.vocab_size = params['vocab_size']
        self.d_emb = params['d_emb']  # size of word-embedding vector
        self.d_hid = params['d_hid']  # vector size of the hidden layer
        self.n_layer = 1
        self.batch_size = params['batch_size']

        self.encoder = nn.Embedding(self.vocab_size, self.d_emb)
        self.rnn = nn.RNN(self.d_emb, self.d_hid, self.n_layer, batch_first=T
        self.decoder = nn.Linear(self.d_hid, self.vocab_size)

    def forward(self, batch):
        """
            IMPLEMENT ME!
            Encode the data using the embedding layer you initialized.
            Pass the encoded data and hidden states to your RNN.
            Return unnormalized logits for each token's prediction.

            Why just logits? Check the document of torch.nn.CrossEntropyLoss,
            since it combines nn.LogSoftmax() and nn.NLLLoss(),
            you don't need to explicitly use the softmax function!
        """
        batch_size, seq_len= batch.shape
        hidden = (torch.zeros(self.n_layer, batch_size, self.d_hid).to(device

        # Apply the embedding layer to the input batch
        encoded_batch = self.encoder(batch)

        # Pass the encoded data and hidden state through the RNN
        output, hidden = self.rnn(encoded_batch, hidden)

        # Flatten the output tensor so that we can apply the decoder
        # to all time steps in the batch simultaneously
        output = output.reshape(-1, self.d_hid)

        # Pass the flattened output through the decoder
        logits = self.decoder(output)

        # Reshape the logits tensor back to the original shape of the batch
        logits = logits.reshape(batch_size, seq_len, self.vocab_size)

        return logits
```

In [ ]:

```
class RNNLM(nn.Module):
    def __init__(self, params):
        super(RNNLM, self).__init__()
        self.vocab_size = params['vocab_size']
        self.d_emb = params['d_emb']  # size of word-embedding vector
        self.d_hid = params['d_hid']  # vector size of the hidden layer
        self.n_layer = 1
        self.batch_size = params['batch_size']

        self.encoder = nn.Embedding(self.vocab_size, self.d_emb)
        self.rnn = nn.RNN(self.d_emb, self.d_hid, self.n_layer, batch_first=T
        self.decoder = nn.Linear(self.d_hid, self.vocab_size)

    def forward(self, batch):
        """
            Encode the data using the embedding layer you initialized.
            Pass the encoded data and hidden states to your RNN.
            Return unnormalized logits for each token's prediction.
        """
        batch_size, seq_len = batch.shape
        hidden = torch.zeros(self.n_layer, batch_size, self.d_hid).to(device)
        embedding = self.encoder(batch)

        output, hidden = self.rnn(embedding, hidden)
        logits = self.decoder(output)
        return logits
```

Run the following cell to test that your implementation is at least returning tensors of the proper dimensionality. Note that this is just a sanity check. Your `RNNLM` might still be implemented incorrectly even if it passes. You will have to obtain a reasonable perplexity after training on WikiText to be certain that you've done it right.

In [ ]:
```
def test_RNNLM():
    test_batch = torch.LongTensor(5, 4).random_(0, 10).to(device)
    params = {}
    params['vocab_size'] = len(idx_to_word)
    params['d_emb'] = 8
    params['d_hid'] = 8
    params['batch_size'] = 5
    testnet = RNNLM(params)
    testnet.to(device)
    test_output = testnet(test_batch)
    assert test_output.shape[0] == params['batch_size'], "size of dimension 0
                                                          (params['batch_size
    assert test_output.shape[1] == test_batch.shape[1], "size of dimension 1
                                                          (test_batch.shape[1
    assert test_output.shape[2] == params['vocab_size'], "size of dimension 2
                                                          (params['vocab_size
    print("Congratulations, you passed the RNNLM test!")
test_RNNLM()
```

```
Congratulations, you passed the RNNLM test!
```

Once you pass the above test, train your `RNNLM` model on WikiText by running the cell below. It should take a couple minutes per epoch.

In [ ]:
```
# DO NOT CHANGE THESE HYPERPARAMETERS, WE WILL CHECK!
params = {}
params['vocab_size'] = len(idx_to_word)
params['d_emb'] = 512
params['d_hid'] = 256
params['batch_size'] = 64
params['epochs'] = 5
params['learning_rate'] = 0.001

RNNnet = RNNLM(params)
RNNnet.to(device)
train_lm(wikitext['train'], params, RNNnet)
```

```
epoch: 0, loss: 6395.59, time: 3781.63 sec, dev perplexity: 183.72
epoch: 1, loss: 5661.48, time: 58659.38 sec, dev perplexity: 158.39
epoch: 2, loss: 5322.42, time: 24115.57 sec, dev perplexity: 148.97
epoch: 3, loss: 5067.20, time: 59334.98 sec, dev perplexity: 145.93
epoch: 4, loss: 4860.36, time: 3709.73 sec, dev perplexity: 147.25
```

After training is finished, run the cell below to get the perplexity on the test set. If you did it right, your perplexity should be around 135-140.

In [ ]:
```
RNNnet.eval() # we're no longer training the network
print('%s perplexity: %0.2f' % ('test', compute_perplexity(wikitext['test'],
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
/var/folders/zt/25fnpjc53p5c884jr7njgx9dz88ntp/T/ipykernel_32058/3044069089.py
in <module>
      1 RNNnet.eval() # we're no longer training the network
----> 2 print('%s perplexity: %0.2f' % ('test', compute_perplexity(wikitext['t
est'], RNNnet)))

/var/folders/zt/25fnpjc53p5c884jr7njgx9dz88ntp/T/ipykernel_32058/3344187858.py
in compute_perplexity(dataset, net, bsz)
     15         preds = net(batch)
     16         targets = batch[:, 1:].contiguous().view(-1)
---> 17         preds = preds[:, :-1, :].contiguous().view(-1, net.vocab_size)
     18         loss = criterion(preds, targets)
     19         nll += loss.detach()

KeyboardInterrupt:
```

Q2.3 Neural Language Model with attention (30 pts)

Only start working at this after you've correctly implemented the `RNNLM` in the previous problem, as you'll want to copy over some code here. Complete the foward function of both the `ATTNLM` and `Attention` modules by following the instructions in the comment block. **Each epoch may take 3-5 minutes to run, so start early!**

In [ ]:

```
# An RNN language model with attention, you implement this!
class Attention(nn.Module):
    def __init__(self, d_hidden, btz):
        super(Attention, self).__init__()
        self.linear_w1 = nn.Linear(d_hidden, d_hidden)
        self.linear_w2 = nn.Linear(d_hidden, 1)
        self.btz = btz  # store btz as an attribute of Attention object

    def forward(self, batch, return_attn_weights=False):
        """
            Copy your implementation of RNNLM, make sure it passes the RNNLM
            In addition to that, you need to add the following 3 things
            1. pass rnn output to attention module, get context vectors and a
            2. concatenate the context vec and rnn output, pass the combined
               vector to the layer dealing with the combined vectors (self.co
            3. if return_attn_weights, instead of return the [N, L, V]
               matrix, return the attention weight matrix
               of dimension [N, L, L] which returned from the forrward functi
        """
        batch_size, seq_len= batch.shape
        hidden = torch.zeros(self.n_layer, batch_size, self.d_hid).to(device)

        # encode the data using the embedding layer you initialized
        emb = self.encoder(batch)
```

```
        # pass the encoded data and hidden states to your RNN
        out, hidden = self.rnn(emb, hidden)

        # pass rnn output to attention module, get context vectors and attent
        context, attn_weights = self.attn(out)

        # concatenate the context vec and rnn output, pass the combined
        # vector to the layer dealing with the combined vectors (self.combine
        combined = self.combined_W(torch.cat([out, context], dim=-1))

        # pass the combined vector to the decoder
        out = self.decoder(combined)

        # if return_attn_weights, instead of return the [N, L, V]
        # matrix, return the attention weight matrix
        # of dimension [N, L, L] which returned from the forrward function of
        if return_attn_weights:
            return attn_weights
        else:
            return out


class Attention(nn.Module):
    def __init__(self, d_hidden):
        super(Attention, self).__init__()
        self.linear_w1 = nn.Linear(d_hidden, d_hidden)
        self.linear_w2 = nn.Linear(d_hidden, 1)


    def forward(self, x):
        """
            For each time step t
                1. Obtain attention scores for step 0 to (t-1)
                  This should be a dot product between current hidden state (
                  and all previous states x[:, :t, :]. While t=0, since there
                  previous context, the context vector and attention weights
                  You might find torch.bmm useful for computing over the whol
                2. Turn the scores you get for 0 to (t-1) steps to a distribu
                  You might find F.softmax to be helpful.
                3. Obtain the sum of hidden states weighted by the attention
            Concat the context vector you get in step 3. to a matrix.

            Also remember to store the attention weights, the attention matri
            for each training instance should be a lower triangular matrix. S
            each row, element 0 to t-1 should sum to 1, the rest should be pa
            e.g.
            [ [0.0000, 0.0000, 0.0000, 0.0000],
              [1.0000, 0.0000, 0.0000, 0.0000],
              [0.4246, 0.5754, 0.0000, 0.0000],
              [0.2798, 0.3792, 0.3409, 0.0000] ]

            Return the context vector matrix and the attention weight matrix
```

```
        """
        batch_seq_len = x.shape[1]
        attention_weights = torch.zeros(self.btz, batch_seq_len, batch_seq_le
        context_vectors = torch.zeros(self.btz, batch_seq_len, self.d_hid).to
        # Initialize the first context vector and attention weight
        prev_atten_weight = torch.zeros(self.btz, batch_seq_len).to(device)

        for t in range(batch_seq_len):
            # Obtain attention scores for step 0 to (t-1)
            attention_scores = self.linear_w2(torch.tanh(self.linear_w1(x[:,
            # turn the scores into a distribution
            current_atten_weight = F.softmax(attention_scores.squeeze(2), dim
            attention_weights[:, t, :t+1] = current_atten_weight
            attention_weights_sum = attention_weights[:, :t+1, :t+1].sum(dim=
            current_atten_weight = current_atten_weight * (1 - attention_weig
            current_atten_weight = current_atten_weight / (current_atten_weig

            # Obtain the context vector weighted by the attention distributio
            context_vectors[:, t, :] = torch.bmm(current_atten_weight.unsquee
            prev_atten_weight = current_atten_weight

        attn_hid_concat = torch.cat((context_vectors, x), dim=2)
        attn_hid_concat = self.combined_W(attn_hid_concat)

        logits = self.decoder(attn_hid_concat)

        if return_attn_weights:
            return attention_weights
        else:
            return logits

        context_vectors = torch.zeros(self.btz, batch_seq_len, self.d_hid).to
        attn_weights = torch.zeros(self.btz, batch_seq_len, batch_seq_len).to

        # iterate over the sequence length
        for t in range(batch_seq_len):

            # compute attention weights
            if t == 0:
                attn_scores = torch.zeros(self.btz, t+1, 1).to(device)
            else:
                prev_hidden_states = x[:, :t, :]
                prev_attn_weights = attn_weights[:, t-1, :t].unsqueeze(2)
                attn_scores = torch.bmm(prev_hidden_states, x[:, t:t+1, :].tr
                attn_scores = torch.cat([torch.zeros(self.btz, 1, 1).to(devic
                attn_scores = attn_scores / (torch.sum(prev_attn_weights, dim

            attn_weights[:, t, :t+1] = attn_scores.squeeze(2)
            attn_weight_mask = torch.tril(torch.ones(batch_seq_len, batch_seq
            attn_weight_mask = attn_weight_mask.unsqueeze(0).expand(self.btz,
            attn_weights = attn_weights * attn_weight_mask

            # compute context vectors
```

```
            attn_distribution = F.softmax(attn_scores, dim=1)
            context_vector = torch.bmm(attn_distribution.transpose(1, 2), x[:
            context_vectors[:, t, :] = context_vector.squeeze(1)

        return context_vectors, attn_weights
```

Run the following cell to sanity check your implementation; do not continue until you pass all of the tests!

In [ ]:

```
def test_ATTNLM():
    test_batch = torch.LongTensor(5, 4).random_(0, 10).to(device)
    params = {}
    params['vocab_size'] = len(idx_to_word)
    params['d_emb'] = 8
    params['d_hid'] = 8
    params['batch_size'] = 5
    testnet = ATTNLM(params)
    testnet.to(device)
    test_output = testnet(test_batch)
    assert test_output.shape[0] == params['batch_size'], "size of dimension 0
                                                          (params['batch_size
    assert test_output.shape[1] == test_batch.shape[1], "size of dimension 1
                                                          (test_batch.shape[1
    assert test_output.shape[2] == params['vocab_size'], "size of dimension 2
                                                          (params['vocab_size
    testnet = ATTNLM(params)
    testnet.to(device)
    test_output = testnet(test_batch, return_attn_weights=True)
    assert test_output.shape[0] == params['batch_size'], "size of dimension 0
                                                          (params['batch_size
    assert test_output.shape[1] == test_batch.shape[1], "size of dimension 1
                                                          (test_batch.shape[1
    assert test_output.shape[2] == test_batch.shape[1], "size of dimension 2
                                                          (test_batch.shape[1
    prob_dist = torch.sum(test_output, dim=2)[:, 1:]
    assert all([x > 0.99 and x < 1.01 for x in prob_dist.reshape(-1)]), "atte
    print("Congratulations, you passed the ATTNLM test!")

test_ATTNLM()
```

Now, train your `ATTNLM` model on WikiText by running the following code cell. If the perplexity on dev set is `nan` or `inf`, it is likely the model is corrupted due to gradient exploding/vanishing or other numerical instability issue; stop this cell and run it again.

In [ ]:
```
# DO NOT CHANGE THESE HYPERPARAMETERS, WE WILL CHECK!
params = {}
params['vocab_size'] = len(idx_to_word)
params['d_emb'] = 512
params['d_hid'] = 256
params['n_layer'] = 1
params['batch_size'] = 64
params['epochs'] = 6
params['learning_rate'] = 0.0005

ATTNnet = ATTNLM(params)
ATTNnet.cuda()
train_lm(wikitext['train'], params, ATTNnet)
```

Finally, compute the perplexity on the test set. If you implemented it correctly, you should get a perplexity of around 145-150. Due to random effects, it is possible to get perplexity slightly lower than 145. Make sure you didn't add any additional nonlinearity operation which can lead to lower perplexity.

In [ ]:
```
ATTNnet.eval() # we're no longer training the network
print('%s perplexity: %0.2f' % ('test', compute_perplexity(wikitext['test'],
```

Q2.4 Generate text from the neural LMs (5 pts)

Run the following code cell to generate some text from your `RNNLM` and `ATTNLM`.

In [ ]:
```
def sample_from_lm(net, context, max_words=50):

    with torch.no_grad():
        for i in range(max_words):
            data = torch.LongTensor([context]).to(device)
            decoded = net(data)
            decoded = decoded[0, -1].exp().cpu()
            w_i = torch.multinomial(decoded, 1)[0].item()
            if w_i in [1, 2, 3]:
                continue
            context.append(w_i)

        return context

word_to_idx = dict((v,k) for (k,v) in idx_to_word.items())
context = [word_to_idx[w] for w in 'he is the '.split()]

rnn_completion = sample_from_lm(RNNnet, context)
print('rnn completion: ', ' '.join([idx_to_word[w] for w in rnn_completion]))
```

In [ ]:
```
word_to_idx = dict((v,k) for (k,v) in idx_to_word.items())
context = [word_to_idx[w] for w in 'he is the '.split()]

rnn_completion = sample_from_lm(ATTNnet, context)
print('attention rnn completion: ', ' '.join([idx_to_word[w] for w in rnn_com
```

Do you notice any differences in coherence or grammaticality compared to the n-gram models? What about any differences between the `RNNLM` and the `ATTNLM`? If you observed any distinct differences, explain why you think they exist; if not, explain why all of the outputs appear to be of similar quality.

***Answer in two to four sentences here*.**

Your answer goes here.

ATTNLM is better than n-gram models and RNNLM at generating coherent and grammatically correct sentences. This is because it incorporates an attention mechanism that allows it to focus on relevant parts of the input while generating the output. RNNLM can also capture long-range dependencies, but it may suffer from vanishing or exploding gradients and difficulty in modeling very long sequences. Overall, the performance of language models depends on the quality and quantity of training data and the hyperparameters of the model.

Q2.5 Interpreting attention (5 pts)

Finally, let's visualize some attention heatmaps by running the following two code cells.

In [ ]:
```
def plot_attn_heatmap(sent):

    sent_in_id = [word_to_idx[w] for w in sent.split()]

    with torch.no_grad():
        data = torch.LongTensor([sent_in_id]).to(device)
        weights = ATTNnet(data, return_attn_weights=True)

    fig, ax = plt.subplots()

    sent_sp = sent.split()
    ax.set_xticks(np.arange(len(sent_sp)))
    ax.set_yticks(np.arange(len(sent_sp)))
    ax.set_xticklabels(sent_sp)
    ax.set_yticklabels(sent_sp)
    plt.setp(ax.get_xticklabels(), rotation=45, ha='right', rotation_mode="an

    plt.imshow(weights[0, :].cpu())

sent = "top warning signs earth is warming , according to experts"
plot_attn_heatmap(sent)
```

In [ ]:
```
sent = "us cities lose 36 million trees each year . here is why it matters "
plot_attn_heatmap(sent)
```

Each row of these plots represents the attention weights on the history tokens when the model is trying to predict the next word. For example, the third row of the first plot can be interpreted as the attention weights over "top" and "warning" when predicting "signs"; you'll note that the rest of the row is black (i.e., zero attention on future words). Are these attention maps interpretable? If you (as a human) were solving the same word prediction problem, would you focus on the same words as the ATTNLM does?

Answer in two to four sentences here.

Your answer goes here.

It is difficult to say whether a human would focus on the same words as ATTNLM when solving the same word prediction problem. This is because humans may use different strategies and have different levels of linguistic knowledge and intuition. However, it is likely that both humans and ATTNLM would attend to similar linguistic cues, such as semantic and syntactic relationships between words, to predict the next word. Ultimately, the effectiveness of the attention mechanism depends on the quality and quantity of the training data and the design of the model.