Assignment4

January 17, 2022

1 Assignment 4: Language Processing with RNN-Based Autoencoders

Deadline: Sunday, June 15th, by 9pm.

Submission: Submit a PDF export of the completed notebook as well as the ipynb file.

In this assignement, we will practice the application of deep learning to natural language processing. We will be working with a subset of Reuters news headlines that are collected over 15 months, covering all of 2019, plus a few months in 2018 and in a few months of this year.

In particular, we will be building an **autoencoder** of news headlines. The idea is similar to the kind of image autoencoder we built in lecture: we will have an **encoder** that maps a news headline to a vector embedding, and then a **decoder** that reconstructs the news headline. Both our encoder and decoder networks will be Recurrent Neural Networks, so that you have a chance to practice building

- a neural network that takes a sequence as an input
- a neural network that generates a sequence as an output

This assignment is organized as follows:

- Question 1. Exploring the data
- Question 2. Building the autoencoder
- Question 3. Training the autoencoder using *data augmentation*
- Question 4. Analyzing the embeddings (interpolating between headlines)

Furthermore, we'll be introducing the idea of **data augmentation** for improving of the robustness of the autoencoder, as proposed by Shen et al [1] in ICML 2020.

[1] Shen, Tianxiao, Jonas Mueller, Regina Barzilay, and Tommi Jaakkola. "Educating text autoencoders: Latent representation guidance via denoising." In International Conference on Machine Learning, pp. 8719-8729. PMLR, 2020.

[1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

import matplotlib.pyplot as plt
import numpy as np
import random
```

1.1 Question 1. Data (20 %)

Download the files `reuters_train.txt` and `reuters_valid.txt`, and upload them to Google Drive.

Then, mount Google Drive from your Google Colab notebook:

[2]:
```python
from google.colab import drive
drive.mount('/content/gdrive')

train_path = '/content/gdrive/My Drive/reuters_train.txt' # Update me
valid_path = '/content/gdrive/My Drive/reuters_valid.txt' # Update me
```

```
Mounted at /content/gdrive
```

As we did in some of our examples (e.g., training transformers on IMDB reviews) will be using PyTorch's `torchtext` utilities to help us load, process, and batch the data. We'll be using a `TabularDataset` to load our data, which works well on structured CSV data with fixed columns (e.g. a column for the sequence, a column for the label). Our tabular dataset is even simpler: we have no labels, just some text. So, we are treating our data as a table with one field representing our sequence.

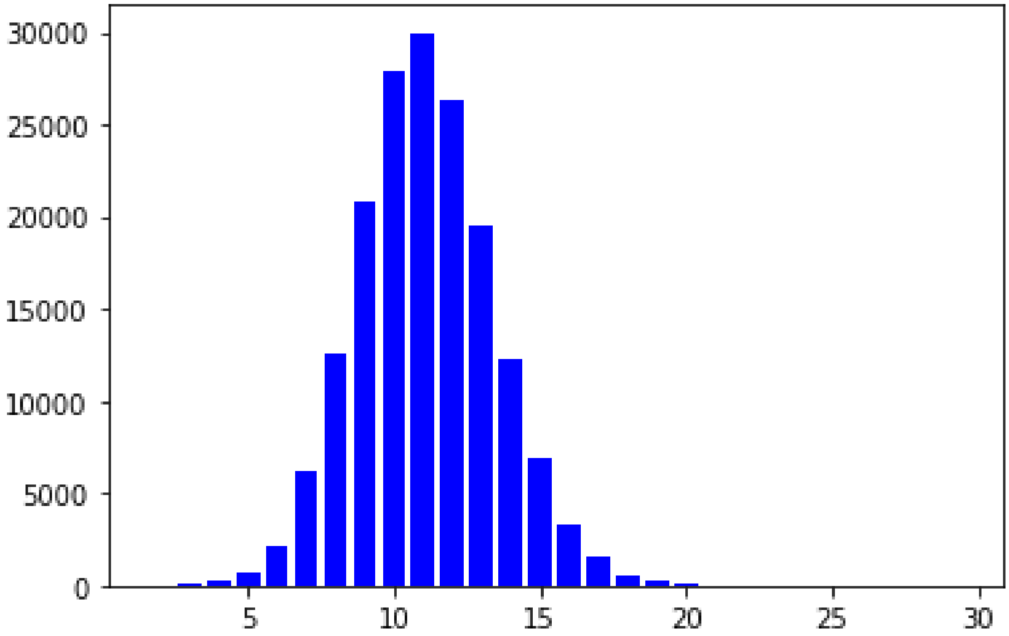
[3]:
```python
import torchtext.legacy.data as data

# Tokenization function to separate a headline into words
def tokenize_headline(headline):
    """Returns the sequence of words in the string headline. We also
    prepend the "<bos>" or beginning-of-string token, and append the
    "<eos>" or end-of-string token to the headline.
    """
    return ("<bos> " + headline + " <eos>").split()

# Data field (column) representing our *text*.
text_field = data.Field(
    sequential=True,             # this field consists of a sequence
    tokenize=tokenize_headline, # how to split sequences into words
    include_lengths=True,        # to track the length of sequences, for batching
    batch_first=True,            # similar to batch_first=True used in nn.RNN demonstrated in lecture
    use_vocab=True)              # to turn each character into an integer index
train_data = data.TabularDataset(
    path=train_path,                 # data file path
    format="tsv",                    # fields are separated by a tab
    fields=[('title', text_field)]) # list of fields (we have only one)
```

1.1.1 Part (a) -- 5%

Draw histograms of the number of words per headline in our training set. Excluding the `<bos>` and `<eos>` tags in your computation. Explain why we would be interested in such histograms.

[4]:

```
# Include your histogram and your written explanations
# Here are some sample code that uses the train_data object:

import collections
dict_counts = {}
for example in train_data:
    key = int(len(example.title) - 2)
    if key in dict_counts.keys():
        dict_counts[key] += 1
    else:
        dict_counts[key] = 1

dict_counts_sorted = collections.OrderedDict(sorted(dict_counts.items()))
plt.bar(list(dict_counts_sorted.keys()), dict_counts_sorted.values(), color='b')
plt.show()
```



Write your explanation here: There are a number of reasons we are interested in the histogram above which shows for each length of headline its amount of instances in our training set. When we know the distribution of headline lengths it helps us understand the expected range of headline lengths that the decoder needs to generate. So when we produce a new headline it helps us to understand whether what has been obtained makes sense or not and thus we will know whether to ignore that specific generation of the model. Secondly while training our model we want to avoid series lengths that are relatively rare (do not appear in a reasonable amount in our training set) so from this histogram we learn which headlines lengths we will refer to during training and which will not.

1.1.2 Part (b) -- 5%

How many distinct words appear in the training data? Exclude the `<bos>` and `<eos>` tags in your computation.

[5]:

```
# Report your values here. Make sure that you report the actual values,
# and not just the code used to get those values
# You might find the python class Counter from the collections package useful
from collections import Counter

concat_list = []
for example in train_data:
    concat_list += example.title

counter_words = Counter(concat_list)
del counter_words['<bos>']
del counter_words['<eos>']
print(f'The num of distincs words in out training data is: {len(counter_words.
 ↪keys())}')
```

```
The num of distincs words in out training data is: 51298
```

1.1.3 Part (c) -- 5%

The distribution of *words* will have a long tail, meaning that there are some words that will appear very often, and many words that will appear infrequently. How many words appear exactly once in the training set? Exactly twice? Print these numbers below

[6]:

```
# Report your values here. Make sure that you report the actual values,
# and not just the code used to get those values
key_list_1 = [k for k,v in counter_words.items() if float(v) == 1]
key_list_2 = [k for k,v in counter_words.items() if float(v) == 2]

print(f"The numer of words that apear once in the dictionary is :␣
 ↪{len(key_list_1)}")
print(f"The numer of words that apear twice in the dictionary is :␣
 ↪{len(key_list_2)}")
```

```
The numer of words that apear once in the dictionary is : 19854
The numer of words that apear twice in the dictionary is : 7193
```

1.1.4 Part (d) -- 5%

We will replace the infrequent words with an `<unk>` tag, instead of learning embeddings for these rare words. `torchtext` also provides us with the `<pad>` tag used for padding short sequences for batching. We will thus only model the top 9995 words in the training set, excluding the tags `<bos>`, `<eos>`, `<unk>`, and `<pad>`.

What percentage of total word count(whole dataset) will be supported? Alternatively, what percentage of total word count(whole dataset) in the training set will be set to the `<unk>` tag?

[7]:

```
# Report your values here. Make sure that you report the actual values,
# and not just the code used to get those values

counter_most_common = counter_words.most_common()
sum_whole_datasest = sum(list(counter_words.values()))

count = 0
for idx , tup in enumerate(counter_most_common[9995:]):
    count += tup[1]
    del counter_words[tup[0]]

counter_words['unk'] = count

print(f"sum of whole dataset excluding the tags `<bos>`, `<eos>`, `<unk>`, and␣
 ↪`<pad>` is: {sum_whole_datasest}")
print(f"sum of the top 9995 words in the training set is: {sum_whole_datasest -␣
 ↪counter_words['unk']}")
print(f"The percentage of the most common words is: {((sum_whole_datasest -␣
 ↪counter_words['unk']) / sum_whole_datasest ) * 100 }%")
```

```
sum of whole dataset excluding the tags `<bos>`, `<eos>`, `<unk>`, and `<pad>`
is: 1898155
sum of the top 9995 words in the training set is: 1783859
The percentage of the most common words is: 93.97857393100142%
```

The `torchtext` package will help us keep track of our list of unique words, known as a **vocabulary**. A vocabulary also assigns a unique integer index to each word.

[8]:

```
# Build the vocabulary based on the training data. The vocabulary
# can have at most 9997 words (9995 words + the <bos> and <eos> token)
text_field.build_vocab(train_data, max_size=9997)

# This vocabulary object will be helpful for us
vocab = text_field.vocab
print(vocab.stoi["hello"]) # for instances, we can convert from string to␣
 ↪(unique) index
print(vocab.itos[10])      # ... and from word index to string

# The size of our vocabulary
vocab_size = len(text_field.vocab.stoi)

# Here are the two tokens that torchtext adds for us:
print(vocab.itos[0]) # <unk> represents an unknown word not in our vocabulary
print(vocab.itos[1]) # <pad> will be used to pad short sequences for batching
```

```
0
on
<unk>
<pad>
```

1.2 Question 2. Text Autoencoder (40%)

Building a text autoencoder is a little more complicated than an image autoencoder like we did in class. So we will need to thoroughly understand the model that we want to build before actually building it. Note that the best and fastest way to complete this assignment is to spend time upfront understanding the architecture. The explanations are quite dense, but it is important to understand the operation of this model. The rationale here is similar in nature to the `seq2seq` RNN model we discussed in class, only we are dealing with unsupervised learning here rather than machine translation.

2 Architecture description

Here is a diagram showing our desired architecture:

There are two main components to the model: the **encoder** and the **decoder**. As always with neural networks, we'll first describe how to make **predictions** with of these components. Let's get started:

The **encoder** will take a sequence of words (a headline) as *input*, and produce an embedding (a vector) that represents the entire headline. In the diagram above, the vector $\mathbf{h}^{(7)}$ is the vector embedding containing information about the entire headline. This portion is very similar to the sentiment analysis RNN that we discussed in lecture (but without the fully-connected layer that makes a prediction).

The **decoder** will take an embedding (in the diagram, the vector $\mathbf{h}^{(7)}$) as input, and uses a separate RNN to **generate a sequence of words**. To generate a sequence of words, the decoder needs to do the following:

1. Determine the previous word that was generated. This previous word will act as $\mathbf{x}^{(t)}$ to our RNN, and will be used to update the hidden state $\mathbf{m}^{(t)}$. Since each of our sequences begin with the `<bos>` token, we'll set $\mathbf{x}^{(1)}$ to be the `<bos>` token.
2. Compute the updates to the hidden state $\mathbf{m}^{(t)}$ based on the previous hidden state $\mathbf{m}^{(t-1)}$ and $\mathbf{x}^{(t)}$. Intuitively, this hidden state vector $\mathbf{m}^{(t)}$ is a representation of *all the words we still need to generate*.
3. We'll use a fully-connected layer to take a hidden state $\mathbf{m}^{(t)}$, and determine *what the next word should be*. This fully-connected layer solves a *classification problem*, since we are trying to choose a word out of $K =$ `vocab_size` distinct words. As in a classification problem, the fully-connected neural network will compute a *probability distribution* over these `vocab_size` words. In the diagram, we are using $\mathbf{z}^{(t)}$ to represent the logits, or the pre-softmax activation values representing the probability distribution.
4. We will need to *sample* an actual word from this probability distribution $\mathbf{z}^{(t)}$. We can do this in a number of ways, which we'll discuss in question 3. For now, you can imagine your favourite way of picking a word given a distribution over words.
5. This word we choose will become the next input $\mathbf{x}^{(t+1)}$ to our RNN, which is used to update our hidden state $\mathbf{m}^{(t+1)}$, i.e., to determine what are the remaining words to be generated.

We can repeat this process until we see an `<eos>` token generated, or until the generated sequence becomes too long.

3 Training the architecture

While our autoencoder produces a sequence, computing the loss by comparing the complete generated sequence to the ground truth (the encoder input) gives rise to multiple challanges. One is that the generated sequence might be longer or shorter than the actual sequence, meaning that there may be more/fewer $\mathbf{z}^{(t)}$s than ground-truth words. Another more insidious issue is that the **gradients will become very high-variance and unstable**, because **early mistakes will easily throw the model off-track**. Early in training, our model is unlikely to produce the right answer in step $t = 1$, so the gradients we obtain based on the other time steps will not be very useful.

At this point, you might have some ideas about "hacks" we can use to make training work. Fortunately, there is one very well-established solution called **teacher forcing** which we can use for training: instead of *sampling* the next word based on $\mathbf{z}^{(t)}$, we will forget sampling, and use the **ground truth $\mathbf{x}^{(t)}$** as the input in the next step.

Here is a diagram showing how we can use **teacher forcing** to train our model:

We will use the RNN generator to compute the logits $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \cdots \mathbf{z}^{(T)}$. These distributions can be compared to the ground-truth words using the cross-entropy loss. The loss function for this model will be the sum of the losses across each $t \in \{1, \ldots, T\}$.

We'll train the encoder and decoder model simultaneously. There are several components to our model that contain tunable weights:

- The word embedding that maps a word to a vector representation. In theory, we could use GloVe embeddings, as we did in class. In this assignment we will not do that, but learn the word embedding from data. The word embedding component is represented with blue arrows in the diagram.
- The encoder RNN (which will use GRUs) that computes the embedding over the entire headline. The encoder RNN is represented with black arrows in the diagram.
- The decoder RNN (which will also use GRUs) that computes hidden states, which are vectors representing what words are to be generated. The decoder RNN is represented with gray arrows in the diagram.
- The **projection MLP** (a fully-connected layer) that computes a distribution over the next word to generate, given a decoder RNN hidden state. The projection is represented with green arrows

3.1 Part (a) -- 20%

Complete the code for the AutoEncoder class below by:

1. Filling in the missing numbers in the `__init__` method using the parameters `vocab_size`, `emb_size`, and `hidden_size`.

2. Complete the `forward` method, which uses teacher forcing and computes the logits $\mathbf{z}^{(t)}$ of the reconstruction of the sequence.

You should first try to understand the `encode` and `decode` methods, which are written for you. The `encode` method bears much similarity to the RNN we wrote in class for sentiment analysis. The `decode` method is a bit more challenging. You might want to scroll down to the `sample_sequence` function to see how this function will be called.

You can (but don't have to) use the `encode` and `decode` method in your `forward` method. In either case, be careful of the input that you feed into ether `decode` or to `self.decoder_rnn`. Refer to the teacher-forcing diagram. **bold text** Notice that batch_first is set to True, understand how deal with it.

[9]:
```
class AutoEncoder(nn.Module):
    def __init__(self, vocab_size, emb_size, hidden_size):
        """
        A text autoencoder. The parameters
            - vocab_size: number of unique words/tokens in the vocabulary
            - emb_size: size of the word embeddings $x^{(t)}$
            - hidden_size: size of the hidden states in both the
                           encoder RNN ($h^{(t)}$) and the
                           decoder RNN ($m^{(t)}$)
        """
        super().__init__()
        self.embed = nn.Embedding(num_embeddings=vocab_size, # TODO
                                  embedding_dim=emb_size)   # TODO
        self.encoder_rnn = nn.GRU(input_size=emb_size, #TODO
                                  hidden_size=hidden_size, #TODO
                                  batch_first=True)
        self.decoder_rnn = nn.GRU(input_size=emb_size, #TODO
                                  hidden_size=hidden_size, #TODO
                                  batch_first=True)
        self.proj = nn.Linear(in_features=hidden_size, # TODO
                              out_features=vocab_size) # TODO

    def encode(self, inp):
        """
        Computes the encoder output given a sequence of words.
        """
        emb = self.embed(inp)
        out, last_hidden = self.encoder_rnn(emb)
        return last_hidden

    def decode(self, inp, hidden=None):
        """
        Computes the decoder output given a sequence of words, and
        (optionally) an initial hidden state.
        """
        emb = self.embed(inp)
        out, last_hidden = self.decoder_rnn(emb, hidden)
        out_seq = self.proj(out)
        return out_seq, last_hidden

    def forward(self, inp):
        """
        Compute both the encoder and decoder forward pass
        given an integer input sequence inp with shape [batch_size,␣
 ↪seq_length],
        with inp[a,b] representing the (index in our vocabulary of) the b-th␣
↪word
        of the a-th training example.

        This function should return the logits $z^{(t)}$ in a tensor of shape
        [batch_size, seq_length - 1, vocab_size], computed using *teaching␣
↪forcing*.

        The (seq_length - 1) part is not a typo. If you don't understand why
        we need to subtract 1, refer to the teacher-forcing diagram above.
        """
        last_hidden = self.encode(inp)
        out, last_hidden = self.decode(inp[:,:-1], hidden=last_hidden)
        return out
```

3.1.1 Part (b) -- 10%

To check that your model is set up correctly, we'll train our autoencoder neural network for at least 300 iterations to memorize this sequence:

[10]:
```
headline = train_data[42].title
input_seq = torch.Tensor([vocab.stoi[w] for w in headline]).long().unsqueeze(0)
```

[11]:
```
print(headline)
print(input_seq)
```

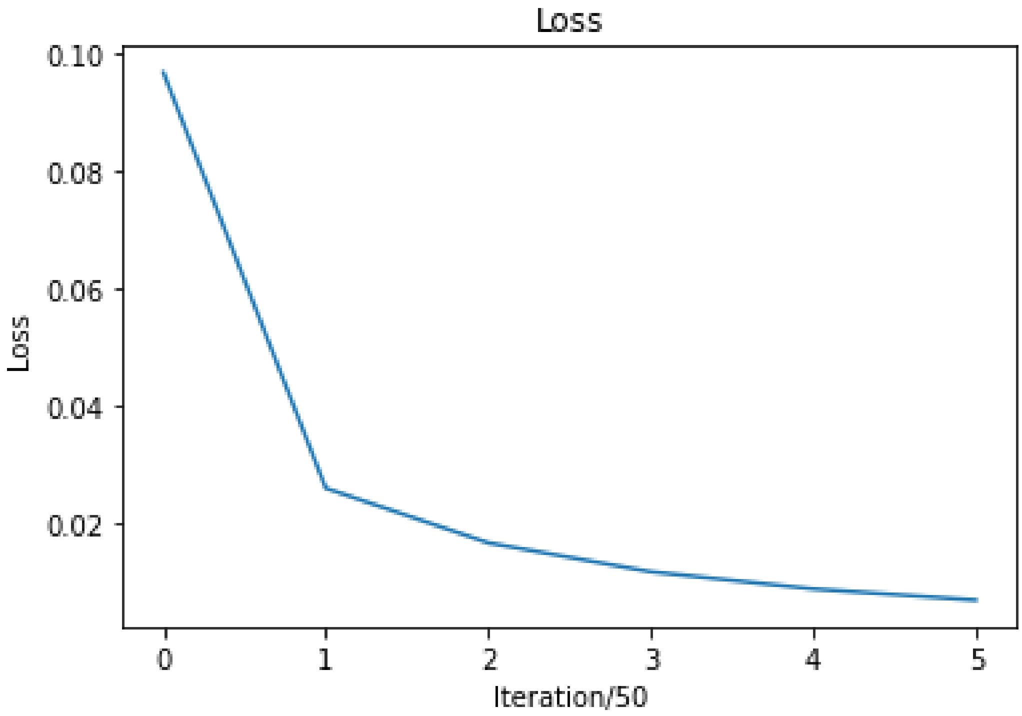
```
['<bos>', 'zambian', 'president', 'swears', 'in', 'new', 'army', 'chief',
'<eos>']
tensor([[   2, 5258,   91, 9117,    6,   25,  637,  118,    3]])
```

We are looking for the way that you set up your loss function corresponding to the figure above. Be careful of off-by-one errors here.

Note that the Cross Entropy Loss expects a rank-2 tensor as its first argument (the output of the network), and a rank-1 tensor as its second argument (the true label). You will need to properly reshape your data to be able to compute the loss.

[12]:
```
import matplotlib.pyplot as plt

model = AutoEncoder(vocab_size, 128, 128)
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
loss_list = []
for it in range(300):

    # compute prediction logit
    zs = model(input_seq)
    # compute the total loss
    loss = criterion(zs[0], input_seq[0][1:].long())
    # zero the gradients before we calc our gradients is a clean up step
    # for PyTorch
    optimizer.zero_grad()
    # backward pass to compute the gradient of loss with respect to our
    # learnable params
    loss.backward()
    # make the updates for each parameter with our optimizer that we
    # define earlier (Adam)
    optimizer.step()

    if (it+1) % 50 == 0:
        print("[Iter %d] Loss %f" % (it+1, float(loss)))
        loss_list.append(float(loss))

plt.plot(loss_list)
plt.ylabel('Loss')
plt.xlabel('Iteration/50')
plt.title('Loss')
plt.show()
```

```
[Iter 50] Loss 0.096804
[Iter 100] Loss 0.025849
[Iter 150] Loss 0.016506
[Iter 200] Loss 0.011563
[Iter 250] Loss 0.008629
[Iter 300] Loss 0.006718
```



3.1.2 Part (c) -- 4%

Once you are satisfied with your model, encode your input using the RNN encoder, and sample some sequences from the decoder. The sampling code is provided to you, and performs the computation from the first diagram (without teacher forcing).

Note that we are sampling from a multi-nomial distribution described by the logits $z^{(t)}$. For example, if our distribution is [80%, 20%] over a vocabulary of two words, then we will choose the first word with 80% probability and the second word with 20% probability.

Call `sample_sequence` at least 5 times, with the default temperature value. Make sure to include the generated sequences in your PDF report.

[13]:

```python
def sample_sequence(model, hidden, max_len=20, temperature=1):
    """
    Return a sequence generated from the model's decoder
        - model: an instance of the AutoEncoder model
        - hidden: a hidden state (e.g. computed by the encoder)
        - max_len: the maximum length of the generated sequence
        - temperature: described in Part (d)
    """
    # We'll store our generated sequence here
    generated_sequence = []
    # Set input to the <BOS> token
    inp = torch.Tensor([text_field.vocab.stoi["<bos>"]]).long()
    for p in range(max_len):
        # compute the output and next hidden unit
        output, hidden = model.decode(inp.unsqueeze(0), hidden)
        # Sample from the network as a multinomial distribution
        output_dist = output.data.view(-1).div(temperature).exp()
        top_i = int(torch.multinomial(output_dist, 1)[0])
        # Add predicted word to string and use as next input
        word = text_field.vocab.itos[top_i]
        # Break early if we reach <eos>
        if word == "<eos>":
            break
        generated_sequence.append(word)
        inp = torch.Tensor([top_i]).long()
    return generated_sequence


# Your solutions go here
headline = train_data[0].title
input_seq = torch.Tensor([vocab.stoi[w] for w in headline]).long().unsqueeze(0)
last_hidden = model.encode(input_seq)
for i in range(5):
    print(sample_sequence(model, last_hidden))
```

```
['zambian', 'president', 'swears', 'in', 'new', 'army', 'chief']
['zambian', 'president', 'swears', 'in', 'new', 'army', 'chief']
['zambian', 'president', 'swears', 'in', 'new', 'army', 'chief']
['zambian', 'president', 'swears', 'in', 'new', 'army', 'chief']
['zambian', 'president', 'swears', 'in', 'new', 'army', 'chief']
```

3.1.3 Part (d) -- 6%

The multi-nomial distribution can be manipulated using the `temperature` setting. This setting can be used to make the distribution "flatter" (e.g. more likely to generate different words) or "peakier" (e.g. less likely to generate different words).

Call `sample_sequence` at least 5 times each for at least 3 different temperature settings (e.g. 1.5, 2, and 5). Explain why we generally don't want the temperature setting to be too **large**.

[14]:

```
# Include the generated sequences and explanation in your PDF report.
headline = train_data[56].title
input_seq = torch.Tensor([vocab.stoi[w] for w in headline]).long().unsqueeze(0)
last_hidden = model.encode(input_seq)
temperature = [1.5, 2, 5]
for temp in temperature:
    print(f'temperature is: {temp}')
    for i in range(5):
        print(sample_sequence(model, last_hidden, max_len=20, temperature=temp))
```

```
temperature is: 1.5
['zambian', 'president', 'finally', 'proposals', 'kone', '-executive',
'intensified', 'rhp', 'earth', 'spacex', 'loading', 'resumption', 'swears',
'kroger', 'idlib', 'doubtful', 'suitors', 'swiss', 'auditor', 'held']
['afghan', 'zambian', 'main', 'qualifying', 'zambian', 'survive', 'outbreaks',
'socialism', 'swears', 'in', 'new', 'army', 'appetite', 'djokovic', 'soaring',
'delisting', 'swears', 'chief', 'migrant', 'heathrow']
['zambian', 'president', 'traffic', 'flash', 'organisers', 'spying', 'manbij',
'chief']
['zambian', 'president', 'swears', 'idai', 'swears', 'ocado', 'elysee', 'bln',
'procedure', 'deere', 'formula', 'carcinogen', 'after', 'new', 'army', 'chief',
'-finance']
['zambian', 'president', 'releases', 'port', 'reviewing', 'figures', 'landing',
'chief', 'champion', 'tainted', 'protection', 'enclave', 'fake']
temperature is: 2
['atletico', 'bubble', 'therapeutics', 'convicts', 'kenyan', 'javid', 'outage',
'_num_-amazon', 'entry', 'agree', 'away', 'deposit', 'a321xlr', 'lures', 'iran-
backed', '_num_-no', 'army', 'heavily', 'chief', 'mid-year']
['reaching', 'zambian', 'include', 'reais', 'militant', 'charity', 'loyal',
'becomes', 'focuses', 't-mobile/sprint', 'sweet', 'new', 'raab', 'european',
'indonesian', 'reason', 'msci', 'schools', 'modern', 'rice']
['gunmen', 'rivals', 'q4', 'with', 'addition', 'feature', 'shifting', 'nordisk',
'moves', 'top', 'blamed', 'firms', 'chad', 'andrew', 'mortgages', 'extends',
'politics', 'stand', 'chuquicamata', 'reviews']
```

```
['discussions', 'hunt', 'judgment', 'injury', 'pemex', 'rb', 'eight-month',
'instructure', 'guardian', 'tiananmen', 'appoint', 'coalition', 'rabbi',
'summer', 'payments', 'interior', '_num_-day', 'ciudadanos', 'talc', 'leasing']
['zambian', 'show', 'disrupts', 'clinical', 'mogadishu', 'legitimate',
'subdued', 'chief', 'gunfire', '_num_-dish', 'shrug', 'industry', 'fly',
'friday', 'pares', 'annual', 'oversupply', 'heightening', 'hiring', 'gap']
temperature is: 5
['norms', 'solidarity', 'jackson', 'restart', 'in', 'a380', 'kids', 'work',
'crazy', '_num_-star', 'mild', 'argentina', 'hearing', '10th', 'renewable',
'teargas', 'auger-aliassime', 'pedo', 'biggest', 'cop']
['darfur', 'record', 'coordination', 'modestly', 'cannabis', 'words', 'all-
stock', 'recognise', 'one-two', 'coordinated', 'back-to-back', 'palo', 'elect',
'armour', 'sabine', 'despair', 'classified', 'survive', 'brighter', 'manbij']
['mtn', '_num_-kraft', 'sale', 'sarri', '-central', 'emerges', 'snb', 'highly',
'cancer-causing', 'nikkei', 'gang', 'evacuates', 'off', 'jan', 'alibaba',
'statement', 'television', 'parties', 'easing', 'kyodo']
['makers', 'decisive', 'cypress', 'first-half', 'cleaner', 'downgrades',
'toxic', 'tass', 'andreescu', 'slides', 'fun', 'graham', 'btg', 'flu',
'narrows', 'sotheby', 'trying', 'whitaker', 'exports', 'shunned']
['arriving', 'weedkiller', 'boxing', 'cows', 'counter', 'over', 'traveling',
'hurting', 'dollars', 'surcharge', 'pledges', 'your', 'tens', 'shrink', 'abb',
'codelco', 'affordable', 'mounts', 'unveil', 'lied']
```

We see that in the sample_sequence function we get the origin of the decoder probabilities for each of the words that are in our vocab and then we divide by temperature and take exp on what was obtained. Let us note that we do not want the value of temperature to be high this is since the distribution for picking a word is flattened.

We will illustrate this by an example: Suppose in our dictionary there are two words w1, w2 (i.e. in the decoder output we extract a 2-length probability vector throughout the series in the output).

Suppose we get at the output of the decoder two words w1, w2 with corresponding probabilities of 0.7 and 0.3 respectively i.e. $p(w1) = 0.7, p(w2) = 0.3$. When we use **temperature=1** then we make a transformation to our probability vector as we explained above and therefore we get $P\left(\omega_1\right)|_{t=1} = e^{0.7}$, $P\left(\omega_2\right)|_{t=1} = e^{0.3}$ ==> $\frac{P(\omega_1)|_{t=1}}{P(\omega_2)|_{t=1}} = e^{0.4}$. When the temperature is high, for example, **temperature=10**, then $P\left(\omega_1\right)|_{t=10} = e^{0.07}$, $P\left(\omega_2\right)|_{t=10} = e^{0.03}$, and therefore $\frac{P(\omega_1)|_{t=10}}{P(\omega_2)|_{t=10}} = e^{0.04} \approx 1$. We can see that as the temperature is increasing,the ratio between the probabilities which had high value to ones with low values is decreasing, and therefore the distribution is flattened, and the model is more likely the generate different words. A very high temperature will destroy the learned distribution and probably produced words that do not make sense to the context of the previous word, and consequently, generate a all sentece which doesn't make sense.

3.2 Question 3. Data augmentation (20%)

It turns out that getting good results from a text auto-encoder is very difficult, and that it is very easy for our model to **overfit**. We have discussed several methods that we can use to prevent overfitting, and we'll introduce one more today: **data augmentation**.

The idea behind data augmentation is to artificially increase the number of training examples by "adding noise" to the image. For example, during AlexNet training, the authors randomly cropped 224 × 224 regions of a 256 × 256 pixel image to increase the amount of training data. The authors also flipped the image left/right. Machine learning practitioners can also add Gaussian noise to the image.

When we use data augmentation to train an *autoencoder*, we typically to only add the noise to the input, and expect the reconstruction to be *noise free*. This makes the task of the autoencoder even more difficult. An autoencoder trained with noisy inputs is called a **denoising auto-encoder**. For simplicity, we will *not* build a denoising autoencoder today.

3.2.1 Part (a) -- 5%

We will add noise to our headlines using a few different techniques:

1. Shuffle the words in the headline, taking care that words don't end up too far from where they were initially
2. Drop (remove) some words
3. Replace some words with a blank word (a `<pad>` token)
4. Replace some words with a random word

The code for adding these types of noise is provided for you:

[15]:
```
def tokenize_and_randomize(headline,
                           drop_prob=0.1,  # probability of dropping a word
                           blank_prob=0.1, # probability of "blanking" out a␣
 ↪word
                           sub_prob=0.1,   # probability of substituting a word␣
 ↪with a random one
                           shuffle_dist=3): # maximum distance to shuffle a␣
 ↪word
    """
    Add 'noise' to a headline by slightly shuffling the word order,
    dropping some words, blanking out some words (replacing with the <pad>␣
 ↪token)
    and substituting some words with random ones.
    """
    headline = [vocab.stoi[w] for w in headline.split()]
    n = len(headline)
    # shuffle
    headline = [headline[i] for i in get_shuffle_index(n, shuffle_dist)]

    new_headline = [vocab.stoi['<bos>']]
    for w in headline:
        if random.random() < drop_prob:
            # drop the word
            pass
        elif random.random() < blank_prob:
            # replace with blank word
            new_headline.append(vocab.stoi["<pad>"])
        elif random.random() < sub_prob:
            # substitute word with another word
            new_headline.append(random.randint(0, vocab_size - 1))
        else:
            # keep the original word
            new_headline.append(w)
    new_headline.append(vocab.stoi['<eos>'])
    return new_headline

def get_shuffle_index(n, max_shuffle_distance):
    """ This is a helper function used to shuffle a headline with n words,
    where each word is moved at most max_shuffle_distance. The function does
    the following:
       1. start with the *unshuffled* index of each word, which
          is just the values [0, 1, 2, ..., n]
       2. perturb these "index" values by a random floating-point value between
          [0, max_shuffle_distance]
       3. use the sorted position of these values as our new index
    """
    index = np.arange(n)
    perturbed_index = index + np.random.rand(n) * 3
    new_index = sorted(enumerate(perturbed_index), key=lambda x: x[1])
    return [index for (index, pert) in new_index]
```

Call the function tokenize_and_randomize 5 times on a headline of your choice. Make sure to include both your original headline, and the five new headlines in your report.

[16]:

```
# Report your values here. Make sure that you report the actual values,
# and not just the code used to get those values
# Include the generated sequences and explanation in your PDF report.

headline = train_data[56].title
headline = ' '.join(word for word in headline)
print("original headline: '{}'".format(headline))
print('\n')
for iter in range(5):
    print(f"headline number {iter}")
    header_output = tokenize_and_randomize(headline)
    header_output= [text_field.vocab.itos[index] for index in header_output]
    header_output = ' '.join(word for word in header_output)
    print(header_output)
```

```
original headline: '<bos> congo cuts internet for second day to avert chaos
before poll results <eos>'


headline number 0
<bos> <bos> congo internet second for day to chaos before results <eos> <eos>
headline number 1
<bos> <pad> congo for cuts <pad> <pad> day to avert before poll results aid
```

```
<eos>
headline number 2
<bos> <pad> congo internet for to day second chaos before poll results epa <eos>
headline number 3
<bos> policymakers congo cuts <pad> internet second to restored tussle poll
before results <pad> <eos>
headline number 4
<bos> <bos> cuts ops internet day second sanaa brisbane chaos before poll <eos>
<eos>
```

3.2.2 Part (b) -- 8%

The training code that we use to train the model is mostly provided for you. The only part we left blank are the parts from Q2(b). Complete the code, and train a new AutoEncoder model for 1 epoch. You can train your model for longer if you want, but training tend to take a long time, so we're only checking to see that your training loss is trending down.

If you are using Google Colab, you can use a GPU for this portion. Go to "Runtime" => "Change Runtime Type" and set "Hardware acceleration" to GPU. Your Colab session will restart. You can move your model to the GPU by typing `model.cuda()`, and move other tensors to GPU (e.g. `xs = xs.cuda()`). To move a model back to CPU, type `model.cpu`. To move a tensor back, use `xs = xs.cpu()`. For training, your model and inputs need to be on the *same device*.

[17]:

```python
def train_autoencoder(model, batch_size=64, learning_rate=0.001, num_epochs=10):
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    criterion = nn.CrossEntropyLoss()
    losses_per_epochs = []
    losses_iter = []
    for ep in range(num_epochs):
        sum = 0

        # We will perform data augmentation by re-reading the input each time
        field = data.Field(sequential=True,
                           tokenize=tokenize_and_randomize, # <-- data augmentation
                           include_lengths=True,
                           batch_first=True,
                           use_vocab=False, # <-- the tokenization function replaces this
                           pad_token=vocab.stoi['<pad>'])
        dataset = data.TabularDataset(train_path, "tsv", [('title', field)])

        # This BucketIterator will handle padding of sequences that are not of the same length
        train_iter = data.BucketIterator(dataset,
                                         batch_size=batch_size,
                                         sort_key=lambda x: len(x.title), # to minimize padding
                                         repeat=False)
        for it, ((xs, lengths), _) in enumerate(train_iter):
            # Fill in the training code here
            # compute prediction logit
            zs = model(xs.to(device))
            target = xs[:, 1:].long().to(device)
            # compute the total loss
            loss = criterion(zs.transpose(1, 2), target)
            # zero the gradients before we calc our gradients is a clean up step
            # for PyTorch
            optimizer.zero_grad()
            # backward pass to compute the gradient of loss with respect to our
            # learnable params
            loss.backward()
            # make the updates for each parameter with our optimizer that we
            # define earlier (Adam)
            optimizer.step()

            # (optional) for calc loss per epoch
            sum += loss

            if ep == 0:
                losses_iter.append(loss)
                if (it+1) % 100 == 0: # we print our iteration loss at epoch 0
                    print("[Iter %d] Loss %f" % (it+1, float(loss)))


        losses_per_epochs.append(sum/(batch_size*it))
    return losses_iter, losses_per_epochs

        # Optional: Compute and track validation loss
        #val_loss = 0
        #val_n = 0
        #for it, ((xs, lengths), _) in enumerate(valid_iter):
        #    zs = model(xs)
        #    loss = None # TODO
        #    val_loss += float(loss)

# Include your training curve or output to show that your training loss is trending down
```

[18]:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Running with {device}")
model = AutoEncoder(vocab_size, 128, 128).to(device)
losses_iter, losses_per_epochs = train_autoencoder(model, batch_size=64, learning_rate=0.001, num_epochs=20)
```

```
Running with cuda
```

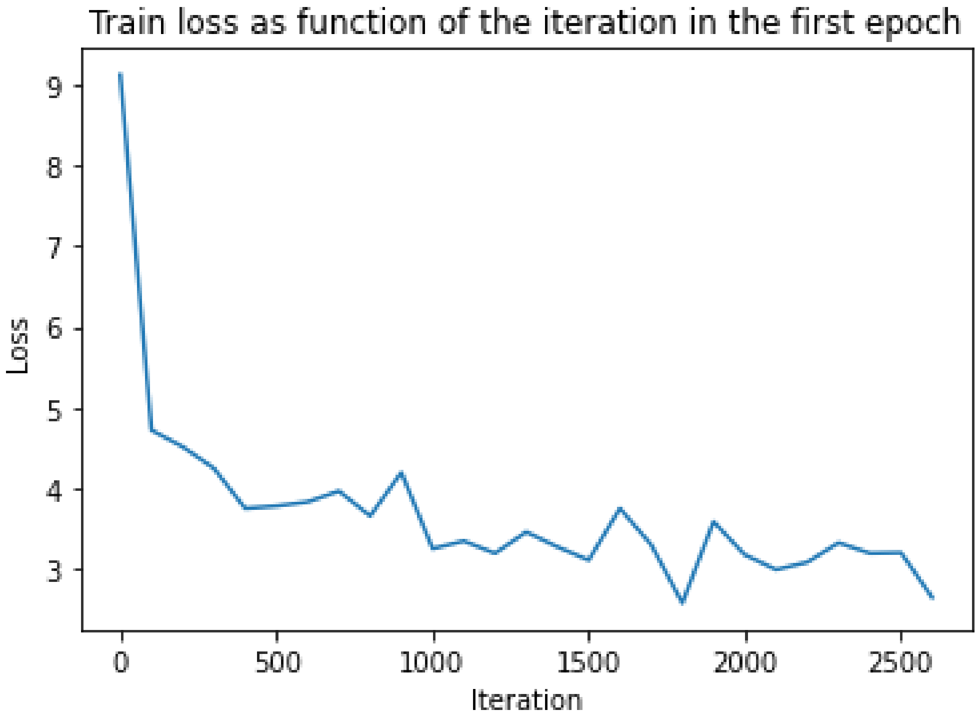
```
[Iter 100] Loss 4.221599
[Iter 200] Loss 4.181262
[Iter 300] Loss 4.847044
[Iter 400] Loss 4.212276
[Iter 500] Loss 3.773951
[Iter 600] Loss 3.705581
[Iter 700] Loss 3.895542
[Iter 800] Loss 3.638643
[Iter 900] Loss 3.564270
[Iter 1000] Loss 3.702795
[Iter 1100] Loss 3.544537
[Iter 1200] Loss 3.583328
[Iter 1300] Loss 2.702441
[Iter 1400] Loss 3.319961
[Iter 1500] Loss 3.106945
[Iter 1600] Loss 3.500937
[Iter 1700] Loss 3.442164
[Iter 1800] Loss 3.529682
[Iter 1900] Loss 3.374691
[Iter 2000] Loss 3.365557
[Iter 2100] Loss 3.066616
[Iter 2200] Loss 3.030756
[Iter 2300] Loss 2.684179
[Iter 2400] Loss 3.173525
[Iter 2500] Loss 3.095062
[Iter 2600] Loss 3.130269
```

[19]:

```python
plt.plot(range(1,len(losses_iter),100), losses_iter[1::100])
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Train loss as function of the iteration in the first epoch")
```

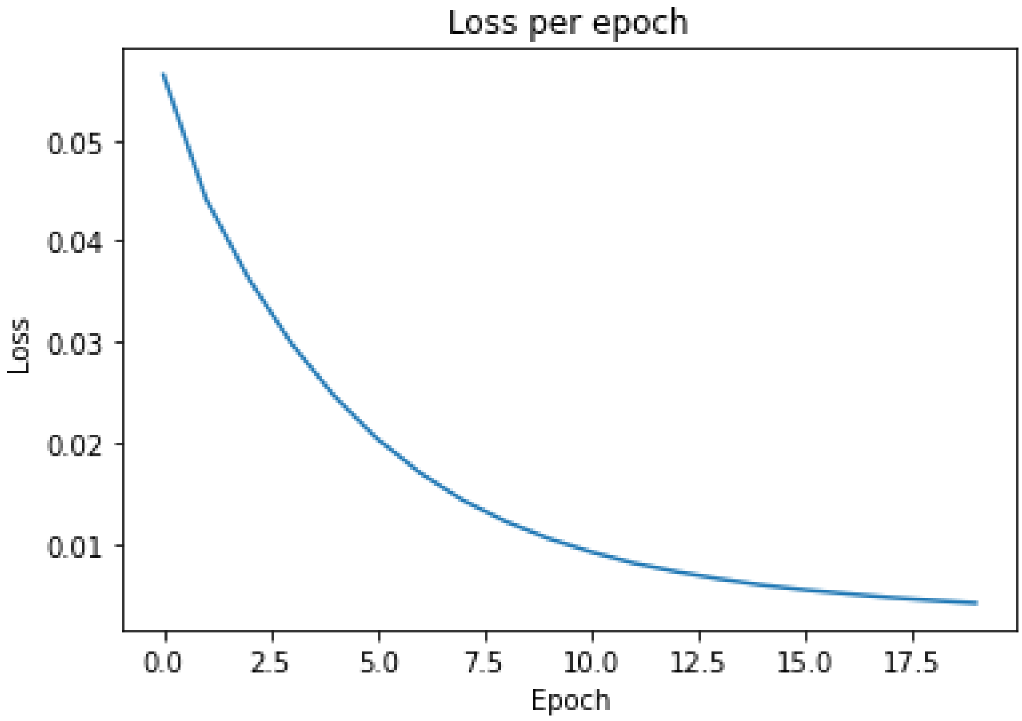
[19]:

```
Text(0.5, 1.0, 'Train loss as function of the iteration in the first epoch')
```



[20]:
```
plt.plot(losses_per_epochs)
plt.title('Loss per epoch')
plt.ylabel('Loss')
plt.xlabel('Epoch')
```

[20]:
```
Text(0.5, 0, 'Epoch')
```



3.2.3 Part (c) -- 7%

This model requires many epochs (>50) to train, and is quite slow without using a GPU. You can train a model yourself, or you can load the model weights that we have trained, and available on the course website (AE_RNN_model.pk).

Assuming that your `AutoEncoder` is set up correctly, the following code should run without error.

[21]:
```
model = AutoEncoder(10000, 128, 128)
checkpoint_path = '/content/gdrive/My Drive/AE_RNN_model.pk' # Update me
model.load_state_dict(torch.load(checkpoint_path))
```

[21]:
```
<All keys matched successfully>
```

Then, repeat your code from Q2(d), for `train_data[10].title` with temperature settings 0.7, 0.9, and 1.5. Explain why we generally don't want the temperature setting to be too **small**.

[22]:
```
# Include the generated sequences and explanation in your PDF report.

headline = train_data[10].title
input_seq = torch.Tensor([vocab.stoi[w] for w in headline]).unsqueeze(0).long()

# Include the generated sequences and explanation in your PDF report.
headline = train_data[56].title
input_seq = torch.Tensor([vocab.stoi[w] for w in headline]).long().unsqueeze(0)
last_hidden = model.encode(input_seq)
```

```python
temperature = [0.7, 0.9, 1.5]
print(f"The headline is: {headline}")
for temp in temperature:
    print(f'temperature is: {temp}')
    for i in range(5):
        print(sample_sequence(model, last_hidden, max_len=20, temperature=temp))
```

```
The headline is: ['<bos>', 'congo', 'cuts', 'internet', 'for', 'second', 'day',
'to', 'avert', 'chaos', 'before', 'poll', 'results', '<eos>']
temperature is: 0.7
['modi', 'indian', 'nursing', 'rise', 'to', 'german', 'completion', 'if',
'economy', 'iif', 'diners', 'data']
['activists', 'risk', '$', 'fuel', 'votes', 'east', 'of', 'levels', 'downturn',
'edges', 'may', 'country']
['border', '<unk>', 'spy', 'fuel', 'to', 'australia', 'protests', 'february',
'lagarde', 'stoking', 'says']
['activists', 'begin', 'coronavirus', 'cuts', 'police', 'for', 'clouds',
'crude', 'jets', '6th', 'johnson', 'says']
['border', '<unk>', 'employees', 'mexico', 'to', 'libya', 'tick', 'erupts',
'grows', 'world', 'stocks-canadian', ':']
temperature is: 0.9
['activists', 'double', 'mexico', 'amid', 'police', 'cuts', 'to', 'cdu',
'maximum', 'biggest', 'risk']
['modi', ',', 'state', 'fuel', 'to', 'leave', 'blow', 'risk', 'freezing',
'u.s-china', 'world', 'report']
['activists', 'risk', 'arms', ',', 'turn', 'five', 'u.s.', 'beirut', 'executed',
'hasbro', 'levels']
['activists', 'risk', 'detention', 'to', 'early', 'heads', 'to', 'grows', 'hub',
'iif', 'place', 'says']
['activists', 'begin', 'coronavirus', 'cuts', 'cup', 'mexico', 'sudan', 'spur',
'madoff', 'ease', 'poll']
temperature is: 1.5
['lethal', 'port', 'again', 'fuel', "'s", 'to', 'student', 'demand', 'stimulus',
'burial', 'says']
['portugal', 'visits', 'indian', 'fuel', '<unk>', 'higher', 'of', 'tankers',
'ambassadors', 'overhauls', ':', '<pad>']
['lawmakers', 'montreal', 'five', 'fuel', 'for', 'meet', 'says', 'tension',
'rate-cut', 'cook', 'jizan', 'april']
['houses', 'home', 'for', 'fuel', 'north', 'residents', 'deal', 'designates',
'javid', 'ria', 'september']
['lawmakers', 'stocks-banks', 'four', 'again', 'antitrust', 'sudan', 'no-show',
'power', 'slowing', 'stocks-saudi', 'poll']
```

Write your explanation here: We see that in the sample_sequence function we get the origin of the decoder probabilities for each of the words that are in our vocab and then we divide by temperature and take exp on what was obtained. Let us note that we do not want the value of temperature to be high this is since the distribution for picking a word is flattened.

We will illustrate this by an example: Suppose in our dictionary there are two words w1, w2 (i.e. in the decoder output we extract a 2-length probability vector throughout the series in the output).

Suppose we get at the output of the decoder two words w1, w2 with corresponding probabilities of 0.7 and 0.3 respectively i.e. $p(w1) = 0.7, p(w2) = 0.3$. When we use **temperature=1** then we make a transformation to our probability vector as we explained above and therefore we get $P\left(\omega_1\right)|_{t=1} = e^{0.7}$, $P\left(\omega_2\right)|_{t=1} = e^{0.3}$ ==> $\frac{P(\omega_1)|_{t=1}}{P(\omega_2)|_{t=1}} = e^{0.4}$. When the temperature is high, for example, **temperature=0.1**, then $P\left(\omega_1\right)|_{t=0.1} = e^{7}$, $P\left(\omega_2\right)|_{t=0.1} = e^{3}$, and therefore $\frac{P(\omega_1)|_{t=0.1}}{P(\omega_2)|_{t=0.1}} = e^{4}$

We will explain what we got: When the temperature was equal to 1 then we would choose w1 at 70% and w2 at 30%. When the temperature was chosen to be 0.1 then we got that the probability ratio is very high so it means we will choose w1 with a very high probability. So when we produce series at the decoder output with low temperature then our word range at the output will be very low and we will choose very specific words (our generalization capability is low). To conclude, If we want the decoder to perform as a generator of headlines, we should not use very low temperature as it withers the model, and will generate very similar headlines.

3.3 Question 4. Latent space manipulations (20%)

In parts 2-3, we've explored the decoder portion of the autoencoder. In this section, let's explore the **encoder**. In particular, the encoder RNN gives us embeddings of news headlines!

First, let's load the **validation** data set:

[23]:
```
valid_data = data.TabularDataset(
    path=valid_path,                # data file path
    format="tsv",                   # fields are separated by a tab
    fields=[('title', text_field)]) # list of fields (we have only one)
```

3.3.1 Part (a) -- 4%

Compute the embeddings of every item in the validation set. Then, store the result in a single PyTorch tensor of shape `[19046, 128]`, since there are 19,046 headlines in the validation set.

[24]:
```
# Write your code here
# Show that your resulting PyTorch tensor has shape `[19046, 128]`
embedded_output = torch.tensor([])
for i in range(len(valid_data)):
    headline = valid_data[i].title
    input_seq = torch.Tensor([vocab.stoi[w] for w in headline]).long().
 ↪unsqueeze(0)
    emb = model.encode(input_seq)
    embedded_output = torch.cat((embedded_output, emb[0, :]), dim=0)

print(embedded_output.size())
```

```
torch.Size([19046, 128])
```

3.3.2 Part (b) -- 4%

Find the 5 closest headlines to the headline `valid_data[13]`. Use the cosine similarity to determine closeness. (Hint: You can use code from assignment 2)

[25]:
```python
def cosine_similarity(t1, t2):
    t1 = t1.cpu().detach().numpy()
    t2 = t2.cpu().detach().numpy()
    return np.dot(t1,t2)/((np.dot(t1,t1)*np.dot(t2,t2))**0.5)
```

[26]:
```python
# Write your code here. Make sure to include the actual 5 closest headlines.
headline = valid_data[13].title

input_seq = torch.Tensor([vocab.stoi[w] for w in headline]).long().unsqueeze(0)
emb_headline_13 = model.encode(input_seq)
emb_headline_13 = torch.reshape(emb_headline_13, (-1, ))

min_cosine_similarity_list = [0, 0, 0, 0, 0]
indices_respetivly = [0, 0, 0, 0, 0]

for idx in range(len(valid_data)):
    if idx != 13: # we dont want to check cosine similarity between two vuctors␣
 ↪that identical because we it equal to 1
        headline = valid_data[idx].title
        input_seq = torch.Tensor([vocab.stoi[w] for w in headline]).long().
 ↪unsqueeze(0)
        emb_headline = model.encode(input_seq)
        emb_headline = torch.reshape(emb_headline, (-1, ))

        dist = cosine_similarity(emb_headline_13, emb_headline)
        if dist > min(min_cosine_similarity_list):
            index = min_cosine_similarity_list.
 ↪index(min(min_cosine_similarity_list))
            min_cosine_similarity_list[index] = dist
            indices_respetivly[index] = idx

print("The 5 closest headlines to valid_data[13] ('{}') are:".format(' '.
 ↪join(valid_data[13].title)))
for i in range(5):
  print("valid_data[{}]:".format(indices_respetivly[i])+str(i)+". ", "'"+' '.
 ↪join(valid_data[indices_respetivly[i]].title)+"'", "with similrary of {:.
 ↪4f}".format(min_cosine_similarity_list[i]))
```

```
The 5 closest headlines to valid_data[13] ('<bos> asia takes heart from new year
gains in u.s. stock futures <eos>') are:
valid_data[18747]:0.  '<bos> eu orders quarantine for staff who traveled to
northern italy <eos>' with similrary of 0.9299
valid_data[17141]:1.  '<bos> italy 's salvini loses aura of invincibility in
emilia setback <eos>' with similrary of 0.9309
```

```
valid_data[11109]:2.  '<bos> portugal 's moura pays tribute to cod fishermen at
milan fashion close <eos>' with similrary of 0.9281
valid_data[14946]:3.  '<bos> saudi , russia look to seal deeper output cuts with
oil producers <eos>' with similrary of 0.9306
valid_data[10372]:4.  '<bos> update _num_-italy 's prime minister says new
government will bicker less <eos>' with similrary of 0.9288
```

3.3.3 Part (c) -- 4%

Find the 5 closest headlines to another headline of your choice.

[27]:

```
# Write your code here. Make sure to include the actual 5 closest headlines.
headline = valid_data[12].title

input_seq = torch.Tensor([vocab.stoi[w] for w in headline]).long().unsqueeze(0)
emb_headline_12 = model.encode(input_seq)
emb_headline_12 = torch.reshape(emb_headline_12, (-1, ))

min_cosine_similarity_list = [0, 0, 0, 0, 0]
indices_respetivly = [0, 0, 0, 0, 0]

for idx in range(len(valid_data)):
    if idx != 12: # we dont want to check cosine similarity between two vwctors␣
 ↪that identical because we it equal to 1
        headline = valid_data[idx].title
        input_seq = torch.Tensor([vocab.stoi[w] for w in headline]).long().
 ↪unsqueeze(0)
        emb_headline = model.encode(input_seq)
        emb_headline = torch.reshape(emb_headline, (-1, ))

        dist = cosine_similarity(emb_headline_12, emb_headline)
        if dist > min(min_cosine_similarity_list):
            index = min_cosine_similarity_list.
 ↪index(min(min_cosine_similarity_list))
            min_cosine_similarity_list[index] = dist
            indices_respetivly[index] = idx

print("The 5 closest headlines to valid_data[12] ('{}') are:".format(' '.
 ↪join(valid_data[12].title)))
for i in range(5):
  print("valid_data[{}]:".format(indices_respetivly[i])+str(i)+". ", "'"+' '.
 ↪join(valid_data[indices_respetivly[i]].title)+"'", "with similrary of {:.
 ↪4f}".format(min_cosine_similarity_list[i]))
```

```
The 5 closest headlines to valid_data[12] ('<bos> south korea 's hyundai target
_num_ global sales of _num_ million vehicles <eos>') are:
valid_data[14358]:0.  '<bos> update _num_-egypt signs energy accords at
conference in new capital <eos>' with similrary of 0.9449
```

```
valid_data[15161]:1.  '<bos> south africa 's retail sales up _num_ % year/year
in october <eos>' with similrary of 0.9480
valid_data[6575]:2.  '<bos> online and discounters to drive _num_ % growth in uk
grocery by _num_ <eos>' with similrary of 0.9450
valid_data[6124]:3.  '<bos> south africa 's manufacturing up _num_ % y/y in
april , highest in _num_ years <eos>' with similrary of 0.9520
valid_data[10067]:4.  '<bos> south africa 's gross domestic spending up _num_ %
in second quarter <eos>' with similrary of 0.9514
```

3.3.4 Part (d) -- 8%

Choose two headlines from the validation set, and find their embeddings. We will **interpolate** between the two embeddings like we did in the example presented in class for training autoencoders on MNIST.

Find 3 points, equally spaced between the embeddings of your headlines. If we let e_0 be the embedding of your first headline and e_4 be the embedding of your second headline, your three points should be:

$$\begin{aligned} e_1 &= 0.75e_0 + 0.25e_4 \\ e_2 &= 0.50e_0 + 0.50e_4 \\ e_3 &= 0.25e_0 + 0.75e_4 \end{aligned}$$

Decode each of e_1, e_2 and e_3 five times, with a temperature setting that shows some variation in the generated sequences. Try to get a logical and cool sentence (this might be hard).

[28]:

```
# Write your code here. Include your generated sequences.

headline0 = valid_data[100].title
headline4 = valid_data[17887].title

input_seq = torch.Tensor([vocab.stoi[w] for w in headline0]).unsqueeze(0).long()
e0 = model.encode(input_seq)
input_seq = torch.Tensor([vocab.stoi[w] for w in headline4]).unsqueeze(0).long()
e4 = model.encode(input_seq)


e1 = 0.75*e0 + 0.25*e4
e2 = 0.50*e0 + 0.50*e4
e3 = 0.25*e0 + 0.75*e4

print("e0:", ' '.join(headline0))
print("e4:", ' '.join(headline4))

temperature = 1.5
print("temperature:",temperature)
es = ["e1", "e2", "e3"]
```

```python
for e_index, e in enumerate([e1, e2, e3]):
  print(es[e_index]," decodings:")
  for i in range(5):
    print(sample_sequence(model, e, temperature=temperature))
```

```
e0: <bos> amid u.s. withdrawal plans , u.s.-backed forces still fighting in
syria <eos>
e4: <bos> u.s. awaits china 's approval to send in experts as part of who team
<eos>
temperature: 1.5
e1  decodings:
['endorsement', 'zone', 'to', 'minerals', 'proof', ',', 'successful', 'safe',
'victory', 'years', 'venezuela']
['u.s.', 'waste', 'optimistic', 'delay', 'paulo', 'raising', 'iran', 'loses',
'forces', 'stena', 'next']
['u.s.', 'limits', 'visit', '_num_', 'arizona', 'wants', 'possible', 'strike',
'pass', 'after', 'findings']
['u.s.', 'troops', 'conte', 'years', 'visas', ',', 'u.s.-backed', 'pakistan',
'sector', 'pm', 'td']
['u.s.', 'favours', 'of', 'facilities', 'passes', 'billion', 'crashed',
'support', 'tells', 'u.s.-mexico', 'well']
e2  decodings:
['u.s.', 'slowdown', 'russia', 'on', 'antitrust', 'use', 'bidder', 'trip', 'in',
'bouteflika', 'talks', 'helm']
['u.s.', 'troops', 'faa', 'hopes', 'with', 'two', 'compromise', 'shortages',
'in', 'time', 'hope', 'government', 'ties']
['u.s.', 'surveys', "'s", 'deal', 'coverage', 'to', 'eastern', 'send', 'iraq',
'direct', 'brexit', 'around']
['u.s.', 'codelco', 'financial', 'hopes', 'or', 'ten', ',', 'forces',
'recognize', 'ties', 'after', 'domestic']
['u.s.', 'spike', 'avoid', 'new', 'plans', 'disrupted', 'over', 'from', 'syria',
'deliver', 'as', 'evacuation']
e3  decodings:
['u.s.', 'knows', 'diplomat', 'look', 'second', 'for', 'keep', 'alleged', 'in',
'whether', 'head', 'as', 'buyers']
['u.s.', 'post-election', 'plan', 'risks', 'indian', 'to', 'judicial', 'mourn',
'could', 'summaries', 'violence', 'u.s.', 'crisis']
['u.s.', 'risks', 'seek', '_num_', ',', 'loan', 'counterpart', 'keep', 'tanker',
'in', 'obamacare', 'election', 'weekend']
['u.s.', 'hearing', 'complete', 'indian', "'s", 'for', 'return', 'using',
'delayed', 'university', 'pm', 'initial', 'job']
['u.s.', 'send', 'says', 'malaysia', 'revive', ',', 'turkey', 'ban',
'accusation', 'proceedings', 'airbus', 'as', 'change']
```

[ ]:

```python
# for printing our word to a pdf file
!apt-get update && apt-get install alien
!apt-get install texlive texlive-xetex texlive-latex-extra pandoc
```

```
!pip install pypandoc

from google.colab import drive
drive.mount('/content/gdrive')
!cp /content/gdrive/My\ Drive/Colab\ Notebooks/Assignment4.ipynb ./
!jupyter nbconvert --to=pdf 'Assignment4.ipynb'
```

```
Get:1 https://cloud.r-project.org/bin/linux/ubuntu bionic-cran40/ InRelease
[3,626 B]
Ign:2 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64
InRelease
Get:3 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
Ign:4 https://developer.download.nvidia.com/compute/machine-
learning/repos/ubuntu1804/x86_64  InRelease
Get:5 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64
Release [696 B]
Hit:6 https://developer.download.nvidia.com/compute/machine-
learning/repos/ubuntu1804/x86_64  Release
Get:7 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64
Release.gpg [836 B]
Get:8 http://ppa.launchpad.net/c2d4u.team/c2d4u4.0+/ubuntu bionic InRelease
[15.9 kB]
Hit:10 http://archive.ubuntu.com/ubuntu bionic InRelease
Get:11
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64
Packages [867 kB]
Get:12 http://archive.ubuntu.com/ubuntu bionic-updates InRelease [88.7 kB]
Hit:13 http://ppa.launchpad.net/cran/libgit2/ubuntu bionic InRelease
Get:14 http://security.ubuntu.com/ubuntu bionic-security/universe amd64 Packages
[1,459 kB]
Get:15 http://ppa.launchpad.net/deadsnakes/ppa/ubuntu bionic InRelease [15.9 kB]
Get:16 http://archive.ubuntu.com/ubuntu bionic-backports InRelease [74.6 kB]
Get:17 http://archive.ubuntu.com/ubuntu bionic-updates/main amd64 Packages
[2,935 kB]
Hit:18 http://ppa.launchpad.net/graphics-drivers/ppa/ubuntu bionic InRelease
Get:19 http://ppa.launchpad.net/c2d4u.team/c2d4u4.0+/ubuntu bionic/main Sources
[1,823 kB]
Get:20 http://archive.ubuntu.com/ubuntu bionic-updates/universe amd64 Packages
[2,238 kB]
Get:21 http://security.ubuntu.com/ubuntu bionic-security/main amd64 Packages
[2,498 kB]
Get:22 http://archive.ubuntu.com/ubuntu bionic-updates/restricted amd64 Packages
[758 kB]
Get:23 http://security.ubuntu.com/ubuntu bionic-security/restricted amd64
Packages [725 kB]
Get:24 http://ppa.launchpad.net/c2d4u.team/c2d4u4.0+/ubuntu bionic/main amd64
Packages [934 kB]
```