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.

In [ ]:

```
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
import pdb

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

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:

In [ ]:

```
from google.colab import drive
drive.mount('/content/gdrive')

train_path = '/content/gdrive/Shareddrives/DNN_Elad_Raviv/ASS/ass4/reuters_train.txt' # U
pdate me
valid_path = '/content/gdrive/Shareddrives/DNN_Elad_Raviv/ASS/ass4/reuters_valid.txt' # U
pdate me
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call drive.moun
t("/content/gdrive", force_remount=True).
```

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.

In [ ]:

```
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 demonstrate
d 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)
```

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.

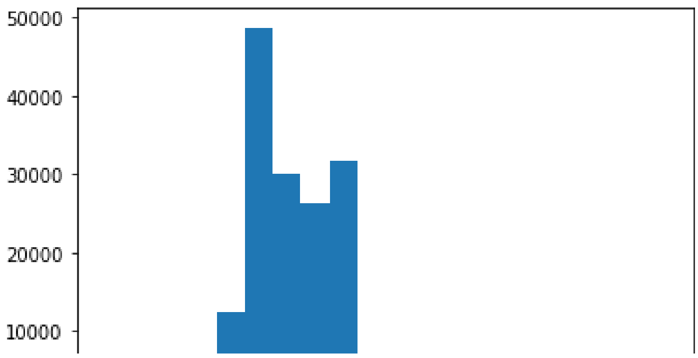
In [ ]:

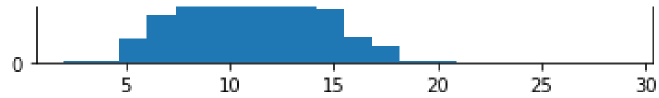
```
title_quntities = [len(data_sample.title) - 2 for data_sample in train_data]

plt.hist(title_quntities, bins=20)
```

Out[ ]:

```
(array([1.1200e+02, 2.9700e+02, 2.9310e+03, 6.1630e+03, 1.2529e+04,
        4.8726e+04, 2.9969e+04, 2.6251e+04, 3.1818e+04, 6.8520e+03,
        3.3640e+03, 2.0980e+03, 2.2700e+02, 7.7000e+01, 2.0000e+01,
        3.0000e+00, 4.0000e+00, 1.0000e+00, 0.0000e+00, 1.0000e+00]),
 array([ 2.  ,  3.35,  4.7 ,  6.05,  7.4 ,  8.75, 10.1 , 11.45, 12.8 ,
        14.15, 15.5 , 16.85, 18.2 , 19.55, 20.9 , 22.25, 23.6 , 24.95,
        26.3 , 27.65, 29.  ]),
 <a list of 20 Patch objects>)
```





Write your explanation here: This histogram may grasp our curiosity for a variety of reasons. 1) understanding the headline length distribution may aid us in determining the expected range of headline lengths that the decoder must create. It allows us to see if the produced headline length is fair (and if it isn't, we can disregard it), and therefore determine whether the model is working well.

2) while training the model, it may be preferable to avoid or treat headlines with short lengths or really long lengths more cautiously since it's deviate from the headlines' distribution.

3) (O) notation - run-time prespective Another reason we can think of is that knowing the distribution of the title's lengths can help us get an indication to the run time of the autoencoder's training, as well as to the decoder's sequence generation procces.

Part (b) -- 5%

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

In [ ]:

```
# 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
words = {}
for sample in train_data:
  for word in sample.title[1:-1]:
    words[word] = words.get(word, 0) + 1

print("There are", len(words.keys()),"distinct words (excluding the '<bos>' and '<eos>')"
)
```

```
There are 51298 distinct words (excluding the '<bos>' and '<eos>')
```

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

In [ ]:

```
# Report your values here. Make sure that you report the actual values,
# and not just the code used to get those values
print("words which appear twice: "+ str(len(list(filter(lambda x: x[1]==1, words.items()
)))))
print("words which appear twice: "+ str(len(list(filter(lambda x: x[1]==2, words.items()
)))))
```

```
words which appear twice: 19854
words which appear twice: 7193
```

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?

In [ ]:

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

words = [w for sample in train_data for w in sample.title]
word_count = Counter(words)

del word_count["<bos>"]
del word_count["<eos>"]
total_words = sum(word_count.values())

ordered_dict = OrderedDict(sorted(word_count.items(), key=lambda x: x[1],reverse=True))
supported_words_count = sum(list(ordered_dict.values())[:9995])
print(supported_words_count/total_words*100, "% of the dataset is supported")
```

```
93.97857393100142 % of the dataset is supported
```

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.

In [ ]:

```
# 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
```

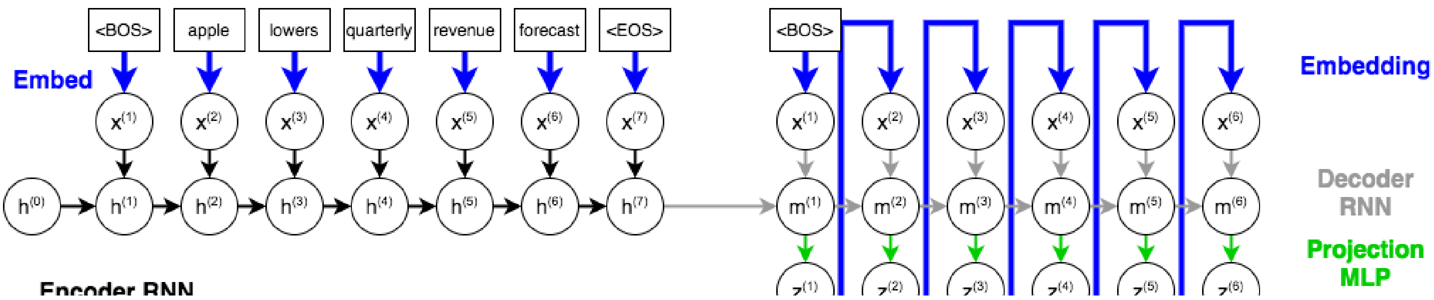
```
0
on
<unk>
<pad>
```

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.

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 $\mathrm{h}^{(7)}$ 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 $\mathrm{h}^{(7)}$) 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 $\mathrm{x}^{(t)}$ to our RNN, and will be used to update the hidden state $\mathrm{m}^{(t)}$. Since each of our sequences begin with the `<bos>` token, we'll set $\mathrm{x}^{(1)}$ to be the `<bos>` token.
2. Compute the updates to the hidden state $\mathrm{m}^{(t)}$ based on the previous hidden state $\mathrm{m}^{(t-1)}$ and $\mathrm{x}^{(t)}$. Intuitively, this hidden state vector $\mathrm{m}^{(t)}$ 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 $\mathrm{m}^{(t)}$, 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 $\mathrm{z}^{(t)}$ 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 $\mathrm{z}^{(t)}$. 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 $\mathrm{x}^{(t+1)}$ to our RNN, which is used to update our hidden state $\mathrm{m}^{(t+1)}$, i.e., to determine what are the remaining words to be generated.

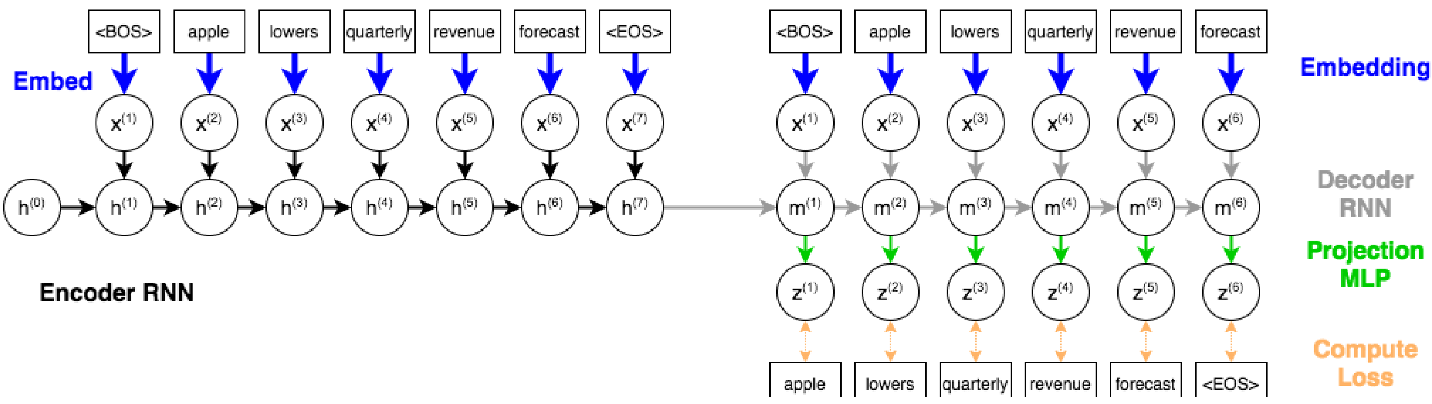
We can repeat this process until we see an `<eos>` token generated, or until the generated sequence becomes too long.

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 $\mathrm{z}^{(t)}$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 $\mathrm{z}^{(t)}$, we will forget sampling, and use the **ground truth** $\mathrm{x}^{(t)}$ 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 $z^{(1)}, z^{(2)}, \cdots z^{(T)}$. 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

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 $z^{(t)}$ 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.

In [ ]:

```
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,
                                  embedding_dim=emb_size)

        self.encoder_rnn = nn.GRU(input_size=emb_size,
                                  hidden_size=hidden_size,
                                  batch_first=True)
        self.decoder_rnn = nn.GRU(input_size=emb_size,
                                  hidden_size=hidden_size,
                                  batch_first=True)
        self.proj = nn.Linear(in_features=hidden_size,
                              out_features=vocab_size)

    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_seq_decoder, _ = self.decode(inp[:,:-1], last_hidden)
      return out_seq_decoder
```

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:

In [ ]:

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

In [ ]:

```
loss = nn.CrossEntropyLoss()
input = torch.randn(3, 5, requires_grad=True)
target = torch.empty(3, dtype=torch.long).random_(5)
print(input)

# target = torch.randn(3, 5)
print(target)
# target = target.softmax(dim=1)
# print(target)
```

```
tensor([[ 0.9666, -0.6603, -0.3783,  1.0674,  1.2788],
        [-0.4968,  0.9112,  0.0792,  2.8912,  0.3756],
        [ 0.0405,  0.0785,  0.5480, -0.1540,  0.1705]], requires_grad=True)
tensor([4, 0, 1])
```

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.

In [ ]:

```
model = AutoEncoder(vocab_size, 128, 128)
```

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
model.train()

def make_one_hot(input, vocab_size):
  identity_mat = np.eye(vocab_size)
  # convert true values from (1,8) to one-hot (8,10000)
  true_values = input[:,1:].view(-1,) # without <BOS>
  return torch.Tensor(np.stack([identity_mat[target] for target in true_values]))

true_values_one_hot = make_one_hot(input_seq, vocab_size)
for it in range(300):
  # pred is (1,8,10000)
  pred = model.forward(input_seq)
  # reshape to (8,10000)
  pred = pred[0] #pred.view(-1, pred.shape[2])

  loss = criterion(pred, true_values_one_hot)
  # loss2 = criterion(pred,true_values)
  optimizer.zero_grad()
  loss.backward()
  optimizer.step()

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

```
[Iter 50] Loss 0.109177
[Iter 100] Loss 0.026971
[Iter 150] Loss 0.016669
[Iter 200] Loss 0.011572
[Iter 250] Loss 0.008588
[Iter 300] Loss 0.006673
```

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.

In [ ]:

```
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)
        # print(output);
        # Sample from the network as a multinomial distribution
        output_dist = output.data.view(-1).div(temperature).exp()
        # print(output_dist);
        pdb.set_trace()
```

```
        # top_i = int(torch.argmax(outout_dist).detach.numpy())
        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

hidden = model.encode(input_seq)
print(sample_sequence(model, hidden))
```

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

In [ ]:

```
headline = train_data[42].title
input_seq = torch.Tensor([vocab.stoi[w] for w in headline]).long().unsqueeze(0)
hidden = model.encode(input_seq)
for i in range(5):
  print(sample_sequence(model, 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']
```

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.

Explanation:

As shown in the code above, we are using "torch.multinomial" distribution upon the exponent of the probability vector.

Because the distribution for picking a word is flattened for high tempratures, we don't want the temperature setting to be too high.

For example, if we have a 2-word vocabulary, w_1 and w_2, and the model produces probabilities of $p(w_1) = 0.9$, $p(w_2) = 0.1$.

1) for $temperature = 1$, after taking the exponent the probability will transform to $P_{t=1}(w_1) = e^{0.9}$, $P_{t=1}(w_2) = e^{0.1}$. so the ratio between them will be $\frac{P_{t=1}(w_1)}{P_{t=1}(w_2)} = e^{0.8}$.

2) If we increase the temprature for let's say, temperature=10, then $P_{t=10}(w_1) = e^{0.09}$, $P_{t=10}(w_2) = e^{0.01}$, and therefore $\frac{P_{t=10}(w_1)}{P_{t=10}(w_2)} = e^{0.08}$.

We can see that when the temperature rises, the ratio of high-value probabilities to low-value probabilities decreases, flattening the distribution and making it more likely for the model to create alternative phrases.

3) Another way to look at it is - if we raise the temprature to infinity, all the probability vector (before taking the exponent) will converge to zeros, since $P_{t=\infty.}(w_i) = \frac{P(w_i)}{\infty.}$ = 0, after taking the exponent of that vector the probability vector will converge to a vector of ones. since "torch.multinomial" receives a "weights" tensor of "ones" all the words will have almost the same probability to be created.

- **taken from torch docs**

• taken from torch docs

In [ ]:

```
# Include the generated sequences and explanation in your PDF report.

headline = train_data[42].title
input_seq = torch.Tensor([vocab.stoi[w] for w in headline]).long().unsqueeze(0)
hidden = model.encode(input_seq)

for temp in [1 ,1.5, 2, 5]:
  print("########## Temprature {0} ###########\n".format(temp))
  for i in range(5):
    print("Sentence {0}: {1}".format(i, sample_sequence(model, hidden,temperature=temp))
)
```

```
########## Temprature 1 ###########

Sentence 0: ['zambian', 'president', 'swears', 'in', 'new', 'army', 'chief']
Sentence 1: ['zambian', 'president', 'swears', 'in', 'new', 'army', 'chief']
Sentence 2: ['zambian', 'president', 'swears', 'in', 'new', 'army', 'chief']
Sentence 3: ['zambian', 'president', 'swears', 'in', 'new', 'army', 'chief']
Sentence 4: ['zambian', 'president', 'swears', 'in', 'new', 'army', 'chief']
########## Temprature 1.5 ###########

Sentence 0: ['raytheon', 'felicity', 'in', 'new', 'new', 'army', 'chief', 'overshadows',
'minimum', 'dp', 'gymnastics', 'analysis', 'liquid', 'handed', 'chief', 'cheer', 'deere',
'correction', 'targeted', 'noise']
Sentence 1: ['zambian', 'president']
Sentence 2: ['zambian', 'president', 'swears', 'in', 'new', 'army', '4th', 'trees', 'fish
ing', 'exhibition', 'floats', 'headwinds', 'vp', 'source', 'electric', 'considered', '_nu
m_-u.s.', 'woos', 'scientists', 'restarting']
Sentence 3: ['caixa', 'testy', 'peril', 'zolgensma', 'contamination', 'gordon', 'modest',
'army', 'cos']
Sentence 4: ['zambian', 'president', 'regarding', 'lags', '_num_-fiat', 'light', 'loved',
'president', 'swears', 'hammer', 'new', 'estonian', 'juan', 'dc', 'keystone', 'new', 'arm
y', 'chief']
########## Temprature 2 ###########

Sentence 0: ['ride-hailing', 'zuckerberg', 'holy', 'jong', 'equatorial', 'ww2', 'tally',
'slovenia', 'reverses', 'scout24', 'sponsorship', 'hunt', 'french', 'bjp', '_num_-second'
, 'scheduled', 'disappoint', 'jails', 'repayment', 'product']
Sentence 1: ['injures', 'carlyle', 'oracle', 'flare', 'anti-abortion', 'behavior', 'spend
s', 'answers', 'memo', 'recommend', 'alaphilippe', 'tampering', 'harming', 'symptoms', 'o
bjections', '_num_-hudson']
Sentence 2: ['zambian', 'outlets', 'itf', 'stir', 'seizure', 'surgical', 'grey', 'italy',
'new', 'make', 'ana', 'loved', 'madagascar', 'rocks', 'cohen', 'strength', 'scene', 'ibra
himovic', 'local', 'digest']
Sentence 3: ['soyoil', 'capacity', 'lagerfeld', 'tata', 'february', 'evaluate', 'army', '
fortress', 'chief', 'rapper', 'mac', 'alleging', 'philippine', 'rebuffs', 'intelligence',
'impeachment', 'ortega', 'should', 'freight', 'finland']
Sentence 4: ['advises', 'philippines', 'kcna', 'glass', 'debt-c', 'independent', 'scenes'
, 'latam', '_num_-faa', 'whales', 'pactual', 'belgium', 'axel', 'minister', 'cvs', 'dip',
'forbes', 'milwaukee', 'eases', 'world']
########## Temprature 5 ###########

Sentence 0: ['gambia', '_num_-ryanair', 'violent', 'tourist', 'wta', 'thinking', 'tanzani
an', 'invites', 'hariri', 'art', 'harmful', 'dollar', 'egypt', 'search', 'spa', 'apache',
'deportations', 'totally', 'ralph', 'chase']
Sentence 1: ['steepest', 'totally', 'hear', 'sit', 'richest', 'level', 'execute', 'merger
s', 'collecting', 'grade', 'authority', 'canadian', 'lull', 'renewable', 'additions', 'fi
ve-week', 'warplanes', 'populist', 'reserves', 'embraces']
Sentence 2: ['excludes', 'swears', 'reactors', 'rafa', 'addressed', '_num_-american', 'ad
opt', 'bargaining', 'quotes', 'co2', 'flooded', 'finding', "p'nyang", 'trial', 'tasnim',
'exclusion', 'plummets', 'fresenius', 'plains', 'revamps']
Sentence 3: ['soaring', 'dozen', 'q1', 'documents', 'winners', 'co-head', 'discount', 'ma
cau', 'nutrition', 'bondholders', 'analyst', 'instruments', 'playing', 'value', 'jewelry'
, 'daniels', 'rock', 'woos', 'last-minute', 'brf']
Sentence 4: ['allegedly', 'gilead', 'divers', 'nfl', 'airbase', '_num_-wework', 'cba', 'e
ffort', '_num_-investor', 'suspicious', 'wirecard', 'kpmg', 'dims', 'lane', 'benefits', '
abb', 'expands', 'advanced', 'mercosur', 'challenged']
```

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.

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:

In [ ]:

```
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 r
andom 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
    #pdb.set_trace()
    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.

In [ ]:

```
# Report your values here. Make sure that you report the actual values,
# and not just the code used to get those values
headline = ' '.join(train_data[100].title)
print("original headline:'{}'".format(headline))
print("5 new headlines:")
for i in range(5):
  new_headline = tokenize_and_randomize(headline)
  new_headline = [vocab.itos[word_token] for word_token in new_headline]
  new_headline = ' '.join(word for word in new_headline)
  print(new_headline)
```

```
original headline:'<bos> brazil 's bolsonaro takes oath of office as president <eos>'
5 new headlines:
<bos> <bos> brazil 's takes bolsonaro <unk> of <pad> president <eos> <eos>
<bos> <bos> brazil bolsonaro 's takes <unk> office lankan as <eos> president <eos>
<bos> brazil <pad> <bos> bolsonaro takes <unk> as president <eos> <eos>
<bos> <bos> 's d-backs <unk> bolsonaro takes office of <pad> president <eos>
<bos> brazil takes 's gilead <unk> cop office as president <eos> <eos>
```

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*.

In [ ]:

```
def train_autoencoder(model, batch_size=64, learning_rate=0.001, num_epochs=10):
    # checkpoint_path = '/content/gdrive/Shareddrives/DNN_Elad_Raviv/ASS/ass4/CHECKPOINT'

    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    criterion = nn.CrossEntropyLoss()

    loss_list = []
    best_loss = 10
    for ep in range(num_epochs):

        # We will perform data augmentation by re-reading the input each time
        field = data.Field(sequential=True,
                                        tokenize=tokenize_and_randomize, # <-- data augment
ation
                                        include_lengths=True,
```

```
                                            batch_first=True,
                                            use_vocab=False, # <-- the tokenization function re
places 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 l
ength
        train_iter = data.BucketIterator(dataset,
                                                          batch_size=batch_size,
                                                          sort_key=lambda x: len(x.title), # t
o minimize padding
                                                          repeat=False)


        for it, ((xs, lengths), _) in enumerate(train_iter):
          xs = xs.to(device)
          #predict
          predicted_values = model(xs) # compute prediction logit

          predicted_values = predicted_values.view(-1, predicted_values.shape[2])
          #reshpe true values
          true_values = xs[:,1:].contiguous().view(-1,)

          loss = criterion(predicted_values, true_values)    # compute the total loss
          optimizer.zero_grad()        # zero the gradients before running the backward
pass. a clean up step for PyTorch
          loss.backward()              # Backward pass to compute the gradient of loss w
.r.t our learnable params.
          optimizer.step()             # Update params

          loss_list.append(loss)
          if (it+1) % 100 == 0:
              print("[Iter %d] Loss %f" % (it+1, float(loss)))

          # if loss<best_loss:
          #   torch.save(model.state_dict(), checkpoint_path.format(epoch))
          #   best_loss=loss


    return loss_list




        # 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
model = AutoEncoder(10000, 128, 128)
model.to(device)
loss_list = train_autoencoder(model, batch_size=50, learning_rate=0.001, num_epochs=20)
```

```
[Iter 100] Loss 4.636387
[Iter 200] Loss 4.648996
[Iter 300] Loss 4.228435
[Iter 400] Loss 3.812885
[Iter 500] Loss 3.843455
[Iter 600] Loss 4.113186
[Iter 700] Loss 4.314880
[Iter 800] Loss 3.437902
[Iter 900] Loss 3.603611
[Iter 1000] Loss 3.728658
[Iter 1100] Loss 3.965219
[Iter 1200] Loss 3.841307
[Iter 1300] Loss 3.626066
[Iter 1400] Loss 3.236610
```

```
[Iter 1500] Loss 3.890812
[Iter 1600] Loss 3.319546
[Iter 1700] Loss 3.477003
[Iter 1800] Loss 3.187819
[Iter 1900] Loss 3.225446
[Iter 2000] Loss 3.276027
[Iter 2100] Loss 3.774119
[Iter 2200] Loss 3.523660
[Iter 2300] Loss 3.361916
[Iter 2400] Loss 3.350587
[Iter 2500] Loss 3.312649
[Iter 2600] Loss 3.269928
[Iter 2700] Loss 3.864903
[Iter 2800] Loss 3.218024
[Iter 2900] Loss 3.217917
[Iter 3000] Loss 3.242973
[Iter 3100] Loss 3.064257
[Iter 3200] Loss 3.243323
[Iter 3300] Loss 2.791823
[Iter 3400] Loss 3.207217
[Iter 100] Loss 3.163588
[Iter 200] Loss 3.007941
[Iter 300] Loss 3.255992
[Iter 400] Loss 3.015993
[Iter 500] Loss 3.177118
[Iter 600] Loss 2.592628
[Iter 700] Loss 2.876420
[Iter 800] Loss 2.575381
[Iter 900] Loss 3.542933
[Iter 1000] Loss 2.789953
[Iter 1100] Loss 3.163635
[Iter 1200] Loss 3.331363
[Iter 1300] Loss 3.106544
[Iter 1400] Loss 2.827133
[Iter 1500] Loss 2.634616
[Iter 1600] Loss 2.763275
[Iter 1700] Loss 3.070427
[Iter 1800] Loss 2.328379
[Iter 1900] Loss 2.184802
[Iter 2000] Loss 2.941585
[Iter 2100] Loss 2.715528
[Iter 2200] Loss 2.751759
[Iter 2300] Loss 3.044740
[Iter 2400] Loss 2.926737
[Iter 2500] Loss 2.939512
[Iter 2600] Loss 2.710236
[Iter 2700] Loss 2.216113
[Iter 2800] Loss 2.733255
[Iter 2900] Loss 2.562204
[Iter 3000] Loss 2.246090
[Iter 3100] Loss 2.542848
[Iter 3200] Loss 2.884103
[Iter 3300] Loss 2.392378
[Iter 3400] Loss 2.963413
[Iter 100] Loss 2.836299
[Iter 200] Loss 2.489727
[Iter 300] Loss 2.607795
[Iter 400] Loss 2.424131
[Iter 500] Loss 2.569359
[Iter 600] Loss 2.426168
[Iter 700] Loss 2.571432
[Iter 800] Loss 2.371367
[Iter 900] Loss 2.610338
[Iter 1000] Loss 2.492638
[Iter 1100] Loss 2.697703
[Iter 1200] Loss 1.966297
[Iter 1300] Loss 2.250059
[Iter 1400] Loss 2.355923
[Iter 1500] Loss 2.133929
[Iter 1600] Loss 2.370968
[Iter 1700] Loss 1.942857
[Iter 1800] Loss 2.053846
```

```
[Iter 1900] Loss 2.309941
[Iter 2000] Loss 2.285791
[Iter 2100] Loss 2.200166
[Iter 2200] Loss 2.443583
[Iter 2300] Loss 2.350886
[Iter 2400] Loss 2.101915
[Iter 2500] Loss 2.233446
[Iter 2600] Loss 2.123340
[Iter 2700] Loss 1.831659
[Iter 2800] Loss 2.038608
[Iter 2900] Loss 1.762475
[Iter 3000] Loss 1.969777
[Iter 3100] Loss 2.212833
[Iter 3200] Loss 2.144027
[Iter 3300] Loss 1.965280
[Iter 3400] Loss 1.849781
[Iter 100] Loss 1.998197
[Iter 200] Loss 2.111289
[Iter 300] Loss 1.770476
[Iter 400] Loss 2.156694
[Iter 500] Loss 2.026703
[Iter 600] Loss 2.208259
[Iter 700] Loss 1.936871
[Iter 800] Loss 1.948255
[Iter 900] Loss 1.735105
[Iter 1000] Loss 1.755815
[Iter 1100] Loss 1.857264
[Iter 1200] Loss 2.118142
[Iter 1300] Loss 1.903619
[Iter 1400] Loss 1.970303
[Iter 1500] Loss 1.941853
[Iter 1600] Loss 1.682957
[Iter 1700] Loss 1.602628
[Iter 1800] Loss 2.039890
[Iter 1900] Loss 1.853565
[Iter 2000] Loss 1.893112
[Iter 2100] Loss 1.945660
[Iter 2200] Loss 1.877821
[Iter 2300] Loss 1.615252
[Iter 2400] Loss 1.760817
[Iter 2500] Loss 1.911680
[Iter 2600] Loss 1.623018
[Iter 2700] Loss 1.628854
[Iter 2800] Loss 1.828104
[Iter 2900] Loss 1.829880
[Iter 3000] Loss 1.780428
[Iter 3100] Loss 1.574929
[Iter 3200] Loss 1.743116
[Iter 3300] Loss 1.773092
[Iter 3400] Loss 1.692722
[Iter 100] Loss 1.521877
[Iter 200] Loss 1.420911
[Iter 300] Loss 1.626619
[Iter 400] Loss 1.592131
[Iter 500] Loss 2.015077
[Iter 600] Loss 1.416083
[Iter 700] Loss 1.509302
[Iter 800] Loss 1.504583
[Iter 900] Loss 1.429983
[Iter 1000] Loss 1.623634
[Iter 1100] Loss 1.480712
[Iter 1200] Loss 1.517681
[Iter 1300] Loss 1.412316
[Iter 1400] Loss 1.507338
[Iter 1500] Loss 1.471512
[Iter 1600] Loss 1.320921
[Iter 1700] Loss 1.653902
[Iter 1800] Loss 1.463387
[Iter 1900] Loss 1.238440
[Iter 2000] Loss 1.519903
[Iter 2100] Loss 1.583793
[Iter 2200] Loss 1.270728
```

```
[Iter 2300] Loss 1.459969
[Iter 2400] Loss 1.399130
[Iter 2500] Loss 1.401207
[Iter 2600] Loss 1.438249
[Iter 2700] Loss 1.442649
[Iter 2800] Loss 1.230053
[Iter 2900] Loss 1.217663
[Iter 3000] Loss 1.444978
[Iter 3100] Loss 1.571142
[Iter 3200] Loss 1.332415
[Iter 3300] Loss 1.293106
[Iter 3400] Loss 1.264016
[Iter 100] Loss 1.289551
[Iter 200] Loss 1.145734
[Iter 300] Loss 1.364020
[Iter 400] Loss 1.451563
[Iter 500] Loss 1.021705
[Iter 600] Loss 1.249901
[Iter 700] Loss 1.331186
[Iter 800] Loss 1.290263
[Iter 900] Loss 1.307682
[Iter 1000] Loss 1.201401
[Iter 1100] Loss 1.256517
[Iter 1200] Loss 1.147439
[Iter 1300] Loss 1.078518
[Iter 1400] Loss 1.197594
[Iter 1500] Loss 1.314813
[Iter 1600] Loss 1.177716
[Iter 1700] Loss 1.051986
[Iter 1800] Loss 1.101707
[Iter 1900] Loss 1.241575
[Iter 2000] Loss 1.110461
[Iter 2100] Loss 1.173523
[Iter 2200] Loss 1.336387
[Iter 2300] Loss 1.237881
[Iter 2400] Loss 1.168314
[Iter 2500] Loss 1.154563
[Iter 2600] Loss 1.086971
[Iter 2700] Loss 1.167454
[Iter 2800] Loss 1.157004
[Iter 2900] Loss 1.049607
[Iter 3000] Loss 1.301668
[Iter 3100] Loss 1.096782
[Iter 3200] Loss 1.037511
[Iter 3300] Loss 1.070730
[Iter 3400] Loss 1.032538
[Iter 100] Loss 1.182261
[Iter 200] Loss 0.953875
[Iter 300] Loss 1.015650
[Iter 400] Loss 1.103638
[Iter 500] Loss 1.038861
[Iter 600] Loss 0.983719
[Iter 700] Loss 1.013059
[Iter 800] Loss 1.193476
[Iter 900] Loss 1.014907
[Iter 1000] Loss 1.083748
[Iter 1100] Loss 0.993502
[Iter 1200] Loss 1.170475
[Iter 1300] Loss 1.199318
[Iter 1400] Loss 0.919360
[Iter 1500] Loss 0.849041
[Iter 1600] Loss 1.018070
[Iter 1700] Loss 0.955869
[Iter 1800] Loss 0.827274
[Iter 1900] Loss 0.880034
[Iter 2000] Loss 0.890935
[Iter 2100] Loss 0.958402
[Iter 2200] Loss 0.864802
[Iter 2300] Loss 1.004321
[Iter 2400] Loss 0.898732
[Iter 2500] Loss 1.031767
[Iter 2600] Loss 0.919528
```

```
[Iter 2700] Loss 0.994166
[Iter 2800] Loss 0.814996
[Iter 2900] Loss 0.863968
[Iter 3000] Loss 0.828466
[Iter 3100] Loss 0.892772
[Iter 3200] Loss 1.032387
[Iter 3300] Loss 0.929513
[Iter 3400] Loss 0.901712
[Iter 100] Loss 0.887218
[Iter 200] Loss 0.857640
[Iter 300] Loss 0.877516
[Iter 400] Loss 0.895523
[Iter 500] Loss 0.781069
[Iter 600] Loss 0.951032
[Iter 700] Loss 0.967578
[Iter 800] Loss 0.908218
[Iter 900] Loss 0.824521
[Iter 1000] Loss 0.982480
[Iter 1100] Loss 0.871014
[Iter 1200] Loss 0.604722
[Iter 1300] Loss 0.862815
[Iter 1400] Loss 0.829493
[Iter 1500] Loss 1.036782
[Iter 1600] Loss 0.810921
[Iter 1700] Loss 0.927137
[Iter 1800] Loss 0.734447
[Iter 1900] Loss 0.805085
[Iter 2000] Loss 0.868567
[Iter 2100] Loss 0.852231
[Iter 2200] Loss 0.670552
[Iter 2300] Loss 0.727109
[Iter 2400] Loss 0.677050
[Iter 2500] Loss 0.788432
[Iter 2600] Loss 0.690561
[Iter 2700] Loss 0.790557
[Iter 2800] Loss 0.873471
[Iter 2900] Loss 0.798178
[Iter 3000] Loss 0.865610
[Iter 3100] Loss 0.673587
[Iter 3200] Loss 0.840726
[Iter 3300] Loss 0.731250
[Iter 3400] Loss 0.792386
[Iter 100] Loss 0.584045
[Iter 200] Loss 0.810132
[Iter 300] Loss 0.807514
[Iter 400] Loss 0.869575
[Iter 500] Loss 0.754118
[Iter 600] Loss 0.869949
[Iter 700] Loss 0.736172
[Iter 800] Loss 0.741866
[Iter 900] Loss 0.644932
[Iter 1000] Loss 0.744370
[Iter 1100] Loss 0.801423
[Iter 1200] Loss 0.737664
[Iter 1300] Loss 0.594332
[Iter 1400] Loss 0.687725
[Iter 1500] Loss 0.624924
[Iter 1600] Loss 0.762298
[Iter 1700] Loss 0.725979
[Iter 1800] Loss 0.610236
[Iter 1900] Loss 0.686258
[Iter 2000] Loss 0.713148
[Iter 2100] Loss 0.696989
[Iter 2200] Loss 0.708887
[Iter 2300] Loss 0.713805
[Iter 2400] Loss 0.617105
[Iter 2500] Loss 0.677618
[Iter 2600] Loss 0.789608
[Iter 2700] Loss 0.701641
[Iter 2800] Loss 0.792417
[Iter 2900] Loss 0.590761
[Iter 3000] Loss 0.758980
```

```
[Iter 3100] Loss 0.675089
[Iter 3200] Loss 0.724601
[Iter 3300] Loss 0.767793
[Iter 3400] Loss 0.715245
[Iter 100] Loss 0.635940
[Iter 200] Loss 0.475057
[Iter 300] Loss 0.642701
[Iter 400] Loss 0.614098
[Iter 500] Loss 0.634458
[Iter 600] Loss 0.607682
[Iter 700] Loss 0.652330
[Iter 800] Loss 0.581221
[Iter 900] Loss 0.686749
[Iter 1000] Loss 0.662677
[Iter 1100] Loss 0.501487
[Iter 1200] Loss 0.772035
[Iter 1300] Loss 0.708294
[Iter 1400] Loss 0.539991
[Iter 1500] Loss 0.653628
[Iter 1600] Loss 0.714477
[Iter 1700] Loss 0.726340
[Iter 1800] Loss 0.709442
[Iter 1900] Loss 0.657543
[Iter 2000] Loss 0.694610
[Iter 2100] Loss 0.523886
[Iter 2200] Loss 0.558795
[Iter 2300] Loss 0.600882
[Iter 2400] Loss 0.567693
[Iter 2500] Loss 0.595576
[Iter 2600] Loss 0.453522
[Iter 2700] Loss 0.567814
[Iter 2800] Loss 0.620153
[Iter 2900] Loss 0.573498
[Iter 3000] Loss 0.630183
[Iter 3100] Loss 0.563283
[Iter 3200] Loss 0.560781
[Iter 3300] Loss 0.649373
[Iter 3400] Loss 0.586630
[Iter 100] Loss 0.630634
[Iter 200] Loss 0.479319
[Iter 300] Loss 0.531463
[Iter 400] Loss 0.463272
[Iter 500] Loss 0.478373
[Iter 600] Loss 0.524915
[Iter 700] Loss 0.674834
[Iter 800] Loss 0.527226
[Iter 900] Loss 0.595291
[Iter 1000] Loss 0.614062
[Iter 1100] Loss 0.530396
[Iter 1200] Loss 0.620664
[Iter 1300] Loss 0.503340
[Iter 1400] Loss 0.531881
[Iter 1500] Loss 0.603877
[Iter 1600] Loss 0.476055
[Iter 1700] Loss 0.515766
[Iter 1800] Loss 0.518348
[Iter 1900] Loss 0.461113
[Iter 2000] Loss 0.551499
[Iter 2100] Loss 0.586384
[Iter 2200] Loss 0.480926
[Iter 2300] Loss 0.655299
[Iter 2400] Loss 0.485212
[Iter 2500] Loss 0.571959
[Iter 2600] Loss 0.583852
[Iter 2700] Loss 0.602839
[Iter 2800] Loss 0.577274
[Iter 2900] Loss 0.574115
[Iter 3000] Loss 0.485293
[Iter 3100] Loss 0.533442
[Iter 3200] Loss 0.537956
[Iter 3300] Loss 0.609019
[Iter 3400] Loss 0.598650
```

```
[Iter 100] Loss 0.522572
[Iter 200] Loss 0.573690
[Iter 300] Loss 0.619983
[Iter 400] Loss 0.457630
[Iter 500] Loss 0.505464
[Iter 600] Loss 0.572269
[Iter 700] Loss 0.456228
[Iter 800] Loss 0.459767
[Iter 900] Loss 0.562062
[Iter 1000] Loss 0.453560
[Iter 1100] Loss 0.427063
[Iter 1200] Loss 0.564882
[Iter 1300] Loss 0.540323
[Iter 1400] Loss 0.553557
[Iter 1500] Loss 0.571848
[Iter 1600] Loss 0.452459
[Iter 1700] Loss 0.535748
[Iter 1800] Loss 0.528098
[Iter 1900] Loss 0.582627
[Iter 2000] Loss 0.474899
[Iter 2100] Loss 0.494199
[Iter 2200] Loss 0.394233
[Iter 2300] Loss 0.519675
[Iter 2400] Loss 0.398307
[Iter 2500] Loss 0.413843
[Iter 2600] Loss 0.486023
[Iter 2700] Loss 0.563806
[Iter 2800] Loss 0.550138
[Iter 2900] Loss 0.461394
[Iter 3000] Loss 0.554656
[Iter 3100] Loss 0.437095
[Iter 3200] Loss 0.574941
[Iter 3300] Loss 0.507423
[Iter 3400] Loss 0.573129
[Iter 100] Loss 0.479879
[Iter 200] Loss 0.439221
[Iter 300] Loss 0.403734
[Iter 400] Loss 0.519758
[Iter 500] Loss 0.520244
[Iter 600] Loss 0.360386
[Iter 700] Loss 0.480759
[Iter 800] Loss 0.447830
[Iter 900] Loss 0.401040
[Iter 1000] Loss 0.402322
[Iter 1100] Loss 0.338496
[Iter 1200] Loss 0.456262
[Iter 1300] Loss 0.417474
[Iter 1400] Loss 0.345582
[Iter 1500] Loss 0.354165
[Iter 1600] Loss 0.464916
[Iter 1700] Loss 0.475744
[Iter 1800] Loss 0.426818
[Iter 1900] Loss 0.458867
[Iter 2000] Loss 0.424972
[Iter 2100] Loss 0.459031
[Iter 2200] Loss 0.420408
[Iter 2300] Loss 0.443293
[Iter 2400] Loss 0.509041
[Iter 2500] Loss 0.519368
[Iter 2600] Loss 0.425077
[Iter 2700] Loss 0.469215
[Iter 2800] Loss 0.386150
[Iter 2900] Loss 0.497008
[Iter 3000] Loss 0.445639
[Iter 3100] Loss 0.367656
[Iter 3200] Loss 0.503287
[Iter 3300] Loss 0.394662
[Iter 3400] Loss 0.452629
[Iter 100] Loss 0.501186
[Iter 200] Loss 0.294163
[Iter 300] Loss 0.349128
[Iter 400] Loss 0.350963
```

```
[Iter 500] Loss 0.487872
[Iter 600] Loss 0.420526
[Iter 700] Loss 0.416181
[Iter 800] Loss 0.319050
[Iter 900] Loss 0.459276
[Iter 1000] Loss 0.302690
[Iter 1100] Loss 0.431717
[Iter 1200] Loss 0.427766
[Iter 1300] Loss 0.415465
[Iter 1400] Loss 0.392144
[Iter 1500] Loss 0.411111
[Iter 1600] Loss 0.425798
[Iter 1700] Loss 0.405120
[Iter 1800] Loss 0.501959
[Iter 1900] Loss 0.378231
[Iter 2000] Loss 0.400561
[Iter 2100] Loss 0.474113
[Iter 2200] Loss 0.438781
[Iter 2300] Loss 0.469161
[Iter 2400] Loss 0.393023
[Iter 2500] Loss 0.256721
[Iter 2600] Loss 0.474860
[Iter 2700] Loss 0.402002
[Iter 2800] Loss 0.338901
[Iter 2900] Loss 0.269408
[Iter 3000] Loss 0.435020
[Iter 3100] Loss 0.356142
[Iter 3200] Loss 0.492911
[Iter 3300] Loss 0.398441
[Iter 3400] Loss 0.333939
[Iter 100] Loss 0.574784
[Iter 200] Loss 0.393868
[Iter 300] Loss 0.365664
[Iter 400] Loss 0.406015
[Iter 500] Loss 0.341264
[Iter 600] Loss 0.402566
[Iter 700] Loss 0.396166
[Iter 800] Loss 0.360457
[Iter 900] Loss 0.342772
[Iter 1000] Loss 0.328351
[Iter 1100] Loss 0.526849
[Iter 1200] Loss 0.331125
[Iter 1300] Loss 0.460499
[Iter 1400] Loss 0.591168
[Iter 1500] Loss 0.423472
[Iter 1600] Loss 0.284963
[Iter 1700] Loss 0.462131
[Iter 1800] Loss 0.410801
[Iter 1900] Loss 0.388295
[Iter 2000] Loss 0.419218
[Iter 2100] Loss 0.422678
[Iter 2200] Loss 0.354495
[Iter 2300] Loss 0.338120
[Iter 2400] Loss 0.414052
[Iter 2500] Loss 0.361904
[Iter 2600] Loss 0.420118
[Iter 2700] Loss 0.350048
[Iter 2800] Loss 0.359559
[Iter 2900] Loss 0.365727
[Iter 3000] Loss 0.393906
[Iter 3100] Loss 0.294573
[Iter 3200] Loss 0.285792
[Iter 3300] Loss 0.460788
[Iter 3400] Loss 0.340824
[Iter 100] Loss 0.329175
[Iter 200] Loss 0.424722
[Iter 300] Loss 0.444627
[Iter 400] Loss 0.365608
[Iter 500] Loss 0.462945
[Iter 600] Loss 0.362346
[Iter 700] Loss 0.322943
[Iter 800] Loss 0.430753
```

```
[Iter 900] Loss 0.319512
[Iter 1000] Loss 0.302285
[Iter 1100] Loss 0.338729
[Iter 1200] Loss 0.345691
[Iter 1300] Loss 0.265207
[Iter 1400] Loss 0.255111
[Iter 1500] Loss 0.410814
[Iter 1600] Loss 0.343819
[Iter 1700] Loss 0.337871
[Iter 1800] Loss 0.280479
[Iter 1900] Loss 0.349033
[Iter 2000] Loss 0.369413
[Iter 2100] Loss 0.289604
[Iter 2200] Loss 0.207216
[Iter 2300] Loss 0.360594
[Iter 2400] Loss 0.269724
[Iter 2500] Loss 0.307435
[Iter 2600] Loss 0.266953
[Iter 2700] Loss 0.377019
[Iter 2800] Loss 0.352726
[Iter 2900] Loss 0.347650
[Iter 3000] Loss 0.286502
[Iter 3100] Loss 0.382376
[Iter 3200] Loss 0.381238
[Iter 3300] Loss 0.272142
[Iter 3400] Loss 0.264394
[Iter 100] Loss 0.384674
[Iter 200] Loss 0.320862
[Iter 300] Loss 0.276194
[Iter 400] Loss 0.366990
[Iter 500] Loss 0.319475
[Iter 600] Loss 0.259716
[Iter 700] Loss 0.312622
[Iter 800] Loss 0.277763
[Iter 900] Loss 0.358976
[Iter 1000] Loss 0.362227
[Iter 1100] Loss 0.405670
[Iter 1200] Loss 0.409575
[Iter 1300] Loss 0.455195
[Iter 1400] Loss 0.449960
[Iter 1500] Loss 0.368670
[Iter 1600] Loss 0.335638
[Iter 1700] Loss 0.418333
[Iter 1800] Loss 0.330533
[Iter 1900] Loss 0.329129
[Iter 2000] Loss 0.301446
[Iter 2100] Loss 0.280265
[Iter 2200] Loss 0.286561
[Iter 2300] Loss 0.285877
[Iter 2400] Loss 0.377051
[Iter 2500] Loss 0.287223
[Iter 2600] Loss 0.304492
[Iter 2700] Loss 0.342167
[Iter 2800] Loss 0.359156
[Iter 2900] Loss 0.285813
[Iter 3000] Loss 0.371990
[Iter 3100] Loss 0.374839
[Iter 3200] Loss 0.357801
[Iter 3300] Loss 0.347463
[Iter 3400] Loss 0.336712
[Iter 100] Loss 0.333057
[Iter 200] Loss 0.335888
[Iter 300] Loss 0.379738
[Iter 400] Loss 0.290631
[Iter 500] Loss 0.260871
[Iter 600] Loss 0.292809
[Iter 700] Loss 0.299010
[Iter 800] Loss 0.312885
[Iter 900] Loss 0.276648
[Iter 1000] Loss 0.250648
[Iter 1100] Loss 0.311988
[Iter 1200] Loss 0.411361
```

```
[Iter 1300] Loss 0.313979
[Iter 1400] Loss 0.260089
[Iter 1500] Loss 0.407342
[Iter 1600] Loss 0.457596
[Iter 1700] Loss 0.303332
[Iter 1800] Loss 0.323705
[Iter 1900] Loss 0.374448
[Iter 2000] Loss 0.269204
[Iter 2100] Loss 0.359844
[Iter 2200] Loss 0.210134
[Iter 2300] Loss 0.314590
[Iter 2400] Loss 0.262064
[Iter 2500] Loss 0.218146
[Iter 2600] Loss 0.400276
[Iter 2700] Loss 0.312998
[Iter 2800] Loss 0.251755
[Iter 2900] Loss 0.309394
[Iter 3000] Loss 0.277980
[Iter 3100] Loss 0.328942
[Iter 3200] Loss 0.260958
[Iter 3300] Loss 0.271110
[Iter 3400] Loss 0.391679
[Iter 100] Loss 0.313362
[Iter 200] Loss 0.356550
[Iter 300] Loss 0.248244
[Iter 400] Loss 0.273143
[Iter 500] Loss 0.304885
[Iter 600] Loss 0.287401
[Iter 700] Loss 0.282890
[Iter 800] Loss 0.283505
[Iter 900] Loss 0.276710
[Iter 1000] Loss 0.246545
[Iter 1100] Loss 0.266470
[Iter 1200] Loss 0.360947
[Iter 1300] Loss 0.268934
[Iter 1400] Loss 0.327780
[Iter 1500] Loss 0.388145
[Iter 1600] Loss 0.278673
[Iter 1700] Loss 0.348292
[Iter 1800] Loss 0.278102
[Iter 1900] Loss 0.246008
[Iter 2000] Loss 0.433511
[Iter 2100] Loss 0.307603
[Iter 2200] Loss 0.314105
[Iter 2300] Loss 0.392475
[Iter 2400] Loss 0.259276
[Iter 2500] Loss 0.273814
[Iter 2600] Loss 0.270254
[Iter 2700] Loss 0.312436
[Iter 2800] Loss 0.337106
[Iter 2900] Loss 0.239536
[Iter 3000] Loss 0.224850
[Iter 3100] Loss 0.239985
[Iter 3200] Loss 0.263577
[Iter 3300] Loss 0.228140
[Iter 3400] Loss 0.263424
[Iter 100] Loss 0.330287
[Iter 200] Loss 0.282525
[Iter 300] Loss 0.325662
[Iter 400] Loss 0.287158
[Iter 500] Loss 0.213613
[Iter 600] Loss 0.278635
[Iter 700] Loss 0.236164
[Iter 800] Loss 0.364323
[Iter 900] Loss 0.342717
[Iter 1000] Loss 0.254244
[Iter 1100] Loss 0.278865
[Iter 1200] Loss 0.251429
[Iter 1300] Loss 0.315473
[Iter 1400] Loss 0.257813
[Iter 1500] Loss 0.337532
[Iter 1600] Loss 0.239574
```

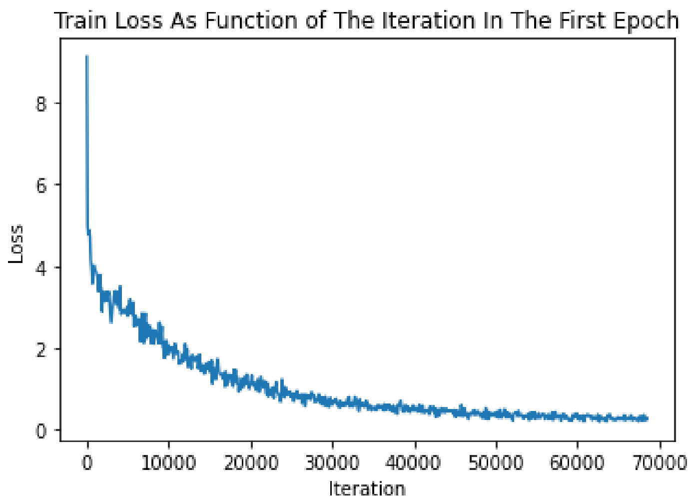
```
[Iter 1700] Loss 0.211271
[Iter 1800] Loss 0.288354
[Iter 1900] Loss 0.263468
[Iter 2000] Loss 0.341426
[Iter 2100] Loss 0.288879
[Iter 2200] Loss 0.304698
[Iter 2300] Loss 0.321635
[Iter 2400] Loss 0.353950
[Iter 2500] Loss 0.242844
[Iter 2600] Loss 0.281894
[Iter 2700] Loss 0.247489
[Iter 2800] Loss 0.263417
[Iter 2900] Loss 0.324930
[Iter 3000] Loss 0.321539
[Iter 3100] Loss 0.223338
[Iter 3200] Loss 0.241496
[Iter 3300] Loss 0.251400
[Iter 3400] Loss 0.250923
```

In [ ]:

```
plt.plot(range(1,len(loss_list), 100), loss_list[1::100])
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Train Loss As Function of The Iteration In The First Epoch")
```

Out[ ]:

```
Text(0.5, 1.0, 'Train Loss As Function of The Iteration In The First Epoch')
```



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.

In [ ]:

```
model = AutoEncoder(10000, 128, 128)
checkpoint_path = '/content/gdrive/Shareddrives/DNN_Elad_Raviv/ASS/ass4/AE_RNN_model.pk'
# Update me
model.load_state_dict(torch.load(checkpoint_path))
```

Out[ ]:

```
<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.

In [ ]:

```
# 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.
hidden = model.encode(input_seq)

for temp in [0.7 ,0.9, 1.5]:
  print("########## Temprature {0} ###########\n".format(temp))
  for i in range(5):
    print("Sentence {0}: {1}".format(i, sample_sequence(model, hidden, temperature=temp)
))

# ...
```

```
########## Temprature 0.7 ###########

Sentence 0: ['wall', 'street', 'rises', ',', 'limps', 'heist', ',', 'chances', 'blast', '
support', 'after', 'commission', '-minister']
Sentence 1: ['wall', 'street', 'rises', ',', 'limps', 'die', 'win', 'at', 'of', 'sciences
', '<pad>', 'presidential', 'amid']
Sentence 2: ['wall', 'street', 'rises', ',', 'limps', 'die', 'win', "'s", 'employees', '<
pad>', 'one-time', 'after', 'wto']
Sentence 3: ['wall', 'street', 'rises', ',', 'limps', 'die', 'win', 'at', 'of', 'sciences
', ':', 'libya', 'power']
Sentence 4: ['wall', 'street', 'rises', ',', 'limps', 'open', 'sentence', 'second', ',',
'message', 'next', 'abu', 'bites']
########## Temprature 0.9 ###########

Sentence 0: ['wall', 'street', 'rises', ',', 'limps', 'die', 'win', "'s", 'point', 'only'
, 'for', 'friday', 'month']
Sentence 1: ['wall', 'street', 'rises', ',', 'limps', 'and', 'turn', 'coaching', 'on', '<
pad>', 'against', 'kenya', 'showdown']
Sentence 2: ['wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish', 'line',
'of', 'a', 'turbulent', 'year']
Sentence 3: ['wall', 'street', 'rises', ',', 'limps', 'australia', 'young', ',', 'starts'
, '<pad>', 'on', 'war', 'target']
Sentence 4: ['wall', 'street', 'rises', ',', 'limps', 'die', 'win', 'at', 'of', 'sciences
', 'europe', 'for', 'protest']
########## Temprature 1.5 ###########

Sentence 0: ['wall', 'premier', 'handling', 'knock', 'amazon', 'form', 'editor', 'food',
'to', 'election', 'conditional', 'after', 'oversubscribed']
Sentence 1: ['wall', 'street', 'rises', ',', 'scales', 'gic', 'relief', 'to', 'illnesses'
, 'court', 'norway', 'despite', 'keeps']
Sentence 2: ['wall', 'street', 'rises', ',', 'limps', 'australia', 'protection', 'of', '<
unk>', 'drag', ':', 'big', 'havoc']
Sentence 3: ['wall', 'street', 'rises', ',', 'hut', 'tickets', 'with', 'takeover', 'leade
r', 'boat', 'investments', ':', 'spur']
Sentence 4: ['wall', 'street', 'rises', ',', 'limps', 'die', 'win', 'at', 'adds', '<pad>'
, 'libya', 'series', 'yemen']
```

Write your explanation here: **As satated before, by changing the temperature we can control the output-vector's variance and as an extension the probability of words to be drawn out of the transformed distribution. If we want the decoder to perform as a *generator* of headlines, we should not use too low of a temperature as it withers the model, and forces it to generate very similar headlines. we need to be setting the tempature high enough so that the model will be able to generate a variety of words, and not generates words that are very similar to the ones which were inserted initialy .**

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:

In [ ]:

```
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)
```

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.

In [ ]:

```
# Write your code here
# Show that your resulting PyTorch tensor has shape `[19046, 128]`
embeddings = []
for data_point in valid_data:
  headline = data_point.title
  input_seq = torch.Tensor([vocab.stoi[w] for w in headline]).unsqueeze(0).long()
  embeddings.append(model.encode(input_seq))

embeddings = [embedding.squeeze() for embedding in embeddings]
embeddings = torch.stack(embeddings)
print(embeddings.size(), 'is the shape of the embedding')
```

```
torch.Size([19046, 128]) is the shape of the embedding
```

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)

In [ ]:

```
headline_sim = similarities[13, :]
headline_sim[13] = -np.inf
closest_headlines_idxes = headline_sim.argsort()[-5:][::-1]

print("The 5 closest headlines to '{}':".format(' '.join(valid_data[13].title)))
for i, cls_headline_idx in enumerate(closest_headlines_idxes):
  print(str(i+1)+". ", "Similarity {:.3f}  :".format(headline_sim[cls_headline_idx]), "'"
"+' '.join(valid_data[cls_headline_idx].title)+"'")
```

```
The 5 closest headlines to '<bos> asia takes heart from new year gains in u.s. stock futu
res <eos>':
1.  Similarity 0.931  : '<bos> italy 's salvini loses aura of invincibility in emilia set
back <eos>'
2.  Similarity 0.931  : '<bos> saudi , russia look to seal deeper output cuts with oil pr
oducers <eos>'
3.  Similarity 0.930  : '<bos> eu orders quarantine for staff who traveled to northern it
aly <eos>'
4.  Similarity 0.929  : '<bos> update _num_-italy 's prime minister says new government w
ill bicker less <eos>'
5.  Similarity 0.928  : '<bos> portugal 's moura pays tribute to cod fishermen at milan f
ashion close <eos>'
```

Part (c) -- 4%

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

In [ ]:

```
headline_sim = similarities[12, :]
headline_sim[12] = -np.inf
closest_headlines_idxes = headline_sim.argsort()[-5:][::-1]
```

```
print("The 5 closest headlines to '{}':".format(' '.join(valid_data[12].title)))
for i, cls_headline_idx in enumerate(closest_headlines_idxes):
  print(str(i+1)+". ", "Similarity {:.3f}  :".format(headline_sim[cls_headline_idx]), "'"+' '.join(valid_data[cls_headline_idx].title)+"'")
```

```
The 5 closest headlines to '<bos> south korea 's hyundai target _num_ global sales of _num_ million vehicles <eos>':
1.  Similarity 0.952  : '<bos> south africa 's manufacturing up _num_ % y/y in april , highest in _num_ years <eos>'
2.  Similarity 0.951  : '<bos> south africa 's gross domestic spending up _num_ % in second quarter <eos>'
3.  Similarity 0.948  : '<bos> south africa 's retail sales up _num_ % year/year in october <eos>'
4.  Similarity 0.945  : '<bos> online and discounters to drive _num_ % growth in uk grocery by _num_ <eos>'
5.  Similarity 0.945  : '<bos> update _num_-egypt signs energy accords at conference in new capital <eos>'
```

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:

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

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).

In [ ]:

```
# Write your code here. Include your generated sequences.
head0 = valid_data[200].title
input_seq = torch.Tensor([vocab.stoi[w] for w in head0]).unsqueeze(0).long()
e0 = model.encode(input_seq)
print("e0:", ' '.join(head0))

head4 = valid_data[300].title
input_seq = torch.Tensor([vocab.stoi[w] for w in head4]).unsqueeze(0).long()
e4 = model.encode(input_seq)
print("e4:", ' '.join(head4))

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

temp = 1.6
print("Temperature:",temp)
ex = ["e1", "e2", "e3"]

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

```
e0: <bos> new york weatherman fired over racial slur in forecast <eos>
e4: <bos> honda to shut uk production for six days due to brexit logistics <eos>
Temperature: 1.6
e1 decodings:
['<unk>', 'u.s.', 'philadelphia', 'set', 'former', 'infected', 'explore', 'profit', ':']
['abused', 'canada', 'nigeria', 'washington', 'vehicles', ',', 'four', 'supreme', 'annual
']
['new', 'genetic', 'india', 'channels', 'pleads', 'with', 'employees', '<pad>', 'results'
```

```
['new', 'genetic', 'india', 'channels', 'pleads', 'with', 'employees', '<pad>', 'results'
, '-sources']
['dubai', 'charged', '<unk>', 'foreign', 'general', 'lack', 'since', 'ups', 'cuts']
['corbyn', 'tiffany', 'business', 'an', 'military', 'file', 'chemical', '<unk>', 'christm
as']
e2 decodings:
['wework', 'may', "'", 'guilt', 'u.s.', 'funds', 'wound', "'s", 'reuters', 'helm']
['wpp', 'bank', 'pm', 'patients', 'one', 'by', 'outraged', 'after', 'department', 'survey
']
['score', 'airlines', 'among', 'japan', 'knights', 'coronavirus', 'northern', 'trial', 't
itle', 'december']
['indicted', 'air', 'prepares', 'sale', 'forces', 'marks', 'ports', ':', 'maker', '<pad>'
]
['atp', 'preferred', "'", 'mexico', 'domestic', 'to', 'barcelona', 'embassy', 'calendar',
'risk']
e3 decodings:
['honda', 'china', 'to', 'dutch', 'halt', 'for', 'disorder', 'from', 're-elected', 'to',
'stable']
['honda', 'to', 'u.n.', 'amazon', 'funding', 'in', 'operate', '<pad>', ';', 'sign', 'cull
']
['honda', 'to', 'energy', 'image', 'broadband', 'in', 'access', 'after', 'tourism', 'ceme
x', 'britain']
['_num_-abbvie', 'proposes', 'apple', 'help', 'refinery', 'related', 'in', ';', '<pad>',
'blocking', 'results']
['honda', 'to', 'indonesia', 'plans', 'over', 'sanders', 'cut', 'due', 'for', 'rapper', '
cpi']
```