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 [5]:

```python
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
```

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 [6]:

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

train_path = '/content/gdrive/My Drive/Colab Notebooks/reuters_train.txt' # Update me
valid_path = '/content/gdrive/My Drive/Colab Notebooks/reuters_valid.txt' # Update 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 [7]:

```
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)
```

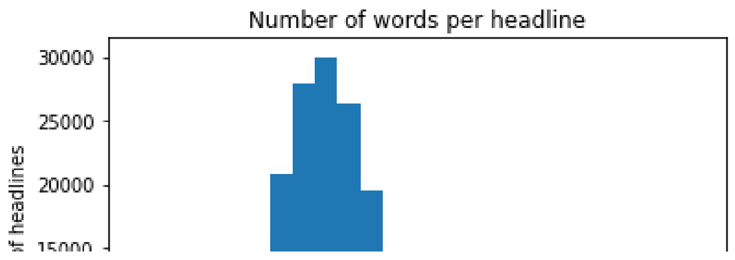
Part (a) -- 5%

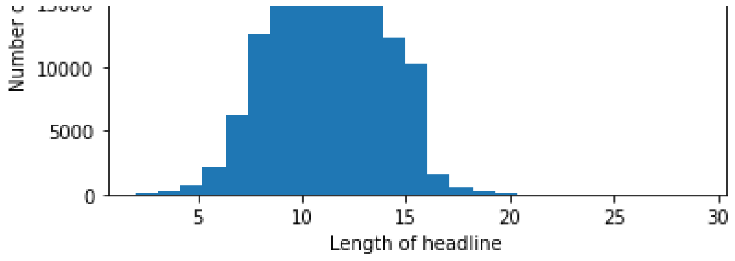
Draw histograms of the number of words per headline in our training set. Excluding the `<bos>` and `<eos>` tags in your computation. Explain why we would be interested in such histograms.

In [8]:

```
# Include your histogram and your written explanations
length_seq = []
for example in train_data:
    length_seq.append(len(example.title)-2)
plt.hist(length_seq,bins =25)
plt.xlabel('Length of headline')
plt.ylabel('Number of headlines')
plt.title('Number of words per headline')

# Here is an example of how to plot a histogram in matplotlib:
# plt.hist(np.random.normal(0, 1, 40), bins=20)

# Here are some sample code that uses the train_data object:
print(train_data[0].title)
for example in train_data:
    print(example.title)
    break
```

```
['<bos>', 'dems', 'move', 'to', 'end', 'shutdown', ',', 'without', 'wall', 'money', '<eos
>']
['<bos>', 'dems', 'move', 'to', 'end', 'shutdown', ',', 'without', 'wall', 'money', '<eos
>']
```





Explanation:

We would be interested in such histograms because our model is seq to seq based on RNN, which means that our predictions are made based on the previous words. So, Knowing the length of the headlines will help us understand how far in the past we need to look, and how much padding we need to do, according to the variety we see in the histogram. We can see that most of the headlines are from the same length so the variance is not so high, which means that we won't be needing a lot of padding.

Part (b) -- 5%

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

In [9]:

```
# 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
import collections
def flatten_list(list1):
  flat_list = []
  for element in list1:
      if type(element) is list:
          # If the element is of type list, iterate through the sublist
          for item in element:
              flat_list.append(item)
      else:
          flat_list.append(element)
  return flat_list

flat_train = flatten_list(train_data.title)
common_words = collections.Counter(flat_train)
print('The number of distinct words appear in the training data is',len(common_words)-2)
```

```
The number of distinct words appear in the training data is 51298
```

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 [10]:

```
# Report your values here. Make sure that you report the actual values,
# and not just the code used to get those values
value = common_words.values()
once = 0
twice = 0
for item in value:
  if item == 1:
    once +=1
  elif item == 2:
    twice +=1

print('The number of words appear exactly once in the training set is',once)
print('The number of words appear exactly twice in the training set is',twice)
```

```
The number of words appear exactly once in the training set is 19854
The number of words appear exactly twice in the training set is 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 [11]:

```python
# Report your values here. Make sure that you report the actual values,
# and not just the code used to get those values
model_top = common_words.most_common()
sum_top = sum(j for i, j in model_top[2:9997])
sum_all = sum(sorted(list(value), reverse=True)[2:])
print('The percentage of total words count that will be supported is',100*sum_top/sum_all
,'%')
sum_unk = sum(j for i, j in model_top[9997:])
print('The percentage of total words count that will be set to <unk> tag is',100*sum_unk/
sum_all,'%')
```

```
The percentage of total words count that will be supported is 93.97857393100142 %
The percentage of total words count that will be set to <unk> tag is 6.02142606899858 %
```

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 [12]:

```python
# 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>
```

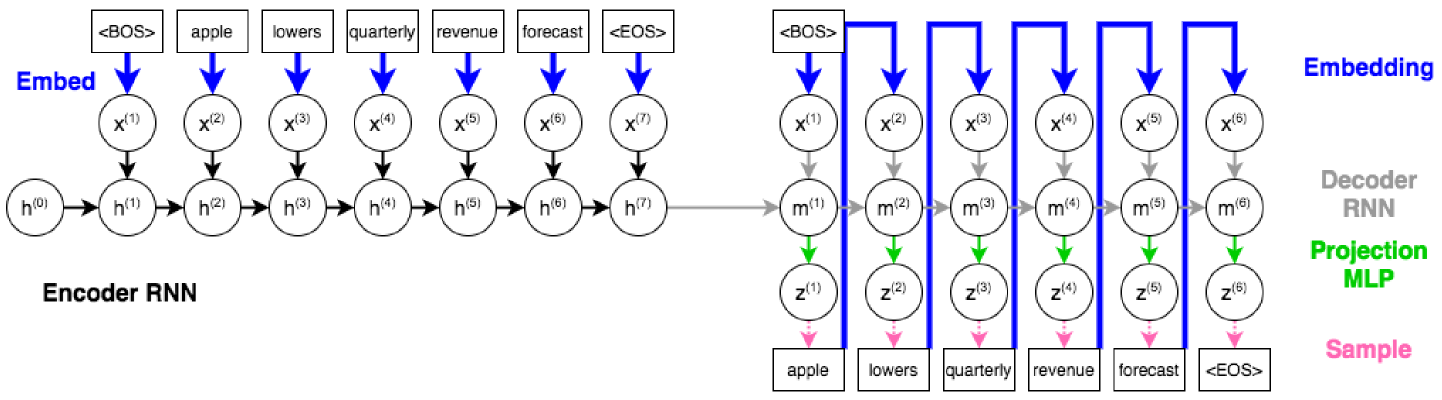
Question 2. Text Autoencoder (40%)

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:

Here is a diagram showing our desired architecture:



<BOS>
apple
lowers
quarterly
revenue
forecast
<EOS>
Embed
$x^{(1)}$ $x^{(2)}$ $x^{(3)}$ $x^{(4)}$ $x^{(5)}$ $x^{(6)}$ $x^{(7)}$
$h^{(0)}$ $h^{(1)}$ $h^{(2)}$ $h^{(3)}$ $h^{(4)}$ $h^{(5)}$ $h^{(6)}$ $h^{(7)}$
Encoder RNN
<BOS>
$x^{(1)}$ $x^{(2)}$ $x^{(3)}$ $x^{(4)}$ $x^{(5)}$ $x^{(6)}$
$m^{(1)}$ $m^{(2)}$ $m^{(3)}$ $m^{(4)}$ $m^{(5)}$ $m^{(6)}$
$z^{(1)}$ $z^{(2)}$ $z^{(3)}$ $z^{(4)}$ $z^{(5)}$ $z^{(6)}$
apple
lowers
quarterly
revenue
forecast
<EOS>
Embedding
Decoder RNN
Projection MLP
Sample

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.

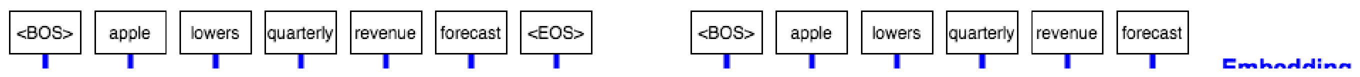
We can repeat this process until we see an `<eos>` token generated, or until the generated sequence becomes too long.

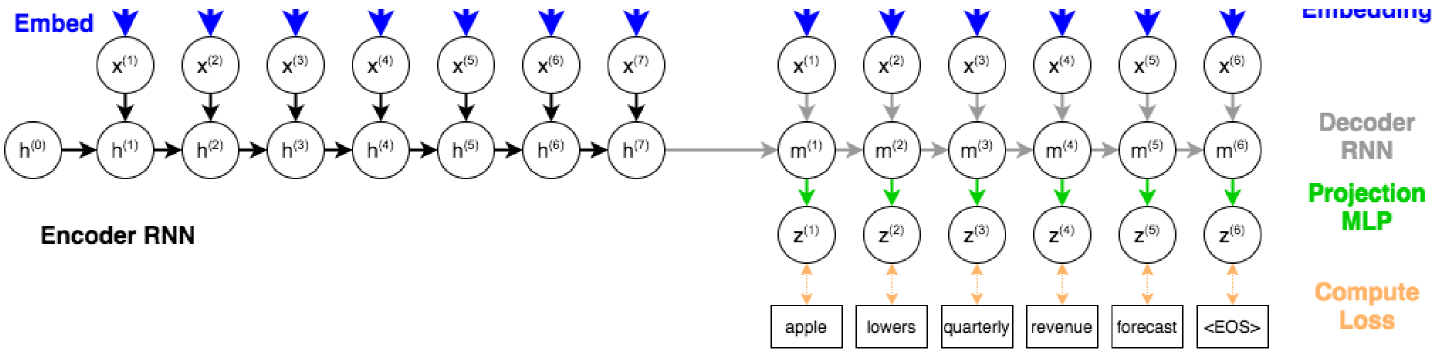
Training the architecture

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 [13]:

```
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.
        """

        out_seq ,last_hidden= self.decode(inp[:,:-1],self.encode(inp))
        return out_seq
```

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 [14]:

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

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 [15]:

```
model = AutoEncoder(vocab_size, 128, 128)
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

for it in range(300):
    zt = model(input_seq)
    loss = criterion(zt.squeeze(0).double(),input_seq[0,1:])
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

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

```
[Iter 50] Loss 0.109590
[Iter 100] Loss 0.028804
[Iter 150] Loss 0.017951
[Iter 200] Loss 0.012529
[Iter 250] Loss 0.009340
[Iter 300] Loss 0.007280
```

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 [16]:

```
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
for i in range(5):
  headline = train_data[i].title
  print("True seq: ",headline)
  input_seq = torch.Tensor([vocab.stoi[w] for w in headline]).long().unsqueeze(0)
  prediced_seq = sample_sequence(model, model.encode(input_seq), max_len=20, temperature
=1)
  print("Predicted seq: ",prediced_seq, "\n")
```

```
True seq:  ['<bos>', 'dems', 'move', 'to', 'end', 'shutdown', ',', 'without', 'wall', 'mo
ney', '<eos>']
Predicted seq:  ['zambian', 'president', 'swears', 'in', 'new', 'army', 'chief']

True seq:  ['<bos>', 'hearts', 'broken', 'after', '_num_-year-old', 'killed', 'before', '
True seq:  ['<bos>', 'hearts', 'broken', 'after', '_num_-year-old', 'killed', 'before', '
nye', '<eos>']
Predicted seq:  ['zambian', 'president', 'swears', 'in', 'new', 'army', 'chief']

True seq:  ['<bos>', 'russia', 'arrests', 'american', 'suspected', 'of', 'espionage', '<e
os>']
Predicted seq:  ['zambian', 'president', 'swears', 'in', 'new', 'army', 'chief']

True seq:  ['<bos>', 'warren', 'takes', 'step', 'to', 'challenge', 'trump', 'in', '_num_'
, '<eos>']
Predicted seq:  ['zambian', 'president', 'swears', 'in', 'new', 'army', 'chief']

True seq:  ['<bos>', 'u.s.', 'navy', 'pursuing', 'block', 'buy', 'of', 'two', 'aircraft',
'carriers', ':', 'senator', '<eos>']
Predicted seq:  ['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.

Explanaion:

We generally don't want the temperature to be too large, because when using a large temperature, it "flattens" the distribution of the words, so the model will generate different words. That is because the high-temperature causes higher randomness in the predictions, rather than a prediction based on the likelihood of the tokens. We can see that when using high temperature, the predictions are only random, and there are no words predicted based on similarity or likely tokens that would make sense.

In [17]:

```python
# Include the generated sequences and explanation in your PDF report.
temp = [1.5,2,5]
for j in temp:
  print('The temperature is: ',j)
  for i in range(5):
    headline = train_data[i].title
    print("True seq: ",headline)
    input_seq = torch.Tensor([vocab.stoi[w] for w in headline]).long().unsqueeze(0)
    prediced_seq = sample_sequence(model, model.encode(input_seq), max_len=20, temperatu
re=j)
    print("Predicted seq: ",prediced_seq, "\n")
```

```
The temperature is:  1.5
True seq:  ['<bos>', 'dems', 'move', 'to', 'end', 'shutdown', ',', 'without', 'wall', 'mo
ney', '<eos>']
Predicted seq:  ['president', 'emerge', 'airplane', 'reed', 'imported', 'platforms', 'sis
ters', 'exporter', 'excess', 'major', 'wnba', 'handelsblatt', 'capture', 'sharara', 'heig
hts', 'entire']

True seq:  ['<bos>', 'hearts', 'broken', 'after', '_num_-year-old', 'killed', 'before', '
nye', '<eos>']
Predicted seq:  ['endo', 'slavery', 'rpt-india', 'sweetens', 'swift', 'erase', 'army', 'c
riticized', 'chief', 'dior', 'curb', 'steps', '2,500']

True seq:  ['<bos>', 'russia', 'arrests', 'american', 'suspected', 'of', 'espionage', '<e
os>']
Predicted seq:  ['in', 'new', 'halted', '_num_-california', 'ola', 'rupees', 'snow', 'chi
ef', 'dramatic', 'bridge', '_num_-climate', 'lakers', 'overcome', 'army', 'chief', 'swear
s', 'events', 'new', 'globes', 'army']

True seq:  ['<bos>', 'warren', 'takes', 'step', 'to', 'challenge', 'trump', 'in', '_num_'
, '<eos>']
Predicted seq:  ['gordon', 'visits', 'abiy', 'abusing', 'army', 'payrolls', 'donald', 'sp
orts', 'oecd', 'impairment', 'manager', 'medicaid', 'popularity', 'dp', 'army', 'chief']

True seq:  ['<bos>', 'u.s.', 'navy', 'pursuing', 'block', 'buy', 'of', 'two', 'aircraft',
'carriers', ':', 'senator', '<eos>']
Predicted seq:  ['zambian', 'nowhere', 'busy', 'auction', 'new', 'army', 'chief']

The temperature is:  2
True seq:  ['<bos>', 'dems', 'move', 'to', 'end', 'shutdown', ',', 'without', 'wall', 'mo
ney', '<eos>']
Predicted seq:  ['anti-government', 'red', 'mastercard', 'john', 'lpga', 'before', 'expan
d', 'kids', 'heart', 'harming', 'chief', 'extension', 'intensifies', '_num_-pga', 'animal
', 'informed', 'controlled', 'beaten', 'cryptocurrencies', 'greenlight']

True seq:  ['<bos>', 'hearts', 'broken', 'after', '_num_-year-old', 'killed', 'before', '
nye', '<eos>']
Predicted seq:  ['sclerosis', '_num_-in', 'dortmund', 'down', 'nowhere', 'flint', 'organi
c', 'revolution', 'retail', 'authorizes', 'suing', 'bosnian', 'already', 'stores', 'verst
appen', 'inspire', 'christchurch', 'hamburg', 'celebration', 'security']

True seq:  ['<bos>', 'russia', 'arrests', 'american', 'suspected', 'of', 'espionage', '<e
os>']
Predicted seq:  ['shooter', 'priest', 'heightening', 'studies', 'in', 'investigations', '
hidden', 'state-backed', 'intended', 'need', 'togo', 'lvmh', 'blamed', 'flies', 'cans', '
sense', 'volumes', 'bars', 'surcharge', 'stresses']

True seq:  ['<bos>', 'warren', 'takes', 'step', 'to', 'challenge', 'trump', 'in', '_num_'
, '<eos>']
Predicted seq:  ['_num_-congo', 'releasing', 'soup', 'fame', 'exchange', 'patrols', 'pour
', '_num_-swiss', 'warriors', 'graham', 'mediation', 'raided', 'valencia', 'bump', 'bln-'
, 'till', 'duterte', 'associates', 'golden', 'army']

True seq:  ['<bos>', 'u.s.', 'navy', 'pursuing', 'block', 'buy', 'of', 'two', 'aircraft',
'carriers', ':', 'senator', '<eos>']
Predicted seq:  ['zambian', 'drown', 'president', 'associate', 'soldier', 'canceled', 'lo
uis', 'transport', 'sparkle', '_num_-ab', 'ally', 'millions', 'went', 'semi-final', 'even
t', 'vegas', 'majeure', 'blaze', 'reit', 'hess']

The temperature is:  5
True seq:  ['<bos>', 'dems', 'move', 'to', 'end', 'shutdown', ',', 'without', 'wall', 'mo
ney', '<eos>']
Predicted seq:  ['americans', 'parade', 'konta', 'cattle', 'shot', 'w', 'table-uae', '_nu
m_-canada', 'grind', 'arson', 'neutrality', 'angering', 'pump', 'lane', 'fourth', 'fear',
'nasdaq-style', 'goal', 'bullying', 'endorses']

True seq:  ['<bos>', 'hearts', 'broken', 'after', '_num_-year-old', 'killed', 'before', '
nye', '<eos>']
Predicted seq:  ['charter', 'retailer', 'defamation', 'corrected-us', 'nascar', 'prompts'
, 'films', 'anything', 'view', 'therapy', 'four-day', 'bertens', 'arguments', 'creditors'
, 'sadr', 'liquidation', 'intent', 'kaepernick', 'body', 'colts']

True seq:  ['<bos>', 'russia', 'arrests', 'american', 'suspected', 'of', 'espionage', '<e
os>']
Predicted seq:  ['appears', 'long-term', 'shares', 'data', 'micron', 'rough', 'believe',
'sanctuary', 'eric', 'improved', 'nevada', 'leviathan', 'manager', 'athlete', 'iea', 'sun
day', 'ericsson', 'adelaide', 'adani', 'president-elect']

True seq:  ['<bos>', 'warren', 'takes', 'step', 'to', 'challenge', 'trump', 'in', '_num_'
, '<eos>']
Predicted seq:  ['detail', 'cholera', 'defense', 'knee', 'credit', 'tops', 'county', 'sis
i', 'unicorn', 'two-month', 'swears', 'buys', 'carter', 'percent', 'design', 'timeline',
'depression', 'lewis', 'foot', 'pro-government']

True seq:  ['<bos>', 'u.s.', 'navy', 'pursuing', 'block', 'buy', 'of', 'two', 'aircraft',
'carriers', ':', 'senator', '<eos>']
Predicted seq:  ['modi', 'messaging', 'defense', 'kyrgyz', 'stock', 'deny', 'trade-fueled
', 'league', 'malaria', 'temperatures', 'saa', 'kingdom', 'jailed', 'confrontation', 'ris
e', 'biosimilar', 'testify', 'alleges', 'heatwave', 'steep']
```

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 [18]:

```
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
    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 [19]:

```
# Report your values here. Make sure that you report the actual values,
# and not just the code used to get those values
chosen_headline = train_data[40].title
print('Original headline:',chosen_headline)

for i in range(5):
  l =[]
  noisy_headline = tokenize_and_randomize(' '.join(chosen_headline[1:-1]),drop_prob=(i+1
)*0.1,blank_prob=(i+1)*0.1,sub_prob=(i+1)*0.1,shuffle_dist=3)
  for j in noisy_headline:
    l.append(vocab.itos[j])
  print('Noisy headline',i+1,' is: ',l)
```

```
Original headline: ['<bos>', 'nfl', 'notebook', ':', 'ab', 'was', 'reportedly', 'benched'
, 'for', 'week', '_num_', '<eos>']
Noisy headline 1  is:  ['<bos>', 'nfl', 'notebook', ':', 'reportedly', 'ab', 'was', '<unk
>', 'week', '_num_', '<eos>']
Noisy headline 2  is:  ['<bos>', '<pad>', ':', '<pad>', 'notebook', 'aboard', '<unk>', 'f
or', '_num_', 'week', '<eos>']
Noisy headline 3  is:  ['<bos>', '<pad>', 'was', '<pad>', '<pad>', '<eos>']
Noisy headline 4  is:  ['<bos>', '<pad>', 'ab', 'reportedly', '<pad>', '<eos>']
Noisy headline 5  is:  ['<bos>', '<pad>', 'enter', '<pad>', '<pad>', '<pad>', '<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 [20]:

```
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()

    model.cuda()
    iter, val_loss_list = [], []

    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
```

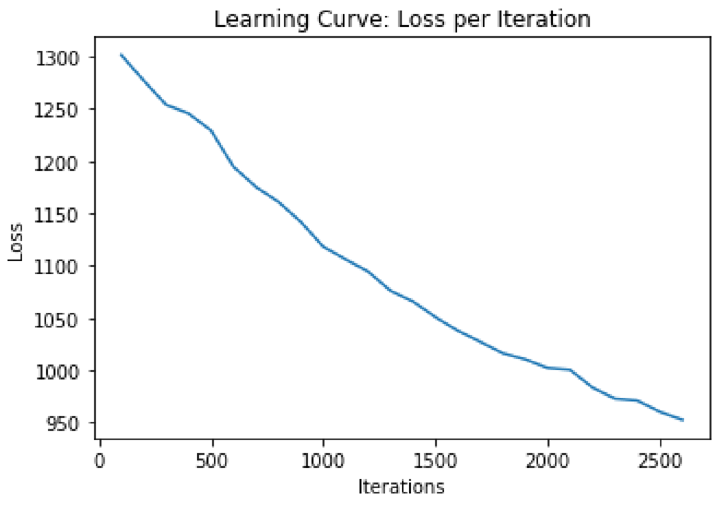
```
                                             pad_token=vocab.stoi['<pad>'])
        dataset = data.TabularDataset(train_path, "tsv", [('title', field)])

        # 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), # to minimize
padding
                                          repeat=False)

        dataset_valid = data.TabularDataset(valid_path, "tsv", [('title', field)])

        valid_iter = data.BucketIterator(dataset_valid,
                                          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
            xs = xs.cuda()
            zt = model(xs)
            loss = criterion(zt.reshape(-1, vocab_size).double(),xs[:,1:].reshape(-1))
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

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

                sum_val_loss = 0
                for j, ((xv, lengths), _) in enumerate(valid_iter):
                    xv = xv.cuda()
                    zs = model(xv)
                    val_loss = criterion(zs.reshape(-1, vocab_size).double(),xv[:,1:].re
shape(-1)) # TODO
                    sum_val_loss += float(val_loss)
                iter.append(it)
                val_loss_list.append(sum_val_loss)

        return iter,val_loss_list
def plot_learning_curve(iter, val_loss_list):
    """
    Plot the learning curve.
    """
    plt.title("Learning Curve: Loss per Iteration")
    plt.plot(iter, val_loss_list, label="Valid")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.show()


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

In [21]:

```
learning_curve_info = train_autoencoder(model, batch_size=64, learning_rate=0.001, num_e
pochs=1)
iter, val_loss_list = learning_curve_info
plot_learning_curve(iter, val_loss_list)
```

```
[Iter 100] Train Loss 4.461610
[Iter 200] Train Loss 4.466244
[Iter 300] Train Loss 4.719491
[Iter 400] Train Loss 4.025659
[Iter 500] Train Loss 3.858963
[Iter 600] Train Loss 4.011607
[Iter 700] Train Loss 4.313838
[Iter 800] Train Loss 4.033022
[Iter 900] Train Loss 4.300333
```

```
[Iter 1000] Train Loss 3.914034
[Iter 1100] Train Loss 3.434082
[Iter 1200] Train Loss 3.533071
[Iter 1300] Train Loss 3.438324
[Iter 1400] Train Loss 3.916340
[Iter 1500] Train Loss 3.397478
[Iter 1600] Train Loss 3.681675
[Iter 1700] Train Loss 3.410680
[Iter 1800] Train Loss 3.543047
[Iter 1900] Train Loss 3.244884
[Iter 2000] Train Loss 3.003610
[Iter 2100] Train Loss 3.504615
[Iter 2200] Train Loss 3.289493
[Iter 2300] Train Loss 3.575633
[Iter 2400] Train Loss 3.805813
[Iter 2500] Train Loss 3.294019
[Iter 2600] Train Loss 3.034972
```



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 [22]:

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

Out[22]:

```
<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 [23]:

```
# 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()

temp = [0.7,0.9,1.5]
for j in temp:
  print('The temperature is: ',j)
  for i in range(5):
    print("True seq: ",headline)
    prediced_seq = sample_sequence(model, model.encode(input_seq), max_len=20, temperatu
re=j)
    print("Predicted seq: ",prediced_seq, "\n")
```

```
The temperature is:  0.7
True seq:  ['<bos>', 'wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish',
'line', 'of', 'a', 'turbulent', 'year', '<eos>']
Predicted seq:  ['wall', 'street', 'rises', ',', 'limps', 'die', 'win', "'s", 'employees'
, '<pad>', 'but', 'january', 'marches']

True seq:  ['<bos>', 'wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish',
'line', 'of', 'a', 'turbulent', 'year', '<eos>']
Predicted seq:  ['wall', 'street', 'rises', ',', 'limps', 'australia', 'young', ',', 'pla
n', 'day', 'stepping', 'january', 'quarter']

True seq:  ['<bos>', 'wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish',
'line', 'of', 'a', 'turbulent', 'year', '<eos>']
Predicted seq:  ['wall', 'street', 'rises', ',', 'limps', 'die', 'win', 'at', "'s", 'near
s', 'her', 'commission', 'high']

True seq:  ['<bos>', 'wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish',
'line', 'of', 'a', 'turbulent', 'year', '<eos>']
Predicted seq:  ['wall', 'street', 'rises', ',', 'limps', 'open', 'sentence', ',', 'kick'
, '<unk>', 'yemen', 'big', 'ebay']

True seq:  ['<bos>', 'wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish',
'line', 'of', 'a', 'turbulent', 'year', '<eos>']
Predicted seq:  ['wall', 'street', 'rises', ',', 'limps', 'die', 'win', 'at', 'of', 'scie
nces', 'election', 'four', 'sensitive']

The temperature is:  0.9
True seq:  ['<bos>', 'wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish',
'line', 'of', 'a', 'turbulent', 'year', '<eos>']
Predicted seq:  ['wall', 'street', 'rises', ',', 'limps', 'heist', 'joins', ',', 'date',
'<pad>', 'three', 'after', 'outlook']

True seq:  ['<bos>', 'wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish',
'line', 'of', 'a', 'turbulent', 'year', '<eos>']
Predicted seq:  ['wall', 'street', 'rises', ',', 'limps', 'die', 'win', 'at', ',', 'pay',
'despite', 'recession', 'paris']

True seq:  ['<bos>', 'wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish',
'line', 'of', 'a', 'turbulent', 'year', '<eos>']
Predicted seq:  ['wall', 'street', 'rises', ',', 'limps', 'die', 'win', 'at', 'of', 'scie
nces', '<pad>', 'presidential', 'amid']

True seq:  ['<bos>', 'wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish',
'line', 'of', 'a', 'turbulent', 'year', '<eos>']
Predicted seq:  ['wall', 'street', 'rises', ',', 'limps', 'back', 'scandal', 'detroit', "
's", 'bashes', '<pad>', 'this', 'year']

True seq:  ['<bos>', 'wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish',
'line', 'of', 'a', 'turbulent', 'year', '<eos>']
Predicted seq:  ['wall', 'street', 'rises', ',', 'limps', 'across', 'spanish', 'with', "'
s", '<pad>', 'transitional', 'percent', 'group']

The temperature is:  1.5
True seq:  ['<bos>', 'wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish',
'line', 'of', 'a', 'turbulent', 'year', '<eos>']
Predicted seq:  ['wall', 'street', 'tops', 'futures', 'equality', 'risk', 'roll', 'quaran
tined', 'after', 'latin', 'prime', 'people', 'delayed']

True seq:  ['<bos>', 'wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish',
'line', 'of', 'a', 'turbulent', 'year', '<eos>']
Predicted seq:  ['wall', 'street', 'rises', ',', 'look', 'drones', 'royals', 'lewis', 'to
', '_num_-wework', 'fall', 'issues', 'nepal']

True seq:  ['<bos>', 'wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish',
'line', 'of', 'a', 'turbulent', 'year', '<eos>']
Predicted seq:  ['wall', 'street', 'rises', 'per', 'deport', 'for', 'plant', 'prayer', 't
o', 'after', 'season', 'stays', 'ruler']
```

```
True seq:  ['<bos>', 'wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish',
'line', 'of', 'a', 'turbulent', 'year', '<eos>']
Predicted seq:  ['wall', 'street', 'rises', ',', 'jd.com', 'fierce', '<unk>', 'end', 'to'
, 'man', 'clear', 'pimco', 'after', '-sources']

True seq:  ['<bos>', 'wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish',
'line', 'of', 'a', 'turbulent', 'year', '<eos>']
Predicted seq:  ['wall', "'s", 'nutrien', 'trash', 'magna', 'socialists', 'immediate', 'h
igher', 'islamic', 'for', '<pad>', 'remains', 'black']
```

Explanation:

We generally don't want the temperature setting to be too small, because low values of the temperature cause the distribution of the words to be "peakier" , so the variance is low, which means that the model predicts most common words (very likely tokens). so the predicted headlines may not make sense.

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 [24]:

```
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 [25]:

```
valid_iter = data.BucketIterator(valid_data,
                                 batch_size=len(valid_data),
                                 sort_key=lambda x: len(x.title),
                                 repeat=False)

for it, ((xs, lengths), _) in enumerate(valid_iter):
  seq_emb = model.encode(xs).squeeze(0)

print('PyTorch tensor of shape : ',seq_emb.shape)
```

```
PyTorch tensor of shape :  torch.Size([19046, 128])
```

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 [26]:

```
list_val = []
for i in range(19046):
  headline = ' '.join(valid_data[i].title[1:-1])
  list_val.append(headline)
vocab_itos = dict(enumerate(list_val))
vocab_stoi = {hedline:index for index, hedline in vocab_itos.items()}
```

In [50]:

```
seq_emb = seq_emb.detach().numpy()
norms = np.linalg.norm(seq_emb, axis=1)
seq_emb_norm = (seq_emb.T / norms).T
similarities = np.matmul(seq_emb_norm, seq_emb_norm.T)

headline = valid_data[13].title
headlines_similarity = {}
for i in range(19046):
  k = similarities[vocab_stoi[' '.join(valid_data[i].title[1:-1])],vocab_stoi[' '.join(h
eadline[1:-1])]]
  headlines_similarity.update({' '.join(valid_data[i].title[1:-1]):k})
sort_dict = ((sorted(headlines_similarity.items(), key=lambda x:x[1])[-6:-1])[::-1])
common_5 = [tp[0] for tp in sort_dict]
print('The 5 closest headlines to "'+ ' '.join(headline[1:-1]) +'" are:')
for hed in common_5:
  print("\n",hed)
```

```
The 5 closest headlines to "asia takes heart from new year gains in u.s. stock futures" a
re:

 erdogan loses hold over turkish capital , istanbul disputed

 factbox : u.s. prepared to tap emergency oil reserves , perry says no decision yet

 cargill names new head of grain trading and processing

 whiteley out for six to eight weeks

 world leaders risk anger , more protests over inequality : u.n. official
```

Part (c) -- 4%

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

In [47]:

```
headline = valid_data[100].title
headlines_similarity = {}
for i in range(19046):
  k = similarities[vocab_stoi[' '.join(valid_data[i].title[1:-1])],vocab_stoi[' '.join(h
eadline[1:-1])]]
  headlines_similarity.update({' '.join(valid_data[i].title[1:-1]):k})
sort_dict = ((sorted(headlines_similarity.items(), key=lambda x:x[1])[-6:-1])[::-1])
common_5 = [tp[0] for tp in sort_dict]
print('The 5 closest headlines to "'+ ' '.join(headline[1:-1]) +'" are:')
for hed in common_5:
  print("\n",hed)
# Make sure to include the original headline and the 5 closest headlines.
```

```
The 5 closest headlines to "amid u.s. withdrawal plans , u.s.-backed forces still fightin
g in syria" are:

 faa proposes fining boeing $ _num_ million for installing defective parts on _num_ plane
s

 lebanon prosecutor to question ghosn on thursday : state news agency

 mali jihadists say attack on passenger bus was targeting french troops

 slovak far-right leader on trial for hate speech may play election kingmaker

 factbox : roe v. wade at risk : legal challenges to u.s. abortion rights
```

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:

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

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).

In [51]:

```
# Write your code here. Include your generated sequences.
headline_0 = list_val[23]
headline_4 = list_val[24]
e_0 = seq_emb[39]
e_4 = seq_emb[30]
print('the chosen two headlines are: ',"\n",'"',headline_0,'" ,','embedding size: ',e_0.
shape,"\n",'"',headline_4,'" ,','embedding size: ',e_4.shape)

e_0 = torch.Tensor(e_0)
e_0 = torch.unsqueeze(e_0,0)
e_0 = torch.unsqueeze(e_0,0)

e_4 = torch.Tensor(e_4)
e_4 = torch.unsqueeze(e_4,0)
e_4 = torch.unsqueeze(e_4,0)

e_1 = 0.75*e_0+0.25*e_4
e_2 = 0.5*e_0+0.5*e_4
e_3 = 0.25*e_0+0.75*e_4

e_list = [e_0,e_1,e_2,e_3,e_4]
for j in range(5):
    print("\nIteration", j+1, ":")
    for i,n in enumerate(e_list):
        print("e _",i," = ", sample_sequence(model, n, max_len=20, temperature=1.7))
```

```
the chosen two headlines are:
 " fifa studying potential for _num_ world cup to be expanded to _num_ teams : infantino
" , embedding size:  (128,)
 " bulgaria extends bid deadline for sofia airport tender again " , embedding size:  (128
,)

Iteration 1 :
e _ 0  =  ['uk', 'government', 'tariff', 'emir', 'protests', 'dialysis', 'lawmakers', 'co
mpromise', 'after', "n't", 'if', 'may', 'challenge', 'border', 'brussels']
e _ 1  =  ['uk', 'of', 'government', 'signals', 'preserve', 'ag', 'after', 'officials', '
mixed', 'divide', 'acquits', 'northern', 'congress', 'look']
e _ 2  =  ['americans', 'brazil', 'minimum', 'data', 'curtailments', 'second', 'tells', '
russia', 'on', 'narrow', 'still', 'hopes', 'senate']
e _ 3  =  ['amro', 'of', 'exports', 'forecast', 'sentiment', 'sit', 'pm', 'eight', 'on',
'putin', 'limit']
e _ 4  =  ['australia', "'s", 'lng', 'exports', 'slip', 'france', 'sought', 'women', 'cal
l', 'car', 'stockpiles']

Iteration 2 :
e _ 0  =  ['uk', 'government', 'turmoil', 'fail', 'approval', 'lawmakers', 'after', 'fate
', 'to', 'if', 'undecided', 'brexit', 'finance', 'says', 'this']
e _ 1  =  ['britain', 'deadline', 'raises', 'ousting', 'nuclear', 'tensions', 'issue', 's
tepping', ':', 'stem', 'council', 'welcomes', 'christmas']
e _ 2  =  ['government', 'update', 'drops', 'steam', 'tightens', 'not', 'bourses', 'summi
t', 'on', 'eve', 'white', 'northern', 'unity']
e _ 3  =  ['australia', "'s", 'efforts', '$', 'sk', 'stance', 'ahead', 'intelligence', 'i
n', 'deal', 'depot', 'deadlock']
e _ 4  =  ['australia', "'s", 'lng', 'exports', 'slip', 'behind', 'qatar', 'in', 'january
', 'on', 'heavy']

Iteration 3 :
```

```
Iteration 3 :
e _ 0  =  ['uk', 'government', 'will', 'hurdle', 'hopes', 'moment', 'with', 'yemen', 'say
s', 'climate', 'pass', 'to', 'pressure', 'says', 'ceremony']
e _ 1  =  ['get', 'concerns', 'die', 'update', 'deadline', 'petersburg', 'on', 'but', 'co
lombia', 'migrants', 'deal', 'guaido', 'decision', 'solskjaer']
e _ 2  =  ['latvia', 'to', 'economic', 'rpt-china', 'hopes', 'other', 'last', 'fate', ':'
, ',', 'essar', 'measures']
e _ 3  =  ['australia', 'exports', '_num_', 'user', 'board', 'reverse', 'the', 'somalia',
'on', 'u.s.', 'slow', 'cable']
e _ 4  =  ['australia', 'at', 'goyal', 'rise', 'fresh', 'nike', 'call', 'responsibility',
'other', ';', 'freeze']

Iteration 4 :
e _ 0  =  ['uk', 'government', 'will', 'pull', 'planned', 'vote', 'generation', 'clearing
', 'people', 'to', 'stands', 'union', 'brexit', 'pass', 'woes']
e _ 1  =  ['uk', 'government', '/', 'cuts', 'spat', 'vote', 'safe', 'between', 'pm', 'chi
ef', 'fight', 'deal', 'alexa', 'fear']
e _ 2  =  ['end', 'ceo', "'s", '$', 'accounts', 'speech', 'hell', 'northern', 'his', 'agr
ee', 'rba', 'europe', 'trump']
e _ 3  =  ['australia', 'exports', '_num_', 'u.s.-israeli', 'volatility', 'in', 'set', 'l
oyal', 'summit', 'on', 'pension', 'germany']
e _ 4  =  ['australia', "'s", 'lng', 'exports', 'slip', 'behind', 'qatar', 'in', 'january
', 'on', 'crude']

Iteration 5 :
e _ 0  =  ['uk', 'government', 'recycling', 'rule', 'st', 'vote', 'stay', 'xl', 'of', 'pm
', 'reform', 'britain', 'going', 'change', 'meeting']
e _ 1  =  ['pressure', 'at', 'china', 'send', 'told', 'as', 'drifts', 'talks', ':', 'alli
es', 'we', 'parliament', 'jobs', 'fails']
e _ 2  =  ['orban', 'at', 'output', 'rise', 'rpt-india', 'declare', 'as', 'nord', 'rate',
':', 'rule', 'brink', 'homeless']
e _ 3  =  ['australia', "'s", 'consumer', 'cuts', 'fannie', 'butina', 'monday', 'trump',
'server', 'on', 'workers', 'colombia']
e _ 4  =  ['australia', "'s", 'lng', 'exports', 'slip', 'other', 'security', 'headache',
'his', 'base', 'fall']
```