CSC321 Project 4.

Deadline: Thursday, April. 2, by 9pm

Submission: If you are using Google Colab or Jupyter Notebook, the easiest way to submit the assignment is to submit a PDF export of the completed notebook. If you are using Python, please submit a PDF file containing your code, written solutions, and any outputs that we are using to evaluate your work.

Late Submission: Please see the syllabus for the late submission criteria.

This time, we're back to 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 project 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].

[1] Shen et al (2019) "Educating Text Autoencoders: Latent Representation Guidance via Denoising"
https://arxiv.org/pdf/1905.12777.pdf

In [2]:

```
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

%matplotlib inline
```

Question 1

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 [3]:

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

#train_path = '/content/gdrive/My Drive/CSC321/reuters_train.txt' # Update me
#valid_path = '/content/gdrive/My Drive/CSC321/reuters_valid.txt' # Update me

train_path = 'C:/Users/aaron/Documents/School/CSC321/Projects/Project 4/reuters_train.txt' #
Update me
valid_path = 'C:/Users/aaron/Documents/School/CSC321/Projects/Project 4/reuters_valid.txt' #
Update me
```

As promised in previous lecture, we 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 [4]:

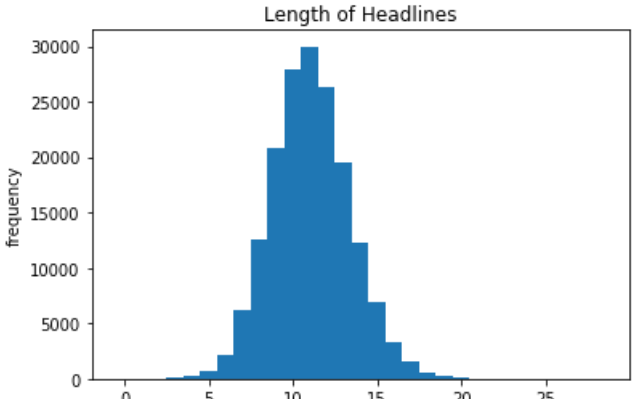
```
import torchtext

# 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 = torchtext.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 in nn.RNN demonstrated in lecture
    use_vocab=True)             # to turn each character into an integer index
train_data = torchtext.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)
```

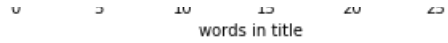
Part (a) -- 2 points

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 [5]:

```
# Include your histogram and your written explanations

# 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:
l = []
for ex in train_data:
    l.append(len(ex.title)-2)

plt.hist(l, bins=np.arange(0,30,dtype=int)-0.5)
plt.title("Length of Headlines")
plt.xlabel("words in title")
plt.ylabel("frequency")
plt.show()

"""
We would be interested in such histograms because we want to be able to
train our model to predict that headlines might be of a certain length
and if it were to see a headline that is too long or too short, it would
try normalize the headline's length to closer match the average.
"""
```





Out[5]:

```
"   \nWe would be interested in such histograms because we want to be able to\ntrain our model to p
redict that headlines might be of a certain length\nand if it were to see a headline that is too l
ong or too short, it would\ntry normalize the headline's length to closer match the average.\n"
```

Part (b) -- 2 points

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

In [6]:

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

# You might find the python class Counter from the collections package useful
from collections import Counter

c = Counter()
for ex in train_data:
    c.update(Counter(ex.title))

del c['<bos>']
del c['<eos>']
print(len(c))
```

```
51298
```

Part (c) -- 2 points

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?

In [7]:

```
# Report your values here. Make sure that you report the actual values,
# and not just the code used to get those values
ones = 0
twos = 0
for word in c:
    if c[word] == 1:
        ones += 1
    elif c[word] == 2:
        twos += 1
print("Number of words that appear exactly once in the training set is " + str(ones))
print("Number of words that appear exactly twice in the training set is " + str(twos))
```

```
Number of words that appear exactly once in the training set is 19854
Number of words that appear exactly twice in the training set is 7193
```

Part (d) -- 2 points

Explain why we may wish to replace these infrequent words with an `<unk>` tag, instead of learning embeddings for these rare words. (Hint: Consider words in the validation set that might not appear in training)

In [8]:

```
# Include your explanation here
'''
These words appear so infrequently that if there ever was a case where the model
decided they should be used, it would most likely be an overfit, that is, the
model has seen the exact sentence before and is putting in the exact word it
knows would fit since it has memorized the sentence. This is bad because
usually nouns and adjectives can be exchanged in a sentence with similar nouns
and adjectives to make an equally valid sentence. If the validation set contains
```

```
sentences of the exact same form but with a different word, the model may
overfit due to a single appearance of a word in the vocabulary. This means that it
cannot properly generate ANY OTHER SENTENCES of the same form even if they
are equally valid. This is why we should add the <unk> tag. Because unk is short
for unknown, if we leave the model with using unknown as a prediction, it is
less prone to overfit due to single instance sentence and word choices.
'''
```

Out[8]:

```
'\nThese words appear so infrequently that if there ever was a case where the model\ndecided they
should be used, it would most likely be an overfit, that is, the\nmodel has seen the exact
sentence before and is putting in the exact word it\nknows would fit since it has memorized the se
ntence. This is bad because\nusually nouns and adjectives can be exchanged in a sentence with simi
lar nouns\nand adjectives to make an equally valid sentence. If the validation set
contains\nsentences of the exact same form but with a different word, the model may\noverfit due t
o a single appearance of a word in the vocabulary. This means that it\ncannot properly generate AN
Y OTHER SENTENCES of the same form even if they\nare equally valid. This is why we should add the
<unk> tag. Because unk is short\nfor unknown, if we leave the model with using unknown as a
prediction, it is\nless prone to overfit due to single instance sentence and word choices.\n'
```

Part (e) -- 2 points

We will only model the top 9995 words in the training set, excluding the tags `<bos>`, `<eos>`, and other possible tags we haven't mentioned yet (including those, we will have a vocabulary size of exactly 10000 tokens).

What percentage of word occurrences will be supported? Alternatively, what percentage of word occurrences in the training set will be set to the `<unk>` tag?

In [9]:

```
# Report your values here. Make sure that you report the actual values,
# and not just the code used to get those values
print("The percentage of word occurrences will be supported is " + str((9995/len(c))*100) + "%")
print("The percentage of word occurrences will be set to the <unk> tag are " + str(100*(ones + two
s)/len(c)) + "%")
```

```
The percentage of word occurrences will be supported is 19.484190416780383%
The percentage of word occurrences will be set to the <unk> tag are 52.725252446489144%
```

Our `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. You can interpret these indices as sparse representations of one-hot vectors.

In [10]:

```
# 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 is actually 10000
vocab_size = len(text_field.vocab.stoi)
print(vocab_size) # should be 10000

# The reason is that torchtext adds two more tokens 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
```

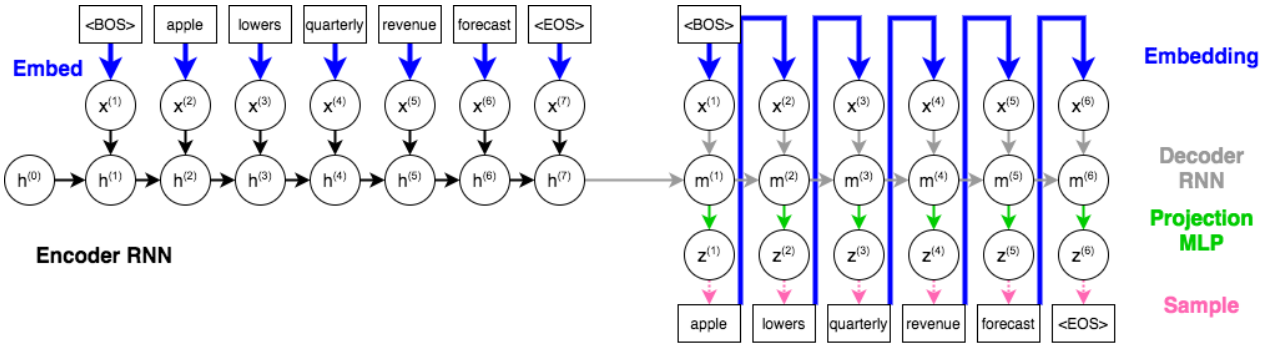
```
0
on
10000
<unk>
<pad>
```

Question 2

Question 2

Building a text autoencoder is a little more complicated than an image autoencoder, so we'll need to thoroughly understand the model that we want to build before actually building our model. Note that the best and fastest way to complete this assignment is to spend a *lot* of time upfront understanding the architecture. The explanations are quite dense, and you might need to stop every sentence or two to understand what's going on. You won't feel productive for a while since you won't be writing code, but this initial investment will help you become more productive later on. Understanding this architecture will also help you understand other machine learning papers you might come across. So, take a deep breath, and let's do this!

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 ${\bf h}^{(7)}$ 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 ${\bf h}^{(7)}$) 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 ${\bf x}^{(t)}$ to our RNN, and will be used to update the hidden state ${\bf m}^{(t)}$. Since each of our sequences begin with the `<bos>` token, we'll set ${\bf x}^{(1)}$ to be the `<bos>` token. 2) Compute the updates to the hidden state ${\bf m}^{(t)}$ based on the previous hidden state ${\bf m}^{(t-1)}$ and ${\bf x}^{(t)}$. Intuitively, this hidden state vector ${\bf m}^{(t)}$ 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 ${\bf m}^{(t)}$, 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 $K=10000$ distinct words. As in a classification problem, the fully-connected neural network will compute a *probability distribution* over these 10,000 words. In the diagram, we are using ${\bf z}^{(t)}$ 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 ${\bf z}^{(t)}$. 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 ${\bf x}^{(t+1)}$ to our RNN, which is used to update our hidden state ${\bf m}^{(t+1)}$---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.

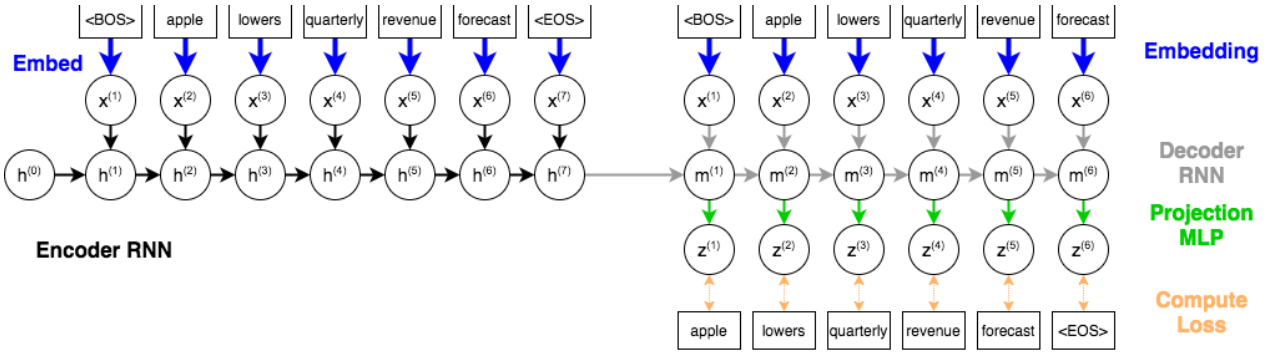
Unfortunately, we can't *train* this autoencoder in the way we just described. That is, we can't just compare our generated sequence with our ground-truth sequence, and get gradients. Both sequences are **discrete** entities, so we won't be able to compute gradients at all! In particular, **sampling is a discrete process**, and so we won't be able to back-propagate through any kind of sampling that we do.

You might wonder whether we can get away with computing gradients by comparing the distributions ${\bf z}^{(t)}$ with the ground truth words at each time step. Like any multi-class classification problem, we can represent the ground-truth words as a one-hot vector, and use the cross-entropy loss.

In theory, we can do this. In practice, there are a few issues. One is that the generated sequence might be longer or shorter than the actual sequence, meaning that there may be more/fewer ${\bf z}^{(t)}$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 ${\bf z}^{(t)}$, we will forgo sampling, and use the **ground truth** ${\bf x}^{(t)}$ 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 ${\bf z}^{(1)},{\bf z}^{(2)}, \cdots {\bf z}^{(T)}$. 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. (This is similar to what we did in a pixel-wise prediction problem.)

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, or initialize our parameters to GloVe embeddings. To prevent students who don't have Colab access from having to download a 1GB file, we won't do that. The word embedding component is represented with blue arrows in the diagram.
- The encoder RNN (which will use Gated Recurrent Units) 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 Gated Recurrent Units) 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.

Part (a) -- 10 pts

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`. (4 points)
2. Complete the `forward` method, which uses teacher forcing and computes the logits $z^{(t)}$ of the reconstruction of the sequence. (4 points)

You should first try to understand the `encode` and `decode` methods, which are written for you. The `encode` method mimics the discriminative 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 very careful of the input that you feed into ether `decode` or to `self.decoder_rnn`. Refer to the teacher-forcing diagram.

In [11]:

```
class AutoEncoder(nn.Module):
    def __init__(self, vocab_size=10000, emb_size=1000, hidden_size=1000):
        """
        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.vocab_size = vocab_size
        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=emb_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.
        """

        out = torch.zeros([inp.shape[0], inp.shape[1]-1, self.vocab_size])

        last_hidden = self.encode(inp)
        for i in range(inp.shape[1]-1):
            out[:, i, :] = torch.FloatTensor(self.decode(inp[:, 0:i+1], last_hidden)[0][:, i, :])
        return out
```

Part (b) -- 5 pts

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 [12]:

```
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 very careful of off-by-ones.**

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

In [13]:

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

ts = input_seq

for it in range(300):

    zs = model(input_seq)

    loss = criterion(zs[:, 0, :], torch.tensor([ts[0, 1]]))
    for i in range(1, zs.shape[1]):
        loss += criterion(zs[:, i, :], torch.tensor([ts[0, i+1]]))
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    if (it+1) % 50 == 0:
```

```
        print("[Iter %d] Loss %f" % (it+1, float(loss)))
```

```
[Iter 50] Loss 0.855394
[Iter 100] Loss 0.249132
[Iter 150] Loss 0.152625
[Iter 200] Loss 0.105321
[Iter 250] Loss 0.077865
[Iter 300] Loss 0.060357
```

Part (c) -- 2 pt

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 [14]:

```
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
hidden = model.encode(input_seq)
print(sample_sequence(model, hidden))
print(sample_sequence(model, hidden))
print(sample_sequence(model, hidden))
print(sample_sequence(model, hidden))
print(sample_sequence(model, hidden))
print(sample_sequence(model, hidden))
print(sample_sequence(model, hidden))
print(sample_sequence(model, hidden))
print(sample_sequence(model, hidden))
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']
['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) -- 3 pt

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**.

In [15]:

```
# Include the generated sequences and explanation in your PDF report.
print(sample_sequence(model, hidden, temperature=1.5))
print(sample_sequence(model, hidden, temperature=2))
print(sample_sequence(model, hidden, temperature=5))
print(sample_sequence(model, hidden, temperature=10))
print(sample_sequence(model, hidden, temperature=20))
print(sample_sequence(model, hidden, temperature=0.95))
print(sample_sequence(model, hidden, temperature=0.8))
print(sample_sequence(model, hidden, temperature=0.5))
print(sample_sequence(model, hidden, temperature=0.25))
print(sample_sequence(model, hidden, temperature=0.2))

"""
We don't want to set the temperature setting to be too large because
a larger temperature setting will dictate that the generated sequences
will have more random changes in their wording. The final generated
sequence will be less reliant on the model that we trained, and more
reliant on random probability. This will result in more incoherent sentence
structure since there will be no "sense" of grammer or form like there
would be if the model was followed more closely.
"""
```

```
['softens', 'army', 'amd', 'controversial', 'kfc', 'army', 'stuttgart', 'considered', 'brf',
'summary', 'intesa', 'new', 'determine']
['schools', 'including', 'following', 'zambian', 'kerr', 'casts', 'peugeot', 'islamic', 'markets-s
tocks', 'army', 'allergan', 'responsible', 'study', 'priests', 'every', 'received', 'information',
'divorce', 'stability', 'handelsblatt']
['henry', 'prayers', 'district', 'surprised', 'tricky', 'disrupt', 'psa', 'whale', 'commitments',
'endorse', 'wynn', 'peloton', 's', 'blasts', 'ammunition', 'ceremony', 'z', '_num_-as', 'kill', 't
rouble']
['faa', 'imposes', 'inaugural', 'traveling', 'vettel', 'carefully', 'big', 'conviction',
'longest', 'barkindo', 'tumble', 'ackman', 'giant', 'cenbank', 'bask', 'self-driving', 'treating',
'crop', 'goma', 'secretary']
['down', 'transneft', 'lift', 'lender', 'afghanistan', 'pressures', 'ea', 'wipe', 'barring',
'opinion', 'prepares', 'weaken', 'flowers', 'bonuses', 'finally', 'doping', 'importer', 'pro-
government', 'babies', 'nio']
['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']
```

Out[15]:

```
'\nWe don\'t want to set the temperature setting to be too large because\na larger temperature set
ting will dictate that the generated sequences\nwill have more random changes in their wording. Th
e final generated\nsequence will be less reliant on the model that we trained, and more\nreliant o
n random probability. This will result in more incoherent sentence\nstructure since there will be
no "sense" of grammer or form like there\nwould be if the model was followed more closely.\n'
```

Question 3

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 (but not up/down---why?). Machine learning practitioners can also add Gaussian noise to the image.

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) -- 3pt

Give three more examples of data augmentation techniques that we could use if we were training an **image** autoencoder. What are different ways that we can change our input?

In [16]:

```
# Include your three answers
"""
Three examples of data augmentation techniques that we could use if we were training an image auto
encoder are:
scaling the image, and rotating the image, and image translations. This would change the input by
making many or
every pixel used as input in an image have a different value when compared to the original.
"""
```

Out[16]:

```
'\nThree examples of data augmentation techniques that we could use if we were training an image a
utoencoder are:\nscaling the image, and rotating the image, and image translations. This would cha
nge the input by making many or\nevery pixel used as input in an image have a different value when
compared to the original.\n'
```

Part (b) -- 2pt

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 [17]:

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

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

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

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

In [18]:

```
# Report your values here. Make sure that you report the actual values,
# and not just the code used to get those values
headline_array = train_data[7].title
headline = ""
for i in range(len(headline_array)):
    if i == 1:
        headline = headline_array[i]
    elif i != 0 and i != len(headline_array)-1:
        headline += " " + headline_array[i]


print("Original headline")
print(headline)
for i in range(5):
    new_headline = ""
    print("New headline " + str(i+1))
    tar = tokenize_and_randomize(headline)
    for j in range(len(tar)):
        if j == 1:
            new_headline = vocab.itos[tar[j]]
        elif j != 0 and j != len(tar)-1:
            new_headline += " " + vocab.itos[tar[j]]
    print(new_headline)
```

```
Original headline
update _num_-glowing new year 's eve ball rises above a sodden times square
New headline 1
<unk> update te 's year eve ball rises a <pad> times <unk> nineteen
New headline 2
special <unk> <pad> new eve 's swap a <unk> times square
New headline 3
update new <unk> <pad> 's lima eve rises a above times square <unk>
New headline 4
update overshadow <unk> year 's eve ball a rises ebitda square
New headline 5
update <pad> <unk> year 's <pad> eve rises a above times aleppo square
```

Part (c) -- 3 pt

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 [28]:

```
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()

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

        # This BucketIterator will handle padding of sequences that are not of the same length
        train_iter = torchtext.data.BucketIterator(dataset,
                                                   batch_size=batch_size,
                                                   sort_key=lambda x: len(x.title), # to minimize p
ding
                                                   repeat=False)
        for it, ((xs, lengths), _) in enumerate(train_iter):
            # Fill in the training code here
            zs = model(xs)
            loss = 0
            for batch in range(xs.shape[0]):
                for i in range(zs.shape[1]-1):
                    loss += criterion(zs[batch, i, :].unsqueeze(0), torch.tensor([xs[batch, i+1]]))
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

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

            if it == 5:
                return

        # 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
train_autoencoder(model, num_epochs=1)
```

```
[Iter 1] Loss 193.750580
[Iter 2] Loss 143.186676
[Iter 3] Loss 174.300903
[Iter 4] Loss 118.735611
[Iter 5] Loss 107.530113
[Iter 6] Loss 129.876495
```

Part (d) -- 2 pt

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 https://www.cs.toronto.edu/~lczhang/321/files/p4model.pk (11MB).

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

In [20]:

```
model = AutoEncoder(10000, 128, 128)
checkpoint_path = 'C:/Users/aaron/Documents/School/CSC321/Projects/Project 4/p4model.pk'
model.load_state_dict(torch.load(checkpoint_path))
```

Out[20]:

```
<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 [21]:

```
# 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()

hidden = model.encode(input_seq)
print(sample_sequence(model, hidden, temperature=0.7))
print(sample_sequence(model, hidden, temperature=0.9))
print(sample_sequence(model, hidden, temperature=1.5))

"""
We generally don't want the temperature setting to
be too **small** because there will not be much deviation
between sample sequences. Lower temperature will mean that
less of our model's predictions will be left up to probability.
"""
```

```
['wall', 'street', 'rises', ',', 'limps', 'across', 'the', 'finish', 'line', 'of', 'a',
'turbulent', 'year']
['wall', 'street', 'rises', ',', 'limps', 'early', 'man', 'unresolved', '<pad>', '_num_', ',', 'go
vernor', 'last']
['wall', 'street', 'rises', ',', 'stem', 'adds', 'grand', 'flat', 'sentence', 'police', 'market',
'northern', 'leads']
```

Out[21]:

```
"\nWe generally don't want the temperature setting to\nbe too **small** because there will not be
much deviation\nbetween sample sequences. Lower temperature will mean that\nless of our model's pr
edictions will be left up to probability.\n"
```

Question 4

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 [22]:

```
valid_data = torchtext.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) -- 2 pt

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 [23]:

```
# Write your code here
# Show that your resulting PyTorch tensor has shape `[19046, 128]`
valid = torch.zeros([19046, 128])
for i in range(len(valid_data)):
    valid[i, :] = model.encode(torch.Tensor([vocab.stoi[w] for w in valid_data[i].title]).long().un
squeeze(0))[0, 0, :]
```

Part (b) -- 2 pt

Find the 5 closest headlines to the headline `valid_data[13]`. Use the cosine similarity to determine closeness. (Hint: You can

use code from Project 2)

In [24]:

```
# Write your code here. Make sure to include the actual 5 closest headlines.
norms = torch.norm(valid, dim=1)
title_emb_norm = (valid.T/norms).T
similarities = torch.matmul(title_emb_norm, title_emb_norm.T)

l = similarities[13,:]
maxes = []
for i in range(5):
    big = None
    for proximal in range(len(l)):
        if (proximal != 13):
            if big == None or (l[proximal] > l[big] and proximal not in maxes):
                big = proximal
    maxes.append(big)

print("Title 13 is: ")
print(valid_data[13].title)
print("The five closest titles to title 13 are: ")
for title in maxes:
    print(valid_data[title].title)
```

```
Title 13 is:
['<bos>', 'asia', 'takes', 'heart', 'from', 'new', 'year', 'gains', 'in', 'u.s.', 'stock', 'future
s', '<eos>']
The five closest titles to title 13 are:
['<bos>', 'italy', "'s", 'salvini', 'loses', 'aura', 'of', 'invincibility', 'in', 'emilia',
'setback', '<eos>']
['<bos>', 'saudi', ',', 'russia', 'look', 'to', 'seal', 'deeper', 'output', 'cuts', 'with', 'oil',
'producers', '<eos>']
['<bos>', 'eu', 'orders', 'quarantine', 'for', 'staff', 'who', 'traveled', 'to', 'northern', 'ital
y', '<eos>']
['<bos>', 'update', '_num_-italy', "'s", 'prime', 'minister', 'says', 'new', 'government', 'will',
'bicker', 'less', '<eos>']
['<bos>', 'portugal', "'s", 'moura', 'pays', 'tribute', 'to', 'cod', 'fishermen', 'at', 'milan', '
fashion', 'close', '<eos>']
```

Part (c) -- 2 pt

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

In [25]:

```
# Write your code here.
# Make sure to include the original headline and the 5 closest headlines.
l = similarities[19045,:]
maxes = []
for i in range(5):
    big = None
    for proximal in range(len(l)):
        if (proximal != 19045):
            if big == None or (l[proximal] > l[big] and proximal not in maxes):
                big = proximal
    maxes.append(big)

print("Title 19045 is: ")
print(valid_data[19045].title)
print("The five closest titles to title 19045 are: ")
for title in maxes:
    print(valid_data[title].title)
```

```
Title 19045 is:
['<bos>', 'washington', 'state', 'man', 'becomes', 'first', 'u.s.', 'coronavirus', 'fatality', '<e
os>']
The five closest titles to title 19045 are:
['<bos>', 'uber', 'stripped', 'of', 'london', 'operating', 'license', ',', 'again', '<eos>']
['<bos>', 'u.s.', 'trade', 'offensive', 'takes', 'out', 'wto', 'as', 'global', 'arbiter', '<eos>']
['<bos>', 'special', 'report-san', 'francisco', 'treasure', 'island', "'s", 'toxic', 'history', '<
eos>']
['<bos>', 'u.s.', 'house', 'leader', 'to', 'back', 'bill', 'limiting', 'court', 'secrecy',
```

```
'<eos>']
['<bos>', 'feature-u.s.', 'lawmakers', 'tuck', 'into', 'juicy', 'debate', 'over', 'meat',
'substitutes', '<eos>']
```

Part (d) -- 4 pts

Choose two headlines from the validation set, and find their embeddings. We will **interpolate** between the two embeddings like we did in https://www.cs.toronto.edu/~lczhang/321/lec/autoencoder_notes.html

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{align*} e_1 &= 0.75 e_0 + 0.25 e_4 \\ e_2 &= 0.50 e_0 + 0.50 e_4 \\ e_3 &= 0.25 e_0 + 0.75 e_4 \\ \end{align*}

Decode each of e_1, e_2 and e_3 five times, with a temperature setting that shows some variation in the generated sequences, while generating sequences that makes sense.

In [27]:

```
# Write your code here. Include your generated sequences.
e1 = valid[13]
e2 = valid[19045]
embedding_values = []
for i in range(0, 4):
    e = e1 * (i/4) + e2 * (4-i)/4
    embedding_values.append(e)

embedding_values = torch.stack(embedding_values)

print("Interpolation 1:")

print(sample_sequence(model, embedding_values[1].unsqueeze(0).unsqueeze(0), temperature=0.5))
print(sample_sequence(model, embedding_values[1].unsqueeze(0).unsqueeze(0), temperature=0.7))
print(sample_sequence(model, embedding_values[1].unsqueeze(0).unsqueeze(0), temperature=0.9))
print(sample_sequence(model, embedding_values[1].unsqueeze(0).unsqueeze(0), temperature=1.5))
print(sample_sequence(model, embedding_values[1].unsqueeze(0).unsqueeze(0), temperature=2))

print("Interpolation 2:")

print(sample_sequence(model, embedding_values[2].unsqueeze(0).unsqueeze(0), temperature=0.5))
print(sample_sequence(model, embedding_values[2].unsqueeze(0).unsqueeze(0), temperature=0.7))
print(sample_sequence(model, embedding_values[2].unsqueeze(0).unsqueeze(0), temperature=0.9))
print(sample_sequence(model, embedding_values[2].unsqueeze(0).unsqueeze(0), temperature=1.5))
print(sample_sequence(model, embedding_values[2].unsqueeze(0).unsqueeze(0), temperature=2))

print("Interpolation 3:")

print(sample_sequence(model, embedding_values[3].unsqueeze(0).unsqueeze(0), temperature=0.5))
print(sample_sequence(model, embedding_values[3].unsqueeze(0).unsqueeze(0), temperature=0.7))
print(sample_sequence(model, embedding_values[3].unsqueeze(0).unsqueeze(0), temperature=0.9))
print(sample_sequence(model, embedding_values[3].unsqueeze(0).unsqueeze(0), temperature=1.5))
print(sample_sequence(model, embedding_values[3].unsqueeze(0).unsqueeze(0), temperature=2))
```

```
Interpolation 1:
['washington', 'world', 'canada', 'victim', 'first', 'fuel', 'weeks', 'trump']
['major', '<unk>', 'bidens', 'lebanon', 'eight', 'u.s.', 'profit', 'billion']
['major', '<unk>', 'admissions', 'house', 'leaps', 'u.s.', 'delivery', 'seat']
['major', 'sues', 'hospital', 'heat', 'pence', 'china', 'controversial', 'jv']
['experimental', 'open', 'british', 'worldwide', 'envoy', 'gdp', 'budget', 'at']
Interpolation 2:
['major', 'markets-stocks', 'local', 'control', 'u.s.', '<unk>', 'export', 'coronavirus', '_num_']
['major', 'markets-stocks', 'state', 'dates', 'wall', 'taiwan', 'impact', 'in', 'stock']
['major', 'disarray', 'firm', 'back', 'death', '-official', 'china', 'fuel', '_num_']
['pressed', 'volcanic', 'office', 'dirty', 'lowest', 'first', 'groups', 'and']
['york', 'as', 'fires', 'ebola', 'of', 'first', 'weekly', 'prosecutors', 'paris']
Interpolation 3:
['major', 'pulled', 'new', 'tests', 'party', ';', 'fund', 'bill', '_num_', '<pad>']
['asia', 'spanish', 'at', 'facing', 'first', 'bln', 'after', 'power', 'focus', '<pad>']
['seconds', "'s", 'stall', 'as', 'cyclone', 'judge', 'field', 'up', 'next', '<pad>']
['wta', 'storm', 'britain', 'in', 'regulator', 'top', 'around', 'u.s.', 'gold', 'alibaba']
['day', 'considers', 'two', 'research', 'ahead', 'first', 'india', 'large', 'to', 'sars']
```

In [0]: