AI 539 Natural Language Processing, Fall 2022
HW 6: Recurrent Neural Language Models

Prof. Liang Huang

Instructions:

- Submit **individually** by Saturday Dec 10, 11:59pm on **Canvas**. No late submissions will be accepted.
- Only `hw6.zip` (which contains `report.pdf` and all your code) should be submitted.
- Worth **12%** of the final grade.
- This HW aims to give you some basic exposure to recurrent neural language models and deep learning.
- This HW compares NLM with n-gram models from EX2 in many of the same tasks: entropy, random generation, space and vowel recovery.

To make it simple, our TA has trained two recurrent neural language models (NLMs) for you, the smaller one ("`base`") on your EX2 `train.txt`, the larger one ("`large`") on a subset of wiki-2 corpus, and a huge one ("`huge`") on 10% of the "one-billion-word" corpus. This Exercise asks you to train and use char-based n-gram language models. Please download `http://classes.engr.oregonstate.edu/eecs/fall2022/ai539-003/hw6/hw6-data.tgz` which contains

1. Trigram models (cf. EX2) trained on base and large settings (the first one is almost identical to the one from EX2 solutions)

   `trigram.base.wfsa.norm` and `trigram.large.wfsa.norm`.

2. NLMs (base, large, and huge) in directory `saved_models/*`. The use of 'huge' is optional.
3. The NLM code: `nlm.py` which provides a class `NLM`.
4. The evaluation scripts: `eval_space.py` (from EX2) and `eval_vowels.py` (from HW1).

You need to run these experiments on `Pytorch` (but you don't need to know anything about it), which we have installed on `pelican.eecs.oregonstate.edu` machines (all 4 of them, `pelican01` to `pelican04`). This HW does **not** need GPUs, though each `pelican` machine does have reasonably good GPUs.

```
$ ssh <ONID>@pelican.eecs.oregonstate.edu

<pelican04:~> ~huanlian/anaconda3/bin/python3
Python 3.9.12 (main, Apr  5 2022, 06:56:58)
[GCC 7.5.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
```

1 Playing with the NLM (0 pts)

You can use the NLM to evaluate the probability of a given sequence:

```
>>> from nlm import NLM
>>> NLM.load('large') # load neural model from file; choose 'base', 'large', or 'huge'
>>> h = NLM() # initialize a state (and observing <s>)
>>> p = 1 # probability
>>> for c in 't h e _ e n d _ '.split():
>>>     print("%.3f" % p, h) # cumulative prob and current state (and distribution of the next char)
>>>     p *= h.next_prob(c)  # include prob (c | ...)
```

```
>>>     h += c  # observe another character (changing NLM state internally)

1.000 "<s>": [t: 0.19, i: 0.10, a: 0.09, m: 0.07, s: 0.06, h: 0.06, c: 0.05, b: 0.04, p: 0.04, f: 0.04,
0.189 "<s> t": [h: 0.82, o: 0.09, r: 0.03, e: 0.02, i: 0.01, w: 0.01]
0.156 "<s> t h": [e: 0.88, i: 0.07, a: 0.02, r: 0.01]
0.138 "<s> t h e": [_: 0.88, y: 0.04, r: 0.03, s: 0.01]
0.121 "<s> t h e _": [s: 0.13, c: 0.09, f: 0.08, t: 0.07, p: 0.06, a: 0.05, m: 0.05, r: 0.05, b: 0.05, n
0.004 "<s> t h e _ e": [n: 0.24, a: 0.13, x: 0.11, v: 0.08, p: 0.07, r: 0.05, l: 0.05, m: 0.04, s: 0.04
0.001 "<s> t h e _ e n": [d: 0.41, g: 0.23, t: 0.18, v: 0.03, c: 0.03, o: 0.02, e: 0.01, r: 0.01]
0.000 "<s> t h e _ e n d": [_: 0.84, </s>: 0.05, i: 0.05, e: 0.02, s: 0.02]
```

You can also use the NLM to greedily generate (i.e., always choose the most likely next character):

```
>>> NLM.load("large")
>>> h = NLM()
>>> for i in range(100):
>>>     c, p = max(h.next_prob().items(), key=lambda x: x[1])
>>>     print(c, "%.2f <- p(%s | ... %s)" % (p, c, " ".join(map(str, h.history[-4:]))))
>>>     h += c
```

You get something like:

```
t 0.19 <- p(t | ... <s>)
h 0.82 <- p(h | ... <s> t)
e 0.88 <- p(e | ... <s> t h)
_ 0.88 <- p(_ | ... <s> t h e)
s 0.13 <- p(s | ... t h e _)
e 0.18 <- p(e | ... h e _ s)
c 0.31 <- p(c | ... e _ s e)
o 0.88 <- p(o | ... _ s e c)
n 1.00 <- p(n | ... s e c o)
d 1.00 <- p(d | ... e c o n)
_ 0.97 <- p(_ | ... c o n d)
s 0.13 <- p(s | ... o n d _)
e 0.22 <- p(e | ... n d _ s)
a 0.30 <- p(a | ... d _ s e)
s 0.82 <- p(s | ... _ s e a)
o 0.99 <- p(o | ... s e a s)
n 1.00 <- p(n | ... e a s o)
_ 0.88 <- p(_ | ... a s o n)
o 0.21 <- p(o | ... s o n _)
f 0.92 <- p(f | ... o n _ o)
_ 0.98 <- p(_ | ... n _ o f)
t 0.28 <- p(t | ... _ o f _)
h 0.90 <- p(h | ... o f _ t)
e 0.94 <- p(e | ... f _ t h)
_ 0.96 <- p(_ | ... _ t h e)
s 0.12 <- p(s | ... t h e _)
e 0.18 <- p(e | ... h e _ s)
c 0.26 <- p(c | ... e _ s e)
o 0.86 <- p(o | ... _ s e c)
n 1.00 <- p(n | ... s e c o)
d 1.00 <- p(d | ... e c o n)
_ 0.94 <- p(_ | ... c o n d)
s 0.13 <- p(s | ... o n d _)
e 0.21 <- p(e | ... n d _ s)
a 0.29 <- p(a | ... d _ s e)
s 0.81 <- p(s | ... _ s e a)
o 0.99 <- p(o | ... s e a s)
n 1.00 <- p(n | ... e a s o)
_ 0.87 <- p(_ | ... a s o n)
o 0.18 <- p(o | ... s o n _)
f 0.89 <- p(f | ... o n _ o)
_ 0.98 <- p(_ | ... n _ o f)
t 0.28 <- p(t | ... _ o f _)
h 0.90 <- p(h | ... o f _ t)
e 0.94 <- p(e | ... f _ t h)
_ 0.96 <- p(_ | ... _ t h e)
s 0.12 <- p(s | ... t h e _)
e 0.18 <- p(e | ... h e _ s)
c 0.26 <- p(c | ... e _ s e)
o 0.86 <- p(o | ... _ s e c)
n 1.00 <- p(n | ... s e c o)
d 1.00 <- p(d | ... e c o n)
_ 0.94 <- p(_ | ... c o n d)
s 0.13 <- p(s | ... o n d _)
e 0.21 <- p(e | ... n d _ s)
a 0.29 <- p(a | ... d _ s e)
s 0.81 <- p(s | ... _ s e a)
o 0.99 <- p(o | ... s e a s)
n 1.00 <- p(n | ... e a s o)
_ 0.87 <- p(_ | ... a s o n)
o 0.18 <- p(o | ... s o n _)
f 0.88 <- p(f | ... o n _ o)
_ 0.98 <- p(_ | ... n _ o f)
t 0.28 <- p(t | ... _ o f _)
```

2 Evaluating Entropy (2 pts)

1. Like EX2, please first evaluate the trigram entropies (base and large) on EX2 `test.txt`

   ```
   cat <text> | sed -e 's/ /_/g;s/\(.\)/\1 /g' | awk '{printf("<s> %s </s>\n", $0)}' \
              | carmel -sribI <your_wfsa>
   ```

   Hint: both should be around 2.9.

   Q: Is the large version intuitively better than the small version? If so, why? If not, why?

2. Now write a short program to evaluate the entropy of NLMs on the same `test.txt`.

   Hint: they should be around 2.6 and 2.0, respectively.

   Q: Which version (base or large) is better? Why?

3. Is NLM better than trigram in terms of entropy? Does it make sense?

3 Random Generation (2 pts)

1. Random generation from n-gram models.

   Use `carmel -GI 10 <your_wfsa>` to stochastically generate character sequences. Show the results. Do these results make sense?

2. Write a short code to randomly generate 10 sentences (from `<s>` to `</s>`) from NLMs.

   Hint: use `random.choices()` to draw the random sample from a distribution.

3. Compare the results between NLMs and trigrams.

4 Restoring Spaces (4 pts)

1. Recall from EX2 that you can use LMs to recover spaces:

   `therestcanbeatotalmessandyoucanstillreaditwithoutaproblem`

   `thisisbecausethehumanminddoesnotreadeveryletterbyitselfbutthewordasawhole.`

   First, redo the trigram experiments using our provided trigrams, and using Carmel.

   What are the precisions, recalls, and F-scores? (use `eval_space.py`).

   Hint: F-scores (for base and large) should be around 61% and 64%, respectively.

2. Then design an algorithm to recover spaces using NLMs. Note: you can't use dynamic programming any more due to the infinite history that NLM remembers. You can use beam search instead.

   Describe your algorithm in English and pseudocode, and analyze the complexity.

3. Implement it, and report the precisions, recalls, and F-scores.

   Hint: F-scores should be ∼81% and ∼94% using beam size of 20 (and 100% using the 'huge' model).

5 Restoring vowels (4 pts)

1. Redo trigram vowel recovery and report the accuracy. (use `eval_vowels.py`)

   Hint: should be around 42% and 41% (for base and large).

2. Now design an algorithm to recover spaces using NLMs.

   Describe your algorithm in English and pseudocode, and analyze the complexity.

3. Implement it, and report the precisions, recalls, and F-scores.

   Hint: should be around 54% and 80% using beam size of 40. If you use the 'huge' NLM, it will be around 93% (it might be slow to run).