CS4740/CS5740/LING4474/COGST4740
Fall ’22 midterm solutions

Edits made after the initial release of these solutions are in orange.

1. **[14 points] N-gram modeling**

   Here is a small, 3-sentence training corpus. Assume that the vocabulary *only* consists of words occurring in it.

   > mares eat oats and goats eat oats and little lambs eat ivy .
   > a kid will bleat and eat ivy too .
   > piglets sleep and eat .

   No unknown word handling is required in this question. Assume no preprocessing whatsoever: word tokens are simply separated by a space or a newline.

   Note: For all the sub-questions within this question (and elsewhere, wherever applicable), indicate the final answer as a product of fractions that make up the computation, and not just the final value. As an example, consider $\mathsf{count}(\text{language}) = 10$ and $\mathsf{count}(\text{natural}) = 20$; a properly-formatted (but, irrelevant) answer might be $\frac{\mathsf{count}(\text{language})}{\mathsf{count}(\text{natural})}$ (this would be $\frac{10}{20}$ but just giving the count fraction is fine).

   (a) [6 points] Using the maximum likelihood estimation and *unigram* modeling, show the computation for $P(\text{eat oats and eat ivy})$.

$$
\begin{aligned}
&P(\text{eat oats and eat ivy}) = \\
&\quad = P(\text{eat}) \times P(\text{oats}) \times P(\text{and}) \times P(\text{eat}) \times P(\text{ivy}) \\
&\quad = \frac{\mathsf{count}(\text{eat})}{\sum_{i=1}^{|V|} \mathsf{count}(w_i)} \times \frac{\mathsf{count}(\text{oats})}{\sum_{i=1}^{|V|} \mathsf{count}(w_i)} \times \frac{\mathsf{count}(\text{and})}{\sum_{i=1}^{|V|} \mathsf{count}(w_i)} \\
&\qquad \times \frac{\mathsf{count}(\text{eat})}{\sum_{i=1}^{|V|} \mathsf{count}(w_i)} \times \frac{\mathsf{count}(\text{ivy})}{\sum_{i=1}^{|V|} \mathsf{count}(w_i)}
\end{aligned}
$$

   Students did not need to provide the numerical computations, but, for those curious: The vocabulary $V = \{$., a, and, bleat, eat, goats, ivy, kid, lambs, little, mares, oats, piglets, sleep, too, will$\}$ ($|V| = 16$) and $\sum_{i=1}^{|V|} \mathsf{count}(w_i) = 27$; so the numerical fractions are: $\frac{5}{27} \times \frac{2}{27} \times \frac{4}{27} \times \frac{5}{27} \times \frac{2}{27}$.

   (b) [6 points] Using the maximum likelihood estimation and *bigram* modeling, show the computation for $P(\text{eat oats and eat ivy})$. When computing the sentence-initial bigram, use the unigram probability.

Read "$w_i \to w_j$" as "w_i *immediately precedes* w_j":

$$\begin{aligned}
&P(\text{eat oats and eat ivy}) = \\
&\quad = P(\text{eat}) \times P(\text{oats}|\text{eat}) \times P(\text{and}|\text{oats}) \times P(\text{eat}|\text{and}) \times P(\text{ivy}|\text{eat}) \\
&\quad = \frac{\mathsf{count}(\text{eat})}{\sum_{i=1}^{|V|} \mathsf{count}(w_i)} \times \frac{\mathsf{count}(\text{eat} \to \text{oats})}{\sum_{i=1}^{|V|} \mathsf{count}(\text{eat} \to w_i)} \times \frac{\mathsf{count}(\text{oats} \to \text{and})}{\sum_{i=1}^{|V|} \mathsf{count}(\text{oats} \to w_i)} \\
&\qquad \times \frac{\mathsf{count}(\text{and} \to \text{eat})}{\sum_{i=1}^{|V|} \mathsf{count}(\text{and} \to w_i)} \times \frac{\mathsf{count}(\text{eat} \to \text{ivy})}{\sum_{i=1}^{|V|} \mathsf{count}(\text{eat} \to w_i)}
\end{aligned}$$

Students did not need to fill in the numbers, but they are: $\frac{5}{27} \times \frac{2}{5} \times \frac{2}{2} \times \frac{2}{4} \times \frac{2}{5}$.

(c) [2 points] Now using add-k smoothing with $k = 1$ and *bigram* modeling, show the computation of $P(\text{oats} \,|\, \text{eat})$.

Read "$w_i \to w_j$" as "w_i *immediately precedes* w_j":

$$\begin{aligned}
P(\text{oats} \mid \text{eat}) &= \frac{k + \mathsf{count}(\text{eat} \to \text{oats})}{\sum_{i=1}^{|V|}(k + \mathsf{count}(\text{eat} \to w_i))} \\
&= \frac{1 + \mathsf{count}(\text{eat} \to \text{oats})}{|V| + \sum_{i=1}^{|V|} \mathsf{count}(\text{eat} \to w_i)}
\end{aligned}$$

Students did not need to fill in the numbers, but they are (noting that we said to do <u>no</u> unknown word handling, so $|V|$ stays at 16, rather than expanding to 17 due to the addition of an unknown word): $= \frac{1+2}{16+5} = \frac{3}{21}$.

2. **[21 points] Markov models**

   (a) [4 points] The Viterbi algorithm requires two sets of parameters, (1) lexical generation probabilities (i.e., observation likelihoods) and (2) transition probabilities. Assume the task of named-entity recognition (NER) tagging and that an appropriately annotated corpus is provided. Under the assumption of add-k smoothing, how would you mathematically compute the following probabilities (no unseen word handling) in a system of m tags and n tokens:

   i. [2 points] The probability of obtaining tag t_j at time $\mathcal{T}$, if you obtained tag t_i at time $\mathcal{T}-1$.

   With m distinct tokens (read "$t_i \to t_j$" as: t_i at time $\mathcal{T}-1$ *transitions to* t_j at time $\mathcal{T}$):

$$P(t_j \mid t_i) = P(t_i \to t_j) = \frac{k + \mathsf{count}(t_i \to t_j)}{\sum_{x=1}^{m}(k + \mathsf{count}(t_i \to t_x))}$$

   ii. [2 points] The probability of tag t_i emitting a token w_j at time $\mathcal{T}-1$.

   With n distinct tokens (read "$t_i \to w_j$" as: t_i *emits* w_j at time $\mathcal{T}-1$):

$$P(w_j \mid t_i) = P(t_i \to w_j) = \frac{k + \mathsf{count}(t_i \to w_j)}{\sum_{x=1}^{n}(k + \mathsf{count}(t_i \to w_x))}$$

   (b) [3 points] Consider the task of NER tagging, with tags (t_i; $i \in \{1, 2, \ldots, m\}$; t_{start} is a special tag associated with the start of the sequence) and tokens (w_j). Consider the following pseudocode implementation to tag an input sequence of n tokens, $\{w_1, w_2, \ldots, w_n\}$, using a hidden Markov model (HMM):

```
**function** tag-sequence($\{w_1, w_2, \ldots, w_n\}, t_{\text{start}}$):
    tags: $T \leftarrow [\,]$
    $t_{\text{prev}} \leftarrow t_{\text{start}}$
    for $j = 1$ to $n$:
        scores: $S \leftarrow [\,]$
        for $i = 1$ to $m$:
            $S[i] \leftarrow P(t_i|t_{\text{prev}}) \times P(w_j|t_i)$
        $t_{\text{prev}} \leftarrow \arg\max_{t_i} S$; $T[j] \leftarrow \arg\max_{t_i} S$
    output $T$
```

   Under the assumption that all necessary probability values are precomputed, justify if the above implementation is better than the brute-force search, and if not, then propose and justify an alternate approach.

   Hint: The search space for the HMM model includes approximately m^n total paths.

   The pseudocode implementation is a greedy search strategy that searches exactly $m \times n$ total paths (which is significantly better than searching m^n paths in the brute-force approach). However, the greedy algorithm provides a sub-optimal solution. Hence, the above implementation is worse than the brute-force approach.

Although the "true" answer is that the above implementation is worse than brute force due to accuracy concerns, we decided not to deduct points for answers that only mentioned efficiency.

The Viterbi search algorithm can be used to find a maximization optimization solution. The algorithm does *not* assume optimality at each step, and rather postpones any hard decision making until the last tag. By maintaining a back-pointer, we can trace the path that lead to the global optimal solution. The Viterbi algorithm searches $m^2 \times n$ total paths, which is significantly better than searching m^n paths in the brute-force approach.

(c) [7 points] For the task of NER tagging, consider using a bigram HMM with the Viterbi algorithm. Let us assume the tag set is as follows:

[org]	:	organization
[per]	:	person
[loc]	:	location
[misc]	:	miscellaneous
[o]	:	*not* a named entity
[s-tag]	:	special tag for the start of the sentence token ([s-tok])
[e-tag]	:	special tag for the end of the sentence token ([e-tok])

Further, assume that we train our bigram HMM model on the following 2-sentence corpus (each token is tagged to its corresponding NER tag below it):

[s-tok]	Messi	scored	four	in	World	Cup	.	[e-tok]
[s-tag]	[per]	[o]	[o]	[o]	[misc]	[misc]	[o]	[e-tag]

[s-tok]	Hamm	scored	100	in	NLP	.	[e-tok]
[s-tag]	[per]	[o]	[o]	[o]	[misc]	[o]	[e-tag]

Compute the score [=likelihood, as announced during the exam] assigned by the bigram HMM model to the following sequence (with the associated NER tags) as shown below:

[s-tok]	Hamm	scored	four	in	NLP	.	[e-tok]
[s-tag]	[per]	[o]	[o]	[o]	[misc]	[o]	[e-tag]

Note: Please do *not* attempt to reduce the obtained solution into a single fraction or a decimal equivalent. It is expected that you retain the fractions that make up the computations.

$$\begin{array}{llr}
 & P(\text{[s-tok]} \mid \text{[s-tag]}) & \times \\
P(\text{[per]} \mid \text{[s-tag]}) & \times\, P(\text{Hamm} \mid \text{[per]}) & \times \\
P(\text{[o]} \mid \text{[per]}) & \times\, P(\text{scored} \mid \text{[o]}) & \times \\
P(\text{[o]} \mid \text{[o]}) & \times\, P(\text{four} \mid \text{[o]}) & \times \\
P(\text{[o]} \mid \text{[o]}) & \times\, P(\text{in} \mid \text{[o]}) & \times \\
P(\text{[misc]} \mid \text{[o]}) & \times\, P(\text{NLP} \mid \text{[misc]}) & \times \\
P(\text{[o]} \mid \text{[misc]}) & \times\, P(\text{.} \mid \text{[o]}) & \times \\
P(\text{[e-tag]} \mid \text{[o]}) & \times\, P(\text{[e-tok]} \mid \text{[e-tag]}) &
\end{array}$$

(d) [3 points] String identity can be expensive to use as a feature in MEMMs. Briefly describe an efficient method of including string-based features in an MEMM classifier.

A simple solution is to use binary string-indicator functions for a small set of relevant terms (e.g., "word_is_cat"). It was important to specify a reduced set; creating a binary feature for many or all possible terms is not efficient. Other acceptable answers include using word shape represented through indicators (binary or numeric values), capitalization features, numeric features capturing some specifics, and others.

(e) [4 points] Each of the following is a possible MEMM input feature that could be used for NER tagging. Circle all that are valid to use. If none are valid, write "None".

A) Conditional probability of the next token given the current token

B) Conditional probability of the previous token given the current NER tag

C) Conditional probability of the current token given the current POS tag

D) HMM start probability of the current NER tag

Note that the question asks to indicate *valid* features:

- Valid: A, C, D (although D requires building an HMM). Answers that listed at least one of A, C, or D received the (single) point allocated for indicating correct valid answers.
- Invalid: B (attempt to condition on Y when computing $P(Y|X)$ without an explicit Bayes flip)

3. **[13 points] Word embeddings**

   (a) [4 points] What is the purpose of using negative samples when training word embeddings?

   To provide an comparison class to define a partition against. To put it another way, if you only have positive samples in your training data, you could learn a classifer that always says "everything has the positive label".

   Advanced: Sampling (as opposed to using all possible negative examples) makes the class sizes more balanced and makes training faster than evaluating the cross-product of all lexical pairs.

   (b) [4 points] Explain the distributional hypothesis that is foundational to word embeddings.

   Similar words occur in similar contexts. References to Zipf's law were not correct responses to this question.

   (c) [2 points] Distributed word representations (like skipgram) do not explicitly handle unknown tokens. What can you do to get an embedding for an unknown token that was not seen during training?

   Several answers are possible:

   - replace the word with an [unk] token for which you do have embeddings, or
   - use the hidden layer from when you first encounter that word as its embedding, if the hidden-layer dimension is the same as the embedding dimension, or
   - average the context word embeddings to represent the unseen word

   (d) [3 points] Using precomputed word vectors (such those learned by Word2Vec) is common in language modeling. But we can also learn word embeddings on the fly when training a feed-forward neural network for specific tasks. Why would you train word embeddings instead of using precomputed Word2Vec embeddings?

   Because the trained word embeddings will be tuned to your particular task (as long as you have the time and computing power to do your own training).

4. **[11 points] Neural networks and backpropagation**

Each of the following questions presents a pseudocode implementation of (a part of) a neural network. Looking at the pseudocode, please note if there is anything missing and/or incorrect in the given implementation, if yes, then indicate and correct it, and if not, then just record your answer as nothing being wrong. (Explicitly state and justify any and all assumptions made.)

Notation: $x^{(i)} \in \mathbb{R}^n$ denotes the i-th training sample (of total m samples), $y^{(i)} \in \mathbb{R}$ denotes the prediction for that i-th training sample, and θ_js are the parameters (weights) of the algorithm.

Bonus: For all of the following pseudocode implementations, there is the added error that θ is $\vec{0}$ (since θ gets *reset* to $\vec{0}$, not updated, at every forward pass, effectively nullifying any gradient ascent/descent updates). So, any answers that read "need to be assigned randomly" are wrong—the parameter initialization needs to moved out of the forward$_{nn}(x)$.

However, given that the hint for at least one of these tells people to look at the activation functions, people need to have identified the issues with the activation functions to get full credit on these problems.

Note: Technically speaking, the given activation must be defined and differentiable everywhere (or at least the sub-gradients must be explicitly defined since this is a pseudocode implementation (e.g., for ReLU at $\theta^T x^{(i)} = 0$)). But given the lack of a calculus prerequisite in the course, we are only requiring students to propose solutions that address the input domain issue, and not requiring solutions involving differentiability.

(a) [4 points] The pseudocode is as follows:

```
function $f(x \in \mathbb{R})$:
    $z \leftarrow \exp(x + 1)$
    $z \leftarrow \log(z)$
    output $z$

function forward$_{\text{nn}}(x \in \mathbb{R}^{m \times n})$:
    $\theta \in \mathbb{R}^n \leftarrow \vec{0}$
    $y \in \mathbb{R}^m$
    for $i = 1$ to $m$:
        $h_{\text{out}} \leftarrow \theta^T x^{(i)}$
        $y^{(i)} \leftarrow f(h_{\text{out}})$
    output $y$
```

The activation function is linear, $\log(\exp(\theta^T x^{(i)} + 1)) = \theta^T x^{(i)} + 1$, which is incorrect; instead, use a nonlinear activation (e.g., $\log(\exp(\theta^T x^{(i)}) + 1)$, sigmoid, etc.).

(b) [4 points] The pseudocode is as follows:

```
function $f(x \in \mathbb{R})$:
    $z \leftarrow 2 + \exp(-x)$
```

```
        $z \leftarrow 2/z$
        output $z$

    **function** forward$_{\text{nn}}(x \in \mathbb{R}^{m\times n})$:
        $\theta \in \mathbb{R}^n \leftarrow \vec{0}$
        $y \in \mathbb{R}^m$
        for $i = 1$ to $m$:
            $h_{\text{out}} \leftarrow \theta^T x^{(i)}$
            $y^{(i)} \leftarrow f(h_{\text{out}})$
        output $y$
```

The activation function is nonlinear (a modified version of the standard sigmoid). Following the recommendations in (a), we can indicate that there is *no* error in the implementation.

(c) [3 points] The pseudocode is as follows:

```
    **function** $f(x \in \mathbb{R})$:
        $z \leftarrow \sqrt{x}$
        output $z$

    **function** forward$_{\text{nn}}(x \in \mathbb{R}^{m\times n})$:
        $\theta \in \mathbb{R}^n \leftarrow \vec{0}$
        $y \in \mathbb{R}^m$
        for $i = 1$ to $m$:
            $h_{\text{out}} \leftarrow \theta^T x^{(i)}$
            $y^{(i)} \leftarrow f(h_{\text{out}})$
        output $y$
```

Hint: Visualize the network activation.

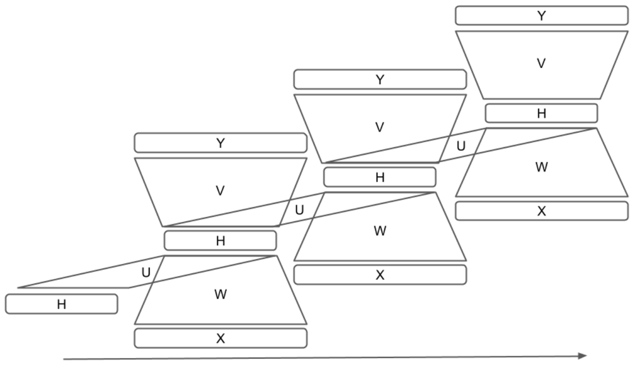
Using optimization methods such as gradient descent requires the gradient to be computed. Hence, an ideal activation function is both differentiable and nonlinear. Observe that $\sqrt{\theta^T x^{(i)}}$ is not defined for $\theta^T x^{(i)} < 0$ and its derivative $\frac{1}{2\sqrt{\theta^T x^{(i)}}}$ is not defined for $\theta^T x^{(i)} \leq 0$.

A simple fix is to use the absolute value in the activation function, such as like this (for some constant γ):

$$f(\theta^T x^{(i)}) = \begin{cases} \gamma, & \theta^T x^{(i)} = 0 \\ \sqrt{|\theta^T x^{(i)}|}, & \text{otherwise} \end{cases}$$

Note that the above activation is differentiable and nonlinear. Any other nonlinear and differentiable activations are acceptable as corrections.

5. **[14 points] Recurrent neural network theory**



Consider the task of using a simple recurrent neural network (RNN) (shown above) for the task of language modeling (i.e. to predict the next word in a sequence of words).

(a) [4 points] What do W and U represent? That is, what functions do they perform?

W projects a *word embedding* to the *hidden* vector, while U transitions the hidden vector *from one time step to the next* (*italicized text* denotes the key phrases). Responses needed to at least indicate both the inputs and the outputs to receive full credit.

(b) [4 points] In some language models X and Y will not have the same dimensionality. Why not, and in those cases, what is their dimensionality?

This occurs when X is a word embedding while Y represents a one-hot vocabulary vector—X is of length $\mathsf{dim}(\text{embedding})$ and Y is length $|V|$. Note that the question specifies that we are dealing with the language-modeling setting, not arbitrary classification settings.

(c) [3 points] At test time, which of the objects denoted by variables in the above image change over time, and which do not? In your answer, account for all of H, U, V, W, X, Y.

Change: X (the value of input), Y (the value of the output), H (the value of the hidden layer)
Unchanging: U, W, V

(d) [3 points] What is the dimensionality/shape of U? You can use the function **len(·)** in your answer, which takes a vector as an argument and returns its length.

$\mathsf{len}(H) \times \mathsf{len}(H)$

6. **[3 points] Recurrent neural network applications**

Consider a system with the following characteristics:

- Input: note made by a doctor
- Step 1: run a recurrent neural network language model (RNN-LM) on the input
- Step 2: run a classifier on the final hidden layer activations of the RNN-LM
- Output: "serious" (=patient has a serious medical condition, according to the doctor), or "not serious"

You notice that that the system mistakenly classifies the note below as "serious". You think the problem might involve the last words, "deadly disease", whose occurrence is misleading here.

> The patient is 24 years old and has not been diagnosed with any chronic health conditions. All vital signs normal. A smallpox test was negative. I am so looking forward to the end of that deadly disease!

(a) [3 points] Explain how the architecture of an RNN-LM could be leading to the classification error.

The words "deadly disease" (which is expected to be a strong indicator of note criticality) could have caused the note to be misclassified, especially because it appears towards the end of a long sequence—RNN "forgets" stuff near the beginning, as it all gets mushed up in the hidden state over a long pass over many words.