Assignment 1

Version: 1.0
Version Release Date: 2023-01-16

Deadline: Friday, Feb. 03, at 11:59pm.

Submission: You must submit two files through MarkUs[1]: (1) a PDF file containing your writeup, titled `a1-writeup.pdf`, and (2) your code file `a1-code.ipynb`. There will be sections in the notebook for you to write your responses. Your writeup must be typed. Make sure that the relevant outputs (e.g. `print_gradients()` outputs, plots, etc.) are included and clearly visible.
See the syllabus on the course website[2] for detailed policies. You may ask questions about the assignment on Piazza[3]. *Note that 10% of the assignment mark (worth 2 pts) may be removed for lack of neatness.*

You may notice that some questions are worth 0 pt, which means we will not mark them in this Assignment. Feel free to skip them if you are busy. However, you are expected to see some of them in the midterm. So, we won't release the solution for those questions.

The teaching assistants for this assignment are Denny Wu and Yangjun Ruan. Send your email with subject "*[CSC413] A1 ...*" to `csc413-2023-01-tas@cs.toronto.edu` or post on Piazza with the tag `a1`.

[1] `https://markus.teach.cs.toronto.edu/2023-01`
[2] `https://uoft-csc413.github.io/2023/assets/misc/syllabus.pdf`
[3] `https://piazza.com/class/lcp8mp3f9dl7lp`

Important Instructions

Read the following before attempting the assignment.

Overview and Expectations

You will be completing this assignment with the aid of large language models (LLMs) such as ChatGPT, text-davinci-003, or code-davinci-002. The goal is to help you (*i*) develop a solid understanding of the course materials, and (*ii*) gain some experience in using LLMs for problem solving. Think of this as analogous to (*i*) understanding the rules of addition and multiplication, and (*ii*) learning how to use a calculator. Note that LLMs may not be a reliable "calculator" (yet) — as you will see, GPT-like models can generate incorrect and contradicting answers. It is therefore important that you have a good grasp of the lecture materials, so that you can evaluate the correctness of the model output, and also prompt the model toward the correct solution.

Prompt engineering. In this assignment, we ask that you try to (*i*) solve the problems yourself, and (*ii*) use LLMs to solve a selected subset of them. You will "guide" the LLMs toward desired outcomes by typing *text prompts* into the models. There are a number of different ways to prompt an LLM, including direct copy-pasting LaTeX strings of a written question, copying function docstrings, or interactively editing the previously generated results. Prompting offers a natural and intuitive interface for humans to interact with and use LLMs. However, LLM-generated solutions depend significantly on the quality of the prompt used to steer the model, and most effective prompts come from a deep understanding of the task. You can decide how much time you want to spend as a university student vs. a prompt engineer, but we'd say it's probably not a good idea to use more than 25% of your time on prompting LLMs. See Best Practices below for the basics of prompt engineering.

What are LLMs good for? We have divided the assignment problems into the following categories, based on our judgment of how difficult it is to obtain the correct answer using LLMs.

- **[Type 1]** LLMs can produce almost correct answers from rather straightforward prompts, e.g., minor modification of the problem statement.
- **[Type 2]** LLMs can produce partially correct and useful answers, but you may have to use a more sophisticated prompt (e.g., break down the problem into smaller pieces, then ask a sequence of questions), and also generate multiple times and pick the most reasonable output.
- **[Type 3]** LLMs usually do not give the correct answer unless you try hard. This may include problems with involved mathematical reasoning or numerical computation (many GPT models do not have a built-in calculator).
- **[Type 4]** LLMs are not suitable for the problem (e.g., graph/figure-related questions).

Program trace and show your work. For questions in [Type 1] [Type 2] , you must include the *program trace* (i.e., screenshots of your interaction with the model) in your submission; the model output does not need to be the correct answer, but you should be

able to *verify* the solution or find the mistake. Whereas for questions in [Type 3] [Type 4] , you do not have to include the program traces, but we encourage you to experiment with LLMs and get creative in the prompting; for problems labeled [EC] in these two categories, extra credit will be given to prompts that lead to the correct solution.

Grading. We will be grading the assignments as follows.

- **Written Part**
  - For questions in [Type 1] [Type 2] , we require you to submit (*i*) **the program trace**, (*ii*) **critique of the model output** (is it correct? which specific part is wrong?), (*iii*) **your own solution**. You will not lose points if you cannot prompt the LLM to generate the correct answer, as long as you can identify the mistake. However, you will receive 0 point on the question if you do not include the program trace, even if you solve the problem by yourself; the goal of this is to help you familiarize with the use of LLMs.
    If you are confident that the model output is correct, you can directly use it as your own solution. Make sure you cite the model properly, that is, **include the model name, version (date), and url if applicable**.
  - For questions in [Type 3] [Type 4] , you will be graded based on the correctness of your own solution, and you do not need to include screenshots of the model output. Creative prompts that lead to correct model output will be rewarded.
- **Programming Part**
  - For writing questions labeled [Type 1] [Type 2] please submit (*i*) the program trace, (*ii*) critique of the model output, (*iii*) your own solution. The grading scheme is the same as the previous written part.
  - For coding questions labeled [Type 1] [Type 2] , submit (*i*) the program trace, (*ii*) your own solution. You will be graded based on the correctness of your own solution, that is, you will receive full marks if your code can execute and produce the desired outcome. From our experience, the most efficient way is to start with the LLM-generated code, which often gives the correct solution under minor modifications. Again, **make sure you cite the model properly**.
  - For questions in [Type 3] [Type 4] , you will be graded based on the correctness of your own solution, and you do not need to include screenshots of the model output.

Best Practices for Program Generation

Most questions in the programming part (including both written and coding questions) could be (at least partially) solved by a GPT-like model (e.g., ChatGPT/text-davinci-003/code-davinci-002). While the written part is more challenging for LLMs, the following guide will still help you with the derivation questions. You could start by naively copy-pasting the question and the context as the prompt, and try to improve the generated answers by trial and error. There are a few tips that may be helpful for you to improve the prompt:

1. **Structure your prompt.** It is useful to structure your prompt as `[Instruction]` + `[Question]` + `[Hint]`:

- `[Instruction]` could specify what you want the model to do, e.g., "Answer the following question" or "Implement the GloVe loss function". Feel free to try more creative prompts like "You are a graduate student working with Geoffrey Hinton and are working on a project on neural networks. Please...".
- `[Question]` specifies the question to answer or the code block to implement.
- `[Hint]` could specify some additional information that you would like to provide for helping the model answer the question.
- Again, feel free to try other formats!

2. **Function signature and docstrings are very useful.** For coding questions where you are required to implement a function (or a part of a function), simply copy-pasting the function signature and docstrings (down to the blank part) to `[Question]` usually already gives reasonable answers. This is probably because the docstring usually provides sufficient information for implementing a function, e.g., the meaning of a function and the meaning/shape/data type of each input/output argument. Be careful to specify the correct shapes for inputs/outputs in the docstring, in particular, when solving 4.4 and 4.5.

3. **Incorporate more useful information into the prompt.** Sometimes you may need to provide some additional information for the model the correctly implement the function. For example, you may need to specify the (vectorized) mathematical formula of the GloVe loss to either `[Instruction]` or `[Hint]` (as included comment) when solving 4.4.

4. **Try to prompt the model to solve simpler questions**. GPT-like models are not that "smart" yet! They typically struggle to solve a question that involves complex reasoning, thus it may be easier to prompt the model to solve simpler decomposed sub-questions step-by-step. For example, you can try to incorporate something like "Let's think step by step"[4]into your `[Hint]` when solving written questions in Sec. 5. You may also try to explicitly decompose the question into several sub-questions and prompt the model to solve them one by one.

5. **Try to generate several times.** GPT-like models randomly generate outputs given the prompt, thus it is not always the case that the generated answers are reasonable. Try to generate the answers 3-5 times and pick the best answer (e.g., using provided test cases or your own judgment). Sometimes the most consistent solution across different random generations may also be the correct one[5]!

6. **Remove useless context in your prompt.** GPT-like models typically have a fixed context window of 4096 tokens (you can think of tokens as pieces of words, where 1000 tokens is about 750 words) which restricts the maximum length of your prompt. Also note that longer prompts could cost you more money if you use an on-demand model like text-davinci-003[6]! (Both ChatGPT and Codex are free). This may be an issue for coding questions if you want to put the whole class definitions into your prompt, where you may need to remove irrelevant class functions from your prompt.

7. **Fill in the blank by extracting from outputs.** In 6.2 where you are required to fill in the blank in a function, you may simply prompt the model to generate the whole function first and extract the answer corresponding to the blank from the output.

[4]See `https://arxiv.org/abs/2205.11916`.
[5]See `https://arxiv.org/abs/2203.11171`.
[6]See `https://openai.com/api/pricing/`.

Written Assignment

What you have to submit for this part

See the top of this handout for submission directions. Here are the requirements.

- The zero point questions (in black below) will not be graded, but you are more than welcome to include your answers for these as well in the submission.
- For (nonzero-point) questions labeled [Type 1] [Type 2] you need to submit (*i*) **the LLM program trace**, (*ii*) **critique of the model output**, (*iii*) **your own solution**. You will receive 0 point on the question if you do not include the program trace, even if you solve the problem by yourself. Your own solution can be a copy-paste of the LLM output (if you verify that it is correct), but make sure you **cite the model properly**.
- For (nonzero-point) questions in [Type 3] [Type 4] you only need to submit your own written solution, but we encourage you to experiment with LLMs on some of them.
- If you attempt the extra credit problems [EC] using LLMs, include your program trace in the beginning of the writeup document.

For reference, here is everything you need to hand in for the first half of the PDF report `a1-writeup.pdf`.

- **Problem 1:** 1.2.1[Type 1] , 1.2.2[Type 2] , 1.3.1[Type 1] , 1.3.2[Type 3] , 1.3.4[Type 4]
- **Problem 2:** 2.1.2[Type 3] , 2.2.1[Type 3] , 2.2.2[Type 3] , 2.2.3[Type 2]
- **Problem 3:** 3.1[Type 4] , 3.2[Type 4]

Useful prompts

You could start by naively copy-pasting the question and the context as the prompt, and try to improve the generated answers by trial and error. Raw LaTeX dumps are made available for the written questions to facilitate this process.

- `https://uoft-csc413.github.io/2023/assets/assignments/a1_raw_latex_dump.tex`
- `https://uoft-csc413.github.io/2023/assets/assignments/a1_macros.tex`

1 Linear Regression (2.5pts)

The reading on linear regression located at `https://uoft-csc413.github.io/2023/assets/readings/L01a.pdf` may be useful for this question.

Given n pairs of input data with d features and scalar label $(\mathbf{x}_i, t_i) \in \mathbb{R}^d \times \mathbb{R}$, we wish to find a linear model $f(\mathbf{x}) = \hat{\mathbf{w}}^\top \mathbf{x}$ with $\hat{\mathbf{w}} \in \mathbb{R}^d$ that minimizes the squared error of prediction on the training samples defined below. This is known as an empirical risk minimizer. For concise

notation, denote the data matrix $X \in \mathbb{R}^{n \times d}$ and the corresponding label vector $\mathbf{t} \in \mathbb{R}^n$. The training objective is to minimize the following loss:

$$\min_{\hat{\mathbf{w}}} \frac{1}{n} \sum_{i=1}^{n} (\hat{\mathbf{w}}^\top \mathbf{x}_i - t_i)^2 = \min_{\hat{\mathbf{w}}} \frac{1}{n} \|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2.$$

We assume X is full rank: $X^\top X$ is invertible when $n > d$, and $XX^\top$ is invertible otherwise. Note that when $d > n$, the problem is *underdetermined*, i.e. there are less training samples than parameters to be learned. This is analogous to learning an *overparameterized* model, which is common when training of deep neural networks.

1.1 Deriving the Gradient [0pt] [Type 1]

Write down the gradient of the loss w.r.t. the learned parameter vector $\hat{\mathbf{w}}$.

1.2 Underparameterized Model

1.2.1 [0.5pt] [Type 1]

First consider the underparameterized $d < n$ case. Show that the solution obtained by gradient descent is $\hat{\mathbf{w}} = (X^\top X)^{-1} X^\top \mathbf{t}$, assuming training converges. Show your work.

1.2.2 [0.5pt] [Type 2]

Now consider the case of noisy linear regression. The training labels $t_i = \mathbf{w}^{*\top} \mathbf{x}_i + \epsilon_i$ are generated by a ground truth linear target function, where the noise term, ϵ_i, is generated independently with zero mean and variance σ^2. The final training error can be derived as a function of X and $\boldsymbol{\epsilon}$, as:

$$\textit{Error} = \frac{1}{n} ||(X(X^\top X)^{-1} X^\top - I)\boldsymbol{\epsilon}||_2^2,$$

Show this is true by substituting your answer from the previous question into $\frac{1}{n}\|X\hat{\mathbf{w}} - \mathbf{t}|_2^2$. Also, find the expectation of the above training error in terms of n, d and σ.

Hints: you might find the cyclic property [7] *of trace useful.*

1.3 Overparameterized Model

1.3.1 [0.5pt] [Type 1]

Now consider the overparameterized $d > n$ case. We first illustrate that there exist multiple empirical risk minimizers. For simplicity we let $n = 1$ and $d = 2$. Choose $\mathbf{x}_1 = [1; 1]$ and $t_1 = 3$, i.e. the one data point and all possible $\hat{\mathbf{w}}$ lie on a 2D plane. Show that there exists infinitely many $\hat{\mathbf{w}}$ satisfying $\hat{\mathbf{w}}^\top \mathbf{x}_1 = y_1$ on a real line. Write down the equation of the line.

[7] https://en.wikipedia.org/wiki/Trace_(linear_algebra)#Cyclic_property

1.3.2 [0.5pt] [Type 3] [EC]

Now, let's generalize the previous 2D case to the general $d > n$. Show that gradient descent from zero initialization i.e. $\hat{\mathbf{w}}(0) = 0$ finds a unique minimizer if it converges. Show that the solution by gradient decent is $\hat{\mathbf{w}} = X^\top (XX^\top)^{-1}\mathbf{t}$. Show your work.

Hints: You can assume that the gradient is spanned by the rows of X and write $\hat{\mathbf{w}} = X^\top \mathbf{a}$ for some $\mathbf{a} \in \mathbb{R}^n$.

1.3.3 [0pt] [Type 3]

Repeat part 1.2.2 for the overparameterized case.

1.3.4 [0.5pt] [Type 4]

Visualize and compare underparameterized with overparameterized polynomial regression: `https://colab.research.google.com/github/uoft-csc413/2023/blob/master/assets/assignments/LS_polynomial_regression.ipynb` Include your code snippets for the `fit_poly` function in the write-up. Does overparameterization (higher degree polynomial) always lead to overfitting, i.e. larger test error?

1.3.5 [0pt] [Type 4]

Give n_1, n_2 with $n_1 \leq n_2$, and fixed dimension d for which $L_2 \geq L_1$, i.e. the loss with n_2 data points is greater than loss with n_1 data points. Explain the underlying phenomenon. Be sure to also include the error values L_1 and L_2 or provide visualization in your solution.
Hints: use your code to experiment with relevant parameters, then vary to find region and report one such setting.

2 Backpropagation (4pts)

This question helps you to understand the underlying mechanism of back-propagation. You need to have a clear understanding of what happens during the forward pass and backward pass and be able to reason about the time complexity and space complexity of your neural network. Moreover, you will learn a commonly used trick to compute the gradient norm efficiently without explicitly writing down the whole Jacobian matrix.

Note: The reading on backpropagation located at `https://uoft-csc413.github.io/2023/assets/readings/L02b.pdf` may be useful for this question.

2.1 Automatic Differentiation

Consider a neural network defined with the following procedure:

$$
\begin{aligned}
\mathbf{z}_1 &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\
\mathbf{h}_1 &= \text{ReLU}(\mathbf{z}_1) \\
\mathbf{z}_2 &= \mathbf{W}^{(2)}\mathbf{x} + \mathbf{b}^{(2)} \\
\mathbf{h}_2 &= \sigma(\mathbf{z}_2) \\
\mathbf{g} &= \mathbf{h}_1 \circ \mathbf{h}_2 \\
\mathbf{y} &= \mathbf{W}^{(3)}\mathbf{g} + \mathbf{W}^{(4)}\mathbf{x}, \\
\mathbf{y}' &= \text{softmax}(\mathbf{y}) \\
\mathcal{S} &= \sum_{k=1}^{N} \mathbb{I}(t = k) \log(\mathbf{y}'_k) \\
\mathcal{J} &= -\mathcal{S}
\end{aligned}
$$

for input $\mathbf{x}$ with class label t where $\text{ReLU}(\mathbf{z}) = \max(\mathbf{z}, 0)$ denotes the ReLU activation function, $\sigma(\mathbf{z}) = \frac{1}{1+e^{-\mathbf{z}}}$ denotes the Sigmoid activation function, both applied elementwise, and $\text{softmax}(\mathbf{y}) = \frac{\exp(\mathbf{y})}{\sum_{i=1}^{N}\exp(\mathbf{y}_i)}$. Here, $\circ$ denotes element-wise multiplication.

2.1.1 Computational Graph [0pt] [Type 4]

Draw the computation graph relating $\mathbf{x}$, t, $\mathbf{z}_1$, $\mathbf{z}_2$, $\mathbf{h}_1$, $\mathbf{h}_2$, $\mathbf{g}$, $\mathbf{y}$, $\mathbf{y}'$, $\mathcal{S}$ and $\mathcal{J}$.

2.1.2 Backward Pass [1pt] [Type 3]

Derive the backprop equations for computing $\bar{\mathbf{x}} = \frac{\partial \mathcal{J}}{\partial \mathbf{x}}^\top$, one variable at a time, similar to the vectorized backward pass derived in Lec 2.

Hints: Be careful about the transpose and shape! Assume all vectors (including error vector) are column vector and all Jacobian matrices adopt numerator-layout notation [8]*. You can use softmax′(*$\mathbf{y}$*) for the Jacobian matrix of softmax.*

2.2 Gradient Norm Computation

Many deep learning algorithms require you to compute the L^2 norm of the gradient of a loss function with respect to the model parameters for every example in a minibatch. Unfortunately, most differentiation functionality provided by most software frameworks (Tensorflow, PyTorch) does not support computing gradients for individual samples in a minibatch. Instead, they only give one gradient per minibatch that aggregates individual gradients for you. A naive way to get the per-example gradient norm is to use a batch size of 1 and repeat the back-propagation N times, where N is the minibatch size. After that, you can compute the L^2 norm of each gradient vector. As you can imagine, this approach is very inefficient. It can not exploit the parallelism of minibatch operations provided by the framework.

[8] Numerator-layout notation: `https://en.wikipedia.org/wiki/Matrix_calculus#Numerator-layout_notation`

In this question, we will investigate a more efficient way to compute the per-example gradient norm and reason about its complexity compared to the naive method. For simplicity, let us consider the following two-layer neural network.

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$$
$$\mathbf{h} = \text{ReLU}(\mathbf{z})$$
$$\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h},$$

where $\mathbf{W}^{(1)} = \begin{pmatrix} 1 & 2 & 1 \\ -2 & 1 & 0 \\ 1 & -2 & -1 \end{pmatrix}$ and $\mathbf{W}^{(2)} = \begin{pmatrix} -2 & 4 & 1 \\ 1 & -2 & -3 \\ -3 & 4 & 6 \end{pmatrix}$.

2.2.1 Naive Computation [1pt] [Type 3]

Let us assume the input $x = \begin{pmatrix} 1 & 3 & 1 \end{pmatrix}^\top$ and the error vector $\overline{\mathbf{y}} = \frac{\partial \mathcal{J}}{\partial \mathbf{y}}^\top = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^\top$. In this question, write down the Jacobian matrix (numerical value) $\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}}$ and $\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(2)}}$ using back-propagation. Then, compute the square of Frobenius Norm of the two Jacobian matrices, $\|A\|_{\text{F}}^2$. The square of Frobenius norm of a matrix A is defined as follows:

$$\|A\|_{\text{F}}^2 = \sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2 = \text{trace}\left(A^\top A\right)$$

Hints: Be careful about the transpose. Show all your work for partial marks.

2.2.2 Efficient Computation [0.5pt] [Type 3]

Notice that weight Jacobian can be expressed as the outer product of the error vector and activation $\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}} = (\overline{\mathbf{z}}\mathbf{x}^\top)^\top$ and $\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(2)}} = (\overline{\mathbf{y}}\mathbf{h}^\top)^\top$. We can compute the Jacobian norm more efficiently using the following trick:

$$\begin{aligned}
\|\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}}\|_{\text{F}}^2 &= \text{trace}\left(\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}}^\top \frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}}\right) && \text{(Definition)} \\
&= \text{trace}\left(\overline{\mathbf{z}}\mathbf{x}^\top\mathbf{x}\overline{\mathbf{z}}^\top\right) \\
&= \text{trace}\left(\mathbf{x}^\top\mathbf{x}\overline{\mathbf{z}}^\top\overline{\mathbf{z}}\right) && \text{(Cyclic Property of Trace)} \\
&= \left(\mathbf{x}^\top\mathbf{x}\right)\left(\overline{\mathbf{z}}^\top\overline{\mathbf{z}}\right) && \text{(Scalar Multiplication)} \\
&= \|\mathbf{x}\|_2^2\|\overline{\mathbf{z}}\|_2^2
\end{aligned}$$

Compute the **square** of the Frobenius Norm of the two Jacobian matrices by plugging the value into the above trick.
Hints: Verify the solution is the same as naive computation. Show all your work for partial marks.

2.2.3 Complexity Analysis [1.5pt] [Type 2]

Now, let us consider a general neural network with $K-1$ hidden layers (K weight matrices). All input units, output units, and hidden units have a dimension of D. Assume we have N input vectors. How many scalar multiplications T (integer) do we need to compute the per-example gradient norm using naive and efficient computation, respectively? And, what is the memory cost M (big $\mathcal{O}$ notation)?

For simplicity, you can ignore the activation function and loss function computation. You can assume the network does not have a bias term. You can also assume there are no in-place operations. Please fill up the table below.

	T (Naive)	T (Efficient)	M (Naive)	M (Efficient)
Forward Pass				
Backward Pass				
Gradient Norm Computation				

Hints: The forward pass computes all the activations and needs memory to store model parameters and activations. The backward pass computes all the error vectors. Moreover, you also need to compute the parameter's gradient in naive computation. During the Gradient Norm Computation, the naive method needs to square the gradient before aggregation. In contrast, the efficient method relies on the trick. Thinking about the following questions may be helpful. 1) Do we need to store all activations in the forward pass? 2) Do we need to store all error vectors in the backward pass? 3) Why standard backward pass is twice more expensive than the forward pass? Don't forget to consider K and N in your answer.

2.3 Inner product of Jacobian: JVP and VJP [0pt] [Type 3]

A more general case of computing the gradient norm is to compute the inner product of the Jacobian matrices computed using two different examples. Let f_1, f_2 and y_1, y_2 be the final outputs and layer outputs of two different examples respectively. The inner product Θ of Jacobian matrices of layer parameterized by θ is defined as:

$$\Theta_\theta(f_1, f_2) := \frac{\partial f_1}{\partial \theta}\frac{\partial f_2}{\partial \theta}^\top = \frac{\partial f_1}{\partial y_1}\frac{\partial y_1}{\partial \theta}\frac{\partial y_2}{\partial \theta}^\top\frac{\partial f_2}{\partial y_2}^\top = \underbrace{\frac{\partial f_1}{\partial y_1}}_{\mathbf{O}\times\mathbf{Y}}\underbrace{\frac{\partial y_1}{\partial \theta}}_{\mathbf{Y}\times\mathbf{P}}\underbrace{\frac{\partial y_2}{\partial \theta}^\top}_{\mathbf{P}\times\mathbf{Y}}\underbrace{\frac{\partial f_2}{\partial y_2}^\top}_{\mathbf{Y}\times\mathbf{O}},$$

Where $\mathbf{O}, \mathbf{Y}, \mathbf{P}$ represent the dimension of the final output, layer output, model parameter respectively. How to formulate the above computation using Jacobian Vector Product (JVP) and Vector Jacobian Product (VJP)? What are the computation cost using the following three ways of contracting the above equation?

(a) Outside-in: $M_1M_2M_3M_4 = ((M_1M_2)(M_3M_4))$
(b) Left-to-right and right-to-left: $M_1M_2M_3M_4 = (((M_1M_2)M_3)M_4) = (M_1(M_2(M_3M_4)))$
(c) Inside-out-left and inside-out-right: $M_1M_2M_3M_4 = ((M_1(M_2M_3))M_4) = (M_1((M_2M_3)M_4))$

3 Hard-Coding Networks (2.5pts) [Type 4]

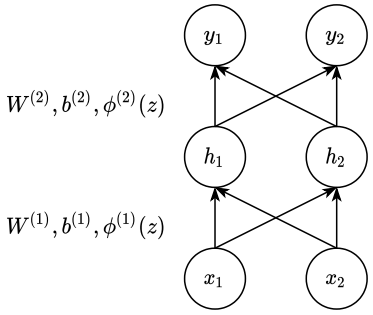
Can we use neural networks to tackle coding problems? Yes! In this question, you will build a neural network to find the k^{th} smallest number from a list using two different approaches: sorting

and counting (Optional). You will start by constructing a two-layer perceptron "Sort_2" to sort two numbers and then use it as a building block to perform your favorite sorting algorithm (e.g., Bubble Sort, Merge Sort). Finally, you will output the k^{th} element from the sorted list as the final answer.

Note: Before doing this problem, you need to have a basic understanding of the key components of neural networks (e.g., weights, activation functions). The reading on multilayer perceptrons located at `https://uoft-csc413.github.io/2023/assets/readings/L02a.pdf` may be useful.

3.1 Sort two numbers [1pt]

In this problem, you need to find a set of weights and bias for a two-layer perceptron "Sort_2" that sorts two numbers. The network takes a pair of numbers (x_1, x_2) as input and output a sorted pair (y_1, y_2), where $y_1 \leq y_2$. You may assume the two numbers are distinct and positive for simplicity. You will use the following architecture:



Please specify the weights and activation functions for your network. Your answer should include:

- Two weight matrices: $\mathbf{W}^{(1)}$, $\mathbf{W}^{(2)} \in \mathbb{R}^{2\times 2}$
- Two bias vector: $\mathbf{b}^{(1)}$, $\mathbf{b}^{(2)} \in \mathbb{R}^2$
- Two activation functions: $\phi^{(1)}(z)$, $\phi^{(2)}(z)$

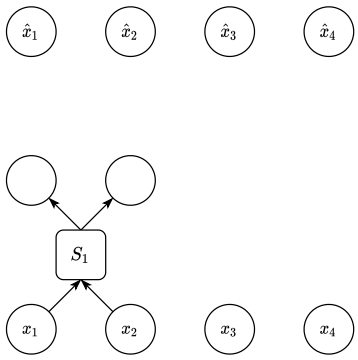
You do not need to show your work.

Hints: Sorting two numbers is equivalent to finding the min and max of two numbers.

$$\max(x_1, x_2) = \frac{1}{2}(x_1 + x_2) + \frac{1}{2}|x_1 - x_2|, \qquad \min(x_1, x_2) = \frac{1}{2}(x_1 + x_2) - \frac{1}{2}|x_1 - x_2|$$

3.2 Perform Sort [1.5pt][EC]

Draw a computation graph to show how to implement a sorting function $\hat{f} : \mathbb{R}^4 \to \mathbb{R}^4$ where $\hat{f}(x_1, x_2, x_3, x_4) = (\hat{x}_1, \hat{x}_2, \hat{x}_3, \hat{x}_4)$ where $(\hat{x}_1, \hat{x}_2, \hat{x}_3, \hat{x}_4)$ is (x_1, x_2, x_3, x_4) in sorted order. Let us assume $\hat{x}_1 \leq \hat{x}_2 \leq \hat{x}_3 \leq \hat{x}_4$ and x_1, x_2, x_3, x_4 are positive and distinct. Implement $\hat{f}$ using your favourite sorting algorithms (e.g. Bubble Sort, Merge Sort). Let us denote the "Sort_2" module as S, please complete the following computation graph. Your answer does not need to give the label for intermediate nodes, but make sure to index the "Sort_2" module.



Hints: Bubble Sort needs 6 "Sort_2" blocks, while Merge Sort needs 5 "Sort_2" blocks.

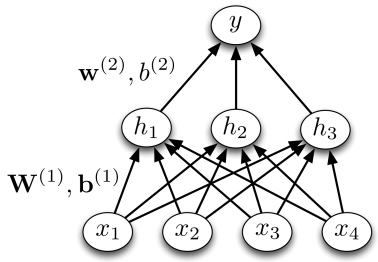
3.3 Find the k^{th} smallest number [0pt]

Based on your sorting network, you may want to add a new layer to output your final result (k^{th} smallest number). Please give the weight $\mathbf{W}^{(3)}$ for this output layer when $k = 3$.

Hints: $\mathbf{W}^{(3)} \in \mathbb{R}^{1\times 4}$.

3.4 Counting Network [0pt]

The idea of using a counting network to find the k^{th} smallest number is to build a neural network that can determine the rank of each number and output the number with the correct rank. Specifically, the counting network will count how many elements in a list are less than a value of interest. And you will apply the counting network to all numbers in the given list to determine their rank. Finally, you will use another layer to output the number with the correct rank.
The counting network has the following architecture, where y is the rank of x_1 in a list containing x_1, x_2, x_3, x_4.



Please specify the weights and activation functions for your counting network. Draw a diagram to show how you will use the counting network and give a set of weights and biases for the final layer to find the k^{th} smallest number. In other words, repeat the process of sections 1.1, 1.2, 1.3 using the counting idea.

Hints: You may find the following two activation functions useful.
1) Hard threshold activation function:

$$\phi(z) = \mathbb{I}(z \geq 0) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

2) Indicator activation function:

$$\phi(z) = \mathbb{I}(z \in [-1, 1]) = \begin{cases} 1 & \text{if } z \in [-1, 1] \\ 0 & \text{otherwise} \end{cases}$$

Programming Assignment

What you have to submit for this part

See the top of this handout for submission directions. Here are the requirements.

- The zero point questions (in black below) will not be graded, but you are more than welcome to include your answers for these as well in the submission.
- For (nonzero-point) writing questions labeled [Type 1] [Type 2] you need to submit (*i*) **the program trace**, (*ii*) **critique of the model output**, (*iii*) **your own solution**. You will receive 0 point on the question if you do not include the program trace, even if you solve the problem by yourself. **Make sure you cite the model properly**.
- For (nonzero-point) coding questions labeled [Type 1] [Type 2] you need to submit (*i*) **the LLM program trace** (in the writeup PDF), (*ii*) **your own solution**.
- For (nonzero-point) questions in [Type 3] [Type 4] you only need to submit your own solution, but we encourage you to experiment with LLMs on some of them.
- If you attempt the extra credit problems [EC] using LLMs, include your program trace in the beginning of the writeup document.

For reference, here is everything you need to hand in:

- This is the second half of your PDF report `a1-writeup.pdf`. Please include the solutions to the following problems. You may choose to export `a1-code.ipynb` as a PDF and attach it to the first half of `a1-writeup.pdf`. Do not forget to append the following program traces/screenshots to the end of `a1-writeup.pdf`:
  - **Question 4:** 4.2[Type 1] , 4.3[Type 2] , LLM program trace (screenshot) for `loss_GloVe()` in 4.4[Type 2] and `grad_GloVe()` in 4.5[Type 2] , output plot in 4.5.
  - **Question 5:** 5.1[Type 2] , 5.2[Type 1] .
  - **Question 6:** LLM program trace for `compute_loss()` in 6.1[Type 2] and `back_propagate()` in 6.2[Type 2] , output of `print_gradients()` in 6.3[Type 4]
  - **Question 7:** LLM program trace for `weat_association_score()` in 7.1[Type 2] and output, 1-word subsets with outputs in 7.3.1[Type 4] , 7.3.2[Type 3] .
- Your code file `a1-code.ipynb`

Word embeddings

In this assignment, we will learn about word embeddings and make neural networks learn about words. We could try to match statistics about the words, or we could train a network that takes a sequence of words as input and learns to predict the word that comes next.

This assignment will ask you to implement a linear embedding and then the backpropagation computations for a neural language model and then run some experiments to analyze the learned representation. The amount of code you have to write is very short but each line will require you to think very carefully. You will need to derive the updates mathematically, and then implement them using matrix and vector operations in NumPy.

Starter code and data

The starter code is at `https://colab.research.google.com/github/uoft-csc413/2023/blob/master/assets/assignments/a1-code.ipynb`.

The starter helper function will download the specific the dataset from `http://www.cs.toronto.edu/~jba/a1_data.tar.gz`. Look at the file `raw_sentences.txt`. It contains the sentences that we will be using for this assignment. These sentences are fairly simple ones and cover a vocabulary of only 250 words (+ 1 special `[MASK]` token word).

We have already extracted the 4-grams from this dataset and divided them into training, validation, and test sets. To inspect this data, run the following within IPython:

```
import pickle
data = pickle.load(open('data.pk', 'rb'))
```

Now `data` is a Python dict which contains the vocabulary, as well as the inputs and targets for all three splits of the data. `data['vocab']` is a list of the 251 words in the dictionary; `data['vocab'][0]` is the word with index 0, and so on. `data['train_inputs']` is a $372,500 \times 4$ matrix where each row gives the indices of the 4 consecutive context words for one of the $372,500$ training cases. The validation and test sets are handled analogously.

Now look at the notebook ipynb file `a1-code.ipynb`, which contains the starter code for the assignment. Even though you only have to modify a few specific locations in the code, you may want to read through this code before starting the assignment.

4 Linear Embedding – GloVe (3pts)

In this section we will be implementing a simplified version of GloVe [Jeffrey Pennington and Manning]. Given a corpus with V distinct words, we define the co-occurrence matrix $X \in \mathbb{N}^{V \times V}$ with entries X_{ij} representing the frequency of the i-th word and j-th word in the corpus appearing in the same *context* - in our case the adjacent words. The co-occurrence matrix can be *symmetric* (*i.e.*, $X_{ij} = X_{ji}$) if the order of the words do not matter, or *asymmetric* (*i.e.*, $X_{ij} \neq X_{ji}$) if we wish to distinguish the counts for when i-th word appears before j-th word. GloVe aims to find a d-dimensional embedding of the words that preserves properties of the co-occurrence matrix by representing the i-th word with two d-dimensional vectors $\mathbf{w}_i, \tilde{\mathbf{w}}_i \in \mathbb{R}^d$, as well as two scalar biases $b_i, \tilde{b}_i \in \mathbb{R}$. Typically we have the dimension of the embedding d much smaller than the number of words V. This objective can be written as [9]:

$$L(\{\mathbf{w}_i, \tilde{\mathbf{w}}_i, b_i, \tilde{b}_i\}_{i=1}^V) = \sum_{i,j=1}^{V} (\mathbf{w}_i^\top \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij})^2 \tag{1}$$

When the bias terms are omitted and we tie the two embedding vectors $\mathbf{w}_i = \tilde{\mathbf{w}}_i$, then GloVe corresponds to finding a rank-d symmetric factorization of the co-occurrence matrix.

4.1 GloVe Parameter Count [0pt] [Type 1]

Given the vocabulary size V and embedding dimensionality d, how many trainable parameters does the GloVe model have? Note that each word in the vocabulary is associated with 2 embedding vectors and 2 biases.

[9] We have simplified the objective by omitting the weighting function. For the complete algorithm please see [Jeffrey Pennington and Manning]

4.2 Expression for the vectorized loss function [0.5pt] [Type 1]

In practice, we concatenate the V embedding vectors into matrices $\mathbf{W}, \tilde{\mathbf{W}} \in \mathbb{R}^{V \times d}$ and bias (column) vectors $\mathbf{b}, \tilde{\mathbf{b}} \in \mathbb{R}^{V}$, where V denotes the number of distinct words as described in the introduction. Rewrite the loss function L (Eq. 1) in a vectorized format in terms of $\mathbf{W}, \tilde{\mathbf{W}}, \mathbf{b}, \tilde{\mathbf{b}}, X$. You are allowed to use elementwise operations such as addition and subtraction as well as matrix operations such as the Frobenius norm and/or trace operator in your answer.

Hint: Use the all-ones column vector $\mathbf{1} = [1 \dots 1]^T \in \mathbb{R}^V$. *You can assume the bias vectors are column vectors, i.e. implicitly a matrix with* V *rows and 1 column:* $\mathbf{b}, \tilde{\mathbf{b}} \in \mathbb{R}^{V \times 1}$

4.3 Expression for the vectorized gradient $\nabla_{\mathbf{W}} L$ [0.5pt] [Type 2]

Write the vectorized expression for $\nabla_{\mathbf{W}} L$, the gradient of the loss function L with respect to the embedding matrix $\mathbf{W}$. The gradient should be a function of $\mathbf{W}, \tilde{\mathbf{W}}, \mathbf{b}, \tilde{\mathbf{b}}, X$.

Hint: Make sure that the shape of the gradient is equivalent to the shape of the matrix. You can use the all-ones vector as in the previous question.

Hint: Equation (119) in the matrix cookbook[10] *may be useful. Be careful with transpose.*

4.4 Implement Vectorized Loss Function [1pt] [Type 2]

Implement the `loss_GloVe()` function of GloVe in `a1-code.ipynb`. Look for the `## YOUR CODE HERE ##` comment for where to complete the code. Note that you need to implement both the loss for an *asymmetric* model (from your answer in question 4.2) and the loss for a *symmetric* model which uses the same embedding matrix $\mathbf{W}$ and bias vector $\mathbf{b}$ for the first and second word in the co-occurrence, i.e. $\tilde{\mathbf{W}} = \mathbf{W}$ and $\tilde{\mathbf{b}} = \mathbf{b}$ in the original loss.

Hint: You may take advantage of NumPy's broadcasting feature[11] *for the bias vectors*

4.5 Implement the gradient update of GloVe [1pt] [Type 2]

Implement the `grad_GloVe()` function which computes the gradient of GloVe in `a1-code.ipynb`. Look for the `## YOUR CODE HERE ##` comment for where to complete the code. Again, note that you need to implement the gradient for both the symmetric and asymmetric models.

We added a gradient checker function using finite difference called `check_GloVe_gradients()`. You can run the specified cell in the notebook to check your gradient implementation for both the symmetric and asymmetric models before moving forward.

Once you have implemented the gradient, run the following cell marked by the comment `### TODO: Run this cell ###` in order to train an asymmetric and symmetric GloVe model. The code will plot a figure containing the training and validation loss for the two models over the course of training. **Include this plot in your write up**.

Hint: In the symmetric model case, you can use what you have derived for the asymmetric model case. For example, consider a function $f(\mathbf{a}, \mathbf{b}) = \mathbf{a}^\top \mathbf{b}$, *where* $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$. *If we define* $\mathbf{a} = \mathbf{x}$ *and* $\mathbf{b} = \mathbf{x}$, *where* $\mathbf{x} \in \mathbb{R}^d$, *then*

$$\begin{aligned} \nabla_{\mathbf{x}} f &= \nabla_{\mathbf{a}} f + \nabla_{\mathbf{b}} f \\ &= \mathbf{b} + \mathbf{a} \\ &= \mathbf{x} + \mathbf{x} \\ &= 2\mathbf{x} \end{aligned}$$

[10] `https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf`
[11] https://numpy.org/doc/stable/user/basics.broadcasting.html

4.6 Effects of a buggy implementation [0pt] [Type 2]

Suppose that during the implementation, you initialized the weight embedding matrix $\mathbf{W}$ and $\tilde{\mathbf{W}}$ with the same initial values (i.e., $\mathbf{W} = \tilde{\mathbf{W}} = \mathbf{W}_0$). The bias vectors were also initialized the same, i.e., $\mathbf{b} = \tilde{\mathbf{b}} = \mathbf{b}_0$. Assume also that in this case, the co-occurrence matrix is also symmetric: $X_{ij} = X_{ji}$

What will happen to the values of $\mathbf{W}$ and $\tilde{\mathbf{W}}$ over the course of training? Will they stay equal to each other, or diverge from each other? Explain your answer briefly.

Hint: Consider the gradient $\nabla_{\mathbf{W}} L$ *versus* $\nabla_{\tilde{\mathbf{W}}} L$

4.7 Effects of embedding dimension [0pt] [Type 3]

Train the both the symmetric and asymmetric GloVe model with varying dimensionality d. Comment on the results:

1. Which d leads to optimal validation performance for the asymmetric and symmetric models?
2. Why does / doesn't larger d always lead to better validation error?
3. Which model is performing better (asymmetric or symmetric), and why?

5 Neural Language Model Network architecture (2pt)

In this assignment, we will train a neural language model like the one we covered in lecture and as in Bengio et al. [2003]. However, we will modify the architecture slightly, inspired by the Masked Language Modeling (MLM) objective introduced in BERT [Devlin et al., 2018]. The network takes in N consecutive words, where one of the words is replaced with a `[MASK]` token[12]. The aim of the network is to predict the masked word in the corresponding output location. See Figure 1 for the diagram of this architecture.

The network consists of an input layer, embedding layer, hidden layer and output layer. The input consists of a sequence of N consecutive words, with each word given as integer valued indices. (*e.g.*, the 250 words in our dictionary are arbitrarily assigned integer values from 0 to 249.) The embedding layer maps each word to its corresponding vector representation. Each of the N context words are mapped independently using the same `word_embedding_weights` matrix. The embedding layer has $N \times D$ units, where D is the embedding dimension of a single word. The embedding layer is fully connected to the hidden layer with H units, which uses a logistic nonlinearity. The hidden layer in turn is connected to the logits output layer, which has $N \times V$ units. Finally, softmax over V logit output units is applied to each consecutive V logit output units, where V is the number of words in the dictionary (including the `[MASK]` token).[13]

5.1 Number of parameters in neural network model [1pt] [Type 2]

The trainable parameters of the model consist of 3 weight matrices and 2 sets of biases. What is the total number of trainable parameters in the model, as a function of V, N, D, H? In the diagram given above, which part of the model (i.e., `word_embbeding_weights`, `embed_to_hid_weights`, `hid_to_output_weights`, `hid_bias`, or `output_bias`) has the largest number of trainable parameters if we have the constraint that $V \gg H > D > N$?[14] Explain your reasoning.

[12] In the original BERT paper, they mask out 15% of the tokens randomly.
[13] For simplicity we will include the `[MASK]` token in the output softmax as well.
[14] The symbol $\gg$ means "much greater than"

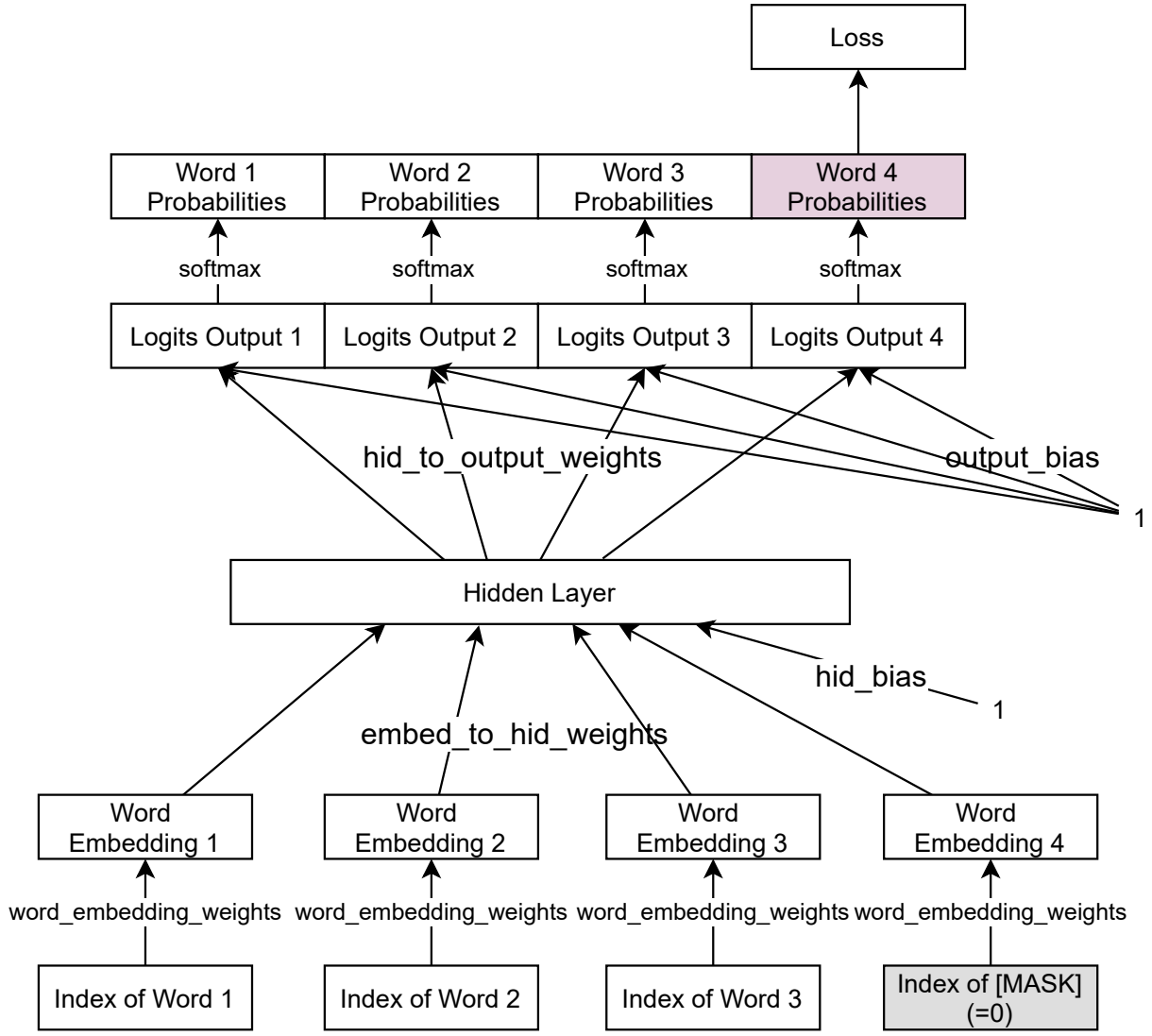


Figure 1: A simplified architecture with N words input and N words output. During training, we mask out one of the input words by replacing it with a `[MASK]` token, and try to predict the masked out word in the corresponding position in the output. Only that output position is used in the cross entropy loss.

5.2 Number of parameters in n-gram model [1pt] [Type 1]

Another method for predicting the next words is an *n-gram model*, which was mentioned in Lecture 3[15]. If we wanted to use an n-gram model with the same context length $N-1$ as our network[16], we'd need to store the counts of all possible N-grams. If we stored all the counts explicitly and suppose that we have V words in the dictionary, how many entries would this table have?

[15] https://uoft-csc413.github.io/2022/assets/slides/lec03.pdf
[16] Since we mask 1 of the N words in our input

5.3 Comparing neural network and n-gram model scaling [0pt] [Type 2]

How do the parameters in the neural network model scale with the number of context words N versus how the number of entries in the n-gram model scale with N? Which model has a more compact representation for the words?

6 Training the Neural Network (2pts)

In this part, you will learn to implement and train the neural language model from Figure 1. As described in the previous section, during training, we randomly sample one of the N context words to replace with a `[MASK]` token. The goal is for the network to predict the word that was masked, at the corresponding output word position. In our implementation, this `[MASK]` token is assigned the index 0 in our dictionary. In practice, it is more efficient to take advantage of parallel computing hardware, such as GPUs, to speed up training. Instead of predicting one word at a time, we achieve parallelism by lumping N output words into a single output vector that can be computed by a single matrix product. Now, the hidden-to-output weight matrix `hid_to_output_weights` has the shape $NV \times H$, as the output layer has NV neurons, where the first V output units are for predicting the first word, then the next V are for predicting the second word, and so on. Note here that the softmax is applied in chunks of V as well, to give a valid probability distribution over the V words[17]. Only the output word positions that were masked in the input are included in the cross entropy loss calculation:

$$C = -\sum_{i}^{B}\sum_{n}^{N}\sum_{v}^{V} m_n^{(i)}(t_{v+nV}^{(i)} \log y_{v+nV}^{(i)}) \tag{2}$$

Where:

- $y_{v+nV}^{(i)}$ denotes the output probability prediction from the neural network for the i-th training example for the word v in the n-th output word. Denoting z as the output logits, we define the output probability y as a softmax on z over contiguous chunks of V units (see Figure 1):

$$y_{v+nV}^{(i)} = \frac{e^{z_{v+nV}^{(i)}}}{\sum_l^V e^{z_{l+nV}^{(i)}}} \tag{3}$$

- $t_{v+nV}^{(i)} \in \{0, 1\}$ is 1 if for the i-th training example, the word v is the n-th word in context
- $m_n^{(i)} \in \{0, 1\}$ is a mask that is set to 1 if we are predicting the n-th word position for the i-th example (because we had masked that word in the input), and 0 otherwise

Now, you are ready to complete the implementation in the notebook `https://colab.research.google.com/github/uoft-csc413/2023/blob/master/assets/assignments/a1-code.ipynb`, you will implement a method which computes the gradient using backpropagation. The `Model` class contains several important methods used in training:

- `compute_activations` computes the activations of all units on a given input batch

[17]For simplicity we also include the `[MASK]` token as one of the possible prediction even though we know the target should not be this token

- `compute_loss_derivative` computes the gradient with respect to the output logits $\frac{\partial C}{\partial z}$
- `evaluate` computes the average cross-entropy loss for a given set of inputs and targets

You will need to complete the implementation of two additional methods to complete the training, and print the outputs of the gradients.

6.1 Implement Vectorized Loss [0.5pt] [Type 2]

Implement a vectorized `compute_loss` function, which computes the total cross-entropy loss on a mini-batch according to Eq. 2. Look for the `## YOUR CODE HERE ##` comment for where to complete the code. The docstring provides a description of the inputs to the function.

6.2 Implement gradient with respect to parameters [1pt] [Type 2]

`back_propagate` is the function which computes the gradient of the loss with respect to model parameters using backpropagation. It uses the derivatives computed by `compute_loss_derivative`. Some parts are already filled in for you, but you need to compute the matrices of derivatives for `embed_to_hid_weights` and `output_bias`. These matrices have the same sizes as the parameter matrices. Look for the `## YOUR CODE HERE ##` comment for where to complete the code.

In order to implement backpropagation efficiently, you need to express the computations in terms of matrix operations, rather than `for` loops. You should first work through the derivatives on pencil and paper. First, apply the chain rule to compute the derivatives with respect to individual units, weights, and biases. Next, take the formulas you've derived, and express them in matrix form. You should be able to express all of the required computations using only matrix multiplication, matrix transpose, and element-wise operations — no `for` loops! If you want inspiration, read through the code for `Model.compute_activations` and try to understand how the matrix operations correspond to the computations performed by all the units in the network.

Hints: Your implementations should also be similar to `hid_to_output_weights_grad`, `hid_bias_grad` *in the same function call.*

6.3 Print the gradients [0.5pt][Type 4]

To make your life easier, we have provided the routine `check_gradients`, which checks your gradients using finite differences. You should make sure this check passes (prints OK for the various parameters) before continuing with the assignment. Once `check_gradients` passes, call `print_gradients` and include its output in your write-up.

6.4 Run model training [0 pt]

Once you've implemented the gradient computation, you'll need to train the model. The function `train` in `a1-code.ipynb` implements the main training procedure. It takes two arguments:

- `embedding_dim`: The number of dimensions in the distributed representation.
- `num_hid`: The number of hidden units

For example, execute the following:

```
model = train(16, 128)
```

As the model trains, the script prints out some numbers that tell you how well the training is going. It shows:

- The cross entropy on the last 100 mini-batches of the training set. This is shown after every 100 mini-batches.
- The cross entropy on the entire validation set every 1000 mini-batches of training.

At the end of training, this function shows the cross entropies on the training, validation and test sets. It will return a `Model` instance.

7 Bias in Word Embeddings (2pts)

Unfortunately, stereotypes and prejudices are often reflected in the outputs of natural language processing algorithms. For example, Google Translate is more likely to translate a non-English sentence to "*He* is a doctor" than "*She* is a doctor" when the sentence is ambiguous. In this section, you will explore how bias[18] enters natural language processing algorithms by implementing and analyzing a popular method for measuring bias in word embeddings.

7.1 WEAT method for detecting bias [1pt] [Type 2]

Word embedding models such as GloVe attempt to learn a vector space where semantically similar words are clustered close together. However, they have been shown to learn problematic associations, e.g. by embedding "man" more closely to "doctor" than "woman" (and vice versa for "nurse"). To detect such biases in word embeddings, Caliskan et al. [2017] introduced the Word Embedding Association Test (WEAT). The WEAT test measures whether two *target* word sets (e.g. {programmer, engineer, scientist, ...} and {nurse, teacher, librarian, ...}) have the same relative association to two *attribute* word sets (e.g. {man, male, ...} and {woman, female ...}).[19]

Formally, let A, B be two sets of attribute words. Then

$$s(w, A, B) = \text{mean}_{a \in A} \cos(\vec{w}, \vec{a}) - \text{mean}_{b \in B} \cos(\vec{w}, \vec{b}) \tag{4}$$

measures the association of a target word w with the attribute sets - for convenience, we will call this the WEAT association score. A positive score means that the word w is more associated with A, while a negative score means the opposite. For example, a WEAT association score of 1 in the following test $s(\text{“}programmer\text{”}, \{man\}, \{woman\}) = 1$, implies the "programmer" has a stronger association to $\{man\}$. For reference, the cosine similarity between two word vectors $\vec{a}$ and $\vec{b}$ is given by:

$$\cos(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{\| \vec{a} \| \| \vec{b} \|} \tag{5}$$

In the notebook, we have provided example target words (in sets X and Y) and attribute words (in sets A and B). You must implement the function `weat_association_score()` and compute the WEAT association score for each target word.

[18] In AI and machine learning, **bias** generally refers to prior information, a necessary prerequisite for intelligent action. However, bias can be problematic when it is derived from aspects of human culture known to lead to harmful behaviour, such as stereotypes and prejudices.

[19] There is an excellent blog on bias in word embeddings and the WEAT test at `https://developers.googleblog.com/2018/04/text-embedding-models-contain-bias.html`

7.2 Reasons for bias in word embeddings [0pt] [Type 1]

Based on the results of the WEAT test, do the pretrained word embeddings associate certain occupations with one gender more than another? What might cause word embedding models to learn certain stereotypes and prejudices? How might this be a problem in downstream applications?

7.3 Analyzing WEAT

While WEAT makes intuitive sense by asserting that closeness in the embedding space indicates greater similarity, more recent work Ethayarajh et al. [2019] has further analyzed the mathematical assertions and found some drawbacks this method. Analyzing edge cases is a good way to find logical inconsistencies with any algorithm, and WEAT in particular can behave strangely when A and B contain just one word each.

7.3.1 1-word subsets [0.5pts] [Type 4]

In the notebook, you are asked to find 1-word subsets of the original A and B that reverse the association between some of the occupations and the gendered attributes (change the sign of the WEAT score).

7.3.2 How word frequency affects embedding similarity [0.5pts] [Type 3] [EC]

Next, consider this fact about word embeddings, which has been verified empirically and theoretically: The squared norm of a word embedding is linear in the log probability of the word in the training corpus. In other words, the more common a word is in the training corpus, the larger the norm of its word embedding. Following this fact, we will show how one may exploit the WEAT association score for a specific word embedding model. Let us start with three word embedding vectors: a target word $\mathbf{w}_i$ and two attributes $\{\mathbf{w}_j\}$, $\{\mathbf{w}_k\}$.

$$\begin{aligned} s(\mathbf{w}_i, \{\mathbf{w}_j\}, \{\mathbf{w}_k\}) &= cos(\mathbf{w}_i, \mathbf{w}_j) - cos(\mathbf{w}_i, \mathbf{w}_k) \\ &= \frac{\mathbf{w}_i^\top \mathbf{w}_j}{\|\mathbf{w}_i\|\|\mathbf{w}_j\|} - \frac{\mathbf{w}_i^\top \mathbf{w}_k}{\|\mathbf{w}_i\|\|\mathbf{w}_k\|} \end{aligned}$$

Remember the GloVe embedding training objective from Part 1 in this assignment. Assume tied weights and ignore the bias units, we can write the training loss as following:

$$\text{Simplified GloVe} \quad L(\{\mathbf{w}_i\}_{i=1}^V) = \sum_{i,j=1}^{V} (\mathbf{w}_i^\top \mathbf{w}_j - \log X_{ij})^2,$$

where X_{ij} denotes the number of times word i and word j co-occured together in the training corpus. In the special case, X_{ii} denotes the number of times word i appeared in the corpus. When this model reaches zero training loss, the inner product of the GloVe embedding vectors will simply equal to the entries in the log co-occurrence matrix $\log X$. We can then express the WEAT association score in terms of the original co-occurrence matrix:

$$s(\mathbf{w}_i, \{\mathbf{w}_j\}, \{\mathbf{w}_k\}) = \frac{1}{\sqrt{logX_{ii}}} \left(\frac{logX_{ij}}{\sqrt{logX_{jj}}} - \frac{logX_{ik}}{\sqrt{logX_{kk}}} \right)$$

Briefly explain how this fact might contribute to the results from the previous section when using different attribute words. Provide your answers in no more than three sentences.

Hint: The paper cited above is a great resource if you are stuck.

7.3.3 Relative association between two sets of target words [0 pts] [Type 3]

In the original WEAT paper, the authors do not examine the association of individual words with attributes, but rather compare the relative association of two sets of target words. For example, are insect words more associated with positive attributes or negative attributes than flower words.

Formally, let X and Y be two sets of target words of equal size. The WEAT test statistic is given by:

$$s(X, Y, A, B) = \sum_{x \in X} s(x, A, B) - \sum_{y \in Y} s(y, A, B) \tag{6}$$

Will the same technique from the previous section work to manipulate this test statistic as well? Provide your answer in no more than 3 sentences.

References

Richard Socher Jeffrey Pennington and Christopher D Manning. Glove: Global vectors for word representation. Citeseer.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Aylin Caliskan, Joanna J Bryson, and Arvind Narayanan. Semantics derived automatically from language corpora contain human-like biases. *Science*, 356(6334):183–186, 2017.

Kawin Ethayarajh, David Duvenaud, and Graeme Hirst. Understanding undesirable word embedding associations. *arXiv preprint arXiv:1908.06361*, 2019.