Assignment 1

Deadline: October 6, 2025 by 2pm.
Assignment structure: There are two sections of problems. One includes written problems, which are presented in this file. The other set involves programming. More information about these are in the Notebook "hw1_code.ipynb".
Submission:

- You must submit your solutions of written problems as a PDF file through **GradeScope**. Note that the PDF file of written problems needs to be concatenated with the PDF file generated from the Notebook before being submitted to GradeScope. You may access the GradeScope page through the navigation bars of the CourseWorks page. Please make sure you link the solutions and questions correctly to receive credits on Gradescope. Please include justification for all your answers - an answer with no work shown will not receive credit.
- The `hw1_code_<YOUR_UNI>.ipynb` iPython Notebook, where `<YOUR_UNI>` is your uni. This file needs to be submitted to the **CourseWorks assignment page**.

Please read the following instructions before attempting the assignment.

Collaboration Policy: You are welcome to work together with other students on the homework. However, you must write up your solutions to the assignment individually. You are also welcome to use online resources that you find helpful (e.g. tutorials, course notes, textbooks, etc.) However, directly seeking and copying/paraphrasing answers or hints for the homework questions is prohibited. If you submit an assignment that contains copied/paraphrased text or code, either from someone else or online, you will receive a 0 for the homework. Note also that if you rely too much on outside resources, you may not learn the material (the main goal!) and would likely do poorly on exams (where such resources are not available).

Generative AI Policy. You may ask general questions about concepts related to the homework problems. However, you may not directly ask them for hints or answers to the homework questions. For example, you should not be copying and pasting directly from the assignment handout to a chat interface. In addition, you may not directly use the output of a chatbot (or a paraphrased output) in your answers.

1 Written Problems

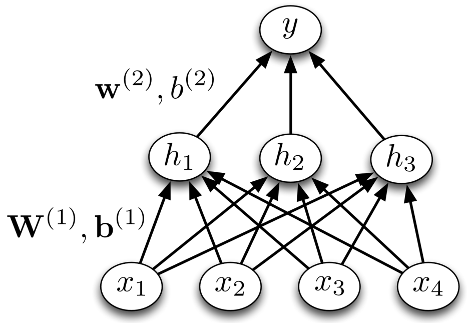
1.1 Hard-Coding Networks [10 pts]

The reading on multilayer perceptrons located at `https://www.cs.columbia.edu/~zemel/Class/nndl-2025/Readings/Grosse-Multilayer-Perceptrons.pdf` may be useful for this question.

A *superincreasing sequence*[1] is a sequence of numbers $(s_1, s_2, \ldots, s_n)$ such that every number s_i is greater than the sum of all previous numbers in the sequence. That is,

$$\forall i \in \{2, \ldots, n\}, s_i > \sum_{j=1}^{i-1} s_j.$$

For this problem, you will hard-code a neural network to be a *superlative superincreasing sequence solver*. The model will receive four inputs: x_1, x_2, x_3, and x_4. It should output 1 if the inputs form a superincreasing sequence, and 0 otherwise. Your network will have the following architecture:


y
$\mathbf{w}^{(2)}, b^{(2)}$
h_1
h_2
h_3
$\mathbf{W}^{(1)}, \mathbf{b}^{(1)}$
x_1
x_2
x_3
x_4

In this network, $\mathbf{W}^{(1)}, \mathbf{b}^{(1)}$ are the 3×4 weight matrix and 3-dimensional bias respectively for the hidden layer, and $\mathbf{w}^{(2)}, b^{(2)}$ are the 3-dimensional weight vector and scalar bias for the output layer. Both the hidden units and the output unit will use activation functions (you have to specify the activation functions for each layer ϕ_1, ϕ_2).

Note: You may find the following two activation functions useful.
1) Hard threshold activation function:[2]

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

2) Indicator activation function:

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

1.2 Backpropagation

The reading on backpropagation located at `https://www.cs.columbia.edu/~zemel/Class/nndl-2025/Readings/Grosse-Backpropagation.pdf` may be useful for this question.

[1] `https://en.wikipedia.org/wiki/Superincreasing_sequence`

[2] This is often defined as $\phi(z) = \mathbb{I}(z \geq 0)$, but we require strict inequality here.

Consider a neural network defined with the following procedure:

$$\begin{aligned}
\mathbf{a}_1 &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\
\mathbf{c}_1 &= \sigma(\mathbf{a}_1) \\
\mathbf{a}_2 &= \mathbf{W}^{(2)}\mathbf{x} + \mathbf{b}^{(2)} \\
u &= \mathbf{a}_1 + \mathbf{a}_2 \\
\mathbf{c}_2 &= \mathrm{ReLU}(u) \\
\mathbf{g} &= \mathbf{W}^{(g)} \begin{bmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{bmatrix} + \mathbf{b}^{(g)} \\
\mathbf{y} &= \mathbf{W}^{(3)}\mathbf{g} + \mathbf{W}^{(4)}\mathbf{x}, \\
\mathbf{y}' &= \mathrm{softmax}(\mathbf{y}) \\
\mathcal{S} &= \sum_{k=1}^{N} \mathbb{I}(t_k = 1)\log(\mathbf{y}'_k) \\
\mathcal{J} &= -\mathcal{S}
\end{aligned}$$

for input $\mathbf{x}$ with class label t where $\sigma(\mathbf{a}) = \frac{1}{1+e^{-\mathbf{a}}}$ denotes the Sigmoid activation function, $\mathrm{ReLU}(\mathbf{a}) = \max(\mathbf{a}, 0)$ denotes the ReLU activation function, both applied elementwise, and $\mathrm{softmax}(\mathbf{y}) = \frac{\exp(\mathbf{y})}{\sum_{i=1}^{N}\exp(\mathbf{y}_i)}$. Here, $[\,\mathbf{c}_1; \mathbf{c}_2\,]$ denotes concatenation.

Computation Graph [5 pts]: Draw the computation graph relating $\mathbf{x}$, t, $\mathbf{a}_1$, $\mathbf{c}_1$, $\mathbf{a}_2$, $\mathbf{c}_2$, $\mathbf{g}$, $\mathbf{y}$, $\mathbf{y}'$, $\mathcal{S}$ and $\mathcal{J}$.

Backward Pass [5 pts]: Derive the backprop equations for computing $\bar{\mathbf{x}} = \frac{\partial \mathcal{J}}{\partial \mathbf{x}}$, one variable at a time, similar to the vectorized backward pass derived in Lec 2.

1.3 Automatic Differentiation

Consider the function $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$ where $\mathcal{L}(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{v}\mathbf{v}^\top \mathbf{x}$, and $\mathbf{v} \in \mathbb{R}^{n\times 1}$ and $\mathbf{x} \in \mathbb{R}^{n\times 1}$. Here, we will explore the relative costs of evaluating Jacobians and vector-Jacobian products. Specifically, we will study vector-Hessian products, which is a special case of vector-Jacobian products, where the Jacobian is of the gradient of our function. We denote the gradient of $\mathcal{L}$ with respect to $\mathbf{x}$ as $\mathbf{g} \in \mathbb{R}^{1\times n}$ and the Hessian of $\mathcal{L}$ w.r.t. $\mathbf{x}$ with $\mathbf{H} \in \mathbb{R}^{n\times n}$. The Hessian of $\mathcal{L}$ w.r.t. $\mathbf{x}$ is defined as:

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 \mathcal{L}}{\partial x_1^2} & \frac{\partial^2 \mathcal{L}}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial x_1 \partial x_n} \\ \frac{\partial^2 \mathcal{L}}{\partial x_2 \partial x_1} & \frac{\partial^2 \mathcal{L}}{\partial x_2^2} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathcal{L}}{\partial x_n \partial x_1} & \frac{\partial^2 \mathcal{L}}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial x_n^2} \end{pmatrix}$$

Compute Hessian [5 pts]: Compute $\mathbf{H}$ for $n = 3$ and $\mathbf{v}^\top = [3, 1, 4]$ at $\mathbf{x}^\top = [2, 1, 3]$. In other words, write down the numbers in:

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 \mathcal{L}}{\partial x_1^2} & \frac{\partial^2 \mathcal{L}}{\partial x_1 \partial x_2} & \frac{\partial^2 \mathcal{L}}{\partial x_1 \partial x_3} \\ \frac{\partial^2 \mathcal{L}}{\partial x_2 \partial x_1} & \frac{\partial^2 \mathcal{L}}{\partial x_2^2} & \frac{\partial^2 \mathcal{L}}{\partial x_2 \partial x_3} \\ \frac{\partial^2 \mathcal{L}}{\partial x_3 \partial x_1} & \frac{\partial^2 \mathcal{L}}{\partial x_3 \partial x_2} & \frac{\partial^2 \mathcal{L}}{\partial x_3^2} \end{pmatrix}$$

Computation Cost [5 pts]: What is the number of scalar multiplications and memory cost to compute the Hessian $\mathbf{H}$ in terms of n? You may use big $\mathcal{O}$ notation.

1.3.1 Vector-Hessian Products [10 pts]

Compute $\mathbf{z} = \mathbf{H}\mathbf{y} = \mathbf{v}\mathbf{v}^\top\mathbf{y}$ where $n = 3$, $\mathbf{v}^\top = [4, 1, 2]$, $\mathbf{y}^\top = [2, 2, 1]$ using two algorithms: reverse-mode and forward-mode autodiff. In backpropagation (also known as reverse-mode autodiff), you will compute $\mathbf{M} = \mathbf{v}^\top\mathbf{y}$ first, then compute $\mathbf{v}\mathbf{M}$. Whereas, in forward-mode, you will compute $\mathbf{H} = \mathbf{v}\mathbf{v}^\top$ then compute $\mathbf{H}\mathbf{y}$. Write down the numerical values of $\mathbf{z}^T = [z_1, z_2, z_3]$ for the given $\mathbf{v}$ and $\mathbf{y}$.

Consider computing $\mathbf{Z} = \mathbf{H}\mathbf{y}_1\mathbf{y}_2^\top$ where $\mathbf{v} \in \mathbb{R}^{n\times 1}, \mathbf{y}_1 \in \mathbb{R}^{n\times 1}$ and $\mathbf{y}_2 \in \mathbb{R}^{m\times 1}$. What are number of scalar multiplications and memory cost in evaluating $\mathbf{Z}$ with reverse-mode in terms of n and m? What about forward-mode? When is forward-mode a better choice? (Hint: Think about the shape of Z, "tall" versus "wide".)

2 Programming Problems

The programming assignments are individual work.

You should attempt all questions for this assignment. Most of them can be answered at least partially even if you were unable to finish earlier questions. If you think your computational results are incorrect, please say so; that may help you get partial credit.

Introduction

In this section 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 notebook, as well as the data files for the assignment, are available on the course website: `https://www.cs.columbia.edu/~zemel/Class/nndl-2025/`. You can download the data files and put them in the same folder as where you store the notebook. In the notebook, you will see a list of packages to be imported to start the programming questions.

The file *raw_sentences.txt* 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).

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 x 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.

Even though you only have to modify two specific locations in the code, you may want to read through this code before starting the assignment.

2.1 GLoVE Word Representations [Total of 16 pts]

In this part, you will implement a simplified version of the GLoVE embedding (please see the lecture notes for a detailed description of the algorithm) with the loss defined as

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

.

Note that each word is represented by two d-dimensional embedding vectors $\mathbf{w}_i$, $\tilde{\mathbf{w}}_i$ and two scalar biases $b_i, \tilde{b}_i$.

Answer the following questions:

2.1.1 GLoVE Parameter Count [1 pt]

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

2.1.2 Expression for gradient $\frac{\partial L}{\partial \mathbf{w}_i}$ and $\frac{\partial L}{\partial \mathbf{b}_i}$ [6 pts]

Write the expression for $\frac{\partial L}{\partial \mathbf{w}_i}$ and $\frac{\partial L}{\partial \mathbf{b}_i}$, the gradient of the loss function L with respect to parameter vector $\mathbf{w}_i$ and $\mathbf{b}_i$. The gradient should be a function of $\mathbf{w}, \tilde{\mathbf{w}}, b, \tilde{b}, X$ with appropriate subscripts (if any).

2.1.3 Implement the gradient update of GLoVE. [6 pts]

See `YOUR CODE HERE` Comment below for where to complete the code

We have provided a few functions for training the embedding:

- `calculate_log_co_occurence` computes the log co-occurrence matrix of a given corpus
- `train_GLoVE` runs momentum gradient descent to optimize the embedding
- `loss_GLoVE`:
- INPUT - $V \times d$ matrix `W` (collection of V embedding vectors, each d-dimensional); $V \times d$ matrix `W_tilde`; $V \times 1$ vector `b` (collection of V bias terms); $V \times 1$ vector `b_tilde`; $V \times V$ log co-occurrence matrix.
- OUTPUT - loss of the GLoVE objective
- `grad_GLoVE`: **TO BE IMPLEMENTED.**
- INPUT:
  - $V \times d$ matrix `W` (collection of V embedding vectors, each d-dimensional), embedding for first word;
  - $V \times d$ matrix `W_tilde`, embedding for second word;

  - $V \times 1$ vector `b` (collection of V bias terms);
  - $V \times 1$ vector `b_tilde`, bias for second word;
  - $V \times V$ log co-occurrence matrix.
- OUTPUT:
  - $V \times d$ matrix `grad_W` containing the gradient of the loss function w.r.t. `W`;
  - $V \times d$ matrix `grad_W_tilde` containing the gradient of the loss function w.r.t. `W_tilde`;
  - $V \times 1$ vector `grad_b` which is the gradient of the loss function w.r.t. `b`.
  - $V \times 1$ vector `grad_b_tilde` which is the gradient of the loss function w.r.t. `b_tilde`.

Run the code to compute the co-occurrence matrix. Make sure to add a 1 to the occurrences, so there are no 0's in the matrix when we take the elementwise log of the matrix.

- [ ] **TO BE IMPLEMENTED**: Calculate the gradient of the loss function w.r.t. the parameters W, $\tilde{W}$, $\mathbf{b}$, and $\mathbf{b}$. You should vectorize the computation, i.e. not loop over every word. The reading of the matrix cookbook from `https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf` might be useful.

2.1.4 Effect of embedding dimension d [3 pts]

Train both the symmetric and asymmetric GLoVe model with varying dimensionality d by running the cell below. Comment on: 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, and why?

2.2 Training the model [Total of 26 pts]

We will modify the architecture slightly inspired by BERT [3]. Instead of having only one output, the architecture will now take in $N = 4$ context words, and also output predictions for $N = 4$ words.

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 practice, this `[MASK]` token is assigned the index 0 in our dictionary. The weights $W^{(2)}$ = `hid_to_output_weights` now 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. We call this as *concatenating* output uniits across all word positions, i.e. the $(j + nV)$-th column is for the word j in vocabulary for the n-th output word position. Note here that the softmax is applied in chunks of V as well, to give a valid probability distribution over the V words. Only the output word positions that were masked in the input are included in the cross entropy loss calculation:

$$C = -\sum_{i}^{B}\sum_{n}^{N}\sum_{j}^{V} m_n^{(i)}(t_{n,j}^{(i)} \log y_{n,j}^{(i)}),$$

where $y_{n,j}^{(i)}$ denotes the output probability prediction from the neural network for the i-th training example for the word j in the n-th output word, and $t_{n,j}^{(i)}$ is 1 if for the i-th training example, the

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of NAACL-HLT 2019, Volume 1 (Long and Short Papers), pages 4171–4186, Minneapolis, Minnesota. ACL.

word j is the n-th word in context. Finally, $m_n^{(i)} \in \{0,1\}$ 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.

There are three classes defined in this part: `Params`, `Activations`, `Model`. You will make changes to `Model`, but it may help to read through `Params` and `Activations` first.

In this part of the assignment, you implement a method which computes the gradient using backpropagation. To start you out, the *Model* class contains several important methods used in training:

- `compute_activations` computes the activations of all units on a given input batch
- `compute_loss` computes the total cross-entropy loss on a mini-batch
- `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 which are needed for training, and print the outputs of the gradients.

2.2.1 Implement gradient with respect to output layer inputs [8 pts]

* [ ] **TO BE IMPLEMENTED**: `compute_loss_derivative` computes the derivative of the loss function with respect to the output layer inputs.

In other words, if C is the cost function, and the softmax computation for the j-th word in vocabulary for the n-th output word position is:

$$y_{n,j} = \frac{e^{z_{n,j}}}{\sum_l e^{z_{n,l}}}$$

This function should compute a $B \times NV$ matrix where the entries correspond to the partial derivatives $\partial C / \partial z_j^n$. Recall that the output units are concatenated across all positions, i.e. the $(j + nV)$-th column is for the word j in vocabulary for the n-th output word position.

2.2.2 Implement gradient with respect to parameters [8 pts]

- [ ] **TO BE IMPLEMENTED**: `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`, `hid_bias`, `hid_to_output_weights`, and `output_bias`. These matrices have the same sizes as the parameter matrices (see previous section).

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 elementwise 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.

To make your life easier, we have provided the routine `checking.check_gradients`, which checks your gradients using finite differences. You should make sure this check passes before continuing with the assignment.

2.2.3 Print the gradients [8 pts]

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 before continuing with the assignment. Once `check_gradients()` passes, call `print_gradients()` and include its output in your write-up.

2.2.4 Run model training [2 pts]

Once you've implemented the gradient computation, you'll need to train the model. The function *train* 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

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.

To convince us that you have correctly implemented the gradient computations, please include the following with your assignment submission:

- [ ] You will submit `hw1_code_<YOUR_UNI>.ipynb` through CourseWorks. You do not need to modify any of the code except the parts we asked you to implement.
- [ ] In your PDF file, include the output of the function `print_gradients`. This prints out part of the gradients for a partially trained network which we have provided, and we will check them against the correct outputs. **Important:** make sure to give the output of `print_gradients`, **not** `check_gradients`.

Since we gave you a gradient checker, you have no excuse for not getting full points on this part.

2.3 Arithmetics and Analysis [8 pts]

In this part, you will perform arithmetic calculations on the word embeddings learned from previous models and analyze the representation learned by the networks with t-SNE plots.

t-SNE

You will first train the models discussed in the previous sections; you'll use the trained models for the remaining of this section.

Important: if you've made any changes to your gradient code, you must reload the hw1_code module and then re-run the training procedure. Python does not reload modules automatically, and you don't want to accidentally analyze an old version of your model.

These methods of the Model class can be used for analyzing the model after the training is done: * `tsne_plot_representation` creates a 2-dimensional embedding of the distributed representation space using an algorithm called t-SNE. (You don't need to know what this is for the assignment, but we may cover it later in the course.) Nearby points in this 2-D space are meant to correspond to nearby points in the 16-D space. * `display_nearest_words` lists the words whose embedding vectors are nearest to the given word * `word_distance` computes the distance between the embeddings of two words

Plot the 2-dimensional visualization for the trained model from part 3 using the method `tsne_plot_representation`. Look at the plot and find a few clusters of related words. What do the words in each cluster have in common? Plot the 2-dimensional visualization for the GloVe model from part 1 using the method `tsne_plot_GLoVe_representation`. How do the t-SNE embeddings for both models compare? Plot the 2-dimensional visualization using the method `plot_2d_GLoVe_representation`. How does this compare to the t-SNE embeddings? Please answer in 2 sentences for each question and show the plots in your submission.