Assignment 1

Version: 1.0
Version Release Date: 2025-01-13

Deadline: Friday, Jan. 31, at 1:00pm.

Submission: You must submit two files through MarkUs: (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 for detailed policies. You may ask questions about the assignment on Piazza.

The assignment is marked out of 100 points. 10 points are allocated to neatness, so be sure that your answers and code are clear and readable!

You may notice that some questions are worth 0 points, which means we will not mark them in this assignment. However, we think these questions are educationally valuable and may help you prepare for the tests, so we recommend you spend some time thinking about them.

Important Instructions

Read the following before attempting the assignment.

Collaboration Policy

You are welcome to work together with other students on the homework. You are also welcome to use any resources you find (online tutorials, textbooks, papers, etc.) to help you complete the homework. However, you must write up your submission by yourself and not use any content generated by someone else or generative AI. You must cite all the collaboration and resources you used to complete each assignment. If you hand in homework that makes use of content that you did not create or you do not disclose the collaboration or resources, you will get a 0 for that homework. Note also that if you rely too much on outside resources, you will likely not learn the material and will do poorly on the exams, during which such resources will not be available.

Generative AI Policy

You may ask general questions about concepts related to the homework questions. However, you may not directly ask them for hints on homework assignment questions; e.g., you should not be copying and pasting directly from the assignment handout into the chat interface. Also, you may not directly use the outputs of AI chatbots in your homework solutions (even paraphrased) unless instructed to do so.

You must include any chat transcripts related to a homework assignment along with your submission for the assignment. We will interpret the above policy leniently when judging whether the GenAI use is appropriate, though we reserve the right to update the GenAI policy if we find that students are using it in a way that reduces the educational value of the homeworks.

If you use GenAI, then your transcripts should be submitted on MarkUs with filenames starting with `chat_transcript`.

Written Assignment

What you have to submit for this part

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

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, 1.2.2, 1.3.1, 1.3.2, 1.3.4
- **Problem 2:** 2.1.2, 2.2.1, 2.2.2, 2.2.3
- **Problem 3:** 3.1, 3.2

1 Linear Regression (20 points)

The reading on linear regression at `https://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/readings/L02a%20Linear%20Regression.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 [0 points]

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

$$\nabla_w \mathcal{L}(\hat{\mathbf{w}}) = \nabla_w (X\hat{\mathbf{w}} - \mathbf{t})^2 = \nabla_w [(X\hat{\mathbf{w}} - \mathbf{t})^T (X\hat{\mathbf{w}} - \mathbf{t})] = \nabla_w [\hat{\mathbf{w}}^T X^T X \hat{\mathbf{w}} - 2\hat{\mathbf{w}}^T X^T \mathbf{t} + \mathbf{t}^T \mathbf{t}]$$

$$\nabla_w \mathcal{L}(\hat{\mathbf{w}}) = 2X^T X \hat{\mathbf{w}} - 2X^T \mathbf{t} \tag{1}$$

1.2 Underparameterized Model

1.2.1 [4 points]

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.

Hint: If training converges for this problem, how is the solution obtained by gradient descent related to the direct solution?

In gradient descent, the update rule is

$$\hat{\mathbf{w}}_{t+1} \leftarrow \hat{\mathbf{w}}_t - \alpha \nabla_w \mathcal{L}_w(\hat{\mathbf{w}}).$$

If the training converges, then the updates approach zero, which requires the loss to approach zero. When the loss is zero, we can use the result from Equation 1 to determine an expression for $\hat{\mathbf{w}}$.

$$\begin{aligned} \nabla_w \mathcal{L}(\hat{\mathbf{w}}) = 0 &= 2X^T X\hat{\mathbf{w}} - 2X^T\mathbf{t} \\ \rightarrow \quad & 2X^T X\hat{\mathbf{w}} = 2X^T\mathbf{t} \\ \rightarrow \quad & \hat{\mathbf{w}} = (X^T X)^{-1}X^T\mathbf{t} \end{aligned} \tag{2}$$

Note that we assume $X^T X$ is invertible when $n > d$, which is true in the over determined case. Therefore, we have shown that the solution obtained by gradient descent is the same as the closed form solution when it converges.

1.2.2 [4 points]

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 an independent random variable for each i, with zero mean and variance σ^2. The final training loss at convergence can be derived as a function of X and $\boldsymbol{\epsilon}$, as:

$$Loss = \frac{1}{n}\|X\hat{\mathbf{w}} - \mathbf{t}|_2^2 = \frac{1}{n}||(X(X^\top X)^{-1}X^\top - I)\boldsymbol{\epsilon}||_2^2,$$

where $\boldsymbol{\epsilon}$ is the vector of noise terms. Show this is true using your answer from the previous question, as well as the noisy targets defined above. Also, find the expectation of the above training error in terms of n, d and σ.

Hint 1: Think about the trace operator. Remember that it is linear and so, expectation of a trace is equal to trace of the expectation.
Hint 2: You might also find the cyclic property [1] *of trace useful.*

We first note that at convergence, the weight vector is given by Equation 2, as shown in 1.2.1.

$$\begin{aligned} \mathcal{L} &= \frac{1}{n}\|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2 = \frac{1}{n}\|XX^\dagger\mathbf{t} - \mathbf{t}\|_2^2 \\ &= \frac{1}{n}\|(XX^\dagger - I)\mathbf{t}\|_2^2 = \frac{1}{n}\|(XX^\dagger - I)(X\hat{\mathbf{w}} + \hat{\epsilon})\|_2^2 \\ &= \frac{1}{n}\|(XX^\dagger - I)X\hat{\mathbf{w}} + (XX^\dagger - I)\hat{\epsilon})\|_2^2 \\ &= \frac{1}{n}\|XX^\dagger X\hat{\mathbf{w}} - X\hat{\mathbf{w}} + (XX^\dagger - I)\hat{\epsilon})\|_2^2 \\ &= \frac{1}{n}\|X(X^T X)^{-1}X^T X\hat{\mathbf{w}} - X\hat{\mathbf{w}} + (XX^\dagger - I)\hat{\epsilon})\|_2^2 \end{aligned}$$

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

$$= \frac{1}{n}\|X\hat{\mathbf{w}} - X\hat{\mathbf{w}} + (XX^\dagger - I)\hat{\epsilon})\|_2^2 = \frac{1}{n}\|(XX^\dagger - I)\hat{\epsilon})\|_2^2$$

$$\mathcal{L} = \frac{1}{n}\|(X(X^TX)^{-1}X^T - I)\hat{\epsilon})\|_2^2$$

Now we compute the expectation.

$$\mathbb{E}[\mathcal{L}] = \mathbb{E}\left[\frac{1}{n}\|(X(X^TX)^{-1}X^T - I)\hat{\epsilon})\|_2^2\right]$$

$$= \mathbb{E}\left[\frac{1}{n}(X(X^TX)^{-1}X^T - I)\hat{\epsilon})^T(X(X^TX)^{-1}X^T - I)\hat{\epsilon})\right]$$

$$= \frac{1}{n}\mathbb{E}\left[\left(\hat{\epsilon}^TX(X^TX)^{-1}X^T - \hat{\epsilon}^T\right)\left(X(X^TX)^{-1}X^T\hat{\epsilon} - \hat{\epsilon}\right)\right]$$

$$= \frac{1}{n}\mathbb{E}\left[\hat{\epsilon}^TX(X^TX)^{-1}X^TX(X^TX)^{-1}X^T\hat{\epsilon} - \hat{\epsilon}^TX(X^TX)^{-1}X^T\hat{\epsilon} - \hat{\epsilon}^TX(X^TX)^{-1}X^T\hat{\epsilon} + \hat{\epsilon}^T\hat{\epsilon}\right]$$

$$= \frac{1}{n}\mathbb{E}\left[\hat{\epsilon}^TX(X^TX)^{-1}X^T\hat{\epsilon} - \hat{\epsilon}^TX(X^TX)^{-1}X^T\hat{\epsilon} - \hat{\epsilon}^TX(X^TX)^{-1}X^T\hat{\epsilon} + \hat{\epsilon}^T\hat{\epsilon}\right]$$

$$= \frac{1}{n}\mathbb{E}\left[\hat{\epsilon}^T\hat{\epsilon} - \hat{\epsilon}^TX(X^TX)^{-1}X^T\hat{\epsilon}\right]$$

$$= \frac{1}{n}\mathbb{E}\left[\hat{\epsilon}^T\hat{\epsilon}\right] - \frac{1}{n}\mathbb{E}\left[\hat{\epsilon}^TX(X^TX)^{-1}X^T\hat{\epsilon}\right]$$

We can solve the individual terms individually for neatness.

$$\mathbb{E}\left[\hat{\epsilon}^T\hat{\epsilon}\right] = \mathbb{E}\left[\sum_{i=1}^n \hat{\epsilon}_i^2\right] = \sum_{i=1}^n \mathbb{E}\left[\hat{\epsilon}_i^2\right] = n\sigma^2$$

$$\mathbb{E}\left[\hat{\epsilon}^TX(X^TX)^{-1}X^T\hat{\epsilon}\right] = \mathbb{E}\left[tr\left(\hat{\epsilon}\hat{\epsilon}^TX(X^TX)^{-1}X^T\right)\right]$$

$$= \mathbb{E}\left[tr\left(X^T\hat{\epsilon}\hat{\epsilon}^TX(X^TX)^{-1}\right)\right]$$

$$= \mathbb{E}\left[tr\left(X^T\sigma^2I_nX(X^TX)^{-1}\right)\right]$$

$$= \mathbb{E}\left[tr\left(\sigma^2X^TX(X^TX)^{-1}\right)\right]$$

$$= \sigma^2\,\mathbb{E}\left[tr\left(X^TX(X^TX)^{-1}\right)\right]$$

$$= \sigma^2\,\mathbb{E}\left[tr\left(I_d\right)\right]$$

$$= \sigma^2 d$$

Now plugging these results in, we get the expectation of the loss.

$$\mathbb{E}[\mathcal{L}] = \frac{1}{n}\left(\mathbb{E}\left[\hat{\epsilon}\hat{\epsilon}^T\right] - \mathbb{E}\left[\hat{\epsilon}^TX(X^TX)^{-1}X^T\hat{\epsilon}\right]\right)$$

$$= \frac{1}{n}\left(n\sigma^2 - \sigma d\right)$$

$$\mathbb{E}[\mathcal{L}] = \sigma^2\left(1 - \frac{d}{n}\right)$$

As required.

1.3 Overparameterized Model

1.3.1 [4 points]

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 $\hat{\mathbf{w}}^\top \mathbf{x}_1 = t_1$, all of which lie on a real line. Write down the equation of the line.

$$\hat{\mathbf{w}}^T \mathbf{x}_1 = t_1 \rightarrow w_0 x_0 + w_1 x_1 = t_1$$

$$w_0 + w_1 = 3 \rightarrow w_0 = -w_1 + 3$$

$$\hat{\mathbf{w}}^T \mathbf{x}_1 = \begin{bmatrix} -w1 + 3 \\ w1 \end{bmatrix}^T \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} -w1 + 3 \\ w1 \end{bmatrix}^T \begin{bmatrix} 1 \\ 1 \end{bmatrix} = (-w_1 + 3) + w_1 = 3$$

Therefore if we pick $w_0 = 3 - w_1$, there is an infinite number of solutions to $\hat{\mathbf{w}}^T \mathbf{x}_1 = t_1$ for $\mathbf{x}_1 = [1; 1]$ and $t_1 = 3$. The line of solutions is given by $w_0 = 3 - w_1$ and there exist infinite points along this line, therefore there are infinite solutions to this case.

1.3.2 [4 points]

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 $\mathbf{a} \in \mathbb{R}^n$.

The gradient descent update rule is,

$$\hat{\mathbf{w}}_{t+1} = \hat{\mathbf{w}}_t + \alpha \nabla_w \mathcal{L}_w(\hat{\mathbf{w}}).$$

If training converges, it means $\nabla_w \mathcal{L}_w(\hat{\mathbf{w}}) = 0$ as the updates are not changing the weights. So we can solve for $\hat{\mathbf{w}}$ in $\nabla_w \mathcal{L}_w(\hat{\mathbf{w}}) = 0$ to find the optimal weights. Then we can determine if this solution is unique.

Lets begin by using the the hint that $\hat{\mathbf{w}} = X^T a$ for some $a \in \mathbb{R}^n$.

$$\mathcal{L} = \frac{1}{n}\|X\hat{\mathbf{w}} - \mathbf{t}\|_2^2 = \frac{1}{n}\|XX^T a - \mathbf{t}\|_2^2 = \frac{1}{n}(XX^T a - \mathbf{t})^T (XX^T a - \mathbf{t})$$

$$\mathcal{L} = \frac{1}{n}(a^T XX^T XX^T a - 2XX^T a\mathbf{t} + \mathbf{t}^T \mathbf{t})$$

At convergence from gradient descent, the gradient term will be 0. Since X^T is a constant, the gradient of the loss w.r.t a will be 0 as well.

$$\nabla_a \mathcal{L}_a(a) = \frac{1}{n}(2XX^T XX^T a - 2XX^T \mathbf{t} = 0)$$

$$(XX^T)(XX^T)a = (XX^T)\mathbf{t}$$

$$a = (XX^T)^{-1}(XX^T)^{-1}(XX^T)\mathbf{t}$$

$$a = (XX^T)^{-1}\mathbf{t}$$

Substituting back into the hit equation,

$$\hat{\mathbf{w}} = X^T a = X^T (XX^T)^{-1}\mathbf{t},$$

demonstrating the final value of $\hat{\mathbf{w}}$ at convergence. The uniqueness arises from gradient descent, starting from zero, as it will converge to this specific vector if it converges at all.

1.3.3 [0 points]

Repeat part 1.2.2 for the overparameterized case.

ur crazy

1.3.4 [4 points]

Visualize and compare underparameterized with overparameterized polynomial regression: `https://colab.research.google.com/drive/19z9a-j3DZ6xS5lseee79D1ZFaW73fgxz`

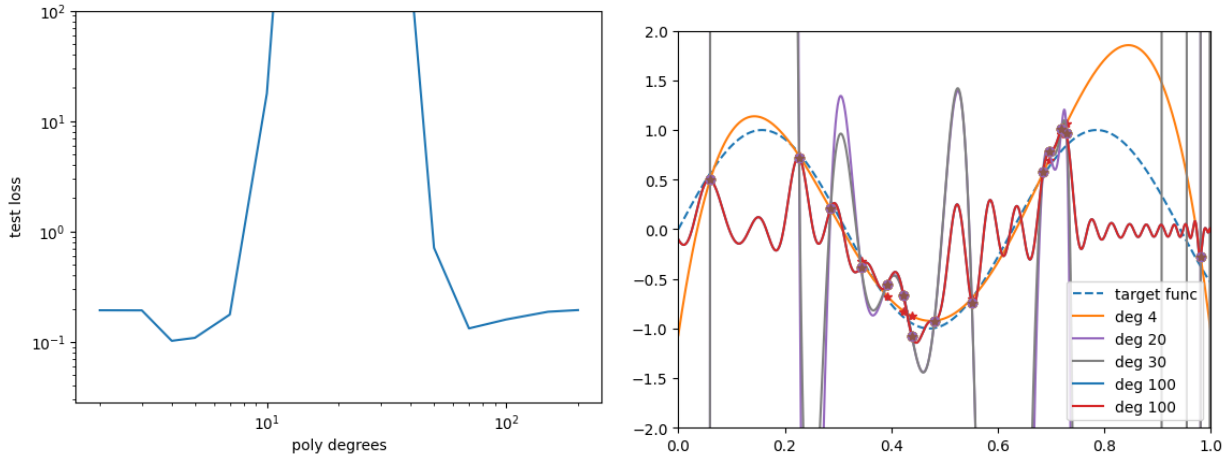
Include your code snippets for the `fit_poly` function in the write-up. Does overparameterization (higher degree polynomial) always lead to overfitting, i.e. larger test error?

Below is my implementation of the `fit_poly` function.

```
def fit_poly(X, d, t):
    X_expand = poly_expand(X, d=d, poly_type=poly_type)
    n = X.shape[0]
    if d > n:
        ## W = ... (Your solution for Part 1.3.2)
        W = X_expand.T @ np.linalg.inv(X_expand @ X_expand.T) @ t
    else:
        ## W = ... (Your solution for Part 1.2.1)
        W = np.linalg.inv(X_expand.T @ X_expand) @ X_expand.T @ t

    return W
```

We see that overgeneralization does not always lead to over fitting. Figure 1a shows the test loss plotted with degree. Initially, the test loos is near 10^{-1}, but once we approach 10 degrees, it increases well beyond 10^2. However, as we approach 10^2 degrees, it reduces back and remains there. From inspecting the figure, the best models is one with fewer degrees than 10. This makes sense as initially, it is able to meet a generalized fit of the data, which looks to only have 3 turning points, indicating that degree 4 is best. Beyond this, it cannot generalize to all the points as well, so the loss increases.



(a) Plot comparing test error to degree.

(b) Different fits of varying degrees. .

Figure 1: 1.3.4 Plots.

The loss explodes when we get higher degrees like 20, where in order to fit all the points it must have a huge range. This means it spends more time outside the range of the data so it has a less chance of hitting the points. It comes back down as the high degree is able to hit all the points exactly due to the increased turning points. This allows it to accidentally catch more cases and remain in the range of the test cases. Its loss is still not as good as before indicating the best model was before degree 10. Figure 1b demonstrates this phenomenon.

1.3.5 [0 points]

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.
Hint: use your code to experiment with relevant parameters, then vary to find region and report one such setting.

2 Backpropagation (26 points)

This question helps you to understand the underlying mechanism of backpropagation. 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 at `https://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/readings/L04%20Backpropagation.pdf` may be useful for this question.

2.1 Automatic Differentiation

Consider a neural network defined with the following procedure:

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

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

2.1.1 Computational Graph [2 points]

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 $\mathcal{J}$.

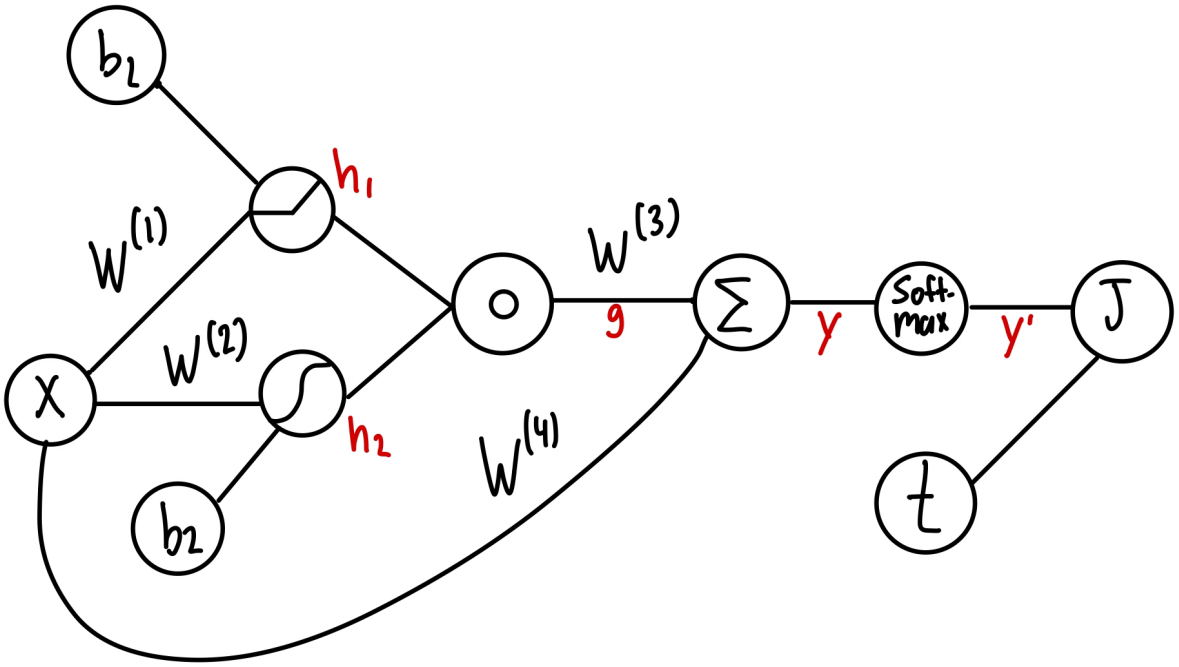


Figure 2: Computational Graph.

2.1.2 Backward Pass [6 points]

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.

Hint 1: Be careful about the transpose and shape! It is useful practice to sanity check the shapes of all terms in the computation. Assume all vectors (including error vector) are column vector and all Jacobian matrices adopt numerator-layout notation [2].
*Hint 2: You can use softmax′(**y**) for the Jacobian matrix of softmax.*

We derive the loss gradients w.r.t $\mathbf{x}$, t, $\mathbf{z}_1$, $\mathbf{z}_2$, $\mathbf{h}_1$, $\mathbf{h}_2$, $\mathbf{g}$, $\mathbf{y}$, $\mathbf{y}'$, $\mathcal{S}$ and $\mathcal{J}$.

$$\frac{\partial \mathcal{J}}{\partial \mathcal{J}} = 1 \qquad \frac{\partial \mathcal{J}}{\partial \mathcal{S}} = -1$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{y}'_k} = \frac{\partial \mathcal{S}}{\partial \mathbf{y}'}^T \frac{\partial \mathcal{J}}{\partial \mathcal{S}} = -1 \cdot \begin{cases} \frac{1}{\mathbf{y}'_k} \text{if } t = k \\ 0 \quad \text{if } t \neq k \end{cases} = -\frac{\mathbb{I}(t=k)}{\mathbf{y}'}$$

For notational convenience, we construct $\mathbf{t}$ which is a one hot encoded vector corresponding to the condition $t = k$. In essence, it is a vector of indicator functions where it is 1 if the label is correct, 0 otherwise. Thus we conclude

$$\frac{\partial \mathcal{J}}{\partial \mathbf{y}'} = -\frac{\mathbf{t}}{\mathbf{y}'}.$$

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

Now we can solve,

$$\frac{\partial \mathcal{J}}{\partial \mathbf{y}} = \frac{\partial \mathbf{y}'}{\partial \mathbf{y}}^T \frac{\partial \mathcal{J}}{\partial \mathbf{y}'} = softmax'(\mathbf{y})^T \frac{\partial \mathcal{J}}{\partial \mathbf{y}'}.$$

Here we take the transpose to maintain shape consistency.
Using this we can write the gradients with $W^{(3)}$ and $W^{(4)}$. Note that the gradients are matrices and $\mathbf{g}$, $\mathbf{x}$, and $\frac{\partial \mathcal{J}}{\partial \mathbf{y}}$ are vectors. Thus we take their outer product as required.

$$\frac{\partial \mathcal{J}}{\partial W^{(3)}} = \frac{\partial \mathbf{y}}{\partial W^{(3)}}^T \frac{\partial \mathcal{J}}{\partial \mathbf{y}} = \mathbf{g}^T \frac{\partial \mathcal{J}}{\partial \mathbf{y}} \qquad \frac{\partial \mathcal{J}}{\partial W^{(4)}} = \frac{\partial \mathbf{y}}{\partial W^{(4)}}^T \frac{\partial \mathcal{J}}{\partial \mathbf{y}} = \mathbf{x}^T \frac{\partial \mathcal{J}}{\partial \mathbf{y}}$$

We can compute the loss with respect to the intermediate value $\mathbf{g}$.

$$\frac{\partial \mathcal{J}}{\partial \mathbf{g}} = \frac{\partial \mathbf{y}}{\partial \mathbf{g}}^T \frac{\partial \mathcal{J}}{\partial \mathbf{y}} = {W^{(3)}}^T \frac{\partial \mathcal{J}}{\partial \mathbf{y}}$$

Now we can use the above gradients to solve for the hidden values $\mathbf{h}_i$.

$$\frac{\partial \mathcal{J}}{\partial \mathbf{h}_1} = \frac{\partial \mathbf{g}}{\partial \mathbf{h}_1}^T \frac{\partial \mathcal{J}}{\partial \mathbf{g}} = \mathbf{h}_2 \circ \frac{\partial \mathcal{J}}{\partial \mathbf{g}} \qquad \frac{\partial \mathcal{J}}{\partial \mathbf{h}_2} = \frac{\partial \mathbf{g}}{\partial \mathbf{h}_2}^T \frac{\partial \mathcal{J}}{\partial \mathbf{g}} = \mathbf{h}_1 \circ \frac{\partial \mathcal{J}}{\partial \mathbf{g}}$$

From here, we can compute the gradients in $\mathbf{z}_i$.

$$\frac{\partial \mathcal{J}}{\partial \mathbf{z}_1} = \frac{\partial \mathcal{J}}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial \mathbf{z}_1} = \frac{\partial \mathcal{J}}{\partial \mathbf{h}_1}\text{ReLU}'(\mathbf{z}_1) = \frac{\partial \mathcal{J}}{\partial \mathbf{h}_1} \circ \begin{cases} \mathbf{z}_{1i} > 0 \\ 0 \quad \mathbf{z}_{1i} < 0 \end{cases}$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{z}_2} = \frac{\partial \mathcal{J}}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{z}_2} = \frac{\partial \mathcal{J}}{\partial \mathbf{h}_2} \circ \sigma'(\mathbf{z}_2) = \frac{\partial \mathcal{J}}{\partial \mathbf{h}_2} \circ \mathbf{h}_2 \circ (1 - \mathbf{h}_2)$$

Using this, we can derive the final gradients w.r.t the first set of weights and biases.

$$\frac{\partial \mathcal{J}}{\partial W^{(1)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{z}_1}\frac{\partial \mathbf{z}_1}{\partial W^{(1)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{z}_1}\mathbf{x} \qquad \frac{\partial \mathcal{J}}{\partial W^{(2)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{z}_2}\frac{\partial \mathbf{z}_2}{\partial W^{(2)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{z}_2}\mathbf{x}$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{b}^{(1)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{z}_1}\frac{\partial \mathbf{z}_1}{\partial \mathbf{b}^{(1)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{z}_1} \qquad \frac{\partial \mathcal{J}}{\partial \mathbf{b}^{(2)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{z}_2}\frac{\partial \mathbf{z}_2}{\partial \mathbf{b}^{(2)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{z}_2}$$

Now we can compute the loss w.r.t. the input.

$$\frac{\partial \mathcal{J}}{\partial \mathbf{x}} = \frac{\partial \mathcal{J}}{\partial \mathbf{z}_1}\frac{\partial \mathbf{z}_1}{\partial \mathbf{x}} + \frac{\partial \mathcal{J}}{\partial \mathbf{z}_2}\frac{\partial \mathbf{z}_2}{\partial \mathbf{x}} + \frac{\partial \mathcal{J}}{\partial \mathbf{y}}\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{x}} = {\mathbf{W}^{(1)}}^T \frac{\partial \mathcal{J}}{\partial \mathbf{z}_1} + {\mathbf{W}^{(2)}}^T \frac{\partial \mathcal{J}}{\partial \mathbf{z}_2} + {\mathbf{W}^{(4)}}^T \frac{\partial \mathcal{J}}{\partial \mathbf{y}}$$

Thus we have derived the autodifferentiation backpropagation components for this network.

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.

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.

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

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 [6 points]

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:

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

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

$$\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(2)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{y}}\frac{\partial \mathbf{y}}{\partial \mathbf{W}^{(2)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{y}}\mathbf{h}^T = \frac{\partial \mathcal{J}}{\partial \mathbf{y}}\left(\text{ReLU}(\mathbf{W}^{(1)}\mathbf{x})\right)^T$$

$$= \begin{bmatrix}1\\1\\1\end{bmatrix}\text{ReLU}\left(\begin{bmatrix}1 & 3 & 1\end{bmatrix}\begin{bmatrix}1 & -2 & 1\\ 2 & 1 & -2\\ 1 & 0 & -1\end{bmatrix}\right)$$

$$= \begin{bmatrix}1\\1\\1\end{bmatrix}\text{ReLU}\left(\begin{bmatrix}8 & 1 & -6\end{bmatrix}\right) = \begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}8 & 1 & 0\end{bmatrix}$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(2)}} = \begin{bmatrix}8 & 1 & 0\\ 8 & 1 & 0\\ 8 & 1 & 0\end{bmatrix}$$

Now the gradient of the loss w.r.t. We note that $\mathbf{W}^{(1)}$.

$$\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}} = \left[\frac{\partial \mathcal{J}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right] \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}} = \left[\left(\frac{\partial \mathcal{J}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{h}}\right) \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right] \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}$$

$$= \left[\left(\mathbf{W}^{(2)^T} \begin{bmatrix}1\\1\\1\end{bmatrix}\right) \circ \text{ReLU}'(\mathbf{z})\right] \mathbf{x}^T$$

$$= \left(\begin{bmatrix}-2 & 1 & -3\\ 4 & -2 & 4\\ 1 & -3 & 6\end{bmatrix} \begin{bmatrix}1\\1\\1\end{bmatrix}\right) \circ \begin{bmatrix}1\\1\\0\end{bmatrix} \mathbf{x}^T = \begin{bmatrix}-4\\6\\0\end{bmatrix} \begin{bmatrix}1 & 3 & 1\end{bmatrix}$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}} = \begin{bmatrix}-4 & -12 & -4\\ 6 & 18 & 6\\ 0 & 0 & 0\end{bmatrix}$$

Given these we can compute the Frobenius Norm of the two Jacobian matrices as the sum of the elements squared.

$$\left|\left|\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}}\right|\right|_F^2 = \sum_{i=1}^{3}\sum_{j=1}^{3}\left|\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}}_{ij}\right|^2 = (-4)^2 + (-12)^2 + (-4)^2 + (6)^2 + (18)^2 + (6)^2$$

$$\left|\left|\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}}\right|\right|_F^2 = 572$$

$$\left|\left|\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(2)}}\right|\right|_F^2 = \sum_{i=1}^{3}\sum_{j=1}^{3}\left|\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(2)}}_{ij}\right|^2 = 3(8^2 + 1^2 + 0^2)$$

$$\left|\left|\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}}\right|\right|_F^2 = 195$$

2.2.2 Efficient Computation [3 points]

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

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

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

$$\left|\left|\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(1)}}\right|\right|_F^2 = \|\mathbf{x}\|_2^2\|\overline{\mathbf{z}}\|_2^2 = (1^2+3^2+1^2)((-4)^2+6^2+0^2) = (11)(52) = 572$$

$$\left|\left|\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(2)}}\right|\right|_F^2 = \|\overline{\mathbf{y}}\|_2^2\|\mathbf{h}\|_2^2 = (1^2+1^2+1^2)(8^2+1^2+0^2) = (3)(65) = 195$$

As expected. Note that the derivation of the trick for $\mathbf{W}^{(1)}$ is identical and thus trivial to include here.

2.2.3 Complexity Analysis [9 points]

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	NKD^2	NKD^2	$\mathcal{O}(KD^2+NKD)$	$\mathcal{O}(KD^2+NKD)$
Backward Pass	$(2NKD^2$	$(NKD^2$	$\mathcal{O}(NKD^2)$	$\mathcal{O}(KD^2+NKD)$
Gradient Norm Computation	NKD^2	$(2D+1)NK$	$\mathcal{O}(NKD^2)$	$\mathcal{O}(NKD)$

Hint: 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.

In the forward pass, the number of computations is not affected by the method. Each layer has D^2 weights and thus the same amount of scalar multiplications. For K layers, it becomes KD^2 scalar multiplications. This is just for 1 input, for N, the computation becomes NKD^2 scalar multiplications.
The memory requirement is ND for the input, and the weight matrics is KD^2. Then we must store the hidden vectors in each layer, KD, and there are N inputs so, an additional NKD. Thus the memory is $\mathcal{O}(ND+KD^2+NKD$. We assume $N << D$ for a given batch, thus $\mathcal{O}(KD^2)$

2.3 Inner product of Jacobian: JVP and VJP [not graded but recommended, 0 points]

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 can we 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: $M_1M_2M_3M_4 = ((M_1(M_2M_3))M_4) = (M_1((M_2M_3)M_4))$

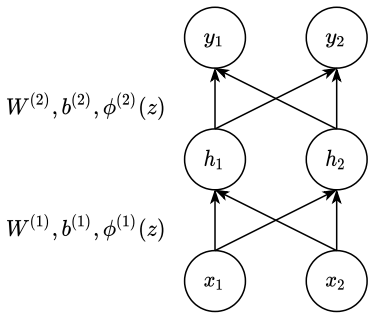
3 Hard-Coding Networks (14 points)

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 at `https://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/readings/L03%20Multilayer%20Perceptrons.pdf` may be useful.

3.1 Sort two numbers [6 points]

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 outputs 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.

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

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

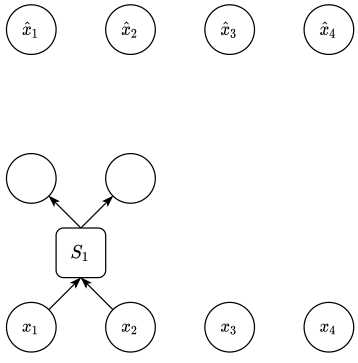
$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.5 & -0.5 \\ 0.5 & 0.5 \end{bmatrix} \qquad \mathbf{W}^{(2)} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$\mathbf{b}^{(1)} = \mathbf{b}^{(2)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\phi^{(1)}(z) = |z| \qquad \phi^{(2)}(z) = z$$

3.2 Perform Sort [8 points]

Draw a computation graph to show how to implement a sorting function $\hat{f} : \mathbb{R}^4 \to \mathbb{R}^4$ where $\hat{f}(x_1, x_2, x_3, x_4) = (\hat{x}_1, \hat{x}_2, \hat{x}_3, \hat{x}_4)$ and $(\hat{x}_1, \hat{x}_2, \hat{x}_3, \hat{x}_4)$ is (x_1, x_2, x_3, x_4) in sorted order. Let us assume $\hat{x}_1 \le \hat{x}_2 \le \hat{x}_3 \le \hat{x}_4$ 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.



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

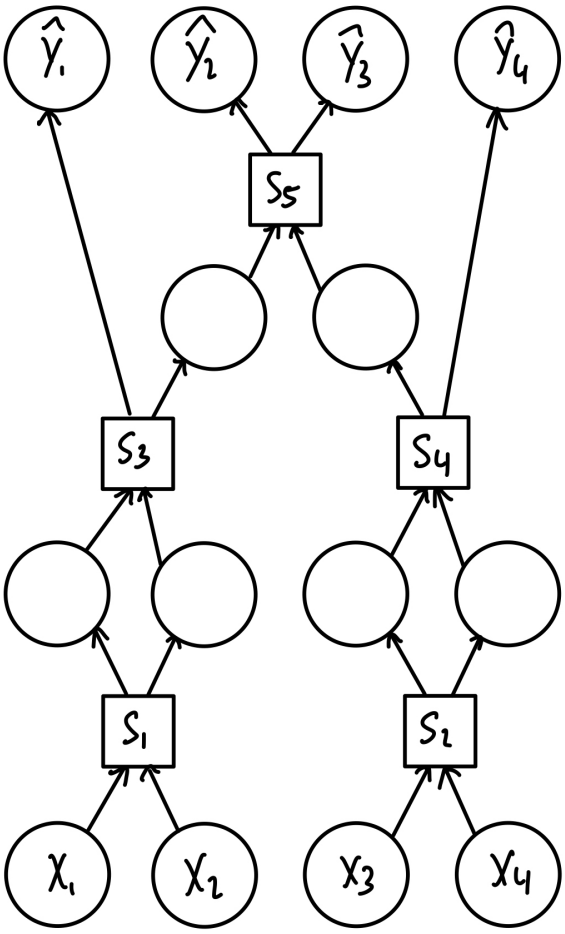


Figure 3: Sorting Network.

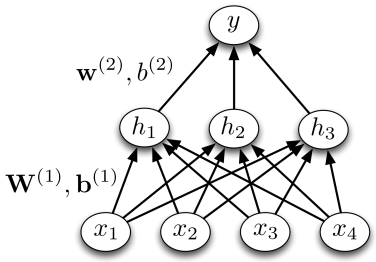
3.3 Find the k^{th} smallest number [not graded but recommended, 0 points]

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 $k = 3$.

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

3.4 Counting Network [not graded but recommended, 0 points]

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.

Hint: 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:

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