Homework 1

Version: 1.0
Version Release Date: 2022-01-15

Deadline: Friday, Jan. 28, at 11:59pm.

Submission: You must submit your solutions as a PDF file through MarkUs[1]. You can produce the file however you like (e.g. LaTeX, Microsoft Word, scanner), as long as it is readable.

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 homework mark (worth 1 pt) may be removed for a lack of neatness.*

You may notice that some questions are worth 0 pt, which means we will not mark them in this Homework. 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 Yongchao Zhou and Shervin Mehryar. Send your email with subject "*[CSC413] HW1 ...*" to `csc413-2022-01-tas@cs.toronto.edu` or post on Piazza with the tag `hw1`.

1 Hard-Coding Networks

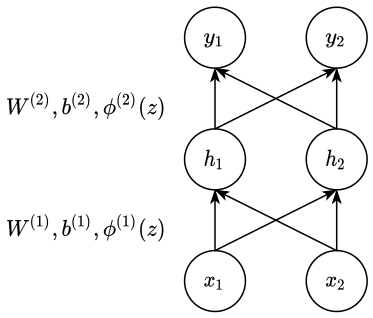
Can we use neural networks to tackle coding problems? Yes! In this question, you will build a neural network to find the k^{th} smallest number from a list using two different approaches: sorting and counting (Optional). You will start by constructing a two-layer perceptron "Sort_2" to sort two numbers 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/2022/assets/readings/L02a.pdf` may be useful.

1.1 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:

[1] `https://markus.teach.cs.toronto.edu/2022-01/`
[2] `https://uoft-csc413.github.io/2022/assets/misc/syllabus.pdf`
[3] `https://piazza.com/class/ky8yug7i9ol3b1`



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: $\phi^{(1)}(z)$, $\phi^{(2)}(z)$

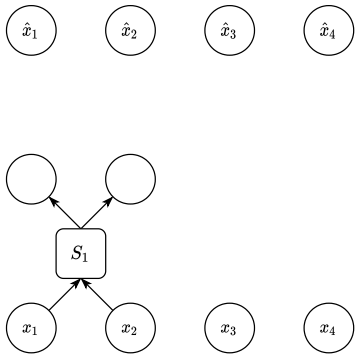
You do not need to show your work.

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

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

1.2 Perform Sort [1.5pt]

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 $\hat{x}_1 \le \hat{x}_2 \le \hat{x}_3 \le \hat{x}_4$ 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.



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

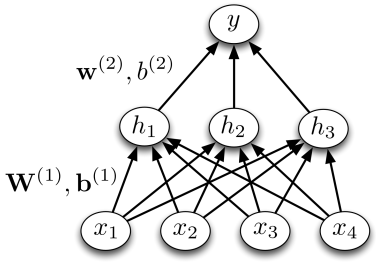
1.3 Find the k^{th} smallest number [0pt]

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

1.4 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:

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

2 Backpropagation

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/2022/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]

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]

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 [4]*. You can use* $\text{softmax}'(\mathbf{y})$ *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.

[4] 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.

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

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 [1pt]

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]

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

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

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]

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]

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 Linear Regression

The reading on linear regression located at `https://uoft-csc413.github.io/2022/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.

3.1 Deriving the Gradient [0pt]

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

3.2 Underparameterized Model

3.2.1 [0.5pt]

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.

3.2.2 [0.5pt]

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 n and $\boldsymbol{\epsilon}$, as:

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

Show this is true by substituting your answer from 3.2.1 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 and σ.

Hints: you might find the cyclic property [5] *of trace useful.*

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

3.3 Overparameterized Model

3.3.1 [0pt]

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.

3.3.2 [1pt]

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 $\hat{\mathbf{w}} = X^\top (XX^\top)^{-1}\mathbf{t}$, assuming that the gradient is spanned by the rows of X and write $\hat{\mathbf{w}} = X^\top \mathbf{a}$ for some $\mathbf{a} \in \mathbb{R}^n$ in the stationary condition for gradient decent. Show your work.

3.3.3 [0pt]

Repeat part 3.2.2 for the overparameterized case.

3.3.4 [0.5pt]

Visualize and compare underparameterized with overparameterized polynomial regression: `https://colab.research.google.com/github/uoft-csc413/2022/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?

3.3.5 [0pt]

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.