Homework 1

Version: 1.1
Version Release Date: 2021-01-17
Changes by Version: (v1.1) $\mathbf{g}_2 = \mathbf{h} \odot \mathbf{g}_1$, same assumptions for 1.2 as 1.1
Deadline: Thursday, 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.*

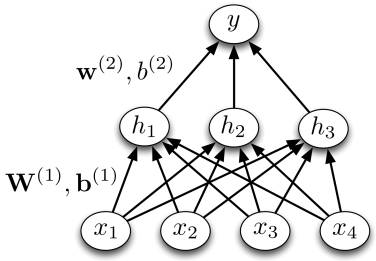
The teaching assistants for this assignment are Jonathan Lorraine and Mustafa Ammous.
`mailto:csc413-2021-01-tas@cs.toronto.edu`

1 Hard-Coding Networks

The reading on multilayer perceptrons located at `https://csc413-uoft.github.io/2021/assets/readings/L02a.pdf` may be useful for this question.

1.1 Verify Element in List [1pt]

In this problem, you need to find a set of weights and biases for a multilayer perceptron that determines if an input is in a list of length 3. You receive an input list with three elements x_1, x_2, x_3, and fourth input x_4 where $x_i \in \mathbb{Z}$. If $x_4 = x_j$ for $j = 1, 2$ or 3, the network will out 1, and 0 otherwise. You may assume all elements in the input list are distinct for simplicity. You will use the following architecture:



All of the hidden units and the output unit use an indicator activation function:

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

Please give a set of weights and biases for the network which correctly implements this function. Your answer should include:

[1] `https://markus.teach.cs.toronto.edu/csc413-2021-01/main`
[2] `https://csc413-uoft.github.io/2021/assets/misc/syllabus.pdf`
[3] `https://piazza.com/class/kjt32fc0f7y3kb`

- A 3×4 weight matrix $\mathbf{W}^{(1)}$ for the hidden layer
- A 3-dimensional vector of biases $\mathbf{b}^{(1)}$ for the hidden layer
- A 3-dimensional weight vector $\mathbf{w}^{(2)}$ for the output layer
- A scalar bias $b^{(2)}$ for the output layer

You do not need to show your work.

1.2 Verify Permutation [1pt]

Describe how to implement a neural network that takes 6 inputs $x_1, \ldots, x_6$ and verifies if the last 3 elements are a permutation of the first 3 elements. You may assume the elements in each list are distinct integers for simplicity. Specifically, we want to see if $[x_1, x_2, x_3] = [x_4, x_5, x_6]\mathbf{P}$ for some permutation matrix $\mathbf{P}$. Describe how to compose smaller, modular networks like the one you made in Section 1.1. You do not need to explicitly give the weight matrices or biases for the entire function.

1.3 Optional - Perform Sort [0pt]

Describe 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. In other words, $\hat{x}_1 \leq \hat{x}_2 \leq \hat{x}_3 \leq \hat{x}_4$, and each $\hat{x}_i$ is a distinct x_j. Implement $\hat{f}$ using a feedforward or recurrent neural network with element-wise activations. You may combine information across nodes via summation as in 1.1, or with multiplication.

Hint: There are multiple solutions. You could brute-force the answer by attempting to verify every permutation of the input, or you could implement a more scalable sorting algorithm where each hidden layer i is the algorithms state at step i.

2 Backpropagation

The reading on backpropagation located at `https://csc413-uoft.github.io/2021/assets/readings/L02b.pdf` may be useful for this question.

2.1

Consider a neural network defined with the following procedure:

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

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 $\text{softmax}(\mathbf{y}) = \frac{\exp(\mathbf{y})}{\sum_{i=1}^{M}\exp(\mathbf{y}_i)}$. Here, $\odot$ denotes element-wise multiplication.

2.1.1 Computational Graph [0pt]

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

2.1.2 Backward Pass [1pt]

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

2.2 Automatic Differentiation

Consider the function $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$ where $\mathcal{L}(\mathbf{x}) = \mathbf{x}^\top \mathbf{v}\mathbf{v}^\top \mathbf{x}$, 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:

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

2.2.1 Compute Hessian [0pt]

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

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

2.2.2 Computation Cost [1pt]

What is the number of scalar multiplications and memory cost of computing the Hessian $\mathbf{H}$ in terms of n?

2.3 Vector-Hessian Products [1pt]

Compute $\mathbf{z} = \mathbf{H}\mathbf{y} = \mathbf{v}\mathbf{v}^\top\mathbf{y}$ where $n = 3$, $\mathbf{v}^\top = [1, 2, 3]$, $\mathbf{y}^\top = [1, 1, 1]$ 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 $\mathbf{y}$. What is the time and memory cost of evaluating $\mathbf{z}$ with backpropagation (reverse-mode) in terms of n? What about forward-mode?

2.4 Trade-off of Reverse- and Forward-mode Autodiff [1pt]

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 the time and memory cost of 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" v.s. "wide".)

3 Linear Regression

The reading on linear regression located at `https://csc413-uoft.github.io/2021/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 [1pt]

First consider the underparameterized $d < n$ case. Write down the solution obtained by gradient descent assuming training converges. Show your work. Is the solution unique?

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. Write down the solution and show your work.

3.3.3 [1pt]

Visualize and compare underparameterized with overparameterized polynomial regression: `https://colab.research.google.com/drive/1Atkk9hjUaXV-bDttCxAv9WiMsB6EJTKU`. 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?