CS4803-7643: Deep Learning
Fall 2020
Problem Set 2

Instructor: Dhruv Batra
TAs: Sameer Dharur, Joanne Truong, Yihao Chen, Michael Piseno,
Hrishikesh Kale, Tianyu Zhan, Pravhav Chawla, Guillermo Nicolas Grande.

Discussions: https://piazza.com/gatech/fall2020/cs48037643

Due: Wednesday, September 23, 11:59pm

Instructions

1. We will be using Gradescope to collect your assignments. Please read the following instructions for submitting to Gradescope carefully!

   - For the **HW2** component on Gradescope, you could upload one single PDF containing the answers to all the theory questions and the completed Jupyter notebooks for the coding problems. **However, the solution to each problem or subproblem must be on a separate page. When submitting to Gradescope, please make sure to mark the page(s) corresponding to each problem/sub-problem.** Likewise, the pages of the Jupyter notebooks must also be marked to their corresponding subproblems.
   - For the **HW2 Code** component on Gradescope, please use the `collect_submission.sh` script provided and upload the resulting **hw2_code.zip** here. Please make sure you have saved the most recent version of your Jupyter notebook before running this script.
   - Note: This is a large class and Gradescope's assignment segmentation features are essential. Failure to follow these instructions may result in parts of your assignment not being graded. We will not entertain regrading requests for failure to follow instructions.

2. LaTeX'd solutions are strongly encouraged (solution template available at cc.gatech.edu/classes/AY2021/cs7643_fall/assets/sol2.tex), but scanned handwritten copies are acceptable. Hard copies are **not** accepted.

3. We generally encourage you to collaborate with other students.

   You may talk to a friend, discuss the questions and potential directions for solving them. However, you need to write your own solutions and code separately, and *not* as a group activity. Please list the students you collaborated with.

1 Gradient Descent

1. (3 points) We often use iterative optimization algorithms such as Gradient Descent to find $\mathbf{w}$ that minimizes a loss function $f(\mathbf{w})$. Recall that in gradient descent, we start with an initial value of $\mathbf{w}$ (say $\mathbf{w}^{(1)}$) and iteratively take a step in the direction of the negative of the gradient of the objective function *i.e.*

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla f(\mathbf{w}^{(t)}) \tag{1}$$

for learning rate $\eta > 0$.

In this question, we will develop a slightly deeper understanding of this update rule, in particular for minimizing a convex function $f(\mathbf{w})$. Note: this analysis will not directly carry over to training neural networks since loss functions for training neural networks are typically not convex, but this will (a) develop intuition and (b) provide a starting point for research in non-convex optimization (which is beyond the scope of this class).

Recall the first-order Taylor approximation of f at $\mathbf{w}^{(t)}$:

$$f(\mathbf{w}) \approx f(\mathbf{w}^{(t)}) + \langle \mathbf{w} - \mathbf{w}^{(t)}, \nabla f(\mathbf{w}^{(t)}) \rangle \tag{2}$$

When f is convex, this approximation forms a lower bound of f, *i.e.*

$$f(\mathbf{w}) \geq \underbrace{f(\mathbf{w}^{(t)}) + \langle \mathbf{w} - \mathbf{w}^{(t)}, \nabla f(\mathbf{w}^{(t)}) \rangle}_{\text{affine lower bound to } f(\cdot)} \quad \forall \mathbf{w} \tag{3}$$

Since this approximation is a 'simpler' function than $f(\cdot)$, we could consider minimizing the approximation instead of $f(\cdot)$. Two immediate problems: (1) the approximation is affine (thus unbounded from below) and (2) the approximation is faithful for $\mathbf{w}$ close to $\mathbf{w}^{(t)}$. To solve both problems, we add a squared ℓ_2 *proximity term* to the approximation minimization:

$$\operatorname*{argmin}_{\mathbf{w}} \underbrace{f(\mathbf{w}^{(t)}) + \langle \mathbf{w} - \mathbf{w}^{(t)}, \nabla f(\mathbf{w}^{(t)}) \rangle}_{\text{affine lower bound to } f(\cdot)} + \underbrace{\frac{\lambda}{2}}_{\text{trade-off}} \underbrace{\left|\left| \mathbf{w} - \mathbf{w}^{(t)} \right|\right|^2}_{\text{proximity term}} \tag{4}$$

Notice that the optimization problem above is an unconstrained quadratic programming problem, meaning that it can be solved in closed form (hint: gradients).

What is the solution $\mathbf{w}^*$ of the above optimization? What does that tell you about the gradient descent update rule? What is the relationship between λ and η?

2. (4 points) Let's prove a lemma that will initially seem devoid of the rest of the analysis but will come in handy in the next sub-question when we start combining things. Specifically, the analysis in this sub-question holds for any $\mathbf{w}^\star$, but in the next sub-question we will use it for $\mathbf{w}^\star$ that minimizes $f(\mathbf{w})$.

Consider a sequence of vectors $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_T$, and an update equation of the form $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \mathbf{v}_t$ with $\mathbf{w}^{(1)} = 0$. Show that:

$$\sum_{t=1}^{T} \langle \mathbf{w}^{(t)} - \mathbf{w}^\star, \mathbf{v}_t \rangle \leq \frac{\|\mathbf{w}^\star\|^2}{2\eta} + \frac{\eta}{2} \sum_{t=1}^{T} \|\mathbf{v}_t\|^2 \tag{5}$$

3. (4 points) Now let's start putting things together and analyze the convergence rate of gradient descent *i.e.* how fast it converges to $\mathbf{w}^\star$.

   First, show that for $\bar{\mathbf{w}} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{w}^{(t)}$

$$f(\bar{\mathbf{w}}) - f(\mathbf{w}^\star) \leq \frac{1}{T}\sum_{t=1}^{T}\langle \mathbf{w}^{(t)} - \mathbf{w}^\star, \nabla f(\mathbf{w}^{(t)})\rangle \tag{6}$$

   Next, use the result from part 2, with upper bounds B and ρ for $\|\mathbf{w}^\star\|$ and $\left\|\nabla f(\mathbf{w}^{(t)})\right\|$ respectively and show that for fixed $\eta = \sqrt{\frac{B^2}{\rho^2 T}}$, the convergence rate of gradient descent is $\mathcal{O}(1/\sqrt{T})$ *i.e.* the upper bound for $f(\bar{\mathbf{w}}) - f(\mathbf{w}^\star) \propto \frac{1}{\sqrt{T}}$.

2 Estimating Hessians [Extra credit for 4803 and 7643]

4. (6 points) Optimization is an extremely important part of deep learning. In the previous question, we explored gradient descent, which uses the direction of maximum change to minimize a loss function. However, gradient descent leaves a few questions unresolved – how do we choose the learning rate η? If η is small, we will take a long time to reach the optimal point; if η is large, it will oscillate between one side of the curve and another. So what should we do?

   One solution is to use Hessians, which is a measure of curvature, or the rate of change of the gradients. Intuitively, if we knew how steep a curve were, we would know how fast we should move in a given direction. This is the intuition behind second-order optimization methods such as Newton's method.

   Let us formally define a Hessian matrix $\mathbf{H}$ of a function f as a square $n \times n$ matrix containing all second partial derivatives of f, i.e.:

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

   Recall the second-order Taylor approximation of f at $\mathbf{w}^{(t)}$:

$$f(\mathbf{w}) \approx f(\mathbf{w}^{(t)}) + \langle \mathbf{w} - \mathbf{w}^{(t)}, \nabla f(\mathbf{w}^{(t)})\rangle + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{(t)})^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^{(t)}) \tag{7}$$

   (a) What is the solution to the following optimization problem?

$$\underset{\mathbf{w}}{\operatorname{argmin}} \left[f(\mathbf{w}^{(t)}) + \langle \mathbf{w} - \mathbf{w}^{(t)}, \nabla f(\mathbf{w}^{(t)})\rangle + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{(t)})^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^{(t)}) \right] \tag{8}$$

   What does that tell you about how to set the learning rate η in gradient descent?

   Now that we've derived Netwon's update algorithm, we should also mention that there is a catch to using Newton's method. Newton's method requires us to 1) calculate $\mathbf{H}$, and 2) invert $\mathbf{H}$. Having to compute a Hessian is expensive; $\mathbf{H}$ is massive and we would also have to figure out how to store it.

(b) Consider an MLP with 3 fully-connected layers, each with 50 hidden neurons, except for the output layer, which represents 10 classes. We can represent the transformations as $\mathbf{x} \in \mathbb{R}^{50} \longrightarrow \mathbf{h}^{(1)} \in \mathbb{R}^{50} \longrightarrow \mathbf{h}^{(2)} \in \mathbb{R}^{50} \longrightarrow \mathbf{s} \in \mathbb{R}^{10}$. Assume that $\mathbf{x}$ does not include any bias feature appended to it. How many parameters are in this MLP? What is the size of the corresponding Hessian?

Rather than store and manipulate the Hessian $\mathbf{H}$ directly, we will instead focus on being able to compute the result of a Hessian-vector product $\mathbf{Hv}$, where $\mathbf{v}$ is an arbitrary vector. Why? Because in many cases one does not need the full Hessian but only $\mathbf{Hv}$. Computing $\mathbf{Hv}$ is a core building block for computing a number of quantities including $\mathbf{H}^{-1}\nabla f$ (hint, hint). You will next show a surprising result that it is possible to 'extract information from the Hessian', specifically to compute the Hessian-vector product without ever explicitly calculating or storing the Hessian itself!

Consider the Taylor series expansion of the gradient operator about a point in weight space:

$$\nabla_{\mathbf{w}}(\mathbf{w} + \Delta\mathbf{w}) = \nabla_{\mathbf{w}}(\mathbf{w}) + \mathbf{H}\Delta\mathbf{w} + O(||\Delta\mathbf{w}||^2) \tag{9}$$

where $\mathbf{w}$ is a point in weight space, $\Delta\mathbf{w}$ is a perturbation of $\mathbf{w}$, ∇_w is the gradient, and $\nabla_{\mathbf{w}}(\mathbf{w} + \Delta\mathbf{w})$ is the Jacobian matrix of $\mathbf{w} + \Delta\mathbf{w}$ w.r.t. $\mathbf{w}$.

If you have difficulty understanding this expression above, consider starting with Eqn (2), replacing $\mathbf{w} - \mathbf{w}^{(t)}$ with $\Delta\mathbf{w}$ and $f(\cdot)$ with $\nabla_{\mathbf{w}}(\cdot)$.

(c) Use Eqn (9) to derive a numerical approximation of $\mathbf{Hv}$ (in terms of $\nabla_{\mathbf{w}}$).
Hint: Consider choosing $\Delta\mathbf{w} = r\mathbf{v}$, where $\mathbf{v}$ is a vector and r is a small number.

Let's now define a useful operator, known as the $\mathcal{R}$-operator. The $\mathcal{R}$-operator with respect to $\mathbf{v}$ is defined as:

$$\mathcal{R}_{\mathbf{v}}\{f(\mathbf{w})\} = \frac{\partial}{\partial r} f(\mathbf{w} + r\mathbf{v})\Big|_{r=0} \tag{10}$$

(d) The $\mathcal{R}$-operator has many useful properties. Let's first prove some of them. Show that:

$$\mathcal{R}_{\mathbf{v}}\{cf(\mathbf{w})\} = c\mathcal{R}_{\mathbf{v}}\{f(\mathbf{w})\} \quad \text{[Linearity under scalar multiplication]}$$
$$\mathcal{R}_{\mathbf{v}}\{f(\mathbf{w})g(\mathbf{w})\} = \mathcal{R}_{\mathbf{v}}\{f(\mathbf{w})\}g(\mathbf{w}) + f(\mathbf{w})\mathcal{R}_{\mathbf{v}}\{g(\mathbf{w})\} \quad \text{[Chain Rule of R-operators]}$$

(e) Now, instead of numerically approximating $\mathbf{Hv}$, use the $\mathcal{R}$-operator to derive an equation to exactly calculate $\mathbf{Hv}$.

(f) Explain how might you implement $\mathbf{Hv}$ in MLPs if you already have access to an auto-differentiation library.

3 Automatic Differentiation

5. (4 points) In practice, writing the closed-form expression of the derivative of a loss function f w.r.t. the parameters of a deep neural network is hard (and mostly unnecessary) as f becomes complex. Instead, we define computation graphs and use the automatic differentiation algorithms (typically backpropagation) to compute gradients using the chain rule. For example, consider the expression

$$f(x, y) = (x + y)(y + 1) \tag{11}$$

Let's define intermediate variables a and b such that

$$a = x + y \tag{12}$$
$$b = y + 1 \tag{13}$$
$$f = a \times b \tag{14}$$

] A computation graph for the "forward pass" through f is shown in Fig. 1.

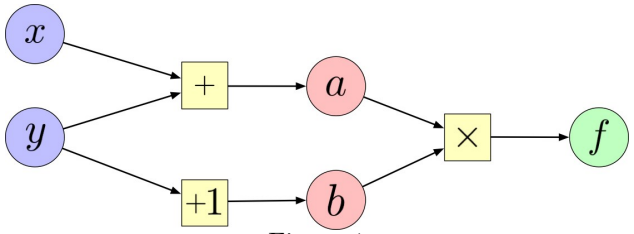

Figure 1

We can then work backwards and compute the derivative of f w.r.t. each intermediate variable ($\frac{\partial f}{\partial a}$, $\frac{\partial f}{\partial b}$) and chain them together to get $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$.

Let $\sigma(\cdot)$ denote the standard sigmoid function. Now, for the following vector function:

$$f_1(w_1, w_2) = e^{e^{w_1} + e^{2w_2}} + \sin(e^{w_1} + e^{2w_2}) \tag{15}$$
$$f_2(w_1, w_2) = w_1 w_2 + \sigma(w_1) \tag{16}$$

(a) Draw the computation graph. Compute the value of f at $\vec{w} = (1, 2)$.

(b) At this $\vec{w}$, compute the Jacobian $\frac{\partial \vec{f}}{\partial \vec{w}}$ using numerical differentiation (using $\Delta w = 0.01$).

(c) At this $\vec{w}$, compute the Jacobian using forward mode auto-differentiation.

(d) At this $\vec{w}$, compute the Jacobian using backward mode auto-differentiation.

(e) Don't you love that software exists to do this for us?

4 Convolutions

6. (5 points) We'll start to introduce the properties of convolutions here that serve as a foundation for many computer vision applications in deep learning. In class, we discussed convolutions. In this question, we will develop formal intuition around a slight modification of that idea – circular convolutions.

   First, let's define a circular convolution of two n-dimensional vectors $\mathbf{x}$ and $\mathbf{w}$:

$$(\mathbf{x} * \mathbf{w})_i = \sum_{k=0}^{n-1} x_k w_{(i-k) \bmod n} \tag{17}$$

   We can write the above equation as a matrix-vector multiplication.

Given an n-dimensional vector $\mathbf{a} = (a_0, ..., a_{n-1})$, we define the associated matrix $C_\mathbf{a}$ whose first column is made up of these numbers, and each subsequent column is obtained by a circular shift of the previous column.

$$C_\mathbf{a} = \begin{bmatrix} a_0 & a_{n-1} & a_{n-2} & \dots & a_1 \\ a_1 & a_0 & a_{n-1} & & a_2 \\ a_2 & a_1 & a_0 & & a_3 \\ \vdots & & \ddots & \ddots & \vdots \\ a_{n-1} & a_{n-2} & a_{n-3} & \dots & a_0 \end{bmatrix}$$

Such matrices are called *circulants*. Any convolution $\mathbf{x} * \mathbf{w}$ can be equivalently represented as a multiplication by the circulant matrix $C_\mathbf{a} x$.

Note that a circulant matrix is a kind of Toeplitz matrix with the additional property that $a_i = a_{i+n}$. Next, let's introduce a special type of circulant called a shift matrix. A shift matrix is a circulant matrix where only one dimension of the vector $\mathbf{a}$ can be set to 1, *i.e.*, $\mathbf{a} = (0, 1, ..., 0)$. Let S be the circular right-shift operator, defined by the following action on vectors:

$$S\mathbf{x} = \begin{bmatrix} 0 & & & 1 \\ 1 & & & \\ & \ddots & \ddots & \\ & & 1 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} x_{n-1} \\ x_0 \\ \vdots \\ x_{n-2} \end{bmatrix}$$

Notice that after applying the shift-matrix, all the element of $\mathbf{x}$ have been shifted by 1.

(a) Prove that *any* circulant matrix is commutative with a shift matrix. Note that this directly implies that convolutions are commutative with shift operators.

This leads to very important property called translation or shift equivariance. A function is shift equivariant if $f(S\mathbf{x}) = Sf(\mathbf{x})$. Convolution's commutativity with shift implies that it does not matter whether we first shift a vector and then convolve it, or first convolve and then shift – the result will be the same. Notice that you just proved that circular convolutions are shift equivariant.

(b) Now prove that the a (circular) convolution is the *only* linear operation with shift equivariance. (Hint: how do you prove a bidirectional implication?)

(c) (Open-ended question) What does this tell you about designing deep learning architectures for processing spatial or spatio-temporal data like images and videos?

5 Paper Review [Extra credit for 4803, regular credit for 7643]

The paper we will study in this homework is **'Understanding deep learning requires rethinking generalization'**, presented at the International Conference on Learning Representations (ICLR) in 2017.

The paper presents a set of interesting experiments and results to explore the phenomenon of generalization in deep neural networks, i.e, the difference in performance on the training and test sets, and the role of explicit and implicit regularization towards achieving this.

The paper can be viewed here.

The evaluation rubric for this section is as follows :

7. **[2 points]** Briefly summarize the key contributions, strengths and weaknesses of this paper.
8. **[2 points]** What is your personal takeaway from this paper? This could be expressed either in terms of relating the approaches adopted in this paper to your traditional understanding of learning parameterized models, or potential future directions of research in the area which the authors haven't addressed, or anything else that struck you as being noteworthy.

Guidelines: Please restrict your reviews to no more than 350 words (total length for answers to both the above questions).

6 Implement and train a network on CIFAR-10

9. (Upto 20 points) In PS1, you learned how to implement a softmax classifier and vanilla neural networks. Now, we will learn how to implement ConvNets. You will begin by writing the forward and backward passes for convolution and pooling, and then go on to train a shallow ConvNet on the CIFAR-10 dataset in Python. Next you will learn to use PyTorch, a popular open-source deep learning framework, and use it to replicate the experiments from before.

   Follow the instructions provided here