Problem Set 1

Due: 10/04/2024 by 11:59 pm

Work on this assignment is individual. No group work is allowed.

Submission: Your submission on eLearning is a writeup in PDF format. We've provided enough space for the answers, don't write/type on the backside of any pages. Late homework will not be accepted.

Problem 1: Theory (50pt)

To answer questions in this part, you need some basic knowledge of linear algebra and matrix calculus. Also, you need to follow the instructions:

1. Every vector is treated as column vector.
2. You need to use the numerator-layout notation for matrix calculus. Please refer to Wikipedia about the notation.
3. You are only allowed to use vector and matrix. You cannot use tensor in any of your answer.
4. Missing transpose are considered as wrong answer.

1.1 Two-Layer Neural Nets

You are given the following neural net architecture:

$$Linear_1 \rightarrow f \rightarrow Linear_2 \rightarrow g$$

where $Linear_i(x) = W^i x + b$ is the i^{th} affine transformation, and f, g are element-wise nonlinear activation functions. When an input $x \in \mathrm{R}^n$ is fed to the network, $y' \in \mathrm{R}^K$ is obtained as the output.

1.2 Regression Task

We would like to perform regression task. We choose $f(\cdot) = (\cdot)^+ = \mathrm{ReLU}(\cdot)$ and g to be the identity function. To train this network, we choose MSE loss function $\ell_{MSE}(\hat{y}, y) = \left|\left|\hat{y}, -u\right|\right|^2$, where y is the target output. (25 pts)

(a) Name and mathematically describe the 5 programming steps you would take to train this model with PyTorch using SGD on a single batch of data.

(b) For a single data point (x, y), write down all inputs and outputs for forward pass of each layer. You can only use: $\{x, y, W^1, b^1, W^2, b^2\}$ in your answer. (Note that $Linear_i(x) = W^i x + b$)

Layer	Input	Output
$\mathtt{Linear}_1$		
f		
$\mathtt{Linear}_2$		
g		
Loss		

(c) Write down the gradient calculated from the backward pass. You can only use the following variables: $\left\{x, y, W^1, b^1, W^2, b^2, \frac{\partial \ell}{\partial \hat{y}}, \frac{\partial z_2}{\partial z_1}, \frac{\partial \hat{y}}{\partial z_3}\right\}$ in your answer, where $z_1, z_2, z_3, \hat{y}$ are the outputs of $Linear_1, f, Linear_2, g$.

Parameter	Gradient
$\boldsymbol{W}^{(1)}$	
$\boldsymbol{b}^{(1)}$	
$\boldsymbol{W}^{(2)}$	
$\boldsymbol{b}^{(2)}$	

(d) Show us the elements of $\frac{\partial \ell}{\partial \hat{y}}$, $\frac{\partial z_2}{\partial z_1}$ and $\frac{\partial \hat{y}}{\partial z_3}$ (be careful about the dimensionality)?

1.3 Classification Task

We would like to perform multi-class classification task, so we set both f, g = σ, the logistic sigmoid function $\sigma(z) := \frac{1}{1 + \exp(-z)}$.(25 pts)

(a) If you want to train this network, what do you need to change in the equations of (b), (c) and (d), assuming we are using the same MSE loss function.

(b) Now you think you can do a better job by using a Binary Cross Entropy (BCE) loss function $\ell_{\mathrm{BCE}}(\hat{y}, y) = \frac{1}{K}\Sigma_{i=1}^{K} - [y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$. What do you need to change in the equations of (b), (c) and (d)?

(c) Things are getting better. You realize that not all intermediate hidden activations need to be binary (or soft version of binary). You decide to use $f(\cdot) = (\cdot)^+$ but keep g as σ. Explain why this choice of f can be beneficial for training a (deeper) network.

Problem 2: Implementation (50 pts)

You need to implement the forward pass and backward pass for Linear, ReLU, Sigmoid, MSE loss, and BCE loss in the attached mlp.py file. We provide three example test cases test1.py, test2.py, test3.py. We will test your implementation with other hidden test cases, so please create your own test cases to make sure your implementation is correct.

Note: We will grade you only based on forward pass on all functions and backward pass on Linear. Backward pass on other functions will be used as extra credit.

Extra instructions:

1. Please use Python version ≥ 3.7 and PyTorch version 1.7.1. We recommend you use Miniconda the manage your virtual environment.
2. We will put your mlp.py file under the same directory of the hidden test scripts and use the command python hiddenTestScriptName.py to check your implementation. So please make sure the file name is mlp.py and it can be executed with the example test scripts we provided.
3. You are not allowed to use PyTorch autograd functionality in your implementation.
4. Be careful about the dimensionality of the vector and matrix in PyTorch. It is not necessarily follow the Math you got from part 1.

Bonus Problem: Perceptron Learning (40 pts)

Consider the data set (perceptron.data) attached to this homework. This data file consists of M rows of data elements of the form $(x_1^{(m)}, x_2^{(m)}, x_3^{(m)}, x_4^{(m)}, y^{(m)})$ where $x_1^{(m)}, \ldots, x_4^{(m)} \in \mathbb{R}$ define a data point in $\mathbb{R}^4$ and $y^{(m)} \in \{-1,1\}$ is the corresponding class label.

In this problem, you are to implement the perceptron algorithm (no need to upload the code) using two different subgradient descent strategies. Report the gradient function for the weight w and the bias b. For each strategy below, report the number of iterations that it takes to find a perfect classifier for the data, the values of w and b for the first three iterations, and the final weights and bias (or "diverge" if absolute values of parameters gradient are larger than 10^{-5}). Each descent procedure should start from the initial point

Additional Information:

Standard subgradient descent is a generalization of gradient descent used to minimize non-differentiable convex functions. Unlike gradient descent, which relies on the gradient (the vector of partial derivatives) of a differentiable function, subgradient descent uses a subgradient, which exists for convex, non-differentiable functions.

Here's how it works:

1. Subgradient: For a convex, non-differentiable function $f(x)$, a subgradient at a point x is a vector g that satisfies:
$$f(y) \geq f(x) + g^T(y - x) \text{ for all } y$$
This generalizes the idea of the gradient for convex functions.

2. Subgradient Descent Algorithm:
•Start with an initial guess x_0 .
•At each *step* k , compute a subgradient g_k of the function at x_k .
•Update the current solution using:
$$x_{k+1} = x_k - \alpha_k g_k$$
where α_k is a step size (or learning rate), which can be fixed or decreasing over time.

3. Convergence: Subgradient descent typically has a slower convergence rate compared to gradient descent, especially if the function is non-smooth. However, if the step size is chosen properly, the algorithm can still converge to an optimal solution.

Stochastic Subgradient Descent (SSD) is an extension of Stochastic Gradient Descent (SGD) that is used to optimize non-differentiable convex functions. In standard gradient-based methods like SGD, we assume the objective function is differentiable, meaning it has well-defined gradients everywhere. However, many optimization problems, especially in machine learning (such as those involving L1 regularization or hinge loss), are non-differentiable at certain points.

In SSD, instead of computing gradients, we compute subgradients at points where the function is not differentiable. A subgradient is a generalization of the gradient for non-differentiable functions and gives a direction in which the function decreases.

The process of Stochastic Subgradient Descent involves:

1. Randomly selecting a subset of data points (a stochastic step).
2. Computing a subgradient for the current point.
3. Updating the model parameters using the subgradient.

The update rule in SSD is similar to that in SGD:

$$\theta_{t+1} = \theta_t - \alpha_t \, g_t$$

Where:

- θ_t are the model parameters at step t,
- α_t is the learning rate at step t,
- g_t is a subgradient of the objective function at step t.

SSD is particularly useful when working with non-differentiable optimization problems, but like SGD, it requires careful tuning of the learning rate and other hyperparameters to converge properly.

$$w^0 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad b^0 = 0.$$

1. Standard sub-gradient descent with the step size $\gamma_t = 1$ for each iteration. (15pts)

2. Stochastic sub-gradient descent where exactly one component of the sum is chosen to approximate the gradient at each iteration. Instead of picking a random component at each iteration, you should iterate through the data set starting with the first element, then the second, and so on until the Mth element, at which point you should start back at the beginning again. Again, use the step size $\gamma_t = 1$. (15 pts)

3. How does the rate of convergence change as you change the step size? Provide one example step sizes to back up your statements (e.g., total number of iterations and loss value figures). (10 pts)