CS69000-DPL Spring 2023 Homework — Homework 2— Last update: February 27, 2023

https://www.cs.purdue.edu/homes/ribeirob/courses/Spring2023 Due: **6:00pm**, Friday, March 10th (open until 6:00am next day)

Name _____ PUID _____

Instructions and Policy: Each student should write up their own solutions independently, no copying of any form is allowed. You MUST to indicate the names of the people you discussed a problem with; ideally you should discuss with no more than two other people.
YOU MUST INCLUDE YOUR NAME IN THE HOMEWORK
You need to submit your answer in PDF. LaTeX is typesetting is encouraged but not required. Please write clearly and concisely - clarity and brevity will be rewarded. Refer to known facts as necessary.

Q0 (0pts correct answer, -1,000pts incorrect answer: (0,-1,000) pts): A correct answer to the following questions is worth 0pts. An incorrect answer is worth -1,000pts, which carries over to other homeworks and exams, and can result in an F grade in the course.

(1) Student interaction with other students / individuals:

   (a) I have copied part of my homework from another student or another person (plagiarism).

   (b) Yes, I discussed the homework with another person but came up with my own answers. Their name(s) is (are) _____

   (c) No, I did not discuss the homework with anyone

(2) On using online resources:

   (a) I have copied one of my answers directly from a website (plagiarism).

   (b) I have used online resources to help me answer this question, but I came up with my own answers (you are allowed to use online resources as long as the answer is your own). Here is a list of the websites I have used in this homework:
   _____

   (c) I have not used any online resources except the ones provided in the course website.

Homework 2: Stochastic Gradient Descent; Adaptive Learning Rates; Loss Surface; Metropolis Hastings & Hamiltonian Monte Carlo

Learning Objectives: Let students understand Stochastic Gradient Descent (SGD) and optimization algorithms for adaptive learning rates. Students will also learn about the loss surface of neural networks and Bayesian deep learning with Metropolis Hastings & Hamiltonian Monte Carlo.

Learning Outcomes: After you finish this homework, you should be capable of explaining and implementing SGD with minibatch from scratch and realize algorithms for adaptive learning rates. Students should be able to understand the importance of flat minima in the generalization performance of neural networks. Students should also be able to implement Hamiltonian Monte Carlo algorithm from scratch for Bayesian deep learning.

1 Q1: Conceptual Part (2 pts)

Please answer the following questions **concisely**. All the answers, along with your name and email, should be clearly typed in some editing software, such as Latex or MS Word.

1. **(0.4 pt)** Minibatch Stochastic Gradient Descent (SGD) has been shown to give competitive results using deep neural networks in many tasks. Compared to the regular Gradient Descent (GD), the gradients computed by minibatch SGD are noisy. Can you explain why the noisy gradients need to be unbiased (i.e., in average, the noisy gradient is the true gradient). What would happen if the gradients had a bias?
   Hint: Think about SGD as a stochastic approximation, where we are trying to find the zeros of a function (zero gradient).

2. **(0.4 pt)** Early stopping uses the validation data to decide which parameters we should keep during our SGD optimization. Explain why models obtained by early stopping tend to generalize better than models obtained by running a fixed number of epochs. Also explain why early stopping should never use the training data.

3. **(0.4 pt)** How can Metropolis-Hastings be used for Bayesian deep learning? More specifically, give the peseudocode of a simple algorithm that uses Metropolis-Hastings to produce sampled parameters of $p(\mathbf{W}|\text{Data})$ of a neural network. (inefficient is OK and it is easier)
Hint: Your starting proposal can be just a small zero-mean multivariate Normal perturbation around an existing accepted sample (e.g., $\mathbf{W}^\star$).

4. **(0.8 pt)** Consider a one hidden-layer Multi Layer Perceptron with ReLU as activation without biases (for simplicity). The output $\hat{y} \in \mathbb{R}$ of this network for an input $x \in \mathbb{R}^d$ can be defined as $\hat{y} = f(x)$, where $f(x; W_1, W_2) := ReLU(W_2^T ReLU(W_1^T x))$, where $W_1 \in \mathbb{R}^{d \times d_1}, W_2 \in \mathbb{R}^{d_1 \times 1}$ are arbitrary weight matrices we will learn from data.

   (a) Prove that $\forall x \in \mathbb{R}^d$ and $\forall \alpha \in \mathbb{R}^+$ (positive real numbers), we have $f(x; W_1, W_2) = f(x; \alpha W_1, W_2/\alpha)$.

   (b) Consider the same setting as above. Let the negative log-likelihood be $L(W_1, W_2, x, y) = (f(x; W_1, W_2) - y)^2$ (hence, we are assuming a model with additive standard Gaussian noise as error). Assume (W_1^*, W_2^*) is a critical point for the negative log-likelihood, i.e., $\frac{\partial L}{\partial W_1}|_{(W_1, W_2) = (W_1^*, W_2^*)} = 0$ and $\frac{\partial L}{\partial W_2}|_{(W_1, W_2) = (W_1^*, W_2^*)} = 0$, where $W_1^{\alpha,*} = \alpha W_1^*$, $W_2^{\alpha,*} = W_2^*/\alpha$. Prove that $\forall \alpha \in \mathbb{R}^+$, $(W_1^{\alpha,*}, W_2^{\alpha,*})$ is also a critical point.
   Warning: The derivative of a ReLU function at point 0 is undefined. For simplicity, you do not need to consider this edge case.

(c) Using the same setting as (b), assume (W_1^*, W_2^*) is a local minima of the negative log-likelihood. We can show $(W_1^{\alpha,*}, W_2^{\alpha,*})$ is also a local minimum point (no need to prove this property), where $W_1^{\alpha,*} = \alpha W_1^*$, $W_2^{\alpha,*} = W_2^*/\alpha$. We define a sharp function using the Hessian's Frobenius norm

$$\text{sharp}(W_1, W_2) := \left\| \begin{bmatrix} \nabla^2_{W_1}(L(W_1, W_2, x, y)) & \mathbf{0} \\ \mathbf{0} & \nabla^2_{W_2}(L(W_1, W_2, x, y)) \end{bmatrix} \right\|_F,$$

which can be used as a proxy measure of the sharpness of the likelihood landscape near the local minimum of the negative log-likelihood (since the Hessian $\nabla^2_{W_h}(L(W_1, W_2, x, y))$, $h \in \{1, 2\}$, at a local minimum is usually positive definite, it measures the local curvature).

Function $f(x; W_1, W_2)$ is said to have a sharper loss at a local minimum (W_1^*, W_2^*) than another local minimum (W_1^{**}, W_2^{**}) if

$$\text{sharp}(W_1^*, W_2^*) > \text{sharp}(W_1^{**}, W_2^{**}).$$

Show that, for an arbitrary local minima (W_1, W_2), there is an arbitrary constant $\alpha > 0$ such that the sharpness of the loss function increases, i.e.,

$$\text{sharp}(W_1^{\alpha,*}, W_2^{\alpha,*}) > \text{sharp}(W_1^*, W_2^*).$$

Hint 1: Calculate the relationship between $\nabla^2_{W_1}(L(W_1, W_2, x, y))|_{(W_1,W_2)=(W_1^{\alpha,*}, W_2^{\alpha,*})}$ and $\nabla^2_{W_1}(L(W_1, W_2, x, y))|_{(W_1,W_2)=(W_1^*, W_2^*)}$ (also for $\nabla^2_{W_2}$). Both of them are (assumed to be) positive definite, which means positive eigenvalues. To calculate the sharpness, we can then use the property that trace of a matrix is equal to the sum of its eigenvalues to determine if there exists positive elements in the diagonal. From the definition of the Frobenius norm, we can easily see the norm is bounded below by the sum of any subset of positive element in the matrix. Finally, we can show how the sharpness can be increased by adjusting the positive element (with α) in the matrix to increase the Frobenius norm.

Hint 2 Frobenius norm of a $m \times n$ matrix $\mathbf{A}$ is defined as,

$$||\mathbf{A}||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}.$$

(d) This question will explain why covariance matrix of Metropolis-Hasting proposals are very important. Given the training data $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^n$ for a supervised learning model task $p(W_1, W_2|\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^n)$, where W_1, W_2 are the paremeters of the model $f(x; W_1, W_2)$ stated at the beginning of this question, we would like to apply the following Bayesian averaging procedure

$$p_{Bayesian}(\mathbf{y}|\mathbf{x}; \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^n) = \frac{1}{K}\sum_{i=1}^{k} p(\mathbf{y}|\mathbf{x}; (W_1^{(k)}, W_2^{(k)}))$$

using a Metropolis-Hastings (MH) procedure to obtain K independent samples of the posterior as follows

$$(W_1^{(k)}, W_2^{(k)}) \sim P(W_1, W_2|\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^n), \qquad k \in 1, 2, \ldots, K.$$

Assume we have calculated the rejection rate $\gamma = \frac{\text{Number of times the samples were rejected}}{\text{Number of sampled MH proposals}}$. Now assume the Metropolis-Hastings procedure has a sampling proposal with covariance matrix $\mathbf{I}$, i.e., $q(W_{1,t+1}|W_{1,t}) \sim \text{Normal}(W_{1,t}, \mathbf{I})$. Consider W_1, W_2 as vectors, assume we know (W_1^*, W_2^*) is a local minimum point as in 4(b), we further assume the initial state for the k-th MH sampling procedure is defined as $(W_{1,0}^{(k)}, W_{2,0}^{(k)}) = (W_1^{\alpha^{(k)},*}, W_2^{\alpha^{(k)},*})$, $\alpha^{(1)}, ..., \alpha^{(K)} \in \mathbb{R}^+$. Now assume $\alpha^{(1)} = 1$, you observe there exists $k \in 1, ..., K$, such that the k-th Metropolis-Hastings procedure has much higher rejection rate than the 1-st Metropolis-Hastings procedure. Can you give one possible explanation for this phenomenon?

Hint 1: Use the insights from Q4(b) about the sharpness of the likelihood at different local minimum points.

Hint 2: A 2D drawing of the phenomenon might help you explain it.

Programming Part

In this part, you are going to implement multiple (1) gradient descent variants and (2) regularization methods (3) HMC sampler for GMM parameters. The rule of thumbs is that you can do any changes, but files which are not marked (by red color in 1, e.g., "main.py") will be overwritten by TA in the test stage.

A GPU is essential for some of the tasks, thus make sure you follow the instruction to set up the GPU environment on "scholar.rcac.purdue.edu". A tutorial is available at
`https://www.cs.purdue.edu/homes/ribeirob/courses/Spring2023/howto/cluster-how-to.html`.

We provide a brief command summary for GPU environment setup on "scholar-fe04.rcac.purdue.edu" to "scholar-fe06.rcac.purdue.edu". Pay attention that you should use **Python 3.8, CUDA 10.1 and Pytorch 1.8.0** for best support.

```
module load learning/conda-5.1.0-py36-gpu
conda create -n DPLClass python=3.8 ipython ipykernel
source activate DPLClass
module load cuda/11.8.0
module load cudnn
conda install pytorch pytorch-cuda=11.7 -c pytorch -c nvidia
conda install torchvision -c pytorch
conda install seaborn
conda install scikit-learn
```

Note that every time you login to the scholar cluster, you should run the $1, 3, 4, 5$ commands again.

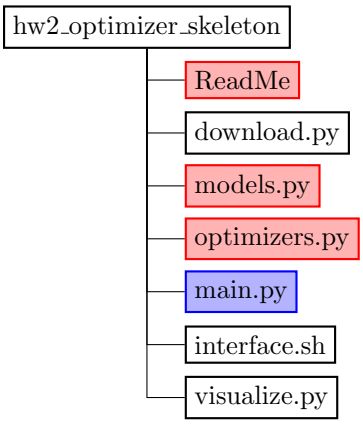
HW Overview

There are three tasks in this homework. The first two tasks, namely optimizers and regularizations task, will use the codebase named "hw2_optimizer_skeleton".

2 Q2: Optimizers and regularizations

You are going to create in a few new components to the HW2 package located at
`https://www.dropbox.com/scl/fo/d2fotkxx256vys64rc5rf/h?dl=0&rlkey=e2uya49wtvw7ii1agwpluzesc`.

Here is the folder structure that you should use:



- **hw2_optimizer_skeleton**: the top-level folder that contains all the files required in this homework.
- **ReadMe**: Your ReadMe should begin with a couple of **example commands**, e.g., "python hw2.py data", used to generate the outputs you report. TA would replicate your results with the commands provided here. More detailed options, usages and designs of your program can be followed. You can also list any concerns that you think TA should know while running your program. Note that put the information that you think it's more important at the top. Moreover, the file should be written in pure text format that can be displayed with Linux "less" command. You can refer to interface.sh for an example.
- **main.py**: The main executable to run training with different minibatch SGD algorithms.
- **download.py**: The executable script to download all essential data for this homework.
- **visualize.py**: The executable script to render plots for your results. It can give you a better understanding of your implementations.
- **interface.sh**: The executable bash script to give you examples of main.py usage. It also works as an example for writing ReadMe.
- **models.py**: The module defines the learning framework for this homework. You will need to **fill this file** as homework.
- **optimizers.py**: The module defines the customized optimizers for this homework. You will need to **fill this file** as homework.

  The two modules that you are going to develop:

  – **models.py**

  – **optimizers.py**

  The detail will be provided in the task descriptions. All other modules are just there for your convenience. It is not requried to use them, but exploring that will be a good practice of re-using code. Again, you are welcome to architect the package in your own favorite. For instance, adding another module, called `utils.py`, to facilitate your implementation.

Data: MNIST

You are going to conduct a simple classification task, called MNIST (`http://yann.lecun.com/exdb/mnist/`). It classifies images of hand-written digits (0-9). Each example thus is a 28 × 28 image.

- The full dataset contains 60k training examples and 10k testing examples.
- We provide **download.py** that will automatically download the data. Make sure that torchvision library is available.

2.1 Gradient Descent Variants

Gradient Descent does its job decently when the dataset is small. However, that is not the case we see in Deep Learning. Multiple variants of learning algorithms have been proposed to realize training with huge amount of data. We've learned several in the class. Now it's time to test your understanding about them.

Task 1a: Minibatch Stochastic Gradient Descent (SGD) (1 pt)

This is a warm-up task. You are going adapt your HW1 from full-batch to minibatch SGD. All the codes are given.

In Minibatch SGD, the gradients come from minibatches:

$$L(W) = \frac{1}{N_b} \sum_{i=1}^{N_b} L_i(x_i, y_i, W) \tag{1}$$

$$\nabla_W L(W) = \frac{1}{N_b} \sum_{i=1}^{N_b} \nabla_W L_i(x_i, y_i, W), \tag{2}$$

where N_b is the number of examples in one batch. It is similar to the full-batch case, but now you only feed a subset of the training data and update the weights according to the gradients calculated from this subset.

Related Modules:

- main.py
- models.py
- optimizers.py

Action Items:

1. Go through all the related modules. Specifically, you should understand the **main.main(.)** and **main.train(.)** well. Note that the "main.train" method will use minibatch, but does the same thing as full batch in HW1: consider all the input examples and update the weights.

2. You should notice that we move the weight updates to the **optimizers.py**. The **optimizers.SGD** is fully implemented. You should look into it and understand its interactions with the network. Inside **optimizers.py** you will find a basic skeleton for your implementation. You are required to fill in the missing parts of the code, denoted with "# YOU SHOULD FILL IN THIS FUNCTION".

3. Now, move to **main.py**. This file takes care of the main training framework for all experiments.

   (a) There is a `--batch-size` command-line option that specifies that minibatch size. And you should use the "main.train(.)" for training with minibatches.

   (b) Run your code with the command `python main.py --source data/mnist --batch-size 300` (your data folder will be called "data"), which specifies that minibatch size is 300 and in default the maximum number of epochs is 50 (for faster results you can use a smaller epoch).

4. Report your results by filling up Table 1 for **losses, training accuracies, and testing accuracies** (You should have 3 tables, and training is not validation). Use the batch sizes and learning rates given and report the results in the final epoch (they are reported in the log at the end of execution in the "*.stdout.txt" in "sbatch" folder). You can choose a fixed number of epoch for all the results (e.g., 20 epochs). Describe your observations in the report. A short example to run each of the experiment is `python main.py --sbatch account:scholar --batch-size minibatch --lr lr` where minibatch and lr are the minibatch size and learning rate, respectively. Complete examples are also provided at `interface.sh`. (**PDF report on Gradescope**)

5. Running `python visualize.py --minibatch` can output two plots: "Loss vs. Epochs" (add `--ce` argument) and "Accuracies vs. Epochs," if you collect the results correctly in the code. Also, "main.py" will save running logs "ptlog/*.stdout.txt" to "sbatch" folder. Make use of those plots and logs for debugging and analyzing the models. You do not need to submit them in the report.

6. Running the program without specifying the batch size (remove the `--batch-size 300` option or replace by `--batch-size -1` option) gives you the regular Gradient Descent (GD). Compare the results of using regular GD and minibatch SGD, and state your observations in the **PDF report on Gradescope**.

LearningRate / BatchSize	1e-3	1e-4	1e-5
100			
500			
3000			
5000			

Table 1: The results should be reported.

Task 1b: Adaptive Learning Rate Algorithms (2 pts)

You should have learned the adaptive learning rate algorithms in the lecture. In addition to SGD, here we are going to implement several more: **Momentum, Nesterov Momentum, and Adam**.

Since each algorithm might have minor versions, here we define the exact one we want you to implement:

- **SGD**: (You already have it. Just for reference). For each parameter x_t at the iteration t, you update it with:

$$x_{t+1} = x_t - \alpha \nabla f(x_t), \tag{3}$$

  where α is the learning rate.

- **Momentum**: for each parameter x_t at the iteration t, and the corresponding velocity v_t, we have

$$v_{t+1} = \rho v_t + \alpha \nabla f(x_t) \tag{4}$$
$$x_{t+1} = x_t - v_{t+1}, \tag{5}$$

  where α is the learning rate, ρ is the hyperparameter for the momentum rate. A common default option for ρ is 0.9.

- **Nesterov Momentum**: for each parameter x_t at the iteration t, and the corresponding velocity v_t, we have

$$v_{t+1} = \rho v_t + \alpha \nabla f(x_t - \rho v_t) \tag{6}$$
$$x_{t+1} = x_t - v_{t+1}, \tag{7}$$

  where α is the learning rate, ρ is the hyperparameter for the momentum rate. A common default option for ρ is 0.9.

- **Adam**: for each parameter x_t at the iteration t, we have

$$m_1 = \beta_1 * m_1 + (1 - \beta_1) \nabla f(x_t) \tag{8}$$
$$m_2 = \beta_2 * m_2 + (1 - \beta_2) (\nabla f(x_t))^2 \tag{9}$$
$$u_1 = \frac{m_1}{1 - \beta_1^t} \tag{10}$$
$$u_2 = \frac{m_2}{1 - \beta_2^t} \tag{11}$$
$$x_{t+1} = x_t - \alpha \frac{u_1}{(\sqrt{u_2} + \epsilon)}, \tag{12}$$

  where α is the learning rate, m_1 and m_2 are the first and second moments, u_1 and u_2 are the first and second moments' bias correction, and β_1, β_2, and ϵ are hypterparameters. A set of common choices of the parameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$. We initialize $m_1 = m_2 = 0$.

Related Modules:

- optimizers.py

Action Items (points uniformly distributed among SGD, Momentum, Nesterov and Adam implementations):

1. The corresponding class prototypes for the later three algorithms are given in the skeleton. You should finish each of them. Pay attention that Nesterov additionally requires implementation for "prev(.)" method which is different from other optimizers. (**code, not gradescope**)
2. There is a `--optim-alg` command-line option in main.py. So you can test your optimizers individually with the following commands:
   - `python main.py --optim-alg sgd --batch-size 100`
   - `python main.py --optim-alg momentum --batch-size 100`
   - `python main.py --optim-alg nesterov --batch-size 100`
   - `python main.py --optim-alg adam --batch-size 100`

   Full examples are also provided in `interface.sh`.
3. Run `python visualize.py --optimizer`. Plot "Loss vs. Epochs" (Add `--ce` argument) and "Accuracies vs. Epochs" for each algorithm (include SGD) and attach them to the **PDF report on Gradescope**. You should have 2 plots in total containing all four optimizers.
4. Describe your observations from the plots in the **PDF report on Gradescope**. Make algorithm comparisons.

2.2 Regularization

You will be implementing two common regularization methods in neural networks and analyze their difference.

Task 2a: L2-Regularization (1 pt)

The first method is L2-Regularization. It is a general method that not only works for neural networks but also for many other parameterized learning models. It has a hyperparameter for determining the weight of the L2 term, i.e., it is the coefficient of the L2 term. Let's call it λ. In this task, you should implement the L2-regularization and conduct hyperparameter tuning for λ. Think about what should be added to the gradients when you have a L2-Regularization term in the objective function.

Here we give the mathematical definition of the L2-regularized objective function that you should look at:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W) \tag{13}$$

$$R(W) = \sum_k \sum_j W_{k,j}^2, \tag{14}$$

where λ is a hyperparameter to determine the weight of the regularization part. You should think about how to change the gradients according to the added regularization term.

Related Modules:

- optimizers.py

Action Items:

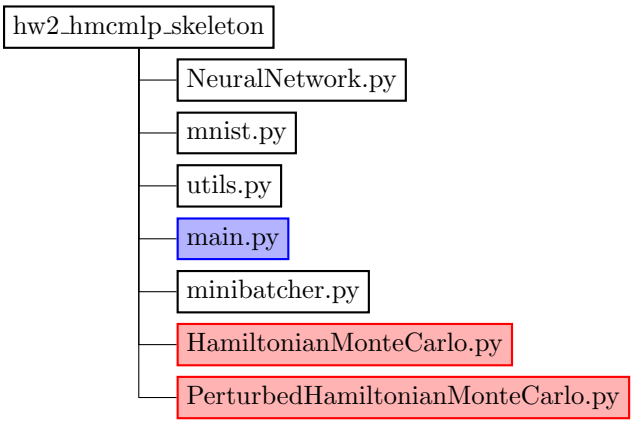
1. There is a command-line option `--l2-lambda`, which specifies the coefficient of the L2 term. In defaut, it is zero, which means no regularization.
2. Add L2-Regularization to all of your optimizers with respect to the command-line option. You need to make changes for all the methods to support L2-regularization, especially for the "SGD.step(.)".
3. Test your network with the following commands:
   - `python main.py --l2-lambda 1 --batch-size 100`
   - `python main.py --l2-lambda 0.1 --batch-size 100`
   - `python main.py --l2-lambda 0.01 --batch-size 100`

   Complete examples are also provided at `interface.sh`.
4. Run `python visualize.py --regularization`. Plot "Loss vs. Epochs" (Add `--ce` argument) and "Accuracies vs. Epochs" for each lambda value, and include them in the **PDF report on Gradescope**. You should have 2 plots in total containing all three settings.
5. Describe your observations from the plots in the PDF report on Gradescope. Make comparisons.

3 Q3: Hamiltonian Monte Carlo for Multi Layer Perceptron in Image classification

In this part, you are going to implement HMC Sampler for learning Multi Layer Perceptron (MLP) weights.

Skeleton Package: A skeleton package is available at
`https://www.dropbox.com/scl/fo/gkpw3czp2t6ifn2h5xuz9/h?dl=0&rlkey=im43kcufi8v2xgext2a6cxzod`
with the execution scripts. You should be able to download it and use the folder structure provided. The zip file should have the following folder structure:



- **hw2_hmcmlp_skeleton** the top-level folder that contains all the files required in this homework.
- **ReadMe:** Your ReadMe should begin with a couple of **execution commands**, e.g., "python hw2.py data", used to generate the outputs you report. TA would replicate your results with the commands provided here. More detailed options, usages and designs of your program can be followed. You can also list any concerns that you think TA should know while running your program. Note that put the information that you think it's more important at the top. Moreover, the file should be written in pure text format that can be displayed with Linux "less" command.
- **utils.py:** Utility functionalities used in main execution files.
- **main.py:** The main executable to run HMC sampling for MLP parameters. You are asked to not to change the code. Even if you change any part of the code, while doing the evaluation the TA will replace it with the original main.py file.
- **minibatcher.py:** Python Module to implement batchifying in MNIST dataset
- **mnist.py:** A mnist data structure to load the MNIST dataset.
- **NeuralNetwork.py:** Multi Layer Perceptron model implemented with PyTorch defined for this homework. You should not change the code.
- **HamiltonianMonteCarlo.py** You will need to implement the necessary functions here for developing a Hamiltonian Monte Carlo Sampler Module
- **PerturbedHamiltonianMonteCarlo.py** You will need to implement the necessary functions here for developing a Hamiltonian Monte Carlo Sampler Module where only the last upper layer will be sampled

HMC for MLP parameters (4.0 pts)

Consider both a 1-hidden layer ("Shallow") and a 2-hidden layered ("Deep") Multi Layer perceptrons (MLP). You are going to implement a posterior sampler using Hamiltonian Monte Carlo (HMC) algorithm presented in the class to classify MNIST images. For simplicity purpose, the problem of image classification has been converted into a binary one, instead of multi-class: every image has been labeled either 0 for even digits (0, 2, 4, 6, 8) or 1 for odd digits (1, 3, 5, 7)

HMC recap: In general, given a model (say, our MLP) with parameters $\boldsymbol{W}$ and a training dataset D. A Bayesian sampler of this model obtains m samples $\boldsymbol{W}_t \sim P(\boldsymbol{W}|D)$, where $t \in \{0, \ldots, m-1\}$ is the sample index. To achieve this via HMC, we need two measurements, the *potential* energy $U(\boldsymbol{W})$ and the *kinetic* energy $K(\boldsymbol{\Phi})$, where $\boldsymbol{\Phi} \sim \mathcal{N}(0, \boldsymbol{R})$ is the auxiliary momentum in HMC algorithm randomly sampled from zero-mean Gaussian distribution with covariance matrix $\boldsymbol{R}$. The choice of $\boldsymbol{R}$ is left to you.

Given an arbitrary dataset $\mathcal{D}$, we have

$$U(\boldsymbol{W}) = -\log P(\mathcal{D}|\boldsymbol{W}) + Z_U,$$

and

$$K(\boldsymbol{\Phi}) = 0.5 \cdot \boldsymbol{\Phi}^\mathsf{T} \boldsymbol{R}^{-1} \boldsymbol{\Phi} + Z_K,$$

where $-\log P(\mathcal{D}|\boldsymbol{W})$ is negative log-likelihood (mean) of model parameter on dataset $\mathcal{D}$ and Z_U, Z_K are arbitrary constants. Thus, we can regard the total energy as

$$H(\boldsymbol{W}, \boldsymbol{\Phi}) = -\log P(\mathcal{D}|\boldsymbol{W}) + 0.5 \cdot \boldsymbol{\Phi}^\mathsf{T} \boldsymbol{R}^{-1} \boldsymbol{\Phi}.$$

The HMC algorithm can be described as Algorithm 1:

Algorithm 1 Single Step Sampling of Hamilton Mento Carlo

Require: Previous sample $\boldsymbol{W}_t$, Size of Leapfrog Step δ, Number of Leapfrog Steps L, Covariance $\boldsymbol{R}$
Ensure: New sample $\boldsymbol{W}_{t+1}$
  $\boldsymbol{\Phi}_0 \sim \mathcal{N}(0, \boldsymbol{R})$
  $\boldsymbol{X}_0 = \boldsymbol{W}_t$
  for $l = 0, \cdots, L-1$ **do**
    $\boldsymbol{\Phi}_{\left(l+\frac{1}{2}\right)\delta} = \boldsymbol{\Phi}_{l\delta} - \frac{\delta}{2} \left.\frac{\partial U(\boldsymbol{W})}{\partial \boldsymbol{W}}\right|_{\boldsymbol{W}=\boldsymbol{X}_{l\delta}}$
    $\boldsymbol{X}_{(l+1)\delta} = \boldsymbol{X}_{l\delta} + \delta \boldsymbol{R}^{-1} \boldsymbol{\Phi}_{\left(l+\frac{1}{2}\right)\delta}$
    $\boldsymbol{\Phi}_{(l+1)\delta} = \boldsymbol{\Phi}_{\left(l+\frac{1}{2}\right)\delta} - \frac{\delta}{2} \left.\frac{\partial U(\boldsymbol{W})}{\partial \boldsymbol{W}}\right|_{\boldsymbol{W}=\boldsymbol{X}_{(l+1)\delta}}$
  end for
  $\alpha = \min\left(1, \exp(-H(\boldsymbol{X}_{L\delta}, \boldsymbol{\Phi}_{L\delta}) + H(\boldsymbol{X}_0, \boldsymbol{\Phi}_0))\right)$
  if $\text{Uniform}(0,1) \leq \alpha$ **then**
    $\boldsymbol{W}_{t+1} = \boldsymbol{X}_{L\delta}$
  else
    $\boldsymbol{W}_{t+1} = \boldsymbol{W}_t$
  end if

Action Items: Let $\mathbf{W}$ are the weights of the Multi Layer Perceprton, and $\mathcal{D} = \{\mathbf{x_i}, \mathbf{y_i}\}_{i=1}^n$ are the training data ($\mathbf{x_i}$ being the MNIST image and $\mathbf{y_i}$ is the image label). After sampling K samples of MLP weights $\mathbf{W^{(1)}}, \mathbf{W^{(2)}}, \ldots, \mathbf{W^{(k)}}$, the Bayesian average model for classification will be,
$p(\mathbf{y}|\mathbf{x}; \{\mathbf{x_i}, \mathbf{y_i}\}_{i=1}^n) = \frac{1}{K}\sum_{k=1}^K p(\mathbf{y}|\mathbf{x}; \mathbf{W}^{(k)})$ where $\boldsymbol{W}^{(k)} \sim P(\boldsymbol{W}|\mathcal{D})$.

We will implement $\boldsymbol{W}^{(k)} \sim P(\boldsymbol{W}|\mathcal{D})$ by HMC in `HamiltonianMonteCarlo.py` and `PerturbedHamiltonianMonteCarlo.py` according to Algorithm 1. Go through all related modules. Specifically, you should understand `main.py`, `utils.py`, `NeuralNetworks.py`. Run the `main.py` with default arguments: `python main.py` to run the programs.

1. (1.5 pts) Fill in the missing parts of `get_sampled_velocities()`, `leapfrog()`, `accept_or_reject()` functions in `HamiltonianMonteCarlo.py`. Go through the comments in the starter code for each function to understand what they are expected to do.

   In short, `get_sampled_velocities()` sample the initial values of velocities $\boldsymbol{\Phi}_0$; `leapfrog()` implements the update of $\boldsymbol{\Phi}, \boldsymbol{X}$ through leapfrog steps; `accept_or_reject()` implements the acceptance or rejection procedeure in the algorithm based on the kinetic and potential energies; and `sample()` combine all these three functions in a way to generate K samples of MLP weight parameters by generating initial velocities, calling leapfrog function multiple times to generate new velocities, decide whether to accept or reject new sample and then prepare the samples.

2. (1 pts) Fill in the missing parts of `get_sampled_velocities()`, `leapfrog()`, `accept_or_reject()` functions in `PerturbedHamiltonianMonteCarlo.py` in such a way that it only updates the last layers weights and biases through sampling. In previous implementation all the layers had their weights and biases updated.

3. (Introduction to model calibration) In measuring model performance, we do not only care about accuracy or loss, but also care about if the model is "calibrated", which means we want it to output the ground-truth probability.

   Consider our MLP model $f_{\boldsymbol{W}}$, parametrized by weight parameters $\boldsymbol{W}$. $\mathcal{D} = \{\mathbf{x_i}, \mathbf{y_i}\}_{i=1}^n$ are the training data ($\mathbf{x_i}$ being the MNIST image and $\mathbf{y_i}$ is the image label). For a given input $\boldsymbol{x}_i$, the prediction $\hat{y}_i$ can be denoted as,

   $$\hat{y}_i := \underset{k\in 0,1}{\text{argmax}}[f_{\boldsymbol{W}}(\boldsymbol{x}_i)]_k,$$

   and the prediction probability (confidence) $\hat{p}_i$ can be denoted as,

   $$\hat{p}_i := \max_{k\in 0,1}[f_{\boldsymbol{W}}(\boldsymbol{x}_i)]_k.$$

   The perfect calibrated model is defined as $P(\hat{Y} = Y|\hat{p} = \alpha) = \alpha$. One notion of miscalibration is the difference in expecctation between confidence and accuracy, i..e, $\mathbb{E}_{\hat{p}}(|P(\hat{Y} = Y|\hat{p} = \alpha) - \alpha|)$. To approximate it by finite samples, we divide the probability/confidence interval $[0, 1]$ into M equal sized bins $B_1, B_2, \ldots B_M$. For each of the example $\boldsymbol{x}_i$ we group them into these bins according to their $\hat{p}_i$ value, i.e., $B_j = \{i : \hat{p}_i \in [\frac{j-1}{M}, \frac{j}{M})\}$. Then, for each bucket B_j, we find out

   $$\rho_j := \frac{1}{|B_j|}\sum_{i\in B_j}\hat{p}_i,$$

   and

   $$\phi_j := \frac{1}{|B_j|}\sum_{i\in B_j}\mathbf{1}[\hat{y}_i \text{ is the true label}].$$

We plot these (ρ_j, ϕ_j) in the reliability diagram where X-axis is for ρ and Y-axis is for ϕ. ρ_j, ϕ_j are respectively called the average confidence and average accuracy for bucket B_j. We also define Expected Calibration Error (ECE),

$$ECE := \frac{1}{n} \sum_{j=1}^{M} |B_j||\rho_j - \phi_j|.$$

4. (1.5 pts) To perform this task, we need to understand how `main.py` works. This script will at first learn the Multi Layer perceptron network using traditional MLE based learning. The default configuration for the MLP is shallow (only 1 hidden layer). The accuracy and losses are plotted in the process. Then after being pre-trained for a fixed number of epochs, more networks will be sampled through HMC sampling and the averaged output from the ensemble will be used in prediction. After that, perturbed version of HMC sampling is done where only the last layers' weights and biases are sampled. These sampled networks are averaged again for another set of predictions.

   Go through the code to understand how it works. After all the predictions using the MLE, HMC sampling and perturbed HMC sampling are done, ROC curves and reliability curves for all the models on their training and test data are plotted for analysis. Your tasks are:

   (a) (0.5pt) Run the `main.py` for both shallow and deep networks using the **- - depth** argument while running the code. If you run **python main.py –depth shallow**, the whole procedure will be run for shallow network, generating loss, accuracy, ROC and reliability plots. If you run **python main.py - - depth deep**, the whole procedure will be run for deep network. You need to understand other command line arguments to tune the necessary hyperparameters (learning rate, leapfrog step numbers, leapfrog step size, delta etc.). You can set different values for these hyperparameters using the command line arguments. Also, to reduce the run time, using the **- - loaded** argument, you can decide whether you will learn a neural network and save it or you will load from a already saved network for the procedures.

   In the report, include all the plots that will be generated. Also, mention the training and test accuracies for MLE, sampled and perturb-sampled models. Also, for each of these three models, write their Expected Calibration Error (ECE) and Expected calibration error at 50% threshold.

   (b) (0.5 pt) Explain the findings you got from the generated ROC curves and AUC scores. Does introducing Bayessian sampling improve the performances? If we only sample the last layers, but with more steps, how does the performance change?

   (c) (0.5 pt) Explain the findings you got from the generated Reliability curves and ECE scores. Does introducing Bayessian sampling improve the performances? If we only sample the last layers, but with more steps, how does the performance change?

Submission Instructions

Please read the instructions carefully. Failed to follow any part might incur some score deductions.

Naming convention: [firstname]_[lastname]_hw2

All your submitting files, including a report, a ReadMe, and codes, should be included in "homework/template" one folder. The folder should be named with the above naming convention. For example, if my first name is "Bruno" and my last name is "Ribeiro", then for Homework 2, my submitting folder name should be "bruno_ribeiro_hw2".

Tar your folder: [firstname]_[lastname]_hw2.tar.gz

Create a folder [firstname]_[lastname]_hw2 and put both the implemented skeleton folders inside it. Remove any unnecessary files in your folder, such as training datasets. Make sure your folder structured as the tree shown in Overview section. Compress your folder with the the command: **tar -czvf bruno_ribeiro_hw2.tar.gz bruno_ribeiro_hw2** .

Submit: **TURNIN INSTRUCTIONS**

Please submit your compressed file on **data.cs.purdue.edu** by turnin command line, e.g. **"turnin -c cs690dl -p hw2 bruno_ribeiro_hw2.tar.gz"**. Please make sure you didn't use any library/source explicitly forbidden to use. If such library/source code is used, you will get 0 pt for the coding part of the assignment. If your code doesn't run on scholar.rcac.purdue.edu, then even if it compiles in another computer, your code will still be considered not-running and the respective part of the assignment will receive 0 pt.