BME646 and ECE60146: Homework 3

Spring 2024
Due Date: 11:59pm, Jan 29, 2024
TA: Akshita Kamsali (akamsali@purdue.edu)

Turn in typed solutions via BrightSpace. Additional instructions can be found at BrightSpace. Late submissions will be accepted with penalty: **-10 points per-late-day, up to 5 days**.

1 Introduction

The goal of this homework is for you to develop a greater appreciation for the step-size optimization logic that is ubiquitous in training deep neural networks. To that end, this homework will first ask you to execute the scripts in the Examples directory of your instructor's CGP class that are based on a vanilla implementation of SGD (Stochastic Gradient Descent). Subsequently, you will be asked to improve the learning performance that can be achieved with those scripts by replacing the basic SGD step-size calculation with your implementation based on what's known as the Adam optimizer.

The Adam optimizer involves two parameters that are commonly denoted by β_1 and β_2. A second goal of this homework is for you to carry out hyperparameter tuning with respect to these two parameters. What that means is that you need to search through a designated range of values for these two parameters in order to find the values that give you the best performance. That begs the question: How to measure the performance of a network for any given values for β_1 and β_2? For now, just use the least value of the loss achieved with N iterations of training.

For further information regarding the concepts described above, please refer to Prof. Kak's slides on Autograd [1].

2 Becoming Familiar with the Primer

1. Download the `tar.gz` archive and install version 1.1.3 of your instructor's ComputationalGraphPrimer. You will be notified via Piazza or Brightspace if there are any version updates. You may not want to `sudo pip install` the Primer since that would not give you the Examples directory of the distribution that you are going to need for

the homework. The main documentation page for the Primer can be accessed though the following link:

https://engineering.purdue.edu/kak/distCGP/

2. Execute the following scripts in the Examples directory:

   ```
   python3  one_neuron_classifier.py
   ```

   ```
   python3  multi_neuron_classifier.py
   ```

   The final output of both these scripts is a display of the training loss versus the training iterations.

3. Now, execute the following script in the Examples directory

   ```
   python3  verify_with_torchnn.py
   ```

   If you did not make changes to the script in the Examples directory, the loss vs. iterations graph that you will see is for a network that is a `torch.nn` version of the handcrafted network you get through the script `multi_neuron_classifier.py`

   Compare visually the output you get with the above call with what you saw for the second script in Step 2.

4. Now make appropriate changes to the file `verify_with_torchnn.py` in order to see the `torch.nn` based output for the one-neuron model. The changes you need to make are mentioned in the documentation part of the file `verify_with_torchnn.py`.

   Again compare visually the loss-vs-iterations for the one-neuron case with the handcrafted network vis-a-vis the `torch.nn` based network.

5. Now comes the somewhat challenging part of this homework:

   If you'd look at the code for the one-neuron and multi-neuron models in the Primer, you will notice that the step-size calculations do no use any optimizations. [For the one-neuron case, you can also see the backprop and update code on Slide 59 and, for the multi-neuron case, on Slide 80 of the Week 3 slides.] The implemented parameter update steps are based solely on the current value of the gradient of the loss with respect to the parameter in question. That is,

   $$p_{t+1} = p_t - \text{lr} * g_{t+1} \tag{1}$$

   where p_t denotes learnable parameters from the previous time step, *e.g.* layer weights at iteration t, and g_{t+1} denotes the corresponding gradient for the current time step $t + 1$.

It is your job to improve the estimation of p_{t+1} using the ideas discussed on Slides 105 through 117 of the Week 3 slides. In order to fully appreciate what that means, it is recommended that you carefully review the material on those slides[1].

As you will see in the slides mentioned above, the two major components of step-size optimization are: (1) using momentum; and (2) adapting the step sizes to the gradient values of the different parameters. (The latter is also referred to as dealing with sparse gradients.) Adam (Adaptive Moment Estimation) currently incorporates both of these components and stands as the world's most popular step-size optimizer. However, in some cases, practitioners choose SGD+ over Adam. Feel free to consult your TA to understand the reasons behind this choice. Also, feel free to inititate a conversation on Piazza over the same topic.

What follows is a brief description of the two choices for the optimizer in order to help you do your homework.

- **SGD with Momentum (SGD+)**: In its simplest form, incorporating momentum involves retaining the step size from the previous iteration. The current step-size decision is then based on the current gradient value and the preceding step size. To invoke momentum for step optimization, separate step updates are computed for individual learnable parameters. This approach facilitates determining the current step size by considering both its prior value and the current gradient value. The recursive update formula for the step size (v) is expressed as follows:

$$\begin{aligned} v_{t+1} &= \mu * v_t + g_{t+1}, \\ p_{t+1} &= p_t - \mathrm{lr} * v_{t+1}. \end{aligned} \tag{2}$$

  In the formulas shown, v is the step size and the first equation is the recursive update formula for its update. v_0 is typically initialized with all zeros.

  When determining the step size for the current iteration $t + 1$, only a fraction μ of its value from the previous iteration is utilized. The momentum scalar $\mu \in [0, 1]$ determines the weight assigned to the previous time step update. If you set $\mu = 0$, it corresponds to Vanilla Gradient Descent. Since this exercise aims for you to comprehend what goes under the hood, what variable you think μ corresponds to in the torch implementation of SGD.

You can find the torch documentation of SGD in the following link:
https://pytorch.org/docs/stable/generated/torch.optim.SGD.html

- **Adaptive Moment Estimation (Adam)**: Adam is one of the most widely used step-size optimizers for SGD in deep learning owing to its efficiency and robust performance especially on large datasets. The key idea behind Adam is a joint estimation of the momentum term and the gradient adaptation term in the calculation of the step sizes. To this end, it keeps running averages of both the first and second moments of the gradients, and takes both the moments into account for calculating the step size. The equations below demonstrate the key logic:

$$\begin{aligned} m_{t+1} &= \beta_1 * m_t + (1-\beta_1) * g_{t+1}, \\ v_{t+1} &= \beta_2 * v_t + (1-\beta_2) * (g_{t+1})^2, \\ p_{t+1} &= p_t - \text{lr} * \frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1} + \epsilon}}, \end{aligned} \tag{3}$$

  where the definitions of the bias-corrected moments $\hat{m}$ and $\hat{v}$ can be found on Slide 115 of [1]. In practice, β_1 and β_2, which control the decay rates for the moments, are generally set to 0.9 and 0.99, respectively.

- **Hyperparameter Tuning the the Adam Optimizer**:
  Hyperparameter tuning is crucial in deep learning as it involves optimizing the settings that control the learning process, impacting model performance. The right hyperparameter values can significantly enhance a model's accuracy, generalization, and ability to extract meaningful patterns from data. Effective tuning ensures that a model adapts well to diverse datasets and problem domains, ultimately leading to more robust and reliable models. This exercise is aimed to provide insights into the sensitivity of the Adam optimizer to changes in β_1 and β_2 values and enhance your understanding of hyperparameter tuning in deep learning.

3 Programming Task

- Your main programming task is two-fold: implementing SGD+ and Adam based on the basic SGD you see in `one_neuron_classifier.py` and `multi_neuron_classifier.py`.

As explained in Section 2, the Steps 1-4 are for you to become familiar with Version 1.1.3 of the Primer. Prof. Kak's slides on Autograd explain the basic logic of the implementation code for `one_neuron_classifier.py` and `multi_neuron_classifier.py`.

More specifically, your programming task is to create new versions of the one-neuron and multi neuron-classifiers that are based on SGD+ as well as Adam.

- Note that for the implementation of both SGD+ and Adam, **modifying the main module file `ComputationalGraphPrimer.py` is NOT recommended.** Instead, you should create subclasses that inherit the `ComputationalGraphPrimer` class provided by the module. In your subclasses, create or override any class methods as your implementation requires. Also, it should be stressed that you are not allowed to use PyTorch's built-in SGD optimizer.

- Do include your observations on why the results with `torch.nn` are better. Also, talk about the effect of beta values in 3.

- Fig. 1 shows an example of the comparative plots from the one-neuron classifier. This plot is shown just to give you an idea of the improvement achieved from SGD+ over SGD. Your results could vary based on your choice of the parameters, such as learning rate, μ, batch size, number of iterations, etc.

- In this final exercise, you will explore how the performance of the Adam optimizer is affected by two hyperparameters: β_1 (for the exponential decay of the first moment estimates) and β_2 (for the exponential decay of the second moment estimates). Using your own implemention for the Adam optimizer, train your network with 3 different values for β_1 and β_2. For example, you can set β_1 to [0.8, 0.95, 0.99] and β_2 to [0.89, 0.9, 0.95]. Pick a reasonable value for N. You may continue with the same number of iterations as in previous exercises. Now, tabulate the time taken, final and minimum losses in these nine different configurations.

  Based on your observations, state your conclusions about the impact of β_1 and β_2 on the Adam optimizer's performance.

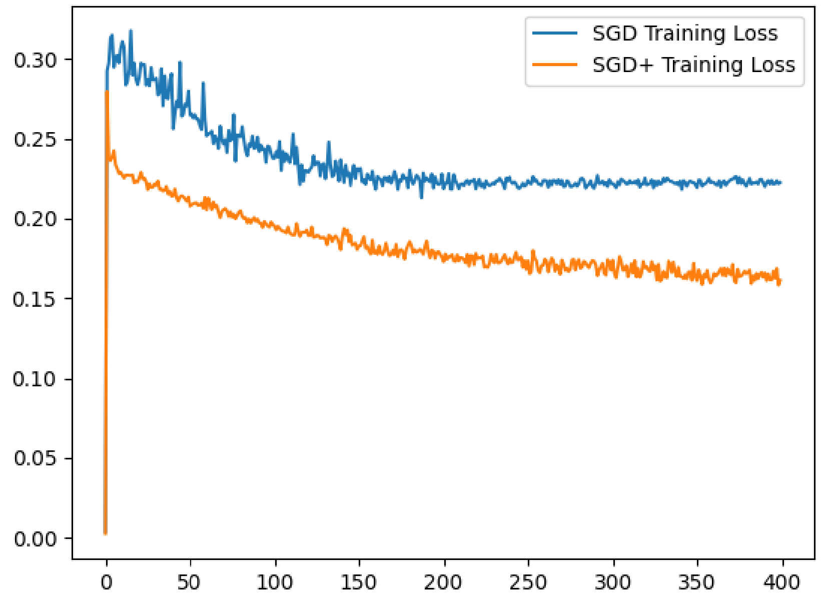

Figure 1: Sample comparative plot (SGD+ vs SGD) for the one-neuron network. Your results could vary depending on your choice of the training parameters. All the plot formatting related options are also flexible.

4 Submission Instructions

Include a typed report explaining how did you solve the given programming tasks.

1. Do NOT include CGP Primer downloaded folder or any datasets. Only submit the .py files you have modified. If you have made any changes to CGP Primer, your code won't run on our test scripts. Please be warned and adhere to instructions in 3 on modifying the main module files.

2. Your pdf must include a description of

    - A description of both SGD+ and Adam in your own words with key equations.

- For the one-neuron classifier, a plot of training loss vs iteration comparing all three optimizers (SGD, SGD+, Adam). Another 2 sets of the same plot but with two different learning rates of your choice covering a good spectrum of low to high learning rates. What are your observations in terms of loss smoothness and convergence?
- The same comparative plots with 3 different learning rates for multi-neuron and state your observations.
- Discuss your findings comparing the performance of the three optimizers in one or two paragraphs.
- Discuss your findings comparing the performance of the Adam optimizer under 9 configurations in one or two paragraphs.
- Your source code. Make sure that your source code files are adequately commented and cleaned up.

3. Turn in a pdf file a typed self-contained pdf report with source code and results. Rename your .pdf file as hw3_<First Name><Last Name>.pdf

4. Turn in a zipped file, it should include all source code files (only .py files are accepted). Rename your .zip file as hw3_<First Name><Last Name>.zip .

5. For all homeworks, you are encouraged to use `.ipynb` for development and the report. If you use `.ipynb`, please convert it to `.py` and submit that as source code.

6. You can resubmit a homework assignment as many times as you want up to the deadline. Each submission will overwrite any previous submission. **If you are submitting late, do it only once on BrightSpace.** Otherwise, we cannot guarantee that your latest submission will be pulled for grading and will not accept related regrade requests.

7. The sample solutions from previous years are for reference only. **Your code and final report must be your own work.**

8. To help better provide feedback to you, make sure to **number your figures and tables**.

References

[1] Autograd for Automatic Differentiation and for Auto-Construction of Computational Graphs. URL https://engineering.purdue.edu/DeepLearn/pdf-kak/AutogradAndCGP.pdf.