BME646 and ECE60146: Homework 3

Spring 2025
Due Date: Monday, Feb 3, 2025, 11:59pm
TA: Akshita Kamsali (akamsali@purdue.edu)

Turn in typed solutions via Gradescope. Post questions to Piazza. Additional instructions can be found at the end. Late submissions will be accepted with penalty: -10 points per-late-day, up to 5 days.

NOTE: Apart from the Extra Credit section, this assignment is exactly the same as was given last year. This is a foundational assignment and it would be difficult to come up with a new variant for each offering of this class. Although you should feel free to look over the solutions submitted during the previous years, the code you submit must be your own.

1 Introduction

In this homework you will develop an understanding of step-size optimization in deep neural network training by implementing two popular optimizers, SGD and the Adam. You will also perform hyperparameter tuning for the Adam optimizer's β_1 and β_2 parameters to identify the values that minimize the loss after N iterations.

For more information on the concepts covered in this homework, please refer to Prof. Kak's slides on Autograd [1].

2 Using ComputationalGraphPrimer

1. Download the `tar.gz` archive and install version 1.1.4 of your instructor's ComputationalGraphPrimer (CGP) module. You will be notified via Piazza if there are any version updates. Do NOT `sudo pip install` the CGP module 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 CGP module can be accessed though the following link:

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

2. Execute the following two scripts in the Examples directory that are based on handcrafted neural networks (as explained on Slides 52 through 83 of your instructor's Week 3 slides) :

   ```
   python one_neuron_classifier.py
   ```

   ```
   python 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 (explained on Slides 83 through 90) in the Examples directory

   ```
   python 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 CGP module, 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,

   $$\mathbf{p}_{t+1} = \mathbf{p}_t - \text{lr} * grad_t \tag{1}$$

   where $\mathbf{p}_t$ denotes the values for the learnable parameters at time t, and $grad_t$ is the gradient of the loss function with respect to the learnable

parameters at t. For a more detailed explanation of the notation, see Eq. (31) on Slide 110 of your instructor's Week 3 slides.

Your homework consists of improving the estimation of $\mathbf{p}_{t+1}$ using the ideas discussed on Slides 107 through 118 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 in stochastic gradient descent (SGD) involves carrying forward a portion of the previous update to influence the current step. Instead of relying solely on the current gradient, the update is computed as a combination of the past step and the present gradient, smoothing out fluctuations and often times accelerating convergence. By maintaining a velocity term for each parameter, momentum helps navigate ravines in the loss landscape more effectively (essentially, help you come out of a local minimum). This reduces oscillations and improving optimization stability.

  Let:

  - w_t be the model parameters at time step t,
  - $g_t = \nabla L(w_t)$ be the gradient of the loss function L at step t,
  - v_t be the velocity (accumulated gradient),
  - η be the learning rate,
  - β be the momentum coefficient (typically between 0.9 and 0.99).

  The updates are computed as:

$$v_{t+1} = \beta v_t + g_t \tag{2}$$
$$w_{t+1} = w_t - \eta v_{t+1} \tag{3}$$

- **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:

$$\begin{aligned} m_{t+1} &= \beta_1 * m_t + (1 - \beta_1) * grad_t, \\ v_{t+1} &= \beta_2 * v_t + (1 - \beta_2) * (grad_t)^2, \\ p_{t+1} &= p_t - \mathrm{lr} * \frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1} + \epsilon}}, \end{aligned} \tag{4}$$

  where the definitions of the bias-corrected moments $\hat{m}$ and $\hat{v}$ can be found on Slides 117-118 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 Tasks (100 pts)

1. **Implementing SGD+ and Adam:** Modify the existing one-neuron and multi-neuron classifiers to incorporate enhancements to the basic SGD implementation. Implement two updated versions of these

classifiers: One using SGD+, which incorporates momentum or other improvements over the basic SGD. Another using the Adam optimizer, known for its adaptive learning rate mechanism.

NOTE: Preserve Original Code Structure
Do not modify the primary module file `ComputationalGraphPrimer.py`. Instead, create subclasses that inherit from `ComputationalGraphPrimer` and implement or override methods as necessary.

2. **Performance Analysis:** Compare the results of SGD, SGD+, and Adam in terms of convergence and final loss values. Provide your observations on why the results with Adam outperform basic SGD in most scenarios.

3. **Effect of Hyperparameters:** Explore how the Adam optimizer's performance is affected by its hyperparameters β_1 and β_2. Train the network with three different values for β_1 (e.g., [0.8, 0.95, 0.99]) and β_2 (e.g., [0.89, 0.9, 0.95]). Tabulate results showing the time taken, final loss, and minimum loss for each configuration.

4. **Visualizing Results:** Generate comparative plots (e.g., loss vs. iterations) to illustrate the improvement achieved by SGD+ and Adam over the basic SGD. There is no wrong answer. Your results may vary depending on training parameters such as learning rate, momentum, batch size, and the number of iterations.

4 For Extra Credit (25 pts)

For extra credit, you'd need to have a good understanding of the topics of "pixel value scaling" and "pixel value normalization" covered in your instructor's Week 2 lecture. The main point of those steps is to convert the $(0, 255)$-range integer data extracted from the images into the floating-point interval $(-1.0, 1.0)$ that is needed by the neural networks.

With that introduction the to "Extra Credit" challenge, go to Slide 54 of your instructor's Week 3 lecture. That slide defines the `DataLoader` class for demo scripts `one_neuron_classifier.py` and `multi_neuron_classifier.py`. Focus on the following two statement in the definition of the `DataLoader`:

```
maxval = 0.0 ## For approx normalization of shifted-mean std
                                    Gaussian
...
...
```

```
batch_data = [item/maxval for item in batch_data] ## Normalize
                                    batch data
```

Your instructor has obviously attempted to "normalize" the training data. The important question here is: Does this data normalization make any sense at all for the two demo scripts mentioned above?

The training data for the scripts is drawn initially from a zero-mean unit-variance Gaussian as shown on Slide 53 of the same slide deck. That means, the floating-point values for training data are in the infinite interval $(-\infty, \infty)$ and that range is not altered by simply shifting the means for the two classes involved.

So it would seem that the data normalization of the sort attempted by the code lines shown above would be of limited utility, if any at all.

Your extra-credit assignment consists of the following two parts:

- Compare the performance of the two demo scripts (as measured by just the training loss vs. iterations plots) for the two cases of with and without the data normalization as currently used. This part of the Extra Credit is worth 10 points.

- Truncate the input data to the interval $(\mu - 5\sigma, \mu + 5\sigma)$, where μ is the mean and σ the standard-deviation of the Gaussian and then remap the data to the $(-1.0, 1.0)$ interval. The truncation to the $(\mu - 5\sigma, \mu + 5\sigma)$ interval is justified by the fact that the nonlinearity of the activation function will make any input values outside of this interval irrelevant. This part of the Extra Credit is worth 15 points.

5 Submission Instructions

Include a typed report explaining how you solved the given programming tasks. You may refer to the homework solutions posted at the class website for the previous years for examples of how to structure your report

1. **Turn in a PDF file and mark all pages on gradescope**.

2. Submit your code files(s) as zip file.

3. **Code and Output Placement:** Include the output directly next to the corresponding code block in your submission. Avoid placing the code and output in separate sections as this can make it difficult to follow.

4. **Output Requirement:** Ensure that all your code produces outputs and that these outputs are included in the submitted PDF. Submissions without outputs may not receive full credit, even if the code appears correct.

5. For this homework, you are encouraged to use `.ipynb` for development and the report. If you use `.ipynb`, please convert code to `.py` and submit that as source code. **Do NOT submit .ipynb notebooks.**

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.** 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.**

References

[1] Autograd Lecture. URL https://engineering.purdue.edu/DeepLearn/pdf-kak/AutogradAndCGP.pdf.