Missouri University of Science & Technology — Department of Computer Science

Spring 2023 — **CS 6406: Machine Learning for Computer Vision (Sec: 101/102)**

Homework 2: Efficient Learning

Instructor: *Sid Nadendla* — **Due:** *Mar 27, 2023*

Goals and Directions:

- The main goal of this assignment is to implement efficient neural networks from scratch, and train them on any given dataset while splitting it into multiple mini-batches.
- Comprehend the impact of hyperparameters and learn to tune them effectively.
- You are **not** allowed to use neural network libraries like PyTorch, Tensorflow and Keras.
- You are also **not** allowed to add, move, or remove any files, or even modify their names.
- You are also **not** allowed to change the signature (list of input attributes) of each function.

Problem 1 Model Aggregation *5 points*

Implement dropout layer in hw2/mlcvlab/nn/basis.py

Linear Function with Inverse Dropout: Accomplish ensemble training via randomly "turning-off" nodes using a binary mask with parameter p, for each mini-batch. In other words, create a mask for each neuron's input by sampling a Bernoulli random variable (instantiated using `numpy.random.binomial`). If the sample is '1', multiply $\frac{1}{p}$ to the corresponding neuron's input. Otherwise, multiply the input with '0'.

Note: The pattern of dropped nodes changes for each input (i.e. each forward pass).

For more details, please refer to Section 10 (Page 1951) in the following paper:

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929-1958, 2014.

Problem 2 Regularization *5 points*

Implement regularizers in hw2/mlcvlab/nn/basis.py.

Batch Normalization: Following are the sequence of steps that need to be followed in a Batch-Norm layer.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

<u>Gradient of Batch Normalization</u>: Following are the sequence of steps needed for computing the gradient of BatchNorm for backpropagation.

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma^2_{\mathcal{B}}} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma^2_{\mathcal{B}} + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$$

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} + \frac{\partial \ell}{\partial \sigma^2_{\mathcal{B}}} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

For more details, please refer to

S. Ioffe, and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *International Conference on Machine Learning*, pp. 448-456, 2015.

Problem 3 Architecture *5 points*

Implement a four-layer NN in hw2/mlcvlab/models/nn4.py

<u>NN4 model</u>: Implement in *nn4* definition with batchnorm and dropout features at each layer.

- Layer 1: $\boldsymbol{z}_1 = \text{Dropout}(W_1\boldsymbol{x})$, $\tilde{\boldsymbol{z}}_1 = \text{ReLU}(\boldsymbol{z}_1)$, $\boldsymbol{y}_1 = \text{BatchNorm}(\tilde{\boldsymbol{z}}_1, \gamma_1, \beta_1)$.
- Layer 2: $\boldsymbol{z}_2 = \text{Dropout}(W_2\boldsymbol{y}_1)$, $\tilde{\boldsymbol{z}}_2 = \text{ReLU}(\boldsymbol{z}_2)$, $\boldsymbol{y}_2 = \text{BatchNorm}(\tilde{\boldsymbol{z}}_2, \gamma_2, \beta_2)$.
- Layer 3: $\boldsymbol{z}_3 = \text{Dropout}(W_3\boldsymbol{y}_2)$, $\tilde{\boldsymbol{z}}_3 = \text{ReLU}(\boldsymbol{z}_3)$, $\boldsymbol{y}_3 = \text{BatchNorm}(\tilde{\boldsymbol{z}}_3, \gamma_3, \beta_3)$.

- Layer 4: $z_4 = \text{Dropout}(w_4^T \boldsymbol{y}_3)$, $y = \text{Sigmoid}(z_4)$

Gradient of NN4 model: Compute the gradients for NN4 model using backprop algorithm and implement them in *nn4_grad* definition.

Problem 4 Data-Parallelism in Optimization[1] *5 points*

Implement SyncSGD in hw2/mlcvlab/optim/sync_sgd.py

Synchronous Mini-Batch SGD: Implement the synchronous mini-batch SGD over multiple GPUs.

- Hyperparameter: δ, K
- Divide training data into K mini-batches.
- Compute the gradient estimate of empirical loss on each mini-batch with respect to $\boldsymbol{W}^{r-1}$ using *emp_loss_grad* function in the model class.
- Wait for all the gradient computations across different mini-batches on different GPUs and aggregate them to obtain a gradient estimate $\nabla \hat{L}_N(\mathbb{W}^{(r-1)})$.
- Compute the update step using the gradient estimate from the previous step:

$$\mathbb{W}^{(r)} = \mathbb{W}^{(r-1)} - \delta \cdot \nabla \hat{L}_N(\mathbb{W}^{(r-1)})$$

Note that the above mini-batch SGD algorithm should leverage the presence of multiple GPUs, for which you will need to use the *just-in-time* (`@cuda.jit`) decorator provided by `numba` package. A necessary dependency for this package to run is the CUDA toolkit, which can be installed by executing the following commands in the Terminal:

```
conda install cudatoolkit
```

For more details about numba, please refer to
https://numba.pydata.org/numba-doc/latest/cuda/index.html

Remark: When using `@cuda.jit`, the input array is first copied from RAM to GPU for processing. The returned values are then copied from GPU to CPU back. However, special care should be taken when the function under `@cuda.jit` attempts to call any other function. In that case, both functions should be optimized with `@cuda.jit`. Otherwise, `@cuda.jit` may slow down the overall computation.

Handout: For your reference, a handout code is provided that demonstrates how you can distribute your computations across GPUs using `numba` package and `cuda` toolkit.

[1]You need to run this on Google's colab, AWS SageMaker Studio Lab, or Foundry for GPU access.

Problem 5 Training and Testing *5 points*

For this question, write your code in hw2/HW2_MNIST_NN4.ipynb.

Train and test your NN4 using `sync_SGD()` on MNIST, similar to that of HW1.