Missouri University of Science & Technology — Department of Computer Science
Spring 2022 — **CS 6406: Machine Learning for Computer Vision (Sec: 101/102)**

Homework 2: Efficient Learning

Instructor: *Sid Nadendla* — **Due:** *April 11, 2022*

Goals and Directions:

- The main goal of this assignment is to implement perceptrons and neural networks from scratch, and train them on any given dataset
- 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 Efficient Neural Networks *25 points*

- **MODEL AGGREGATION (5 points):** Implement dropout layer in hw1/mlcvlab/nn/basis.py

  Linear Function with Dropout: Accomplish ensemble training via randomly "turning-off" nodes using a Bernoulli random variable with parameter p, for each data tuple. In other words, create a mask and turn the output to zero if the Bernoulli random sample (instantiated using `numpy.random.binomial`) is '1'.

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

  Remark: In the testing phase, we need to rescale the

  For more details, please refer to 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.

- **REGULARIZATION (5 points):** Implement regularizers in hw1/mlcvlab/nn/basis.py.

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

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

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

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

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

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

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

$$\begin{aligned}
\frac{\partial \ell}{\partial \widehat{x}_i} &= \frac{\partial \ell}{\partial y_i} \cdot \gamma \\
\frac{\partial \ell}{\partial \sigma^2_{\mathcal{B}}} &= \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma^2_{\mathcal{B}} + \epsilon)^{-3/2} \\
\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} &= \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \\
\frac{\partial \ell}{\partial x_i} &= \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} + \frac{\partial \ell}{\partial \sigma^2_{\mathcal{B}}} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m} \\
\frac{\partial \ell}{\partial \gamma} &= \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i \\
\frac{\partial \ell}{\partial \beta} &= \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}
\end{aligned}$$

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.

- **ARCHITECTURE (5 points):** Implement a three-layer NN in hw1/mlcvlab/models/nn4.py

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

  - Layer 1: $\boldsymbol{z}_1 = W_1\boldsymbol{x}$, $\tilde{\boldsymbol{z}}_1 = \text{ReLU}(\boldsymbol{z}_1)$, $\boldsymbol{y}_1 = \text{BatchNorm}(\tilde{\boldsymbol{z}}_1, \gamma_1, \beta_1)$.
  - Layer 2: $\boldsymbol{z}_2 = W_2\boldsymbol{y}_1$, $\tilde{\boldsymbol{z}}_2 = \text{ReLU}(\boldsymbol{z}_2)$, $\boldsymbol{y}_2 = \text{BatchNorm}(\tilde{\boldsymbol{z}}_2, \gamma_2, \beta_2)$.
  - Layer 3: $\boldsymbol{z}_3 = W_3\boldsymbol{y}_2$, $\tilde{\boldsymbol{z}}_3 = \text{ReLU}(\boldsymbol{z}_3)$, $\boldsymbol{y}_3 = \text{BatchNorm}(\tilde{\boldsymbol{z}}_3, \gamma_3, \beta_3)$.
  - Layer 4: $z_4 = w_4^T\boldsymbol{y}_3$, $y = \text{Sigmoid}(z_4)$

Gradient of NN4 model: Implement in *nn4_grad* definition using backprop algorithm.

- **DATA-PARALLELISM IN OPTIMIZATION**[1] **(5 points):** Here, we will implement two different variants of mini-batch SGD by leveraging computations over multiple GPUs.

  Asynchronous Mini-Batch SGD: Implement SGD in hw1/mlcvlab/optim/async_sgd.py

  - 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.
  - Compute the update step asynchronously: $\mathbb{W}^{(r)} = \mathbb{W}^{(r-1)} - \delta \cdot \nabla L_K(\mathbb{W}^{(r-1)})$

  Synchronous Mini-Batch SGD: Implement SyncSGD in hw1/mlcvlab/optim/sync_sgd.py

  - 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 and aggregate them to obtain a gradient estimate for $\nabla L_N(\mathbb{W}^{(r-1)})$.
  - Compute the update step asynchronously: $\mathbb{W}^{(r)} = \mathbb{W}^{(r-1)} - \delta \cdot \nabla \hat{L}_N(\mathbb{W}^{(r-1)})$

  Note that both the above mini-batch SGD algorithms should leverage the presence of multiple GPUs, for which you will need to use the `@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

  Note: When using `@jit`, the input array is first copied from RAM to GPU for processing. The returned values are then copied from GPU to CPU back. Therefore, for small datasets, the speed can be improved even for small data sets by passing target as "CPU".

  Remark: Special care should be taken when the function under `@jit` attempts to call any other function. In that case, both functions should be optimized with `@jit`. Otherwise, `@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, or AWS SageMaker Studio Lab for GPU access.

- **TRAINING AND TESTING (5 points):** Train and test your NN4 using both `async_SGD()` and `sync_SGD()` on MNIST, similar to that of HW1. Compare the run-time of the two training approaches.