CS 434: Implementation Assignment 3

Due May 14th 11:59PM, 2019

General instructions.

1. You can work in team of up to 3 people. Each team will only need to submit one copy of the source code and report.
2. You need to submit your source code (self contained, well documented and with clear instruction for how to run) and a report via TEACH. In your submission, please clearly indicate all your team members.
3. Your code will be tested on flip. Please make sure that your program can run on flip without any issue.
4. Be sure to answer all the questions in your report. Your report should be typed, submitted in the pdf format. You will be graded based on both your code as well as the report. In particular, the clarity and quality of the report will be worth 10 % of the pts. So please write your report in clear and concise manner. Clearly label your figures, legends, and tables.

1 Multi-layer Perceptron (MLP) for CIFAR10

For this assignment, you will use the **PyTorch** neural network package to perform a set of experiments on the famous CIFAR10 dataset and report your results and findings together with discussions of the results.

The CIFAR10 Dataset. The data can be downloaded from

`https://www.cs.toronto.edu/~kriz/cifar.html`

This webpage contains all the information that is needed for unpacking and using the data.

The Neural Network Package and sample code. You will not be implementing your own network training algorithm. Instead, you will use the neural network package provided by PyTorch. The following page provides some example code for using pytorch for training a MLP on the MNIST data.

Installing Pytorch. You can **install** pytorch from the following page: `https://pytorch.org/`
Because you are required to run your code on *filp*, please use the following specific step to install Pytorch on flip (ignore steps 1,3,4,5 if you already did them for assignment 1):

1. Create a python virtual environment by running the instruction:

   python3.5 -m venv path_to_your_env_folder_in_cs434

2. Access your virtual environment by running the instruction:

   source path_to_your_env_folder_in_cs434/bin/activate.csh

3. pip install –upgrade pip
4. pip install numpy
5. pip install scipy
6. pip3 install https://download.pytorch.org/whl/cpu/torch-1.0.1.post2-cp35-cp35m-linux_x86_64.whl
7. pip3 install torchvision

Sample code. The following page provides some example code for using pytorch for training a MLP on the MNIST data.`https://github.com/CSCfi/machine-learning-scripts/blob/master/notebooks/pytorch-mnist-mlp.ipynb`
Specifically, in this example,

- the MLP has 3-layer (2 hidden layer and 1 output layer), with 50 hidden units in each hiden layer with ReLu as the activation function, and 10 output nodes in the output layer with softmax activation
- Loss function is negative loglikelihood (NLL_loss) (similar to what is used for Logistic regression, but for $C = 10$ classes)
- Training is performed by applying Stochastic Gradient Descent(SGD) to minimize the loss with a learning rate of 0.01
- A drop out rate of 0.2 is applied to the first two layers, which will random zero out 20% of the input to these layers. This is an effective technique for regularization and preventing the co-adaptation of neurons.

For your experiments, you will create a multilayer perceptron neural network and train it on the CIFAR10 dataset to predict object class based on the input image. You can use the example code provided above for MNIST as the basis and modify the necessary part to work with the CIFAR10 data. The main difference is that the CIFAR10 image dimensions are 32 by 32, with 3 values for each pixel (R, G, B) whereas MNIST is 28 by 28 with only a single greyscale value per

pixel. You will also want to normalize all values by 255 so that the value will be between 0 and 1 (you can try the training without any normalization and observe its impact). We will use the same output layer, and the same loss function, and the same optimizer for training.

1. (20 pts) For the first set of experiments, you will create and train a 2-layer network with one hidden layer of 100 hidden nodes using sigmoid activation function. The training data is divided into five batches. You will use the first four batches for training, the fifth batch for validation. The test batch will be reserved for testing. When training the network, you need to monitor the loss (negative log loss) on the training data (similar to what is shown in the Karpathy demo) as well as the error (or accuracy) on the validation data, and plot them as a function of training epoches. Train your network with the default parameters for drop-out, momentum, and weight decay, but experiment with different learning rates (lr) (e.g., $0.1, 0.01, 0.001, 0.0001$). What is a good learning rate that works for this data and this network structure? Present your plots for different choices of learning rates to help justify your final choice of the learning rate. How do you decide when to stop training? Evaluate your final trained network on the testing data (the test batch) and report its accuracy.

   Write your code so that you get the results for questions 1 using the following command:

   python q1.py lr

   The output should include:

   - A plot of the training loss and the validation error (one plot) as a function of training epoch number.
   - Plots (like the one, you did in the previous step) for different choices of learning rates.
   - Test accuracy.

2. (15 pts) Repeat the same experiment as (1) but use Relu as the activation function for the hidden layer and answer the same questions as in (1). Write your code so that you get the results for questions 2 using the following command:

   python q2.py lr

   The output should include:

   - A plot of the training loss and the validation error (one plot) as a function of training epoch number.
   - Plots (like the one, you did in the previous step) for different choices of learning rates.
   - Test accuracy.

3. (25 pts) Experiment with other parameters: drop out (d), momentum (m) and weight decay (wd). The goal is to improve the performance of the network by changing these hyper-parameters. Please describe what you have tried for each of these parameters. How do the choices influence the behavior of learning (as examined by monitoring the loss as a function of the training epochs)? Does it change how fast the training converges? How do they influence the testing performance? Please provide a summary of the results and discuss the impact of these parameters. Note that your discussion/conclusion should be supported by experimental evidences like test accuracy, training loss curve, validation error curves etc.

   python q3.py d m wd

   The output should include:

   - Plots of the training loss and the validation error (one plot) as a function of training epoch number for different d, m , wd.
   - A plot of test accuarcy as a function of different choices of d (fix m and wd to some reasonable value).
   - A plot of test accuarcy as a function of different choices of m (fix d and wd to some reasonable value).
   - A plot of test accuarcy as a function of different choices of wd (fix m and d to some reasonable value).

4. (30 pts) Now we will alter the structure of the network. In particular, we will keep the same number of hidden nodes (100) but split them into two hidden layers (50 each)[1] Train the new network with the same loss function, the SGD optimizer, and your choice of activation function for the hidden layers. What do you observe in terms of training convergence behavior? Do you find one structure to be easier to train than the other? How about the final performance of the network in terms of training error and testing error? Which one gives you better testing performance? Provide a discussion of the results. Please provide necessary plots and figures to support your discussion.

   python q4.py lr

   The output should include:

   - Test error and train error for previous vs current architecture.
   - Plots and figures of your choice to support your findings.

[1] In case this does not produce anything significantly different, feel free to consider even deeper alternatives.