The University of Texas at Austin
Chandra Department of Electrical and Computer Engineering
Cockrell School of Engineering

Spring 2023

ECE 361E: Machine Learning and Data Analytics for Edge AI
HW1 - In depth explanations

On your local computer: You can use either ***conda*** or ***virtualenv*** to install the required packages which you can find in the ***requirements.txt*** file. You should test your code on your local computer before deploying it to the cloud.

The solution code for the entire homework should run **only on the GPUs** of the Maverick2 machine in TACC (GTX nodes, to be more precise). You do not need to change the loss, the optimizer, the batch size, the number of training epochs or the learning rate when training.

IMPORTANT! All plots in all homeworks must have: a title, labels on both axes, a grid and a legend with corresponding labels for each line in the plot.

Use ***matplotlib.pyplot*** to draw figures and save your figures as PNG files. Save all data for the plots in a CSV file so there is no need to re-train in case of an error or system malfunctioning.

Problem 1: Logistic Regression (using TACC Machines)

In the first part of this assignment, you will implement Logistic Regression (LR), a basic linear classifier for MNIST dataset. The starter code of LR is provided to you in the ***starter.py*** file. Read the comments in the file and understand the code as this is important for developing a complete machine learning model.[1]

Question 1

If you put the model on the GPU, you also need to put the images and labels on the GPU, otherwise you will get a CUDA error saying that the images are on CPUwhile the model is on GPU, so it cannot train.

Question 3

The total time for *training* (in seconds [s]) should be measured only for the **Training phase** from the **Training loop**. The *inference* time should be measured **only once**, on the test dataset, ***after*** the model has been fully trained. When measuring the inference time, you only need to measure the actual inference, which is the line of code: ***outputs = model(images)***. In real-world applications, the time measurements are typically repeated multiple times and only average values are reported. In this homework, however, only one run is enough for all time measurements. The average time for inference per image (in milliseconds [ms]) is the total time for inference divided by the number of images used for the inference.

You need to write some extra code required for testing **after the Training loop** (Hint: look at **Testing phase** code). When evaluating the inference time, set ***batch_size=1*** for the ***test_loader***, set ***model=model.eval()*** and use ***with torch.no_grad()***. This way you will not make inference batch by batch, but rather process each image at once, set the model in the evaluation mode and disable gradient calculation for inference. For details on measuring the GPU memory usage check **Appendix A1.3**.

Problem 2: Overfitting, Dropout and Normalization

In this part of the homework, you will experiment with a Fully Connected (FC) network and learn what overfitting is, how to counter it using dropout, and why normalization can be beneficial.

Question 1

Overfitting is the phenomenon where the training loss is decreasing, but the test loss tends to remain the same or even increase. Overfitting basically means the network is “memorizing” the training set instead of being able to generalize well on the new test data.

Question 2

[1] You are also encouraged to check the PyTorch documentation and try a few tutorials on your own to better understand the details of various models.

Dropout is an effective method to prevent overfitting and relies only on one *hyperparameter*. A hyperparameter is a parameter chosen manually by the programmer, which in the case of dropout is the probability of dropping a neuron. During training, the network "drops" some neurons randomly, such that not all neurons learn during a specific epoch. In other words, during every epoch, a different set of neurons are allowed to learn; this way, the model does not memorize the training dataset, hence the model is able to generalize and perform better on unseen data (i.e., testing data in our case).

Question 3

Normalization is a technique that helps the model learn faster because the neural networks have a tendency to "like" small numbers. The terms normalization and standardization are sometimes used interchangeably, but they refer to different things: *normalization* scales data to a value between 0 and 1, while *standardization* transforms data to have a mean of zero and a standard deviation of 1.

PyTorch has a built-in library which can "transform" the entire dataset. If you look carefully at lines **36-37** in ***simpleFC.py,*** then you will notice that there is a transformation for the entire dataset called ***ToTensor()***; this transformation changes each data point (i.e., each image) into a tensor. We typically combine multiple such transformations using the ***Compose()*** method which receives a list with an ordered sequence of such transformations. Use ***Compose()*** to combine the ***ToTensor()*** transformation with a normalization transformation using ***Normalize(mean=0.1307, std=0.3081)***.

Problem 3: CNNs and Model Complexity

In this problem, you will experiment with Convolutional Neural Networks (CNNs) and discover another method to handle overfitting. You will also evaluate the model complexity and design your own model.

Question 1

CNNs are the most popular models in today's deep learning industry, thus all frameworks are developed to make the training job easier for programmers. The images coming from the dataset loaders are directly in the corresponding format, i.e., *(Batch, Channel, Height, Width)* or *(B, C, H, W)* for short. The input images do ***not*** need to be vectorized as they are in **Problem 1** and **Problem 2**. This format is specific to the PyTorch framework and the CNNs are created to take as input 4D tensors like above.

Every training or inference process can be reduced to additions and multiplications. This is why a fundamental metric for model complexity is typically measured using *Floating Point Operations per Second (FLOPs)*. Because of the heavy use of addition and multiplication operations, the *Multiply and Accumulate (MAC)* metric also became very relevant. **One MAC will count as two FLOPs** (i.e., one multiplication and one addition). Both metrics are widely used in the scientific literature and in industry.

You can obtain the MACs for ***SimpleCNN*** using ***profile()*** from the ***thop*** library. Using ***summary()*** from the ***torchsummary*** library, you can obtain the number of parameters for ***SimpleCNN*** and its size in MB. Use ***torch.save(model.state_dict(), 'path/simplecnn.pth')*** to save your model after you train it.

We will use the ***.pth*** extension to save the PyTorch models. For ***profile()*** you need an input tensor. You can generate a random input tensor using ***torch.randn(1, 1, 28, 28)***, where the size follows the *(B, C, H, W)* format. For ***summary(),*** you will only need to provide the input size in the *(C, H, W)* format. For both the model and the input tensor use the GPU as a device.

Question 2

Sometimes overfitting can occur because the model is too complex for the task at hand (i.e., for a simple dataset like MNIST), therefore you can lower the number of training epochs or the model complexity.