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

Spring 2021

ECE 379K: Machine Learning and Data Analytics for Edge AI

HW1 Assigned: Jan 26 DUE: Feb 9 (11:59:59pm CST)

Work in groups of two students. At the end of the PDF file add a paragraph where you clearly describe each member's contribution

Introduction

This assignment is meant to be an introduction to understanding training and testing Machine Learning (ML) models in the Cloud. Specifically, you will be working with PyTorch, one of the most popular frameworks for developing ML models. By working on this assignment, you will be able to:

- Understand the basics of ML techniques (e.g., training and testing a model, handling overfitting, comparing and contrasting optimizers, evaluating the complexity of a model and fine-tuning the hyperparameter of a model);
- Visualize various metrics and parameters to understand the impact of different design decisions.

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 on Maverick2.

Problem 1 [20p]: Logistic Regression using TACC machines

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

Question 1: [5p] If you run ***starter.py*** as is, this will train the LR model on the CPU of your machine (the default device). Modify ***starter.py*** by specifying the target device for your model by using ***model = model.to(torch.device('cuda'))*** to train the model on GPU. Use the same device name for all tensors (i.e., images and labels) and your model, otherwise you may encounter errors. Now run ***starter.py*** on Maverick2 (see **Appendix A1** for environment setup and **Appendix A2** to run your code).

Question 2: [6p] Draw a plot showing the *training loss* and the *test loss* of your model for each epoch. Draw a second plot showing the training and testing accuracies of your model for each epoch.

Question 3: [9p] Complete *Table 1*:

Table 1

Training accuracy [%]	Testing accuracy [%]	Total time for training [s]	Total time for inference [s]	Average time for inference per image [ms]	GPU memory during training [MB]

The total time for training should be measured only for the **Training phase loop**. The inference time is measured only once, on the test dataset, *after* the model has been fully trained. 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 each image at a time, set the model in evaluation mode and disable gradient calculation for inference. For details on measuring the GPU memory usage check **Appendix A3**.

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

NOTE:

a) For all plots in this homework, you need to have a title, labels on axes, a grid and a legend with corresponding labels for each line in the plot. Use ***matplotlib.pyplot*** to draw your figures and save your figures as PNG files.

b) We recommend you to save all data for the plots in a CSV file. This is because, if you encounter errors, you will still have the data for your figures saved, therefore redoing training.

c) When measuring the inference time you only need to measure the actual inference ***outputs = model(images)***. In the real applications, the time measurements need to be repeated several times; then the average values for all runs are reported. In this homework, one run is enough for all time measurements.

d) For the remaining problems in this homework, **all your code should run on GPUs** from Maverick2.

Problem 2 [30p]: 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 using normalization can be beneficial.[2]

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 "memorizing" the training set and not being able to generalize well to the test set.

Question 1: [5p] Run the code from ***simpleFC.py*** as is. Draw a plot showing the training and testing accuracies for each epoch, and another plot showing the training and testing loss values for each epoch. Is this model overfitting? Explain.

Dropout is an effective method to prevent overfitting which relies on one *hyperparameter*[3]. 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 during training (i.e., testing data in our case).

Question 2: [10p] In the ***__init__()*** method from the ***SimpleFC*** class, define once ***nn.Dropout(probability)*** and apply this new operation in the ***forward()*** method after every layer of the model except the last one. Run four experiments using the values [0.0, 0.2, 0.5, 0.8] as probabilities for dropout. For each experiment, draw one plot showing *training* vs *testing loss values* for each epoch. What do you observe from these plots? Which dropout probability gives the best/worst results? Explain.

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 usually refer to different things. More precisely, *normalization* usually refers to scaling a variable to a value between 0 and 1, while *standardization* transforms data to have a mean of zero and a standard deviation of 1.

To normalize the training dataset, 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.

[2] Throughout this problem, you do ***not*** need to change the loss, the optimizer, the batch size, the number of training epochs or the learning rate when training.

[3] A *hyperparameter* is a parameter chosen manually by the programmer, which in the case of dropout is the probability of dropping a neuron.

Question 3: [15p] Complete ***Table 2*** by running the same code from **Question 2**. For the rows with "+ norm" keep the specified dropout probabilities and normalize both the training and testing datasets. Use ***Compose()*** to combine the ***ToTensor()*** transformation with a normalization transformation using ***Normalize(mean=0.1307, std=0.3081)***. Compare and contrast these normalized vs unnormalized experiments and explain the differences.

Table 2

Dropout	Training accuracy [%]	Testing accuracy [%]	Total time for training [s]	First epoch when the model reaches 96% training accuracy
0.0				
0.2				
0.5				
0.0 + norm				
0.2 + norm				
0.5 + norm				

Problem 3 [30p]: CNNs and Model Complexity

In this problem[4], you will experiment with Convolutional Neural Networks (CNNs) and discover another method to handle overfitting. You will also evaluate the model complexity in terms of MACs and FLOPs and decide which activation function can benefit the network training the most.

The input images do not need to be vectorized as they are in *Problem 1* and *Problem 2*. 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. This format is specific to the PyTorch framework and the CNNs are created to take as input 4D tensors like these.

The computational expenses are typically measured using *Floating Point Operations per Second (FLOPs)*. To this end, operations like addition (or multiplication) between two numbers in floating point representation represent a fundamental metric. However, with the development of neural networks, another metric, namely *Multiply and Accumulate (MAC)*, also became very relevant; MACs will count as two FLOPs (i.e., one multiplication and one addition). Both metrics are widely used in scientific literature.

Question 1: [15p] In ***simpleCNN.py*** implement the normalized MNIST dataset as you did in ***Problem 2 Q3***. Run the code and complete ***Table 3***. Is there a difference between the estimated total size of the model from ***torchsummary*** and the saved version of the model? Explain?

Table 3

Model name	GMACs	GFLOPs	# parameters	Model size [MB]	Saved model size [KB]
SimpleCNN					

[4] Throughout this problem, you do ***not*** need to change the loss, the optimizer, the batch size, the number of training epochs or the learning rate when training.

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 trained it.

NOTE:

a) For PyTorch, the most common file extensions are ***.pth*** and ***.pt7***; in this class, we will use exclusively the ***.pth*** extension to save the PyTorch models.

b) 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.

c) Because the numbers you will obtain are pretty big, report the number of GigaMACs and GigaFLOPs (i.e., GMACs and GFLOPs).

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

Question 2: [15p] Modify ***SimpleCNN*** to have 4 output feature maps (down from 32) for the ***conv1*** layer and 8 output feature maps (down from 64) for the ***conv2*** layer. Draw one plot of the training vs testing loss values and a second plot of the training vs testing accuracies for every epoch for ***SimpleCNN*** before and after reducing its complexity. Extend ***Table 3*** from ***Q1*** using the results obtained for the smaller ***SimpleCNN***. What do you observe regarding the loss values and accuracies in the plots? Considering the loss, the accuracy and the extended ***Table 3***, which version of the model is better? Explain.

Problem 4 [20p+15Bp]: Optimizers and Hyperparameters

In this problem you will compare and contrast three different optimizers used for training models. You will also finetune some hyperparameters on the ***SimpleCNN*** model to understand the effects of batch and learning rates on model performance. To this end, we use the ***SimpleCNN*** model with 4 output feature maps for the ***conv1*** layer and 8 output feature maps for the ***conv2*** layer as implemented in ***Problem 3***.

Optimizers have a significant impact on the learning phase of deep learning models. To see this, you will experiment with ***SimpleCNN*** while using three of the most popular optimizers: SGD, RMSprop and ADAM. For RMSprop use ***alpha=0.99*** and ***eps=1e-8***, for SGD, use ***momentum=0.9*** and ***weight_decay=5e-4***, and for Adam, use the ***betas=(0.9, 0.999)*** and ***epsilon=1e-08***.

Question 1: [10p] Keep the normalized dataset from **Problem 3 Q1**. For each of the following learning rates [0.01, 0.001, 0.0001] train ***SimpleCNN*** using all three optimizers. For each experiment, draw a plot showing the training loss over each epoch for RMSprop, SGD and Adam. Based on the results you obtain, discuss the best optimizer. What (learning rate, optimizer) combinations would give the best results?

Question 2: [10p] Using ***learning_rate=0.01*** retrain ***SimpleCNN*** for the following batch sizes [1, 2, 8, 128, 256, 2048]. Draw one plot with the training time for each batch size. On the X-axis (i.e., batch size) use the logarithmic scale. What do you observe? Are there any major benefits regarding the training time if we increase the batch-size more than 128?

Question 3: BONUS [15Bp] Train[5] **SimpleCNN** using the batch sizes [2, 128, 2048] and learning rates [0.3, 0.1, 0.03, 0.01, 0.001, 0.0001]. Draw a single plot for all batch sizes showing the training accuracy vs learning rate. Use ***semilogx()*** to plot the learning rate on a logarithmic scale on the X-axis. Do you notice any differences in the learning performance? What do you observe for each batch size? What are the optimal learning rates for each batch size?

[5] You will need to train 3×6=18 times ***SimpleCNN*** using *all* possible combinations between the given batch sizes and learning rates.

Submission Instructions/Hints

1. Please include your solutions into a single zip file named **<GroupNo_FirstName1_LastName1_FirstName2_LastName2>.zip**. The zip file should have:
   - A single PDF file containing all your results and discussions.
   - A *readme.txt* file listing the purpose of all your items in the zip file.
   - Your code.
2. You should verify your implementation as you proceed.
3. Before you begin this assignment, please read the entire description carefully to make sure you have all the tools needed to solve the problems.
4. **Start early!** This homework may take longer than you expect to complete.

Good luck!

Appendix

A1. Setting up TACC environment

We will use TACC to train the models on powerful GPUs. Each account can have 4 running jobs simultaneously on the Maverick2 system. You will access your TACC account remotely from your computer. This can be done via SSH. Open a terminal (or Windows bash/PowerShell for Windows users) and use the command:

```
ssh <your_TACC_username>@maverick2.tacc.utexas.edu
```

You need to introduce your password and then access the DuoMobile app for a second security code required to access UT TACC instances. While in TACC, **always use the $WORK directory**! To change directory to $WORK use the `cdw`[6] command.

1) Load the Python3 module in the system with the following command:

```
module load python3
```

2) To validate, you can check if you have Python3 by typing:

```
which python3
```

3) We use *virtualenv*, which is installed by default. You can check it using:

```
which virtualenv
```

4) To avoid package conflicts with other python packages and versions on the machine, we create a virtual environment using *virtualenv*:

```
virtualenv -p '<output_of_which_python3>' $WORK/HW1_virtualenv
```

5) We activate the virtual environment:

```
source $WORK/HW1_virtualenv/bin/activate
```

6) Check your current working directory (you will need it next) using `pwd`; it should give you something like:

```
/work/<number>/<your_TACC_username>/maverick2
```

7) Download ***HW1_files.zip*** from Canvas and unzip it.
8) Open a new terminal on your computer and go to the location where you downloaded HW1_files:

```
cd Downloads/
```

9) Move the files from your computer to Maverick2 using `scp`[7]

```
scp -r HW1_files <your_TACC_username>@maverick2.tacc.utexas.edu:<pwd_output>
```

10) After moving the files from your computer to Maverick2, on the terminal for TACC install the required packages from ***requirements.txt:***

```
pip install -r $WORK/HW1_files/requirements.txt
```

A2. Running jobs on TACC

1) ***After you write and test the code for a problem on your computer***, copy the Python file from your computer to Maverick2. Open a new terminal window on your computer and execute:

```
scp <your_file_location>
<your_TACC_username>@maverick2.tacc.utexas.edu:<pwd_output>/HW1_files
```

[6] This should be the first command when you login on TACC using SSH.

[7] `scp -r` is used to move recursively all subdirectories and files.

2) After the code has run and plots, CSV files and other result files have been created, you will move them from Maverick2 to your computer. Open a new terminal window on your computer and execute:

```
scp -r <your_username>@maverick2.tacc.utexas.edu:<pwd_output>/HW1_files
                                    <your_destination>
```

3) You will then edit the code in the job file ***config.slurm***

[Required] Make sure to change <username>@utexas.edu to your UT email/ email associated with your TACC account

[Optional] Change the desired name of the job

Click here to read more on Maverick2.

4) From the ***config.slurm*** you can see that the output can be accessed in: `$WORK/HW1_files/output`. Everything you need is already provided there, but feel free to personalize the script according to your needs if needed.

5) Start the job by executing:

```
sbatch config.slurm
```

WARNING: Be sure to *test your code on your local machine* and only when you see it working with no errors and that it starts to train the model, move it to TACC machines. For a good testing experience use 1 epoch for training and the ***'cpu'*** as the device to run your code such that it will not take too much time and it will verify the full functionality of your code (i.e., CSV files generation, plot creation, etc.).

A3. Interacting with TACC instances and managing jobs

1) After you start the job, check the running jobs using `squeue`. You will see:
   - JOBID - the unique ID associated to your submitted job
   - PARTITION - the type of machine running your job (for you should be gtx)
   - NAME - the name you gave your job in the ***config.slurm*** file
   - USER - <your_TACC_username>
   - TIME - time spent running this job
   - NODELIST - the address of the machine running your job

NOTE: If your job is not listed, then it might have been killed due to an error. You will get an email confirming your job has stopped mentioning if it was Completed, Canceled, or Failed due to an error.

2) With the information obtained from `squeue` you can access the machine your job will be running on. Use the NODELIST from above and SSH into that machine:

```
ssh <NODELIST>
```

3) Once you are in that machine you can view the CPU statistics by running htop or you can monitor the GPU continuously, updating the readings at every 0.1 seconds using:

```
watch -n 0.1 nvidia-smi
```

4) To cancel a job simply run:

```
scancel --KILL <JOBID>
```