CS 5600/6600: F23: Intelligent Systems
Assignment 5

Vladimir Kulyukin
Department of Computer Science
Utah State University

September 30, 2023

Learning Objectives

1. Training and Testing ConvNets with PyTorch
2. Persiting ConvNets
3. Plotting ConvNets Performance
4. Image Classification

Introduction

This assignment does not have any reading and writing component. I want you to focus on getting all the PyTorch plumbing installed and working on your machine for the upcoming Project 1. Many third-party plug-and-play tools are seldom plug-and-play. More often then not they are plug-plug-plug-plug and then, at long last, play. Help each other with installation as much as you can. Be generous with your successful experiences and shortcuts.

In the previous assignment, we learned how to build regular neural networks with PyTorch on synthetic datasets. In this assignment, we'll build up on that experience and train and test a ConvNet on an image dataset and anaylyze its performance with plots and classification reports.

We will use an image dataset of handwritten Hiragana characters, also known as the Kuzushiji-MNIST (KMNIST) dataset. In my opinion, this dataset is a little more interesting and challenging than the standard MNIST. KMNIST consists of 70K (60K training and 10K testing) images and their corresponding labels. There are a total of 10 classes corresponding of 10 Hiragana characters. We will train a well-known ConvNet model to classify them. Anohter reason I chose KMNIST is that it is built into PyTroch. But, the steps we'll take to build a ConvNet can generalize to any image dataset so long as each image is has a correct label, i.e., the dataset is curated.

Installation

Let's install a few libraries. If we have them, pip will let us know that what's installed.

```
pip install torch torchvision
pip install opencv-contrib-python
pip install scikit-learn
pip install opencv-contrib-python
pip install matplotlib
pip install imutils
```

Python Scripts and Directories

We'll work with the following Python scripts provided in the zip: `lenet.py`, `train_lenet.py`, and `predict_lenet.py`. The zip directory structure includes two subdirectories – `data` and `output`. The directory `data` cotnains KMNIST. The directory `output` is where we'll persist the trained models and accuracy plots.

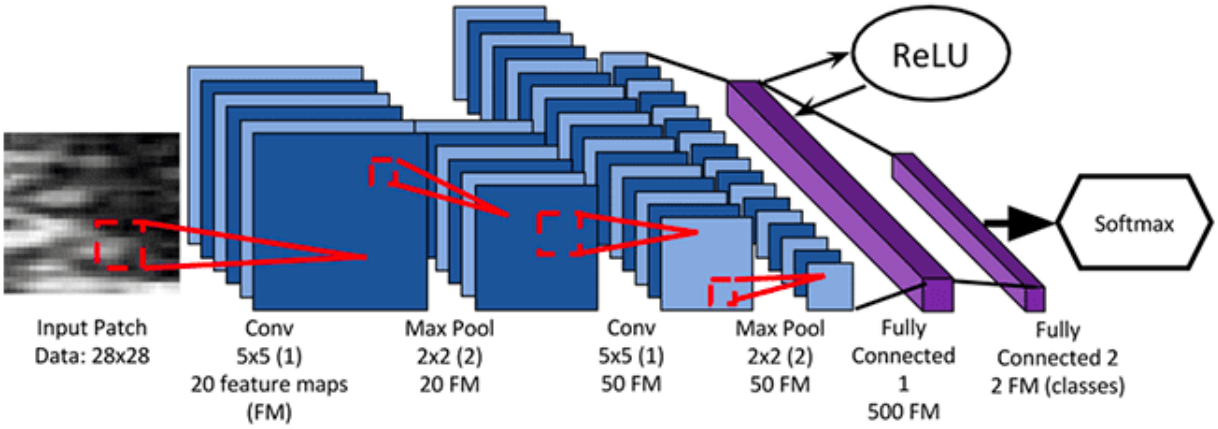

Figure 1: LeNet Architecture.

The file `lenet.py` is a PyTorch implementation of LeNet, a legendary ConvNet originally developed for optical character recognition by Dr. Yann LeCun (hence, the name LeNet) in the mid 1990's. Figure 1 shows the LeNet architecture. You can read more about this architecture and its history on the LeNet Wiki page. Today LeNet would be considered by many researchers and practitioners as a shallow ConvNet.

The file `train_lenet.py` uses PyTorch to train LeNet on KMNIST and perist the trained model in the directory `output`. The file `test_lenet.py` loads and runs persisted LeNet models.

You can read comments in `lenet.py` to understand how it is implemented. In impelementing LeNet, we subclass `Module`, which is a common technique that allows us to re-use variables, implement custom functions, etc. In the `__init__`, `numChannels` is 1 for grayscale images and 2 for RGB and the `classes` is the number of class labels in the dataset, e.g., 10.

Take a few minutes to read the comments in `train_lenet.py` to familiarize yourself with the structure of the code. The main training loop after all the initializations are done and the subsequent evaluation should already be familiar to you from the previous assignment. At the training and evaluation are done, we generate a plot and save it in the `output` directory. Here's how we can run `train_net.py`. The abbreviation `.pth` is used to denote that a file is a persisted PyTorch model.

```
python train_lenet.py --model output/kminst_lenet.pth --plot output/kminst_plot.png
<DBG> loading KMNIST train and test datasets...
<DBG> generating the train/validation split...
<DBG> initializing LeNet...
<DBG> training LeNet...
<DBG> EPOCH: 1/10
Train loss: 0.3635, Train accuracy: 0.8878
Val loss:   0.1369, Val accuracy: 0.9588

<DBG> EPOCH: 2/10
Train loss: 0.1011, Train accuracy: 0.9695
Val loss:   0.0943, Val accuracy: 0.9726
...
<DBG> EPOCH: 10/10
Train loss: 0.0092, Train accuracy: 0.9969
Val loss:   0.0810, Val accuracy: 0.9835

<DBG> total time taken to train the model: 51.89s
<DBG> evaluating network...
              precision    recall  f1-score   support

           o       0.95      0.98      0.96      1000
          ki       0.98      0.91      0.94      1000
```

Figure 2: KMIST Accuracy Plot over 10 epochs.

su	0.92	0.93	0.92	1000
tsu	0.96	0.96	0.96	1000
na	0.96	0.95	0.95	1000
ha	0.98	0.93	0.95	1000
ma	0.93	0.97	0.95	1000
ya	0.94	0.96	0.95	1000
re	0.96	0.97	0.97	1000
wo	0.95	0.96	0.95	1000
accuracy			0.95	10000
macro avg	0.95	0.95	0.95	10000
weighted avg	0.95	0.95	0.95	10000

The characters being classified are: o, ki, su, tsu, na, ha, ma, ya, re, and wo. The classification report is a useful utility that allows us to take a look at the overall recall and precision of our network. 0.95 is not that bad.

My generated accuracy graph in `output/kminst_plot.png` is shown in Figure 2. The accuracy keeps slowing edging up while the loss slowly edges down, which is what we want.

Now we can use the trained ConvNet persisted in `output/kminst_lenet.pth` for prediction.

```
python predict_lenet.py --model output/kminst_lenet.pth
<DBG> loading KMNIST test dataset...
<DBG>: ground truth: tsu, predicted: tsu
<DBG>: ground truth: tsu, predicted: tsu
<DBG>: ground truth: su, predicted: tsu
<DBG>: ground truth: ki, predicted: ki
<DBG>: ground truth: ma, predicted: ma
<DBG>: ground truth: ha, predicted: o
<DBG>: ground truth: tsu, predicted: tsu
<DBG>: ground truth: su, predicted: su
<DBG>: ground truth: su, predicted: su
<DBG>: ground truth: su, predicted: su
<DBG>: ground truth: na, predicted: na
<DBG>: ground truth: na, predicted: na
<DBG>: ground truth: ha, predicted: re
<DBG>: ground truth: su, predicted: su
<DBG>: ground truth: su, predicted: su
```

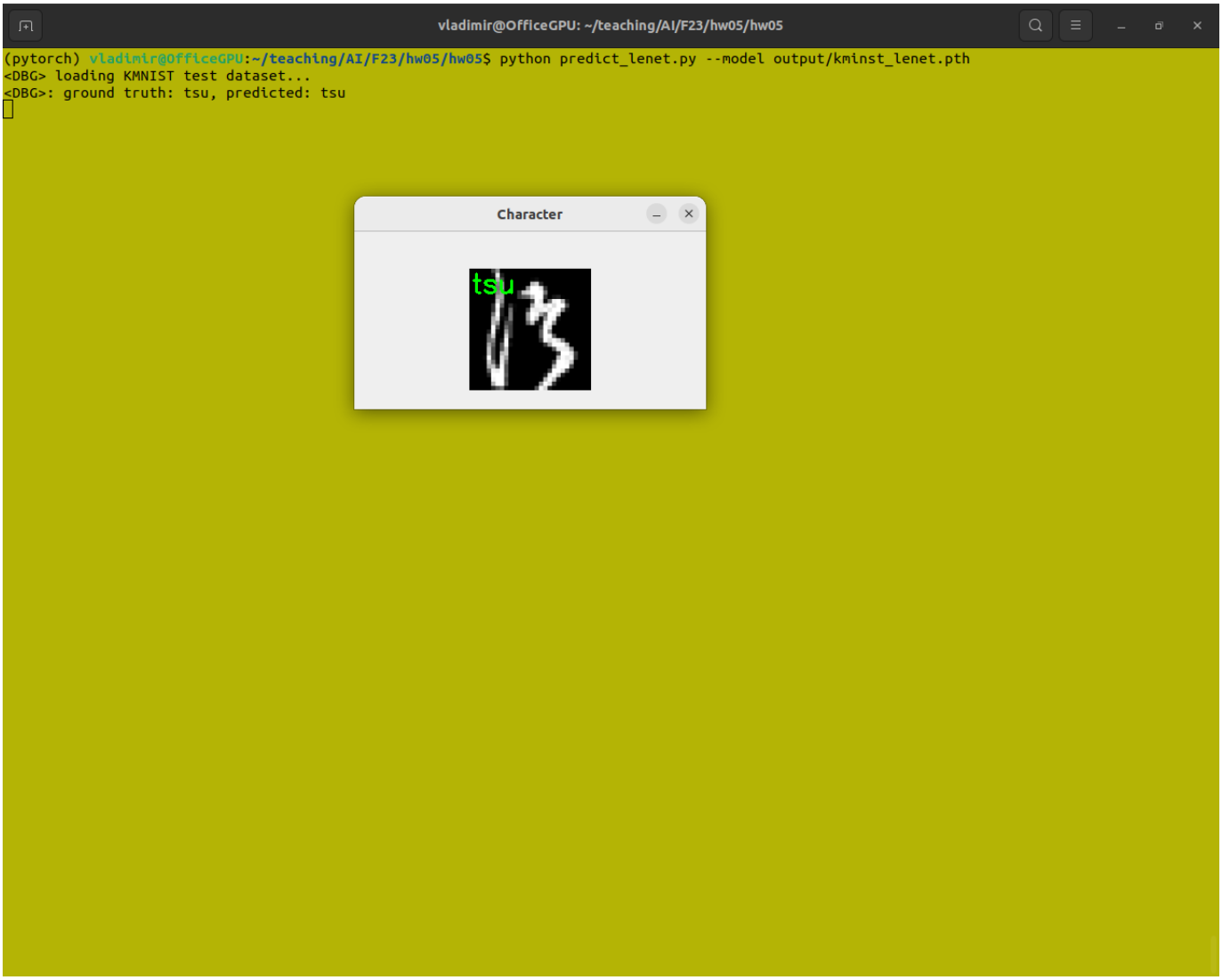

Figure 3: Correct prediction.

```
<DBG>: ground truth: ya, predicted: ya
<DBG>: ground truth: ki, predicted: ki
<DBG>: ground truth: su, predicted: su
<DBG>: ground truth: ma, predicted: ma
<DBG>: ground truth: tsu, predicted: tsu
```

I had 2 misclassifications: ha was misclassified as o and su as tsu. Figures 3 and 4 show my screenshots with one correct and one incorrect prediction, respectively. When a prediction is correct, the character's name in the top left corner is green. When it's incorrect, the character's name is red. You can hit Enter to go to the image of the next prediction.

Problem (4 points)

Now we can, finally, play and work on an DL problem. I want you to play with the following hyperparameters in `train_lenet.py` to see if you can do better than I did, i.e., higher than 0.95. I would start with increasing the number of epochs. Don't go wild, especially if you're on a CPU.

```
## iinitial learning rate
INIT_LR = 1e-3
## batch size
BATCH_SIZE = 64
## number of epochs
```

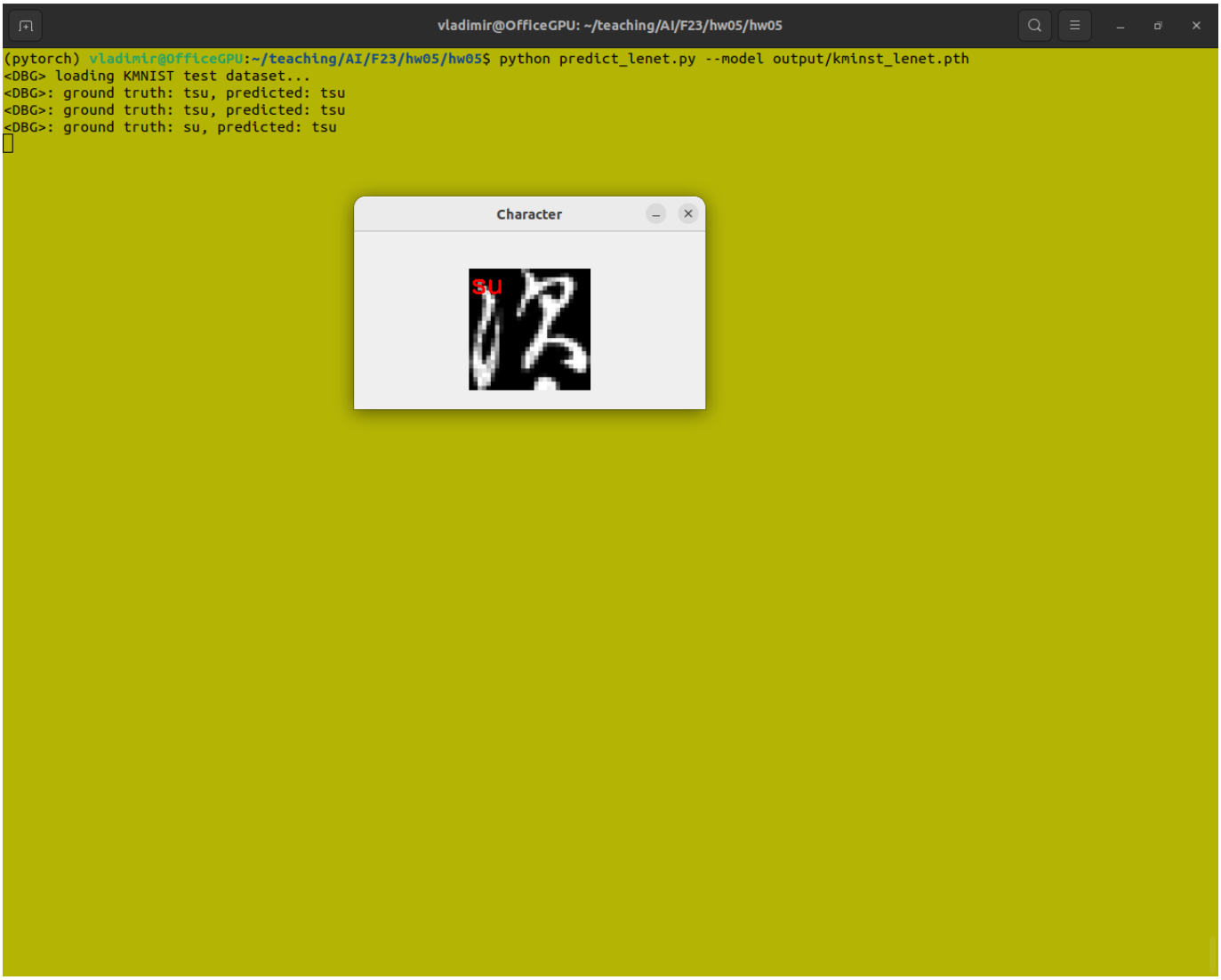

Figure 4: Incorrect prediction.

```
EPOCHS = 10
## train/test split
TRAIN_SPLIT = 0.75
VAL_SPLIT = 1 - TRAIN_SPLIT
```

Write 2 page report documenting what parameters you played with and the accuracy/loss you got. Experiment with at least 2 values of each parameter. You don't have to be verbose in your report, i.e., you can use the generated plots (a picture is worth a 1K words!) and the classification reports.

What to Submit

1. Your 2-page report saved as `lenet_kminst_report.pdf`.
2. Your best trained LeNet model persisted as `best_kminst_lenet.pth`.
3. Zip everything into `hw05.zip` and submit it on Canvas.

Happy Hacking!