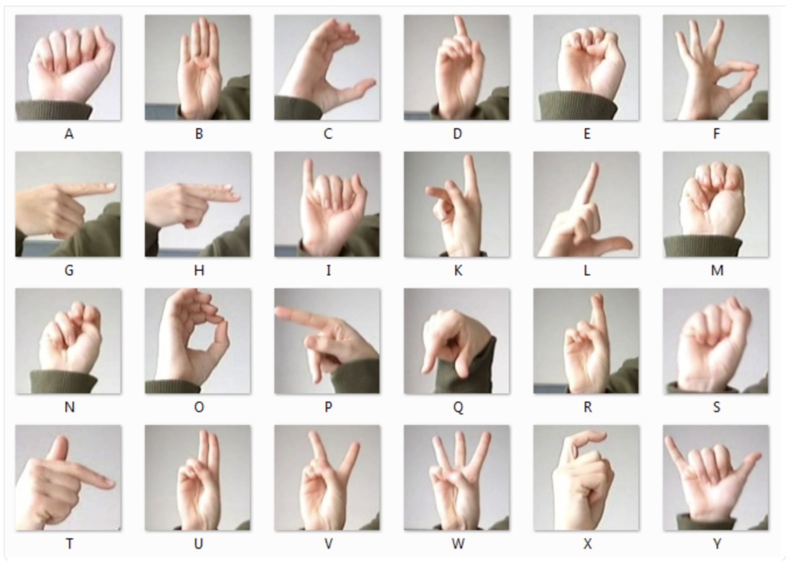
Intro to DL 2023 - Homework 2: Training a Convolutional Neural Network to Classify Images

Due: 26.02.2023

Introduction

In this homework, we will be building a convolutional neural network to classify image data. We will be working on the Sign Language MNIST dataset[1], which is an image dataset designed for the classification task:



The Sign Language MNIST dataset

You will develop a 24-class classifier that takes these images and returns the most likely letter it represents. (J and Z are excluded from the dataset because they require motion to express.) You will be experimenting with different regularization and optimization techniques. In addition, you will visualize the weights, or filters, learned by your model while training for this task.

The dataset is organized as follows *(This section is mostly taken from https://www.kaggle.com/datamunge/sign-language-mnist, refer to this link for more information)*: Each training and test

[1]https://www.kaggle.com/datamunge/sign-language-mnist

case represents a label (0-25) as a one-to-one map for each alphabetic letter A-Z (Label 9=J and and 25=Z are excluded from the dataset because they require motion to express.) The training data (27,455 cases) and test+dev data (7172 cases) are approximately half the size of the standard MNIST, with each image file composed of 28x28 pixels with grayscale values between 0-255. The original hand gesture image data represented multiple users repeating the gesture against different backgrounds. The Sign Language MNIST data came from greatly extending the small number (1704) of the color images included as not cropped around the hand region of interest. To create new data, an image pipeline was used based on ImageMagick and included cropping to hands-only, gray-scaling, resizing, and then creating at least 50+ variations to enlarge the quantity. The modification and expansion strategy was filters ('Mitchell', 'Robidoux', 'Catrom', 'Spline', 'Hermite'), along with 5% random pixelation, +/- 15% brightness/contrast, and finally 3 degrees rotation. Because of the tiny size of the images, these modifications effectively alter the resolution and class separation in interesting, controllable ways.

Accessing the Puhti Server

In this homework, you will be training a larger scale model. Therefore, you may want to use a GPU for your training. We will use the Puhti server of CSC (www.csc.fi) for this purpose.

First you will need to be able to reach the Puhti server. You can use your University of Helsinki username and password for this. Please check that you have access to Puhti by logging in through my.csc.fi, or via trying to remote connect to Puhti:

```
$ ssh -X USERNAME@puhti.csc.fi
```

Once you have a working username, please email or message your username to us, and we will grant you access to the `Intro-to-DL-2022` project with the code `2002605`. You will be using this project number to submit your processes to the gpu queue. Once you are added to the project, you need to go to my.csc.fi and confirm the User Agreement.

IMPORTANT NOTE: PLEASE MAKE SURE YOU SEND US YOUR USERNAME AS SOON AS POSSIBLE, SINCE THERE IS A LONGISH DELAY BETWEEN BEING ADDED TO A PROJECT AND BEING ABLE TO SUBMIT YOUR PROCESSES USING THIS PROJECT.

Getting Started

First download and unpack the archive `intro_to_dl_assignment_2.tgz`. Note that you will need to transfer this archive file to Puhti and then connect to Puhti if you would like to work on the gpu server:

```
$ scp intro_to_dl_assignment_2.tgz USERNAME@puhti.csc.fi:.
$ ssh USERNAME@puhti.csc.fi
puhti> tar -xvf intro_to_dl_assignment_2.tgz
```

This results in a directory `intro_to_dl_assignment_2` with the following structure

```
--data/
----sign_mnist_train
--------A
--------B
--------C
        ...
----sign_mnist_dev
--------A
--------B
--------C
        ...
----sign_mnist_test
--------A
--------B
--------C
        ...
--src/
----DL_hw2.py
----DL_hw2.ipynb
----run_DL_hw2.sh
```

The `data` folder contains train, dev, and test set folders, which are organized in subfolders corresponding to letter labels. The subfolders contain the image files. Do take a look at them! (ALWAYS LOOK AT YOUR DATA! :))

The `src` folder contains the starter code, as well as an example sbatch script for submitting your code as a batch job in the Puhti server.

Please note that the sbatch script supplied in the `src` folder works with a plain python (`.py`) file. However, the jupyter notebook starter code is also provided in case you would like to develop and debug your model in your local machine or on CSC Jupyter Notebooks interface. You may also want to develop your model locally for easy debugging at first, and transfer to working on Puhti when you are reasonably sure of the correct working of the model.

Running Your Program on the Puhti GPU Queues

There are several ways to run your code on Puhti. You can run the program as an interactive job (option 1), as a batch job (option 2) or using the Jupyter Notebooks interface (option 3). Lets see how each of these options work.

Option 1: Running interactive jobs

This option is an easy option especially when debugging, since we see the output of the running code in the terminal in real time. Be careful, however, that if your connection to Puhti is cut during the processing, your job is also automatically terminated.[2] We can use command *srun*

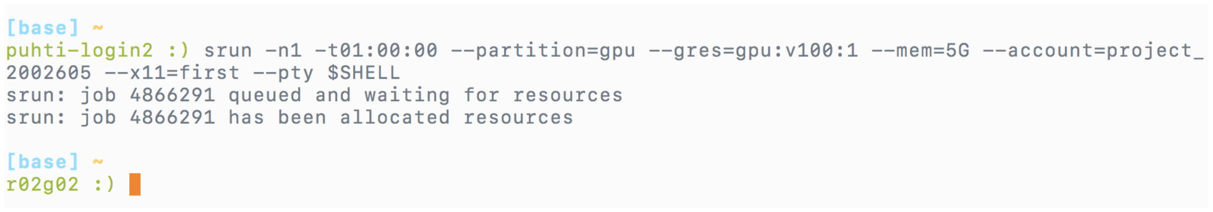
[2]To get around this, you can use the `screen` command, eg. https://www.howtoforge.com/linux$_s$*creen*. Let me know if you need help with `screen` in general.

to run interactive jobs on Puhti directly from the terminal. For example, this command will start an interactive batch job session, running in the default command shell (*$SHELL*):

```
puhti> srun -n1 -t01:00:00 --partition=gpu --gres=gpu:v100:1 --mem=5G \
        --account=project_2002605 --x11=first --pty $SHELL
```

(The `module load` command loads the necessary pytorch libraries for your program.)

When you connect successfully, you will see something like:



```
[base] ~
puhti-login2 :) srun -n1 -t01:00:00 --partition=gpu --gres=gpu:v100:1 --mem=5G --account=project_
2002605 --x11=first --pty $SHELL
srun: job 4866291 queued and waiting for resources
srun: job 4866291 has been allocated resources

[base] ~
r02g02 :) 
```

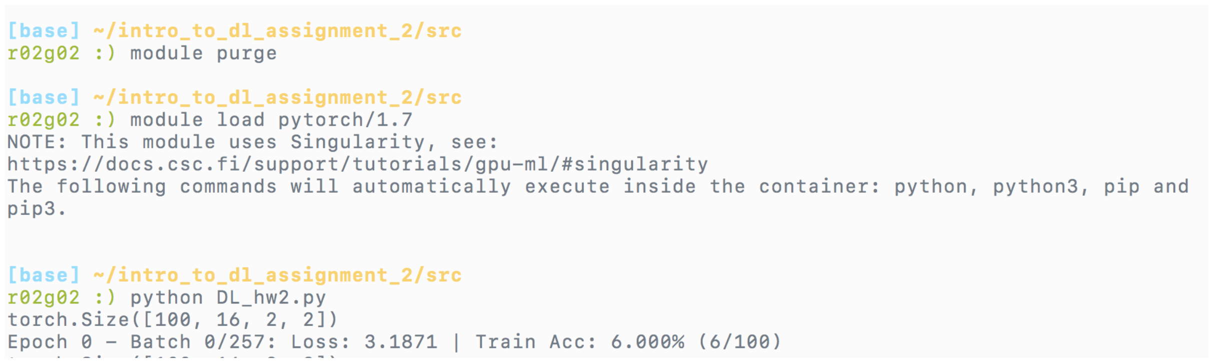
Fig. 1: Interactive job in terminal

If the resources are not readily available, you will have to wait in a queue. With options:

```
    --mail-type=BEGIN  --mail-user=<your email address>
```

you can set up to receive an e-mail once your job starts.

When you have the resources allocated, you can load the modules and run your program like you would on your local machine:



```
[base] ~/intro_to_dl_assignment_2/src
r02g02 :) module purge

[base] ~/intro_to_dl_assignment_2/src
r02g02 :) module load pytorch/1.7
NOTE: This module uses Singularity, see:
https://docs.csc.fi/support/tutorials/gpu-ml/#singularity
The following commands will automatically execute inside the container: python, python3, pip and
pip3.


[base] ~/intro_to_dl_assignment_2/src
r02g02 :) python DL_hw2.py
torch.Size([100, 16, 2, 2])
Epoch 0 - Batch 0/257: Loss: 3.1871 | Train Acc: 6.000% (6/100)
```

Fig. 2: Loading modules and running the program interactively

The `module` commands are important for loading the necessary pytorch libraries to your environment.

Note that the above `srun` command uses the `gpu` job queue in Puhti, however, you can use the `gputest` queue to test your code quickly provided that your job will take less then 15 minutes. The `gputest` queue typically has less waiting time than the `gpu` queue, but the downside is the strict time limit, so unless your job exits in 15 minutes by itself, then it is terminated forcefully. Please consider using this queue especially when you are debugging your script, if the waiting times for the `gpu` queue are too long. You can do this by changing the above command as:

```
puhti> srun -n1 -t00:15:00 --partition=gputest --gres=gpu:v100:1 --mem=5G \
        --account=project_2002605 --x11=first --pty $SHELL
```

For more detailed instructions on interactive jobs, see:
https://docs.csc.fi/computing/running/interactive-usage/.

Option 2: Submitting batch jobs

In the Puhti server, the first option to run our code file is by submitting it via an sbatch script, which looks like this:

```
#!/bin/bash
#SBATCH -n 1
#SBATCH -p gpu
#SBATCH -t 1:00:00
#SBATCH --mem=5000
#SBATCH --gres=gpu:v100:1
#SBATCH -J dl5
#SBATCH -o dl5.out.%j
#SBATCH -e dl5.err.%j
#SBATCH --account=project_2002605
#SBATCH

module purge
module load pytorch/1.7

python DL_hw2.py
```

The first part of the script includes settings about the job to be submitted. The second part loads the necessary pytorch libraries, and the third part runs your program `DL_hw2.py`.

When run with the `sbatch` command, this script submits your program to the gpu queue:

```
puhti> sbatch run_DL_hw2.sh
Submitted batch job 1248568
```

This will return an information message as above, showing the id of the submitted job. You can query your currently running jobs with the command:

```
puhti> squeue -u USERNAME
```

If you currently have a process that is either scheduled or running, its status and job id will be displayed with this command, eg:

```
puhti> squeue -u USERNAME
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           1248568       gpu      dl2  celikkan  R       2:56      1 r18g07
```

If you do not have any running processes (eg., your process has finished running), then this will display an empty line:

```
puhti> squeue -u USERNAME
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
```

The output of a submitted job is not displayed directly on the terminal, but is redirected to an output file, whose path you specify in the sbatch script with the command `#SBATCH -o`. Similarly, any errors that are generated by your process are not directly displayed on the terminal, but are redirected to an error file whose path is specified in the sbatch script with the command `#SBATCH -e`. In the example sbatch script, these files are set respectively as:

```
#SBATCH -o dl2.out.%j
#SBATCH -e dl2.err.%j
```

with `%j` denoting the job id. For example, the output of the above process will be dumped into the file `dl2.out.1248568`, and any errors generated will be dumped into the file `dl2.err.1248568`. You can see these logs via the commands:

```
puhti> less dl2.out.1248568
```

and

```
puhti> less dl2.err.1248568
```

respectively. (Note that due to buffering of the files, these logs may be written only after your process ends. Therefore, if you check these log files while your process is still running, you may see empty files.)

Also note that although the provided sbatch script by default uses the `gpu` job queue in Puhti, you are able to run your process in the `gputest` queue provided that your job will take less then 15 minutes. The `gputest` queue typically has less waiting time than the `gpu` queue, but the downside is the strict time limit, so unless your job exits in 15 minutes by itself, then it is terminated forcefully. Please consider using this queue especially when you are debugging your script, if the waiting times for the `gpu` queue are too long. You can make this change by setting in the sbatch script:

```
#SBATCH -p gputest
#SBATCH -t 0:15:00
```

For more detailed information about how to run batch jobs in the Puhti server, please refer to the CSC documentation: https://research.csc.fi/csc-guide-batch-jobs.

Option 3: Using Jupyter Notebooks

Instructions to be announced separately.

Starter Code

Let's look at the main program in `DL_hw2.py`. It

1. initializes the data loaders to use the image files
2. defines a CNN architecture
3. initializes a CNN model, an optimizer, and a loss function
4. trains the model for 10 epochs on the training data and then uses the model to classify the test data
5. visualizes the learned weights (or "filters") in the lowest convolutional layer

The data loading is handled in the starter code:

```
# We transform image files' contents to tensors
# Plus, we can add random transformations to the training data if we like
# Think on what kind of transformations may be meaningful for this data.
# Eg., horizontal-flip is definitely a bad idea.
# You can use another transformation here if you find a better one.
train_transform = transforms.Compose([
                                        #transforms.RandomHorizontalFlip(),
                                        transforms.ToTensor()])
test_transform = transforms.Compose([transforms.ToTensor()])

train_set = datasets.ImageFolder(DATA_DIR % 'train', transform=train_transform)
dev_set   = datasets.ImageFolder(DATA_DIR % 'dev',   transform=test_transform)
test_set  = datasets.ImageFolder(DATA_DIR % 'test',  transform=test_transform)


# Create Pytorch data loaders
train_loader = torch.utils.data.DataLoader(dataset=train_set,
                                           batch_size=BATCH_SIZE_TRAIN, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_set,
                                          batch_size=BATCH_SIZE_TEST, shuffle=False)
```

The train and test data are loaded from the `data` folder, and converted to torch tensors. Here you can add any random transformations you wish to the train data. Afterwards data loads are created from this data, with the corresponding batch sizes. Note that we shuffle the train data, as it is generally a good practice, but there is no need for shuffling the test data.

1 Assignment: Building up a Convolutional Neural Network

Your first task is to implement a convolutional neural network. The model is going to receive the images from the Sign Language MNIST dataset, and return the most likely label (out of 24 possible labels) for each image. The input will be a tensor of size `BATCH_SIZE_TRAIN x NUM_CHANNELS x WIDTH x HEIGHT`, where `NUM_CHANNELS=3` (R-G-B channels), and `WIDTH=HEIGHT=28` (pixels). As usual, you will fill in the `__init__` and `forward` functions within the skeleton code:

```
class CNN(nn.Module):
def __init__(self, num_classes=NUM_CLASSES):
    super(CNN, self).__init__()
    # WRITE CODE HERE
    pass

def forward(self, x):
    # WRITE CODE HERE
    pass
```

Please make sure your model includes at least two convolutional layers, followed by suitable non-linear functions and max pooling layers. Since there will be a rather large number of layers, please consider also to organize these layers into an `torch.nn.Sequential` module, which may result in cleaner code.

2 Assignment: Optimization and Regularization

Now that we are dealing with larger models, proper optimization and regularization techniques for our model will be more important. Please feel free to experiment with different methods. **Please also make sure that you try at least three different regularization techniques, and three different optimization techniques. Please also make sure to try early stopping as one your regularization techniques. Use validation set here to check for early stopping condition.**

Please include in your code files (as comments) or jupyter notebooks (as cells maybe?) a small report of the accuracy you reach with each of these techniques.

3 Assignment: Training and Testing

As usual, you will need to write the codes for training and testing your model:

```
#--- training ---
for epoch in range(N_EPOCHS):
    train_loss = 0
    train_correct = 0
    total = 0
```

```
        for batch_num, (data, target) in enumerate(train_loader):
            data, target = data.to(device), target.to(device)
            # WRITE CODE HERE

            print('Epoch %d - Batch %d/%d: Loss: %.4f
                 | Train Acc: %.3f%% (%d/%d)'
                 % (epoch, batch_num, len(train_loader),
                 train_loss / (batch_num + 1),
                 100. * train_correct / total,
                 train_correct, total))

        # WRITE CODE HERE
        # Please implement early stopping here.
        # You can try different versions, simplest way is to calculate the dev error and
        # compare this with the previous dev error, stopping if the error has grown.

    #--- test ---
    test_loss = 0
    test_correct = 0
    total = 0

    with torch.no_grad():
        for batch_num, (data, target) in enumerate(test_loader):
            data, target = data.to(device), target.to(device)
            # WRITE CODE HERE

            print('Evaluating: Batch %d/%d: Loss: %.4f
                  | Test Acc: %.3f%% (%d/%d)'
                  % (batch_num, len(test_loader),
                  test_loss / (batch_num + 1),
                  100. * test_correct / total,
                  test_correct, total))
```

Try to pull your accuracy as high as possible (although this will not be graded explicitly). What is the best performance you can achieve?

Note that parts of the starter code check for the possible existence of a CUDA device, and transfers the model and the data into the CUDA device if available. For example:

```
    if torch.cuda.is_available():
        device = torch.device('cuda')
    else:
        device = torch.device('cpu')

    model = CNN().to(device)

    ...
    data, target = data.to(device), target.to(device)
```

Please keep these parts as they are, since they provide seamless switching for us between running your codes on CPU or GPU.

Submission

Upload your `DL_hw2_USERNAME.py` or `DL_hw2_USERNAME.ipynb` file on Moodle.

Enjoy! :)