CS 475/675 Machine Learning: Homework 3

Due: Monday, March 21, 2022 11:59 pm

100 Points Total Version 1.2

Make sure to read from start to finish before beginning the assignment.

Changelog

Changes with v1.2 release:

- Fixed some parts in the instructions where the number of classes was marked as 5 instead of 6
- Fixed some parts of the instructions where input images were referenced as 26x26 instead of 28x28
- Fixed python commands to accept correct argparse arguments

Homework 3

This homework has two parts:

1. Analytical (30 points)
2. Programming (70 points)

All parts of the homework have the same due date. Late hours will be counted based on when the last part is submitted.

You may work with a partner on both parts of this assignment.

1 Programming (70 points total)

1.1 Overview

In this assignment you will learn to use PyTorch (http://pytorch.org/) to implement and train classifiers to recognize images of simple drawings using two distinct subsets of the Google QuickDraw dataset (https://quickdraw.withgoogle.com/).

Your grade will be based on a mixture of a) completing the outlined tasks below, and b) achieving high accuracy on a test set. Your accuracy will be evaluated on test data for which you do not have the labels. Grading will depend on the accuracy of your trained models. However, your grade will *not* depend on the classification accuracy achieved by your peers. Instead, we will assign some points based on previously-determined accuracy thresholds (more details below). Everyone has the opportunity to get full points on this project.

1.2 Requirements

You will need 3.7.6 or higher and you will need access to a Linux or MacOS machine: **PyTorch does not support Windows.** (Strictly speaking, you might be able to get PyTorch up and running on Windows, but we cannot help.) The CS department labs and servers offer support for Python3. Install PyTorch using pip3. We will be providing a requirements file for setting up your virtual environment.

1.2.1 Documentation

The documentation can be found at `http://pytorch.org/docs/stable/index.html`, and there are plenty more resources available online.

1.2.2 Introduction to PyTorch

While it is not required and there is nothing to turn in for this section, we strongly recommend that you go through the tutorial here to learn about pytorch: `http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html`.

- You may not have access to a GPU or a CUDA-enabled environment. This is okay; just modify these parts of the tutorial to run on the CPU, which should be fine since the tutorial is not computationally heavy.
- The point of this tutorial is to familiarize you with PyTorch, and in particular to familiarize you with the `torch.nn` and `torch.optim` packages. We suggest that you go through the examples from scratch, modifying parts whenever you are confused. If you simply copy and paste the code from the tutorial, you will likely have a much tougher time with the rest of this project.

1.3 Data

The QuickDraw dataset comes from a Google project where participants from around the world are asked to sketch an object in a short period of time, with the system attempting to guess what was drawn as the participant draws. To play the game, or to check out the data, visit `https://quickdraw.withgoogle.com/`.

In this project we'll keep things small (in terms of storage, memory, and computation) by dealing with only a small subset of the QuickDraw dataset. We've provided you with 6 classes of sketch data: apples, bananas, grapes, pineapples, strawberries, and watermelons. There are five dataset files provided:

- `fruit_images.npy` contains all of the training images (shape : `[72000,28,28]`)
- `fruit_labels.npy` contains all of the training labels (shape : `[72000,]`)
- `fruit_dev_images.npy` contains all of the validation images (shape : `[21600,28,28]`)
- `fruit_dev_labels.npy` contains all of the validation labels (shape : `[21600,]`)
- `fruit_test_images.npy` contains all of the testing images (shape : `[21600,28,28]`)

The files with `*_images.npy` as a suffix contain images stacked together in a NumPy array. In the case of textttfruit_images.npy, there are 72000 images of size 28 pixels x 28 pixels. The files with `*_labels.npy` contain a 1-D `uint8` array with shape `[72000]`.

This corresponds to the associated 72000 labels, where the i-th label is associated with the i-th image.

We recommend reshaping your input images consistently. For example, below we assume that each image is a *vector* rather than an *image*. Numpy implements this functionality in the `flatten` function (see `https://numpy.org/doc/stable/reference/generated/numpy.ndarray.flatten.html`). Note that you would apply this function to each individual image, not the entire matrix of images (i.e. it would result in a [72000, 784] matrix). However, this is up to you; the only requirement is that your models handle the input data appropriately. In addition, you should leave all labels intact. In particular, you will need to predict 0, 1, 2, 3, 4, or 5, with each integer corresponding to a particular class.

You may not use any data outside of the data provided by us, i.e. do not download additional data from the web.

1.3.1 Command Line Arguments

The code we have provided is a general framework for running Pytorch models. When you download the zip file, there will be several directories containing Python files within them. **The `TODO` comments in the code indicate where you will implement a function and/or part of the code.**

Within the compiled zip file, there are four directories. **DO NOT MOVE THESE DIRECTORIES OR THEIR CONTENTS**, for your code to work it relies on the directory structure.

1. `datasets` : contains all of the `*.npy` files mentioned (training, validation, testing).
2. `log_files` : empty, but when you run any of the `main_*.py` files, the accuracy and loss values will be logged in files stored here automatically.
3. `model_files` : At the end of each training session, all model files are automatically stored in this folder. Models will have suffix "`.pt`". You will be submitting copies of your models (which are stored in this folder)
4. `utils` : contains basic utils for running the program and contains the accuracy library, where you will be implementing some accuracy functions. You will be submitting the updated `accuracies.py` module.

There are four additional files at the root directory; the files starting with `main_*.py` are where you will be implementing your models and evaluating them. The bash script (`run.sh`) demonstrates some sample commands your program should be able to run without raising an error. Each of the `main_*.py` programs accepts the following generic arguments:

- `--mode`: A required argument, either `train` or `predict`. This will determine if your program is running in training or inference mode.
- `--LR`: (required for `train` mode) The learning rate to use with your optimizer during training.
- `--bs`: (required for `train` mode) The number of examples to include in your batch.
- `--epochs`: (required for `train` mode) The number of epochs (or steps) you want to train your model for.

- `--weights`: (required for `predict` mode) The path that points to a trained `*.pt` file that contains your model's weights.
- `--predictionsFile`: (required for `predict` mode) Points to the location your output model predictions will be stored (only used in `predict` mode)

The programs also accept a number of optional arguments, corresponding to directories used to store training data and store model outputs to.

- `--dataDir`: (optional) The directory that contains your training/validation/testing data. By default, it is set to the directory `datasets`.
- `--logDir`: (optional) The directory to write your model accuracy/losses to. By default, it is set to the directory `log_files`
- `--modelSaveDir`: (optional) The directory to write your model weights to after training. By default, it is set to the directory `model_files`

Do not change any of the above arguments. The commands we will run to evaluate your models on for test data are documented in `run.sh` under the header "For inference", with sample commands for training as well.

NOTE: When running each of the `main_*.py` files in training mode with the default optional arguments, your models and your logs will be stored in directories `model_files` and `log_files` by default. For example, if you run the file `main_densenet.py`, two files will be generated in the following directories:

- `model_files\202202280000_densenet.pt` : corresponds to your model's weights and files (with the datetime automatically added)
- `log_files\202202280000_densenet.log` : corresponds to your model's accuracy/losses.

When submitting your model weights in your zip file, **REMOVE THE DATE PREFIX FROM THE FILENAME**, otherwise it will not work with the autograder. (i.e. `202202280000_densenet.pt` becomes `densenet.pt`)

If you want to use Google Colab for running your program, an additional file contains a tutorial on how to set up your environment (`How2SetUpColabHW3.ipynb`) . Using Colab is not required for this assignment, as your models should be able to run on your computer (there is a 1MB model file size limitation described later in this document.) **You are still required to download the relevant files and zip them for submission**.

1.3.2 Introduction to the Data

1. While it is not required and there is nothing to turn in for this section, we strongly recommend that you go through the tutorial here to get familiar with the data. In a new notebook, run the following in the first cell. (The first line makes plots show up directly in your notebook.)

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

2. Identify the path to `fruit_images.npy` and `fruit_labels.npy` in the file downloaded (it should be stored in the directory `datasets`). The first file contains a

3-D `uint8` NumPy array with shape `[72000, 28, 28]`. This corresponds to 72000 images, each 28 pixels x 28 pixels. The second file contains a 1-D `uint8` array with shape `[72000]`. This corresponds to 72000 labels, where the i-th label is associated with the i-th image.

3. Load the images and labels:

```
images = np.load(PATH_TO_IMAGES)
labels = np.load(PATH_TO_LABELS)
```

4. Separate out class 0 and visualize the first image:

```
class_0_images = images[labels == 0]
plt.imshow(class_0_images[0])
plt.set_cmap('gray')
```

5. Repeat this for each of the classes (0-5). Which integer is associated with the apple class? The banana class? What about for grapes, pineapples, strawberries, watermelons?

6. For most of this project, you will treat each image as a *vector* rather than as a *matrix*. As you saw above, your images are currently stored in an array of shape `(num_images, height, width)`. Reshape the inputs to have shape `num_images, height * width`.

7. Print the new shape of `images`.

8. Plot the first 5 flattened vectors (corresponding to the first 5 images) using `plt.plot`. (This should result in 5 different lines on the plot. This is not intended to give you much information, but is included simply to emphasize the fact that your classifiers here will be taking in *flattened* images as input.)

1.4 Deliverables

In this assignment, all the coding is done in multiple python files. The four files you modify are named:

- `utils\accuracies.py`
- `main_densenet.py`
- `main_convnet.py`
- `main_bestmodel.py`

You will be submitting the following files onto gradescope.com as a zip file.

1. **.py files:** `main_densenet.py`, `main_convnet.py`, `main_bestmodel.py`, and `accuracies.py` are necessary, but you can include all .py files.

2. **.pt files:** These are your model files, we expect `densenet.pt`, `convnet.pt`, and `bestmodel.pt`

3. **.csv files:** These are your test prediction files, we expect `densenet_predictions.csv`, `convnet_predictions.csv`, and `best_model_predictions.csv`
4. **README.md:** For `bestmodel.pt`, you will create a README.md file listing what hyperparameters, network architecture, and other methods you tried, what seems to work, what seems to not work for your best model.
5. **Latex writeup:** The accompanying tex document for answers to the questions within this programming assignment.

Your code must be uploaded as `code.zip` with all of your code files in the root directory. By 'in the root directory,' we mean that the zip should contain *.py, *.torch and *.txt at the root (./*.py, ./*.torch, ./*.txt) and not in any sort of substructure (for example hw3/*.py). One simple way to achieve this is to zip using the command line, where you include files directly (e.g., *.py) rather than specifying a folder (e.g., hw3).

We will also be providing a template for you to answer the specific questions that will be graded. The questions will start with *** in these instructions. For each, please answer the question in the Latex template. You may be asked to include a figure in your answer.

Your final zip file should have the following files within them, with no additional directory structure.

1. `accuracies.py`
2. `main_densenet.py`
3. `main_convnet.py`
4. `main_bestmodel.py`
5. `densenet_predictions.csv`
6. `convnet_predictions.csv`
7. `best_model_predictions.csv`
8. `densenet.pt`
9. `convnet.pt`
10. `bestmodel.pt`
11. `README.md`

DO NOT UPLOAD THE DATA FILES TO GRADESCOPE.

1.5 Implementing a Simple Two-Layer NN (20 points)

You will implement a simple model to classify images from the QuickDraw dataset as either apple (0), banana (1), grape (2), pineapple (3), strawberry (4), or watermelon (5). You will use the `Sequential` module to map each input vector of length 28×28 to hidden layers, and essentially to a vector of length 6, which contains *class scores* or *logits*. You can think of these as real-valued scores for each class, **and these scores that should be the output of your models**. For instance, if you submit an apple image into your model, you should get as output $\begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0\end{bmatrix}$.

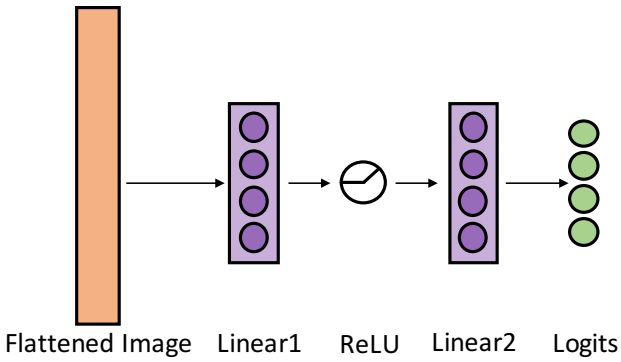

Figure 1: A diagram of the Linear model architecture.

Later, these scores will be passed directly to a loss function. For example, they can be exponentiated (making them nonnegative) and normalized (making them sum to 1) using the softmax function, and then used with cross-entropy loss. **In this section, you will *implicitly* rely on the softmax function through PyTorch's cross-entropy loss function, however you never need to use the softmax function directly.**

1. Complete the `TwoLayerDenseNet` class in `main_densenet.py` by implementing the network architecture diagrammed above (Fig 1.) and the feed forward function. This model uses one linear layer to map the inputs to `hidden_layer_width` hidden units, with ReLU activations, and then uses another linear layer to map from the hidden units to vectors of length 6. Set `hidden_layer_width` to 100.

2. Prepare your data for classification. You may need to be careful with types (unsigned ints vs. ints vs floats), and you may want to normalize your data (for example, you could normalize your data so that each individual image has a mean of 0.0 and a variance of 1.0).

3. Prepare constants as necessary. This should include (and is not limited to) things like `HEIGHT`, the height of each image, `WIDTH`, the width of each image, `N_CLASSES`, the number of classes, and `hidden_layer_width`, the dimension of the hidden layer, and the number of optimization steps to take during training. (under the `--epochs` argument)

4. Implement the simple accuracy function in `utils\accuracies.py`.

5. When you have finished implementing the model, train it using the following command:

```
python3 main_densenet.py --mode "train" \
                  --dataDir "datasets" \
                  --logDir "log_files" \
                  --modelSaveDir "model_files" \
                  --LR 0.0001 \
                  --bs 100 \
                  --epochs 2000
```

   We have set the learning rate to `1E-4`, batch size to 100, and number of training steps (epochs) to 2000.

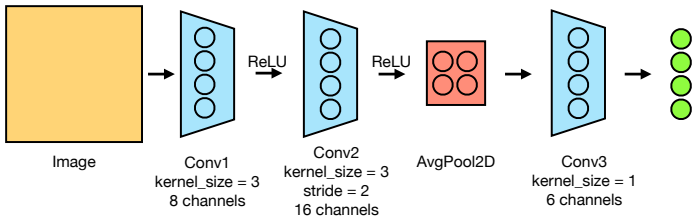

Figure 2: A diagram of the Convolutional net model architecture. (note the number of "green" dots is not reflective of the number of classes you should be outputting (6))

6. Start a hyperparameter search to optimize `TwoLayerDenseNet` performance. For this model, only experiment with `--LR`. A good development accuracy is around 80%.

7. Save your version of the model which uses default hyperparameters for all values except for learning rate. Use the best learning rate you found from your hyperparameter search. Name the model you submit `densenet.pt`. Run this model on the test data using `predict` mode, and name the predictions `densenet_predictions.csv`.

```
python3 main_densenet.py --mode "predict" \
            --dataDir "datasets" \
            --weights "model_files/densenet.pt" \
            --predictionsFile "densenet_predictions.csv"
```

   Submit both the model file and the predictions file.

8. *** What accuracy would a random classifier (which assigns labels randomly) achieve on this task (approximately, answer in 1.5.8a)? What accuracy would a majority-vote classifier achieve on this task (approximately, answer in 1.5.8b)? Round your answer to the closest whole integer.

9. *** Behind the scenes, the `torch.nn.Linear` module is creating parameters for you (and initializing those parameters to reasonable values). In this particular case, how many weights (answer in 1.5.9a) and how many biases (answer in 1.5.9b) are being created?

10. *** What were your best settings for learning rate?

11. *** Include a plot of training accuracy and validation accuracy as a function of epoch step for your best settings (replace blank.png with your file name in the box). You can create a python notebook to aid in plotting these figures.

12. *** For this problem set, we did not give you the full dataset for training. How could you go about increasing your model's performance with a limited dataset (i.e. if we gave you 100 images of each class, how could you get more 'value' out of the dataset while limiting overfitting)?

1.6 Implementing a Simple Convolutional NN (20 points)

In this section you will create a simple convolutional neural network (in fact, *too* simple) for multiclass classification, again using cross-entropy loss. As a reminder, 2-D convolutional neural networks operate on *images* rather than *vectors*. Their key property is that hidden units are formed using local spatial regions of the input image, *with weights that are shared over spatial regions*. To see an animation of this behavior, see `https://github.com/vdumoulin/conv_arithmetic`. In particular, pay attention to the 1st animation (no padding, no strides) and to the 9th animation (no padding, with strides). Here, one 3x3 convolutional filter is swept over the image, with the image shown in blue and the output map (which is just a collection of the hidden units) shown in green. You can think of each convolutional filter as a small, translation-invariant pattern detector.

1. Complete the `TooSimpleConvNN` class in `main_convnet.py` by implementing the network architecture diagrammed in Figure 2 and the feed forward function. In short, this model a) reshapes our vectors back into images; b) applies two convolutional layers; c) averages each channel spatially, so that each 'image' ends up with a height and width of 1; and finally d) applies a final convolution to yield 6 channels. In the end, the output has 6 channels with 1 "pixel", and these 6 channels correspond to class scores (a Tensor with shape [BATCH_SIZE, N_CLASSES]).

2. *** Above we mentioned that convolutional filters are applied to local image regions, with weights shared across regions. How does this compare to fully-connected neural networks? (answer in 1.6.2)

3. Experiment with the batch size, the learning rate, and the number of steps to try to maximize validation accuracy.

4. Save your version of the model. Name the model you submit `convnet.pt`. Run this model on the test data using `predict` mode, and name the predictions `convnet_predictions.csv`.

```
python3 main_convnet.py --mode "predict" \
                --dataDir "datasets" \
                --weights "model_files/convnet.pt" \
                --predictionsFile "convnet_predictions.csv"
```

   Submit both the model file and the predictions file.

5. *** Plot training accuracy and validation accuracy as a function of optimization step for your best settings.

6. *** Describe the best parameters you obtained for your model (batch size, epochs, learning rate). What was the corresponding batch size and learning rate? How many optimization steps did you need to take to reach that accuracy? How long did training take ?

7. *** What was the best validation accuracy achieved?

1.7 Maximizing Performance (15 points)

Here your goal is to generate a model that will maximize performance on validation accuracy. Feel free to vary the optimizer, mini-batch sizes, the number of layers and/or

the number of hidden units / number of filters per layer; include dropout if you'd like do; etc. You can even go the extra mile with techniques such as data augmentation, where input images may be randomly cropped and/or translate and/or blurred and/or rotated etc. We've added the `scikit-image` (`https://scikit-image.org/`) package to the `requirements.txt` file, if you want to take this approach.

However, there are a couple limitations: You may not add additional training data, and you must be able to store your model in a `.pt` file no bigger than 1MB.

Hints: You may want to focus on convolutional networks, since they are especially well suited for processing images. You may often find yourself in a situation where training is just *too slow* (for example where validation accuracy fails to climb after a 5 minute period). It is up to you to cut experiments off early: if training in a reasonable amount of time isn't viable, then you can try to change your network or hyperparameters to help speed things up. In addition, earlier we repeatedly asked that you vary other parameters as necessary to maximize performance. There is obviously an *enormous* number of possible configurations, involving optimizers, learning rates, mini-batch sizes, etc. Our advice is to find settings that consistently work well across simple architectures, and to only adjust these settings if insights based on training and validation curves suggest that you should do so. In particular, remember that Adam helps you avoid manual learning-rate tuning, and remember that very small minibatches and very large minibatches will both lead to slow performance, so striking a balance is important.

1. Complete the `BestModel` model in `main_bestmodel.py`. You need to define the features for this model, as well as the forward function.
2. Train this network. Remember, it's up to you to ensure that you can train your model in a reasonable amount of time! You should be able to run the following commmand:

```
python3 main_bestmodel.py --mode "train" \
                    --dataDir "datasets" \
                    --logDir "log_files" \
                    --modelSaveDir "model_files" \
                    --LR 0.01 \
                    --bs 100 \
                    --epochs 2000
```

3. We care about your ideas and work on designing this model. Create a `README.md` file listing what hyperparameters, network architecture and other methods you tried, what seems to not work, and what seems to work in general.
4. Save your best model as `bestmodel.pt` and its test predictions as `best_model_predictions.csv`. **Submit the model file, the predictions file, and the README.md**

To receive full credit, you need to achieve a test accuracy which indicates that you have put a sufficient amount of effort into building a good classifier. This should improve over the best model in previous sections by a reasonable amount. Your model also should not be identical to the ones generated in prior sections.

1.8 Exploring Failure Modes (15 points)

Here you'll explore the failure modes of your best model.

1. You can continue to modify the `main_bestmodel.py` file for this analysis.
2. Locate some success cases and some failure cases in the validation set. (In other words, find some images that were correctly classified by your best model, and some that were misclassified by your best model).
3. Visualize 10 correctly-classified images and 10 incorrectly-classified images using `plt.imshow`.
4. *** Are there any qualitative differences between these sets of images (answer in 1.8.4a)? Are the misclassified examples more difficult for you to classify as a human (answer in 1.8.4b)?
5. Create a copy of the 10 correctly-classified images and add Gaussian noise to them, with a standard deviation that is approximately one tenth of the image's range. (For example, if the pixel values of the images range from -10 to 10, the range is 20, so you would use a standard deviation of 2.0.)
6. *** Visualize these perturbed images and classify them with your model. Select one of these images and place in in the answer box, replacing Blank.png with the image's filename. (answer in 1.8.6)
7. *** Does the classifier still classify all 10 images correctly? (answer in 1.8.7)
8. Create a copy of the 10 correctly-classified images and flip them vertically.
9. Visualize these flipped images and classify them with your model.
10. *** Does the classifier still classify all 10 images correctly? (answer in 1.8.10)
11. *** Let's assume that your model "failed" to classify the flipped images correctly. First of all, is this "failure" necessarily a failure? In other words, do scenarios exist in which you do not want to remain invariant to horizontal flipping? Now, suppose that in this application you do want to remain invariant to horizontal flipping. How could you change your training process so that the model remains robust to such transformations? (answer in 1.8.11)

2 What to Submit

In this assignment you will submit:

1. *YOURJHIDHERE_hw3_programming.pdf*. **Your writeup must be compiled from latex and uploaded as a PDF**. The writeup should contain all of the answers to the analytical questions asked in the assignment as well as the answers to the questions marked with ******* in the programming. Make sure to include your name in the writeup PDF and to use the provided latex template for your answers following the distributed template. Put your JHID as the prefix to this document's filename. You will submit this to the assignment called "Homework 3: Programming".

2. *YOURJHIDHERE_code.zip*. See the deliverables section for more details on what files we want. As with the the programming writeup, put your JHID as the prefix to the zip archive's filename. You will submit this to the assignment called "Homework 3: Code".

3. *YOURJHIDHERE_hw3_analytical.pdf*. **Your writeup must be compiled from latex and uploaded as a PDF**. This should contain all of the answers to the analytical questions. Make sure to include your name in the writeup PDF and to use the provided LaTeX template for your answers following the distributed template. You will submit this to the assignment called "Homework 3: Analytical".

3 Questions?

Remember to submit questions about the assignment to the appropriate group on Piazza: `https://piazza.com/class/kynwo1nr3fx6ss`.