# CS 475 Machine Learning: Homework 5
# Deep Learning: Programming

Due: Monday November 16, 11:59pm

45 Points Total Version 1.0

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

## Homework 5

For the remaining homeworks, we will combine all three homework types into single assignments. Both homework 5 and 6 will be worth 100 points, and the total amount of homework points will total 400.

Homework 5 has three parts totalling 100 points.

1. Analytical (40 points)
2. Programming (45 points)
3. Lab (15 points)

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

For this assignment you may only work with a partner on the analytical section.

## 1 Programming (45 points)

### 1.1 Overview

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

Your grade will be based on the performance on held-out test data of 3 models which you will implement. Additionally, the 3 models will also be used in the Lab section of this assignment.

**Requirements** This assignment uses Python 3.7. The Python libraries that you will need (and are allowed to use) are given to you in the `requirements.txt` file. Install them using pip3: `pip3 install -r requirements.txt`.

**Suggestions** Deep learning is much faster on a GPU, so you might prefer to run things on the Computer Science undergraduate or graduate grid, or Google colab. This includes the lab (Jupyter notebook) portion of this homework. However, this is not required and you should be able to this assignment on your own computer. The grid will make it easier to run multiple experiments in parallel.

Additionally, since we will be exploring many parameter settings for each model, you may want to store the models while you explore parameters, and then turn in the model with the best performance. This way you don't need to retrain a model once you've decided on the best hyperparameters for that model.

### 1.2 Introduction to PyTorch

PyTorch documentation can be found at `http://pytorch.org/docs/stable/index.html`. There are numerous PyTorch resources online.

While it is not required, we strongly recommend that you go through this tutorial: `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.

### 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 8 classes of sketch data: 'ambulance', 'apple', 'banana', 'basketball', 'bed', 'book', 'bread' and 'broccoli'. There are 12,000 examples in each class, to a combined 96,000 examples. We have already split the dataset into train (57,600), dev (19,200) and test (19,200).

The dataset is provided as 5 separate `.npy` files. In particular, you should have `train.feats.npy`, `train.labels.npy`, `dev.feats.npy`, `dev.labels.npy`, and `test.feats.npy`.

You will train your models on the `train` data, and evaluate it on the `dev` data. As in past assignments, the unlabeled `test` data is just there for you to make sure you can run your model on it without crashing. Ultimately, we will be running your models on `test` data (with actual labels), and evaluating your model's output.

The data loader we have provided expects all these files to be in the same directory, and loads them by name, so if you rename these files or move them into separate directories, be sure to change the data loader, or update the way you load data.

### 1.4 Command Line Arguments

The code we provide is a general framework for running pytorch models. It has the following generic arguments:

- `mode`: A non-optional argument. It is either `train` or `predict`. This must be the first argument you pass when running `main.py`.
- `--data-dir`: Should point to the directory where your data files are stored (see the above section).

- `--log-file`: Points to the location where a `csv` file containing logs from your model's training should be stored (only used during `train` mode).
- `--model-save`: Points to the location where you're model will be stored after training (during `train` model) or where the stored model should be loaded from (during `test` mode).
- `--predictions-file`: Points to where to output model predictions (only used during `test` mode).

Do not change these arguments.

Specifically, when we run your models on test data, the command will be (for instance on a saved ff model):

```
python3 main.py predict --data-dir ./data/ --model-save ff.torch
 --predictions-file ff-preds
```

Before you submit a saved model, make sure you can run this command (with the correct data directory) and that it outputs valid test predictions.

`main.py` also has the following generic hyperparameters (hyperparameters used in the training of all models):

- `--model`: This is the type of model to train. There are 3 models we will build: `simple-ff`, `simple-cnn`, and `best`.
- `--train-steps`: The number of steps to train on. Each step trains on one batch.
- `--batch-size`: The number of examples to include in your batch.
- `--learning-rate`: The learning rate to use with your optimizer during training.

While you should not modify these parameters, you may add to them to experiment with different models. You will then need to modify the `train` function in `main.py`. You can change the training loop to make it better. As long as we can load your stored model using your `test` method, you can train it however you'd like.

The file has a few *model-specific* arguments. You will likely need to add to these when building your `best` model. The feed forward model has *one* model-specific hyperparameter:

- `--ff-hunits`: The number of hidden units in feed-forward layer 1.

And the CNN model has *three* model-specific hyperparameters:

- `--cnn-n1-channels`: The number of channels in the first conv layer.
- `--cnn-n1-kernel`: The size of the square kernel of the first conv layer.
- `--cnn-n2-kernel`: The size of the square kernel of the second conv layer.

What values for these parameters, and what additional parameters you add are up to you!

If you add additional parameters, they should be **optional parameters only.** If we run your train method, we will only provide the parameters listed above. Any other parameters you add must be optional to avoid an error.

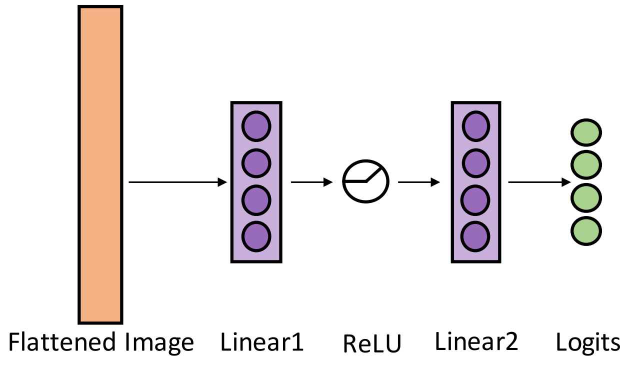

Figure 1: A diagram of the Linear model architecture.

## 1.5 Implementing a Simple Two-Layer NN (10 points)

You will implement a simple model to classify images of size $28 \times 28 = 784$ pixels from the custom QuickDraw dataset into one of the 8 classes. You will use the `Sequential` module to map each input image to hidden layers and eventually to a vector of length 8, which contains *class scores* or *logits*. Since the data comes in the form of vectors of length 784, you can pass it in as is.

These scores will be passed directly to a loss function. In this homework, we will use cross-entropy as our loss function. **You will *implicitly* rely on the softmax function through PyTorch's cross-entropy loss function however, so you never need to use the softmax function directly.**

1. Complete the `SimpleFF` class in `models.py` by implementing the `forward` function. (See Figure 1) This model uses one linear layer to map from the inputs to $n$ hidden units, with ReLU activations, and uses another linear layer to map from the hidden units to vectors of length 8. You can apply relu activations to the output of the first layer with `torch.nn.functional.relu`.

2. When you have finished implementing the model, train it via `main.py`. An example command might be: `python code/main.py train --data-dir data --log-file logs/ff-logs.csv --model-save models/ff.torch --model simple-ff`

3. You may start the hyperparameter search for `SimpleFF`. For this model, only experiment with `--learning-rate`. A good development accuracy is around 80%.

4. Save a 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 `ff.torch`. Run this model on the test data using `predict` mode, and name the predictions `ff-predictions.txt`. **Submit both the model file and the predictions file.**

## 1.6 Implementing a Simple Convolutional NN (15 points)

The next step is to create a convolutional neural network for multiclass classification, again using cross-entropy loss. As a reminder, 2-D convolutional neural networks operate on *images* rather than *vectors*. (This means that your model will need to reshape the input into a 2 dimensional image!) Their key property is that hidden units are formed using local

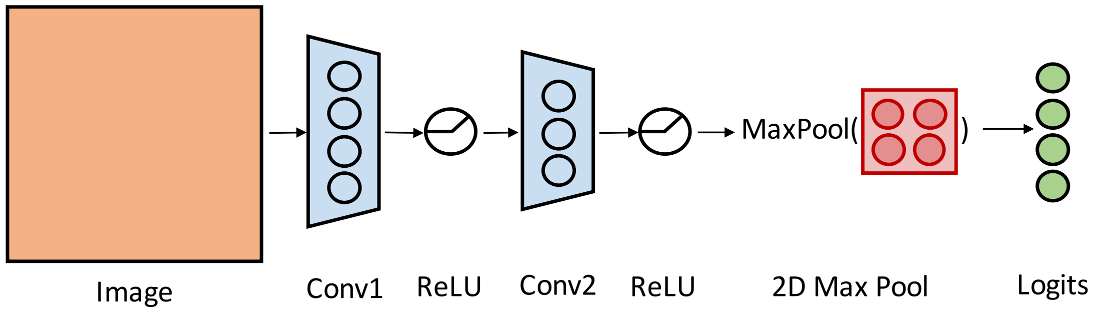

Figure 2: A simple diagram of the CNN model architecture.

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 first animation (no padding, no strides) and to the 9th animation (no padding, with strides). Here, a $3 \times 3$ convolutional filter is run 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.

We've provided the variables you should use for this model. The first CNN layer is a `k` $\times$ `k` convolution that takes in an image with one channel and outputs an image with `c` channels (where `k` is the value of `--cnn-n1-kernel` and `c` is the value of `--cnn-n1-channels`. The second conv layer is a `k2` $\times$ `k2` convolution that takes in an image with `c` channels and outputs an image with 8 channels (where `k2` is the value of `--cnn-n2-kernel`). The output image has approximately half the height and half the width because of the stride of 2.

You can see that this model has 3 hyperparameters (`k`, `k2`, `c`) in addition to the learning rate, optimizer, batch size, and number of training iterations. Use the Lab assignment to help select a good number of channels for our first CNN layer.

1. Complete the `SimpleConvNN` model in `models.py`, by implementing the forward function. Remember, the output of this model should be logits! (See Figure 2) You should apply a ReLU activation to the output of the 2 convolutional layers. Implement this model to feed the output of the first convolution into the second convolution. Then, you should have an output which is of size `[batch_size, 8, 8, 8]`. To convert this into logits, you should use a max-pooling layer (for example, the one here `https://pytorch.org/docs/stable/nn.functional.html#max-pool2d`) over the channels. This will give us a vector of 8 items, each the max value for it's respective channel in the previous output.

2. This model will take a lot more steps to train than the previous model. To know whether you have trained long enough I recommend to look at the graph of the dev loss over training steps. Does it seem like it could still be trending upward? That means you need to train longer! Try to find a setting that at least matches your ff model on dev data.

3. Save your best CNN model as `cnn.torch` and the test predictions it makes as `cnn-predictions.txt`. **Submit both the model file and the predictions file.**

### 1.7 Best Model (20 points)

In this section your goal is to maximize performance. Do whatever you'd like to your network configurations to maximize 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 `.torch` 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 `BestNN` model in `models.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!
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 `best.torch` and it's test predictions as `best-predictions.txt`. **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.

## 2 Miscellaneous

### 2.1 Python Libraries

We will be using Python 3.7.6. We are not using Python 2, and will not accept assignments written for this version. We recommend using a recent release of Python 3.7.x, but anything in this line (e.g. 3.x) should be fine.

For each assignment, we will tell you which Python libraries you may use. We will do this by providing a `requirements.txt` file. You should only use libraries available in the basic Python library or the `requirements.txt` file. In this and future assignments we will allow you to use `numpy`, `torch` and `scikit-image`.

It may happen that you find yourself in need of a library not included in the `requirements.txt` for the assignment. You may request that a library be added by posting to Piazza. This may be useful when there is some helpful functionality in another library that we omitted. However, we are unlikely to include a library if it either solves a major part of the assignment, or includes functionality that isn't really necessary.

## 2.2 Grading Programming

The programming section of your assignment will be graded using an autograder in Gradescope. Your code will be run using the provided command line options, as well as other variations on these options (different parameters, data sets, etc.) The grader will consider the following aspects of your code.

1. **Exceptions:** Does your code run without crashing?
2. **Output:** Some assignments will ask you to write some data to the console. Make sure you follow the provided output instructions exactly.
3. **Accuracy:** If your code works correctly, then it should achieve a certain accuracy on each data set. While there are small difference that can arise, a correctly working implementation will get the right answer.
4. **Speed/Memory:** As far as grading is concerned, efficiency largely doesn't matter, except where lack of efficiency severely slows your code (so slow that we assume it is broken) or the lack of efficiency demonstrates a lack of understanding of the algorithm. For example, if your code runs in two minutes and everyone else runs in 2 seconds, you'll lose points. Alternatively, if you require 2 gigs of memory, and everyone else needs 10 MB, you'll lose points. In general, this happens not because you did not optimize your code, but when you've implemented something incorrectly.

## 2.3 Code Readability and Style

In general, you will not be graded for code style. However, your code should be readable, which means minimal comments and clear naming / organization. If your code works perfectly then you will get full credit. However, if it does not we will look at your code to determine how to allocate partial credit. If we cannot read your code or understand it, then it is very difficult to assign partial credit. Therefore, it is in your own interests to make sure that your code is reasonably readable and clear.

## 2.4 Code Structure

Your code must support the command line options and the example commands listed in the assignment. Aside from this, you are free to change the internal structure of the code, write new classes, change methods, add exception handling, etc. However, once again, do not change the names of the files or command-line arguments that have been provided. We suggest you remember the need for clarity in your code organization.

### 2.5 Knowing Your Code Works

How do you know your code really works? That is a very difficult problem to solve. Here are a few tips:

1. **Use Piazza.** While you **cannot share code**, you can share results. We encourage you to post your results on dev data for your different algorithms. A common result will quickly emerge that you can measure against.
2. **Output intermediate steps.** Looking at final predictions that are wrong tells you little. Instead, print output as you go and check it to make sure it looks right. This can also be helpful when sharing information on Piazza.
3. **Debug.** Find a Python debugger that you like and use it. This can be very helpful.

### 2.6 Debugging

The most common question we receive is "how do I debug my code?" The truth is that machine learning algorithms are very hard to debug because the behavior of the algorithm is unknown. In these assignments, you won't know ahead of time what accuracy is expected for your algorithm on a dataset. This is the reality of machine learning development, though in this class you have the advantage of your classmates, who may post the output of their code to the bulletin board. While debugging machine learning code is therefore harder, the same principles of debugging apply. Write tests for different parts of your code to make sure it works as expected. Test it out on the easy datasets to verify it works and, when it doesn't, debug those datasets carefully. Work out on paper the correct answer and make sure your code matches. Don't be afraid of writing your own data for specific algorithms as needed to test out different methods. This process is part of learning machine learning algorithms and a reality of developing machine learning software.

## 3 What to Submit

You will need to create an account on gradescope.com and signup for this class. The course is https://www.gradescope.com/courses/153788.

There are three file types that you must submit:

1. **.py files:** Only your `classify.py` and `models.py` are necessary, but you can include all .py files.
2. **.torch files:** These are your model files, we expect `ff.torch`, `cnn.torch` and `best.torch`.
3. **.txt files:** These are your test prediction files, we expect `ff-predictions.txt`, `cnn-predictions.txt` and `best-predictions.txt`.
4. **README.md:** file describing your `best.torch` model, what hyperparameters, network architecture and other methods you tried, what seems to work and not work in general.

Submit the files listed above to gradescope.com as a zip file. Your code must be uploaded as code.zip with your code 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 hw5/*.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., hw5).

Remember to submit questions about the assignment to the appropriate group on Piazza: https://piazza.com/class/kdfbbwz3hwb3an.