

CS 475/675 Machine Learning: Homework 3

Due: Monday, October 25, 2021, 11:59 pm US/Eastern
70 Points Total Version 1.0

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

1 Homeworks

This homeworks will contain three parts:

1. **Analytical:** These analytical questions will consider topics from the course. These will include mathematical derivations and analyses. Your answers will be entirely based on written work, i.e. no programming.
2. **Programming:** The goal of this programming assignment is for you to get familiar with PyTorch and train classifiers to recognize simple images of clothing articles using a subset of the FashionMNIST dataset.
3. **Practicum:** In the practicum portion of the assignment, you will try 3 types of simple visualizations surrounding the training of your neural networks.

[Click here for the Practicum Google Colab Notebook](#)

1.1 Collaboration Policy

For this assignment, all three parts (the programming, analytical problems and practicum) is intended to be done with a partner (e.g. teams of two students). However, you may also choose to work on and submit the assignment by yourself. If you choose to work with a partner, you and your partner will make one submission for the two of you on Gradescope (make sure to include your partner's information when you submit). You and your partner will receive the same grade, so please choose your partner carefully.

2 Programming (25 points)

In this assignment you will learn to use [PyTorch](#) to train classifiers to recognize simple images of clothing articles using a subset of the [FashionMNIST dataset](#).

2.1 Requirements

You will need Python 3.6. 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.
```

2.2 Suggestions

Since some of these experiments will take some time to run, and will eat up CPU usage, you might prefer to run things on the Computer Science undergraduate or graduate grid, including the Jupyter notebook. However, this is not required as you should be able to run everything on your own computer. Running on 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.

2.3 Data

The data we will use for this assignment is a subset of the FashionMNIST dataset. Our version consists of 70,000 images (7,000 test, 7,000 dev, 56,000 train). Each image is a 28x28 grayscale article of clothing, and is labeled out of 10 classes. The classes are Shirt, Sneaker, Bag, Ankle Boot, T-shirt, Trouser, Pullover, Dress, Coat, Sandal (with label number in that order). [Read more here](#). Note, though, that the order of our labels doesn't match the order on the website.

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

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.

2.4 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 your model will be stored after training (during train mode) 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.

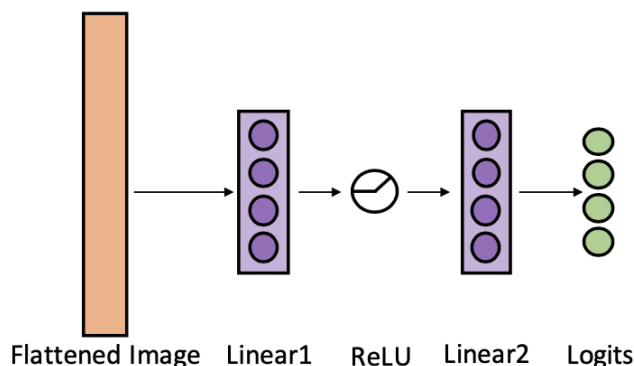


Figure 1: A diagram of the Linear model architecture.

2.5 Jupyter Notebook Visualization

We will use a Jupyter notebook to visualize different aspects of NN training.

You should start this homework in the Jupyter Notebook we provide. You will be asked to walk through some initial data visualization cells, so that you can become familiar with the data. Eventually you will need to return to your code, and implement a model in order to continue using the notebook.

The notebook serves two purposes. First, the notebook will help you debug your model while you are building it. That is, each model has a portion of the notebook devoted to plotting loss and accuracy, as well as doing hyperparameter sweeps. Once you have implemented those cells, you can use them to help you further debug your models and make decisions about hyperparameter values.

Make sure that you have worked through the entire notebook and run all cells before you finish. We will grade your notebook.

2.6 Implementing a Simple Two-Layer NN

You will implement a simple model to classify images of size $28 \times 28 = 784$ pixels from the FashionMNIST dataset into one of the 10 classes. You will use the Sequential module to map each input image to hidden layers and eventually to a vector of length 10, 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. You may recall logits from homework 1, where we were also doing multiclass classification. The output of your models will be these scores.

These scores will be passed directly to a loss function. In this homework, we will use cross-entropy as our loss function. This is exactly the same loss as we used for our multiclass logistic regression model in homework 1. 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.

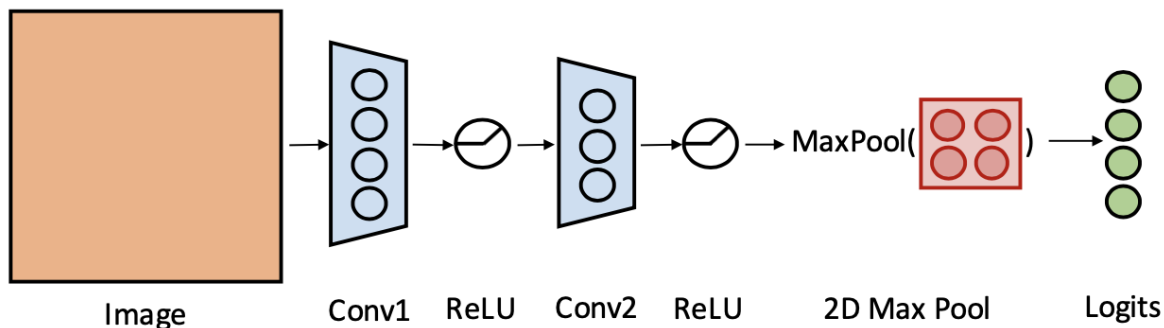


Figure 2: A simple diagram of the CNN model architecture.

(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 10. 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. Complete sections 2.1 and 3.1 of the notebook. You might use these blocks to help you debug the model, if you are not getting good development accuracy (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 section 3.1 as the learning rate for the model you submit. 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.

2.7 Implementing a Simple Convolutional NN

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 spatial regions of the input image, with weights that are shared over spatial regions. To see an animation of this behavior, see [this](#). In particular, pay attention to the first animation (no padding, no strides) and to the 9th animation (no padding, with strides). Here, a 3×3 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.

Another important notion in CNN is channels. In general, a channel is a pathway through which we transmit information, via different types of signals. It is not a concept specific to data science, it also arises in information theory, and digital communication.

Let's have a look at how the word *channel* is used in images and convolutional neural networks.

The images usually consist of 3 channels, corresponding to primary colors: red, green and blue. Thus, in reality, images are not two-dimensional objects but rather third-order tensors, characterized by a height, width, and pixel intensity in each of the three color channels. Alternatively, the channel dimension can be regarded as assigning a multidimensional representation to each pixel location.

The convolutional neural network has to adapt accordingly to deal with channels of information. We can formulate derived features in hidden layers of a CNN as channels, e.g. third-order tensors, as well. That is to say, instead of just having a single kind of hidden representation at each location, we would like to represent it in multiple different ways. Intuitively, you can imagine that some channels in CNN might be focusing on different shapes of edges, while other focusing on different types of textures, with different higher level features based on different original input channels.

You may refer to online resources for better understanding of CNN (eg. [A guide for beginners, CS230 by Stanford](#))

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 $k_2 \times k_2$ convolution that takes in an image with c channels and outputs an image with 10 channels (where k_2 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 , k_2 , c) in addition to the learning rate, optimizer, batch size, and number of training iterations. Use the Jupyter notebook 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, 14, 14, 10]. To convert this into logits, you should use a max-pooling layer (for example, [the one here](#) over the channels. This will give us a vector of 10 items, each the max value for it's respective channel in the previous output.
2. Complete sections 2.2 and 3.2 of the notebook using this model. You should use these to help you select good parameters for this model. Namely, this model will take a lot more steps to train than the previous model. 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.**

2.8 Best Model

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 to; etc. You can even go the extra mile with techniques such as data augmentation, where input images may be randomly cropped and/or translated and/or blurred and/or rotated etc. We've added the [scikit-image 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. Complete sections 2.3 and 3.3 of the notebook using this model. As always, you should use these cells to give you information about which values you should use for certain hyperparameters of your model, and to help you debug your model.
4. Save your best model as best.torch and its test predictions as best-predictions.txt. **Submit both the model file and the predictions file.**

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.

2.9 What to Submit

For this assignment you will submit the following.

1. **Analytical.** You will submit your analytical solutions to Gradescope. **Your writeup must be compiled from L^AT_EX and uploaded as a PDF.** The writeup should contain all of the answers to the analytical questions asked in the assignment. Make sure to include your name in the writeup PDF and to use the provided L^AT_EX template for your answers following the distributed template. You will submit this to the assignment called "Homework 3: Analytical".

2. **Practicum Python Notebook** Submit your Jupyter notebook to Gradescope. Create a PDF file containing your notebooks (e.g. File → Download As → PDF via LaTeX (.pdf)). Be sure you have run the entire notebook so we can see all of your output. You will submit this to the assignment called “Homework 3: Practicum”. For the loss and accuracy graphs, please make sure that the graph you are plotting for each cell is the graph of the metrics for the model that you are submitting.
3. **Programming Code** For the programming part of the assignment you will submit your code (.py files), models (.torch files), and test predictions (predictions.txt files) as a zip file.

Your .torch models must be named ff.torch, cnn.torch, best.torch. When the code loads a saved torch model, it is expecting the respective class to be unchanged, so please make sure that the version of code you submit contains the correct version of the class for each model, so that each model file can be loaded without error. You can test this by running main.py in test mode on each of the models.

The prediction files should similarly be named ff-predictions.txt, cnn-predictions.txt, best-predictions.txt.

Your code must be uploaded as code.zip with your code, models, and predictions in the root directory. By ‘in the root directory,’ we mean that the zip should contain *.py at the root (./*.py) and not in any sort of substructure (for example hw1/*.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., hw1):

```
zip code.zip *.py *.torch *-predictions.txt
```

2.10 Questions?

Remember to submit questions about the assignment to the appropriate group on Piazza: piazza.com/jhu/fall2021/cs601475675.