MASTER PLAN

1 Introduction

In this repository, we are developing code to solve a practice image classification problem that can be found at `https://datahack.analyticsvidhya.com/contest/practice-problem-identify-the-apparels`. Here, we document our data, our model, our code, our repository, and our plans for testing and for tuning parameters.

1.1 What this Document Still Needs

This document is a work in progress. It needs quite a bit of editing and many more hyperlinks, and there are sections that have not been completed. We have not gone over the dense neural network model at all, nor have we discussed all of the parameters of the convolutional network. We also should switch to using bullet-pointed lists to discuss parameters rather than doing so in pure paragraph form, as we are doing at the moment. My attempts at visualizing the networks as graphs had to be postponed because of a prolonged and mostly unproductive fight with bugs in `keras.utils.plot_model()`.

I wrote a script (`docs/scripts/find_docstrings.py`) that extracts docstrings from all the functions in the `code/` directory, and the next project on that front is write another script to convert that documentation into a format that will look good in a LaTeX document. I hope to be able to use such scripts in this project and future projects to produce pdf documentation of my code that updates itself automatically.

The biggest missing piece is the unwritten section on testing strategy. While we touch on that when we discuss the parameters of our image processing and model design functions in §3.2 and §4.1.2, we need a fully organized plan for testing and tuning. The plan will be stored here, while documentation of the actual test runs will be in a separate pdf.

Beyond that, I want to add a section on optimizers. There are some facts that I gleaned from papers on that subject and a few others, so I also plan to add a bibliography. Glossaries are also generally useful, and we will add one.

2 Data

We begin by discussing our data. For now, we only use the 60000 training examples in the `train/` directory. Note that you must extract `train_LbELtWX.zip` to obtain these examples, as the `train/` directory is deliberately omitted from the repository for performance reasons. These examples are images of 10 classes of apparel, with 6000 examples of each image. Each image is 28×28 and grayscale. We plot some samples of each class in Figure 1.

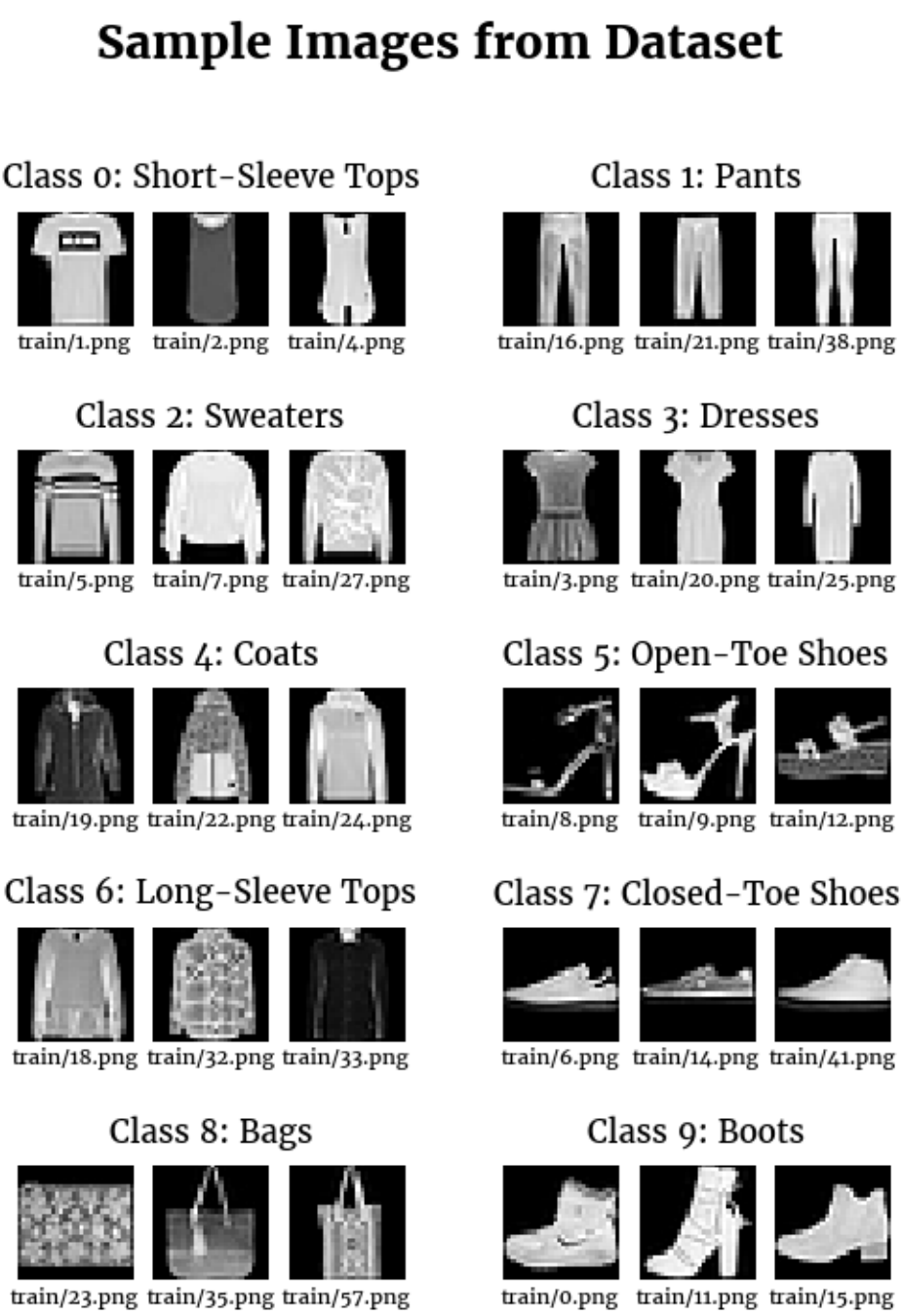

Figure 1: We inspect 3 samples from each class of apparel, with predefined class numbers, class names as interpreted by the author, and source files.

2.1 Basic Analysis

The dataset here is artificially clean: all the images are centered and upright, with nothing in the background. This should make it easier for our models to classify the data. It also means that some of the advantages of convolutional neural networks over dense neural networks may disappear; however, I still expect that a convolutional neural network will outperform a dense neural network, a belief that has been substantiated by the (undocumented) tests I have run up to this point. Nonetheless, I am training dense neural networks in addition to convolutional networks. I am not doing this because I expect great performance; rather, when I solve computer vision problems with "messier" data in the future, I would like to test the hypothesis that dense neural networks will be relatively less effective than in the case of this rather neat dataset.

2.2 Loading the Data

The module `load_data` handles the process of importing data from the `train/` directory. It uses `matplotlib.pyplot.imread()` to load the data, only storing one channel in memory, as the images are grayscale. This will also make it easier to use the GitHub repository. In the future, we can make data loading faster by storing images in grayscale on the drive, preferably in one file; in the dataset, they came as RGB images with three identical channels. When we split the data into training and cross-validation sets in our training scripts, we always use 60% of the images for training and a random seed of `1`. Using the same random seed across all tests is critically important, as otherwise there would be no real distinction between our training and cross-validation data.

2.3 Subdivision of the Classes

We can see that our classes can be divided into several subgroups that are similar to each other. For example, sweaters and coats may be hard to distinguish, as may boots and close-toed shoes, but footwear can likely be distinguished from outerwear rather easily. While we can most likely get good performance without considering this aspect of our dataset, it may be that there is a way to exploit this concept to save computation time or improve accuracy.

I have not implemented or tested this idea as of yet, and I have no idea whether it would really work, but given that we have a small number of classes, including several subgroups of classes that are similar to each other, I wonder whether the optimal solution may not consist of a single 10-class neural network. Perhaps we could improve performance by first separating into groups of classes (perhaps superclass 0: footwear, superclass 1: tops and dresses, superclass 2: everything else), and then using separate second-level networks to break those apart into the true classes. Potential advantages include (a) the ability to tune the second-level networks independently of each other, and (b) wasting less computation time

on relatively trivial distinctions. Potential disadvantages include (i) wasting computation time because the second-level neural networks may recompute features that the first-level network already could have identified, and (ii) overfitting, as the ability to independently tune parameters could make our overall model more complex; (ii) may be partly mitigated by the ability to independently tune image preprocessing and regularization.

3 Image Preprocessing

Dataset augmentation can be a powerful tool in image classification. By adding random translations, rotations, and reflections to a dataset, we make it harder for our model to exactly match the training data. However, the use of preprocessing must be tailored to the particular application; performing reflections on a set of handwritten digits would be harmful, for example. Performing vertical reflections on our set of upright images would be similarly unhelpful.

3.1 Overview

Small translations are most likely safe, and they may well be effective, as in AlexNet. It is less clear whether small rotations or horizontal reflections would be desirable. We will discard vertical reflections out of hand.

My knowledge of more sophisticated image processing techniques is limited, and I don't want to make uninformed guesses or play around with things I don't understand. However, as I progress with this project, I will also be reading up on any other methods of dataset augmentation and dimensionality reduction, and if I find something useful, it is possible that I could make my preprocessing more sophisticated.

One important thing to note about our function is that we return 28×28 images. This choice is not trivial, and we may be wrong. AlexNet also performed random translations of image data, and it cropped the images. However, in comparison with AlexNet, we have the advantage that all our images are on a black background, while AlexNet was working with images whose backgrounds could not reasonably be extended. In order to ensure that we don't "learn" from whatever noise may be present in the black background, I provided a boolean parameter which, when `True`, adjusts very small pixel values to zero, with a smooth transition to the general procedure of not adjusting values at all. It is quite likely that this is entirely unnecessary, and I may eliminate this parameter from my model in early testing.

3.2 Parameters

In `augmentation.py`, we write functions for performing random transformations on a single image, and for augmenting a dataset in memory. At the moment, there are 5 parameters

of `augmentation.randomize_image()` that are also hyperparameters of our model. The first is `p_same`, which is the probability that the function simply returns the original image unaltered. I included this parameter as a means of combating underfitting. I have observed that extreme underfitting can occur with certain values of other parameters, and one simple way to counteract this is to reduce the effective weight of randomly transformed training examples by increasing `p_same`.

Two other parameters are `p_rotate` and `p_flip`, which are the probability of performing a random rotation and a horizontal reflection, respectively. It is quite likely that it is better to reduce the angle of random rotations than to randomly shut them off, in which case perhaps the optimal value of `p_rotate` would be 1. It is also possible that random rotations are not helpful at all, in which case the optimal value is 0. For `p_flip`, we will most likely start by comparing small values of `p_flip` to `p_flip=0`. If `p_flip=0` consistently outperforms small positive values of `p_flip`, we will conclude that horizontal flips are not useful, and eliminate them from future consideration.

The last two parameters are `max_angle` and `max_shift`. `max_angle` specifies the maximum angle, in degrees, that a random rotation can have, while `max_shift` determines the maximum number of pixels by which we may translate the image vertically or horizontally. For rotations, we use a uniform distribution over the ranged (`−max_angle, max_angle`), but it could also be interesting to consider a Gaussian distribution. For translations, we use a uniform distribution over

$$\{(m,n), \mid m,n \in \mathbb{Z} \text{ and } |m|,|n| < \texttt{max\_shift}\}.$$

While other distributions could be considered for translations, I don't know that they would provide a significant advantage over the alternative of altering `p_same`, especially given the small number of possible translations for the values of `max_shift` that are likely to be effective for our rather 28×28 images.

4 Model Design

4.1 Convolutional Network

Our convolutional neural network follows the basic premise of using convolutional layers with 5×5 windows on 28×28 images, then compressing these to 14×14 images and applying convolutional layers with 3×3 windows, then compressing these further and feeding them into a shallow, dense network. We apply batch normalization and the ReLU activation function to all our convolutional layers. For dense layers, we use dropout and ReLU activation, except for in the final layer, where we use softmax.

4.1.1 Visualization

I have been attempting to produce a plots of my networks using `keras.utils.plot_model()`, but for the moment, I am forced to admit defeat. This particular Keras function is very buggy and I simply don't have time to figure out what's wrong with it at the moment. I will look into it more later this week. See `docs/images/conv_model.png` for what I'm getting right now; the softmax layer is off to the side with no edge connecting to it, and I also want to do something about the `None`s, which make perfect sense to the internal logic of Keras (representing the fact that we can use any number of training examples), but which do not look good on the diagram.

4.1.2 Parameters

We have a number of parameters to work with in `model_building.model_build_conv()`,which is the function defining our convolutional neural network. First, there is the number of each class of convolutional layer: the first series of convolutional layers has `conv_layer_count[0]` layers, while the second has `conv_layer_count[1]`. Between these series, we allow a choice between a convolutional layer with a stride of 2 or a max pooling layer, via the boolean parameter `conv_vs_pool`. For the final layer before the dense network, we allow a choice between a global average or max pooling layer using the boolean parameter `global_average_vs_max`, and if it is a max pooling layer, we allow the pool size to be customized via `final_pool_size`. `dense_layers` is a list of the sizes of each dense layer, the last of which is 10. We expect all of these variables to have substantial effects on our model, and only parameter tuning through extensive testing will reveal the optimal configuration.

4.2 Dense Network

This section has yet to be written.

5 Repository

In this section, we outline the directory structure and the purpose of the various directories, as well as the important files.

5.1 Primary Directory

5.1.1 `master_plan.pdf`

This hyperlinked document contains the master plan that will guide our work. It first explains our dataset, our image preprocessing strategy, and our model design. It contains a complete guide to parameter tuning. It also explains the directory structure of the repository.

5.1.2 `tests.pdf`

This pdf, which does not yet exist, will include documentation of my test runs and plots of training and validation classification error and loss functions for various parameters. This directory contains documentation, and the sources for said documentation. Note that we currently only train networks on at most 36000 of these 60000 images, reserving the rest for cross-validation.

5.1.3 `train_LbELtWX.zip`

This zip file contains the training examples.

5.1.4 `train.csv`

This file associates labels with each training example.

5.2 `train/`

This directory contains 60000 `.png` files, each of which is a training example. It does not exist until you unzip `train_LbELtWX.zip`.

5.3 `code/`

This directory contains the core code that defines functions for loading data, image preprocessing, building Keras models, and scripts for running Keras models.

5.3.1 `augmentation.py`

This file is for image preprocessing and dataset augmentation. It contains the functions `randomize_image()` and `augment_dataset()`.

5.3.2 `model_building.py`

This file is for building Keras models for neural networks. It contains the funcions `model_build_conv()` and `model_build_dense()`.

5.3.3 `train_conv_nn_augment_in_memory.py`

This script is currently just a basic test script that trains a convolutional neural network. Its primary function is to ensure there are no obvious bugs in the code. When we move on to doing serious tests, we will revamp this file. In the future, it will be renamed `train_conv_nn.py`, as it will include functionality for choosing between dataset augmentation in memory and on the fly.

5.3.4 `train_dense_nn_augment_in_memory.py`

This script is completely analogous to the convolutional neural network script above, except that it is for dense neural networks.

5.4 `docs/`

5.4.1 Compiling LaTeX Sources

This subsection is a stub. I still need to determine the minimal dependencies for compiling the LaTeX sources to create `master_plan.pdf` and `tests.pdf`. For now, note that I use XeTeX rather than the basic `pdflatex` command. Also note that there are a large number of unnecessary LaTeX packages loaded to create this document; I just copied the preamble from my thesis. I hope to prune the list in the future.

5.4.2 `/docs/images/`

This directory contains the images used in master_plan.pdf.

5.4.3 `/docs/scripts/`

This directory contains the Python scripts used for generating figures and other components of the documentation. They have no other effects, and are not used in training.

5.4.4 `/docs/docstrings`

This directory contains documentation associated with each function. This documentation is automatically extracted from all functions in the `code/` directory. There is a directory for each module and a file for each function within that module.