PAC Research Sample - Detecting the Use of Hand Hygiene Dispensers

Overview: I used a Convolutional Neural Network (CNN) approach for the detection of dispensers' usage. For certain sensors I trained individual, specific models from the subset of data from that sensor. For the rest, I trained a general model on all the data. This decision is to maximize sensor accuracy; because sensors are stationary at one location, there are common topographical features near the sensor that a specific model can better fit to. My intuition was that a general model may have a harder time learning all the nuances of different sensors, including different dispenser locations, walls, objects in the scene, directions people commonly approach from, etc.

Training was done on a Tesla K80 NVIDIA GPU with 12 GB, 0.8755 GHz, DDR5 memory, hosted on FloydHub. I bought 20 hours of GPU time on their platform for training, of which I used 18 hours and 28 minutes by the time I was done. The development and testing of my training/inference scripts and different model architectures was first done locally on my MacBook Air, with a 1.4GHz Intel Core i5 processor and 4GB, 1600 MHz, DDR3 memory. Almost all models achieved 94%+ binary accuracy on respective validation sets. Further results are discussed under "Results & Discussion", and future work on more granular metrics in "Future Work".

The public Github repository is located at https://github.com/Maninae/AIH. The README describes the directory organization. The two main scripts for training and inference are at "training.py" and "inference.py" under the repo root. I've built in some prompts for loading a saved model or indicating data path, but if there's ever trouble with the script regarding this, one can simply go into the code and change the prompt lines into hardcoded paths/names and run as usual.

Data: The distribution of sensor data was unequal in many ways: some sensors had ~15K samples, whereas others only had a couple dozen. The ratio for positive to negative samples for every sensor is small, but the disparity ranged greatly, and some sensors had no positive samples at all. I defined a sensor as having sufficient data if and only if 1) the positive to negative sample ratio was greater than 5% (fortunately all these sensors had a couple thousand samples). The sensors that had sufficient data were: 02, 04, 06, 08, 23, 52, 62, 63, 72. The rest, imbalanced or with no positives, are: 11, 15, 39, 59, 10, 21, 22, 24. The first category had models trained specifically on their sensor data; **please test samples from these sensors with their respective model**. For the rest, we trained a general model over all the data from all sensors; all test samples from sensors without a specific model should be evaluated with the general model.

All samples (train or dev) from sufficient-data sensors were used in the training set for the general model. Training samples from imbalanced/no-positive-data sensors were used in the training set as well, so the model can learn from these sensors. Lastly, validation samples from imbalanced/no-positive-data sensors were reserved for the dev set; this is because the general model's evaluation is most important on the validation from these sensors.

Methods: The CNN model architecture:

- two layers of 16 filters of size 5x5, followed by 2x2 max pooling, swish activation
- two layers of 32 filters of size 3x3, followed by 2x2 max pooling, swish activation
- two layers of 64 filters of size 3x3, followed by 2x2 max pooling, swish activation
- two layers of 64 filters of size 3x3, followed by 2x2 max pooling, swish activation

- 32 filters of size 3x3, swish activation
- 16 filters of size 3x3, swish activation
- Flattening into 4800 units
- Fully-connected layer of 100 units, swish activation. Dropout of 0.5 during training
- Output: fully-connected layer, 1 unit, sigmoid activation

The input features were a (240, 320, 1) depth map, with last axis as a channel for convolution. L2 regularization with $\lambda = 0.0001$ was used on all layers with trainable parameters. Dropout was employed at the penultimate dense layer for regularization also. The Swish activation unit I used was the function $f(x) = x * \text{sigmoid}(x)$; this activation was shown to propagate gradients and accelerate training for deeper models (you can google for the paper). The final loss was a binary cross entropy loss.

I experimented with Adam optimization and SGD with momentum + Nesterov, but these tweaks did not noticeably improve training and actually caused strange dips and peaks in loss - I hypothesize in the former case this is because of Adam's sensitivity to initial gradients. I settled on a traditional SGD optimizer with $\alpha = 0.001$ and a decay of 10^-5 over each update. Furthermore, the loss for positive and negative samples were weighted differently for each model depending on the 0/1 distribution in its data. This was so that we can regularize the model to not bias itself toward the over-represented 0's. The class weighting is displayed in the next section.

Results & Discussion: This chart shows the **validation metrics** results for each model I trained; unfortunately I forgot to record down the training metrics. However, I have graphs for training loss and accuracy are under the Github repo, at "assets/figures/<sensor id>". Please look at them for more info. For brevity, metrics below are on **a scale of ten-thousandths**; that is, 9832 indicates a value of 0.9832.

Sensor:	# 02	# 04	# 06	# 08	# 23	# 52	# 62	# 63	# 72	general
val. loss	1399	1245	2357	0579	2276	2290	2071	2287	2294	1140
val. acc.	9911	9933	9448	9820	9517	9540	9689	9531	9479	9955
1s weight	10	10	1.7	6	3	5	5	2	2.2	7

I used precision and recall during training to gauge the progress of my model carefully, but unfortunately the implementations I had (taken from a Keras user) are known to be inaccurate at an epoch scale, so I did not include them here. Precision was roughly 0.8+ and recall 0.95+ in in general for most models.

Future Work: The use of BatchNorm, histogram of gradients (HOG) features, and a globally connected DenseNet-like architecture are some extensions I'd like to explore, if there had been more time. The model was performing well enough, and I was constrained by GPU hours.

I suspect data augmentation to augment the 1's in all sensors' data as well as to augment sensors with less data overall would have been helpful, for example by horizontally and vertically reflecting the depth maps. If I had done a deep dive on areas the model had trouble with (e.g certain sensors, patterns that cause misclassification, times of day lacking in samples), this data augmentation could have been tailored to those areas too.