Jumio Coding Challenge - Report

# Code Structure - Requirements

The code consists of three directories under its main directory: “data”, “data\_handling” and “model”.

“data” simply contains the .csv file; which has the MNIST samples with a comma separated format. “data\_handling” contains the “mnist\_dataset.py” file, which has the MnistDataset class. MnistDataset class has the required loading operation (“load\_dataset”) which reads the .csv file into a Pandas Dataframe and converting it into Numpy arrays. Training, test and optionally validation sets can be formed in this method, by randomly shuffling the read data. MnistDataset class also provides a Python Iterator mechanism, which allows the user to get minibatches of data. After a dataset is completely read by \_\_next\_\_() calls, the code signals the end of the current epoch and reshuffles the data for the next pass. The class also provides methods to get the label count in the dataset and to visualize a given sample, using the matplotlib library.

“model” folder contains the “model.py” file, which contains the “LeNetModel” class. This class is the main building block of this solution. It implements a simple LeNet type of Convolutional Neural Network (CNN) by using the Tensorflow library. The “build\_model” call will build a LeNet CNN using the appropriate Tensorflow layer calls, sets up the loss and other properties of the neural network. “train\_model()” executes the training procedure, the model will run on the training set for a given epoch count, using Stochastic Gradient Descent based numerical optimization. If the required flag is set, the trained model is also saved by using the Tensorflow Save&Restore system, by building a checkpoint for the model variables. The checkpoint files are saved into the folder “checkpoint” under the “model” folder. Note that concurrent training runs with model saving enabled will result in the previous checkpoint files being overwritten with the new ones. “evaluate\_model()” calculates the overall multiclass accuracy on the given data set. “apply\_decision\_strategy()” applies the decision algorithm we have implemented, using the posterior outputs of the CNN on a given dataset and their corresponding labels. “analyze\_confusion\_matrix()” contains the logic for converting a multiclass confusion matrix into a binary one. It calculates the True Positive Rate (TPR) and the False Positive Rate (FPR) by using this binary confusion information. Finally, “save\_model()” and “load\_model()” methods are calls to be used to save the model variables into a Tensorflow checkpoint and load an existing one, respectively.

“global\_params.py” contains a static class with constants, which adjusts the training and evaluation settings for the algorithm. “main.py” is the entry code; it will ask the user to whether to train the model from scratch or to use a pretrained one. The model which produces the reported results in this document is provided with code. After training and after loading the pretrained version, it will evaluate the decision algorithm on the determined test set and report the results: The overall accuracy, per class TPR, FPR values and the overall coverage.

The code uses numpy, matplotlib and pandas libraries. For the CNN, it requires that the Tensorflow library is installed on the running computer. Since low-level, basic Tensorflow features are used, any version of the Tensorflow would work. In order to run the code, just traverse into the directory which are extracted out of the provided .rar file. The code can be run from the command line, simply by calling “python main.py” command. The code is written and tested by using Python 3.6, under Windows. Note that Tensorflow must be installed on the current Python environment in order to run the code. An Anaconda environment which contains a valid Tensorflow installation can be also used to run the code (I developed it using an Anaconda environment with Tensorflow).

# The ALGORITHM

In order to classify the given MNIST dataset, we used a LeNet type of CNN, which is both simple and can reach satisfactory results on that dataset [1]. We use a test set with exactly 1000 samples per class. The LeNet architecture we used contains two convolutional layers with 5x5 kernels and 1x1 strides. Layers have 20 and 50 feature maps, respectively. After each convolution, a ReLU nonlinearity is used as the activation. After the ReLU, a max-pooling layer with a 2x2 kernel and 2x2 strides is used. It effectively halves the width and height of the incoming feature maps. After these two layers, a fully connected layer follows. It also uses the ReLU activation. We use a 32 dimensional layer here. This layer is linearly projected into a 10-dimensional layer, which are used as the logits and input into the softmax layer. A common cross entropy loss is used as the classification loss. In order to regularize the network and increase generalization, we used a Dropout layer after the first fully connected layer. Dropout layer effectively samples different CNN architectures during each SGD pass. During the evaluation phase, the fully connected layer is automatically scaled by the Tensorflow such that the expected magnitude of the activations during the evaluation and training are the same. Dropout approximates an ensemble of CNNs with this provided structure and improves the generalization. We used a keep probability of 0.5. This probability is an important hyperparameter for deep neural networks in general and it is usually optimized by using a validation set. Due to time restrictions, we set this as 0.5, without optimizing it further.

In order to train the CNN, we use a SGD optimizer with momentum. We use 200 epochs on the training set, with a minibatch size of 125. The learning rate starts with 0.01 and it is halved at each 12800. Iteration; after each quarter of the total training iterations.

After training the CNN, we apply a decision strategy on the test set, in a similar but not exactly equal fashion to [1], which introduces a rejection option, turning the classifier effectively into a selective one. The idea is built on the “Error-Reject Tradeoff” theory, which is proposed by Chow [2]. This is theory is built on the ideal Bayes Classifier, which has access to the complete information about the data distribution and consequently to the marginal distribution , the conditional distribution and the class distribution . According to this theory, the ideal Bayes classifier chooses the classification

We then define the confidence of the Bayes Classifier with respect to the sample x as:

is the occurrence probability of the most confident estimate of the Bayes Classifier for the sample x. is the optimal selection of the ideal Bayes classifier, minimizing the expected error on the whole data. Then is defined, such that is the probability of misclassifying the sample x.

Chow [1] builds a rejection option into the Bayes Classifiers, such that whenever , the class is accepted and whenever , it is rejected. The expected error rate of the ideal Bayes classifier is given as:

In an equivalent fashion, a rejection threshold t’ can be defined on , such that:

This means when the optimal Bayes classifier has a confidence greater than t’ for a sample, it is accepted for classification. Using the t version of the threshold, Chow defines the rejection rate and error rate in the terms of t, as:

Where [.] is indicator function, giving 1 when the input is positive and 0 else [3]. The two important results which are derived from these terms, are:

* and are monotonic in t.
* For small changes, .

These results can be interpreted such that as t is decreased, more samples are rejected but the error rate diminishes in the same time (not always the same amount per infinitesimal , but same per infinitesimal ).

In our algorithm, we do not have access to the actual data distribution . We have only a limited training set sampled from this distribution. But in order to build at least an approximate rejection mechanism like the above, we note that we only need to access the conditional distribution, or class posterior . A good discriminative algorithm which approximates this posterior distribution well enough may be a good start. Due to their universal approximation property, which means approximating a well behaving function arbitrarily close, given enough number of hidden units, neural networks are well suited candidates for approximating this class posterior . Motivated by that, we devise the following simple algorithm:

|  |
| --- |
| 1. Train a neural network (LeNet type CNN) with the training set . 2. Take a test set, , calculate the posteriors . 3. Calculate the estimates and for each sample. 4. Sort , in the ascending order. (Smallest comes as the first, , w.l.o.g.). 5. Calculate the confusion matrix, , using no rejection threshold whatsoever. 6. Set as the step size. 7. while :   for each in do { [[}    Calculate TPR, FPR and Coverage on  If for each class FPR <= 0.015%:  return |

Ideally, if we had a perfect Bayes classifier, as increased, we would expect that the error gets reduced monotonically and could apply a binary search on the test or validation set to find the optimal cutoff, with a run time of . But since this is not possible due to our approximation of the class distribution with a CNN, errors are not guaranteed to decrease monotonically. We conduct a linear search starting from the smallest to the largest. In the case of very large number of samples, we can apply a step size to skip subsets of the dataset in order to reduce the computation time. We use =1 for the sake of completeness. Note that since we have sorted the posterior estimates, , for every increase in the rejection threshold , we only need to decrease the entries corresponding to the samples from the confusion matrix , which is a constant time operation. This prevents the re-calculation of the confusion matrix in every iteration with respect to the non-rejected samples. With this small trick, the worst case run time of this algorithm becomes instead of . (is the test set size).

Another thing we need to give attention is that when training deep convolutional neural networks, it is a common thing that the last softmax layer produces “too confident” outputs. A significant proportion of the samples get “1.0” as (the most confident estimate), and the other probabilities get values assigned that are very close to 0, which are mostly due to numerical instability. When this is the case, the ordering of “ can converge very quickly to 1. Moreover, we can lose the “nuances” between the confidences of different samples, since a large portion of them will readily be 1 (exactly confident). The differences in the logits (pre-softmax layers) across different samples get “swallowed” by the softmax nonlinearity. However, these relative differences between class activations convey valuable discriminative information. For example, if a sample of “3” is given probability for being “2” and probability for being “7”, it can be interpreted that it is semantically more similar to “3” compared to “7”. However very large logits resulting in “1.0” as the most confident probability for “3”, eradicates this information.

The reason for this problem is that when a deep neural network is fairly confident for a sample, the subset of the activations connected to the most confident logit fire with values of large magnitudes. These activations are continuously accumulated through the layers of the network. This problem is more evident when ReLU type of nonlinearities are used, which do not squash activations like sigmoids. (Which is intended for facilitating the vanishing gradients problem to begin with). This problem is of not great importance when the aim is merely getting high classification accuracies. But when one should use the network’s softmax outputs as confidence values, it becomes damaging. To reduce this effect, we use the approach in the “Knowledge Distillation” technique [4] and divide each logit by a temperature value both during the training and evaluation. Our softmax layer becomes:

As , , which dampens the effect of very large logits ( is the class count). A side effect of this modification is that the gradients gets scaled with , resulting in slower converge of the network. In the reported results, we used a temperature value of .

# The RESULTS

The CNN we trained on the training set gives an overall multiclass accuracy of 99.09% on the test set, without any rejection mechanism. The TPR and FPR values, per class, are given as:

|  |
| --- |
| (Class:0 TPR:99.5% FPR:0.077778%)  (Class:1 TPR:99.8% FPR:0.1%)  (Class:2 TPR:98.5% FPR:0.066667%)  (Class:3 TPR:99.5% FPR:0.066667%)  (Class:4 TPR:99.0% FPR:0.1%)  (Class:5 TPR:98.5% FPR:0.088889%)  (Class:6 TPR:99.7% FPR:0.14444%)  (Class:7 TPR:99.0% FPR:0.11111%)  (Class:8 TPR:99.0% FPR:0.13333%)  (Class:9 TPR:98.4% FPR:0.12222%) |

Then we apply the decision algorithm we have described above to find an optimal cutoff threshold on the test set. The selective classifier with the reject option gives now the following results:

|  |
| --- |
| Overall Accuracy:99.96769330174456%  Coverage:92.86%  (Class:0 TPR:100.0% FPR:0.012016%)  (Class:1 TPR:100.0% FPR:0.0%)  (Class:2 TPR:100.0% FPR:0.0%)  (Class:3 TPR:100.0% FPR:0.0%)  (Class:4 TPR:99.784% FPR:0.0%)  (Class:5 TPR:100.0% FPR:0.0%)  (Class:6 TPR:100.0% FPR:0.011989%)  (Class:7 TPR:100.0% FPR:0.0%)  (Class:8 TPR:100.0% FPR:0.0%)  (Class:9 TPR:99.887% FPR:0.011902%) |

Some Important Notes About the Implementation:

* An important thing to give attention is that these results are obtained using the training/test split with a Numpy random number seed of 42, as can be found in global\_params.py (Using this seed should replicate these results across different platforms according to the Numpy documentation).
* RANDOM\_SEED = 42
* The implementation of the provided code took approximately 3 hours and 45 minutes. This amount of time does not include model training, which approximately took 20 minutes on a NVidia 1070 GTX GPU.
* An important portion of the MnistDataset class code has been borrowed from my PhD thesis code, where I also use MNIST as the dataset as well. I mostly changed the code for reading the data from a .csv file and for an iterator supported minibatch reading mechanism. So, that part of the code is not completely written from the scratch.
* For calculating the FPR and TPR rates, I used the reference Confusion Matrix definition at: https://www.wikiwand.com/en/Confusion\_matrix

# Possible Improvements

Since we needed to complete the challenge in a limited time period, in this section, we will state some potential improvements on the code and the methods which can be applied with a larger time budget.

* The decision strategy we have used is based on the outputs of an already trained deep neural network model. In the contemporary deep learning literature, a common trend is to define proxy model objectives besides the main objective (like a good decision strategy coupled with a good accuracy, in our case) as differentiable losses and combine them with the main objective into a unified loss, most probably by scaling either of them accordingly with a coefficient, . For a more ambitious project, we could look for ways to define the selection strategy as a differentiable loss, besides the main cross entropy loss and optimize both of them jointly or iteratively, until convergence, with the SGD. Our main aim is to reduce FPR while keeping a high amount of coverage; being able to interpret this as a differentiable loss would be the key component of such an approach.
* In this project, we did not use validation sets to optimize hyperparameters like the dropout probability, learning rate decaying regime etc. Ideally, if these were optimized on validation sets, we could attain a better coverage-error tradeoff.
* Similar to [1], we apply the decision strategy directly to the test set. For a real production code, we would not have access to the test set directly and the optimal posterior probability which provides the aimed per class FPR values on a held-out set may not guarantee these bounds exactly on the real production data the model will see. To handle that, some interactive – heuristic mechanisms could be designed, which tweak the posterior probability threshold as the model provides estimates on the real data. These mechanisms could work periodically or on the fly.

# REFERENCES

[1] Le Cun, Boser, Denker, Henderson, Howard, Hubbard and Jackel, “Handwritten Digit Recognition with a Back-Propagation Network”, NIPS, 1989.

[2] C.K. Chow, “On Optimum Recognition Error and Reject Tradeoff “ IEEE Transactions on Information Theory IT-16, 41 (1970).

[3] Hansen, Liisberg, Salomon, “The Error-Reject Tradeoff”, Open Systems & Information Dynamics, 2000.

[4] Hinton, Vinyals, Dean, “Distilling the Knowledge in a Network”, https://arxiv.org/abs/1503.02531.