

CORNELL UNIVERSITY

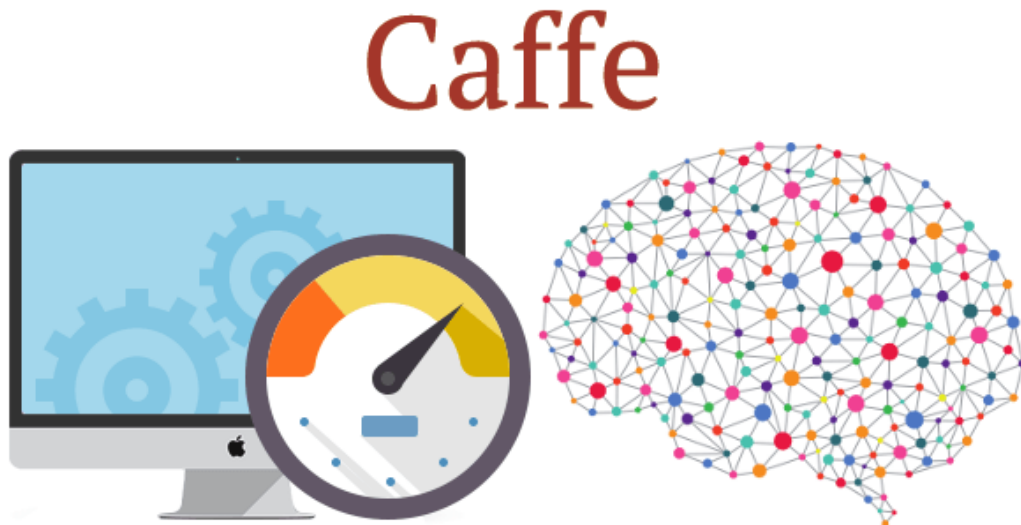
CS 4701: PRACTICUM IN ARTIFICIAL INTELLIGENCE

Food Recognition with Deep Learning

Daryl Sew (dns55)
Cornell Computer Science '17
darylsew@gmail.com

Alice Zhang (gz74)
Cornell Computer Science '17
gz74@cornell.edu

Instructor, Bart Selman (bs54)



December 13, 2016

1 Problem Description

Consider the problem of creating an artificially intelligent grocery shopper who is able to make recipe substitutions and suggestions based on availability. Such an agent would be highly desirable to anyone who would rather do something else with their time than shop for groceries, and the different intelligent components that go into this agent would have widespread applications in other fields.

Recent advances in machine learning algorithms and computing capabilities have enabled deep neural network architectures to surpass the state of the art traditional machine learning classifiers in nearly every problem domain. In particular, Convolutional Neural Networks have surpassed traditional computer vision and machine learning techniques in many tasks, provided with large amounts of data.

Inspired in part by the textbook grocery shopper example and in part by [IBM's work in recipe recommendation with Watson](#), we decided to build the perception system for food recognition, using deep learning methods in order to achieve the best possible performance. The ability to recognize a food item in an image has countless applications; for example, it is useful in building cooking robots, creating an artificially intelligent grocery shopper, automatically tracking shopper actions (i.e. Amazon Go), diet tracking programs, and many more.

2 High Level Design

2.1 Approach

Our first attempt to train our network on resulted in very poor ($< 5\%$) accuracy, so we decided to break the problem down into more manageable chunks.

We picked K Nearest Neighbors and Support Vector Machines as baseline classifiers so we would be able to meaningfully evaluate the Deep Convolutional Neural Network's performance. If we are using CNNs effectively, we should see much better results compared to the baseline. We used the LeNet network, as it is one of the most widely studied and used network architectures in the field, so we knew it would be fairly easy to work with.

We started with using the baseline classifiers to classify several types of hand drawn shapes with no feature extraction, mimicking the MNIST dataset. We simply used the grayscale pixel values of an image as the feature vector. We then trained the CNN on these, evaluated the performance, and compared the results.

After this, we tried using manual feature extraction via corner detection to improve the performance of KNN and SVMs, and tuning network parameters to improve the performance of CNNs.

Once we had completed these sanity checks, we moved on to testing with real datasets. We picked a dataset.

2.2 Software Architecture

2.2.1 potato.py

Loads model weights for a convolutional neural network and feeds forward test inputs through the network, evaluating its performance. The network we used was the off the

shelf LeNet.

2.2.2 knn.py

Evaluates the K Nearest Neighbors classifier on test inputs. Experiments with thresholding and corner detection as feature extractors. We chose to implement this ourselves as we thought it would be a valuable exercise. However we did not implement other classifiers ourselves due to time constraints.

2.2.3 svm.py

Evaluates the Support Vector Machine classifier on test inputs. Uses the Scikit-Learn multiclass SVM with a Radial Basis Function kernel, which allows the SVM to make nonlinear decision boundaries and thus classify data that is not linearly separable.

2.2.4 train_test_split.py

Splits data into train and test data, using a library function to convert it to the binary format that LeNet expects.

2.3 Data Collection Process

TODO

2.4 Logistics

Training for the SVM was done on a 2015 Macbook Pro, and took a trivial ($< 5s$) amount of time. Training for the neural network was done on an Nvidia GTX Titan. The KNN model we used does not require an extensive training period as we compute distances when requested.

2.5 Theory

2.5.1 K Nearest Neighbors

how does it work

2.5.2 Support Vector Machine

how does it work

2.5.3 Convolutional Neural Network (LeNet)

how does it work

We created the following network visualization via Caffe's `draw_net.py`. TODO split up image



2.5.4 Cross Validation

talk about train test split and why it's needed

3 Evaluation

For evaluating performance, we decided to focus on multiclass precision-recall curves, as they are a fairly holistic method of evaluating classifier performance. The area under the curve is a simple metric to optimize, and the curve itself effectively characterizes classifier performance in different situations.

We are not in a position where false positives or false negatives have different desirabilities, either, so no modification or special analyses are required.

3.1 Hand Drawn Data

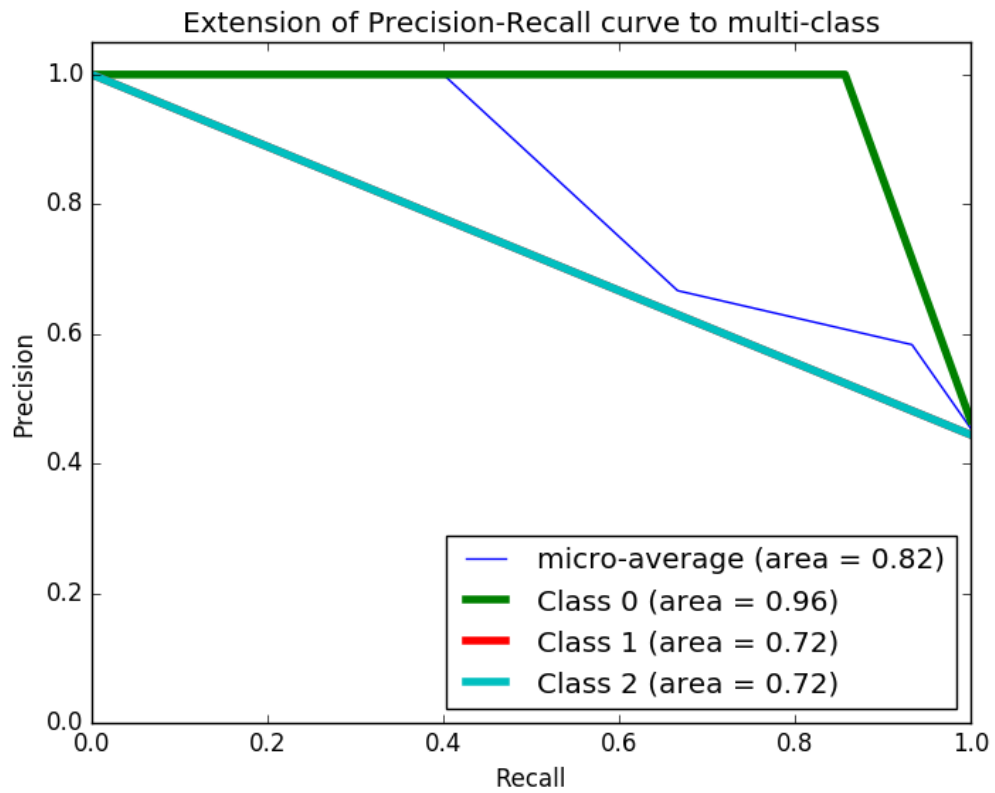
The following are sample images from the three classes of hand drawn images we evaluated. The network scales the inputs to 28x28, so we present the images after scaling below. Resizing was done via the UNIX `convert` command line tool.

The left image is a potato. The right image is a triangle. The lower image is a five point star.



3.1.1 K Nearest Neighbors

3.1.2 Support Vector Machine



(trained on 20 16x16 images; if time redo on 40 28x28 images. there seems to be a bug rn)
classes correct: [4.0, 3.0, 0.0] total: [5, 3, 7] accuracy: [0.8 1. 0.]

3.1.3 Convolutional Neural Network

3.2 Food Dataset

3.2.1 K Nearest Neighbors

3.2.2 Support Vector Machine

3.2.3 Convolutional Neural Network

4 Conclusion

5 Lessons Learned

While the end result is lots of fun to play around with and is rather impressive, there's a lot of work that goes into training a deep neural net. We never thought that we would spend so much time on dataset curation and pipelining; we thought the dataset organization and input part would be quick, and the slow part would be finding a network architecture that is performant. Although we didn't end up having much time to explore different network architectures this time, budgeting more time for data munging work next time we do a project like this would enable more exploratory data science / artificial intelligence work, so next time we might have time to have a bit more fun.

If we could do this again, we'd also try to plan out some of the software design in advance. We made separate scripts to handle different types of datasets (i.e. datasets from other people, a dataset we created ourselves). However, there were actually a lot of things in common. For example, we had a script to split the data we recorded into train and test and a separate script to convert it to MNIST, and then we had a script to split a research dataset into train and test and convert it to MNIST, and much of the functionality is shared between the two.

Additionally, the process to use a different dataset with the neural network or integrate a different dataset into a baseline classifier requires many intermediate steps, and those could be done in one step.

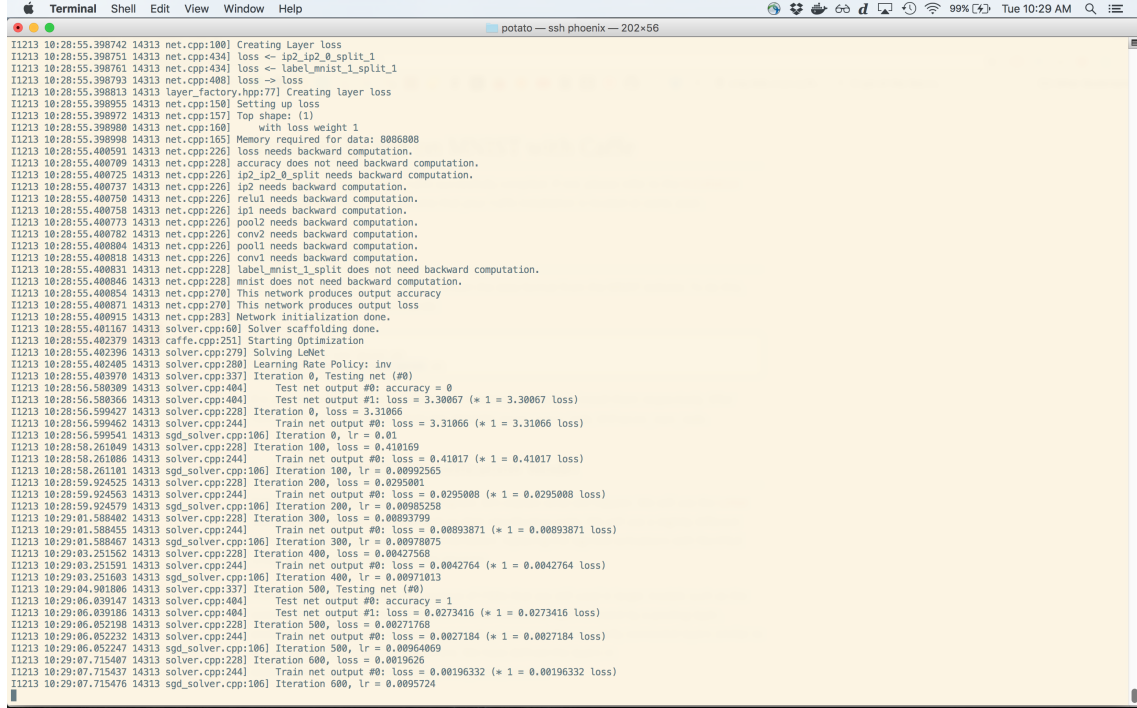
6 Future Work

We would love to try a novel network architecture and achieve better performance in the future. In particular, something that is inspired by the way the human visual system processes images of food would be interesting, and could be grounds for publishing a paper.

On the practical side of things, some future work could be creating an ensemble model of classifiers that will be able to pick out typical foods in a fridge. As far as we can tell, there does not exist a dataset of fridge items, so we would have to create a large dataset of fridge items. We could then build a knowledge representation for recipes or attempt to use machine learning to build a recipe recommendation on top of that.

7 Appendix

7.1 Screenshots



```
121213 10:28:55.398742 14313 net.cpp:1800 Creating Layer loss
121213 10:28:55.398751 14313 net.cpp:4341 loss <- ip2_ip2_0_split_1
121213 10:28:55.398761 14313 net.cpp:4341 loss <- label_mnist_1_split_1
121213 10:28:55.398793 14313 net.cpp:4080 loss -> loss
121213 10:28:55.398813 14313 layer_factory.hpp:777 Creating layer loss
121213 10:28:55.398955 14313 net.cpp:1500 Setting up loss
121213 10:28:55.398972 14313 net.cpp:1571 Top shapes: (1)
121213 10:28:55.398980 14313 net.cpp:1600 with loss weight 1
121213 10:28:55.398998 14313 net.cpp:1651 Memory required for data: 8006800
121213 10:28:55.400591 14313 net.cpp:226 loss needs backward computation.
121213 10:28:55.400769 14313 net.cpp:228 accuracy does not need backward computation.
121213 10:28:55.400725 14313 net.cpp:226 ip2_ip2_0_split needs backward computation.
121213 10:28:55.400737 14313 net.cpp:226 ip2 needs backward computation.
121213 10:28:55.400750 14313 net.cpp:226 relu needs backward computation.
121213 10:28:55.400758 14313 net.cpp:226 pool1 needs backward computation.
121213 10:28:55.400773 14313 net.cpp:226 pool2 needs backward computation.
121213 10:28:55.400782 14313 net.cpp:226 conv2 needs backward computation.
121213 10:28:55.400804 14313 net.cpp:226 pool3 needs backward computation.
121213 10:28:55.400818 14313 net.cpp:226 conv1 needs backward computation.
121213 10:28:55.400831 14313 net.cpp:228 label_mnist_1_split does not need backward computation.
121213 10:28:55.400846 14313 net.cpp:228 mnist does not need backward computation.
121213 10:28:55.400854 14313 net.cpp:270 This network produces output accuracy
121213 10:28:55.400871 14313 net.cpp:270 This network produces output loss
121213 10:28:55.400915 14313 net.cpp:283 Network initialization done.
121213 10:28:55.401167 14313 solver.cpp:60 Solver scaffolding done.
121213 10:28:55.402379 14313 caffe.cpp:251 Starting Optimization
121213 10:28:55.402396 14313 solver.cpp:279 Solving LeNet
121213 10:28:55.402405 14313 solver.cpp:280 Learning Rate Policy: inv
121213 10:28:55.403970 14313 solver.cpp:337 Iteration 0, Testing net (#0)
121213 10:28:56.580369 14313 solver.cpp:404 Test net output #0: accuracy = 0
121213 10:28:56.580366 14313 solver.cpp:404 Test net output #1: loss = 3.30067 (* 1 = 3.30067 loss)
121213 10:28:56.599427 14313 solver.cpp:228 Iteration 0, loss = 3.31066
121213 10:28:56.599462 14313 solver.cpp:244 Train net output #0: loss = 3.31066 (* 1 = 3.31066 loss)
121213 10:28:56.599541 14313 sgd_solver.cpp:106 Iteration 0, lr = 0.01
121213 10:28:58.261049 14313 solver.cpp:228 Iteration 100, loss = 0.410169
121213 10:28:58.261086 14313 solver.cpp:244 Train net output #0: loss = 0.41017 (* 1 = 0.41017 loss)
121213 10:28:58.261101 14313 sgd_solver.cpp:106 Iteration 100, lr = 0.00992565
121213 10:28:59.924525 14313 solver.cpp:228 Iteration 200, loss = 0.0295001
121213 10:28:59.924563 14313 solver.cpp:244 Train net output #0: loss = 0.0295008 (* 1 = 0.0295008 loss)
121213 10:28:59.924579 14313 sgd_solver.cpp:106 Iteration 200, lr = 0.00985258
121213 10:29:01.588462 14313 solver.cpp:228 Iteration 300, loss = 0.00893799
121213 10:29:01.588455 14313 solver.cpp:244 Train net output #0: loss = 0.00893871 (* 1 = 0.00893871 loss)
121213 10:29:01.588467 14313 sgd_solver.cpp:106 Iteration 300, lr = 0.00978075
121213 10:29:03.251562 14313 solver.cpp:228 Iteration 400, loss = 0.00427568
121213 10:29:03.251581 14313 solver.cpp:244 Train net output #0: loss = 0.0042764 (* 1 = 0.0042764 loss)
121213 10:29:03.251603 14313 sgd_solver.cpp:106 Iteration 400, lr = 0.00971813
121213 10:29:04.981806 14313 solver.cpp:337 Iteration 500, Testing net (#0)
121213 10:29:06.039147 14313 solver.cpp:404 Test net output #0: accuracy = 1
121213 10:29:06.039161 14313 solver.cpp:404 Test net output #1: loss = 0.0273416 (* 1 = 0.0273416 loss)
121213 10:29:06.052198 14313 solver.cpp:228 Iteration 500, loss = 0.00271768
121213 10:29:06.052232 14313 solver.cpp:244 Train net output #0: loss = 0.0027184 (* 1 = 0.0027184 loss)
121213 10:29:06.052247 14313 sgd_solver.cpp:106 Iteration 500, lr = 0.00964869
121213 10:29:07.715407 14313 solver.cpp:228 Iteration 600, loss = 0.0019626
121213 10:29:07.715437 14313 solver.cpp:244 Train net output #0: loss = 0.00196332 (* 1 = 0.00196332 loss)
121213 10:29:07.715476 14313 sgd_solver.cpp:106 Iteration 600, lr = 0.0095724
```

Figure 1: Training the network on a GTX Titan X. So fast!

7.2 Code

Here we present an annotated listing of all of the code we wrote for this project. The code is also available on [GitHub](#).

8 Permissions

Our code is under the Apache license; see [GitHub](#) for more details.

If you found our work helpful in your research, we would appreciate it if you could consider listing us as authors. Feel free to contact us.

9 Credits

Thanks to the Cornell Graphics and Vision lab for donating a tiny amount of GPU power for us to train our network. We did not have access to Nvidia GPUs, so we would have had to spend orders of magnitude more time training on CPUs otherwise.

Thanks to [gkielian](#) for the script we used to convert our PNG images into the custom binary format the LeNet expects.

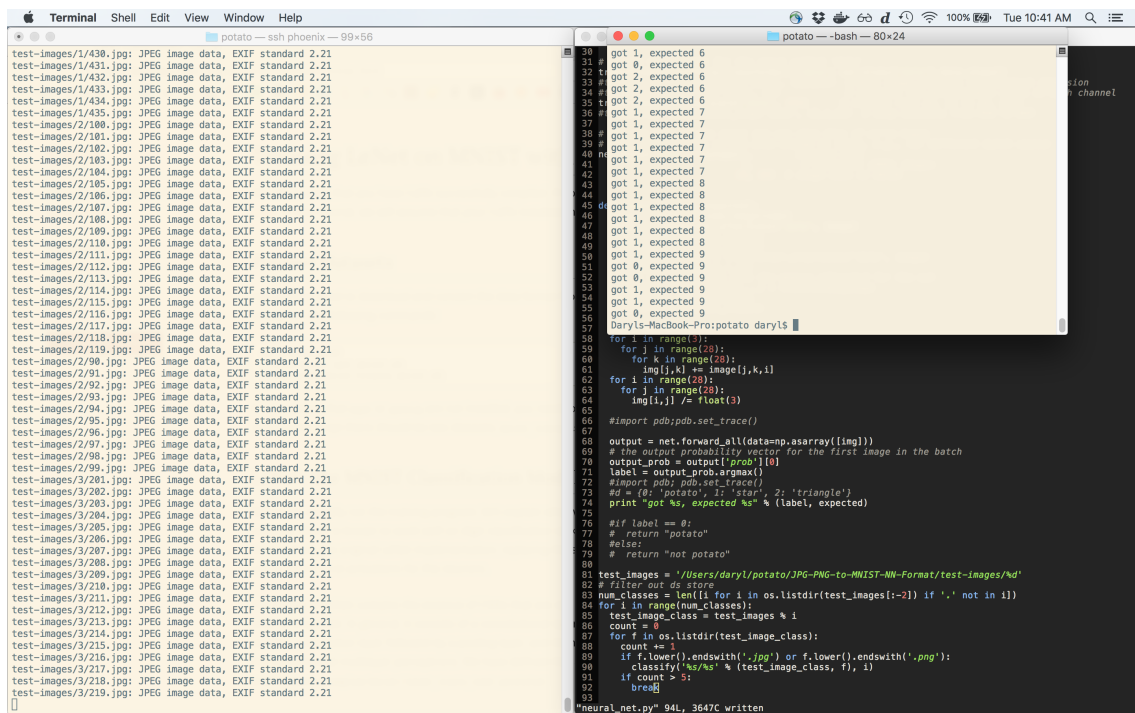


Figure 2: Testing the network while waiting for shell/Python scripts to curate dataset.

References

- [1] Jia, Yangqing, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. *Caffe: Convolutional Architecture for Fast Feature Embedding*. arXiv preprint arXiv:1408.5093, 2014.
- [2] Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E. *Scikit-learn: Machine Learning in Python*. Journal of Machine Learning Research 12.2825-2830, 2011.
- [3] Y. LeCun, L. D. Jackel, B. Boser, J. S. Denker, H. P. Graf, I. Guyon, D. Henderson, R. E. Howard, and W. Hubbard. *Handwritten digit recognition: Applications of neural net chips and automatic learning*. IEEE Communication, pages 41-46, November 1989. invited paper.
- [4] <http://www.site.uottawa.ca/~shervin/pubs/FoodRecognitionDataset-MadiMa.pdf>