Deep Learning for Face Verification

Thapanapong Rukkanchanunt
Department of Computer Science
U. of Illinois at Urbana-Champaign
Illinois, USA
rukkanc1@illinois.edu

Tom Paine
Department of Computer Science
U. of Illinois at Urbana-Champaign
Illinois, USA
paine1@illinois.edu

Abstract

Face recognition has been studied extensively over the past decade. But much of that work has been done with database created by research labs under controlled conditions. Recently large scale datasets collected from the internet are used as more challenging benchmarks for face recognition. As datasets increase in difficulty, more complicated techniques are required to achieve high accuracy for face recognition. Generally these more complicated techniques share common attributes: they use increasingly complicated hand-crafted features, complicated kernel methods that make use of many features, make use of outside data, or some combination of the above. For this project, we will apply a technique called Deep Learning on face recognition problem.

1 Introduction

Deep learning is a family of techniques that uses unsupervised learning to initialize the weights of neural network [4]. In computer vision, the most successful DL networks have utilized convolution or locally connected neural networks [6]. The choice of unsupervised techniques may not be essential for the method to work well. In fact recent work by Coates et. al. [1], suggested many methods can be used with great success, including GMM, K-means, Sparse Coding, and RBMs. In these studies, the most important aspect of the networks were the number of elements in the dictionaries learned by the unsupervised methods. In fact, it showed single unsupervised layers that were wide i.e. had dictionaries with many elements ($>$1000), could outperform skinny deep networks, which were in favor at the time [1].

Deep learning techniques recently attracted great interest in the computer vision community. Most notably when a large scale deep-learning algorithm designed by Google achieved state-of-the-art accuracy on the Image-Net challenge on a large cluster [5]. And more recently when a deep-learning algorithm submitted by Geoffrey Hinton and colleges, outperformed the rest of the computer vision community by 10% on the ECCV 2012 Pascal Challenge [8], also based on Image-Net, this time on a network trained with only two GPUs. Unlike most modern techniques, deep-learning has worked very successfully applied directly to raw image pixels. The algorithms can be interpreted as learning good features for classification. Generally the learned features can be used to train a linear SVM and achieve good classification results.

We have implemented the basic framework to train a single-layer unsupervised network. The first pass with no hyper-parameter tuning achieved an accuracy of 68%. This is notably better than the eigenfaces benchmark, and on par with an algorithm using specialized kernels on raw pixels [7]. Our method uses no outside training data. Of methods that fall into this category, the best listed result on Labed Faces in the Wild is 79%. Currently, on an 8 computer cluster, the deep learning feature extraction algorithm takes 27 hours to run.

For this project, we plan to implement these algorithms and apply them to the face recognition problem. Some work has already been done in this area, most notably by [3]. But their approach uses RBMs,

which are slower, and difficult to scale to many dictionary elements. In fact, their paper does not report the number of dictionary elements used. We hope to achieve similar results using k-means to learn the dictionary, and sparse coding to perform the encoding, a technique suggested in [2]. Furthermore, we employ several optimization technique to greatly reduce the computational time while maintaining the same accuracy rate. This is due to the time constraint for this project.

2 Face Recognition

2.1 Problem Statement

The problem that we are trying to solve is called face recognition, one of classic computer vision problems. We are given a picture of a face, and we want to decide which person from among a set of people the picture represents, if any.

2.2 Motivation

Face recognition has many applications varying from a typical security identification to face tagging currently developed by Facebook and Google. In fact, human's brain spends time on object recognition more than logical thinking. In the past, many studies focused on improving a technique on recognizing frontal aligned faces. However, this cannot be applied in real world problem directly. Recently, there are several face databases that use real world images such as photographs or video recording. In our project, we will use PubFig83 which is a face database consisting of famous celebrities photographs. Most of faces is visible but the face itself is not well aligned. This makes the recognition problem harder but at the same time, this will test the robustness of the underlying algorithm.

3 Feature Extraction

Feature extraction is composed of two parts. The first part is learning dictionary while the second part is encoding. This is the first layer of deep learning.

3.1 K-Mean Clustering

K-Mean clustering partitions the data into k clusters in which each point lies on the cluster with the nearest mean. The algorithm is iterative. For each iteration, assign each point to the nearest cluster and recomputing the mean of the each cluster based on the new assignment. The distance function used in our project is euclidean distance. This algorithm converges when the assignment does not change. However, it may take a long time to converge so we set the upper bound of the iteration to 400 iterations.

3.2 Dictionary

A dictionary is essentially a collection of filters. Instead of providing pre-defined filters, we will learn the filters from the data. The algorithm is as follow:

ComputeDictionary
1. Apply local contrast normalization (LCN) to each image
2. Extract p random sample patches
3. Perform whitening on each sample patch
4. Use K-mean to obtain q mean patches

There are two hyperparameters that we can play with. Initially, we set p small (400,000) to reduce the computational time. However, we increase the value to 1,000,000 due to GPU speed-up. For the parameter q, we would like to have as many as different mean patches so we settle for a moderate number which is 1,600. In fact, any value above 1,000 will give us wide range of filters. (Recall that we are dealing with color images so the number of different color filters will be much more higher than the gray-scaled counterpart.) Figure 1 shows the computed dictionary of 1,600 patches.

3.3 Encoding

All encoding methods take as input a three channel (i.e., RGB) image, and a dictionary of filters. The output is vector of pooled activities. The computational steps are outlined in Figure 2.

We tried many variations of this general outline, specifically we tried with and without Local Contrast Normalization (LCN); three different 'sparse filtering' techniques: CPU-based sparse coding, CPU-based convolution, and GPU-based convolution; and two pooling techniques: mean pooling, and max pooling.

LCN is a local operation. For each pixel the mean and standard deviation of the pixels neighborhood is calculated and used to normalize that pixel value. The resulting images looks like a high-pass filtered version of the original color image.

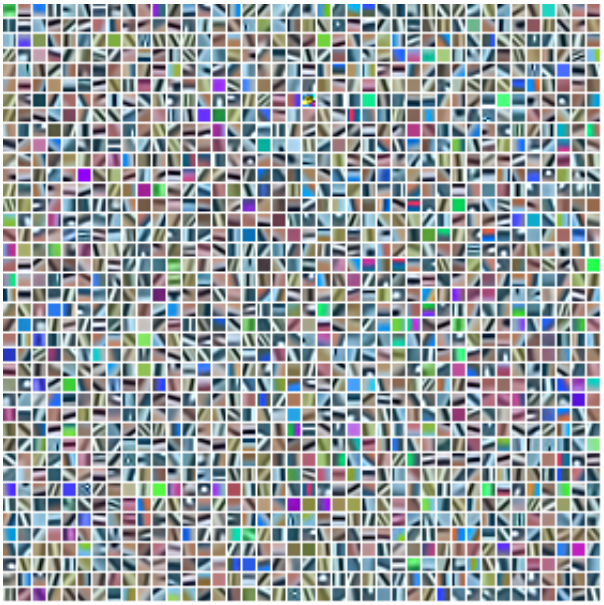

Figure 1: A Dictionary

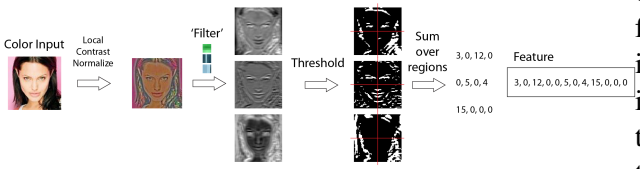

Figure 2: Encoding process

Each filtering operation takes as input a three channel image, and gives as output another multi-channel image, where each channel corresponds to the filtering operation from one filter in the dictionary. So if the original image is $N \times N \times 3$, and we learn a dictionary with K, $M \times M \times 3$ filters, the output is of size $(N-M+1) \times (N-M+1) \times K$.

In CPU-based sparse coding, for each image, we gather all non-overlapping $M \times M$ patches in the image (there are $(N-M+1) \times (N-M+1)$ of them), and use a sparse coding algorithm to find the sparse code which minimizes reconstruction error giving the dictionary of filters, each patch results in a sparse code vector of size K (giving our $(N-M+1) \times (N-M+1) \times K$ output). While this process is relatively fast, it took about 1 second per image with our hardware. For a database with 13,838 images it would take 9.6 days on a single machine. We ran this version of the code once on an 8 computer cluster and it took 27 hours. Without fine-tuning it did not produce very good results (about 68% accuracy on LFW).

In CPU-based convolution the operation is much simpler. We simply convolve the image with each filter, and threshold the result. Filtering RGB images with RGB images may require further explanation: we convolved each channel of the image, with each channel of a filter, and summed up the result. This is equivalent to using the convn operation in MATLAB, which for the CPU, was slower than looping over each channel, and using conv2. In our experiments a single global threshold value was used, which we tuned as a hyper parameter. Running this version took 12 hours for all 13,838 images.

In GPU-based convolution, the same operation is being performed, but we experimented with many ways to speed up the operation instead of looping over channels, filters, and images. We describe our fastest method here. GPUs perform best when maximizing the computations performed while minimizing the data sent, and especially data received from the GPU. In our method we only loop over the filters; channels and images are handled by using the convn function in MATLAB. For a single $M \times M \times 3$ filter, we convolve it with a stack of 2500 images reshaped to size $N \times N \times (3 \times 2500)$. The output is size $(N-M+1) \times (N-M+1) \times (3 \times 2500 - 3 + 1)$. If we drop every third frame, this is equivalent to looping over channels and images, but on a GPU, is much faster. Also, to minimize the data recieved from the GPU, we perform the pooling step, reducing the size of the data, before pulling it back to the CPU. Running this version takes 25 minutes for all 13,838 images.

Pooling is necessary to reduce the dimentionality of the output $(N-M+1) \times (N-M+1) \times K$ is generally too big to give to a classification algorithm. Pooling also has other benefits including allowing features to be translation invariant. In pooling we simply perform an operation over local regions of each channel. Here we pool over the whole image, resulting in a vector of size K, and pool over each quadrant of the image resulting in a vector of size $4 \times K$, we concatenate both of these vectors to create a final feature vector of size $5 \times K$.

3.4 Learning Classifier

We use LIBSVM classifier coded in MATLAB for when the number of training samples and number of classes are small, and LIBLINEAR, when the number of training samples and number of classes are high. Specifically we found using LIBLINEAR for the 83 class recognition problem with its large number of training samples did not perform well.

4 Results

4.1 Data

We use the face database called PubFig83. The database contains 13,838 images of 83 individuals. Each individual does have an equal number of images. The image may have out-of-plane as well as in-plane rotation. The alignment is not perfect but the majority of the face area is visible. Small occlusion and sunglasses may be included. We split the data into training set (80%) and testing set (20%). Figure 3 shows a sample of the database. You can see a wide variety of poses and face angles. Figure 4 shows the average face of 9 subjects. Some of the mean faces are barely recognizible.

Figure 3: PubFig83 Sample Images (4 Subjects)

4.2 Accuracy

We compare the accuracy of our approach to the typical PCA approach which is a decent baseline to beat. We have 3 steps testing procedure. We first test our method on 2 subjects (211 total images) and move onto 10 subjects (approximately 500 images). Because learning on 83 subjects test (full database) takes a long time, we often run the first two steps of testing when we try different modification to the algorithm. We will call our method K-Mean Convolution (KMC). Initially, we did not apply LCN in the encoding process but the result is on par with PCA. After incorperating LCN, we boost our accuracy by 13% in 10 subjects test. We experiment further as we would like to achieve high 80% for 10 subject tests. We switch our pooling from mean to max. The accuracy greatly increases. The result is show in Table 1. For the KMC + LCN with Max Pooling on 83 subjects test, we initially get 17% accuracy rate when using LibSVM. We inspect this behavior further by testing on 20 subjects and 40 subjects. The confusion matrices are showed in Figure 5 and Figure 6. Both tests give us high 90% accuracy rate but they do not explain the result on 83 subjects. We change the classifier from LIBSVM to LIBLINEAR and the result is more reasonable. (97%)

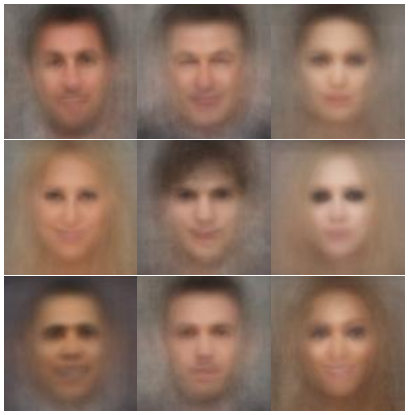

Figure 4: Mean Faces of 9 Subjects

5 Conclusion

We implement a single layer “Deep” Network. We also reduce the computational time from 9 days to 12 hours and 25 minutes (using GPU) for encoding 13,838 images. The slow process becomes the learning part which we have a little control over it as we use external classifier package. We have not inspected carefully why LIBSVM and LIBLINEAR produce such a large gap in accuracy rate on 83 subjects test. It is hypothesized that LIBSVM may suffer from precision error.

Method	Accuracy (2 Subjects)	Accuracy (10 Subjects)	Accuracy (83 Subjects)
PCA	81%	43%	22%
KMC	80%	51%	17%
KMC + LCN		64%	
KMC + LCN with Max Pooling		98%	17% (97%)

Table 1: Accuracy rate for PCA, K-Mean convolution (KMC) and its variants.

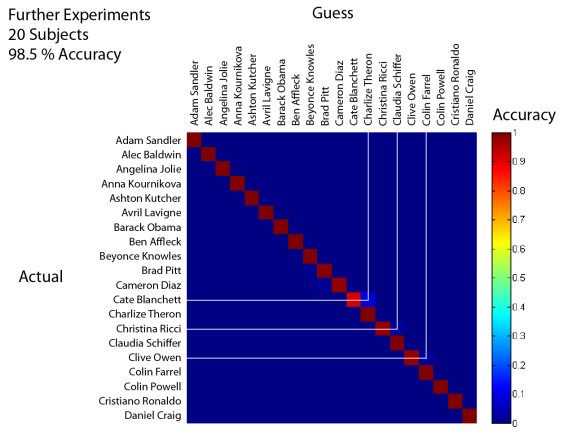

Figure 5: A Confusion Matrices of 20 Subjects Test

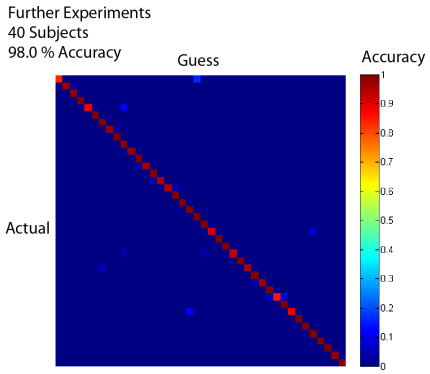

Figure 6: A Confusion Matrices of 40 Subjects Test

References

[1] Coates, A., Lee, H., & Ng, A. Y. (2010). An analysis of single-layer networks in unsupervised feature learning. *Ann Arbor, 1001*, 48109.

[2] Coates, A., & Ng, A. Y. (2011). The importance of encoding versus training with sparse coding and vector quantization. *International Conference on Machine Learning, 8*, 10.

[3] Huang, G. B., Lee, H., & Learned-Miller, E. (2012). Learning hierarchical representations for face verification with convolutional deep belief networks, 25182525.

[4] Hinton, G. E. (2006). Reducing Dimensionality of Data with Neural Networks. *Science, 313(5786)*, 502504. doi:10.1126/science.1129198

[5] Le, Q. V., Monga, R., Devin, M., Corrado, G., Chen, K., Ranzato, M. A., Dean, J., et al. (2011). Building high-level features using large scale unsupervised learning. *Arxiv preprint arXiv:1112.6209.*

[6] Lee, H., Grosse, R., Ranganath, R., & Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. *Proceedings of the 26th Annual International Conference on Machine Learning*, 609616.

[7] Nicolas Pinto, James J. DiCarlo, and David D. Cox. (2009). How far can you get with a modern face recognition test set using only simple features? *Computer Vision and Pattern Recognition.*

[8] http://www.image-net.org/challenges/LSVRC/2012/results.html