Facial Feature Detection Using Convolutional Neural Networks

Linsu Han
Columbia University
Department of Statistics
linsu.han@columbia.edu

Xiaotong Li
Columbia University
Department of Statistics
xiaotong.li@columbia.edu

Abstract

The purpose of this paper is to build facial features detection algorithms on prelabeled data. To do this, we first explore Convolutional Neural Networks (CNN) classification methods on images and visualize the architectures. We first built convolve layers to visualize each layer. Then using Keras built on Tensorflow, we implemented various architectures for training a binary label classification models including VVG, ResNet, Inception. Due to residual networks relative high accuracy and speed in comparison to other networks featured in ImageNet competition, we proceeded with more experiments on ResNet18, ResNet34, and ResNet50 using Fastai. We decided it would be more interesting for the end-user to select individual features to test. Thus, we trained multiple single label models on select features and achieved accuracy levels between 80% to 99% based on the selected feature. (Male attractiveness had 80% accuracy while gender achieved near 99% accuracy.) Finally, using Flask, we took our single label models and built a functional web page ready for deployment.

1. Introduction

Facial image classification has been a topic for deep learning for a long time. Its applications are everywhere– from the criminal detection system deployed by the police to faceID used in the mobile phone. In this paper, we will go through the processes of how we approach this topic and the applications we experimented on this topic. The layout of this paper is that we first make an exploratory attempt on exploring the widely-used deep learning technique - convolutional neural networks. We will discuss how it is constructed and what potential architectures applicable discussed in academia. To perform this, we use the first architecture proposed by Yang LeCun [8], LeNet-5 architecture, and we will break down the structure bit by bit. Then, we want to explore and compare other CNN architectures used for classification (i.e. AlexNet, VGG, ResNet, Inception). After a comprehensive analysis of the CNN network, our next goal is to use a pre-labeled dataset to build an application that uses our fine-tuning models with architectures that yields the optimal results. The details for this part are in the system overview section.

In conclusion, we will discuss the challenges and the caveats we deal with in the process. We will also further discuss the potential in this project and how we can improve in the next step of the research.

2. Related Work

Neural networks algorithms have become prevailing in the practice of machine learning for years, and its basic set up is intuitive and resemble how the human neurons would work: it receives an input, transforms it through a series of hidden layers which are independently functional, and eventually through the output layer that is fully connected. Convolutional Neural Networks are very similar in the process. The popular architecture of a CNN usually consists of Convolutional layers, ReLu layers, pooling layers, and fully connected Layers as seen in the regular neural network architecture. CNN is particularly effective in image classification because of its ability to extract features from the image that gathers related information from the image input and reduces variability. In the model comparison, we witnessed the best possible accuracy rate on classifying the classical MNIST data to be 99.77% [4], compared with other classifiers such as SVM or boosted stump. This proves that CNN can be the most valid candidate when dealing with image data because of its properties in nature.

2.1. Potential Architecture

Due to the stackable nature of layers, the model performances are largely based on how to construct the architecture. The first CNN network is proposed by LeCun is called LeNet, which is a 7-level convolutional network that uses to classifies digit data that is widely known as MNIST dataset. After the LeNet, other variants of the architectures are proposed: AlexNet (2012), ZFNet (2013), GoogleNet/Inception (2014),VGGNet (2015), and ResNet (2015). Therefore, we will look at how to make a reason-

able architecture by visualizing through LeNet. This visualization will help us make sense of how each layer will break down the input and how it will be transmitted through the architecture.

2.2. Haar Cascade Classifiers

Object Detection using Haar feature-based cascade classifiers is an effective object detection method proposed by Paul Viola and Michael Jones in their paper, "Rapid Object Detection using a Boosted Cascade of Simple Features" in 2001 [5]. In this paper, Haar feature-based cascade is used to help us locate the face area in the data preprocessing. We download a pretrained cascade file to detect facial area and it is utilized later in our application as well.

3. Data

In this paper, the primary database we will be looking at is the CelebFaces Attributes dataset (CelebA) [3]. This dataset is published by The Multimedia Laboratory by the Chinese University of Hong Kong and it is a pre-labeled data with images from openly published celebrity pictures. In this section, we will perform exploratory data analysis to have a general outlook on the dataset.

3.1. Exploratory Analysis

The CelebA dataset is a large-scale face attributes dataset with 202,599 face images, 10,177 unique identities, 5 facial landmark locations (i.e. pixel location of the nose), and 40 binary attributes. The attributes involve gender, facial details, accessories, and subjective opinion such as attractiveness [3]. Each image is stored as a .jpg file and has dimensions 178 by 218 by 3 when read using OpenCV. The binary labels are stored in a separate .csv file with the first column referencing the name of each image and remaining columns, the image attributes.

Attached in Figure 1 is a sample set of images from the dataset. Each image is labeled by at least one attribute and some images may have multiple attributes based on the actual image people perceived. To have a closer look at these 40 attributes, we use a simple count plot to check the number of occurrences of these attributes seen in Figure 2.

One obvious problem we see is that some of the attributes are highly unbalanced, such as Baldness, Bangs and 5-oclock Shadow from the figure. When tackling imbalance data, we could attempt one of the following ways: undersampling, oversampling or synthetic oversampling (SMOTE). All of these methods will include randomly distortion of the original data like increasing or decreasing the sample sizes. We have observed that undersampling will introduce the dimensionality problem for us since a large portion of the attributes is not balanced. So the best alternative to solve this issue by far is by oversampling (or SMOTE).

Figure 1. Sample images from the Celeb-A Dataset.

Figure 2. Attributes Distribution (1 means the attribute is labeled for the image and 0 means it is not for this image.

4. Methods

In this section, we will discuss the why and how we are using the CNN as our tool to detect facial feature using our dataset. The Convolutional Neural Networks rooted its idea in the convolution that it stands out from regular neural networks. As we have claimed in the previous sections, CNN is particularly useful when it comes to extracting features from a high-dimensional image than its alternatives. CNN has developed for years and there are developer tools that utilize high-level API that will do the dirty work behind the scene. However, without understanding how the black-box works will harm our mastery of this technique, and when it comes to tuning and understanding the architecture, it became vague and hard to comprehend. Therefore, we stick to the idea that the image is for visualization. And thus we will choose the simplistic architecture to visualize the architecture step by step. The first architecture is proposed by LeCun and we are starting from this architecture to take a deeper look at how it is performed behind the scenes.

4.1. Layers of CNN

An advantage when applying CNN algorithms is that layers are stackable, which allows for complex architecture that trains deeper into the data. In general, four types of lay-

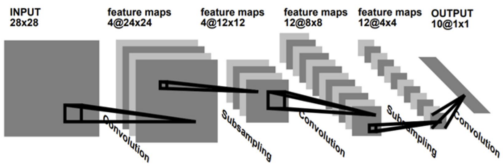

Figure 3. LeNet Architecture Pipeline

ers are commonly used, and we will go into details of these layers in the following subsections by using the illustration for a sample image from CelebA dataset.

4.2. LeNet

The LeNet illustrates a basic version of CNN and it can show how the algorithm extracts features. The fundamental architecture, LeNet-1 is best interpreted by Figure 3. However, the problem of this network is that the algorithm is not optimal since the scale of the parameter is relatively small. Thus a deeper architecture, LeNet-5 is with more connections and parameters involved. However, since the LeNet is designed specifically for MNIST dataset, thus it is not applicable to use it on our ad hoc dataset and some transition will be made. For the purpose of researching, this paper will only talk about the performance of each layer following the ideas of LeNet-5 instead of actual training on our training dataset. Figure 4 is a reference for the general pipeline of how we are going to train the model after constructing the layers architecture.

4.3. A deeper look at the layers

An advantage when applying CNN algorithms is that layers are stackable, which allows for complex architecture that trains deeper into the data. In general, four types of layers are commonly used, and we will go into details of these layers in the following subsections by using the illustration for a sample image from Celeb-A dataset.

4.4. Covolutional Layers

As the name suggests, convolutional layers are the fundamentals in a CNN architecture, which generally consists of a set of learnable filters. It extends through the full depth of the input volume. A typical filter is illustrated by the figure 1 which we see a filter with 5x5 to shrink down the original to smaller feature maps. We generally have a filter size of odd numbers, such as 3x3 or 5x5 because the intent of convolution is to encode source data matrix (entire image) in terms of filter or kernel. So when we are trying to encode, we need to make sure the filter is symmetrically shaped and there is an equal number of pixels on all sides. The length of each side of the filter would be odd regardless of the number of source pixels.

The figure 5 is an illustration of how the convolutional

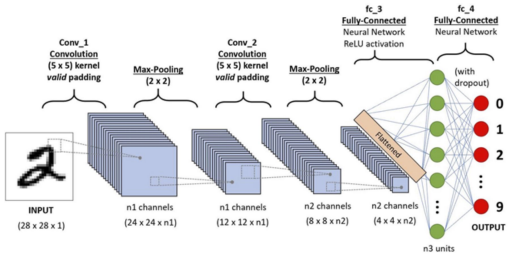

Figure 4. LeNet-5

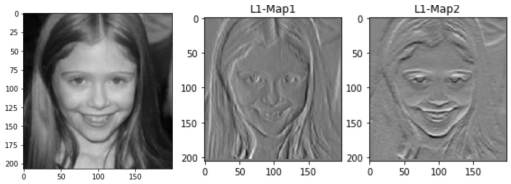

Figure 5. Right: original image in grayscale; Left: image visualization after applying the filters in convolutional layers

layer will work after applying three filters. The filters we choose here extracts features vertically, horizontally and the edges accordingly. We pass a randomly chosen digit 5 image as input and the output after applying once through each of the filters is displayed. Notice that since the MNIST data is grayscale and we are also converting the RGB image to grayscale to avoid channel issues. We will use grayscale

4.5. ReLu Layer

The ReLu layer is where the model will apply the activation function. ReLu is short for a rectified linear unit where the function returns 0 if observes negative input and return the positive values. It helps to reduce the gradient to vanish. Therefore, it is sparse In the figure 6, three outputs separately from each of the data output is shown as the results of feeding the output from the convolutional layer to the ReLu layer, and the results are shown that a majority of the grey area becomes black (which is the 0 in the pixels output) as a matter of the ReLu only output the positive values and 0. Also, we notice that the size of the image does not change after the ReLu layer since we do not apply filters at this layer.

4.6. Pooling Layer

Pooling layer in-between successive convolutional layers. Usually, we use pooling layer to reduce the number of parameters and computation of the network, thus controlling the overfitting. It is an independent layer between the Conv layers. Typically, there are two pooling: general (average) pooling or max pooling. The former will take the average of the values in each kernel while the latter will take the maximum value. Since the max pooling will eliminate

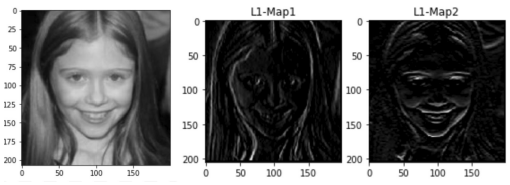

Figure 6. Right: original image in grayscale; Left: image visualization after applying the ReLu activation layer from the output of convolutional layer

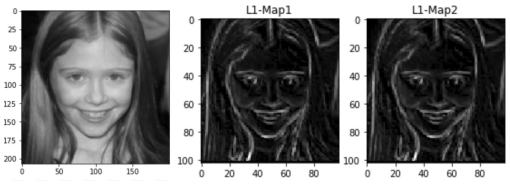

Figure 7. Right: original image in grayscale; Left: image visualization after applying max-pooling from the activation layer

some noisy activations when down-scaling the dimension, it has better performance over average pooling. In the data interpretation, we see in the figure that the features after the pooling layers a level of conformity as the features became more prominent. And the pixels sizes shrink after the pooling because in this example we are applying 2x2 filter with a stride of 2 to max pool the input. We can observe these changes in figure 7 as a result of the max pooling.

4.7. Fully Connected Layers

We eventually constructing the LeNet model based on the previously stacked layers and add flatten and fully-connected layer to achieve the classification prediction. The fully-connected layers we are talking works in a similar fashion as the regular neural networks, which will compute the classification scores that result in the volume of size. Also, each neuron in this layer will be connected to all numbers in the previous volume as we have expected in the regular neural networks.

4.8. Model Architectures

The simplistic model, LeNet-5, is a 7-layer architecture that is specifically designed for MNIST 28x28x1 input data. Figure 8 is the sample output of applying the LeNet-5 architecture on a sample of the image. However, applying LeNet-5 on large dataset seems unreasonable since the architecture is not approachable for the size of the image. Also, we may shrink down our image sizes but the loss in the information is unaffordable so we turn to its variants such as VGG-16 and VGG-19. For other candidates, we are looking at the Inception and ResNet since they have acquired huge fame because of its efficiency and the accuracy compared to the former two architectures.

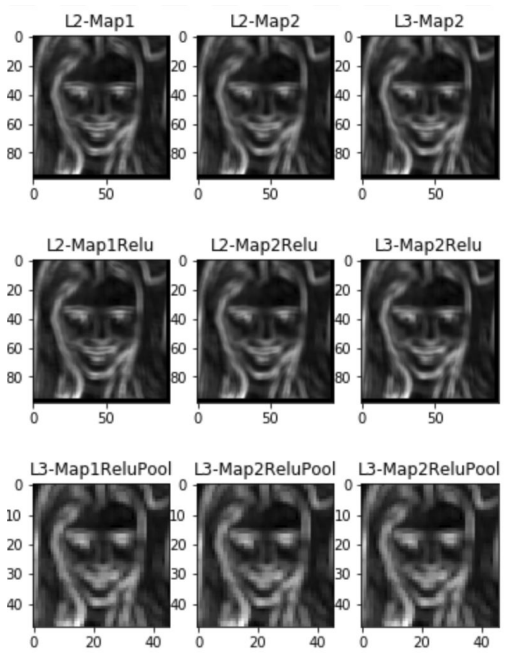

Figure 8. Applying three randomly created filters after the results from the previous input; Top: results after applying the Conv layer; Middle: results after the ReLu layer; Bottom: results after the max pooling layer.

VGG-16 and VGG-19 are two models adopt the simplistic and 3x3 convolutional filter and 2x2 pooling filter [4]. VGG-19 is not much different to VGG-16, except that it contains more layers. VGG shows the depth of the network but the network is really big. Which means longer runtime as we have seen in the results.

The next model we evaluate is the Inception architecture by Google [5]. And another groundbreaking architecture ResNet [2]. ResNet architecture has become a seminal work, demonstrating that extremely deep networks can be trained using standard SGD (and a reasonable initialization function) through the use of residual modules. Table 1 displays the results after training the models for one epoch. The accuracy column is the training accuracy, and the run-time column displays the time trained for each step. The best result comes from the ResNet50. Not only does it have deeper training network, but the run-time per step is also relatively lower. So the architecture we will be experimented on is ResNet in the section followed.

5. Experiments

With the concepts and framework at hand, a systematic way of quickly building deep neural network models was needed. Although Keras provides more control and supervision on model building, it lacks ease of use and readability. This is where Fast.ai [1] comes in. Building a model using

Architecture	Accuracy	Run-time	Parameters
VGG-16	.4909	3034s	17,217,346
VGG-19	.4978	2945s	22,527,042
Inception-v3	0.5987	1079s	24,426,786
ResNet-50	.6081	1068s	26,211,714

Table 1. Training Results for VGG, Inception and ResNet50

Keras requires 31 lines of code, while, on Fast.ai, requires only 5 lines. Fast.ai uses a very high-level neural network API and was developed with a focus on enabling fast experimentation, allowing one to go from idea to result with the least possible delay. We use Fast.ai for our experiments to quickly test out different deep learning architectures and tune our hyperparameters.

There are three main steps in using Fastais platform: 1) the creation of a data block; 2) initializing a learner object (model creation); and 3) model training

5.1. Preparation

Before we begin, notice how our image dataset lacks some key components. One, our images are not square and ResNet architectures requires inputs of 224 by 224 by 3 pixels. We also notice how some faces are non-frontal facing.

This may cause high variance in our model and lower our overall prediction accuracy. This is when Haar cascade classifiers comes in handy as it tackles both problems at the same time. Using OpenCVs detectMultiScale() [6], we load a frontal face cascade and have it pass through each image, returning a cropped output of the detected face. If there are multiple faces detected, it returns the face with the largest area. If there are none detected, or if, for example, the image features a side-face view of the person, it returns none. Figure 9 shows before and after filtering our dataset.

During face detection and extraction process, each loop taking around 90 minutes, there are 3 parameters we experimented with: minSize, scaleFactor, and minNeighbors.

minSize is the minimum possible object size; objects smaller than that are ignored. Too small of a size will incur a lot of false positives; too big, false negatives. Knowing images are 178 by 218 pixels, we began with a minimum size of 50 by 50 pixels. After reviewing extracted images sorted by smallest size first, we saw a couple of partial face images like only the nose and mouth extracted. Increasing the minSize to 60 by 60 reduced most of the occurences, however some still remain. We decided to keep it at 60 by 60. scaleFactor is the parameter specifying how much the image size is reduced at each image scale. The cascade classifier was trained to detect a fixed size, meaning if the image is bigger than the defined size,it may not be detected by the algorithm. The scaleFactor downsizes the image by a given ratio each iteration until it reaches the minSize. A small step size (i.e. 1.05 or 5%) increases the chance of detection

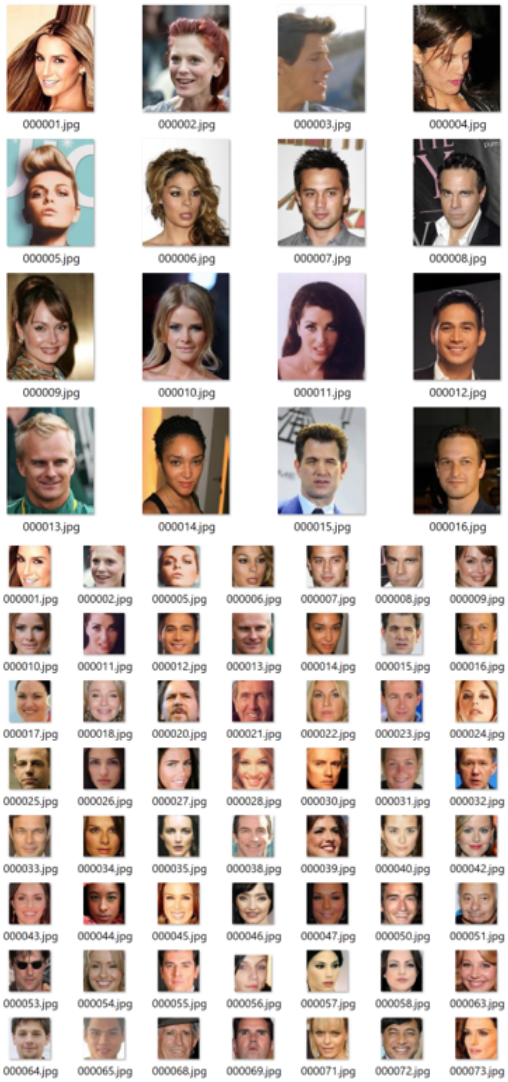

Figure 9. Before and after using Haar Cascades to extract faces from our dataset.

but has a slow speed. We chose 1.2.

minNeighbors is a parameter specifying how many neighbors each candidate rectangle should have to retain it. It is essentially a tolerance level for detection. A low number results in more detections but high false positive rate. A higher value would increase detection quality. With it set to 3 (default), it removes some obvious non-frontal face pictures like 000003.jpg and 000004.jpg. The original dataset is size is 202,499. When we experimented with the parameter at 10, our data set reduced to 175,625; and, at 20, to 147,814. We chose to proceed with the latter, as to preserve quality.

Architecture	Accuracy
ResNet18	.772201
ResNet34	.764801
ResNet50	.776370

Table 2. Architecture Accuracy

5.2. From Data Block to Training

The creation of a data block can be broken down in steps. First we initialize its creation by reading in a labels dataframe containing the image paths on the first column and specifying which column the label we want to analyze is on. For our initial model training we randomly split 80% of our dataset as training and 20% as validation. This allows us to assess the model and adjust hyperparameters later. For production, we use a 95% to 5% split to maximize our model accuracy. We call in some image transformations to augment our data and increase model robustness. This step also automatically resize all input images to be 224 by 224 by 3, the default for ResNet. Details on specific transformations will be discussed later. Finally, we bunch everything together and do some initial normalization of our data bunch by subsampling the mean and standard deviation of each color channel. We store the completed data block object under the alias data.

With the data block in hand, we just need to specify a model architecture found in PyTrochs API [7] (i.e. AlexNet, VGG, ResNet), and initialize a learning object using learn = cnn_learner(data, model, metrics=[accuracy]). Now all that is needed to fit each model is calling learn.fit(...). However, to find optimal hyperparameters we will perform some additional experiments.

5.3. Architecture

Recall from our methods section, we concluded that we will use ResNet to perform our analysis. In this section, we will compare the model accuracy of ResNet18, ResNet34, and ResNet50 on a predefined binary label. In our initial experiment on the label Male, all results displayed an approximate 99% accuracy. We decided to test this on a much more difficult task such as attractiveness, particularly male attractiveness since attractiveness standards differ between gender. We kept the Fastai default transformations. Since we are only fitting one epoch for quick experimental tests, we set the learning rate to be a reasonable .01 for each model. Details on each model architecture can be found in the attached fastai_experiments.ipynb. The results are displayed in Table 2.

Notice model accuracy did not change much between the models. This is an expected property with Residual Networks due to its identity mapping which generally ensures a nondecreasing accuracy. However, It is important

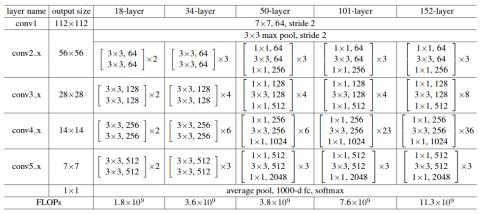

Figure 10. ResNet Layers.

Transformations	Accuracy
Default	.776370
No Warp	.790235
No Zoom	.788448
Max Lighting (40%)	.787938
Max Rotate (20 degrees)	.785301

Table 3. Transformation Accuracy

to note that ResNet50 did perform marginally better than both ResNet18 and ResNet34. One possible explanation is a change in architecture; the convolutional layer blocks of ResNet18 and ResNet34 are identical, while convolutional layer block used for ResNet50 differs from the two as shown in Figure 10. For deployment, we use ResNet50 for its slightly higher accuracy.

5.4. Transformations

Our default transformation includes random horizontal flips, brightness/contrast adjustments (up to 20%), rotations (up to 10 degrees), warps (up to 20%), and zooms up to (10%). The probability of each individual random event happening is .75. These default hyperparameters are the optimal hyperparameters trained on ImageNets 14 million images. In our experiments using ResNet50, we adjust each hyperparameter keeping the others constant and train each model with a learning rate of .01. Recall accuracy levels for the label Male is high as is (99%), so we train on the label Attractive for all Male == 1. Results are shown in Table 3.

Results Results from using ResNet50 and default transformations are displayed in Figures 11 to 14.

5.5. Multi-label problem

Initially, we approached facial feature detection as a multi-label classification problem. Since each attribute is binary, we assigned each observation categories for which attributes are present. For example if image X has the following attributes Male: 0, Attractive:1, Smiling:1, we would assign the observation with the two categories: Attractive and Smiling. With 40 binary attributes, we achieved an accuracy threshold value of .90 percent. Note that high accuracy threshold value can be misleading since accuracy threshold only measures the accuracy when predicted out-

Total time: 1:55:12

epoch	train_loss	valid_loss	accuracy	time
0	0.379300	0.398745	0.814382	23:07
1	0.378007	0.372170	0.824270	23:01
2	0.369391	0.355674	0.826292	23:01
3	0.350387	0.360205	0.830562	23:00
4	0.354371	0.349337	0.833483	23:00

Figure 11. Attractive, Female

Total time: 1:16:08

epoch	train_loss	valid_loss	accuracy	time
0	0.475131	0.435338	0.787002	15:19
1	0.443314	0.416178	0.794828	15:11
2	0.430752	0.403103	0.793807	15:13
3	0.418600	0.405550	0.789724	15:11
4	0.410733	0.396587	0.797210	15:11

Figure 12. Attractive, Male

Total time: 3:10:50

epoch	train_loss	valid_loss	accuracy	time
0	0.347901	0.345192	0.836784	38:21
1	0.337327	0.339682	0.840438	38:06
2	0.329978	0.335590	0.840032	38:06
3	0.335990	0.329668	0.844093	38:09
4	0.328828	0.329668	0.843822	38:06

Figure 13. Bags Under Eyes

Total time: 3:10:47

epoch	train_loss	valid_loss	accuracy	time
0	0.157539	0.137549	0.944241	38:09
1	0.141987	0.130647	0.948978	38:08
2	0.142948	0.125754	0.949249	38:09
3	0.134236	0.128653	0.949790	38:11
4	0.129806	0.134499	0.950061	38:08

Figure 14. Mouth Slightly Open

Total time: 3:10:49

epoch	train_loss	valid_loss	accuracy_thresh	time
0	0.235513	0.218691	0.902713	38:16
1	0.225805	0.210983	0.906090	38:09
2	0.223606	0.207097	0.907335	38:06
3	0.219102	0.203905	0.908844	38:07
4	0.217510	0.202060	0.909528	38:09

Figure 15. Multi-label results

put length equals actual output length. When we wanted to narrow down the attributes to only include balanced classes (i.e. Male, Attractive, Mouth_Slightly_Open), problems arose during training since many of the categories had a null attribute (the case when Male:0, Attractive:0, Mouth_Slightly_Open:0). This was not the case with forty attributes since the probability of all forty attributes being null was extremely low. We attempted to correct this by creating labels that complemented each class (i.e. a female would be labeled as Male:0 and Not_Male:1). However, this created huge accuracy problems post training. Thus, we decided to drop multi-label classification and focus on single label classification, as each accuracy measure is more relevant, and results are more applicable to the end user.

5.6. Multi-label Results

Figure 15 shows the results.

6. System Overview

Originally we used our own laptops to run all the machine learning framework, however training a deep neural net with millions of weights is extremely computationally intensive. And a single ResNet50 model built using Fastai took over 10 hours to train. To speed up the process, we used a GPU-accelerated machine from Googles Cloud Compute Engine to train our models. We also utilized the CUDA, which enables developers to speed up compute-intensive applications by harnessing parallelization of GPUs. The time it took to train one model decreased significantly to approximately 34 minutes per model With a CUDA supported NVIDIA Tesla K80.

Transfer of large amounts of data between machines was also fence we encountered along the way. Thankfully, Google Bucket's storage engine allowed us to move data and models between our machine and the cloud servers.

Finally, we built a live video engine to do predictions and send it back as a video feed. Using Flask, we managed to build a local hosted app with nice and tidy UI.

To use our app, make sure to have the following packages installed: flask, pytorch v1.0, fastai, opencv. It is crucial to

have v1.0 of pytorch and not v1.1. Edit the 'path' variable in webpage.py. Then cd to the project directory. On windows type 'set FLASK_APP=webpage.py' on Mac/Linux it's 'export FLASK_APP=webpage.py'. Then 'flask run' and the app will be hosted on localhost:5000.

If you encounter problems with the flask webpage, you can still test our results running capture_multi.py after changing some paths.

7. Conclusion

Throughout this project we learned a lot of fundamental theories on neural networks and image classification as well as idea realization and app development. We began with zero knowledge on CNNs and managed to go from visualizing its inner workings from scratch to building large models and training huge data sets. We gave a lot of thought on product design and how to make it applicable to the end-user. We self-taught ourselves the basics of front-end and back-end development and communication between different project modules. We learned how use Unix systems and how to SSH into Google Cloud's Compute Engine and perform computational requests overnight. In the future we plan on multi-threading our Flask application to serve multiple clients.

References

[1] Fastai. Documentation of fastai. *https://docs.fast.ai/*.

[2] A. Z. K. Simonyan. Very deep convolutional networks for large-scale image recognition. *ICLR*, 2015.

[3] X. W. L. Ziwei, L. Ping and X. Tang. Proceedings of international conference on computer vision (iccv). *Proceedings of International Conference on Computer Vision (ICCV)*, 2015.

[4] Y. LeCun. The mnist database. *PROC OF IEEE*, 1998.

[5] OpenCV. Documentation of haar cascade in opencv. *https://docs.opencv.org/*.

[6] OpenCV. Documentation of opencv. *https://docs.opencv.org/*.

[7] Pytorch.

[8] Y. B. Y. LeCun, L. Bottou and P. Haffner. Gradient-based learning applied to document recognition,. *http://yann.lecun.com/exdb/mnist/*, 1998.