

Machine Learning Engineer Nanodegree:

Image Localization for Emotion Detection

Capstone Proposal

Bartłomiej Chrzaszcz

July 24th, 2017

I. Definition

Project Overview

Over the last couple of years, image classification has been getting a lot of traction in different industries. Tesla and Google have been using this technology for self-driving cars and Facebook has been doing it for facial recognition for example. They are also very accessible through the different APIs out there and these powerful algorithms can even be used on your smartphone today. These algorithms use deep learning algorithms, specifically convolutional neural networks generally, which is a subset of machine learning; they try to learn what something is out of a list of different possibilities. For example, if you train an algorithm on a set of images containing 10 different items, that algorithm would be good at identifying those 10 items in new sets of images you give it. As seen in Figure 1, the algorithm uses techniques such as convolutional layers, ReLU activation functions, and pooling layers, to end up figuring out what's the probability of the item in the image being a car, truck, airplane, ship, horse, or something else using a regular fully connected neural network at the end of the convolutional layers. CNNs are still a fairly new field, with them being explored first in 1998 by researchers at the AT&T Shannon Lab where they discovered how effective CNNs are at recognizing simple objects, such as the familiar MNIST dataset [1]. There have been fairly revolutionary CNN innovations that have created great classification accuracies such as Fractional Max-Pooling [2] and The All Convolutional Net [3].

In this project, I create a CNN that can detect the emotion you are feeling by feeding it the video stream from your webcam. I used a Dataset, called Fer-2013, that was available through Kaggle. Even though the competition was over, my CNN would have likely placed me in the top 5 on the leaderboards.

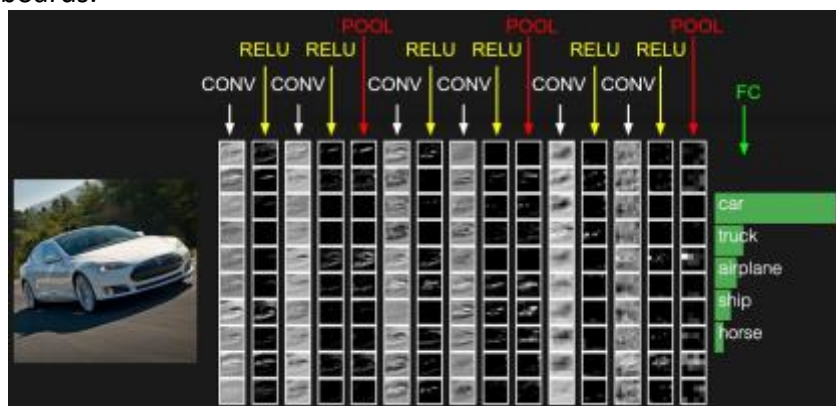


Figure 1

Problem Statement

In this study, we will mainly look into using CNNs to recognize emotions of humans using pictures of people's faces and validate its accuracy in recognizing emotions of a single face against the leaderboard on Kaggle for the dataset. We will train and test this dataset using AWS's EC2 p2.xlarge instance which has a 12gb Nvidia Tesla K80 video card. This will make training and tuning hyper parameters of models very quick. As a secondary goal originally, we wanted to determine their success in object localization so they can be fed an image of multiple individuals and determine each person's emotion but realized this may take a lot more work in modifying the data than I thought; for example, the dataset already has cropped images into including just the face, which makes object localization a lot harder. Thus, instead, our secondary goal will be predicting a user's emotions through their webcam.

Metrics

We will use F-score, which combines precision and recall, and accuracy for this model as the former is a very accurate measure of the strength of a model when the data set is skewed, which we will get into in the Data Exploration section, and the latter as that is what the dataset's Kaggle leaderboard used which makes comparing to other competitors easier.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$FScore = 2 * \frac{Precision * Recall}{Precision + Recall}$$

where

$$Precision = \frac{TP}{TP + FP}, \quad Recall = \frac{TP}{TP + FN}$$

Also, I will want to determine how quickly my CNN classifies images as a CNN that takes minutes to classify isn't very good in some applications such as a self driving; in this case, my CNN wouldn't need to be super-fast but it would be an important metric to know.

II. Analysis

Data Exploration

The dataset can be attained here: <https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/data>

The dataset that will be used has been made available by Pierre-Luc Carrier and Aaron Courville and is available on Kaggle under the name "Challenges in Representation Learning: Facial Expression Recognition Challenge". It consists of 35,887 48x48-pixel grayscale images of faces that have already been classified into seven categories:

- Angry (4953 entries)
- Disgust (547 entries)
- Fear (5121 entries)
- Happy (8989 entries)
- Sad (6077 entries)
- Surprise (4002 entries)

- Neutral (6198 entries)

More specifically, the training set consists of 28,709 examples while the testing set is split up into 2 3,589-image groups: the private tests used for the Kaggle leaderboards and public tests for validating one's algorithm. In this study, I'll use one of the testing sets as validation of my accuracy and the other one to compare my accuracy against people on the Kaggle leaderboards. Given that I want to train my CNN to detect emotions, this dataset is perfect for doing so as all the images are labelled. Also, as you can see, there are very few pictures of disgust, so my CNN may not become so great at identifying that emotion in the real world. Thus, for this model, I will remove that class since for a CNN, a lot of examples to train on are required and in this dataset, there aren't enough images of disgusted faces that lets me be confident that the CNN understands what features a face of a person who feels disgusted looks like (to put into perspective, there are only 28 disgust images per test set). Even though F-score can account for skewed data, my model may be inaccurate when it comes to predicting emotions from the webcam stream.

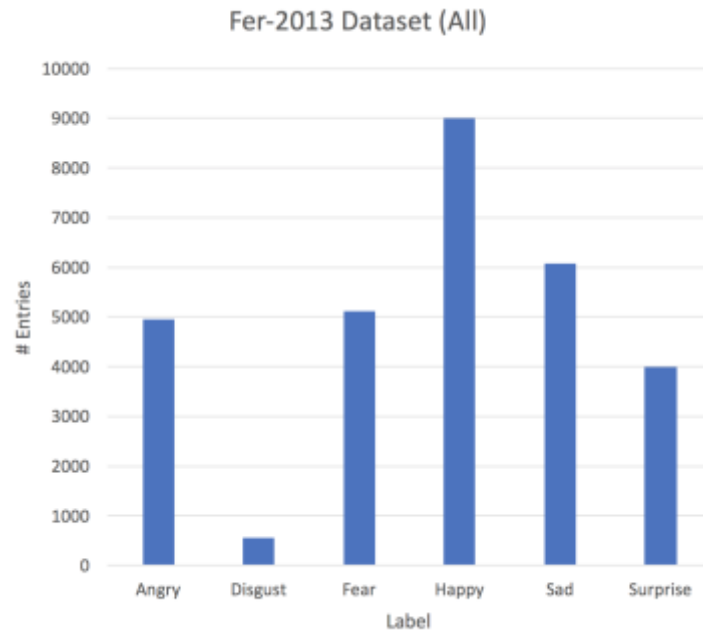
One thing to note is that the .csv file that comes with the dataset has each image as a list of pixels in one long row. Thus, I had to do some data processing/transformation to turn that into actual images. I turned those rows of pixels into images as shown below (using `csv2Image.py` as located in the .zip file). As you can see, this dataset will be difficult to train on due to the watermarks in some images and invalid images in general. I will leave the watermarked outlier images but I'll do my best to skim through the dataset and remove some of those invalid images like the one with the 3 dots in the center of Figure 2.



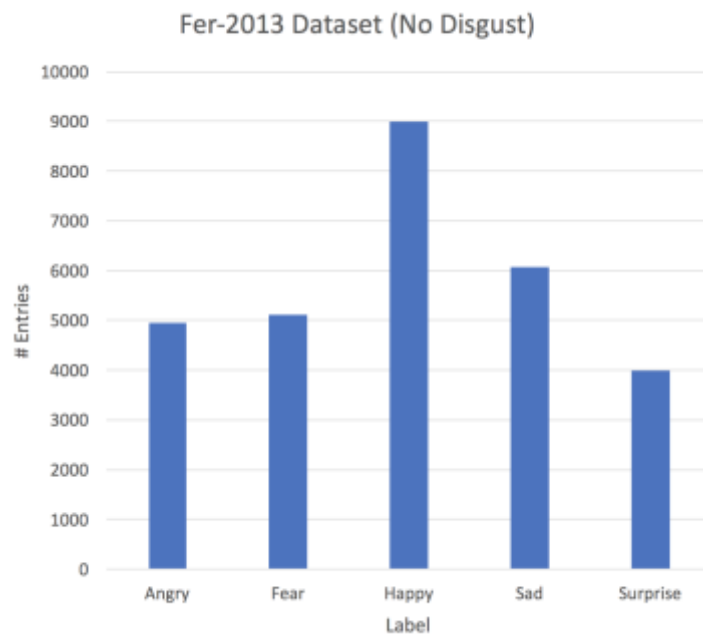
Figure 2

Exploratory Visualization

The plots below compare how much the disgust label skews the data and can affect my model. This helps in deciding whether it is worth training my model on disgust as it can make my model partially learn some features that it doesn't know enough about. In the end, after visualizing data, I'm more confident in my decision to train my model on 6 of the 7 labels.



Graph 1: the plot shows the number of image example entries of each label in the dataset. As you can see, there are at least 4000 entries for every label other than disgust, which has over 7 times less entries.



Graph 1: the plot shows the number of image example entries of each label other than disgust in the dataset. As you can see, the data is less skewed, however happy is still the dominant label. Thus, using F-score for evaluating my model against my benchmark is still required.

[Algorithms and Techniques](#)

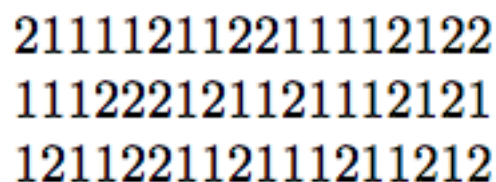
The model is a Convolutional Neural Network as discussed earlier. They are strong suited for image classification problems, but require a lot of data compared to other models like

Independent Component Analysis (ICA) due to the multitude of weights and biases a CNN needs to train. Thankfully, 35,887 is enough to train a fairly shallow CNN.

The very first thing I'll need to do is clean the data. As discussed above, the data consists of one long line of numbers in a .csv file cell. I will need to go through all the data, and convert those lines of numbers to 2-d arrays of size 48x48. After that, I will need to convert all those entries to images and see what images are "outliers" in that they aren't pictures of faces.

Then, I will need to figure out the architecture of my CNN. Specifically, I want to try implementing Fractional Max-Pooling [2] and a learned activation function for my CNN [3][4]. All 3 papers' CNNs mentioned, "Fractional Max Pooling", "The All Convolutional Net", and "Learning Activation Functions" were tested on the CIFAR-10 dataset and achieved stellar results, both classifying 32x32 color images with 10 possible labels with 92-97% accuracy (they were tested on some other datasets too, not just this one). Since my research will focus on similarly sized images (48x48) with similar number of labels (six), I feel these approaches will still be highly affective for my dataset even though the CIFAR-10 dataset consists of RGB images and my dataset is gray scaled.

Firstly, Fractional Max-Pooling, or FMP for short, is a recent new development in CNN research. For a while, an MP2 layer, which is a 2x2 max-pooling layer, has been used primarily in CNNs because they are fast, reduce the size of the layers efficiently, and encode common patterns in images. However, since the size decreases very quickly, you cannot have a convolutional layer followed by a max pooling layer several times when a deep CNN consisting of many layers is needed which may introduce overfitting. FMPs on the other hand, reduce the size of an image by some factor a where $a \in (1,2)$ typically; it can be greater than 2, but the case where one would use FMPs is to slowly reduce the size of an image gradually. Because of this, they can be more arbitrarily inserted into a layer of CNNs to maintain the invariance of the CNN without losing too much information by reducing the size before introducing more convolutional layers. This fraction is usually calculated by dividing the current size of the matrix by the new matrix/image you want (e.g. I have a 100x200px image that I want to reduce to 75*150 so I'll have a 4/3 max-pooling layer). However, they can still be effective when choosing alphas randomly and also in random locations in the CNN. They don't take the max value from a pool of two and a half pixels for example, but instead do a random or pseudo random combination of 1x1, 2x2, or other pooling sizes over the entire image. For example, what if you had an image that had a width of 25 pixels and you wanted to reduce the size to 18 pixels in width?



21111211221112122
111222121121112121
121122112111211212

Figure 3

Taking the first row in Figure 3 for example, the first pool would be of size 2x2, then 1x1, 1x1, 1x1, 1x1, 2x2, 1x1, 1x1, 2x2, 2x2, etc. In total, there are 18 of these pooling layers in that row.

So, each of these pools will grab the max value pixel in the pool (the 1x1 pools would just grab themselves). But what if we were to sum the numbers from left to right? As you can see, 2+1+1+1+1+2+1+1+2+2....etc is equal to 25 which makes sense since the original image had 25 pixels in width. So as you can see, we reduced the size of a 25 pixels width image to 18 pixels by using a random order of 2x2 and 1x1 pooling layers and never had to divide a single pixel in half.

Pseudorandom order on the other hand, is kind of like ordering the "random" pools so that there's a similar amount of 1x1 and 2x2 pooling layers that are spaced out evenly like in Figure 4:

112112121121211212
212112121121121211
211211212112121121

Figure 4

As you can see, there's more of a pattern in the randomness here. In Figure 3, you have 4 consecutive 1x1 pooling layers in the first row. But in Figure 4, you have at most 2 exact same pooling orders in consecutive order. One issue though with random and pseudorandom FMP is that they tend to underfit when combined with dropout.

For a learned activation function, I will use an Adaptive Piecewise Linear (APL) activation unit [4] which has the following formula:

$$h_i(x) = \max(0, x) + \sum_{s=1}^S a_i^s \max(0, -x + b_i^s)$$

Figure 5

where S is a hyper-parameter set in advanced which says how many "hinges" or rectifier activation functions there are. The other variables are a_i^s and b_i^s for $i \in (1, S)$. The a_i^s variables control the slopes of the linear segments while the b_i^s variables control the locations of the rectifiers which are both learned during training. Some interesting things can happen such as non convex activation functions like below and for a large enough S , can create complex shaped activation functions:

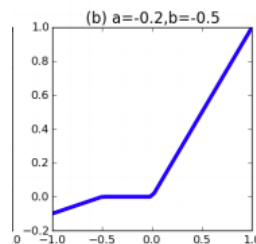


Figure 6

APL units are similar to the PReLU activation function [5] which also learns some value alpha while training as seen in the function below on the right in Figure 7:

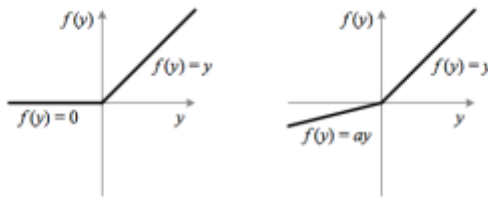


Figure 1. ReLU vs. PReLU. For PReLU, the coefficient of the negative part is not constant and is adaptively learned.

Figure 7

After training my model and verifying that it accurately classifies images, I'll go ahead and validate it on the private set and compare it to the Kaggle leaderboard. And finally, I will try and use openCV so that the CNN can take in a stream of video from a webcam.

Benchmark

I plan to compare my accuracy results of the CNN against the accuracy of those who completed this competition on the Kaggle leaderboards. In addition, I will train my CNN I made for the image classification Nanodegree project (with some slight modifications to account for a different sized image) and test it with the same metrics I'll measure my new CNN with. For my webcam task, I will just explore how my CNN classifies my face but I will not be able to explore how accurate it is over certain examples.

III. Methodology

Data Preprocessing

As discussed earlier, the first thing done in terms of data preprocessing is turning those strings of numbers into 3d arrays that I can then save as .png images. I did this by loading each .csv cell, using Numpy to transform it into a 3d array, and finally save it in a specific folder depending on the label (i.e. 0,1,2,3,4,5,6) and data category (i.e. training, public test, private test). This was done using the csv2image python script. Then, I went through the data and visually scanned for any outlier images like the image with the 3 dots earlier. When training, I normalized it by rescaling every image by 1/255, rotated a random amount in the range of 30 degrees, and random shearing, vertical shifting, horizontal shifting, and zooming in the range of 0.2. Because all this shifting would create blank pixels, I let those missing pixels take on the value of their nearest neighbor's value. Finally, I flipped the image horizontally randomly. I didn't flip vertically as based on my goals of creating a webcam app, my CNN would rarely have to classify someone who is using their computer's webcam upside down. Some examples of those preprocessed images can be found in Figure 8. As you can see, there is some noise/distortion, but generally the important parts of the faces are still there. Thus, when training, I will add some dropout in the first layer to account for this noise. However, in general, this preprocessing will make training a lot more robust as the point at which the training accuracy becomes larger than the validation accuracy will be delayed most likely.



Figure 8

Implementation

There were two parts of the implementation process:

1. The CNN training stage
2. The webcam development stage

During the training stage, the classifier used the regular images but preprocessed them on the fly. Thus, no two training sessions are the same as even though the same image can be viewed by the model twice, it may have been preprocessed differently the second time (e.g. flipped, rotated 20 degrees the other way, etc.).

All training was done on AWS's EC2 p2.xlarge instance. The benefit of the instance is that it has a single Nvidia K80 GPU with 12GB of VRAM, 4 vCPU's, and 61GB of RAM. This made training extremely fast; for a training session that could have taken 35 hours on my 2016 Macbook Pro, it took 30 minutes on the EC2 instance. And at \$1/hr, this was a very economical choice.

My main model I trained on is located in the Python model called "model.py". Next, I will go over section by section what my code does. Then I will explain some of my decisions.

```
14     seed = 7
15     np.random.seed(seed)
16     training_size = 28273
17     validation_size = 3533 # size of private test
```

Figure 9: Lines 14-17 are making sure Numpy uses the exact same random numbers each time so it's easier to recreate results later. I also define the training and validation size based on the subset of the data I'm using.


```

22 def model(weights = None, S = 5, p_ratio = [1.0, 2.6, 2.6, 1.0]):
23
24     model = Sequential()
25
26     model.add(Dropout(0.0, input_shape=(48, 48, 1)))
27     model.add(ZeroPadding2D((1,1)))
28     model.add(Conv2D(64, (5, 5)))
29     model.add(APLUnit(S=S))
30     model.add(MaxPooling2D((3,3), strides=(2,2)))
31
32     model.add(ZeroPadding2D((1,1)))
33     model.add(Conv2D(64, (5, 5)))
34     model.add(APLUnit(S=S))
35     model.add(InputLayer(input_tensor =
    *   tf.nn.fractional_max_pool(model.layers[7].output, p_ratio,
    *   overlapping=True)[0]))
36
37     model.add(ZeroPadding2D((1,1)))
38     model.add(Conv2D(128, (4, 4)))
39     model.add(APLUnit(S=S))
40
41     model.add(Flatten())
42     model.add(Dropout(0.25))
43     model.add(Dense(4096))
44     model.add(APLUnit(S=S))
45     model.add(Dense(6, activation = 'softmax'))
46
47     model.compile(loss = 'categorical_crossentropy', optimizer = 'adam',
    *   metrics = ['accuracy', fbeta_score])
48
49     if weights:
50         model.load_weights(weights)
51
52     return model

```

Figure 10: Lines 22-52 are where I define my model. There were several iterations of how my model would look like. It first started off being a simpler and deeper network of VGG-19 [6] which looks like the diagram in Figure 11. VGG-19 was the model built by the VGG team at the ILSVRC-2014 competition; it got first and second place for the localization and classification tasks respectively.

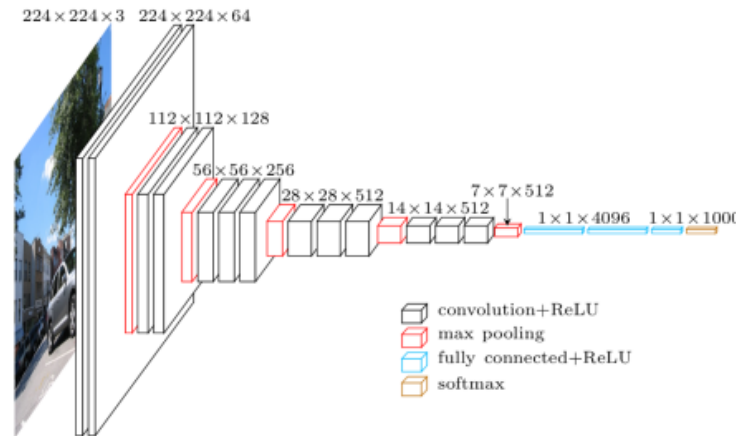


Figure 11

There were also padding layers after each maxpooling layer and on the original image too – just like VGG-19. My model was similar in that it included padding, 2 64 depth layers, 2 128 depth layers, and 3 256 depth layers followed by some dense layers. However, after experiencing some overfitting even with heavy data preprocessing, I had to remove some convolutional layers. I looked at other models and found AlexNet [7] as seen in Figure 12 a good choice to find inspiration from. Even though the overall model did worse than VGG 19 on the ImageNet dataset, it is a shallower model with fewer convolutional layers that may be better suited for creating a smaller model with fewer weights and biases that can be trained on a smaller dataset. However, because of it being shallower, the filter and pooling choice can be complex. For example, AlexNet uses 3×3 max pooling layers with a stride of 2×2 .

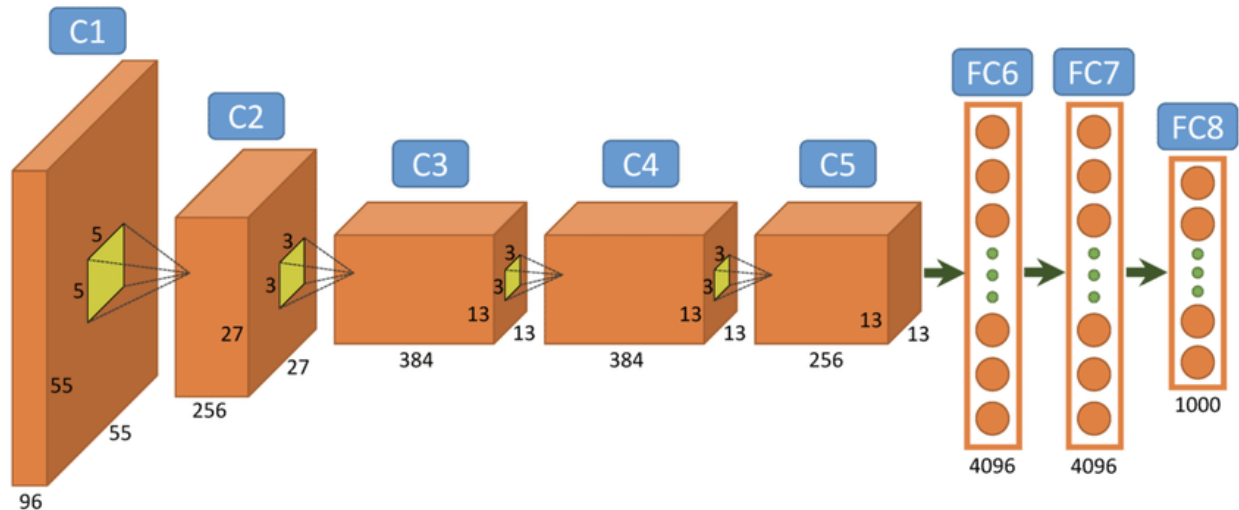


Figure 12

Overall, I chose to maintain my padding layers from VGG-19, but only keep 3 convolutional layers with large filter sizes, 2 pooling layers, and a single hidden dense layer (also seen in Figure 13):

1. Add 0.1 dropout on the input when training (0.0 when testing)
2. Add 1×1 padding which increases the image by 2 pixels in width and height
3. Convolutional layer with depth of 64 and filter size of 5×5

4. APL activation function
5. 3x3 max pooling layer with strides of 2x2
6. Repeat 2-4
7. Approximately $\sqrt{2} \times \sqrt{2}$ fractional max pooling layer (FMP)
8. Dropout of 0.25 (0.0 when testing)
9. Dense layer of size 4096
10. Softmax output layer to classify one of 6 labels



Figure 13

I chose to use categorical cross entropy as my loss function since this was a multi label problem and the Adam optimizer since it does well in many machine learning models. In terms of the implementation of APL units, they aren't available in Keras so I wrote the activation function myself which can be found in APL.py – it was based off the implementation of PReLU in Keras. Also, since when training Keras determines model scores on a batch and not the entire dataset, F-score was removed so I had to write the function in batch_fscore.py so I can use F-score when training and testing (however when testing, it will test over the entire dataset and not just a single batch).

```

55 # build the model
56 model = model('model9.h5') # my weights
57
58 batch_size = 256
59
60 # Preprocess inputted data
61 train_datagen = ImageDataGenerator(
62     rescale=1./255,
63     rotation_range = 30,
64     shear_range=0.2,
65     width_shift_range=0.2,
66     height_shift_range=0.2,
67     zoom_range=0.2,
68     fill_mode='nearest',
69     horizontal_flip=True)
70
71 test_datagen = ImageDataGenerator(rescale = 1./255)
72
73 # Fit the model
74 train_generator = train_datagen.flow_from_directory(
75     '../Training', # this is the target directory
76     target_size = (48, 48), # all images will be resized to 48x48
77     batch_size = batch_size,
78     color_mode = 'grayscale')
79
80 # this is a similar generator, for validation data
81 validation_generator = test_datagen.flow_from_directory(
82     '../PrivateTest',
83     target_size = (48, 48),
84     batch_size = batch_size,
85     color_mode = 'grayscale')

```

Figure 14: Lines 61 to 69 are all my preprocessing settings when training while line 71 doesn't do any type of transformation when training other than normalizing the data. Lines 74-85 passes the images from the specified folder after preprocessing. The folders are structured like this:

-Training

-0

-image1.png

-image2.png

-etc.

-1

-image1.png

-image2.png

-etc.

-etc.

```

95 # ~~~~~ Train model ~~~~~
96 # callback functions
97 save_best = ModelCheckpoint('model10.h5', monitor='val_acc', verbose=2,
    • save_best_only=True, mode='max')
98 reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2,
99                               patience=5, min_lr=0.001, verbose=1)
100
101 model.fit_generator(
102     train_generator,
103     steps_per_epoch = training_size // batch_size,
104     epochs=30,
105     callbacks = [save_best, reduce_lr],
106     validation_data=validation_generator,
107     validation_steps= validation_size // batch_size)
108 #model.save_weights('VGG16_regular_second_try.h5') # always save your weights
    • after training or during training

```

Figure 15: “save_best” on line 97-98 is a callback function only saves the weights which results in the highest accuracy on the validation dataset when training. “reduce_lr” on lines 98-99 reduces the learning rate by 80% if validation loss doesn’t decrease by at least 0.001 for 5 epochs.

Secondly, in terms of the webcam application, it consisted of using Opencv to capture an image from a laptops webcam and then transform that image into a [1, 48, 48, 1] Numpy array and passing it into my trained model and displaying the predictions on a Matplotlib Pyplot. The model used for the CNN is called webcam_CNN.py (same as model.py but with all the training logic removed and easier to get predicted outputs) and the logic for capturing and passing images from the webcam into the model is in webcam_capture.py.

Refinement

As discussed in the previous section, I manipulated the amount of certain layers and type of layers until I was satisfied with my validation accuracy. Below, I’ll list some of those models and their accuracies generally over about 30 epochs:

1. Simple VGG: 58.438%
2. Simple VGG with smaller FC layers: 59.536%
3. Same as #2 but with PReLU instead of APL and mp2 instead of FMP: 57.329%
4. Combination of simple VGG and AlexNet with PReLU and MP2: 60.748%
5. Final model, like #4, but with APL and FMP, many epochs, and varying preprocessing: 66.22% (F-score: 66.07%)

In terms of my 5th model, I was happy with the accuracy I was getting in my 4th model, but wanted to improve upon it. Thus, I ran it over more epochs and changed up the preprocessing every few epochs. For example, sometimes I made the shear and shift ranges 0.5, the rotation 45 degrees, add random vertical flips, and briefly sometimes no preprocessing other than normalization.

For my webcam, there was no need for refinement since it solely relies on how accurate the CNN is.

IV. Results

Model Evaluation and Validation

As mentioned earlier, I chose to go with the model shown in Figure 13. On the validation set, PublicTest, it got an accuracy of 66.22% and F-score of 66.07%. For the PrivateTest set, it got an accuracy of 65.385% and an F-Score of 65.202%. On Kaggle, this put me in 5th place unofficially on the Public and Private Leaderboard. To put this into perspective, the team who got first place have an accuracy 4.404% and 5.777% greater than mine for each 50% portion of the PrivateTest set respectively. Taking the average over the two, they approximately have an accuracy that is 5.0905% greater than mine over the entire dataset. So even though an accuracy or F-score around 66% generally means the model is unreliable, relative to other competitors, I did very well. However, one thing to remember is I didn't train my model on predicting Disgust so these results may be skewed (likely positively for F-score but either positively or negatively for accuracy). As was seen earlier, this dataset was very difficult to train on so even though at this point I felt my model may be unreliable, I still went ahead to see whether my model can accurately classify faces from a webcam video stream. As can be seen in the section Free-Form Visualization, my model was able to very accurately classify my face. Thus, I can now say with confidence that my model is robust through the extensive webcam stream testing I've done.

Justification

Other than the results discussed earlier about how well my model classifies emotions from a webcam video stream, I still need to compare my model to a benchmark. The benchmark in question will be my CNN from the image classification project that was part of the Deep Learning unit in the MLND. The model architecture looks like the following:

1. Convolutional layer with depth of 64 and filter size of 3x3
2. LeakyReLU activation function (alpha=0.2)
3. Convolutional layer with depth of 128 and filter size of 3x3
4. LeakyReLU activation function (alpha=0.2)
5. 2x2 max pooling layer with strides of 2x2
6. Dense layer of size 384
7. Dropout of 0.3
8. Dense layer of size 192
9. Dropout of 0.3
10. Softmax output layer to classify one of 6 labels

For the CIFAR dataset, it got an accuracy of 68.56%. For this dataset for approximately the same number of epochs and same preprocessing steps, it got an accuracy of 50.17% and F-score of 36.55%. Thus, my model made a drastic improvement over my last model.

V. Conclusion

Free-Form Visualization

When implemented with the webcam stream, the model can confidentially classify faces. It's difficult to see in the video attached in the .zip file with the bar graphs like the screenshot in Figure 16:

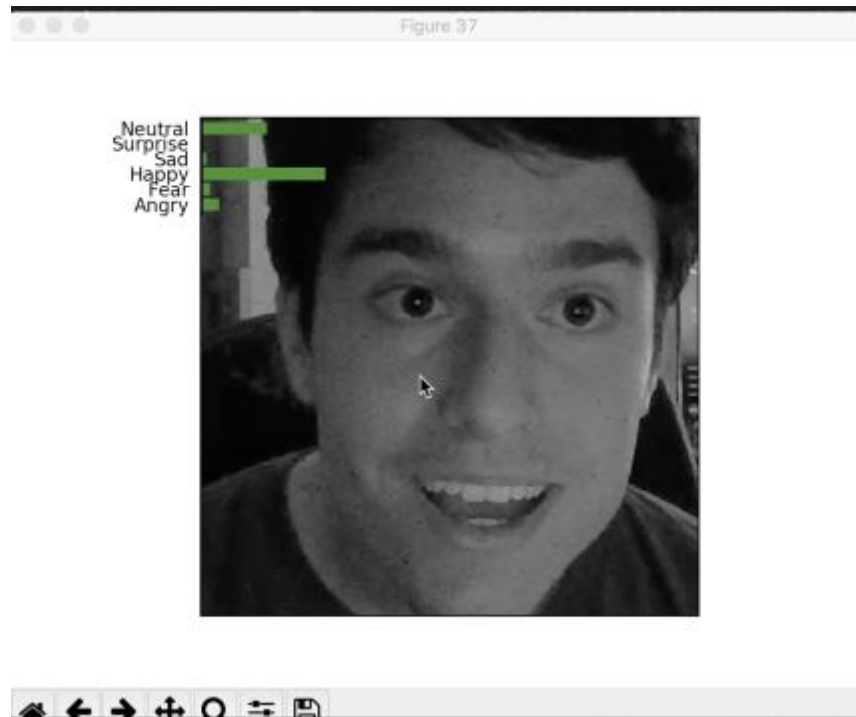


Figure 16

But here is a sample image from the webcam stream of my face and the predictions the model made in Figure 17:

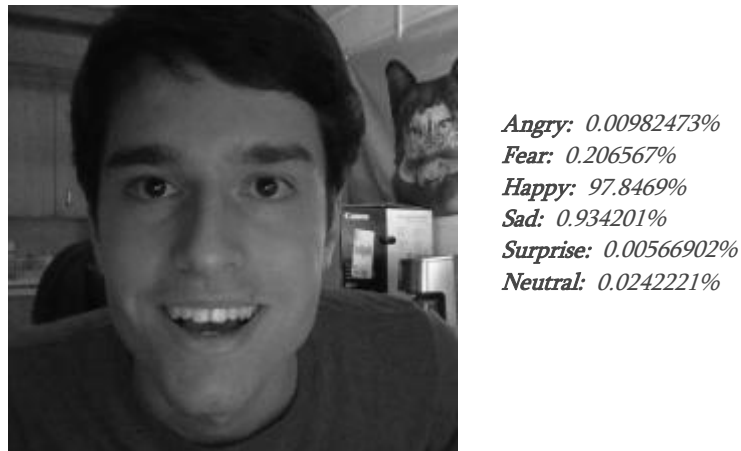


Figure 17

As shown, the model is very confident that I am happy, with a confident of almost 98%.

Reflection

In general, the main steps I took in this project were:

1. Researching APL Units and Fractional Max Pooling and implementing them in Keras
2. Preprocessing the data
3. Experimenting with different model architectures gaining inspiration from successful models like VGG-19 and AlexNet
4. Training my model(s) on a p2.xlarge instance on AWS
5. Training my Udacity image classification model as a benchmark
6. Comparing my benchmark model to my new model
7. Extending the model to using any webcam

Overall, I found the research most interesting. Learning about these new innovations in machine learning that still isn't really known by many people is interesting. Also, trying to understand the proofs of these innovations and convincing myself why I should use them instead of, for example, PReLU and regular max pooling, makes me keep understanding new things in a field that is still growing. And having to write the code for the APL activation function myself because of how new it is and not finding any implementation of them online in Keras yet was also enjoyable as I had to kind of become the expert of my own domain.

The part I found most difficult would have to be training the new models by tuning hyper parameters and also creating the webcam app. Training was difficult solely because of the time it took. I absolutely enjoyed being able to use a p2 instance to train my models as it made training super quick (and because it was cool knowing I had a 12GB video card under my fingertips!). Tuning hyper parameters and figuring out what makes a great CNN architecture great for certain datasets is still an active place of research; thus, figuring out the best model I could possibly make consisted of a lot of guessing and checking. I at least spent 17 hours at my computer tuning my model (a lot of that time was spent doing other things though as I had to wait for the model to train which took around 30 minutes for 30 epochs). Also, creating the webcam was difficult as I had trouble figuring out how I can take the picture from my webcam, grayscale it, and downscale it to one channel. I had bugs, for example, that resulted in images having all their colors inverted or only the green and blue channels showing and the red channel not being there. But I managed to get all those bugs sorted out thankfully.

Improvement

To achieve the highest accuracy, I would try looking for other datasets out there built for emotion classification models. I know they do exist out there, but I didn't want to use it for this project so it wouldn't make my Kaggle leaderboard comparison for this dataset more valid.

Also, instead of creating Pyplots for each frame passed through my model, I would try and maintain only one as the longer you run the webcam python script, the slower my system and thus the predictions get. I'm guessing this is because of the 80 or so Pyplots that are being displayed at the same time.

References

- [1] LeCun, Yann, Patrick Haffner, Leon Bottou, and Yoshua Bengio. (1998). *AT&T Shannon Lab*. Retrieved June 26, 2017, from <http://yann.lecun.com/exdb/publis/pdf/lecun-99.pdf>
- [2] Graham, B. (2015). Fractional Max-Pooling. *Dept of Statistics, University of Warwick*. Retrieved June 23, 2017, from <https://arxiv.org/abs/1412.6071>.
- [3] Springenberg, J., Dosovitskiy, A., Brox,, T., & Riedmiller, M. (2015). *Department of Computer Science, University of Freiburg*. Retrieved June 23, 2017, from <https://arxiv.org/abs/1412.6806>.
- [4] Agostinelli, F., Hoffman, M., Sadowski,, P., & Baldi, P. (2015). Learning Activation Functions To Improve Deep Neural Networks. *ICLR 2015*. Retrieved June 27, 2017, from <https://arxiv.org/pdf/1412.6830.pdf>.
- [5] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *Microsoft Research*. Retrieved June 27, 2017, from <https://arxiv.org/pdf/1502.01852.pdf>.
- [6] Simonyan, K., & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Sacle Image Recognition. *Department of Engineering Science, University of Oxford*. Retrieved June 27, 2017, from <https://arxiv.org/pdf/1409.1556.pdf>.
- [7] Krizhevsky, A., Sutskever, I., Hinton, G., (2012). ImageNet Classification with Deep Convolutional Neural Networks. *University of Toronto*. Retrieved June 27, 2017, from <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>