Efficient Facial Feature Location with Various Neural Network Architectures

Hans B. Gronewold
hans@blackelectricity.com

Abstract

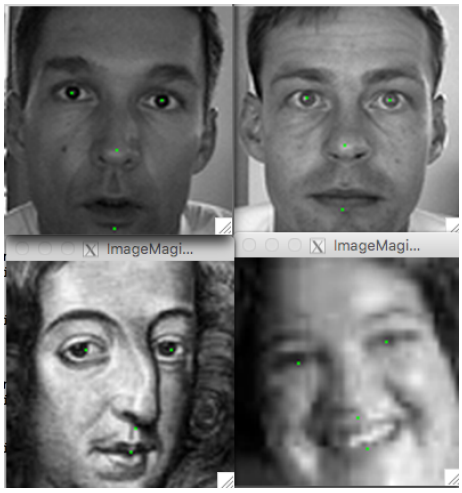
In this paper I examine different architectures for improving the runtime performance of locating facial feature landmarks using neural networks. Models which maintain satisfactory accuracy levels but also reduce computational complexity and improve practical running speeds are valuable. There are many opportunities to implement fast and cheap machine learning functions on resource constrained compact mobile devices within modern applications that reserve the majority of resources for other processes. In these contexts, it is paramount that the models optimize for runtime performance and minimize memory utilization and bandwidth. I review various non-standard models to maintain accuracy but reduce network complexity. These custom models include a cascading network, a single network with weighted voting, wide shallow convolutional networks, and a network with a skipped layer using the histogram of gradients. Finally, I show how an extremely efficient, simple, and fully connected neural network achieves satisfactory accuracy and performance.

1 Introduction

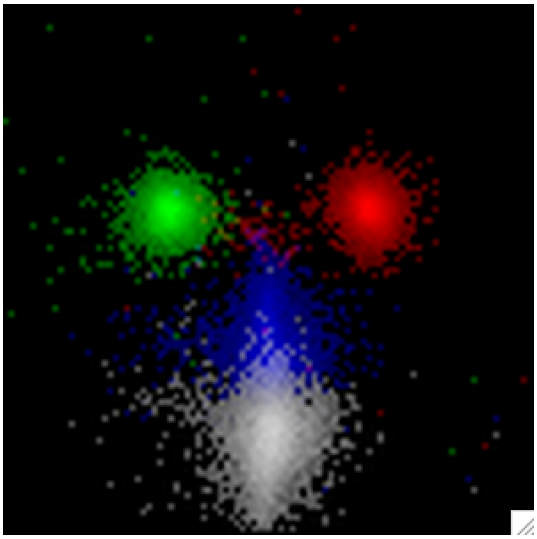
The brain efficiently organizes visual input as signals which discriminate between objects of the same category through minute characteristic features *(*2)*. Specifically, small relative spacial distances between facial feature landmarks allow humans to perform fast facial recognition and many other types of abstract ideation, such as emotive cognizance *(*4)*. Neural networks are already known to perform well on locating these landmarks *(*3)* and are commonly implemented in many platforms and hardware. I constrain this paper's model exploration to neural networks. They have the current distinct advantage of being easily implemented in widely distributed consumer electronic devices which often already include high-speed GPGPU hardware and software frameworks which easily integrate trained network models directly and at impressive performance rates with no additional distribution costs *(*5)*. Because of this ubiquity, for generic third-party applications, software development and deployment costs for neural network models are much lower when compared with many other state of the art machine learning models. Most mobile cameras and many comedic image and video applications, like face filters and face swapping, already widely implement or utilize toolkits and hardware for these models *(*5,6)*. Sate-of-the-art models in the sole pursuit of higher accuracy utilize larger and slower deep learning techniques which are costly. Even simpler methods in open libraries are too costly on modern graphically intensive consumer applications that run on cheaper mobile devices with limited resource allocation *(*7)*. I investigate various smaller and shallower neural networks as alternatives to deeper and larger networks. I examine easily deployable neural network architectures which improve runtime performance while only nominally lowering the final prediction accuracy.

2. Background Information

A. The face images and labels in the ancillary dataset from kaggle.com's facial feature challenge *(*8)* are already preprocessed into grayscale and centered face images of size 96 by 96 pixels, similar to data that can be easily obtained from the interior of a bounding box after running a fast facial detection sliding window on an image. Preliminary facial bounding box detection is not in scope of this paper and is often implemented efficiently via hardware. The dataset includes faces at various angles and types, including drawings, blurred images, partially clipped at various angles, faces of the same person with different expressions, and with varying backgrounds. Here are a few example faces:



B. The data are somewhat noisy and contain a few odd examples and mislabelled faces. These training samples are not removed from the dataset as they will help to show which experiment models are more resilient to bad training input. There are 7049 training set examples, however, only 2140 are not missing features. I separate the dataset into two feature sets as explained below *(4-A)*. Explained in Section 4-A; by separating the target features into two groups, I create target feature Set A, with 7000 full training records for the eyes, nose, and mouth that have no missing features. Here is an example showing a few of the outliers from set A, as well as a heat-map of the locations. (This heat-map was created with a fractional exponent of 0.2, so the single outliers still show up well. Almost all of the data lands in the brighter central areas though.)



C. I normalize all of the target x and y locations by dividing the original values by 96 and subtracting by 0.5 to normalize the data to the range [-0.5, 0.5] *(3-A-II)*. I additionally normalize all of the dataset images by subtracting the original minimum, dividing by the new maximum, and then finally subtracting the resulting mean value, to the resulting range [-1,1] and of 0 mean for each image *(3-A-I)*. When any input training data is resized or adjusted beyond this normalization, the adjustments are described with each of the model experiment details below.

D. The models were all trained using gradient descent with MSQE loss *(3-C)*. Typically, AdamOptimizer usually settled at the lowest validation error and it is chosen for accuracy not speed, as training time is inconsequential to the goals of this paper, and therefore I give all experimental results for the runs which utilized only AdamOptimizer gradient descent optimization method *(*1ab)*. Experiments with other optimization algorithms were not run for all models and are not included in this paper at all. The RMSQE value will serve as the performance benchmark which I try to keep at a sufficiently low value while trying to make the model smaller and faster.

E. All convolutional layers use a step size of 1, horizontally and vertically. The filters are centered over each pixel of the image, so the resulting convolutional output width and height are equivalent to the input width and height, but add depth based on the # of filters. I regularize with dropout between the convolution layer and the pooling layer. Regularization after pooling resulted in lower validation scores. Every convolutional layer in this paper is followed by a max-pooling layer which use a 2x2 filter and 2x2 stepping to shrink the original convolutional output width by a factor of 2 and the height by a factor of 2. After the max-pooling layer the resulting output is 4x smaller than the values generated by the convolution filters alone.

F. The increased accuracy with the hyperbolic tangent *tanh(z)* activation function *(3-D)* outweighed any non-training time speed improvements of the ReLU *max(0, z)* and LeakyReLU *max(z/r, z)* activation functions, because the network sizes were larger using the linear rectifiers in order to achieve the same accuracy levels and in some cases failed to reach the same accuracy. This is a transient and subjective observation only, as the supporting data for this was not recorded for this paper, however this information is included mainly as background to the decision to standardize test results with the hyperbolic tangent function. All of the details, scores, and learning curves in this paper use only the hyperbolic tangent. Further investigation into this area, including differing the weight initializations and experimenting with activation functions are not explored within this paper. Hyper-parameters were adjusted through manual grid search. Also, for this particular paper's experiment results, I initialize all network weights and biases randomly with a normal distribution having a mean of -1E-6 and a standard deviation of 1E-2.

G. All learning curve graphs have the vertical axis of RMSQE and the horizontal axis of values that are batch size * p, where p is the number of mini-batches to wait between running the validation scoring and printing/saving the resulting values. This is generally somewhere between 1/2 to 2 epochs, and is labelled as epoc, despite not being exactly one epoch. All RMSQE values are given on the 20% validation set unless otherwise exclusively specified as the final kaggle.com test RMSQE score. Some of the learning curve graph error axes are also mislabelled as MSQE, however these graphs should still be correctly interpreted as the average RMSQE error of the output features. I did not re-adjust these graphs to be exactly 1 epoch for each model since the curves for these models were utilized solely to inspect between bias, variance, overfitting, etc…, and the training/learning time optimization is irrelevant to the end goal of this paper.

H. I implement early stopping by only saving the model during training when the cross validation score reaches a value lower than its previous score during training. After the patience window is finished the training is automatically stopped.

I. I show training vs. cross-validation curves which I saved for some of the models *(5)*. These graphs show the models running with regularization *(3-B)* as high as possible without negatively impacting the validation score. Regularization cost *(3-B)* is not included in the learning curves as shown or the RMSQE values given.

3. Equation Definitions

A. Dataset Normalization.

I. The image values are normalized as follows:

$$X \leftarrow X - min(X)$$
$$X \leftarrow X / (max(X) + \epsilon)$$
$$X \leftarrow X - mean(X)$$

II. The target features are normalized as follows:

$$Y \leftarrow ((Y/96.0) - 0.5)$$

B. Regularization

I apply dropout regularization post convolution filters and prior to max pooling. The weights and biases are additionally regularized over the following equation:
(*l = layer count/index, j = weight count/index, i = bias count/index, w = weight, b = bias*)

$$Reg. = \frac{\sum_1^l (\frac{1}{j} \sum_1^j w_{lj}^2 + \frac{1}{i} \sum_1^i b_{li}^2)}{2 * l}$$

C. Objective Function (*MSQE*)

The model is trained with the AdamOptimizer gradient descent optimization method *(*1ab)*. The x and y pixel position of each feature is the target final output. The objective function minimizes the Mean Squared Error between output and truth. This penalizes the very bad outliers more than the mean absolute error, and also penalizes large weight values and subsequent overfitting:

$$MSQE = \frac{1}{n} \sum^n (Y_i' - Y_i)^2$$
$$Minimize : MSQE + \beta * Regularization$$

D. Activation Function (*tanh*)

The hyperbolic tangent is used as the activation function for each non-pooling layer.

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

E. Root Mean Squared Error

The final graphs display the RMSQE scaled back up to 96 pixels:

$$RMSQE = 96 * \sqrt{MSQE}$$

4. Experimentation with Various Models

A. Dataset Preparation and Cleansing

I normalize all the data per the equations previously outlined. Also, the dataset contains 2 distinct sets of data with different target features and I split these into separate datasets for training, remove rows with missing features, and refer to them as follows:

Set F - (full training set, no split) [2140 clean examples without any missing labels]
Set A - (eye centers, bottom of nose, bottom lip) [7000 examples without missing labels]
Set B - (eye corners, eyebrow endpoints, lip corners, top lip) [2155 clean examples.]

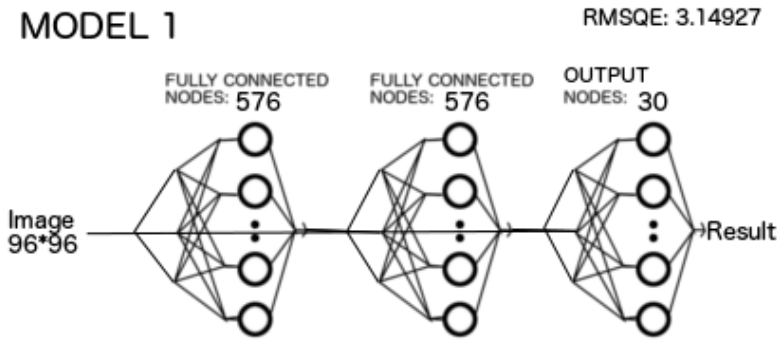
I separate the dataset into 80% of examples for training and 20% for validation.

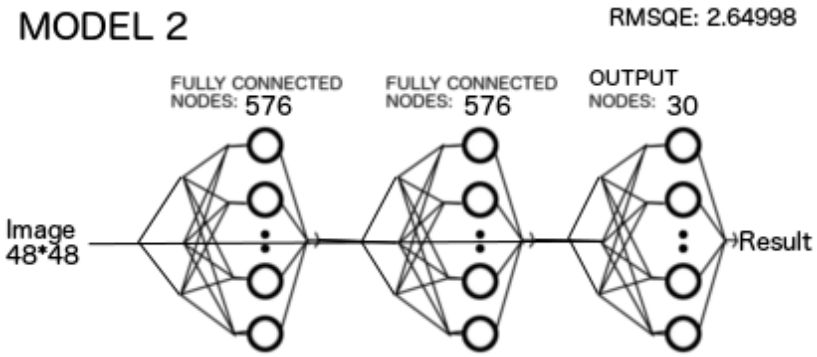
B. Joint Facial Landmark Regression

Typically, to improve accuracy, many state-of-the-art methods apply separated regressions (or in some cases multiple weighted and separated regression models) to each and every target value. I do not separate the features in these experiments (other than splitting the training data due to distinct datasets). My goal is to create a small and very fast network that is secondarily capable of sufficient accuracy.

C. Feature Reduction

To further reduce model size and speed, I reduce the input space to smaller set.
Initially I compare the benefit of reducing the input space over Set F by resizing the Image:

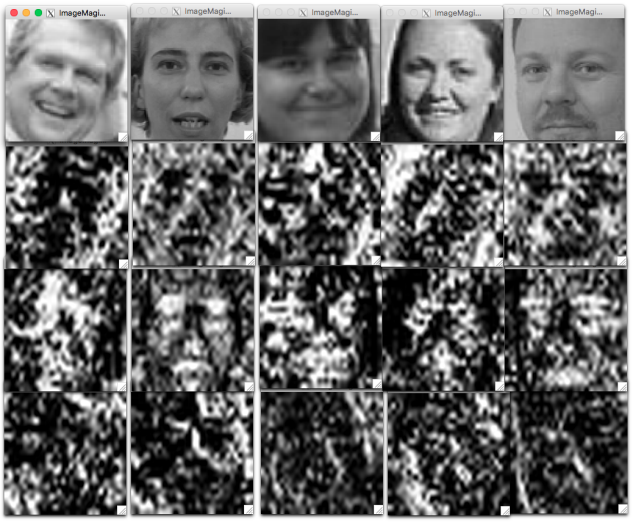


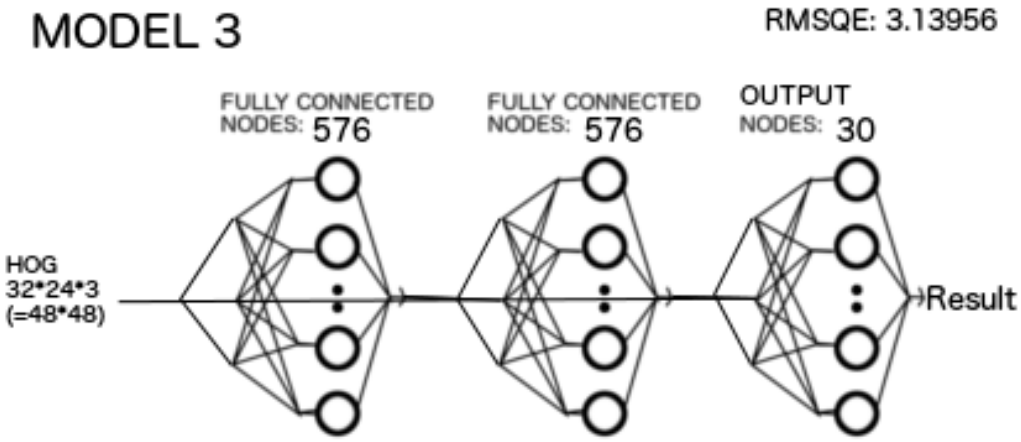


Here is a preview of MODEL 2 predictions shown on a few of the validation dataset images:

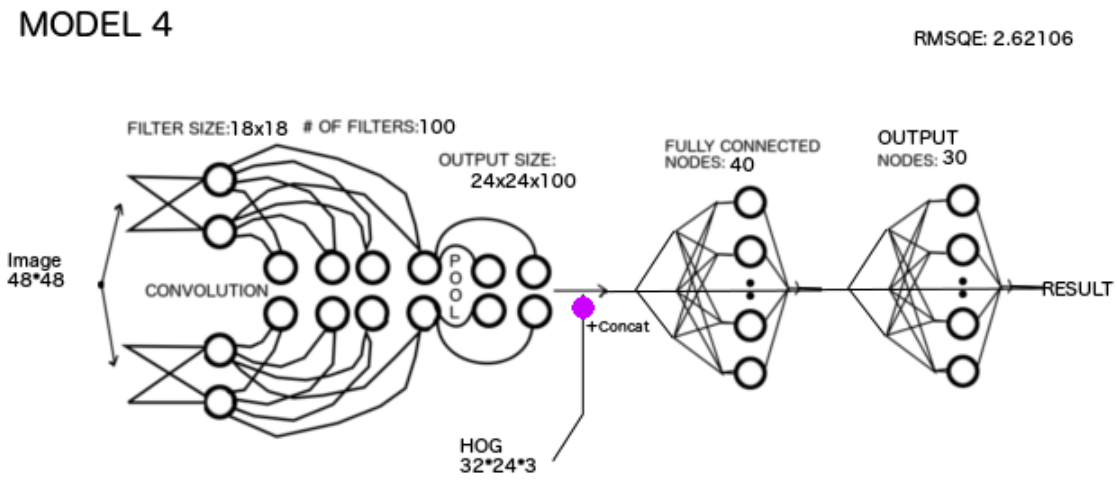
Base truth – Blue, Prediction – Green

I attempt to see if the Histogram of Gradients provides more information than the shrunken image, which it does not in this initial experiment. I create the HOG feature space using only 3 gradient directions and filters of size 3 by 4:



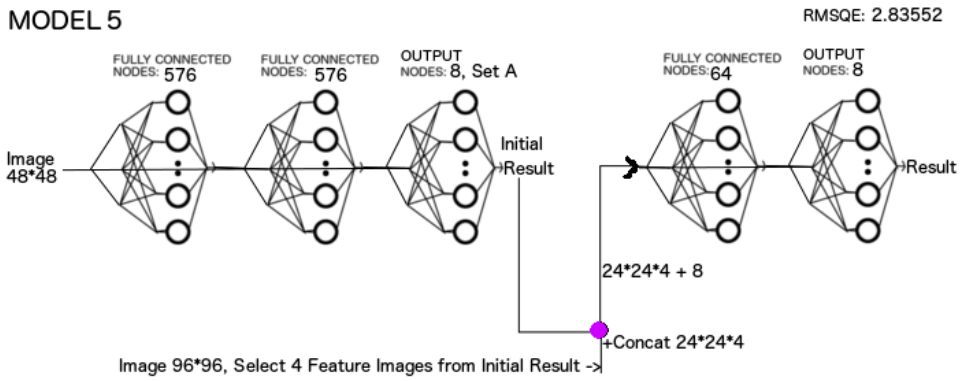


Also, I try another architecture, out of curiosity, which marginally improves on Model 2 but isn't worth the added performance cost. I include an initial convolutional layer and then concatenate the HOG values with the max pooling output before connecting to the first fully connected layer:

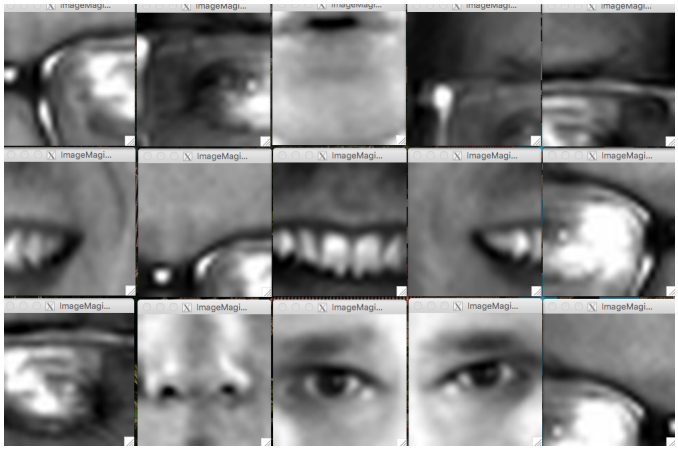


D. Custom Cascading Neural Network

I create a custom neural network cascade, just out of curiosity, not in the goal of runtime performance, (This splits features up, but here I only use Set A so there are only 4 (x,y) pairs to cascade). From the initial prediction, another image is created by cropping the original image by 24x24 around the initial predicted position, and a smaller network tries to correct the initial features using only the new smaller portion created from the initially predicted network. This model is both more costly and less performant than using only the original estimates:

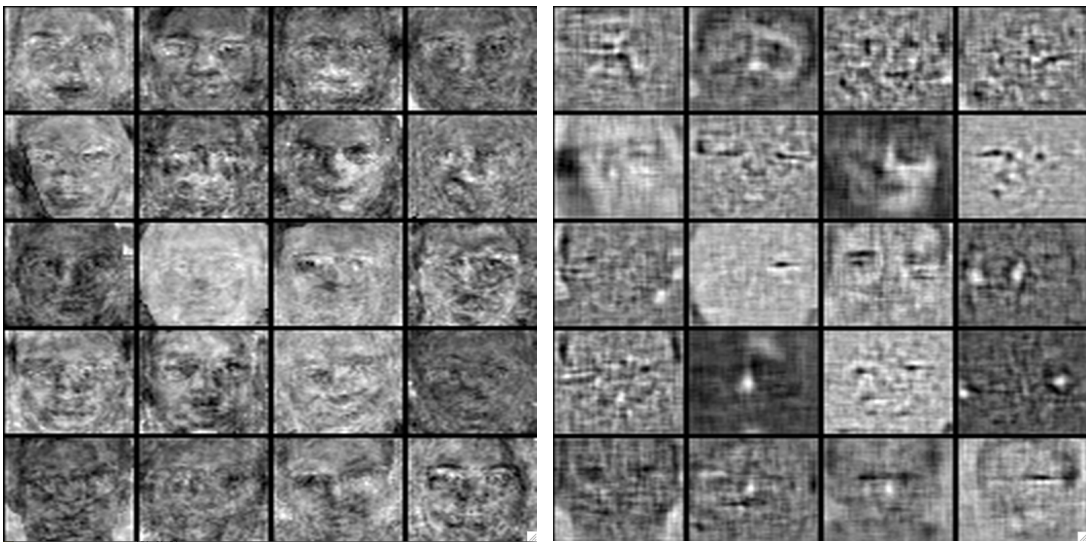


Here is a preview of the secondary images concatenated onto the second part of the cascade above:

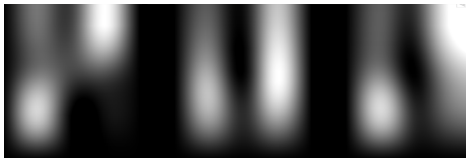


E. Visualizing Convolution Filter Weights

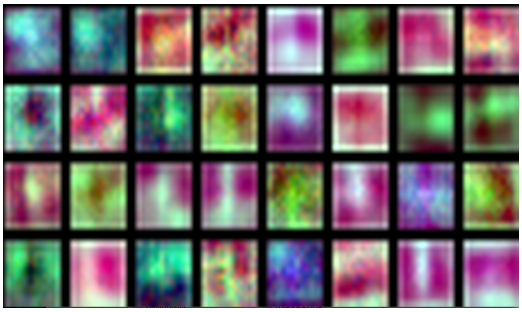
I tried initializing weights with images of faces, which did not help the accuracy score. Here I visualize a few of the convolutional layer filter weights after training is completed:

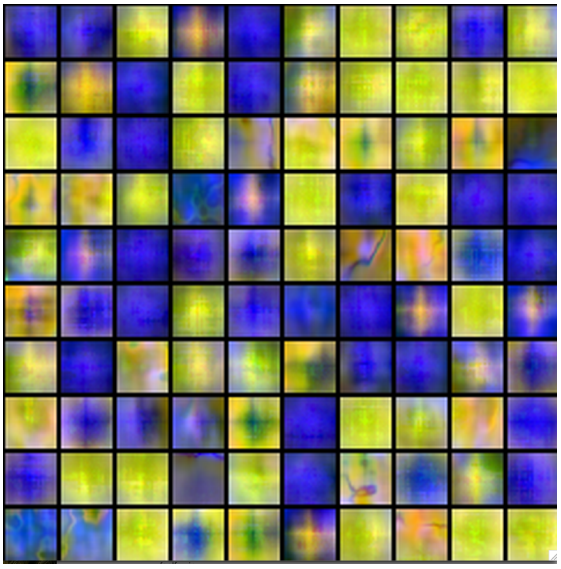


I also ran the networks with several different sized convolution layers to visualize the weights. Also it turns out that the wider 1 layer convolutional networks performed well. I initialized the first layer to have a depth of just 3 so that I could visualize a second layer's filter weights in color. Here is the initial layer after training:



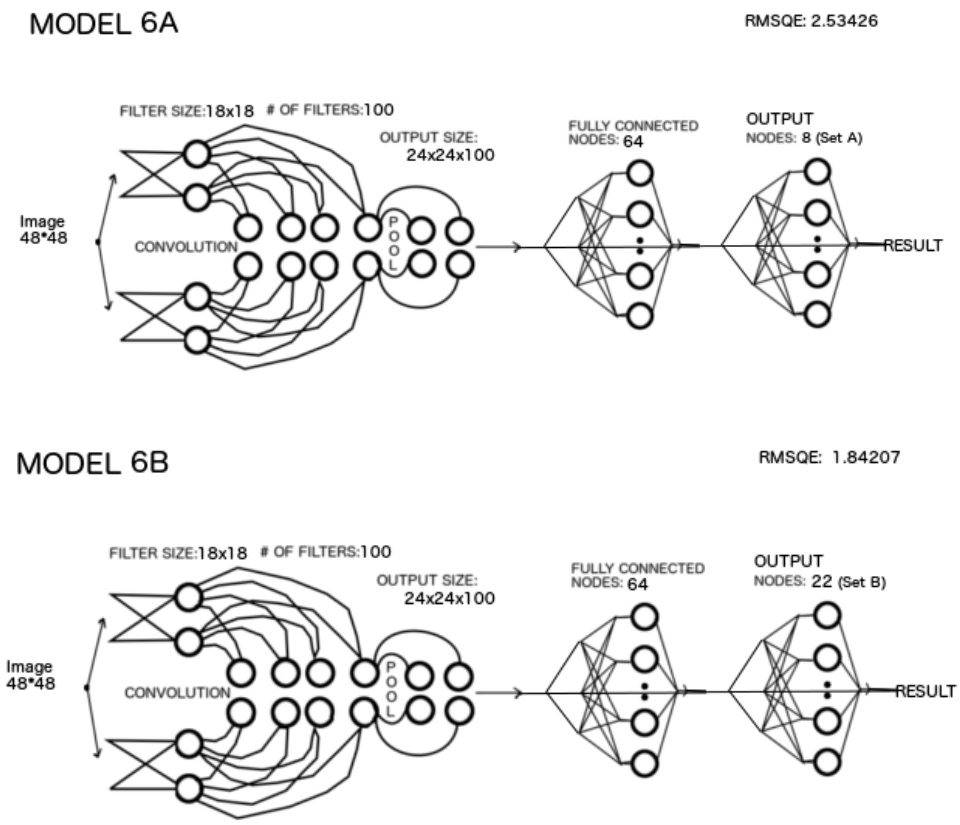
Here are a couple interesting second layer weight visualizations from different filter size training runs. The secondary layer color values comes from an RGB value for each of the 3 input layers from the filters above:





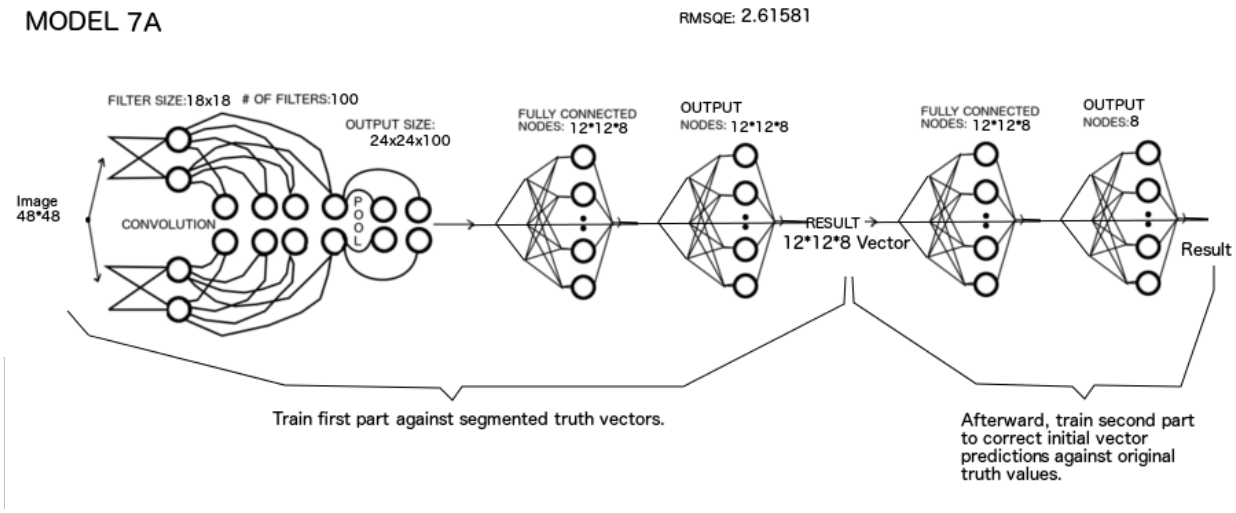
F. Wide Convolutional Neural Networks

One of the somewhat more accurate, but costlier, convolutional models, was a single conv. layer with wide filters and high depth. Full RMSQE = 2.02665, Kaggle.com test RMSQE = 2.7793:

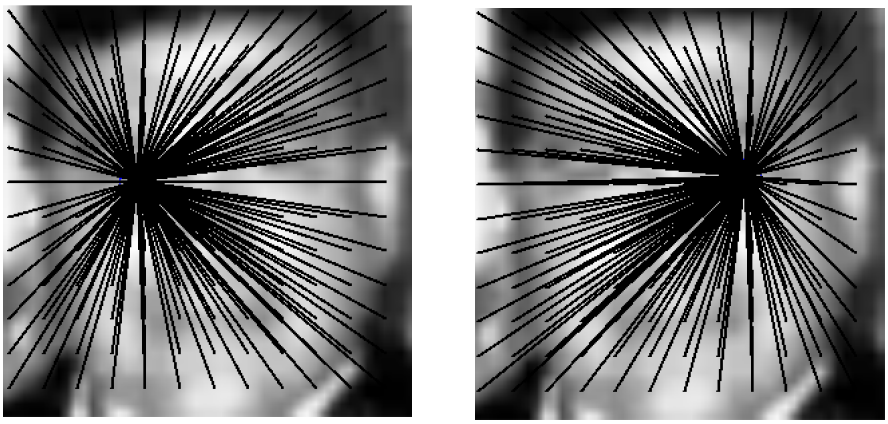


G. Custom Weighted Voting Using a Single Network

I also run an a different architecture which does not improve the accuracy. Inspired by ideas related to hough space transformations and weighted voting of multiple nets, I setup an experiment with the initial 4 x,y target features for Set A transformed as 8 new layers with each output value targeted as 12x12 different distance vectors of that dimension from each section of the image. So initially the network creates 12x12 regression values for each target value. Then a second network learns to weight these votes to get back down to 8 final values. I wanted to see if this would improve the score like weighted voting of different networks would, but using a fewer number of weights than training multiple networks for every single feature. However, this didn't improve scores for Set A, so Set B/F was not run, and I abandon the model as it is also too complex.

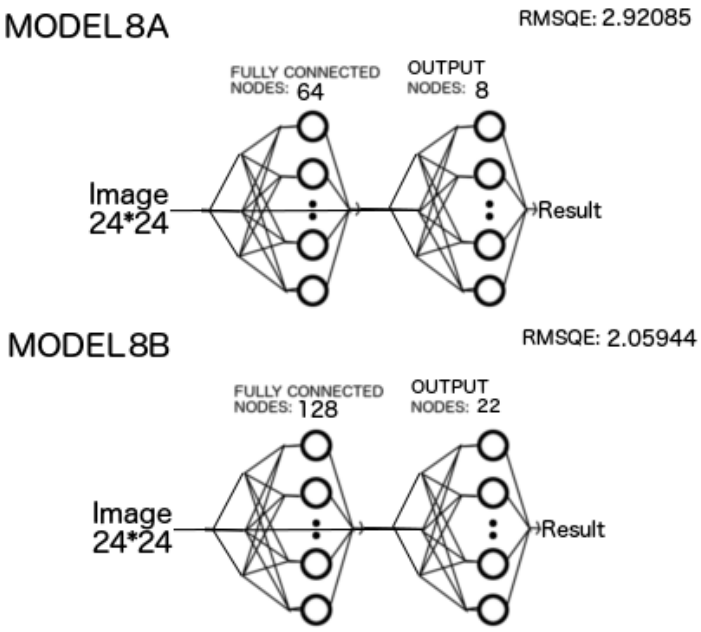


Here is the altered intermediate values layer, for the target truth values, as vectors for two eye features. Each segment of the image is shown as a vector for the corresponding 2 layers per image. Each image has 1 target feature's x and y value layers, for every section, visualized as vectors:



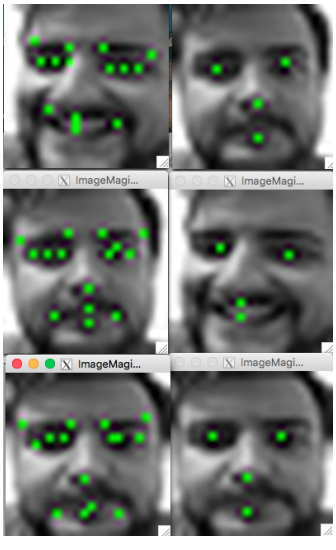
H. Conclusion / Simple Fully Connected Networks

Finally, as the optimal model, I show a simple, fast, and fully connected architecture with satisfactory accuracy. First I scale the input image to 24 x 24 pixels. Model 8A, on its own, could be used in the graphical pipeline to create a very fast feature detector that has enough information to make something like a rough face filter mobile phone application. A small network like this is extremely fast.



Full validation set RMSQE: 2.28914 and Kaggle.com test RMSQE = 3.5978. This is fairly good accuracy for a lightning quick 64 node single hidden layer network! Sometimes the simplest solutions end up working the best. The final model has a learning curve that still shows some variance even after regularization was increased as high as possible. Additional data to reduce that variance and removing the erroneous training images would likely improve the 8A model score even further, but it is sufficiently good already. This simple and very fast network performs at an accuracy high enough to create cool end-user applications like face filter apps on a mobile phone. Here is a preview at how well the trained model performs on totally unseen images of myself:

(The left column shows all A&B features.
The right column are Set A features only.)

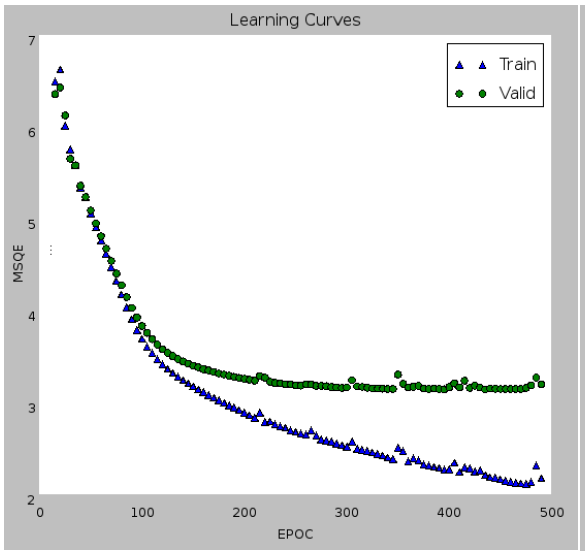


In this table, I compare RMSQE and model sizes. Note: all models shown have sufficient RMSQE score, so the final model is preferred because it is the smallest and fastest running, while having a sufficiently low RMSQE:

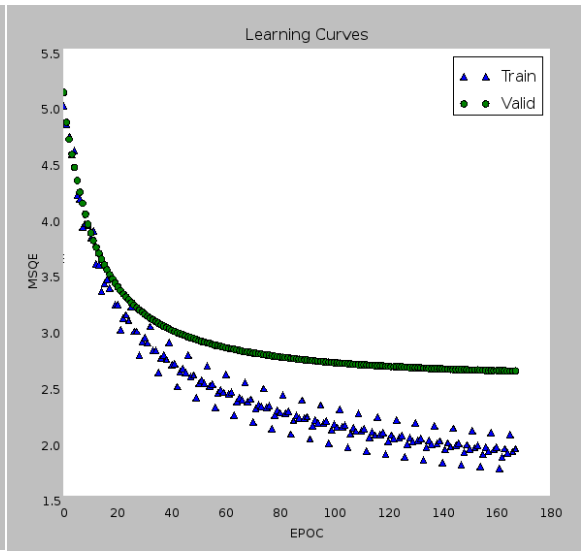
MODEL	DESCRIPTION	SIZE	RMSQE
2	Shrink	1.7E6	2.64998
3	Hog	1.7E6	3.13956
4	C+Hog	7.7E7	2.62106
5A	Cascade	1.9E6	2.83552
6A	Wide	7.8E7	2.53426
7A	Voting	1.4E8	2.61581
8A	Simple	3.7E4	2.92085

5. Learning Curves

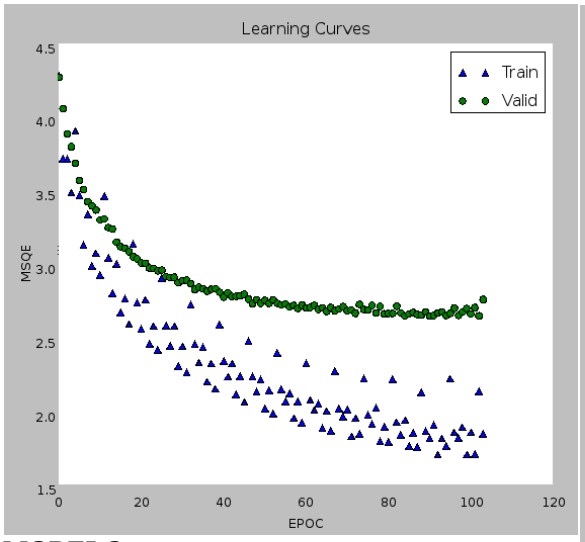
MODEL 1:



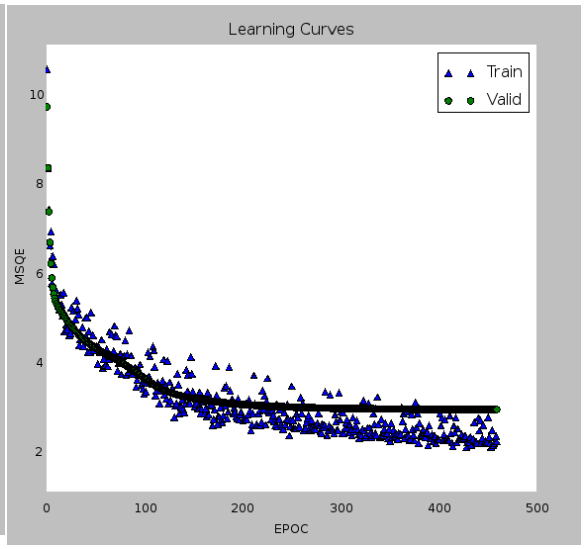
MODEL 4:



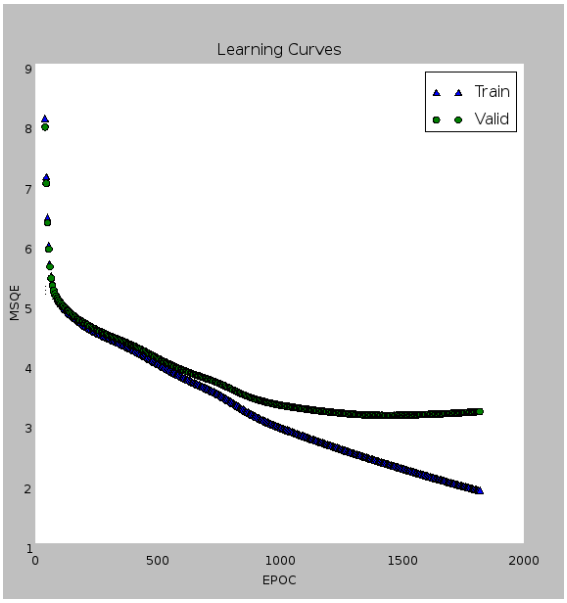
MODEL 2:



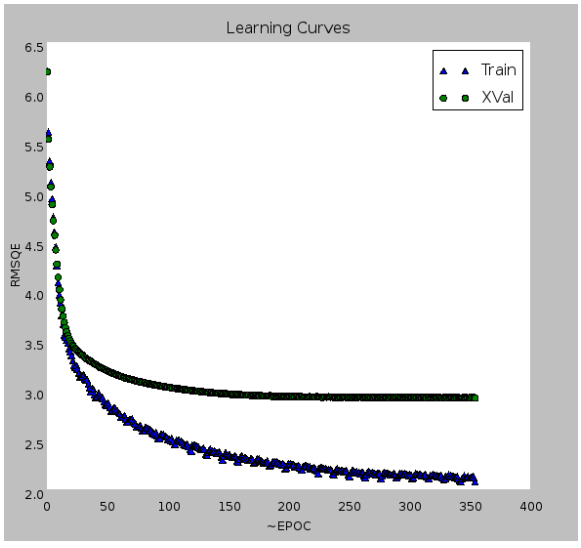
MODEL 5:



MODEL 3:



MODEL 8A:



6. Supplemental Code

Final working code for Models 6AB and 8AB are inside the GitHub associated with this paper. **https://github.com/blackelectricity/FaceyMcFaceFace** *(*9)* The remainder of the code is not in a cleansed/refactored state, and those pieces can be found in the *trash* folder of the github zip file.

* References

1a. Kingma, D. & Ba, J. (2014) Adam: A Method for Stochastic Optimization. https://arxiv.org/abs/1412.6980

1b. Tensorflow API Documentation for AdamOptimizer https://www.tensorflow.org/versions/r0.11/api_docs/python/train.html#AdamOptimizer

2. Tsao, D. Y., & Livingstone, M. S. (2008) Mechanisms of face perception. Annual Review of Neuroscience, 31, 411–437. http://doi.org/10.1146/annurev.neuro.30.051606.094238

3. Ranjan, R., et al. (2016) An All-In-One Convolutional Neural Network for Face Analysis. http://arxiv.org/abs/1611.00851

4. Ghimire D. & Lee J. (2016) Geometric Feature-Based Facial Expression Recognition in Image Sequences Using Multi-Class AdaBoost and Support Vector Machines. https://arxiv.org/abs/1604.03225

5. A few mobile NN tools: https://www.khronos.org/opencl/,http://memkit.com,http://opencv.org

6. e.g. Visage Technogologies: http://visagetechnologies.com/face-detection-tracking-chips/

7. http://www.learnopencv.com/facial-landmark-detection/

8. https://www.kaggle.com/c/facial-keypoints-detection/data

9. https://github.com/blackelectricity/FaceyMcFaceFace