

## Problem Description

The competition undertaken is the Kaggle-hosted 'Facial Keypoint Detection'. This description taken directly from the site (<https://www.kaggle.com/c/facial-keypoints-detection>) is:

"The objective of this task is to predict keypoint positions on face images. This can be used as a building block in several applications, such as:

- tracking faces in images and video
- analysing facial expressions
- detecting dysmorphic facial signs for medical diagnosis
- biometrics / face recognition"

Simply put, given some images of people's faces, can one automatically determine the location of certain features on these images? The goal of this competition is to identify up to 15 keypoints per given test image. [One application of this is the new iPhone security face-recognition method.] These features include eye, eyebrow, nose, and mouth locations.

## References

Esmaeili, Khosravi, & Mirjalili. (2015). Facial Keypoint Detection. Retrieved from [https://pdfs.semanticscholar.org/372c/6af4ca170ee2ca9e926cbb7d8f71aa171ec5.pdf?\\_ga=2.111733508.1885195983.1521399667-931019125.1521399667](https://pdfs.semanticscholar.org/372c/6af4ca170ee2ca9e926cbb7d8f71aa171ec5.pdf?_ga=2.111733508.1885195983.1521399667-931019125.1521399667)

Zhang, & Meng. (2015). Facial Keypoints Detection Using Neural Network. Retrieved from [http://cs231n.stanford.edu/reports/2016/pdfs/007\\_Report.pdf](http://cs231n.stanford.edu/reports/2016/pdfs/007_Report.pdf)

Agarwal, Krohn-Grimberghe, & Vyas. (n.d.). Facial Keypoint Detection using Deep Convolutional Neural Network. Retrieved February, 2018, from <https://arxiv.org/pdf/1710.00977.pdf>

Longpre, & Ajay. (2016, March). Facial Keypoint Detection. Retrieved February, 2018, from [http://cs231n.stanford.edu/reports/2016/pdfs/010\\_Report.pdf](http://cs231n.stanford.edu/reports/2016/pdfs/010_Report.pdf)

<https://www.kaggle.com/c/facial-keypoints-detection>

<https://keras.io/>

The reference documentation researched describes many methods of approaching this type of problem. In some cases, it discusses and evaluates methods while engaging in this very Kaggle competition.

The prime takeaway from here is that convolutional neural nets are a legitimate method for image detection. And, in most cases, appear to be the current best method available. Based on this finding, the focus of this report will be using convolutional nets.

## Data Description

Two datasets are provided: a train and test set. The train set has 7049 images with keypoint locations identified for each feature (two coordinate measurements, given as 'x' and 'y'). The test set has 1783 images, used for prediction/scoring via the host site.

The keypoint features and the non-null counts of each are in the following table:

left eye center x	7039	left eyebrow outer end y	2225
left eye center y	7039	right eyebrow inner end x	2270
right eye center x	7036	right eyebrow inner end y	2270
right eye center y	7036	right eyebrow outer end x	2236
left eye inner corner x	2271	right eyebrow outer end y	2236
left eye inner corner y	2271	nose tip x	7049
left eye outer corner x	2267	nose tip y	7049
left eye outer corner y	2267	mouth left corner x	2269
right eye inner corner x	2268	mouth left corner y	2269
right eye inner corner y	2268	mouth right corner x	2270
right eye outer corner x	2268	mouth right corner y	2270
right eye outer corner y	2268	mouth center top lip x	2275
left eyebrow inner end x	2270	mouth center top lip y	2275
left eyebrow inner end y	2270	mouth center bottom lip x	7016
left eyebrow outer end x	2225	mouth center bottom lip y	7016

Note that there all but two keypoints (nose tip identifiers) have some nulls embedded in the dataset.

The images are coded as a list of numbers. When converted to numpy arrays, they look like the following:

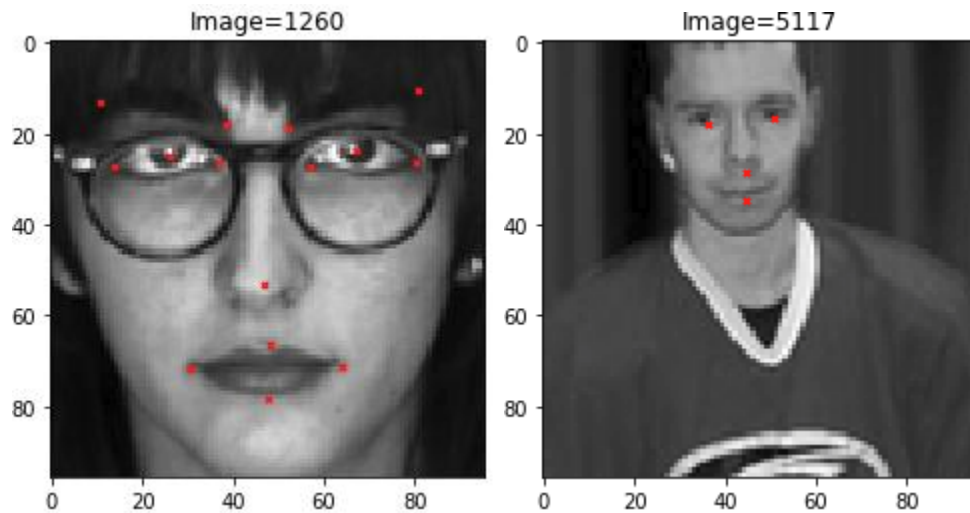
```
0    [238.0, 236.0, 237.0, 238.0, 240.0, 240.0, 239...
1    [219.0, 215.0, 204.0, 196.0, 204.0, 211.0, 212...
2    [144.0, 142.0, 159.0, 180.0, 188.0, 188.0, 184...
3    [193.0, 192.0, 193.0, 194.0, 194.0, 194.0, 193...
4    [147.0, 148.0, 160.0, 196.0, 215.0, 214.0, 216...
```

The images are grayscale (i.e. no color). The range of values is [0,255].

The location data is on a scale of [0,96].

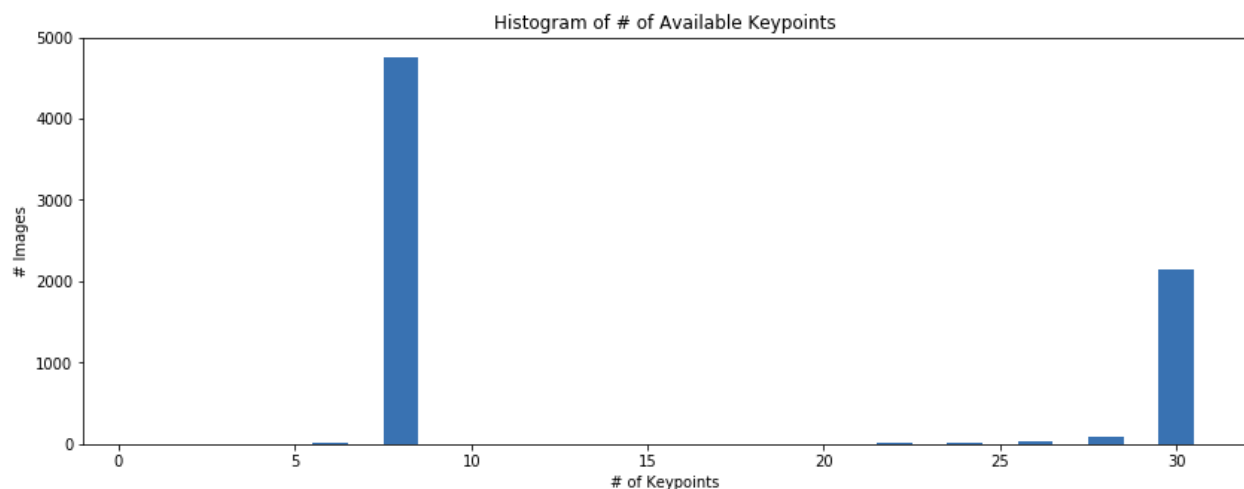
Since it was determined at an early stage that neural net models were to be utilized, scaling of the data was done to best accommodate this type of algorithm. The image data was all divided by 255 to put it on a scale of [0,1]. The location data was centered about zero and scaled to a range of [-1,1] by  $y=(y-48)/48$  for each value.

Two fairly representative examples of the data provided in the train set:



The red dots on each are the keypoint locations. The first has all keypoints identified and in apparent good status (i.e. they look accurate). The second image, though, is an example of much of the data where there are missing / null values present. Only four locations are identified, and the laccuracy of each appear to be somewhat suspect.

It seems apparent that there are two different image data sets that were combined to make up the train data set for this competition. Each of these two examples are typical of each: a dataset with 15 keypoints and one with nominally only 4 keypoints given. In general, the 15 keypoint images appear to be of higher quality – not just having all the features identified, but the images themselves being consistently better. They are mostly straight on (that is, generally not side-views), centered in the 96x96 area, and have a more consistent zoom level. The 4 keypoint group has much more variance in these factors.



Per the chart above, the 4 keypoint data group significantly outnumbered that of the full 15 keypoint group in the train set [note: chart above counts both x and y location as separate keypoints].

The keypoint locations are generally located appropriately relative to each other. There are a number of scatter plots of the keypoints in the appendix that show the general spatial relation of each feature relative to each other. Outside of some non-centered images, there does not appear to be any significant issue with the data [i.e. no left/right improper swapping for any feature].

The only consistent issue found with the data during exploration is that in the 4 keypoint grouping, both the nose tip and bottom lip locations both were consistently off. The nose was always lower than it should be (as if it was indicating the base of the nose) and the lower lip was always higher than what it should be (like the lip/teeth intersection rather than the lower 'outside' portion of the lip).

## Data Cleaning

Beyond the scaling, nothing additional was done at this time to the data. However, there is some additional modification done at a later state that will be more appropriately discussed in the Modeling section.

## Modeling

As indicated above, the intent is to utilize neural nets (NN) as well as convolutional neural nets (CNN) for keypoint prediction. While virtually any regression method can be used, the intent of selecting this project was to specifically become better versed in these types of modeling projects.

All modeling completed in Python (utilizing the Jupyter Notebook), with Tensor Flow and Keras being utilized.

For the initial models, a simplified version of the training data was used, specifically, all NaN (nulls/missing values) were dropped from the data set. This is not good practice in general, but the intent is to test out the loading of data as well as initial model creation to ensure that the process does work without error [this is the first time that both Keras and Tensor Flow were used]. Dropping the NaN resulted in 2140 records being available.

### *Model1 – NN*

A simple neural net model was chosen initially. Some parameters for the model:

- A single initial dense layer of 100 nodes
- RELU activation function
- Output dense 30 node (15 keypoints with x,y locations for each, so 30 total)
- Mean squared error for loss calculation

- 80/20 split for validation data
- Stochastic Gradient Descent for backpropagation

The model ran correctly and results can be seen in the compilation of prediction results. While perhaps not a great model, the proof of concept is the desired result. Examples of prediction results can be found in the compilation table at the end of this section.

### *Model 1 – CNN*

After successful completion of the base NN model, a CNN model was created. Details of the model can be found in the appendix. Some basic notes:

- 5 convolutional layers used with Dropout and MaxPooling used
- RELU activation function used at each
- GlobalAveragePool and two added Dense layers after convolutional
- 30 node dense layer for predictions.

Convergence of both accuracy and loss (mean squared error) appeared to happen within 20 epochs (charts showing each found in appendix). Examples of prediction results can be found in the compilation table at the end of this section for comparison.

### *Model 2 – CNN*

Based on the previous model, a second CNN model was created, changing some of the parameters (details of the model structure can be found in the appendix). This model was run for 300 epochs (previous models run for less), both with and without Dropout at each Convolutional and Dense layer for comparison.

The Kaggle scores for the model without Dropout are 3.27584 / 3.34556 for the private/public leaderboards. These correspond to a ranking of 72/70 out of 175 respectively.

The models with Dropout resulted in 3.50782 / 3.55790. Based on these results, Dropout was not used in future modeling.

### *Data Augmentation*

While traditionally data modification would be discussed in the earlier section discussing the data, it is put here in the Modeling section because it is engrained within the modeling process.

All of the previous models utilized a reduced sample of the train data given due to the elimination of all NaN records. To attempt to rectify the missing data issue, an imputation method was done. The basic steps:

1. Separate out 4 keypoint data from train data.
2. Translate the nose tip and bottom lip entries 5 units in the y direction.
3. Create a new test set that includes all records with NaN values.

4. Apply previous model (Model 2 – CNN without Dropout) to this new test set.
5. Impute NaN values with predictions created on the new test set.
6. Merge all data in to one file.

This is what shall be called the *augmented data* from here on.

The previous model was then re-trained on the augmented data and predictions made. The Kaggle results are 2.88545 / 2.58937, good for 54/53 positions on the leaderboard. So, a jump in nearly 20 spots in the ranking is realized with the augmented data.

Additionally, utilizing the ImageDataGenerator function in Keras allows for creation of new images with modifications, such as flipping, rotation, translation, and contrast change (whitening, dithering, etc). Some unique code is required for this since this Keras function only creates new images and does not do anything with the keypoint data (i.e. does not flip it along with it). Credit <http://danielnouri.org/notes/2014/12/17/using-convolutional-neural-nets-to-detect-facial-keypoints-tutorial/> for the new Class creation to execute this here for testing.

The intent here is to compare the augmented data process with the image flipping process here using only the fully-populated 15 keypoint data.

After training Model2 – CNN (no Dropout) on the new data, the Kaggle results are 3.26164 / 3.34581. This is just slightly better on the private board (by .01) and essentially no difference on the public board. It does not appear that using flipped data of the reduced dataset resulted in any notable improvement.

### *Ensemble*

The final model style attempted is being loosely termed as an ensemble. Rather than a single model to predict all 15 facial features, multiple models are created to focus on smaller sets of the 15 keypoints. The idea being that one model that, focuses on eyes will perform better on predicting eye locations than an all-encompassing model (like the ones completed thus far).

The features were placed in groups: eye centers, nose tip, mouth bottom lip, mouth corners and upper lip, eye corners, eyebrow corners. The code then cycled through each of these groups, using Model2 – CNN and augmented data, and made predictions. The model needed some additional changes with the last layer to change the output number of nodes to match that of the current sub-group being predicted, but the rest of the model remained unchanged. Details of the code can be found in the index.

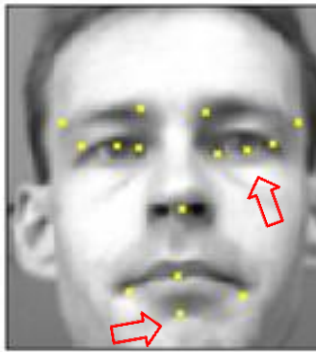

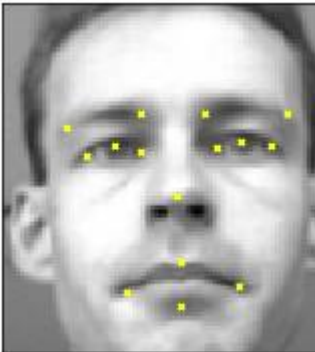




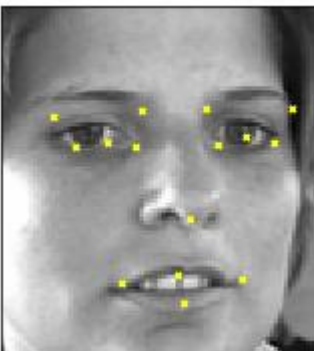
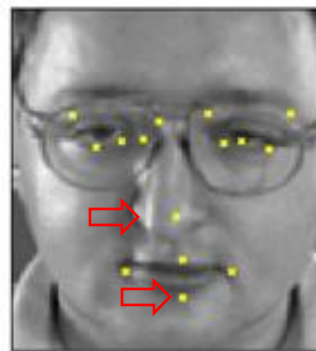


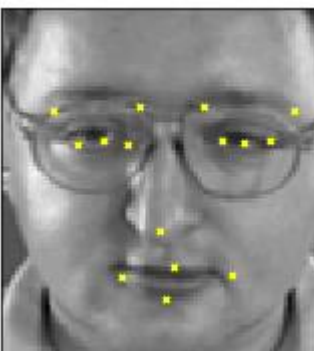

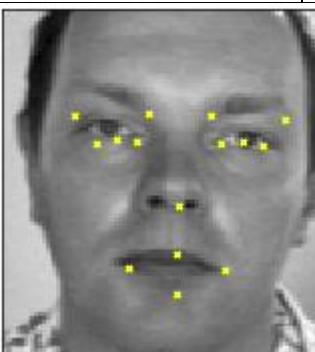

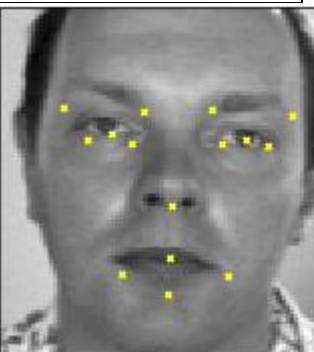
The initial run of the ensemble used 100 epochs and also attempted to use ZCA\_whitening, an image modifier designed to aid the algorithm to better see contrasts in the image. However, running on a CPU proved difficult. A single epoch takes over 2 hours of runtime, and this was going to run for 7 groups at 100 epoch each. Not surprisingly, this trial was abandoned.

A second ensemble model was run, this time with 50 epoch and ZCA\_whitening eliminated, but still using the augmented data. This took 7hr, 45 min to complete the

entire calculation [by eliminating the DataImageGenerator from the process, the time for a single epoch went from 2+ hours to about 90 sec per].

The results from this model are 2.16123 / 2.45462, good for a ranking of 48 on both private and public leaderboard, the best results realized.

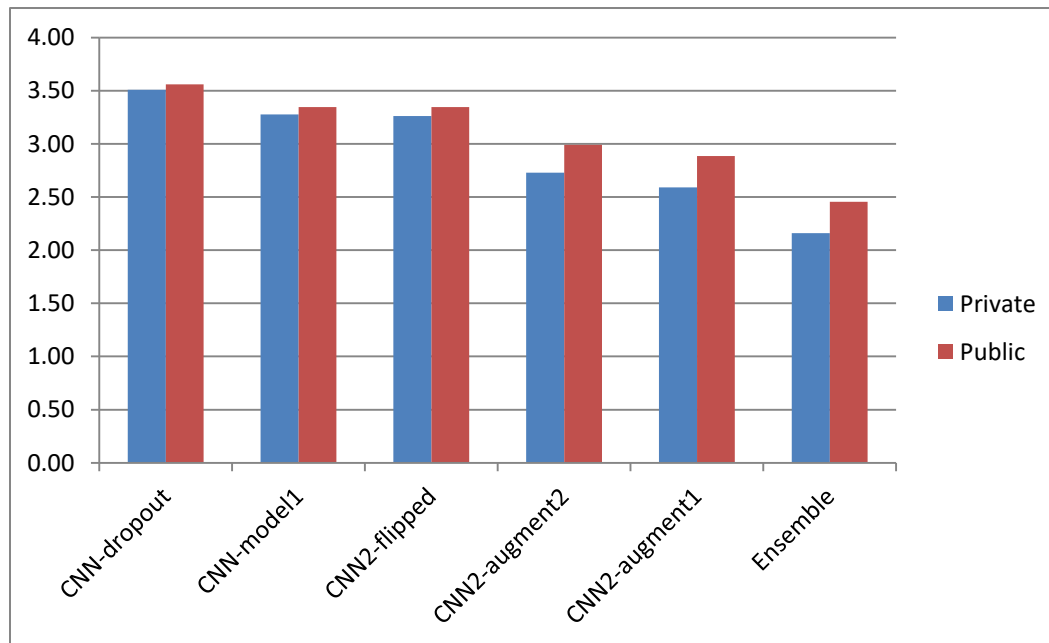
Below are some visual results of the various models created. The yellow points are the predicted locations and red arrows are some features to take note of when comparing. As the chart moves to the right, the higher performing the model was. One can see the general improvement of the 'problem' predictions as the models were improved.

<i>Model1 - NN</i>	<i>Model1 - CNN</i>	<i>Model2 - CNN</i>	<i>Model2 - CNN (augmented)</i>
			
			
			
			



## End Remarks

The top performing model placed nearly in the top 25% of posted results on Kaggle. A graphical summary of results:



Some other general notes and intuitions:

- It is apparent that a 'specialized' model – one that focuses on specific regions of the face image -performs better than a single general model.
- Increasing the train data size from a basic 'no NaN' to an augmented group consisting of imputed values aids in identification.
- Merely flipping images to increase the number did not improve performance [at least in the specific instance here when using just the non-NaN records and flipping those].
- Speculation is that the added variability of the augmented data (using the more widely-varying 4 keypoint data images) helps in generalizing the models for prediction accuracy.
- While an interesting feature, the DataImageGenerator function in Keras is extremely slow, computationally speaking. It may just be the CPU on which the modeling was done, but it dramatically added time to any model processing.

Some additional items and methods to try for future continuation of this project:

- More varying model structures. Modifying the number of layers, nodes, activation functions, learning rates, and optimizers are all possible model enhancements that can be tried.

- Data enhancements like flipping, rotating, translation, and contrast change are all added features that in many scenarios improve upon image model performance.
- Image grabbing. There are some functions that do a grid search on an image file to find features (for example, a face) and then focus future calculations on that region only.

## INDEX

### Data – General

```
Index(['left_eye_center_x', 'left_eye_center_y', 'right_eye_center_x',
      'right_eye_center_y', 'left_eye_inner_corner_x',
      'left_eye_inner_corner_y', 'left_eye_outer_corner_x',
      'left_eye_outer_corner_y', 'right_eye_inner_corner_x',
      'right_eye_inner_corner_y', 'right_eye_outer_corner_x',
      'right_eye_outer_corner_y', 'left_eyebrow_inner_end_x',
      'left_eyebrow_inner_end_y', 'left_eyebrow_outer_end_x',
      'left_eyebrow_outer_end_y', 'right_eyebrow_inner_end_x',
      'right_eyebrow_inner_end_y', 'right_eyebrow_outer_end_x',
      'right_eyebrow_outer_end_y', 'nose_tip_x', 'nose_tip_y',
      'mouth_left_corner_x', 'mouth_left_corner_y', 'mouth_right_corner_x',
      'mouth_right_corner_y', 'mouth_center_top_lip_x',
      'mouth_center_top_lip_y', 'mouth_center_bottom_lip_x',
      'mouth_center_bottom_lip_y', 'Image'],
      dtype='object')
```

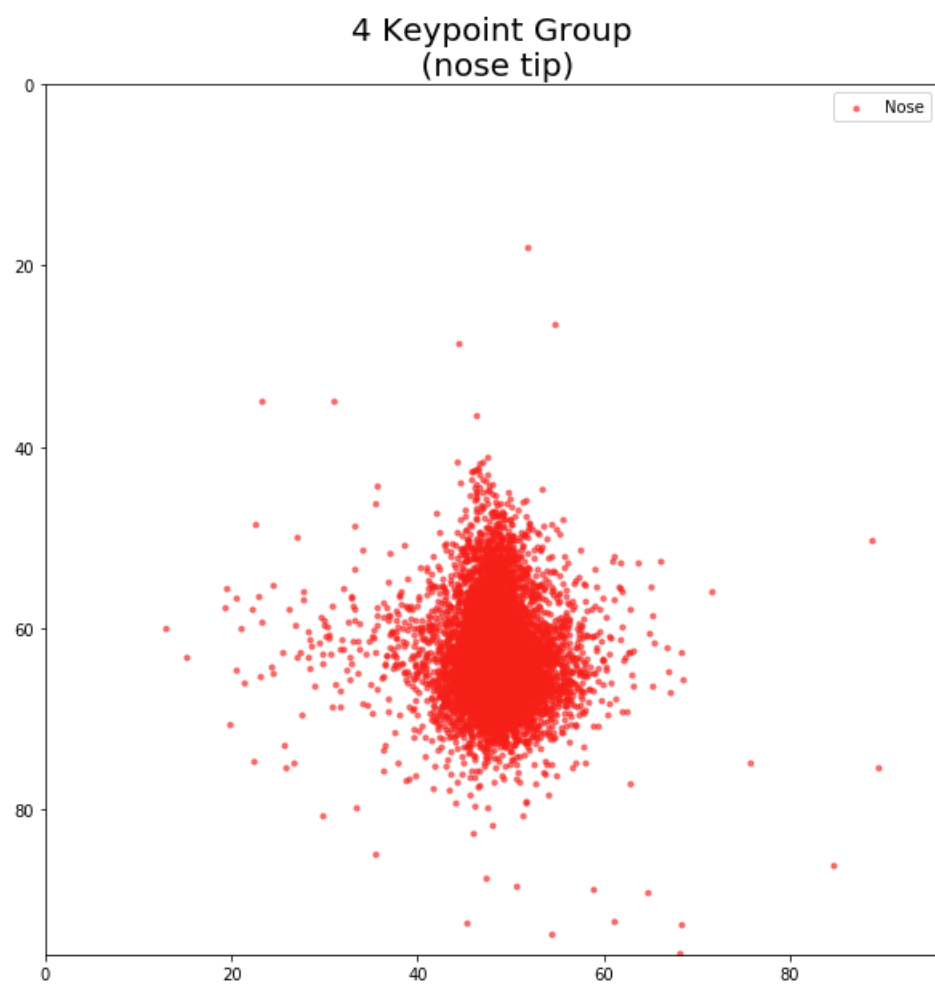
### Image Info:

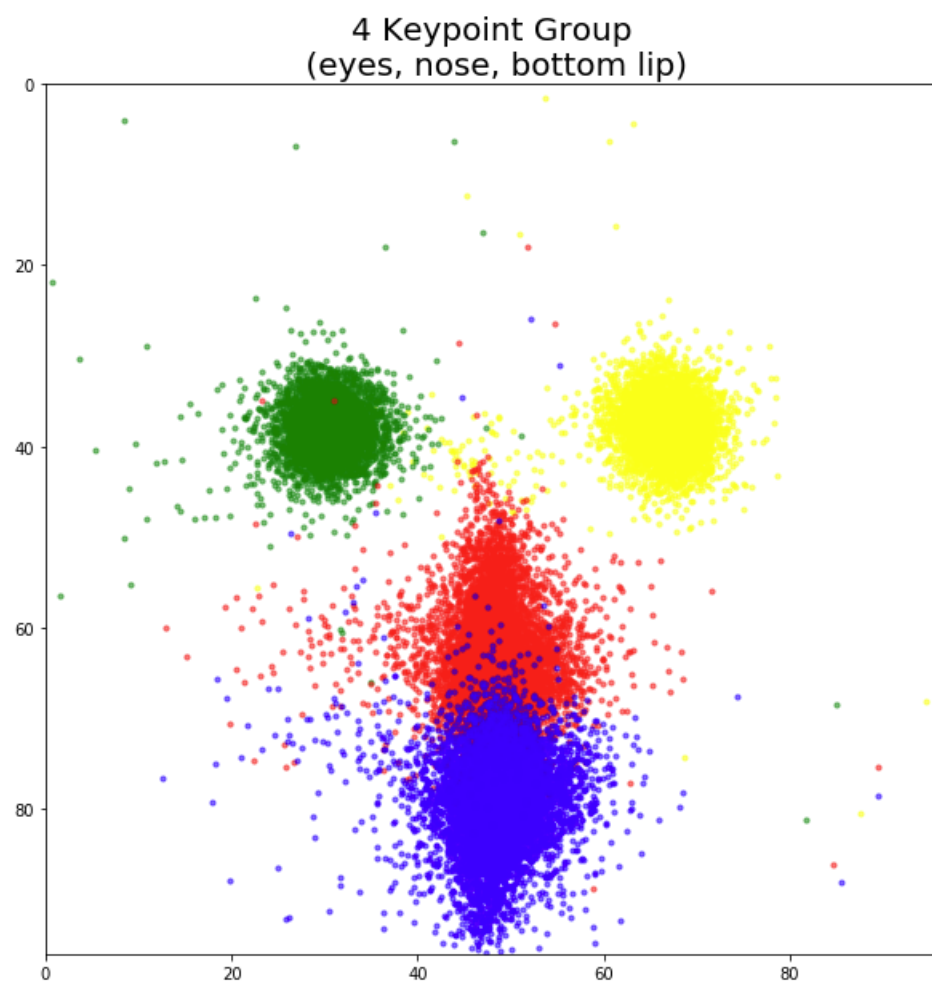
```
0    [238.0, 236.0, 237.0, 238.0, 240.0, 240.0, 239...
1    [219.0, 215.0, 204.0, 196.0, 204.0, 211.0, 212...
2    [144.0, 142.0, 159.0, 180.0, 188.0, 188.0, 184...
3    [193.0, 192.0, 193.0, 194.0, 194.0, 194.0, 193...
4    [147.0, 148.0, 160.0, 196.0, 215.0, 214.0, 216...
Name: Image, dtype: object
```

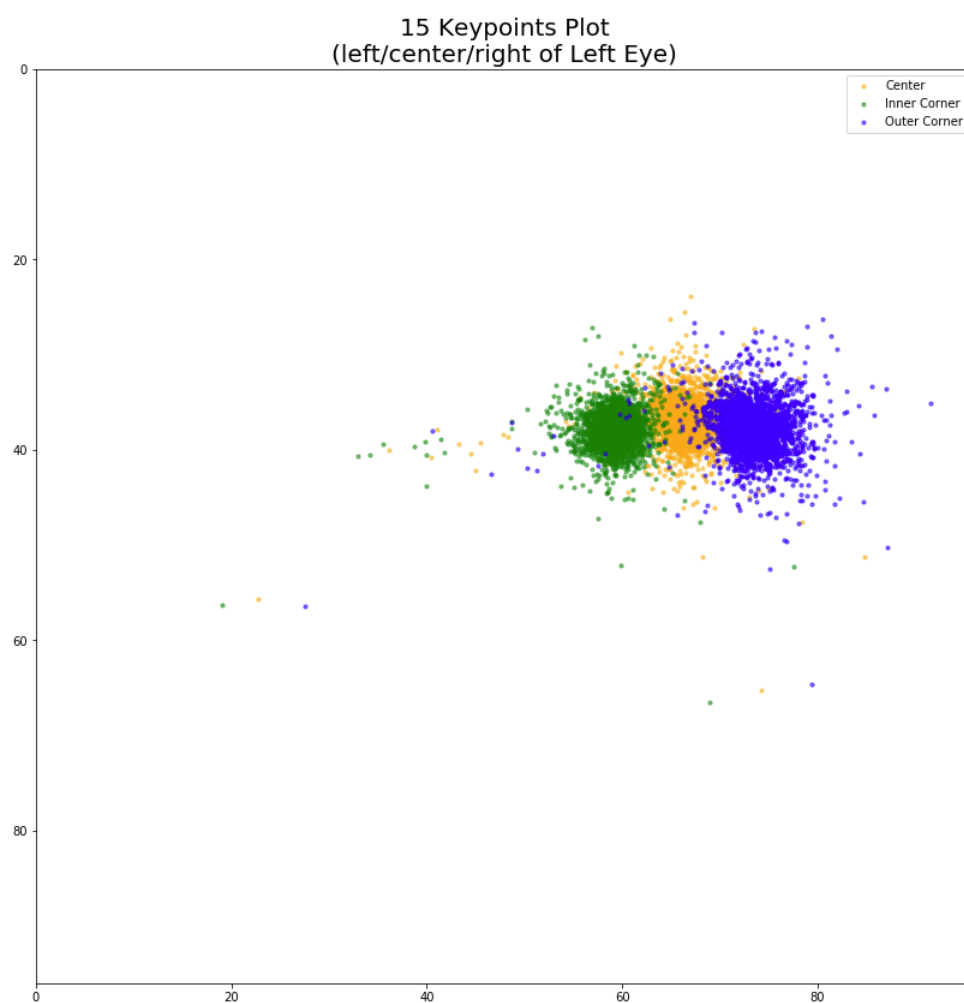
```
array([[ 0.93333334,  0.9254902 ,  0.92941177, ...,  0.27450982,
         0.29411766,  0.35294119],
       [ 0.85882354,  0.84313726,  0.80000001, ...,  0.00392157,
         0.00392157,  0.00392157],
       [ 0.56470591,  0.55686277,  0.62352943, ...,  0.30588236,
         0.30588236,  0.3019608 ],
       ...,
       [ 0.29019609,  0.29019609,  0.29019609, ...,  0.07843138,
         0.07843138,  0.07843138],
       [ 0.99607843,  0.99607843,  0.99607843, ...,  0.99607843,
         0.99607843,  0.99607843],
       [ 0.20784314,  0.24313726,  0.26274511, ...,  0.61960787,
         0.61960787,  0.62352943]], dtype=float32)
```

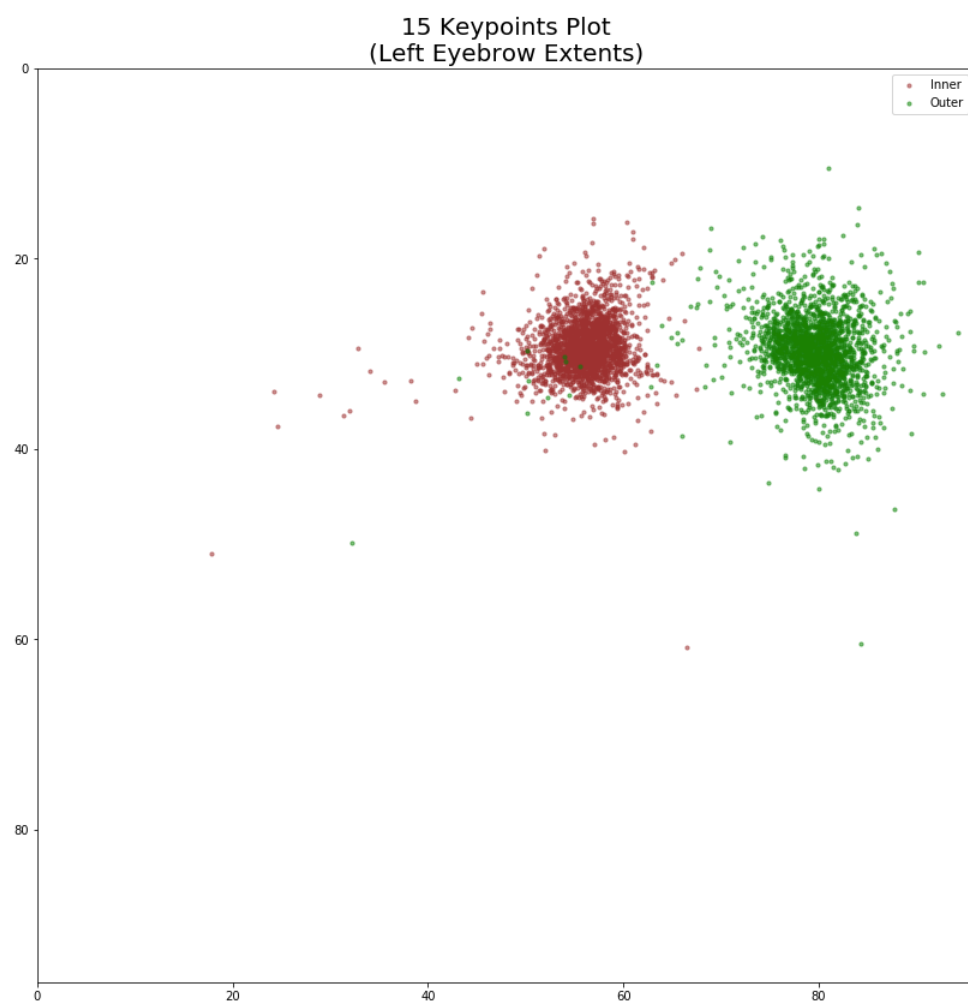
Scatterplots of train data locations:



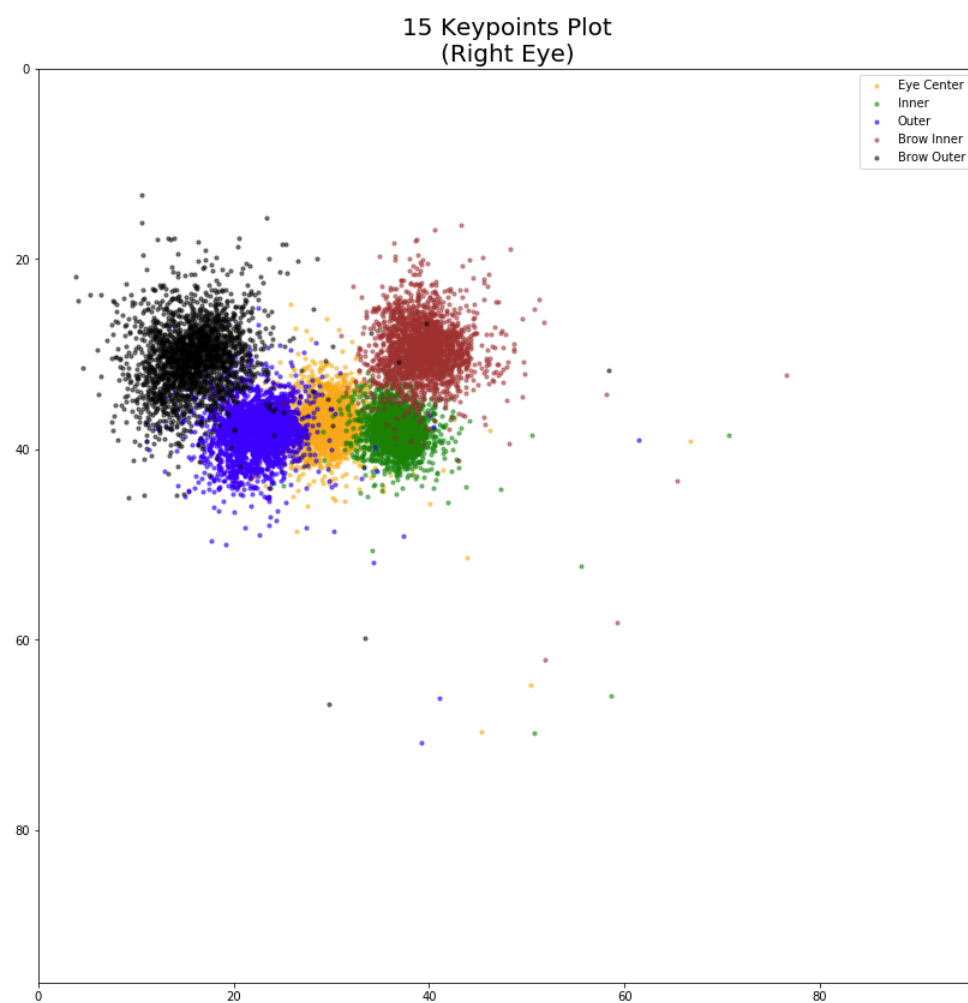


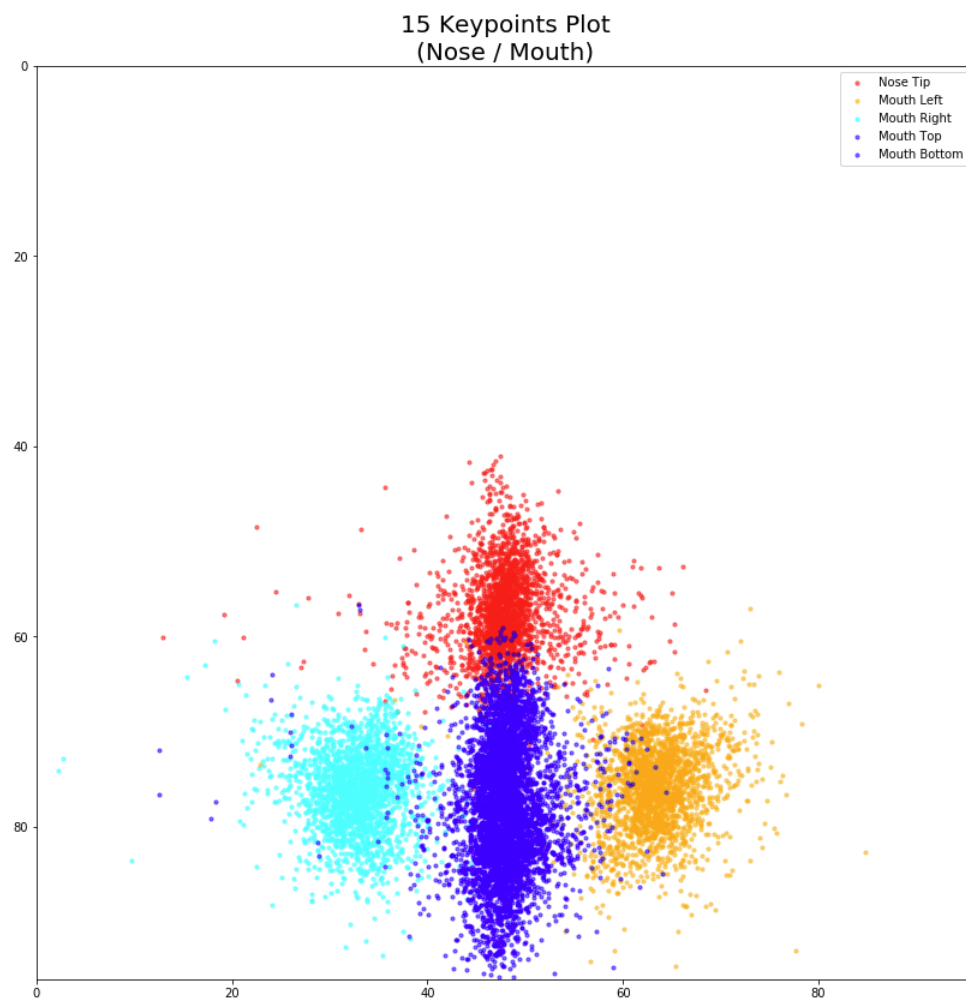








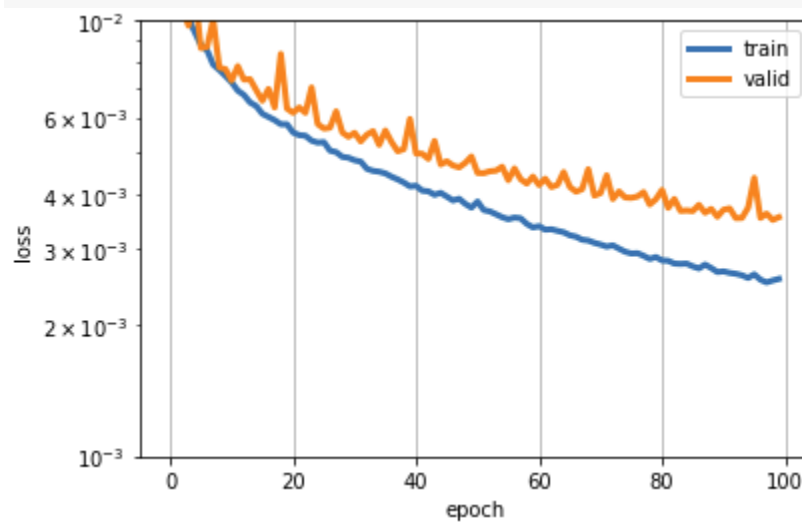




## MODELS

### Model1 – NN

```
model = Sequential()  
model.add(Dense(100, input_dim=9216))  
model.add(Activation('relu'))  
model.add(Dense(30))  
  
sgd = SGD(lr=0.01, momentum=0.9, nesterov=True)  
model.compile(loss='mean_squared_error', optimizer=sgd)  
hist = model.fit(X, y, epochs=100, validation_split=0.2)
```



## Model1 – CNN

```
model = Sequential()

#
model.add(BatchNormalization(input_shape=(96, 96, 1)))

model.add(Conv2D(24, (5, 5), padding='same', kernel_initializer='he_normal'))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid'))

model.add(Conv2D(36, (4, 4), activation='relu'))
model.add(Dropout(0.2))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid'))

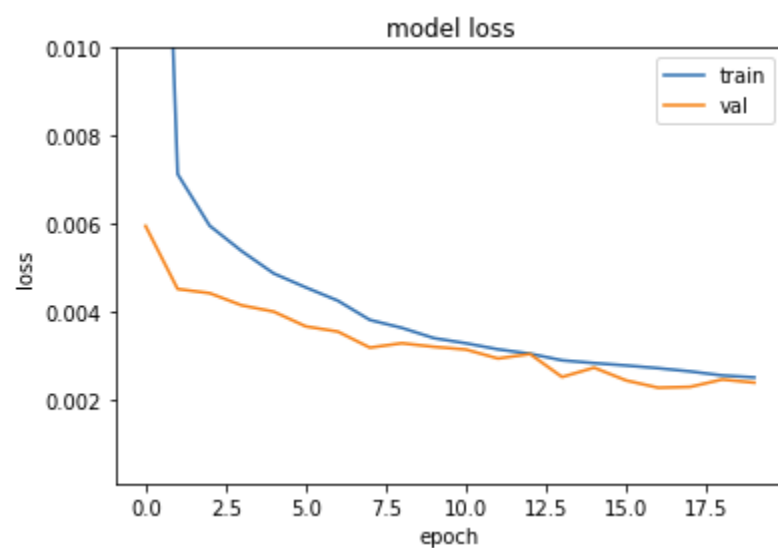
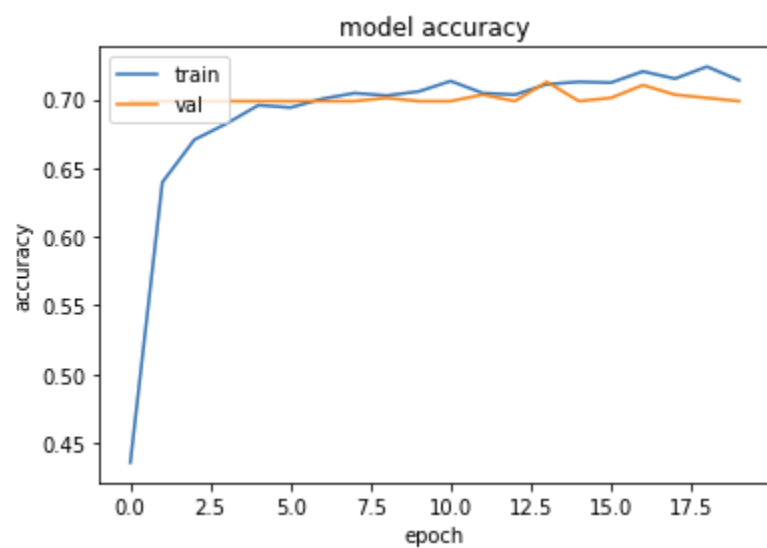
model.add(Conv2D(48, (4, 4), activation='relu'))
model.add(Dropout(0.2))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid'))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid'))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(Dropout(0.2))

model.add(GlobalAveragePooling2D());
model.add(Dropout(0.2))
model.add(Dense(500, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(90, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(30))

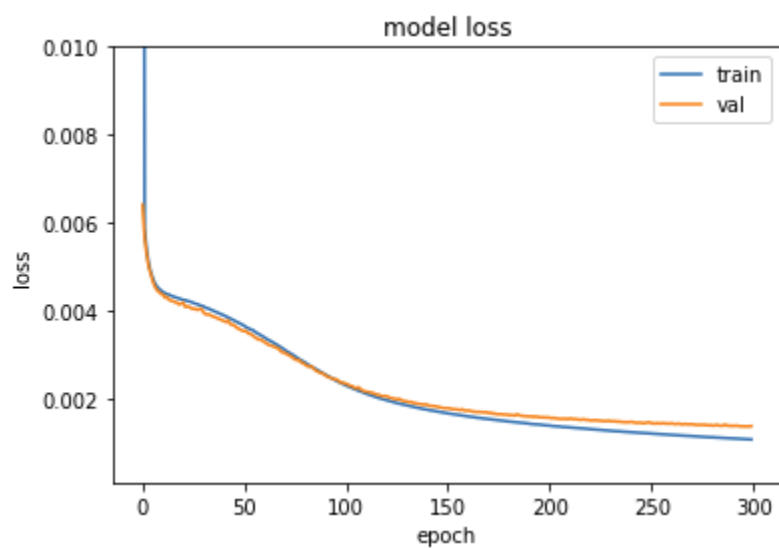
model.compile(optimizer='rmsprop', loss='mse', metrics=['accuracy'])
checkpointer = ModelCheckpoint(filepath='CNN_500-90-30_v5.h5', verbose=1, save_best_only=True)
epochs = 20
hist = model.fit(X, y, validation_split=0.2, shuffle=True, epochs=epochs,
                 batch_size=10, callbacks=[checkpointer], verbose=1)
```



## Model2 – CNN

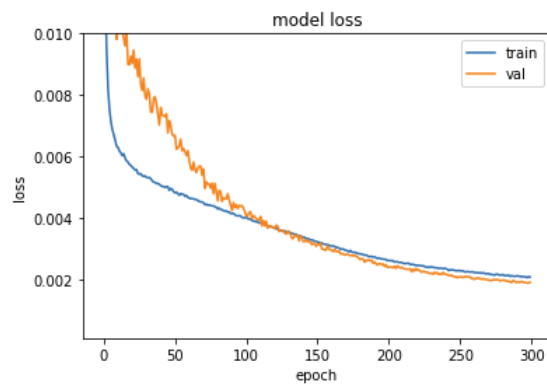
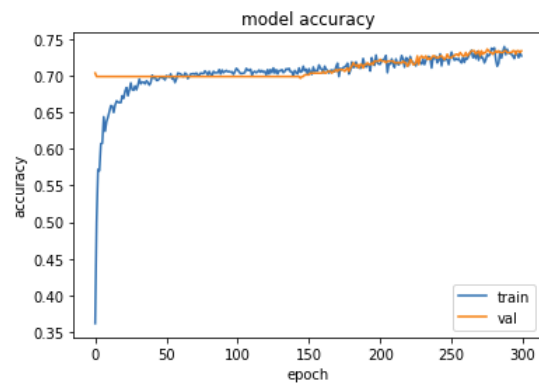
```
def SimpleCNN(withDropout=False):  
|  
    model = Sequential()  
    model.add(Conv2D(32,(3, 3), input_shape = (96, 96, 1)))  
    model.add(Activation('relu'))  
    model.add(MaxPooling2D(pool_size = (2,2)))  
    if withDropout:  
        model.add(Dropout(0.1))  
  
    model.add(Conv2D(64,(2,2)))  
    model.add(Activation('relu'))  
    model.add(MaxPooling2D(pool_size = (2,2)))  
    if withDropout:  
        model.add(Dropout(0.1))  
  
    model.add(Conv2D(128,(2,2)))  
    model.add(Activation('relu'))  
    model.add(MaxPooling2D(pool_size=(2,2)))  
    if withDropout:  
        model.add(Dropout(0.1))  
  
    model.add(Flatten())  
  
    model.add(Dense(500))  
    model.add(Activation('relu'))  
    if withDropout:  
        model.add(Dropout(0.1))  
  
    model.add(Dense(500))  
    model.add(Activation('relu'))  
    if withDropout:  
        model.add(Dropout(0.1))  
  
    model.add(Dense(30))  
    sgd = SGD(lr=0.01,momentum = 0.9,nesterov=True)  
    model.compile(loss="mean_squared_error",optimizer=sgd)  
    return(model)
```

```
%%time  
model2 = SimpleCNN()  
  
hist2 = model2.fit(X,y,epochs=300,validation_split=0.2,verbose=1, callbacks=[checkpointer])
```



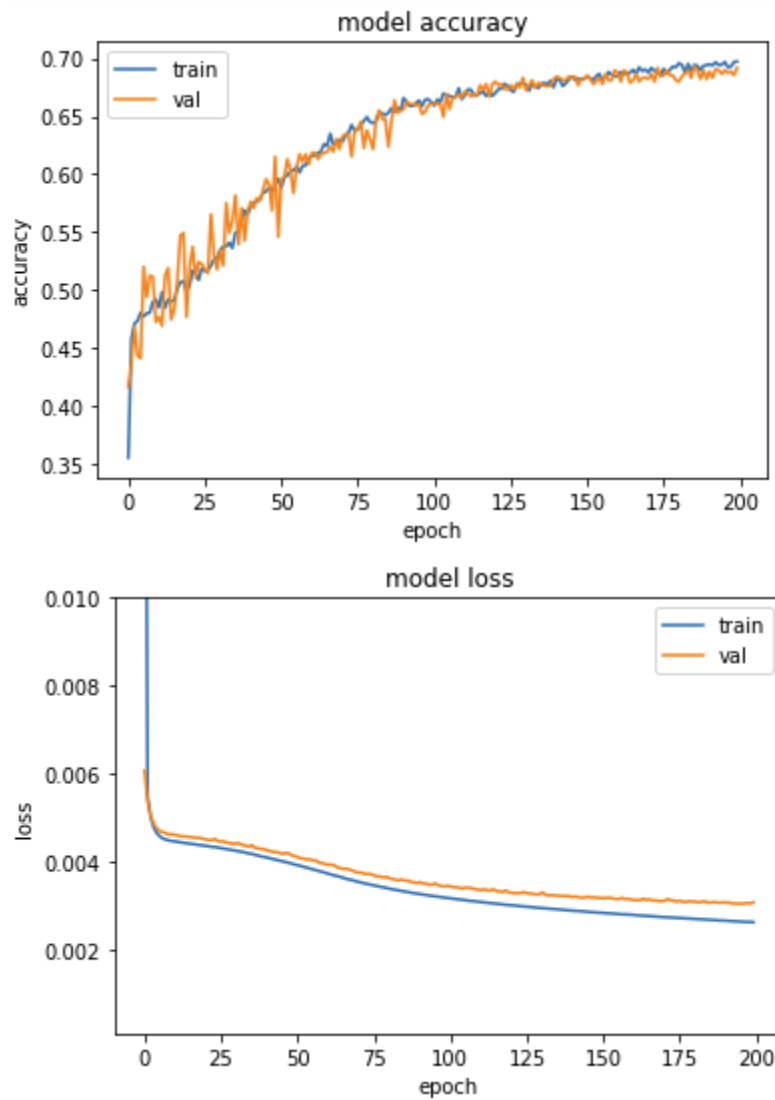
## Model2 – CNN with Dropout

```
%%time  
model2 = SimpleCNN(withDropout=True)  
  
hist2 = model2.fit(X,y,epochs=300,validation_split=0.2,verbose=1, callbacks=[checkpointer])
```





## Model2 – CNN with Data Augmentation



## Data Augmentation – Flipping Code

```
class FlippedImageDataGenerator(ImageDataGenerator):
    flip_indices = [
        (0, 2), (1, 3),
        (4, 8), (5, 9), (6, 10), (7, 11),
        (12, 16), (13, 17), (14, 18), (15, 19),
        (22, 24), (23, 25),
    ]

    def next(self):
        X_batch, y_batch = super(FlippedImageDataGenerator, self).next()
        batch_size = X_batch.shape[0]
        indices = np.random.choice(batch_size, batch_size/2, replace=False)
        X_batch[indices] = X_batch[indices, :, :, ::-1]

        if y_batch is not None:
            y_batch[indices, ::2] = y_batch[indices, ::2] * -1

            # left_eye_center_x -> right_eye_center_x
            for a, b in self.flip_indices:
                y_batch[indices, a], y_batch[indices, b] = (
                    y_batch[indices, b], y_batch[indices, a]
                )

        return X_batch, y_batch
```

```
flipgen = FlippedImageDataGenerator()
```

```
%%time
model2 = SimpleCNN()

hist2 = model2.fit_generator(flipgen.flow(X,y),epochs=300,verbose=1, callbacks=[checkpointer],
                             steps_per_epoch=X.shape[0])
```

## Model – Ensemble

```
ENSEMBLE_GROUPS = [  
    dict(  
        columns=(  
            'left_eye_center_x', 'left_eye_center_y',  
            'right_eye_center_x', 'right_eye_center_y',  
        ),  
    ),  
  
    dict(  
        columns=(  
            'nose_tip_x', 'nose_tip_y',  
        ),  
    ),  
  
    dict(  
        columns=(  
            'mouth_left_corner_x', 'mouth_left_corner_y',  
            'mouth_right_corner_x', 'mouth_right_corner_y',  
            'mouth_center_top_lip_x', 'mouth_center_top_lip_y',  
        ),  
    ),  
  
    dict(  
        columns=(  
            'mouth_center_bottom_lip_x',  
            'mouth_center_bottom_lip_y',  
        ),  
        #flip_indices=(),  
    ),  
]
```

```
def fit_ensemble(
    print_every=20,
    epochs=100,
    prop=0.1,
    transfer_model='CNN2_128-500-500-30_augmentdata.h5'):
    specialists = OrderedDict()

    for setting in ENSEMBLE_GROUPS:

        cols = setting['columns']

        X, y = load2d(cols=cols)
        X_train, X_val, y_train, y_val = train_test_split(X, y,
                                                            test_size=0.2,
                                                            random_state=123)

        model=SimpleCNN()
        model.layers.pop() # get rid of output layer
        model.outputs = [model.layers[-1].output]
        model.layers[-1].outbound_nodes = []
        model.add(Dense(len(cols))) # add new output layer

        model.compile(loss='mse', optimizer="adam")
        #model.summary()
        hist = model.fit_generator(datagen.flow(X_train,y_train), validation_data=(X_val,y_val),
                                    epochs=epochs, verbose=1, steps_per_epoch=X_train.shape[0])
        specialists[cols] = {"model":model,"hist":hist}
```