

# Facial Keypoints Detection

## I. Definition

### Project Overview

Facial keypoints detection has become a very challenging problem, especially with the emergence of applications such as Snapchat, which made people obsessed with placing funny objects on their faces like sunglasses, caps and mustaches. The difficulty of facial keypoints detection is to detect the keypoints such as eyes, nose and mouth which vary from person to another. The problem even becomes even more challenging when the face images are taken under different illumination conditions. Keypoints detection can be used in a lot of application such as tracking faces in images and video, analyzing facial expressions and face recognition<sup>[1]</sup>.

The project is basically a competition in Kaggle. I used the database provided by Kaggle for their facial key-points detection challenge. For training I have 7049 grayscale images of equal size (96x96). For each training image, 15 keypoints are provided with both x and y coordinates, allowed omission for portion keypoints. Hence mostly there are 30 or fewer target labels for each training image. Test data set has 1783 images with no target information.

### Problem Statement

The objective is to build an algorithm that, given an image of a face, automatically locates where these keypoints are. Keypoints detection is a regression problem and training a Convolutional Neural Network will definitely be the best choice for the detection process. The time taken in the training process is an important metric to evaluate the model. I've seen the results of the model loss and testing images for other people who have carried out the same project. But what made their result better than mine is that there are training the model for 1000 or 3000 epochs, which will take approximately an hour and maybe more and it's not exactly inspiring to have to wait for an experiment's results for so long. Also my project handle an offline detection, but what if we would like to fit the detection procedure to a real-time mobile app, we have to complete the detection within seconds. Therefore, the computation complexity of keypoints detection have to be lower than traditional image classification tasks<sup>[3]</sup>. I think time is an important point to address even if I'm not going to need it now.

### Evaluation Metrics

According to Kaggle the best way to evaluate the model is to use 'root mean squared error'. RMSE is a suitable general-purpose error metric, as it punishes large errors<sup>[1]</sup>.

$$RMSE = \sqrt{\frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{n}}$$

where y is the original value,  $\hat{y}$  is the predicted value and n is the number of targets to be predicted.

[1]. <https://www.kaggle.com/c/facial-keypoints-detection/overview>

[3]. [cs231n.stanford.edu/reports/2016/pdfs/007\\_Report.pdf](https://cs231n.stanford.edu/reports/2016/pdfs/007_Report.pdf)

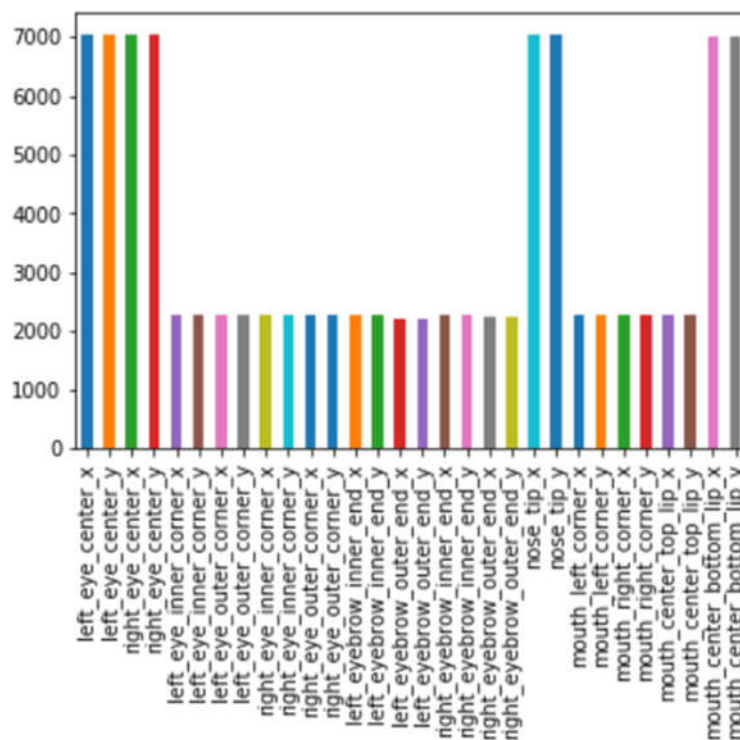
## II. Analysis

### Data Exploration

As I mentioned in project overview section, for each training image, 15 keypoints are provided with both x and y coordinates. The 15 FKPs are left eye center, right eye center, left eye inner corner, left eye outer corner, right eye inner corner, right eye outer corner, left eyebrow inner end, left eyebrow outer end, right eyebrow inner end, right eyebrow outer end, nose tip, mouth left corner, mouth right corner, mouth center top lip and mouth center bottom lip.

The data can be downloaded from here: <https://www.kaggle.com/c/facial-keypoints-detection/data>

### Exploratory Visualization



What this plot tells us is that in this dataset, there's only 2,140 images in the dataset that have all 30 target values present. I can do two things here one remove the rows having missing values and another is the fill missing values with something. So I decided to go with option one and use data augmentation to increase the data. I used dropna () function to drop all rows with missing values.

### Algorithms and Techniques

The classifier is a Convolutional Neural Network, which is the state-of-the-art algorithm for most image processing tasks, including classification. It needs a large amount of training data compared to other approaches; unfortunately, the database I have aren't big enough, but I hope with data augmentation I'll get more data.

The following parameters can be tuned to optimize the classifier:

#### Training parameters:

- Training length (number of epochs)
- Batch size (how many images to look at once during a single training step)
- Solver type (what algorithm to use for learning)
- Learning rate (how fast to learn; this can be dynamic)
- Weight decay (prevents the model being dominated by a few “neurons”)
- Momentum (takes the previous learning step into account when calculating the next one)

#### Neural network architecture:

- Choosing number of hidden layers and hidden units
- Number of Filters
- Layer types (convolutional, fully-connected, pooling, etc.)

#### Benchmark

A simple solution has been introduced to solve this problem<sup>[1]</sup>. They used a convolutional neural net with three convolutional layers and two fully connected layers. Each conv layer is followed by a 2x2 max-pooling layer. Starting with 32 filters, double the number of filters with every conv layer. The densely connected hidden layers both have 500 units. The output was as following:

Epoch	Train loss	Valid loss	Train / Val
1	0.111763	0.042740	2.614934
2	0.018500	0.009413	1.965295
3	0.008598	0.007918	1.085823
4	0.007292	0.007284	1.001139
5	0.006783	0.006841	0.991525
...			
500	0.001791	0.002013	0.889810
501	0.001789	0.002011	0.889433
502	0.001786	0.002009	0.889044
503	0.001783	0.002007	0.888534
504	0.001780	0.002004	0.888095
505	0.001777	0.002002	0.887699
...			
995	0.001083	0.001568	0.690497
996	0.001082	0.001567	0.690216
997	0.001081	0.001567	0.689867
998	0.001080	0.001567	0.689595
999	0.001080	0.001567	0.689089
1000	0.001079	0.001566	0.688874

Quite a nice improvement over the first network. Our RMSE is looking pretty good, too:

```
>>> np.sqrt(0.001566) * 48  
1.8994904579913006
```

[1]. <https://mesin-belajar.blogspot.com/2015/11/>

III. Methodology

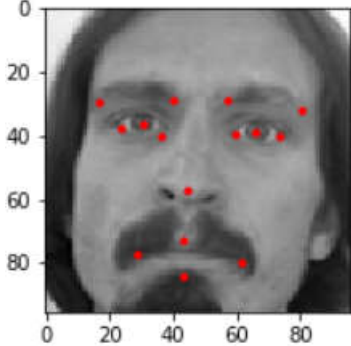
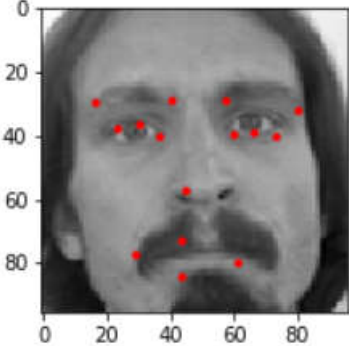
Data Preprocessing

I. Normalization

Neural networks process inputs using small weight values, and inputs with large integer values can disrupt or slow down the learning process. As such it is good practice to normalize the pixel values so that each pixel value has a value between 0 and 1. This is why the image pixels are normalized to the range [0; 1] by dividing by 255.

II. Data Augmentation

Due to the limitation of our training data, it's better to use data Augmentation to help boost the performance of the deep learning model. I have horizontally flipped the images for which target information for all the 15 FKP's are available. After the image flipping I found that the target value keypoints aren't aligned with the image anymore. Since I'm flipping the images, I'll have to make sure I also flip the following target values:

	
Target Facial Key Points	Target Facial Key Points
Left Eye Center X	Right Eye Center X
Left Eye Center Y	Right Eye Center Y
Left Eye Inner Corner X	Right Eye Inner Corner X
Left Eye Inner Corner Y	Right Eye Inner Corner Y
Left Eye Outer Corner X	Right Eye Outer Corner X
Left Eye Outer Corner Y	Right Eye Outer Corner Y
Left Eyebrow Inner Corner X	Right Eyebrow Inner Corner X
Left Eyebrow Inner Corner Y	Right Eyebrow Inner Corner Y
Left Eyebrow Outer Corner X	Right Eyebrow Outer Corner X
Left Eyebrow Outer Corner Y	Right Eyebrow Outer Corner Y
Mouth Left Corner X	Mouth Right Corner X
Mouth Left Corner Y	Mouth Right Corner Y

The images is horizontally flipped with this line of code:

```
flipped_image= X [image num , : , ::-1, : ]
```

To flip the target points I used the code from here<sup>[1]</sup>. Where the output = information (image\_no), returns the x co-ordinate and y coordinate of the 15 landmark points on the face.

```
right_eye_center_x = df_drop['right_eye_center_x'][image_no]/96
right_eye_center_y = df_drop['right_eye_center_y'][image_no]/96
```

```
def flip_points(image_no):
    output = information(image_no)
    i = 0
    while i < 15:
        output[i] = 1 - output[i]
        i = i + 1
    return output
```

From here you can see the points before and after flipping:

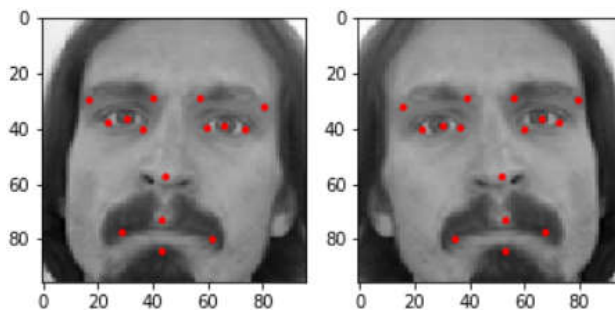
```
print(information(0))
```

```
[0.68784962406041661, 0.31486466165416666, 0.62064661654166675, 0.76177443609062501, 0.37871428571458332, 0.24430075188020836,
0.59326315789479167, 0.83569924811979168, 0.41903759398541668, 0.17037894736875001, 0.46271428571458334, 0.63745112781979163,
0.29806766917291666, 0.45117293233125005, 0.44927819548854164, 0.40627368421041665, 0.37939248120312502, 0.41299398496249995,
0.41635413533854165, 0.41635413533854165, 0.38947293233125002, 0.30243383458645834, 0.33570977443645833, 0.30210751879687497,
0.30882781954895833, 0.59444586466145832, 0.83302255639062495, 0.80613533834583329, 0.7597443609020833, 0.88006015037604168]
```

```
print(flip_points(0))
```

```
[0.31215037593958339, 0.6851353383458334, 0.37935338345833325, 0.23822556390937499, 0.62128571428541668, 0.75569924811979161,
0.40673684210520833, 0.16430075188020832, 0.58096240601458327, 0.82962105263124997, 0.5372857142854166, 0.36254887218020837, 0.
70193233082708328, 0.54882706766874989, 0.55072180451145836, 0.40627368421041665, 0.37939248120312502, 0.41299398496249995, 0.4
1635413533854165, 0.41635413533854165, 0.38947293233125002, 0.30243383458645834, 0.33570977443645833, 0.30210751879687497, 0.30
882781954895833, 0.59444586466145832, 0.83302255639062495, 0.80613533834583329, 0.7597443609020833, 0.88006015037604168]
```

Image after flipping:



I generated additional 2140 horizontal augmented images. Finally I stack the new horizontally flipped data under the original train data to create the augmented train dataset.

[1]. <https://github.com/adibyte95/face-key-points-detection/blob/master/prog.ipynb>

## Implementation

I used Keras framework V2.0.9<sup>[1]</sup>. The CNN classifier was trained on the preprocessed training data and can be further divided into the following steps:

- Load the training images into memory, preprocessing them as described above.
- Define the network architecture, I used 4 convolution2d and 4 maxpooling2d layers, two dense layer. It is worth mentioning that when the model suffered from Overfitting, I used dropout in each layer but it did not reduce the model loss as I expected so I added dropout in the second and fourth layer only, which gave me a good result and also the use of Batch Normalization improved the model.
- Define the loss function. Since it's a regression problem, I preferred to use the Mean Squared Error as a loss function. The squaring means that larger mistakes result in more error than smaller mistakes, meaning that the model is punished for making larger mistakes.
- For the optimizer I used both Adam and SGD optimizers. Although I used SGD on my final model, but Adam optimizer in my first model gave me a better training loss than SGD in the final model.
- I used accuracy as a metric when training the model. Although I think 'MAE' might be a better metric to be used instead.
- Splitting the database into training and validation data with 'validation\_split' with 0.2 of the training data to be used as validation data. According to Keras the validation data is selected from the last samples in the x and y data provided, before shuffling<sup>[2]</sup>.
- Train the network, logging the validation/training loss and the validation accuracy
- Plot the logged values.
- Test the model with image in test.csv file

## Refinement

I have tried various different deep network architectures. I'm going to mention Architecture , Training Parameters and the Results.

### First Model:

Architecture: Input Image: 96x96x1 (the images are gray scale thus have only one channel). 4 convolution2d, starting with 32 filters, double the number of filters with every conv layer, followed by a Batch Normalization layer and 4 maxpooling2d layers with pool\_size=(2, 2), strides=(2, 2) followed by a Flatten layer and one fully connected layer with 500 hidden units followed by a Batch Normalization layer and output Layer of outputs 30 (since there were 30 keypoint locations)

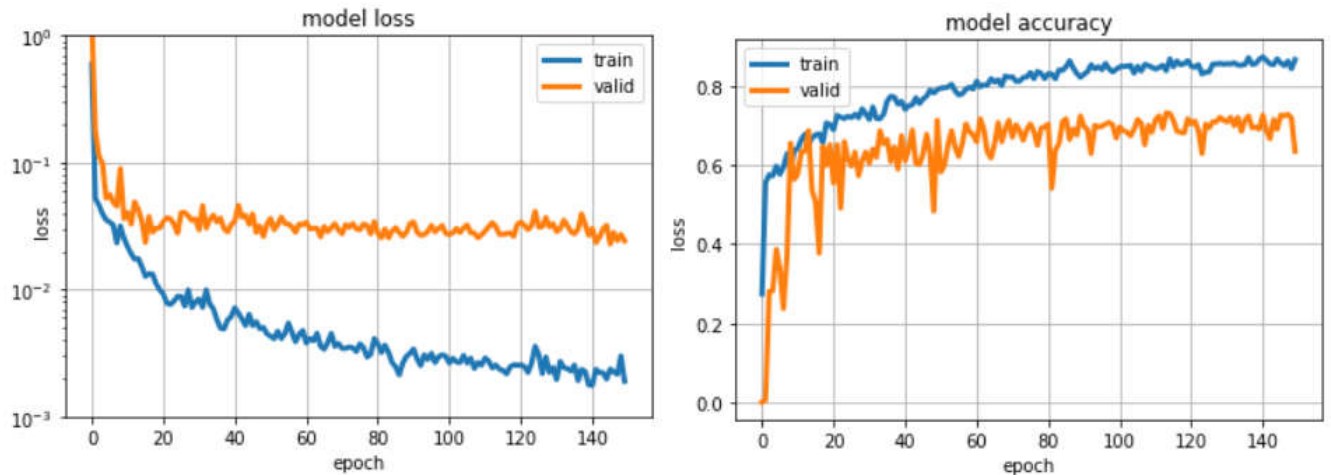
Training Parameters: I used Adam optimizer, with mean square error as a loss function and accuracy as a metric and a learning rate 0.01

Result: I trained the model for 150 epoch and it took 7s 1ms/step. Clearly the timing wasn't good, an overshooting in both loss and accuracy model due to the high learning rate and there's an overfitting.

[1]. <https://keras.io/>

[2]. <https://keras.io/models/sequential/>



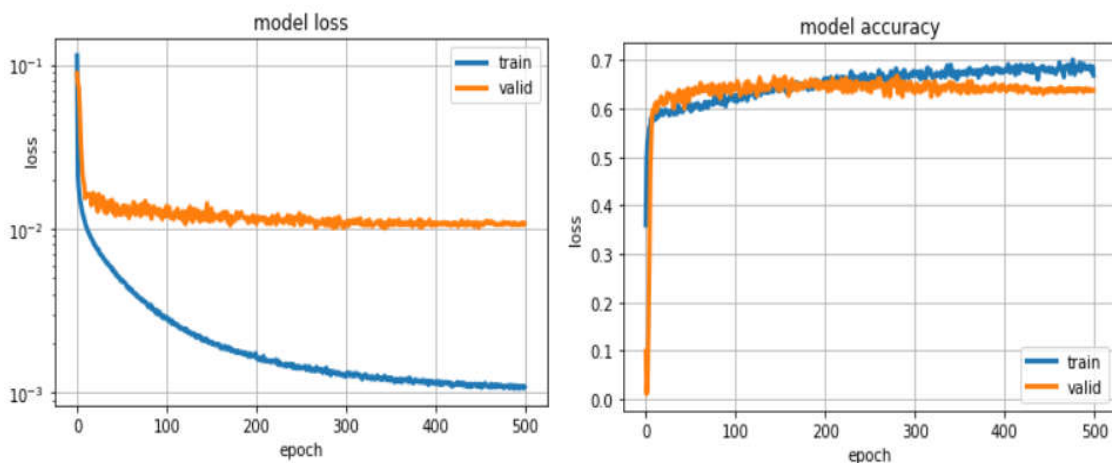


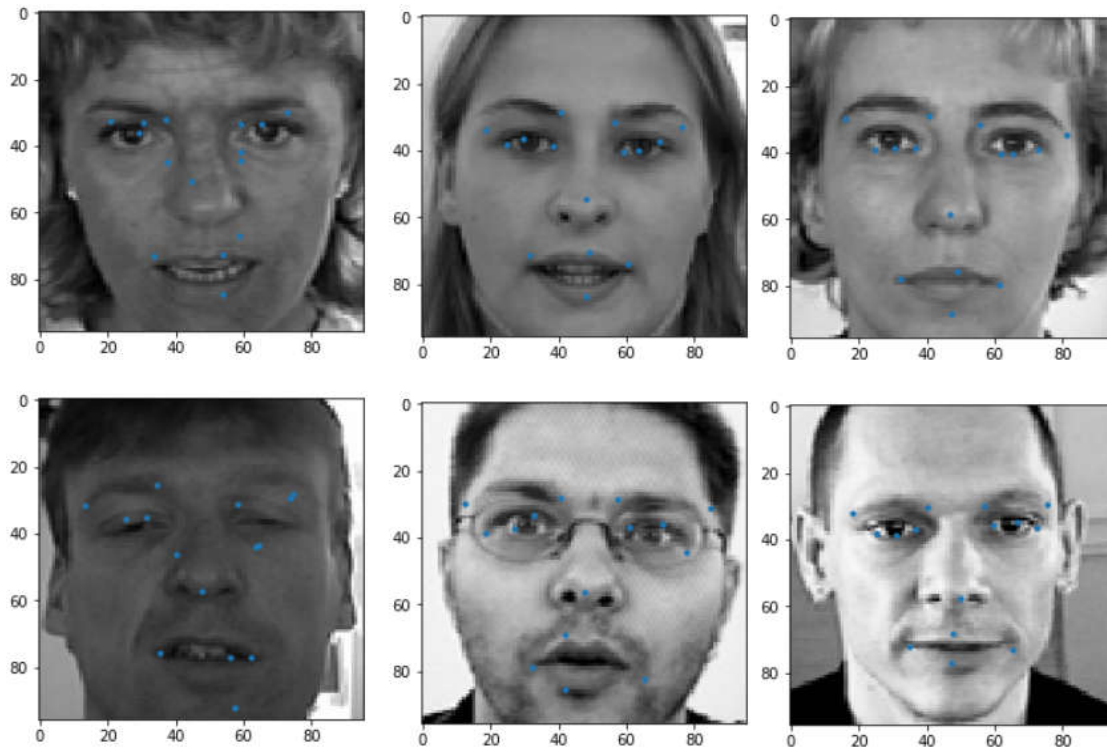
### Second Model:

Architecture: I used 4 convolution2d, starting with 24 filters, double the number of filters with every conv layer, deleted the Batch Normalization from all the layers except for the forth layer, added a dropout of 0.3 after the second layer to solve the overfitting issue and added another fully connected layer with 500 hidden units.

Training Parameters: I used SGD optimizer with momentum=0.9, I used the same loss function and accuracy as a metric. To solve the timing issue I used LearningRateScheduler to reduce the learning rate as the number of training epochs increases. So I started with learning rate 0.03 and stop at 0.001.

Result: I trained the model for 500 epoch. It took 3s 784us/step this time which is a big improvement. The overshooting is reduced in both loss and accuracy model, also the model accuracy is better and the model loss slightly improved, but the testing images wasn't satisfying.





## IV. Results

### Model Evaluation and Validation

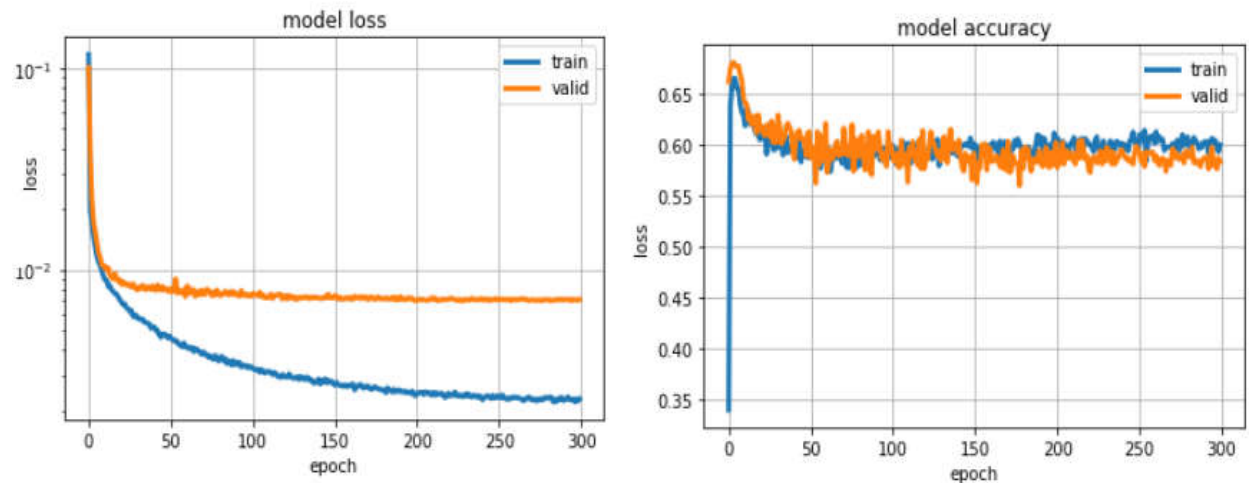
The previous model result wasn't good enough, which is why I did one last modification. A Batch Normalization and dropout layer were added after MaxPooling2D layer 2 and 4. The third layer followed only by a Batch Normalization. The final architecture and hyper parameters were chosen because they performed the best. A complete description of the final model and the training process in the following list:

- The shape of the filters of the convolutional layers is  $3 \times 3$  for the input layer and  $2 \times 2$  for the rest.
- The first convolutional layer starting with 24 filters, double the number of filters with every convolutional layer.
- Every convolutional layer followed by MaxPooling2D with a stride and a pooling layer of 2.
- MaxPooling2D layer 2 and 4 followed by Batch Normalization and dropout. The third layer followed only by a Batch Normalization.
- The forth layer followed by a Flatten layer.
- The two fully connected layers have 500 hidden units.
- An output Layer of 30.
- SGD optimizer with momentum=0.9, MSE as a loss function and accuracy as a metric. A LearningRateScheduler start with learning rate 0.03 and stop at 0.001.
- The training runs for 300 iterations.



**To verify the robustness of the final model, a test was conducted with the test images. The following observations are based on the results of the test:**

Training the model for 300 epoch took 4s 1ms/step which still a good result . The overshooting in the loss model improved a lot , also the model loss improved and the testing images gave a better result than before.



Here you can see the loss and val\_loss over the training process:

```
Train on 3424 samples, validate on 856 samples
Epoch 1/300
3424/3424 [=====] - 7s 2ms/step - loss: 0.1166 - acc: 0.3405 - val_loss: 0.1005 - val_acc: 0.6612
Epoch 2/300
3424/3424 [=====] - 4s 1ms/step - loss: 0.0197 - acc: 0.6376 - val_loss: 0.0380 - val_acc: 0.6741
Epoch 3/300
3424/3424 [=====] - 4s 1ms/step - loss: 0.0169 - acc: 0.6571 - val_loss: 0.0223 - val_acc: 0.6787
Epoch 4/300
3424/3424 [=====] - 4s 1ms/step - loss: 0.0145 - acc: 0.6644 - val_loss: 0.0172 - val_acc: 0.6799
Epoch 5/300
3424/3424 [=====] - 4s 1ms/step - loss: 0.0130 - acc: 0.6641 - val_loss: 0.0155 - val_acc: 0.6764
Epoch 295/300
3424/3424 [=====] - 4s 1ms/step - loss: 0.0023 - acc: 0.6057 - val_loss: 0.0070 - val_acc: 0.5970
Epoch 296/300
3424/3424 [=====] - 4s 1ms/step - loss: 0.0023 - acc: 0.6022 - val_loss: 0.0071 - val_acc: 0.5794
Epoch 297/300
3424/3424 [=====] - 4s 1ms/step - loss: 0.0022 - acc: 0.6037 - val_loss: 0.0071 - val_acc: 0.5841
Epoch 298/300
3424/3424 [=====] - 4s 1ms/step - loss: 0.0023 - acc: 0.5961 - val_loss: 0.0071 - val_acc: 0.5759
Epoch 299/300
3424/3424 [=====] - 4s 1ms/step - loss: 0.0022 - acc: 0.5926 - val_loss: 0.0071 - val_acc: 0.5864
Epoch 300/300
3424/3424 [=====] - 4s 1ms/step - loss: 0.0023 - acc: 0.5993 - val_loss: 0.0071 - val_acc: 0.5829
```

RMSE is used to evaluate the model as following:

$$\text{RMSE} = \sqrt{\text{mean\_squared\_error}(y_{\text{true}}, y_{\text{predict}})} * 96$$

$y_{\text{true}}$ : the target data passed to the fit method

$y_{\text{predict}}$ : the data predicted by your model.

Since we divided the train targets by 96, the result is multiplied by 96.

RMSE value = 4.8549610376358032

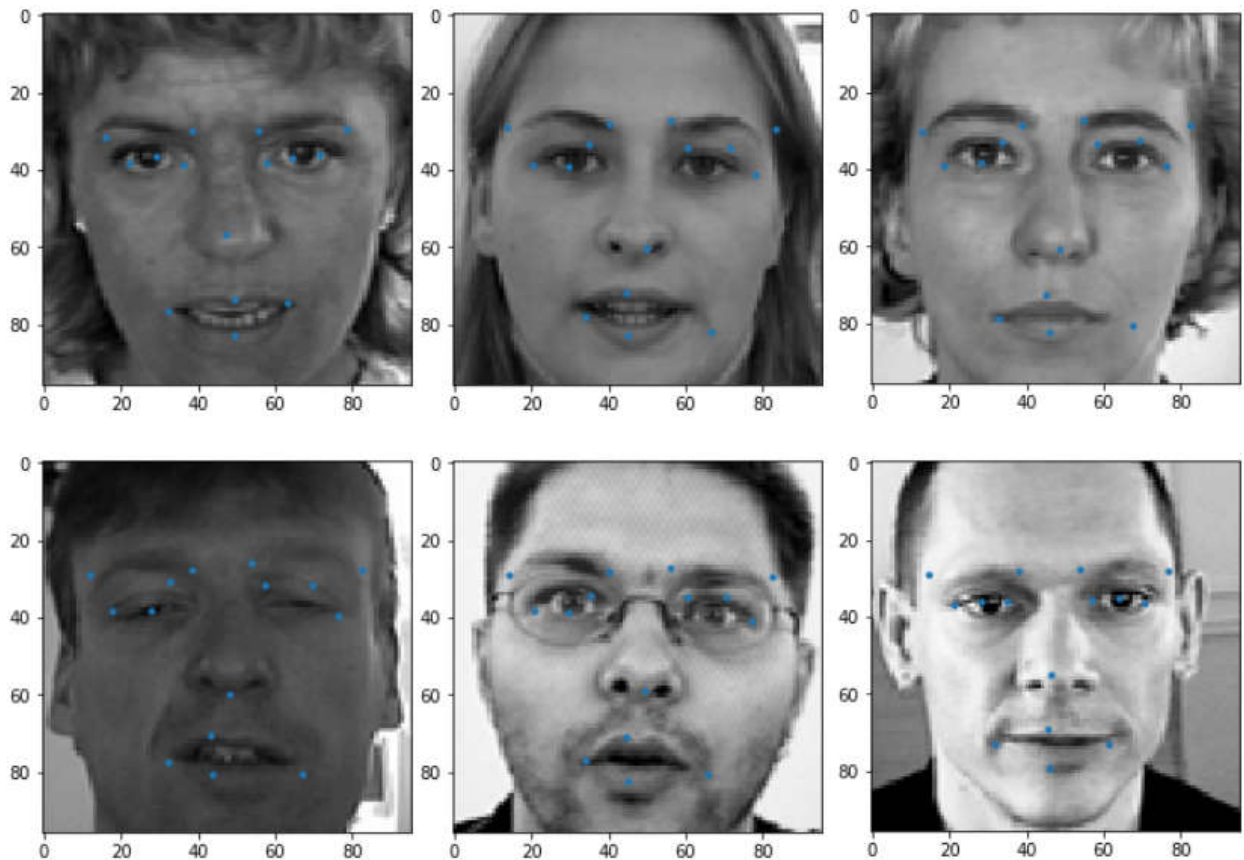
## Justification

- With a model trained only with 300 epochs I think I got a good result, the benchmark model trained over 1000 epochs which explains the overall loss.
- The loss in the 300 epochs is better compared to the same epochs in the benchmark model
- The val\_loss in the benchmark model is better than my val\_loss.
- Training the model with the augmented data 3424 samples, unlike the benchmark model which trained the model with only training data 1712, mine will be able to detect key points better.
- RMSE value for the Benchmark is better than mine.

## V. Conclusion

### Free-Form Visualization

This an example of the detected key points by the model. The testing result is good enough. May be there's some issue regarding the mouse, somehow the model detect only 3 true points.



## **Reflection**

The process used for this project can be summarized using the following steps:

- An initial problem and relevant, public datasets from Kaggle were found.
- The data was downloaded and preprocessed (normalized and augmented).
- A benchmark was created for the classifier.
- The classifier was trained using the data.
- Testing the model with the testing images.

I found choosing an architecture for the model was the most difficult. I trained the model with different architecture that I didn't even mention. Most of the model suffered from a constant val\_accuracy and val\_loss, it's like one number doesn't improve over 300 epochs. As for the most interesting aspects of the project, I'm very glad that I found the training database from Kaggle.

## **Improvement**

I know I mentioned in my proposal that I'll try using different approaches. Due to the large database, I couldn't train the model on my local machine, so I used Udacity dog breed classifier. I only got 4 hours left, I kept it for any modification you may ask for. As for the improvement there are different approach that can definitely increase the accuracy of the model as following:

- Using an Inception Model to solve overfitting problem.
- Using a transfer learning such as Xception.
- Try to increase the training data by rotating and shifting the images.