Applied Machine Learning
Final Project Report

Project Name: Facial Keypoints Detection

Team Members: Dikshya Mohanty
Krithika Vellore Prabhakar

Introduction:

Facial Keypoints Detection is the most challenging task in the field of computer vision and machine learning. Facial features vary from one individual to another and even for the same person, there is a lot of discrepancy at different views and angles. The objective of this task is to predict keypoint positions on face images which 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 This is a multi-domain problem.

It has a wide variety of applications. There are two main challenges for facial keypoints detection, one is that facial features have great variance between different people and different external factors, the other is that we have to reduce time complexity to achieve real-time keypoints detection. Link to this Kaggle Competition can be found here. We chose this topic because this is pretty new and interesting to the both of us. This is our first project in Computer Vision where we're employing Neural Networks.
Dataset to be used is that of the Facial Keypoint Recognition (Kaggle Competition 2019) at this Link. The dataset we use is from Kaggle's facial keypoints detection competition.

Background:

Traditional methods have explored feature extraction strategies includes texture-based and shape-based features and different types of graphic models to detect facial keypoints.

In the last decade a great deal of work has been done on this topic, of facial keypoint detection. Previously, it was common for detection to be based on local image features. For example, Liang et al. [2] searched faces at the component level using discriminative search algorithms and then ’adaptive boosting’. They used direction classifiers to help guide their discriminative algorithms. Others used random forest classifiers [3] or even SVMs [4] to detect facial components. Furthermore, cascading convolutional neural network architectures [8] and gabor filters [12] have been used to avoid local minima caused by ambiguity and data corruption in difficult image samples due to occlusions, large pose variations, and extreme lightings. From these papers it is clear that these approaches are achieving solid results, but further re-search is required to refine component detection. There are also a number of recent regression based approaches. For instance, Valstar et al. [7] used Markov Random Fields along with support vector regressors. It was found that the Markov Random Fields constrained the search space substantially, and the regressors learned the typical appearance of areas surrounding a given keypoint, allowing for rapid and robust detection despite variations in pose and expression. Dantone et al. [5] used local facial image patches to predict keypoints using conditional regression forests, which they found to outperform regression forests since they learn “the relations conditional to global face properties”. Parkhi et al. [10] trained2 a VGGNet [13] for the task of facial recognition, an architecture that secured first in the ImageNet Chal-lenge in 2014. One last example is Caoe et al. [6] who used random ferns as regressors on the entire face image as input. Predicted shapes were represented as linear combinations of training shapes.

[3] was published in 2017 gives an comparative study among the different algorithms that can be used to detect facial keypoints. First, the data is preprocessed and some initial analysis is done. Then they have used LCB (Local Binary Pattern) which is an operator used to describe the local texture features of images. Basic LCB features and the revised versions are explained as well as the extraction of LCB Feature Vector is made. It then uses PCA for dimensionality reduction. The paper compares 8 algorithms - KNN, Linear, Lasso, Elastic, Ridge, Decision tree, Neural Network on the basis of how well it detects the key points. RMSE is used for comparison which suggests the error rate in each of the model. Later, they re-run the codes on the pre-processed data by means of LBP and PCA, and get better results. All in all, the results depict CNN gets the best RMSE but it is very time consuming. Decision tree takes advantage of interpretability though it does not have low RMSE. Linear Regression methods are simple enough and only takes a little time to train but they are not recommended for real world applications since we need to pay much more attention to RMSE rather than algorithm complexity.

Instead of traditional CNNs, [4] dealt with addressed task using a block of pretrained Inception Model to extract intermediate features and using different deep structures to compute the final output vector. They build two models as baselines, which are realized based on simple neural network and convolutional neural network respectively. Then as the major part of this paper, the Inception Model will be introduced and analyzed in detail. In each layer of this model, they use multiple filters with different sizes 1 × 1, 3 × 3 and 5 × 5 instead of one filter in traditional cnn layers. This approach achieves so-called global sparsity since they split a huge amount of parameters from one-size filter up into several groups corresponding to different filter sizes. Meanwhile, it guarantees local density as well because each convolution operation can be realized in a traditional way. Performance was measured in the form of Loss and train/test time.

For the modeling phase in our project, we've implemented Inception cnn model which we have later proven that is better than the baseline model similar to the ones used in the research papers mentioned above. Additionally we have used dropout as regularization in the our model to overcome overfitting, which was not performed in the previous work.

Project:

Exploratory Data Analysis:

Facial Keypoint detection involves detecting points the following unique features per given image:

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, mouth_center_bottom_lip

The training dataset which consists of 7049 images. Each row contains the (x,y) coordinates for 15 keypoints, followed by image data as row-ordered list of pixels. There are 7049 images in total, each 4 of which is a 96 × 96 pixels image. Among all the images, 2140 out of them have ground truth positions for all 15 facial keypoints, which are used to as our dataset to train, validate and test the network. The training and test dataset are both labelled.
The test dataset here contains list of 1783 test images. Each row contains ImageId and image data as row-ordered list of pixels. The problem is to predict the (x, y) real-valued co-ordinates in the space of image pixels of the facial keypoints for a given face image.
The lookup-id dataset consists of 4 columns ('RowId', 'ImageId', 'FeatureName', 'Location') which is given as a sample submission format. 'Location' column is to be filled with our predictions of what a certain coordinate might be.

We perform Exploratory Data Analysis to get better insights of the dataset and make it fit for training. We check the dimension of the dataset and make sure it's in (# of examples) * (height of the image) * (width of the image) format.

```
# check shape of the dataset
print('Shape of training dataset :', images.shape)
```

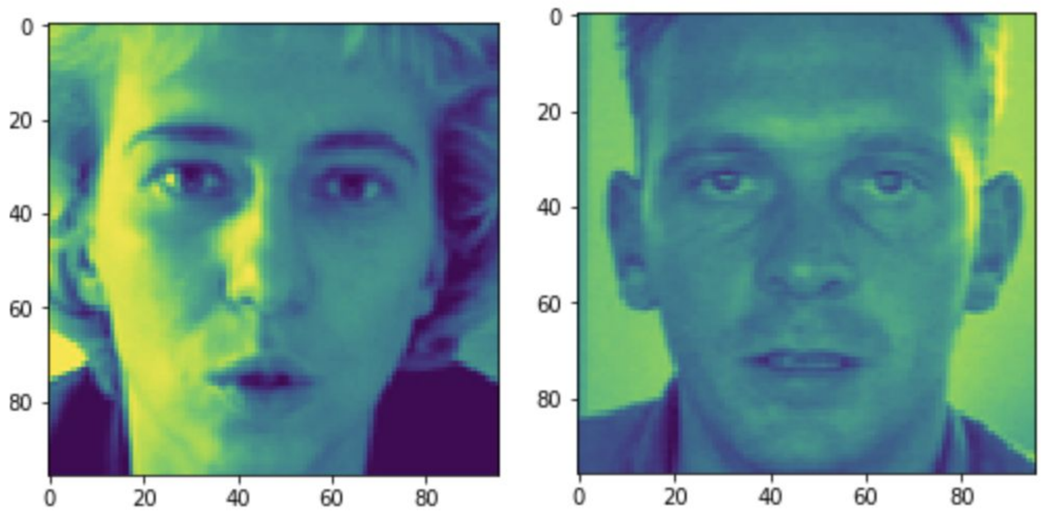
```
Shape of training dataset : (7049, 96, 96)
```

We move on to analyze how data in the 'Image' column in our training dataset looks like. It is given as 96 × 96 pixels represented as a matrix with pixels ranging from 0-255.

```
# lets print the image
images[0]
```

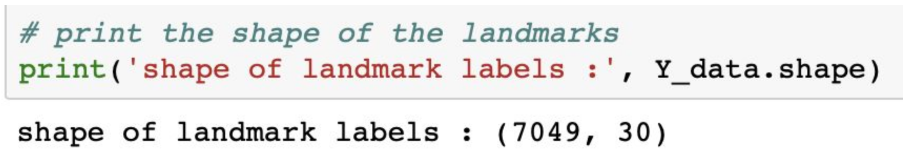
```
array([[238, 236, 237, ..., 250, 250, 250],
       [235, 238, 236, ..., 249, 250, 251],
       [237, 236, 237, ..., 251, 251, 250],
       ...,
       [186, 183, 181, ...,  52,  57,  60],
       [189, 188, 207, ...,  61,  69,  78],
       [191, 184, 184, ...,  70,  75,  90]])
```

We plot some of the images randomly that's given to us using the python package 'matplotlib.pyplot'.



Since the pixeled data is in the range 0-255, we normalize the dataset by dividing it by 255 for faster convergence as gradient descent runs faster on normalised dataset. We additionally add a new axis for channels, making it's shape (7049, 96, 96, 1).

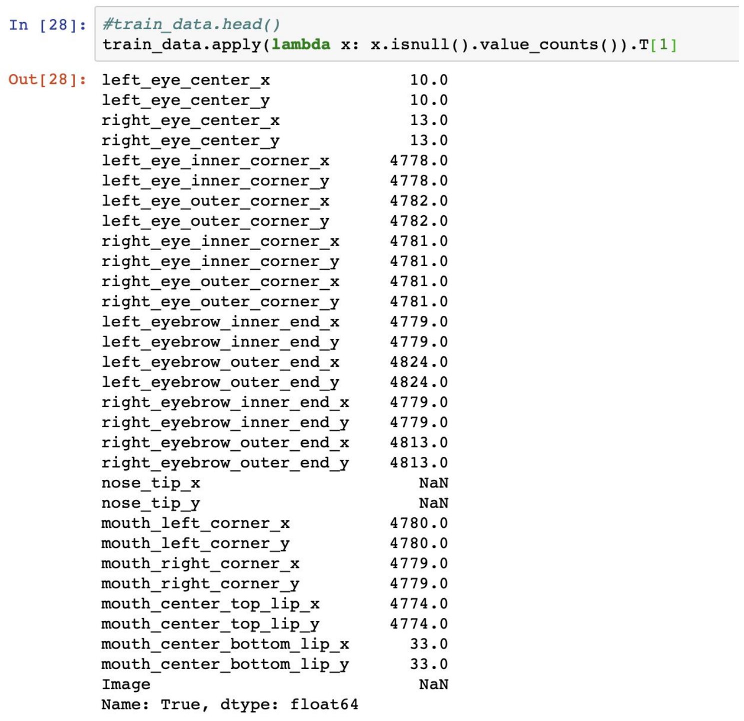
Now we move on to access the face key points given for the training dataset. We save it as a numpy array for easier data handling.



```
# print the shape of the landmarks
print('shape of landmark labels :', Y_data.shape)
```

```
shape of landmark labels : (7049, 30)
```

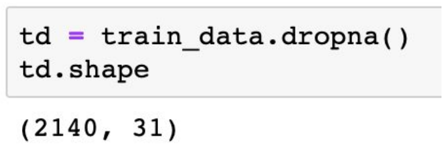
We analyze the presence of missing values in this. As predicted come across several missing values in the training data.



```
In [28]: #train_data.head()
         train_data.apply(lambda x: x.isnull().value_counts()).T[1]
```

```
Out[28]: left_eye_center_x             10.0
         left_eye_center_y             10.0
         right_eye_center_x            13.0
         right_eye_center_y            13.0
         left_eye_inner_corner_x     4778.0
         left_eye_inner_corner_y     4778.0
         left_eye_outer_corner_x     4782.0
         left_eye_outer_corner_y     4782.0
         right_eye_inner_corner_x    4781.0
         right_eye_inner_corner_y    4781.0
         right_eye_outer_corner_x    4781.0
         right_eye_outer_corner_y    4781.0
         left_eyebrow_inner_end_x    4779.0
         left_eyebrow_inner_end_y    4779.0
         left_eyebrow_outer_end_x    4824.0
         left_eyebrow_outer_end_y    4824.0
         right_eyebrow_inner_end_x   4779.0
         right_eyebrow_inner_end_y   4779.0
         right_eyebrow_outer_end_x   4813.0
         right_eyebrow_outer_end_y   4813.0
         nose_tip_x                     NaN
         nose_tip_y                     NaN
         mouth_left_corner_x         4780.0
         mouth_left_corner_y         4780.0
         mouth_right_corner_x        4779.0
         mouth_right_corner_y        4779.0
         mouth_center_top_lip_x      4774.0
         mouth_center_top_lip_y      4774.0
         mouth_center_bottom_lip_x     33.0
         mouth_center_bottom_lip_y     33.0
         Image                          NaN
         Name: True, dtype: float64
```

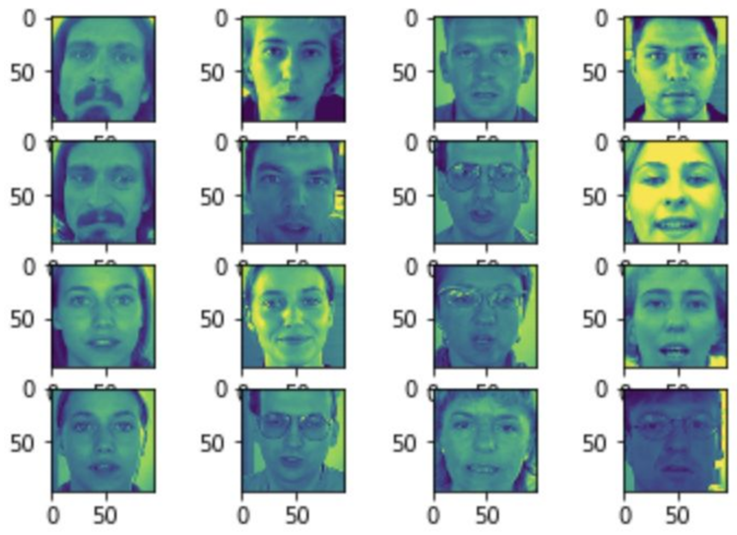
If we were to access only complete cases i.e. drop all observations with missing values in it, we'd be left with only 2140 observations, which is a very small number for training.



```
td = train_data.dropna()
td.shape
```

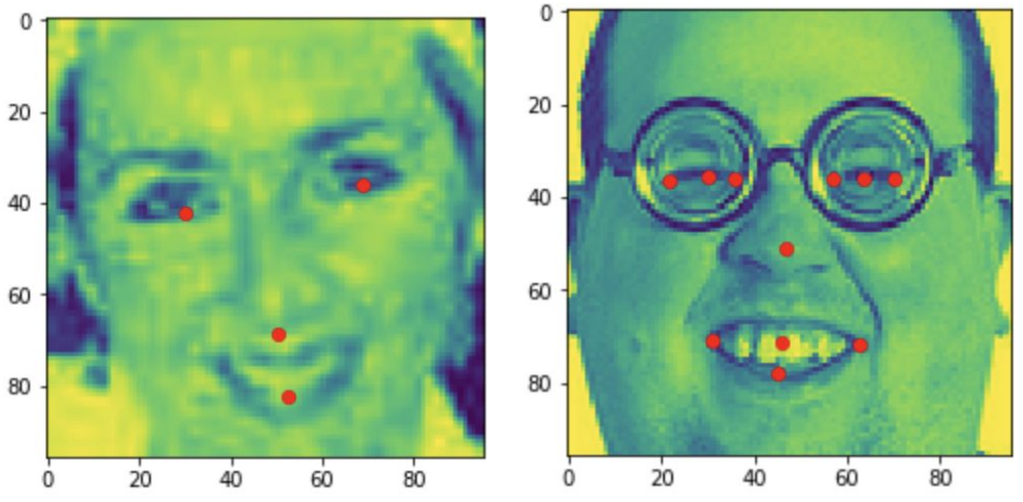
```
(2140, 31)
```

To proceed with imputation, we look at the data distribution.

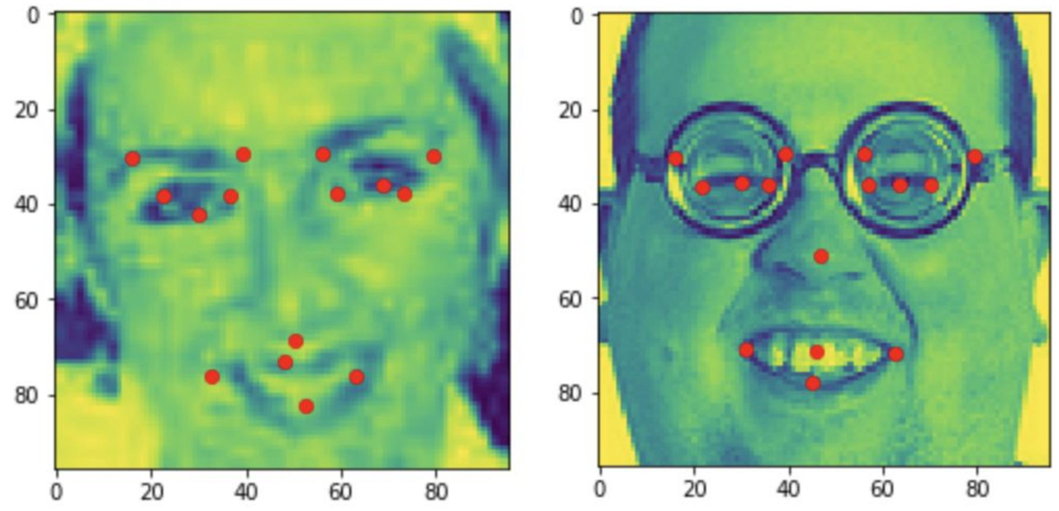


Initially we try imputing all the missing values(NaNs) with zeroes. However, this doesn't work very well for image data. The next step that we perform is replace the missing values with mean of that key-point across the entire training data, which gives us a much better result.

Before imputation (Missing values present) :



After imputation (Replacing missing values with) :



Approach:
We shuffled the data and split it randomly. We performed a 80%, 20% split for train and validation sets. We trained our model on the training set and tuned the hyperparameters like learning rate, regularization parameter, batch size etc. on the validation set. We finally used the test set to predict the key points using our model.

For the Modeling phase, we've decided to create an ensemble model using inception models. Inception v3 is a widely-used image recognition model. It's made up of symmetric and asymmetric building blocks, including convolutions, average pooling, max pooling, concats, dropouts, and fully connected layers. Batchnorm is used extensively throughout the model and applied to activation inputs. Loss is computed via Softmax.

Implementing Convolutional Neural Network:

We start with creating a single step of convolution, where the filter is applied only to a single portion of the input. This will be used to build a convolutional unit, which:

- Takes an input volume
- Applies a filter at every position of the input
- Outputs another volume (usually of different size)

Forward Pass:

In the forward pass, you will take many filters and convolve them on the input. Each 'convolution' gives you a 2D matrix output. You will then stack these outputs to get a 3D volume.

The formula relating output shape of convolution to the input shape is:

$$n_H = \lfloor \frac{n_{H_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$
$$n_W = \lfloor \frac{n_{W_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$
$$n_C = \text{number of filters used in the convolution}$$

where, n_H = hight of the output
n_W = width of the output
n_C = number of output channels
n_H_prev = height of the privious layer
n_W_prev = width of the previous layer
n_C_prev = number of channels in the previous layer
pad = number of padding units
stride = stride of convolution

Pooling layer:

The pooling (POOL) layer reduces the height and width of the input. It helps reduce computation, as well as helps make feature detectors more invariant to its position in the input. The two types of pooling layers are:

- Max-pooling layer: slides an (*f*,*f*f,f) window over the input and stores the max value of the window in the output.
- Average-pooling layer: slides an (*f*,*f*f,f) window over the input and stores the average value of the window in the output.

These pooling layers have no parameters for backpropagation to train. However, they have hyperparameters such as the window size *f*f. This specifies the height and width of the fxf window you would compute a max or average over.

As there's no padding, the formulas binding the output shape of the pooling to the input shape is:

$$n_H = \lfloor \frac{n_{H_{prev}} - f}{stride} \rfloor + 1$$
$$n_W = \lfloor \frac{n_{W_{prev}} - f}{stride} \rfloor + 1$$
$$n_C = n_{C_{prev}}$$

Backpropagation:

In modern deep learning frameworks, we only have to implement the forward pass, and the framework takes care of the backward pass, so most deep learning engineers don't need to bother with the details of the backward pass. The backward pass for convolutional networks is complicated.

Convolutional layer backward pass

Computing dA:

This is the formula for computing dA with respect to the cost for a certain filter Wc and a given training example:

$$dA += \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} W_c \times dZ_{hw}$$

Where Wc is a filter and dZhw is a scalar corresponding to the gradient of the cost with respect to the output of the conv layer Z at the hth row and wth column (corresponding to the dot product taken at the ith stride left and jth stride down). Note that at each time, we multiply the the same filter Wc by a different dZ when updating dA. We do so mainly because when computing the forward propagation, each filter is dotted and summed by a different a_slice. Therefore when computing the backprop for dA, we are just adding the gradients of all the a_slices.

Computing dW:

This is the formula for computing dWc (dWc is the derivative of one filter) with respect to the loss:

$$dW_c += \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} a_{slice} \times dZ_{hw}$$

Where a_slice corresponds to the slice which was used to generate the activation Zij. Hence, this ends up giving us the gradient for W with respect to that slice. Since it is the same *W*W, we will just add up all such gradients to get dW.

Computing db:

This is the formula for computing *db*db with respect to the cost for a certain filter Wc:

$$db = \sum_h \sum_w dZ_{hw}$$

In basic neural networks, db is computed by summing *dZ*dZ. In this case, you are just summing over all the gradients of the conv output (Z) with respect to the cost.

Pooling layer - Backward pass

Next, let's see the backward pass for the pooling layer, starting with the MAX-POOL layer. Even though a pooling layer has no parameters for backprop to update, you still need to backpropagation the gradient through the pooling layer in order to compute gradients for layers that came before the pooling layer.

Max pooling - backward pass
Before jumping into the backpropagation of the pooling layer, you are going to build a helper function called create_mask_from_window() which does the following:

$$X = \begin{bmatrix} 1 & 3 \\ 4 & 2 \end{bmatrix} \quad \rightarrow \quad M = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

As you can see, this function creates a "mask" matrix which keeps track of where the maximum of the matrix is. True (1) indicates the position of the maximum in X, the other entries are False (0). You'll see later that the backward pass for average pooling will be similar to this but using a different mask.

The reason we keep track of the position of the max is because this is the input value that ultimately influenced the output, and therefore the cost. Backprop is computing gradients with respect to the cost, so anything that influences the ultimate cost should have a non-zero gradient. So, backprop will "propagate" the gradient back to this particular input value that had influenced the cost.

Average pooling - Backward pass:

In max pooling, for each input window, all the "influence" on the output came from a single input value--the max. In average pooling, every element of the input window has equal influence on the output. So to implement backprop, you will now implement a helper function that reflects this.

For example if we did average pooling in the forward pass using a 2x2 filter, then the mask we will use for the backward pass will look like:

$$dZ = 1 \quad \rightarrow \quad dZ = \begin{bmatrix} 1/4 & 1/4 \\ 1/4 & 1/4 \end{bmatrix}$$

This implies that each position in the dZ matrix contributes equally to output because in the forward pass, we took an average.

After getting a hold on the dataset, we move to create placehoders. TensorFlow requires that you create placeholders for the input data that will be fed into the model when running the session. We will initialize weights/filters W1 and W2 using tf.contrib.layers.xavier_initializer(seed = 0).

Performing Forward propagation:

In TensorFlow, there are built-in functions that carry out the convolution steps for us.

- tf.nn.conv2d(X,W1, strides = [1,s,s,1], padding = 'SAME'): given an input X and a group of filters W1, this function convolves W1's filters on X. The third input ([1,f,f,1]) represents the strides for each dimension of the input (m, n_H_prev, n_W_prev, n_C_prev). You can read the full documentation.
- tf.nn.max_pool(A, ksize = [1,f,f,1], strides = [1,s,s,1], padding = 'SAME'): given an input A, this function uses a window of size (f, f) and strides of size (s, s) to carry out max pooling over each window. You can read the full documentation.
- tf.nn.relu(Z1): computes the elementwise ReLU of Z1 (which can be any shape). You can read the full documentation.
- tf.contrib.layers.flatten(P): given an input P, this function flattens each example into a 1D vector it while maintaining the batch-size. It returns a flattened tensor with shape [batch_size, k]. You can read the full documentation.
- tf.contrib.layers.fully_connected(F, num_outputs): given a the flattened input F, it returns the output computed using a fully connected layer. You can read the full documentation.

In the last function above (tf.contrib.layers.fully_connected), the fully connected layer automatically initializes weights in the graph and keeps on training them as you train the model. Hence, you did not need to initialize those weights when initializing the parameters.

Cost function: We have implemented the compute cost function using simple squared error. Assumption : Every observation is independent of each other.

Once all the individual tasks are defined, we combine everything to create our model.

The tasks are:

- create placeholders
- initialize parameters
- forward propagate
- compute the cost
- create an optimizer

We have used the adam optimizer for faster convergence.

Finally we have created a session and run a for loop for num_epochs, get the mini-batches, and then for each mini-batch you will optimize the function.
We performed zero padding i.e., adding zeros around the border of the image.
The main benefits of padding are the following:

- It allows you to use a CONV layer without necessarily shrinking the height and width of the volumes. This is important for building deeper networks, since otherwise the height/width would shrink as you go to deeper layers. An important special case is the "same" convolution, in which the height/width is exactly preserved after one layer.
- It helps us keep more of the information at the border of an image. Without padding, very few values at the next layer would be affected by pixels as the edges of an image.

We initially developed an inception model with the following layout:
CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN -> FULLYCONNECTED -> OUTPUT
This model consists of 6 layers, viz - 2 convolution layers followed by 2 pooling layers respectively, we will add 1 fully connected layers to it and finally one output layer.

For our final model, we implemented the forward_propagation function below to build the following model: CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN -> FULLYCONNECTED -> DROPOUT REGULARIZATION -> OUTPUT.

The parameters used for all the steps are as follows:
- Conv2D: stride 1, padding is "SAME"
- ReLU
- Max pool: Use an 8 by 8 filter size and an 8 by 8 stride, padding is "SAME"
- Conv2D: stride 1, padding is "SAME"
- ReLU
- Max pool: Use a 4 by 4 filter size and a 4 by 4 stride, padding is "SAME"
- Flatten the previous output.
- FULLYCONNECTED (FC) layer: we will apply a fully connected layer without an non-linear activation function.

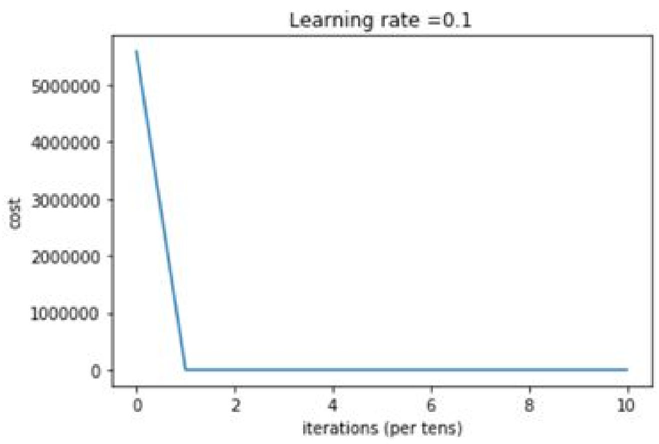
Hyperparameter tuning:

After training the model and optimizing the parameters, we have tested its performance on the validation set. We tried and tested with different values of the learning_rate, minibatch_size, keep_prob and observe the metric in each of the case. Finally we will considered the model which performed best on the validation set and then we tested it on the test set.

Results

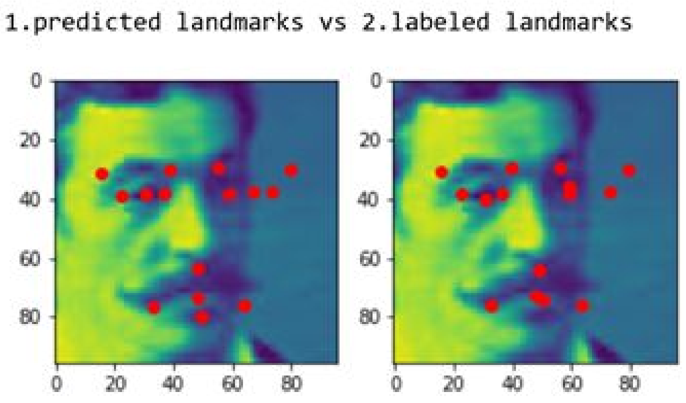
When we first created a session to train the model, we set learning rate = 0.1 and number of epoch = 11 and we obtained a validation cost of 25.843.

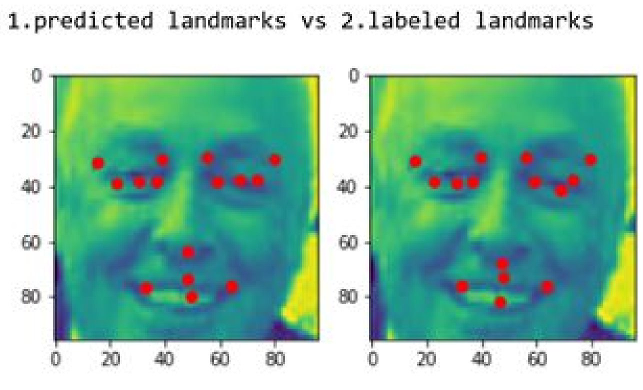
```
Cost after epoch 0: 5582641.820280
Cost after epoch 5: 6.701166
Cost after epoch 10: 6.701166
```



```
Validation cost = 25.843727
```

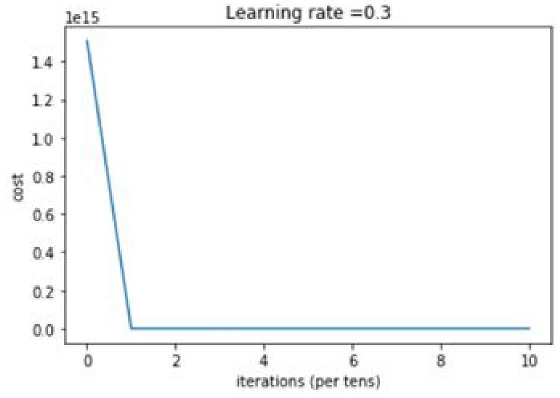
Looking at differences between the predicted landmarks and labeled landmarks of the same image:

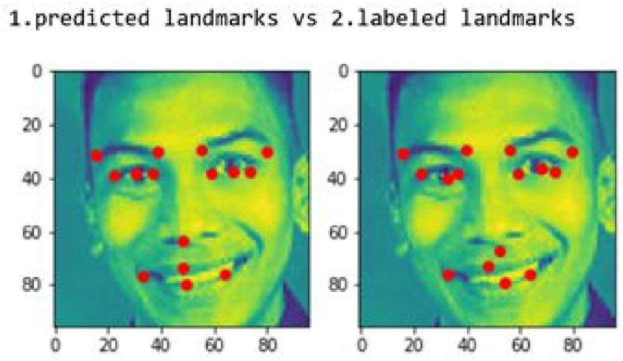




Then, we started to perform hyperparameter tuning with different learning rates and number of epochs to improve the accuracy and reduce the validation cost. The results of tuning are shown below:

```
Cost after epoch 0: 1507465192780345.750000
Cost after epoch 5: 6.701166
Cost after epoch 10: 6.701166
```

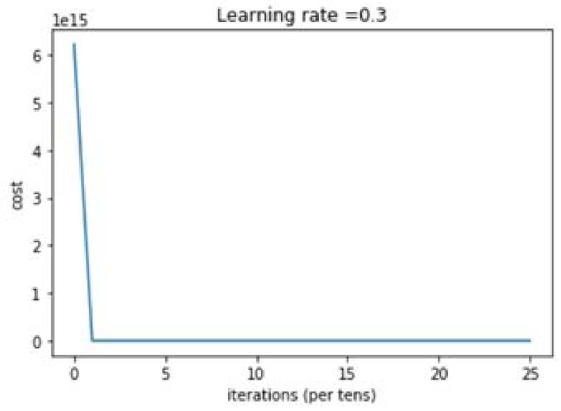


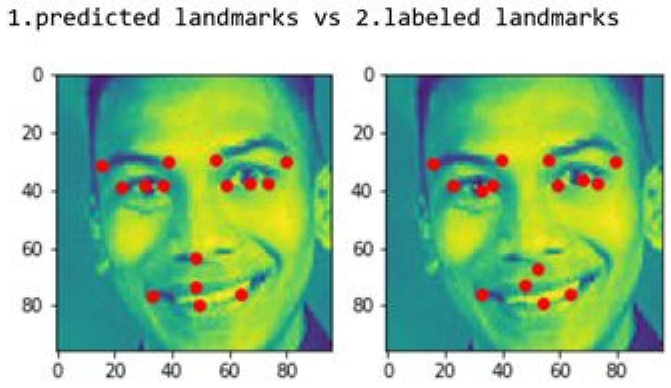


```
Validation cost = 7.658823
```

1. While trying to build the network one layer deep.

```
Cost after epoch 0: 6216016363644366.000000
Cost after epoch 5: 6.701166
Cost after epoch 10: 6.701166
Cost after epoch 15: 9.355445
Cost after epoch 20: 6.701167
Cost after epoch 25: 6.701166
```

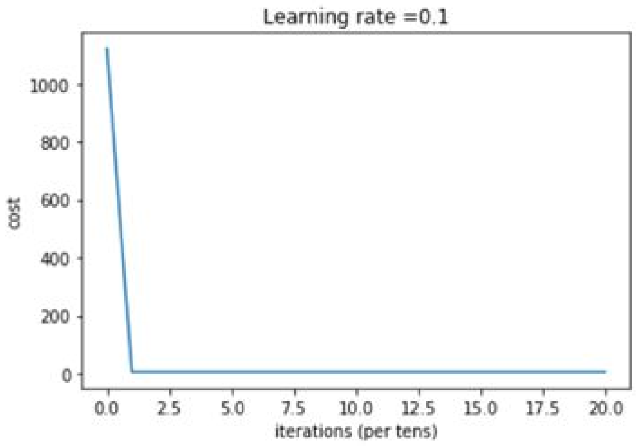




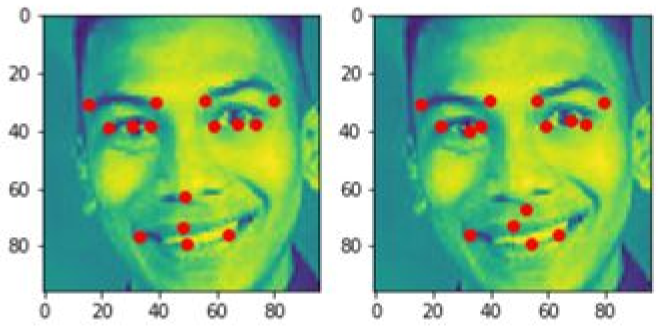
```
Validation cost = 7.6587477
```

2. Trying to increase the batch size.

```
Cost after epoch 0: 1121.172742
Cost after epoch 5: 5.860436
Cost after epoch 10: 5.860436
Cost after epoch 15: 5.860436
Cost after epoch 20: 5.860436
```



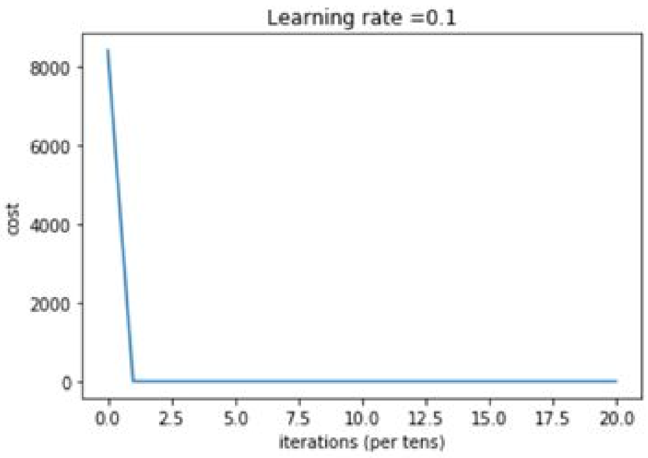
```
1.predicted landmarks vs 2.labeled landmarks
```



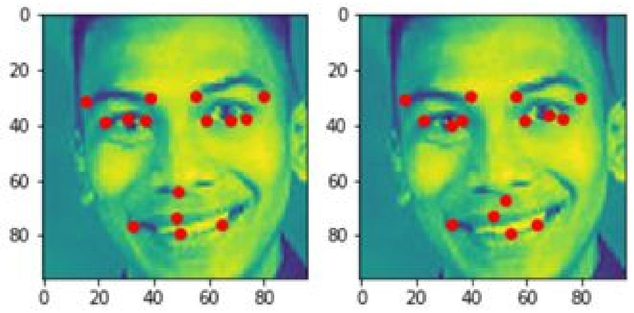
```
Validation cost = 7.546961
```

3. Trying to reduce the minibatch size:

```
Cost after epoch 0: 8420.638247
Cost after epoch 5: 5.060323
Cost after epoch 10: 5.060323
Cost after epoch 15: 5.125229
Cost after epoch 20: 5.119562
```



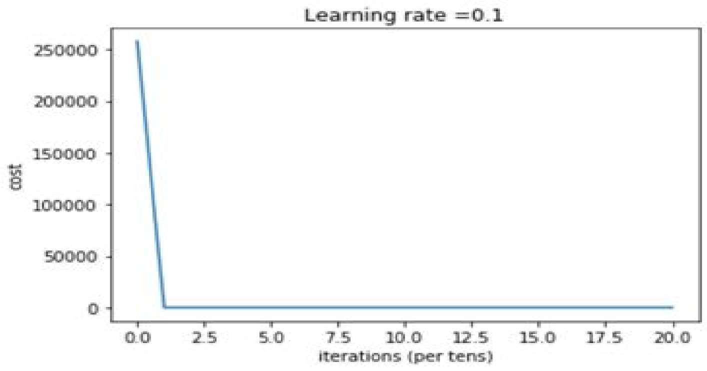
```
1.predicted landmarks vs 2.labeled landmarks
```



After trying with all the different possibilities, we have chosen a model with the smallest validation cost.

```
Validation cost = 7.738833
```

```
Cost after epoch 0: 257907.596554
Cost after epoch 5: 5.860436
Cost after epoch 10: 5.860436
Cost after epoch 15: 5.860436
Cost after epoch 20: 5.860436
```



```
Validation cost = 7.546961
```

Hence, the smallest validation cost we can get is 7.5469 with learning rate = 0.1 and number of epochs = 20.

Team member contributions:

Name	Tasks performed
Dikshya Mohanty	Exploratory Data Analysis, building the Inception model along with the baseline, developing the full CNN model with 13 layers, Implementing Tensorflow, Testing, Validation, Hyperparameter tuning and documentation (50%)
Krithika Vellore Prabhakar	Exploratory Data Analysis, building the Inception model along with the baseline, developing the full CNN model with 13 layers, Implementing Tensorflow, Testing, Validation, Hyperparameter tuning and documentation (50%)

Summary:

The purpose of Facial Keypoint detection is to detect points for the unique features per given image. With a training dataset of 7049 images and 15 key points, we performed EDA, replacing the missing values with mean. Then we split the data into train and validation set (80-20). For our baseline model, we used one layer Neural network and 6-layer NN model (i.e. inception model) with convolution layers, followed by Pooling layer and one fully connected layer. However, the validation score of this model was too high. Hence, we implemented a more complex, deep-layered inception model and got a much lower validation cost. Then we performed hyper-parameter tuning on the model by adding one more deep layer, increasing the batch size and reducing the minibatch size. For every model that we implemented, we made subsequent submissions to kaggle to check the performance of the model on Test data. The lowest root mean squared error that we obtained from our final model is 3.98767.

Conclusion:

In this paper, we have focused on the task of detecting facial keypoints when given raw facial images. Specifically, for a given 96 ∗ 96 image, we would predict 15 sets of (x, y) coordinates for facial keypoints. Two traditional deep structures, One Hidden Layer Neural Network and Convolutional Neural Network, are implemented as our baselines.
We further explored a sparsely connected Inception Model to reduce computational complexity to fit the requirements for detecting facial keypoints. Experiments which conducted on real-world kaggle dataset have shown the effectiveness of deep structures, especially Inception Model.

As for our future work, we can explore from these few aspects,

1. We have already shown the effectiveness of the Inception Model when used as the pretrained model, but the performance has a chance to improve if we train from the scratch.
2. As we can see from the results, using deep structures can increase time complexity when compared to other start-of-art methods but the results have been shown to improve a lot much. What we can do in the future is to design a deep structure specifically for this task to

further improve the performance.
3. Different resolution can greatly affect the results of the facial keypoints detection, thus what we can do is try to reduce the resolution of our given raw images to see the variance of the performance to further evaluate our model.

References:

[1] Facial Keypoint Detection Competition. Kaggle, 7 May 2013. Web. 31 Dec. 2016. https://www.kaggle.com/c/facial-keypoints-detection

[2] Liang, Lin, et al. “Face alignment via component-based discriminative search.” Computer Vision–ECCV 2008. Springer Berlin Heidelberg, 2008. 72-85.

[3] Amberg, Brian, and Thomas Vetter. “Optimal landmark detection using shape models and branch and bound.” Computer Vision (ICCV), 2011 IEEE International Conference on. IEEE, 2011.

[4] Belhumeur, Peter N., et al. “Localizing parts of faces using a consensus of exemplars.” Pattern Analysis and Machine Intelligence, IEEE Transactions on 35.12 (2013): 2930-2940.

[5] M. Dantone, J. Gall, G. Fanelli, and L. J. V. Gool. Real-time facial feature detection using
conditional regression forests. In Proc. CVPR, 2012.

[6] D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image
classification. In Proc. CVPR, 2012

[7] M. Valstar, B. Martinez, X. Binefa, and M. Pantic. Facial point detection using boosted regression and graph models. In Proc. CVPR, 2010.

[8] Sun, Yi, Xiaogang Wang, and Xiaoou Tang. “Deep convolutional network cascade for facial point detection.” Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2013.

[9] Ciresan, Dan, Ueli Meier, and J ¨urgen Schmidhuber. “Multi-column deep neural networks for image classification.” Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Confer-

ence on. IEEE, 2012.

[10] Parkhi, Omkar M., Andrea Vedaldi, and Andrew Zisserman. "Deep face recognition." Proceedings of the British Machine Vision 1.3 (2015): 6.

[11] Nouri, Daniel. 2015. Github. Kaggle Facial Keypoints Detection tutorial. Available from https://github.com/dnouri/kfkdtutorial/blob/master/kfkd.py

[12] Serre, Thomas, Lior Wolf, and Tomaso Poggio. "Object recognition with features inspired by visual cortex." Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on. Vol. 2. IEEE, 2005.

[13] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprintarXiv:1409.1556 (2014).