CS583: Course Final Project

Facial Keypoints Detection Using a Convolutional Neural Network

Sarita Hedaya

December 1st, 2019

1 Summary

I participated in an inactive Kaggle competition with late submission, for facial keypoints detection. Facial keypoints include the locations of the landmarks of the face such as eyes, nose, and lips, in x and y pixel coordinates. I experimented with several neural network architectures, and the final model I chose is a custom convolutional neural network that worked best in practice. I implemented the convolutional neural network using Keras, and ran the code on Google Colab using their GPU runtime. Performance in this competition is evaluated as the root mean squared error of the predicted position of each landmark, versus the true position. My final score is 2.89145, positioning me at 56 among 175 competitors in the private leaderboard, and 54 among 175 competitors in the public leaderboard.

2 Problem Description

Problem. The problem in this competition is to predict the x and y coordinates of the facial key points, using a training set. This is a regression problem, where the output is given by 30 scalars, the x and y coordinates of 15 keypoints. The competition can be found at https://www.kaggle.com/c/facial-keypoints-detection.

Data. The data provided consists of four files.

`training.csv`: list of training 7049 images. Each row contains the (x,y) coordinates for 15 keypoints, and image data as row-ordered list of pixels.

`test.csv`: list of 1783 test images. Each row contains ImageId and image data as row-ordered list of pixels

`SampleSubmission.csv`: list of 27124 keypoints to predict. Each row contains a RowId, and a space for the predicted coordinate.

`IdLookupTable.csv`: Used as reference to match a RowId from SampleSubmission with its ImageId and FeatureName, so we can predict its Location. FeatureName are

```
left_eye_center_x, right_eyebrow_outer_end_y, etc.
```

Location is what we need to predict, which is the pixel coordinate of the keypoint in the image.

Challenges. The training set consists of 7049 images with up to 15 keypoints each. Some images had only 4 keypoints labeled, making the dataset much smaller than I initially thought. Additionally, the training data had holes (missing values) in 4909 out of 7049 rows; spread among 28 out of the 30 features. That is, almost 70 percent of my data points were incomplete.

I considered two options for dealing with these holes:

1. Eliminate rows with missing values
2. Fill missing values with average for that column

It was not viable either to remove 28 features or get rid if 4909 data points, as it would mean losing a lot of the already scarce training data. I choose to fill the missing values by placing the average for that feature in the holes. I deemed this a good solution since adding the mean value will not change the mean, and the standard deviation was low enough such that completing the data with the mean would not add too much noise, harming the network's ability to learn.

3 Solution

Model. The model I finally chose is a custom convolutional neural network inspired by NaimishNet, another convolutional neural network also used for keypoints detection. A description of NaimishNet can be found in the paper online: https://arxiv.org/pdf/1710.00977.pdf. My model is different from NaimishNet in several aspects. First, it replaces conventional dropout with batch normalization. Second, it adds an additional convolutional layer of 64 (3,3) filters. Third, I added additional fully connected layers and Dropout at the end of the model.

Below is the model summary.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 93, 93, 32)        544
_________________________________________________________________
batch_normalization_1 (Batch (None, 93, 93, 32)        128
_________________________________________________________________
activation_1 (Activation)    (None, 93, 93, 32)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 46, 46, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 44, 44, 64)        18496
```

```
_________________________________________________________________
batch_normalization_2 (Batch (None, 44, 44, 64)        256
_________________________________________________________________
activation_2 (Activation)    (None, 44, 44, 64)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 22, 22, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 20, 20, 64)        36928
_________________________________________________________________
batch_normalization_3 (Batch (None, 20, 20, 64)        256
_________________________________________________________________
activation_3 (Activation)    (None, 20, 20, 64)        0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 10, 10, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 9, 9, 128)         32896
_________________________________________________________________
batch_normalization_4 (Batch (None, 9, 9, 128)         512
_________________________________________________________________
activation_4 (Activation)    (None, 9, 9, 128)         0
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 4, 4, 256)         33024
_________________________________________________________________
batch_normalization_5 (Batch (None, 4, 4, 256)         1024
_________________________________________________________________
activation_5 (Activation)    (None, 4, 4, 256)         0
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 2, 2, 256)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               262400
_________________________________________________________________
dense_2 (Dense)              (None, 128)               32896
_________________________________________________________________
dense_3 (Dense)              (None, 30)                3870
=================================================================
Total params: 423,230
Trainable params: 422,142
Non-trainable params: 1,088
```

Implementation. I implemented the convolutional neural network using Keras. I ran the code in Google Colab. Using Colab's GPU runtime, my model trains in approx. 8 minutes. My code can be found at my GitHub page using this link:

.

Settings. This is a regression problem where we are tasked to predict 30 scalars per image. The scalars represent the pixel coordinates of the keypoints. Since closer predictions are exponentially better than farther predictions, i.e. being 2 pixels away is more than twice as good as being 4 pixels away, I chose mean squared error for the loss function. In addition, the competition uses root mean squared error as the score, so optimizing for mean squared error was the best choice.

The optimizer I used is Adam optimizer, with a learning rate of 0.001 that decreases by a factor of 0.1 after 9 epochs without validation loss improvement. The best hyperparameters as found in practice were a batch size of 64 images, trained over 70 epochs, with batch normalization in all convolutional layers.

Advanced tricks. I used an automatic learning rate reducer. In practice I was noticing that if I used too large of a learning rate, the loss would diverge in later epochs, but if I used too small a learning rate, the loss will converge too slowly. Therefore, a reducer allowed me to achieve convergence in the least number of epochs, and consequently the smallest amount of training time.

Cross-validation. Due to the lack of local reliable GPU, I performed cross-validation by randomly partitioning my training set in an 80% - 20% distribution, where the larger portion was used for training and the smaller portion for validation.

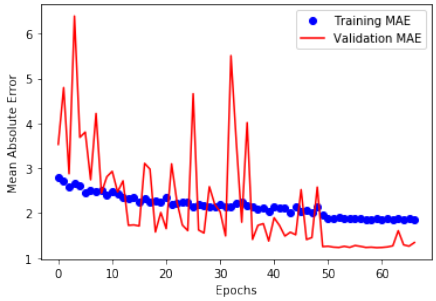
Figure 1 and Figure 2 plot the convergence curves on 80% training data and 20% validation data.

4 Compared Methods

Small Fully Connected Network I started with a very simple sequential model with a two fully connected hidden layers (128 and 64 nodes, respectively). Given that the dataset was small, I used this initial simple model to gauge when will overfitting begin to appear, and how far down the leaderboard such a simple model would get me. In addition, I used this model as a baseline to compare future results to.

As illustrated in Figure 3, the simple model does not overfit on the validation data even after training reaches a plateau around epoch 400. I will assume that plateau is due to reaching the learning capacity of this simple model. To confirm my suspicion I increased the model capacity, to see if, with proper regularization, I can reach lower errors.

a) The mean absolute error in the training and validation set.



b) The loss on the training set and validation set.

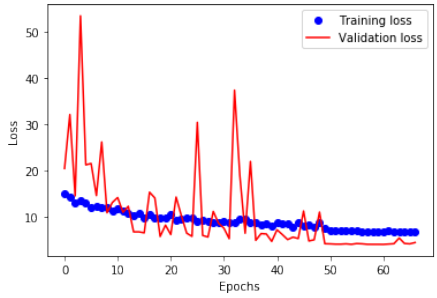
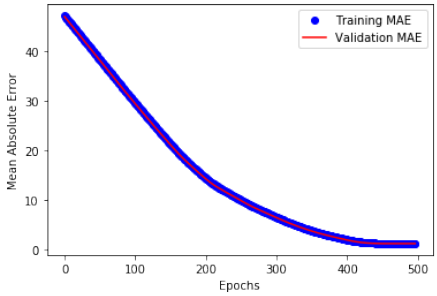


Figure 1: The convergence curves for the chosen CNN model

a) The mean absolute error on the training set and validation set



b) The loss on the training set and validation set.

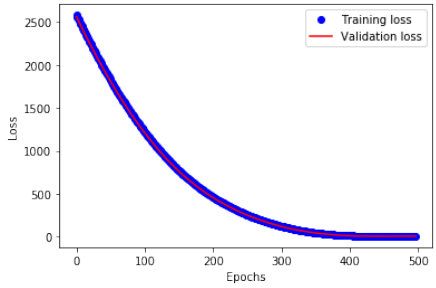


Figure 2: The convergence curves for the Small Fully Connected model

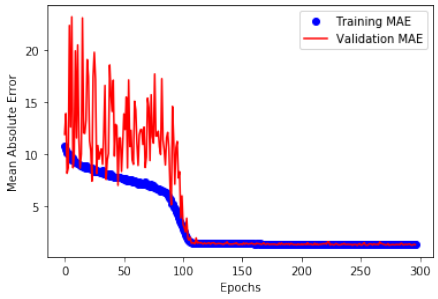
The small 2 layer fully connected network achieves a Root Mean Squared Error of 3.96 on the competition. This is ranking 141 out of 175 competitors.

Medium Fully Connected Network The second submission I made to the competition was a slightly more complex 7-layer fully connected neural network, amounting to 2.5 million parameters. Since the previous attempt appeared to reach the maximum learning capacity of the simple model, I added five additional layers with dropout regularization, and kept the RMSprop optimizer. Several iterations on the cross validation set allowed me to properly tune the dropout and the layer dimensions.

The medium complexity 7 layer fully connected network achieves a Root Mean Squared Error of 3.65 on the competition. This is ranking 99 out of 175 competitors. See Figure 3 for convergence curves.

This second attempt performed much better than the simple model, which ranked 141/175 in the competition. Judging by the plots of training and validation loss in figure 3, the model does not overfit at all during training, especially considering that I only regularized 20% of the nodes on only the two layers with most parameters.

a) The mean absolute error on the training set and validation set



b) The loss on the training set and validation set.

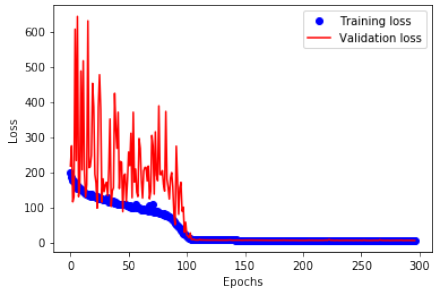


Figure 3: The convergence curves for the Medium sized Fully Connected model

Large Fully Connected Network The third method I attempted was an even more complex 16-layer fully connected neural network with approx. 3.3 million parameters. Since the previous attempt did not appear to overfit, I added nine additional layers with dropout regularization of 20% in all layers, and kept the RMSprop optimizer.

I attempted several different "deeper" fully connected models. Some with a large layers of 512 nodes at the beginning, and some with at most 128 nodes per layer. I tried gradually decreasing the size of the layers, or maintaining them at a constant size. No matter what I tried, during training the validation loss was oscillating, and not decreasing. This larger model was overfitting even with lots of regularization. I concluded that a fully connected model may not be the best solution to this problem. Given that this is a vision problem, I proceeded to attempt a Convolutional Neural Network to solve it.

NaimishNet At this point I decided to do more research on the problem of keypoints detection and found a paper that addressed the same problem, on the same dataset. You can read more on the NaimishNet paper here: https://arxiv.org/pdf/1710.00977.pdf. I replicated the network presented in that paper by building it using a Keras sequential model.

This convolutional neural network achieves a RMSE score of 3.35 on the competition, which is ranking 80/175. This is a nice improvement from the 6 layer fully connected network which had ranked 99/175. See convergence curves in Figure 4.

I did some hyperparameter tuning on the NaimishNet to find that the Adam optimizer oscillated less than the RMSprop optimizer.

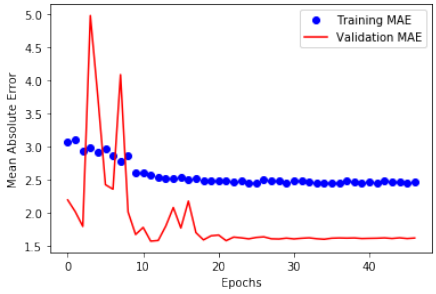
5 Outcome

The final is a custom build upon the NaimishNet. This model achieves position 54/175 that is top 30th percentile. Since this was a competition with late submission, my score does not appear in the leaderboard. Nevertheless, I can position myself in the leaderboard as if it were an active competition. The screen shots of the competition results are in Figures 6 and 7.

6 References

Agarwal, Naimish, et al. "Facial Key Points Detection Using Deep Convolutional Neural Network - NaimishNet." ArXiv.org, 3 Oct. 2017, arxiv.org/abs/1710.00977.

a) The mean absolute error on the training set and validation set



b) The loss on the training set and validation set.

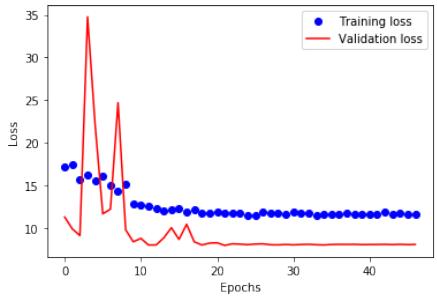


Figure 4: The convergence curves for the NaimishNet model

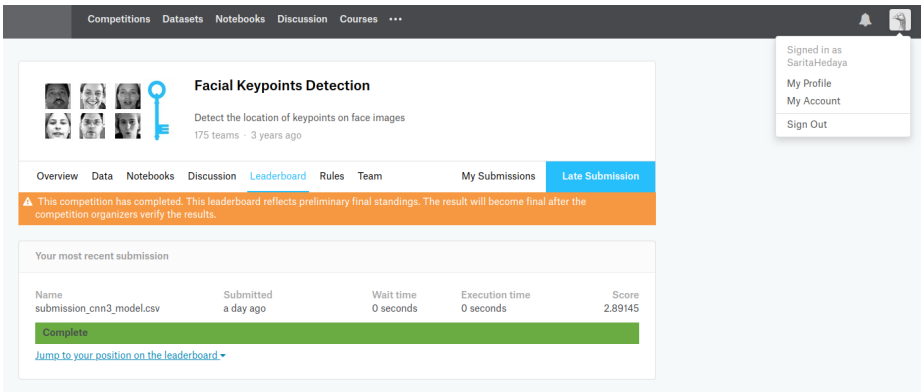


Figure 5: Kaggle screenshot showing my score and my account ID as proof.

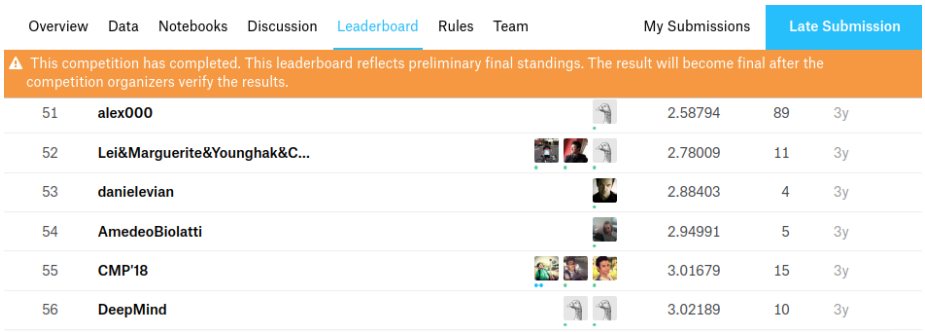


Figure 6: Leaderboard showing that my score of 2.89145 would have attained position 54.