# CS 559 - Deep Learning Homework Assignment

Yusuf Dalva
Computer Engineering Department
Bilkent University
Ankara, Turkey
Email: yusuf.dalva@bilkent.edu.tr

Said Fahri Altındiş
Computer Engineering Department
Bilkent University
Ankara, Turkey
Email: fahri.altindis@bilkent.edu.tr

## I. Introduction

In the recent years, with the recent advances in deep learning studies most of the tasks that seem complex for traditional vision faced significant advances. Together with the significant amount of studies that uncovered and solved many problems in the area, the availability of highly optimized deep learning frameworks benefited to these advances. In this assignment, we benefited from the deep learning framework Tensorflow[1]. As the subject of the assignment, an age estimation model has been developed to efficiently regress age values using the downsampled version of **UTKFace** dataset, which was supplied for this assignment. In this report, we supplied the details of our model development and improvement stage together with the works that we inspired from. As our first stage of our assignment, we provided our data preparation stage which involves the details that we have explored in the dataset and the data compression technique that we have employed. Following this section, we supplied the details about our model development pipeline. Concluding the assignment, we provided the final results that we obtained and our comments regarding our findings.

## II. Data Preparation

As our initial task in the assignment, we tried to discover the details of the dataset supplied. This was performed as our first stage of the assignment in order to obtain a brief idea about the dataset and take necessary actions accordingly. Our data preparation stage consisted two stages, the initial stage was changing the dataset structure for easy access (data compression) and the second stage is the dataset construction and analysis. The details of these stages are supplied in this section. The implementation of this step is given in IPython Notebook *Understanding the data.ipynb*.

### A. Data Compression

Considering the performance improvement that GPUs enabled in deep learning model development, we considered using online platforms like Google Colab for online model training. Considering that file I/O is a problem for these applications, we attempted to minimize this overhead. In order to achieve this we employed HDF5 data compression format[2] to store our dataset in a single binary file. By this way, with a single I/O operation we were able to import the whole dataset. In order to see whether this makes any difference in local environment, we performed a timed test. As the result of this assessment we saw that access to the dataset with brute-force I/O operations took **13.473 sec** where reading from the binary dataset file took **1.66 sec**. These times differ depending on the hardware constraints of the environment but the ratio between these runtimes were preserved in our experiments. We also verified that this compression is lossless. The dataset file constructed is supplied together with our submission.

### B. Analyzing the Dataset

After completing the data retrieval stage, the initial stage that we have performed was performing mix-max scaling to the image pixels. The formulation of this scaling on image pixels are given as equation 1. Here the value $x_{max}$ is determined as 255 and $x_{min}$ is determined as 0, considering the nature of a grayscale image.

$$\tilde{x} = \frac{x - x_{min}}{x_{max}} \tag{1}$$

Following this stage, we monitored dataset details by using data statistics of the dataset. These statistics are presented as table II-B. According to our observations training, validation and test datasets have really similar characteristics in terms of label distribution. This observation convinced us that there is no need for considering class imbalance between the three partitions of the dataset. However, we saw that the dataset is heavily biased towards the age range [20, 30], which was the initial clue that lead us to the conclusion that perfect prediction performance may not be achievable. Another though on the dataset is the exact distribution on each sample group cause not only test the model inaccurately but also can reward the overfitted model. We will return to these claims while discussing the result. The histograms showing label distribution in the datasets are given as figure 1.

## III. Building the Age Estimation Model

To build the optimal age estimation model, we followed a structured pipeline that involves certain step to evaluate an architecture. In this pipeline, we preferred to start with a simple model without any regularization applied. Then we

TABLE I
STATISTICS FOR UTKFACE DATASET

| Dataset | # of samples | Mean | Std | <25% | <50% | <75% |
|---|---|---|---|---|---|---|
| **Training** | 5400 | 32.03 | 18.23 | 23 | 29 | 42 |
| **Validation** | 2315 | 32.02 | 18.23 | 23 | 29 | 42 |
| **Testing** | 1159 | 32.00 | 18.22 | 23 | 29 | 42 |

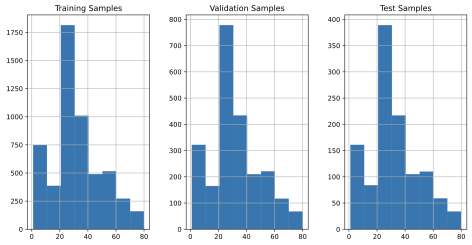


Fig. 1. Label distributions for training, validation and testing datasets

regularized our architecture step by step. The details of this optimization is explained in this section. The implementation of this step is supplied in IPython Notebook *Optimizing the model.ipynb*. We performed our training here with GPU utilization. We used **NVIDIA GTX 1050-TI** and **NVIDIA GTX 970M** GPUs.

### A. Implementation Details

For implementing our model using Tensorflow, we used **keras** library. Our aim for the model implementation stage was creating a small framework that would not require us to implement a new model for every experiment for optimization. In order to achieve that, we implemented *AgeModel* class which takes metadata for layers as a list input. In the IPython Notebook files submitted, we benefited from this implementation by being able to rapidly change the model architecture. The implementation of this class is given in *model.py*. Here we imagined all of the hyperparameters in the model that can be possibly tuned as hyperparameters that can be feed to the implementation. For batch normalization we provided two different options which one adds batch normalization before the activation function and the other implements batch normalization as an independent layer (after applying non-linearity). We used ReLU as the non-linear activation for all of the layers. Considering the implementation that we have provided, the supported layers are *2D Convolution*, *MLP*, *Pooling (max, avg)* and *Batch Normalization*.

### B. Deciding on a non-regularized architecture

The process of building a neural network is a comprehensive task. It includes critical decisions such as depth of network, type of layers, order of layers and many more which all have may an impact on model overall success. In order to have a good starting point with our initial model architecture, we benefited from the existing models that are currently being used as feature extraction backbones. Considering the ease of implementation and its success on ImageNet dataset, we decided to follow the architecture provided by VGG-16 network [3]. Using the architecture provided in this work, we tried to tune the number of filters and the depth of the network considering our task. Since this process can be considered as tuning the complexity of the model architecture, we wanted to tune the model such that it has only the required amount of model complexity. This section of our report aims to explain the changes that we have done in order to optimize our initial non-regularized architecture. The main motivation behind why we wanted to initiate the model building procedure was to clearly measure whether the model is complex enough.

*1) Decreasing the filters:* Comparing the architecture proposed as VGG-16, our first observation was about the compelxity of this network and the task that it was proposed on. As our dataset is based on grayscale human faces, the feature extraction stage seemed less complex compared to ImageNet challenge. In this regard, we initially halved the amount of filters present in the initial architecture. Following this, we tried to adjust the network based on the input shape of the images given, which was (91, 91, 1). After these modifications we had an architecture that applies convolution and pooling until the output size becomes 2x2. At this stage, we made an addition that increases the model complexity by using 1x1 convolutions. Here we tried to double the amount of filters available by considering different linear combinations of them. Our motivation here was to perform a final feature extraction without losing spatial information.

*2) Tuning the number of neurons in dense layers:* This stage involved non-spatial feature extraction (pixel value based). The original VGG-16 architecture had three dense layers with identical dimensions, which caused the parameter number to explode. In order to prevent this, we reduced the dimensions of the dense layers into 256, 128 and 64 respectively. As each successive layer tries to identify a higher-layer feature, we found it sensible to reduce the number of neurons at each successive layer.

### C. Validating that the model architecture is complex enough

In order to validate that our non-regularized model is complex enough to capture the complexity of our dataset, we run a validation stage. To do that, we selected a small subset of the training set and tried to overfit our model to that dataset. Here we used 20 training samples that are randomly selected from the training set. In our final trial run, we reached **0.067** MAE value on the trained samples, which is an indicator of overfitting. The loss change plot is provided as figure 2.

*1) Weight Initialization Problem:* As a reminder, we faced weight initialization issues at this stage which made us to perform several runs for a certain configuration. Since we performed zero regularization at this stage, we followed the approach of performing several runs even with *Xavier initialization*. We worked on how to improve our model even if initialization is poor. First, in order to avoid getting different results in each run, we fixed the random seed to a number that model fails to learn. Following that we did experiments by adding new layers, tweaking the learning rate, increasing the number of filters and using larger subsamples to train the

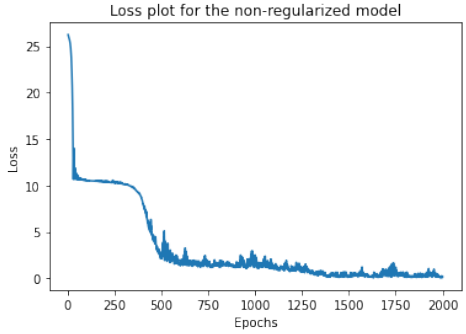


Fig. 2. Loss plot for optimizing model complexity

model. Unfortunately, non of them improve the performance a single bit without having regularization.

### D. Finalizing the model architecture

After completing the first task successfully, we decided to finalize our network structure with a small amount of regularization. In order to perform this task, we considered applying Batch Normalization and Dropout regularization with different configurations.

*1) Applying Batch Normalization:* Following the discussion from the lectures [4], we decided to try our different configurations on where to apply batch normalization. In this regard, we considered three different alternatives. These alternatives are listed as follows:

- Applying Batch Normalization at each convolutional layer, before applying the non-linear activation
- Applying Batch Normalization at the end of each convolutional block, before the pooling layer
- Applying Batch Normalization at each dense layer (MLP), before applying the non-linear activation

Separately, we tried out all of these alternatives to see whether any of these policies benefit to our network. Unlike the previous stage of our model development pipeline, we applied a small amount of L2 regularization (with ratio of $10^{-5}$) in order to make an approximation to overall model performance at the best case. Here the we trained the models with each option using 50 epochs to perform rapid testing and see the behaviour of the model at the same time. At the end of this stage, we found out that applying batch normalization before non-linear activation at each convolutional layer benefited our model the most. The third option that applies batch normalization before every pooling layer also had similar validation set performance, which we also consider as a successful configuration. However, since we did not want to mix up the non-linearity with batch normalization we followed the first option in our architecture. The loss change plots for option 1 and option 3 is supplied as figure 3.

*2) Applying Dropout Regularization:* As another option that may improve the predictive capability of our model, we decided to try out dropout normalization. Considering the existing literature, we applied this regularization after every

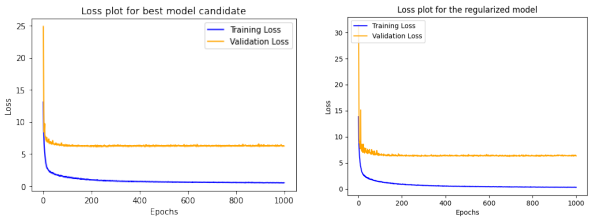


Fig. 3. Model trained with batch normalization option 1 (left) and 3 (right)

dense layer (except the last layer) and the last convolutional layer. However, we observed that dropout regularization oversimplified our model and impacted the learning capabilities of our model in a negative way. We observed this from both the change in training loss and validation loss. The corresponding plot is given as **??**. After considering all of these options, we followed with applying batch normalization at each convolutional layer with the explained VGG-16 variant tuned by us. The visual representation of our architecture is given in figure 4. The blocks in the figure are 2D Convolution, Batch Normalization, ReLU activation, pooling and dense.

## IV. Tuning the Model

In order to tune the selected architecture, we applied certain regularization techniques. We have discussed each of them in this section in detail.

### A. Tuning the regularization parameter and learning rate

Before discussing any further details like batch size and amount of epochs to train our model, here we discussed our procedure for determining the regularization factor and the learning rate to train our model. Based on our experience and works about deep learning architectures, applying regularization and selecting an appropriate learning rate can be crucial for predictive performance. In order to find an acceptable hyper-parameter configuration, we preferred to make experiments in log-space of these parameters. In our experiments, we sampled 50 different learning rate and regularization parameter configurations in a two staged manner, where the second stage is performed by reducing the range used in the first stage. For each of these sampled pairs, we performed a small training on a subset of the training images (500 images) with 5 epochs and then evaluated the trained model on the validation set. After inspecting these results based on the validation set loss (Mean Absolute Error), we got an idea about the optimal range for the parameters for our model. While sampling different parameters we preferred to perform sampling from the log-space from a uniform distribution. For the ranges of this search, we started with the range $[10^{-6}, 10^{1}]$ for the L2 regularization constant and $[10^{-6}, 10^{-3}]$ for learning rate. After the initial stage of this search, by inspecting the 10 best trials, we squeezed this range into $[10^{-5}, 10^{-3}]$ for both the L2 regularization constant and learning rate. The 5 best configurations found after the second stage is given in table IV-A. Considering these results, we set both our learning rate and regularization constant as $10^{-4}$

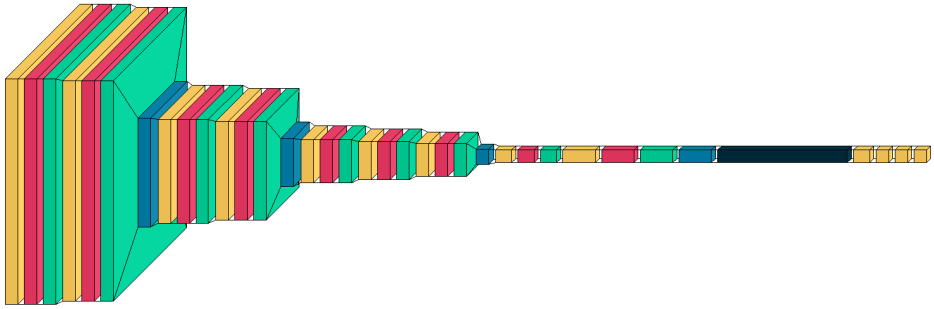

Fig. 4. Proposed Model Architecture

TABLE II
RESULTS OF HYPERPARAMETER SEARCH FOR LEARNING RATE AND L2 RATIO

| Validation Loss | Training Loss | log(learning rate) | log(L2 ratio) |
|---|---|---|---|
| 27.296 | 8.948 | -3.982 | -4.701 |
| 27.883 | 12.427 | -4.548 | -4.556 |
| 28.000 | 7.971 | -3.494 | -4.599 |
| 28.626 | 13.358 | -4.543 | -4.213 |
| 29.190 | 11.189 | -4.252 | -3.923 |

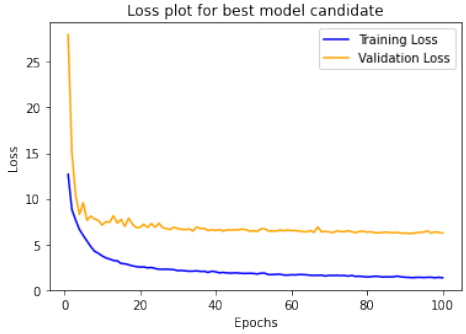


Fig. 5. Training model after tuning $lr$ and L2 ratio

after a search from random space. Then, in order to observe the behaviour of the model with these hyper-parameters, we performed a training on the whole dataset for 100 epochs, which is supplied as figure 5.

### B. Early Stopping as a Regularizer

As our final regularizer, we applied early stopping to our model. Implementation-wise, there is not physical limit for stopping the learning procedure. However, training the model for too long can cause overfitting issues and result in wasting computational resources, considering these arguments we wanted to find a stopping point for training the model. As our first step, we trained the model for 1000 epochs and inspected the obtained training and validation loss values to see whether they change all the time. The loss plot for training the model for 1000 epochs is given as figure 6. Following this step, we identified that the training loss does not change further after 600 epochs where for validation loss this number is much lower. To see whether training the model less effects the model

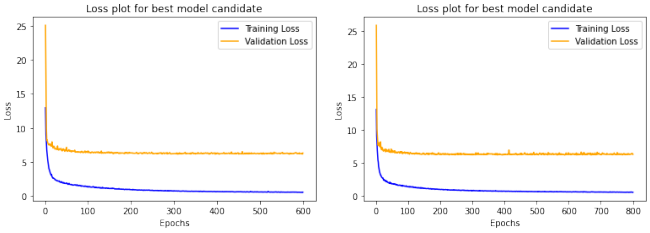


Fig. 6. Training the model with early stopping (left) and overtraining (right)

performance we trained the model for 600 epochs this time to see the effect of this regularization. The loss plot can again be found at 6. In this experiment, our main focus was on whether more training leads to better learning in the training set, by inspecting the training loss. However, since the changes in the validation loss and training loss is too low after some point, training loss may not be that much informative as the stopping criteria. To prevent this issue, we employed another training with 800 epochs but by saving the model weights at every time validation loss improves. With this training we aimed to simulate early stopping the learning process by updating the model only when the validation loss improves considering the overall learning process. In order to observe the effect of early stopping, we performed a comparison based on the validation set error and testing set error. Here the error is defined as Mean Absolute Error, which can be specified as equation 2. This error function is also the loss function specified for our model.

$$L_{MAE} = \frac{\sum_{i=1}^{N} |y_{real} - y_{pred}|}{N} \tag{2}$$

The results of the different models (trained for different number of iterations) are provided in table IV-B. As it can be seen from the obtained results, validation loss seems like a better indicator than training set loss. Originating from the results obtained, it can be stated that early stopping can be an effective strategy. In our sample runs, we obtained the lowest testing set loss 6.22 however, since it can depend on the initialization and to be consistent we provided our final result.

TABLE III
RESULTS OBTAINED WITH FOR MODELS TRAINED WITH DIFFERENT NUMBER OF ITERATIONS

| Training amount | Training Loss | Validation Loss | Testing Loss |
|---|---|---|---|
| 800 epochs | **0.5115** | 6.2636 | 6.4873 |
| 587 epochs | 0.5586 | 6.3150 | 6.4862 |
| 600 epochs | 0.5867 | **6.1663** | **6.4457** |

### C. Other optimization issues

In our hyperparameter tuning procedure, we tuned some of the parameters without performing systematic tests. For this work, we tuned the batch size and the initialization preference by trying out different alternatives. The optimization process and findings are presented in this section.

*1) Type of initialization:* While implementing the class *AgeModel*, we enabled two types of weight initializations which are *Xavier initialization* (from an uniform distribution) and *Gaussian initialization* (from a normal distribution). The end model proposed uses Xavier initialization. Our main concern regarding the weight initialization was encountering failing cases in weight initialization as less as possible. After our trial runs, we decided to follow Xavier initialization. Even though this method is more robust compared to random initialization, we still encountered some failing cases, as specified in the previous parts.

*2) Batch size:* As our last hyperparameter, we tried out different batch sizes to see their effect on performance. The tried out batch sizes were 32 and 64, considering the hardware constraints. In our experiments, we observed that the model can learn better with a batch size of 32 (can achieve lower training loss values), which is the main reason why we selected 32 as the batch size of our model.

## V. RESULTS

Considering the results obtained as the results of our experiments, we saw that the best results were obtained when early stopping has been applied with 600 epochs. In order to evaluate the overall performance of the model on this task, we decided to visualize the best predictions together with the worst ones.

### A. Best predictions

The top 4 predictions considering the Mean Absolute Error metric given in equation 2 are given as figure 7. In this visualization we saw that the baby images are easily classified, even when we look at the whole dataset. We evaluated this result as natural since babies have low feature complexity on their faces in general. For the other examples that are classified almost perfectly, they seemed to be in the age range [20,30]. Considering the statistical findings found in table II-B, the dataset is biased towards this age range and it can be considered as a natural outcome to be that images around this range are classified correctly. These images also seem clear and the facial attributes are not that noisy, which makes these results interpretable and not random.

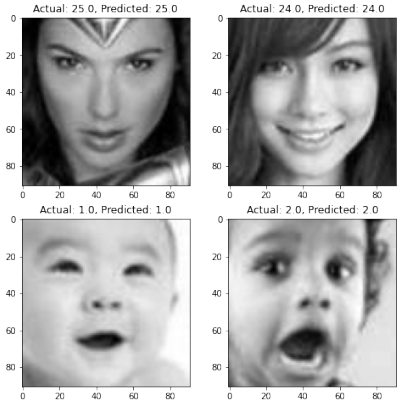


Fig. 7. Top 4 best predictions on the test set

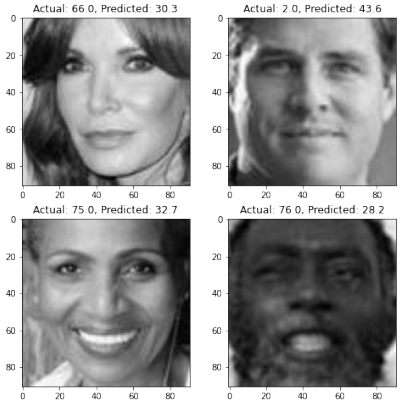


Fig. 8. Worst 4 best predictions on the test set

### B. Worst predictions

For the worst predictions made by the model, we observed unnatural behavior from the dataset as it can be observed in figure 8. In there testing examples, we saw problematic issues in almost all of them. As the first issue, we observed wrong labels as it can be seen from the second failing example. Despite data labeling related issues, details like makeup and darker skin color that makes the facial details indistinguishable is expected to harm the performance of the overall performance of the model. Even though there is no detailed check performed, we assume that these outliers were not present in the dataset which makes it very hard to predict these figures correctly. For the testing instance that is labeled in a false way, the prediction that our model made was more accurate than the labeling, which shows the generalization capabilities of the trained model.

## VI. CONCLUSION

For concluding remarks, we believe that our model gives promising but not perfect results for the task of age estimation. In order to achieve better performance, with residual architectures a better facial attribute extraction can be applied. In addition to that, by performing data augmentation to reduce the class imbalance, which is apparent in table II-B, better final

test set loss as training set will gain more information from the outlying examples (considering the label distribution).

## REFERENCES


[1] Martı'n Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[2] The HDF Group. (2000-2010). "Hierarchical data format version 5," [Online]. Available: http://www.hdfgroup.org/HDF5.

[3] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, 2015. arXiv: 1409.1556 [cs.CV].

[4] H. Dibeklioğlu, "Feedforward neural networks," University Lecture, 2021.


# VII. APPENDIX

## SUMMARY OF THE MODEL PROVIDED BY TENSORFLOW

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d_774 (Conv2D) | (None, 89, 89, 32) | 320 |
| batch_normalization_774 (Bat | (None, 89, 89, 32) | 128 |
| activation_774 (Activation) | (None, 89, 89, 32) | 0 |
| conv2d_775 (Conv2D) | (None, 87, 87, 32) | 9248 |
| batch_normalization_775 (Bat | (None, 87, 87, 32) | 128 |
| activation_775 (Activation) | (None, 87, 87, 32) | 0 |
| max_pooling2d_344 (MaxPoolin | (None, 43, 43, 32) | 0 |
| conv2d_776 (Conv2D) | (None, 41, 41, 64) | 18496 |
| batch_normalization_776 (Bat | (None, 41, 41, 64) | 256 |
| activation_776 (Activation) | (None, 41, 41, 64) | 0 |
| conv2d_777 (Conv2D) | (None, 39, 39, 64) | 36928 |
| batch_normalization_777 (Bat | (None, 39, 39, 64) | 256 |
| activation_777 (Activation) | (None, 39, 39, 64) | 0 |
| max_pooling2d_345 (MaxPoolin | (None, 19, 19, 64) | 0 |
| conv2d_778 (Conv2D) | (None, 17, 17, 128) | 73856 |
| batch_normalization_778 (Bat | (None, 17, 17, 128) | 512 |
| activation_778 (Activation) | (None, 17, 17, 128) | 0 |
| conv2d_779 (Conv2D) | (None, 15, 15, 128) | 147584 |
| batch_normalization_779 (Bat | (None, 15, 15, 128) | 512 |
| activation_779 (Activation) | (None, 15, 15, 128) | 0 |
| conv2d_780 (Conv2D) | (None, 13, 13, 128) | 147584 |
| batch_normalization_780 (Bat | (None, 13, 13, 128) | 512 |
| activation_780 (Activation) | (None, 13, 13, 128) | 0 |
| max_pooling2d_346 (MaxPoolin | (None, 6, 6, 128) | 0 |
| conv2d_781 (Conv2D) | (None, 4, 4, 256) | 295168 |
| batch_normalization_781 (Bat | (None, 4, 4, 256) | 1024 |
| activation_781 (Activation) | (None, 4, 4, 256) | 0 |
| conv2d_782 (Conv2D) | (None, 4, 4, 512) | 131584 |
| batch_normalization_782 (Bat | (None, 4, 4, 512) | 2048 |
| activation_782 (Activation) | (None, 4, 4, 512) | 0 |
| max_pooling2d_347 (MaxPoolin | (None, 2, 2, 512) | 0 |
| flatten_86 (Flatten) | (None, 2048) | 0 |
| dense_344 (Dense) | (None, 256) | 524544 |
| dense_345 (Dense) | (None, 128) | 32896 |
| dense_346 (Dense) | (None, 64) | 8256 |
| dense_347 (Dense) | (None, 1) | 65 |

Total params: 1,431,905
Trainable params: 1,429,217
Non-trainable params: 2,688