Age Estimation Using TensorFlow

Muhammed Çavuşoğlu
Department of Computer Engineering
Bilkent University
Ankara, Turkey
m.cavusoglu@bilkent.edu.tr

Kemal Büyükkaya
Department of Computer Engineering
Bilkent University
Ankara, Turkey
kemal.buyukkaya@bilkent.edu.tr

Abstract—**In this report, we explain our deep age estimation network that predicts the ages of people from their facial images using TensorFlow. We discuss our final architecture, experiments that we conducted and our results. Our MAE (Mean Absolute Error) on the test set using the model with the smallest validation error is 5.688.**

I. Introduction

In this homework, we designed and trained a deep age estimation network that predicts the ages of people from their facial images using TensorFlow. In the following sections, we discuss the final architecture of our network, experiments that we conducted, and our results.

II. Architecture

Our final implementation is in a Jupyter Notebook (.ipynb), and it is run on Google Colab with GPU as hardware accelerator. We use TensorBoard for debugging and visualizing our computational graph, loss values, and MAE (Mean Absolute Error) values. Our CNN model, main function, and utility functions are explained in detail in the following sections.

A. CNN Model

Our CNN model consists of the following layers, listed in the order of execution:

- **Input Layer:** reshapes the features to a [batch_size, 88, 88, 1] tensor.
- **Convolutional Layer 1:** computes 32 features using a 4×4 filter with ReLU activation. This layer is implemented using *tf.contrib.layers.conv2d*. Width and height values are preserved using padding. Xavier initialization (*tf.contrib.layers.xavier_initializer*) and L2 Regularization (*tf.contrib.layers.l2_regularizer*) with a scale of 0.8 is used.
- **Convolutional Layer 2:** computes 32 features using a 4×4 filter with ReLU activation. This layer is implemented using *tf.contrib.layers.conv2d*. Width and height values are preserved using padding. Xavier initialization (*tf.contrib.layers.xavier_initializer*) and L2 Regularization (*tf.contrib.layers.l2_regularizer*) with a scale of 0.8 is used.
- **Pooling Layer 1:** is a max pooling layer with a 2×2 filter and stride of 2. This layer is implemented using *tf.contrib.layers.max_pool2d*. Input tensor to this layer has the shape [batch_size, 88, 88, 32], and the output tensor has the shape [batch_size, 44, 44, 32].
- **Batch Normalization Layer 1:** is implemented using *tf.contrib.layers.batch_norm*, and it does not have an activation function.
- **Convolutional Layer 3:** computes 64 features using a 4×4 filter with ReLU activation. This layer is implemented using *tf.contrib.layers.conv2d*. Width and height values are preserved using padding. Xavier initialization (*tf.contrib.layers.xavier_initializer*) and L2 Regularization (*tf.contrib.layers.l2_regularizer*) with a scale of 0.8 is used.
- **Pooling Layer 2:** is a max pooling layer with a 2×2 filter and stride of 2. This layer is implemented using *tf.contrib.layers.max_pool2d*. Input tensor to this layer has the shape [batch_size, 44, 44, 64], and the output tensor has the shape [batch_size, 22, 22, 64].
- **Batch Normalization Layer 2:** is implemented using *tf.contrib.layers.batch_norm*, and it does not have an activation function.
- **Reshape Layer:** flattens the tensor of shape [batch_size, 22, 22, 64]] to [batch_size, 22 * 22 * 64].
- **Fully Connected Layer 1:** is implemented using *tf.contrib.layers.fully_connected*, and it has 1024 outputs with no activation function. Xavier initialization (*tf.contrib.layers.xavier_initializer*) is used in this layer.
- **Batch Normalization Layer 3:** with ReLU activation is implemented using *tf.contrib.layers.batch_norm*.
- **Dropout Layer 1:** with 0.8 probability of keeping an element is used. This layer is implemented using *tf.contrib.layers.dropout*, and it is applied when the estimator is in training mode.
- **Fully Connected Layer 2:** is implemented using *tf.contrib.layers.fully_connected*, and it has 1024 outputs with no activation function. Xavier initialization (*tf.contrib.layers.xavier_initializer*) is used in this layer.
- **Batch Normalization Layer 4:** with ReLU activation is implemented using *tf.contrib.layers.batch_norm*.
- **Dropout Layer 2:** with 0.8 probability of keeping an element is used. This layer is implemented using *tf.contrib.layers.dropout*, and it is applied when the estimator is in training mode.
- **Fully Connected Layer 3:** is implemented using *tf.contrib.layers.fully_connected*, and it has 1024 out-

puts with no activation function. Xavier initialization (*tf.contrib.layers.xavier_initializer*) is used in this layer.

- **Batch Normalization Layer 5:** with ReLU activation is implemented using *tf.contrib.layers.batch_norm*.
- **Dropout Layer 3:** with 0.8 probability of keeping an element is used. This layer is implemented using *tf.contrib.layers.dropout*, and it is applied when the estimator is in training mode.
- **Regression Layer:** is implemented using *tf.contrib.layers.fully_connected*. It takes the input tensor of shape [batch_size, 1024] and returns an output tensor of shape [batch_size, 1], and it does not have an activation function. The outputs are then rounded since provided age labels are integers.

We use Absolute Difference (*tf.losses.absolute_difference*) as our loss function. We use Adam Optimizer with learning rate of 0.001, $beta_1$ (exponential decay rate for the 1st moment estimates) value of 0.9, $beta_2$ (exponential decay rate for the 2nd moment estimates) value of 0.999, and epsilon (small constant for numerical stability) value of 1e-08. Adam tries to minimize the loss.

We log MAE values to concole using *LoggingTensorHook*. We log loss values, steps, execution times, and information related to the state of the network to console by setting *tf.logging.set_verbosity(tf.logging.INFO)*. We also log loss and MAE values to TensorBoard for visualization and debugging using *tf.summary.scalar*.

B. Main function

In main function, we load training data, training labels, validation data, validation labels, test data, and test labels in two ways. First way is to use a utility function that we implemented (discussed in detail in Utility functions section), that returns *numpy* arrays from image dataset. Second way is to load the generated *numpy* arrays. First approach is run initially, and the resulting arrays are stored (locally for local runs, in Google Drive for Colab runs). After this, in every run, the second approach is used. This speeded up our runs significantly since the program did not generate the *numpy* arrays from images, and we did not have to upload the images to Google Drive to run our code in Colab.

In order to train and evaluate our models, we use an Estimator (*tf.estimator.Estimator*) which is the counterpart of *tf.contrib.learn.Estimator*. we use RunConfig instance to configure our estimator. It stores the checkpoints every 1000 steps, and only keeps the latest two checkpoints (before setting this, our Google Drive memory was filled with checkpoints). We use RunConfig's default value of 100 for *save_summary_steps*, to save summaries for TensorBoard. Estimator loads the model saved in *final_model_dir*, to improve our latest pre-trained version of our model.

We implemented an early stopping mechanism for training based on validation set MAE. For a batch size of 32, estimator trains for 100 steps and checks for validation set MAE. If it it less than 5.75, the training stops. We initially trained for 1000 steps and then checked validation MAE, but we observed that in intermediate steps we miss smaller values. Therefore, we increased the frequency of early stopping check. Estimator then saves the best model based on validation MAE, and loads it to evaluate on the test set.

Using this estimator, we extract some success and failure cases. Based on the absolute value of the predicted and label, we plot one of the best and worst results.

C. Utility functions

We use *drive.mount* function of Google Colab to access files in our Google Drive (e.g. saved models, data, labels). We use *shutil*'s *make_archive* function to download our saved models as a zip file. We use OpenCV to read our images, and then we save them to *numpy* arrays and write them out to a file using *numpy.save*.

See the Appendix for our TensorFlow computation graph generated by TensorBoard.

Experiments that we conducted to obtain this final architecture, together with our motivations, are discussed in the following section.

III. Experiments

Initially, we conducted our experiments on our personal computers. In order to make sure our network functions, we resized 91×91 images to 28×28. However, when we plotted the resized images, it was difficult for us to spot the features to estimate ages. When we moved our codebase to Google Colab with GPU as hardware accelerator, we resized 91×91 images to 88×88, since we have two pooling layers that halves the image width and height.

As described in *Main function* part of the Architecture section, we implemented an early stopping mechanism based on validation set MAE. Prior to this, we trained our model with varying number of steps and epochs. After implementing early stopping, we downloaded our model and inspected MAE and loss values in TensorBoard which are given in Results section. In order to improve our network, we conducted the experiments below.

A. Number and configuration of layers

The number and configurations of different types of layers are discussed in the following section.

- **Convolution Layers**
  We tuned the number of convolutional layers, and experimented with 1, 2, 3, 4 convolution layers. We initially wanted to see that the network overfits the training data. 1 convolutional layer was too shallow, and as we increased the number of convolutional layers, MAE on training data decreased. However, we observed that as we increase the number of convolutional layers, MAE on validation set increased. This may be due to the fact that we are losing generalization power and our network is starting to learn noise as we increase the complexity of the network. Empirically, 3 convolutional

layers gave optimum validation MAE values.

We tuned the kernel sizes of our convolutional layers, and experimented with different combinations of 2×2, 4×4, 5×5, 7×7, 8×8, and 9×9 kernels. When determining the kernel size, our intuition was based on locality of low level features, and generalization of these features to the entire image. We wanted to consider low level features similar to creases in the face, and at the same time consider these features at other locations of the entire image. MAE scores on validation set when 4×4 kernel is used was optimal (with 6.446). Other experiment results when we only change the kernel size are as follows: $2 \times 2 \rightarrow 6.756$, $8 \times 8 \rightarrow 6.855$, $9 \times 9 \rightarrow 6.930$.

We tuned the number of outputs of our convolutional layers, and experimented with output values of 32, 64, 128, and 256. We could only test 32 and 64 on our computer's due to the our limited CPU power. We tried other values when we moved our codebase to Google Colab. As we increase the number of outputs, complexity of our model increases, and it extracts more features since each filter generates a feature map. As we increased the complexity of our model, we observed that MAE on validation set increased. Since input images are already pre-processed, the increased complexity of our network may cause it to learn undesirable features. Empirically, 3 convolutional layers with output numbers of 32, 32, and 64, respectively, gave optimum validation MAE values.

- **Pooling Layers**
  We conducted experiments with max pooling and average pooling. We used max pooling to extract features like edges, and it worked better in practice compared to average pooling which extract features more smoothly. In these pooling layers, we downsample our images using a filter with size 2, and stride 2.

- **Fully Connected Layers**
  We conducted experiments with 1, 2, and 3 fully connected layers with 128, 512, 1024, and 2048 outputs. Our motivation for these experiments to find optimal number of fully connected layers to learn non-linear combinations of the features extracted by convolutional layers. Empirically, 3 fully connected layers, each with 1024 outputs gave optimal results.

B. Loss function

We used two different loss functions: Mean Squared Error (*tf.losses.mean_squared_error*), and Absolute Difference (*tf.losses.absolute_difference*). We initially picked Mean Squared Error to put more penalty on the differences. However, when we inspected the dataset, we observed outlier examples similar to the following figures.

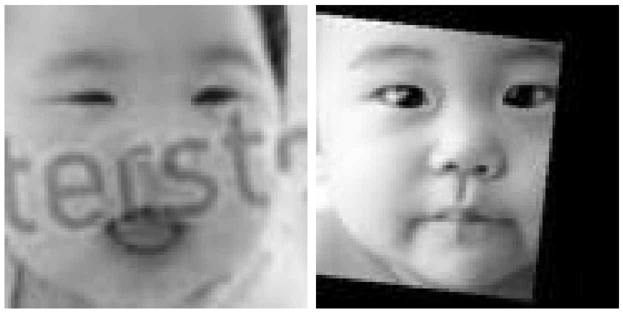

Fig. 1. Outlier examples in dataset

In order to make the network not focus on the outliers and learn redundant features, we switched our loss function to Absolute Difference. Absolute Difference is a suitable loss function for this problem since we try to minimize MAE.

C. Initialization

We experimented with two different weight initializers: Xavier (*tf.contrib.layers.xavier_initializer*), and Gaussian (*tf.initializers.truncated_normal*). When we used Gaussian initialization, we did not observe a smooth decrease in the loss values during training every time we conduct our experiments. However, when we used Xavier initialization, we observed a smooth decrease in the loss values during training which can be seen in Figure 2.

D. Batch Normalization

In order to address the *internal covariate shift* and normalize layer inputs, we used Batch Normalization Layers [1]. We added the Batch Normalization Layers incrementally to the ends of pooling layers and fully connected layers, and observed a decrease on MAE values.

E. Regularization

We used L_2 regularization in convolutional layers as a weight regularizer. We used 0.4, 0.6, 0.8 as L_2 regularizer scale values. We selected 0.8 as our scale value, since we already apply various regularization techniques (e.g. early stopping, dropout) and small values of the scale value prevents overfitting training data more.

We conducted experiments with 1, 2, and 3 dropout layers, and keep probabilities of 0.6 and 0.8. We selected 3 dropout layers to prevent overfitting and to simulate the combination of different architectures [2]. Since we have several dropout layers, we set the keep probability as 0.8. Additionally, this setup decreased our validation MAE.

F. Adam Optimizer

We used Adam optimizer and tried the following parameter values: $\alpha = 0.01, 0.001, 0.0001$; $\epsilon = 10^{-1}, 10^{-5}, 10^{-8}$. Parameters that yielded the best results are the default parameters described in the paper ($\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$) [3].

IV. Results

We evaluated our model with the smallest validation set error, on **test set**, and obtained MAE of **5.688**. Evaluation results on training data, validation data, and test data are given in the table below.

TABLE I
EVALUATION RESULTS OF THE MODEL WITH THE SMALLEST VALIDATION SET ERROR

Evaluation Dataset	**MAE**
Training Data	0.611
Validation Data	5.739
Test Data	**5.688**

We used TensorBoard to debug and visualize our network. The following figures are generated from TensorBoard.

The following two plots represent the change in loss values during training.

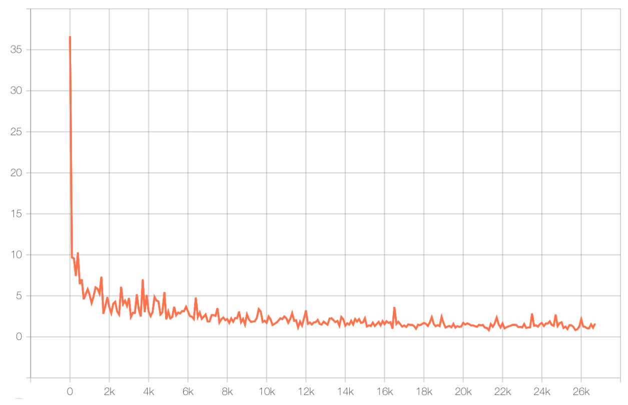

Fig. 2. Loss values during training

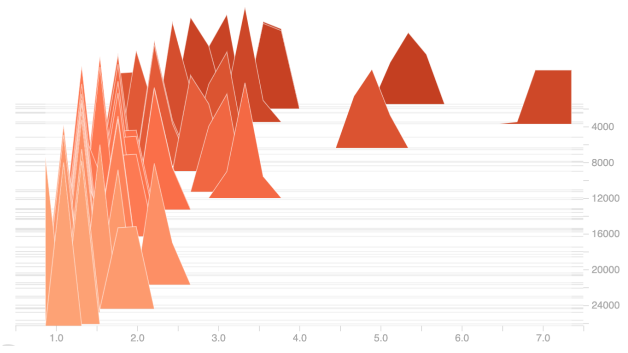

Fig. 3. Loss values during training histogram

From these two plots, we see the learning behaviour and a clear decrease in loss function during training.

Following two figures represent the loss values and MAE scores during evaluation on the validation data, respectively.

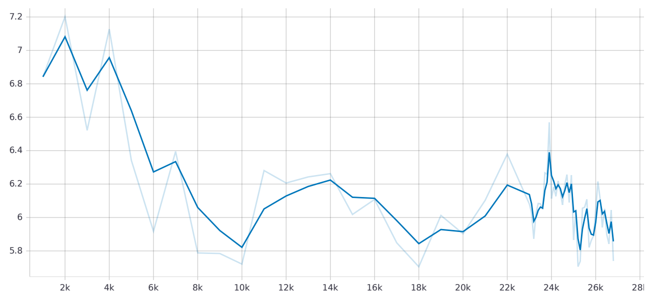

Fig. 4. Loss value during evaluation

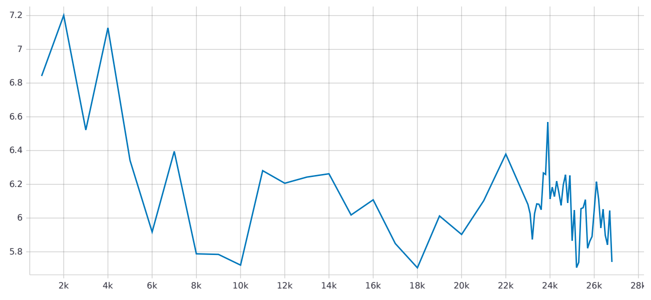

Fig. 5. MAE scores during evaluation

From these plots, we can see the benefits of early stopping; as MAE value reaches close to minimum values several times during training, and if the training is not stopped around these times, it is possible to get higher MAE values.

Following examples represent one success case and one failure case of our network.

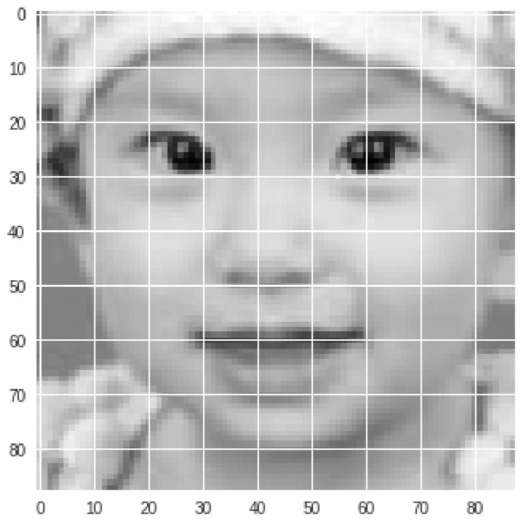

Fig. 6. Predicted Age: 1, Real Age: 1

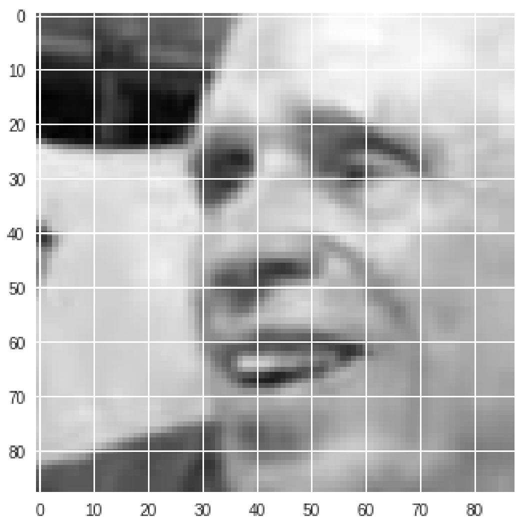

Fig. 7. Predicted Age: 32, Real Age: 76

In the failure case, our network predicts the age as 32, and the real age is 76. This may be due to the pose of the image where the face is not centered, the absence of hair feature in the image, and creases that can occur when smiling.

V. Conclusion

We described our deep age network that predicts the ages of people from their facial images. We discussed the components of the architecture, experiments that we conducted, and the results that we obtained. We obtained a MAE of 5.688 on the test set using the model with the smallest validation error.

References

[1] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015.

[2] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[3] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2015.

Appendix

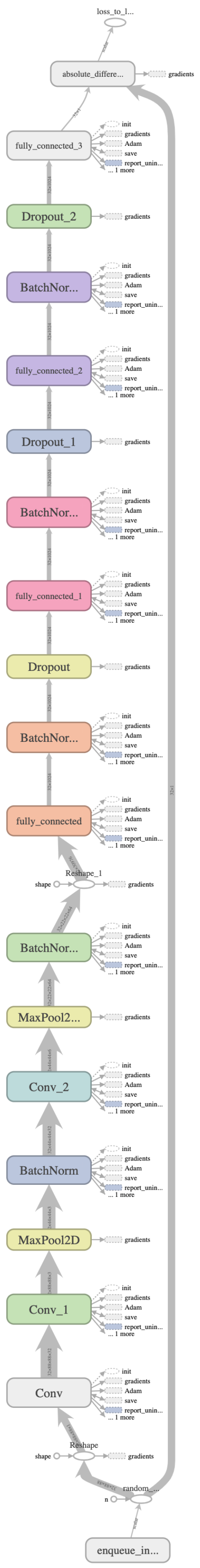

Fig. 8. visualization of our TensorFlow computation graph, generated by TensorBoard