CS 559 Homework

Atakan Serbes, 21200694
Department of Computer Science
Bilkent University
Ankara, Turkey
atakan.serbes@bilkent.edu.tr

INTRODUCTION

The main goal of this homework is to get familiar with TensorFlow which is a powerful machine learning framework. In the homework, I designed, trained and tested a deep age estimator network which estimates the ages from the 91x91 greyscale images. Unlike many other applications, this is a regression task, not a classification task where one of the classes is selected as a result of the network. In order to test and improve the performance of the network, several designs were made and various techniques were investigated such as different number of convolutional layers, batch normalization, different loss functions etc.

In the following sections, the details of the work are discussed.

NOTE: Running the network is done on ***Google Colab*** [1] which gives free use of their cloud GPUs. However this also have limitations which will be discussed in the report.

DATASET

The dataset is comprised of training, validation and test sets which are in size of 91x91 with number of samples 5400, 2316 and 1160 respectively.

(a) Age 2 example (b) Age 65 example

Fig. 1: Some examples from the dataset

DATA PREPROCESSING

Before beginning the architecture of the age estimator network, we should first preprocess the dataset to be able to feed it into the network. Firstly, to be able to feed the network with image-label pairs, we need to extract the labels that are inside the names of all photo files. For example a one (1) year old baby photo is filed as ”001_XXXXXXXX.jpg”. We need to get the second and third digits so that the labels cover the age range of 1 to 80.

A python script was written with the help of packages ***numpy, os, PIL***.

A python script was written with the help of packages ***numpy, os, PIL***.

Labels

For label retrieval, the script first reads all files in the given folder, then finds the index of character ”_” and then using this index information, retrieves the second and third characters of the file name. These labels are then appended into a list which is then saved as a numpy array to be processed.

Images

For images, script stores the images in an n dimensional numpy array. The script opens every image in the folder, converts it into an array (using *asarray* from numpy, reshapes the numpy array into (91, 91, 1] and then appends it to the list of training/validation/test images.

AGE ESTIMATION NETWORK

I started the network with a baseline of MNIST example on TensorFlow website [2], in which there is a convolutional neural network architecture using Estimators given by TensorFlow. The MNIST example includes a convolutional layer + pooling , another convolutional layer + pooling, a dense layer, a logits (includes a neuron for each target class) layer and a prediction layer.

First of all, our problem is not a classification rather a regression. So the ending layer should give us one number which outputs the prediction of age. Network with the following design and specifications was used first,

- 2 convolutional layers, followed by their max pooling layers,
- 1 flatten layer to reshape the tensor for fully connected layers coming next,
- 1 fully connected layer with 1024 neurons,
- 1 fully connected layer with 1 regression output,
- ReLU activations
- 50 epochs, learning rate of 0.01,
- 2 loss functions 1 mean squared error, the other absolute difference (which are the logical ones for a regression task),
- Training image batches of 108, whole image set is randomized at each epoch,
- Validation image batches of 463,

In the following sections, the network is tuned and it will be discussed throughout the report.

Tuning the number of layers and neurons

Firstly the first mentioned network (let us call it vanilla network), is run and did not give any outputs because of some input issues. The issue was the size of training images, 5400 of them would not let me to feed them into the network as a whole, so I used mini-batches to be able to feed the network. The training images were fed with the sizes of 108. Turns out it was a resource exhaustion issue because the sources that were provided by Google Colab [1] were not enough.

My network gave some results with my vanilla network however the loss did not decrease, and I kept getting same loss function outcome from learning, and the same loss function from the optimizer (In this case GradientDescent)

```
Loss after epoch 50: train loss: 32.030182 , loss function: 1156.203735
validation loss: 32.0203
```

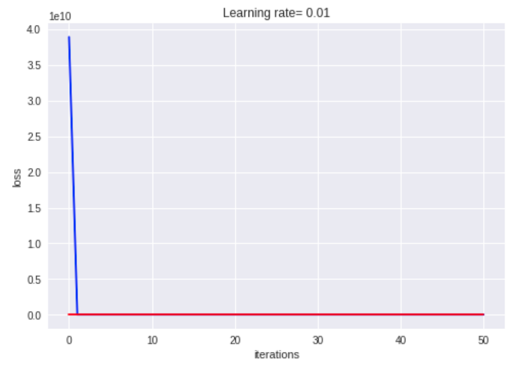


Fig. 2: Running the vanilla network

The reason for this problem is the vanishing gradients, which kept giving the predictons labeled as ”0”. This is solved after using Batch Normalization between the fully connected layers in the network. I will discuss in the later section. After using BatchNorm and obtaining my results that go below 9 absolute error loss, I decided to work with batchnorms and got the result given below in Fig. 3.

```
Loss after epoch 50: train loss: 3.027222 , loss function: 3.257429
validation loss: 8.440265
```

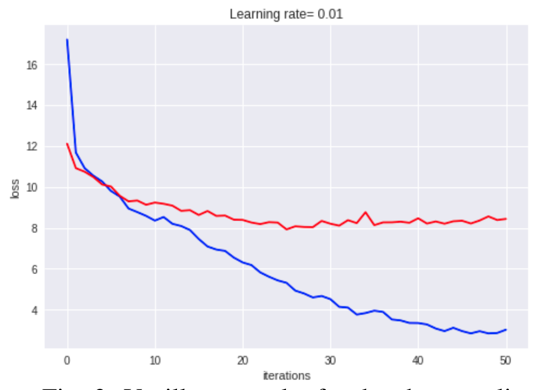


Fig. 3: Vanilla network after batch normalization

In the end my architecture became like the following (I discuss the components in the following sections), the differences and the best cases for the parameters will be discussed in the next sections,

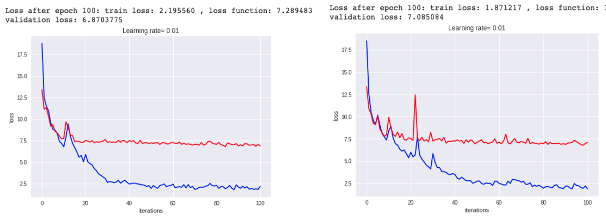
- 6 convolutional layers (all with 3x3 kernel sizes), 2 with filter size 32, 2 with filter size 64 and 2 with filter size 128,
- The first 2 convolutional layers and the second 2 convolutional layers have max pooling, last 2 convolutional layer has none.
- 1 flatten layer
- 4 fully connected layers with 256, 128, 128 and 100 hidden units.
- 1 fully connected layer with 1 regression output,
- ReLU activations, batch normalizations after all components (convolutional layers are considered as a group of 2)
- 100 epochs, learning rate of 0.001,
- 2 loss functions 1 mean squared error, the other absolute difference, mean squared error is used for learning and the absolute difference is used for
- Dropout with probability of dropout 0.2 is applied after every fully connected layer,
- Training image batches of 108, whole image set is randomized at each epoch,
- Validation image batches of 463,

Tuning the loss function

The loss function for a regression problem can be addressed in two different ways, the first one to use is ***tf.losses***’s absolute_difference (it’s mean is MAE) which is used for mean absolute error and the second one is the mean squared error (MSE). The loss function is used to calculate the train, validation and test losses and when both are tested, we can see that MSE gives a more smoothed result.

```
Loss after epoch 100: train loss: 2.195560 , loss function: 7.289483
validation loss: 6.8703775
```

```
Loss after epoch 100: train loss: 1.871217 , loss function: 1.297163
validation loss: 7.085084
```

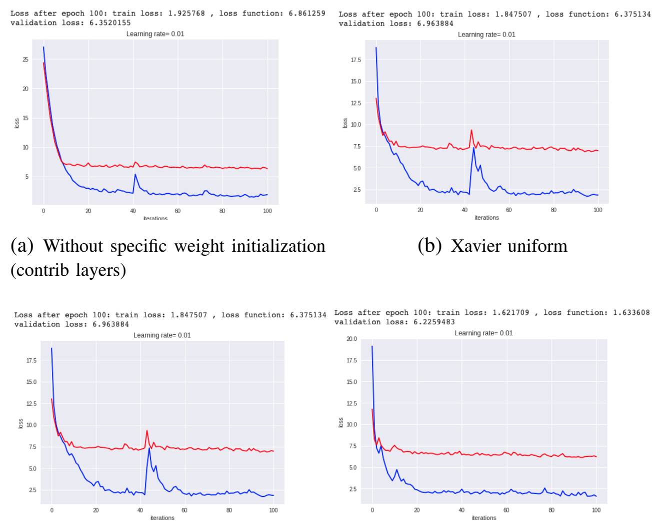


(a) Learning with MSE (b) Learning with Absolute Difference

Fig. 4: Learning with different loss functions

Xavier vs Gaussian Initialization

The layers function from ***tf.contrib*** uses the Xavier initialization as default, in the uniform setting. However I tried setting the weight initialization to xavier uniform, xavier nonuniform and gaussian to see the effects of different initializations. The best result was achieved with the gaussian weight initialization,



```
Loss after epoch 100: train loss: 1.925768 , loss function: 6.861259
validation loss: 6.3520155
```

```
Loss after epoch 100: train loss: 1.847507 , loss function: 6.375134
validation loss: 6.963884
```

```
Loss after epoch 100: train loss: 1.847507 , loss function: 6.375134
validation loss: 6.963884
```

```
Loss after epoch 100: train loss: 1.621709 , loss function: 1.633608
validation loss: 6.2259483
```

(a) Without specific weight initialization (contrib layers)

(b) Xavier uniform

(c) Xavier nonuniform

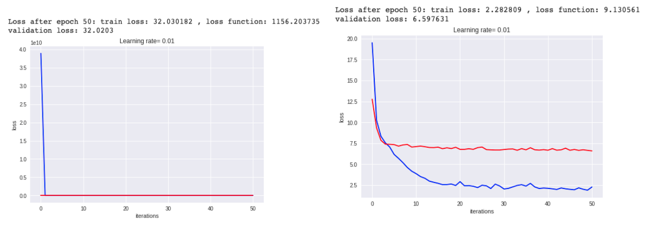
(d) Gaussian initialization

Fig. 5: Xavier and Gaussian Initialization

Xavier initialization [3] can be described as normalized gaussian initialization for each layer. I obtained better results using Gaussian initialization, it can be concluded from figure 5 that Xavier initialization performs slightly worse than gaussian one. Xavier may not be compatible with my network, (for this specific problem) or maybe my network is not deep enough to show better results compared to gaussian initialization.

Batch Normalization layers

Batch Normalization prevents the activations from saturating by bounding their inputs, and my gradients vanish in between the fully connected layers of the network. In my network, I use batch normalization in between my fully connected layers because that is where my gradients vanish and I get the same results in every iteration, ”0” labeled prediction from the last fully connected regression layer.



```
Loss after epoch 50: train loss: 32.030182 , loss function: 1156.203735
validation loss: 32.0203
```

```
Loss after epoch 50: train loss: 2.282809 , loss function: 9.130561
validation loss: 6.597631
```

(a) No batchnorm applied, vanished gradients, no learning

(b) Batchnorm between fully connected layers

Fig. 6: Network performance with BN and without BN

Batch normalization is similar to normalization which aims to make the features comparable between the layers. During the learning process, the weights and parameters are adjusted and the data flowing through the network during this update between parameters can go really high or can go really low. This is called ”internal covariate shift” [4] and batch normalization aims to solve this problem by normalizing batches.

Method	Validation Loss
Without BN	32.02
With BN	6.58

TABLE I: Validation losses for BN and non-BN implementations

L2-regularization & Dropout-regularization

Regularization is used to prevent overfitting, it prevents the network from focusing on the training data too much. Complex networks, as they get more complex tend to memorize on the training data and it is especially the case when there is not enough training data. Dropout randomly kills (with the probability given as the parameter) some of the neurons during the training process.

Method	Validation Loss
Without Dropout	32.02
With 0.1 Drop	6.27
With 0.25 Drop	**6.21**
With 0.4 Drop	6.45

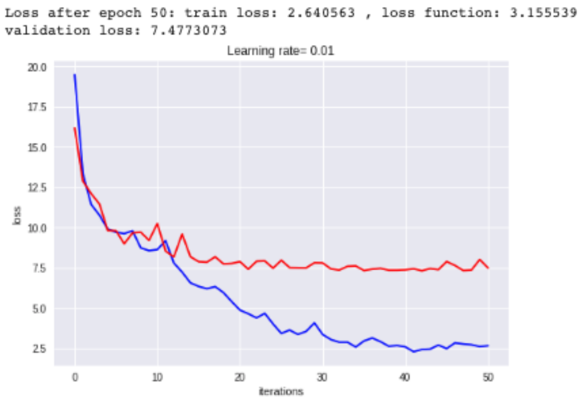
TABLE II: Dropout Losses

Best dropout probability in the network is selected as 0.75 to keep the neurons, (0.25 to drop them) since it gives the least validation errors.

On the other hand, the performance of the network did not change with L2 regularization and only regularization weight 0.00001 was used.

Adam optimizer

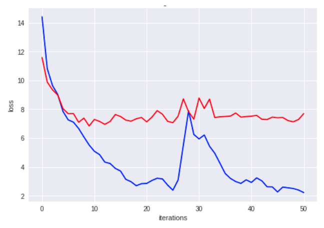
I used gradient descent optimizer at first while trying my vanilla network with the training data. The network worked and learned easily with Gradient Descent, however when I compared it to Adam Optimizer, I clearly saw the difference between the smoothness of two results.



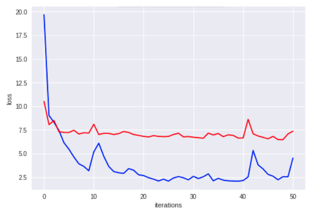
```
Loss after epoch 50: train loss: 2.640563 , loss function: 3.155539
validation loss: 7.4773073
```

Fig. 7: Vanilla network with Gradient Descent

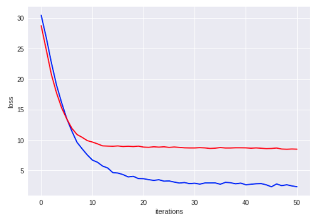
Then, I tried AdamOptimizer with different learning rates, (with 108 batch size and 50 epochs). As it can be seen from the below figures, the learning rate and number of iterations need to be tuned pairwise, what I mean is that if we have a very low learning rate number of epochs need to be higher so that network would be able to learn. However we see that with number of epochs at 50, with decreasing learning rates we get worse and worse results. On the other hand, if we have a higher learning rate, the lower number of epochs would be enough for network to learn.



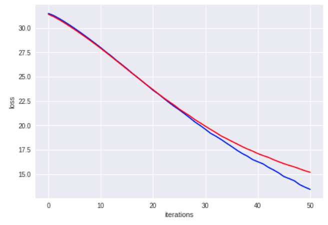
(a) 0.1 Learning rate



(b) 0.01 learning rate



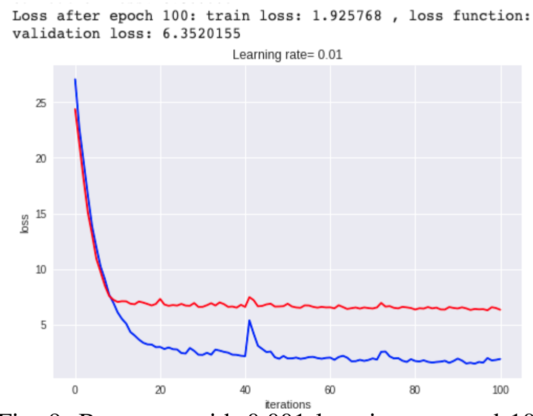
(c) 0.001 learning rate



(d) 0.0001 learning rate

Fig. 8: Different learning rates

My best setup is with number of 100 epochs and with learning rate 0.001 where I find validation errors of less than 6.30 .



```
Loss after epoch 100: train loss: 1.925768 , loss function: 6.861259
validation loss: 6.3520155
```

Fig. 9: Best case with 0.001 learning rate and 100 epochs

Test Loss

Finally the network was tested on the test set with my best setup (0.01 learning rate, 100 epochs) and the result is given below, The result is similar to the validation set results however the only difference here was the batch sizes that were fed to the network.

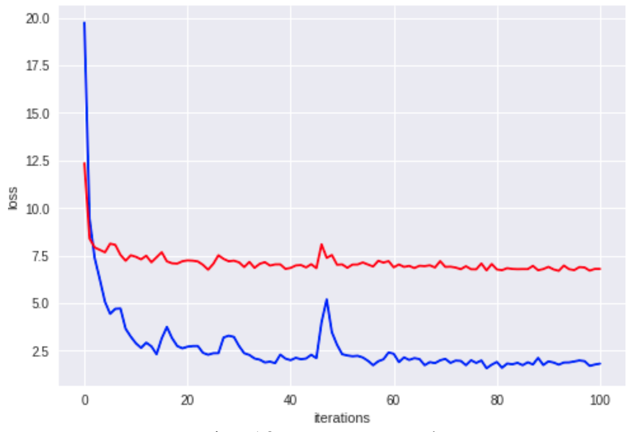


Fig. 10: Test set result

Test Set loss
6.69
6.70
6.76

TABLE III: Test set loss

Used Packages

During my experiments I used numpy, PIL, os, matplotlib libraries in python and in Tensorflow I used from ***tf.contrib***,

- layers.conv2d
- layers.max_pool2d
- layers.batch_norm
- layers.fully_connected
- layers.dropout
- layers.flatten

from ***tf.losses***,

- losses.mean_squared_error
- losses.absolute_difference

from ***tf.initializers***,

- truncated_normal
- xavier_initializer()

from ***tf.train***,

- AdamOptimizer
- GradientDescent

from ***tf.train***,

- AdamOptimizer
- GradientDescent

References

[1] "Google colab," April 2019. [Online]. Available: https://colab.research.google.com/

[2] TensorFlow. (2019, April) Build a convolutional neural network using estimators. [Online]. Available: https://www.tensorflow.org/tutorials/estimators/cnn

[3] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[4] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.