Introduction

The purpose of this paper is to analyze the deep-head-pose project, discuss, evaluate and prioritize ideas for improvement of this work, and finally, propose, implement and test those improvements.

On a slightly lower level, the work will be split into two tasks.

1. Making deep-head-pose more accurate
2. Reducing deep-head-pose inference time by keeping most of the accuracy

The work will be structured as follows. First, we'll go through a short overview of the work done in deep-head-pose project, to be able to justify and reason about the proposed improvements in the rest of this paper. Then, we'll structure the whole effort into areas of potential improvement, so it's easier to understand the scope. After that, we'll propose and describe the reasoning behind some of the ideas, and finally implement and test the most promising ones building on top of the shared deep-head-pose code.

Deep Head Pose - Project Overview

The deep-head-pose project deals with end-to-end image to head pose estimation. In other word, the inputs are 3-channel image tensors, and the outputs are three intrinsic Euler angles - yaw, pitch and roll.

On a more practical note, authors frame this as a classification problem. They bin angles, ranging from -99 to 99 degrees into bins of size = 3, making this a three 66-class classification problems. They use a relatively simple approach and architecture - a ResNet for feature extraction, followed by three independent parallel fully connected layers responsible for the three classification sub-problems. Also, a dual loss is employed, combining a crossentropy loss (for bin classification) and a MSE loss for the angle regressed as a softmax-based weighted average of classification outputs.

On the data side, they train on 300W_LP and evaluate on AFW2000 (they do a lot more things but we will focus on this part of the problem).

In the following sections, we will try to detect weak points of this research, fix them, and improve the performance.

Task 1 - Improving MAE on AFLW2000

Validating results from the paper

Before we start, it will be useful to make the code work and try inference on the pre-trained architectures that the authors provide as .pkl files on Github.

There are two models, with alpha = 1 and alpha = 2. They are trained on 300W_LP. We will test them on AFLW2000, as it's done in Table 1 in the paper - the one we will be comparing against. Let's take a look at the results.

```
python test_hopenet.py --data_dir "/AFLW2000/" --filename_list
test_filenames.txt --gpu 0 --snapshot hopenet_alpha1.pkl --model 'ResNet50'

python test_hopenet.py --data_dir "/AFLW2000/" --filename_list
test_filenames.txt --gpu 0 --snapshot hopenet_alpha2.pkl --model 'ResNet50'
```

Alpha	Yaw	Pitch	Roll	MAE
1	6.9200	6.6373	5.6745	6.4106
2	6.4700	6.5590	5.4358	6.1549

As we can see, the results closely match the ones referenced in the paper. Our first task will be to try to bring these even lower.

Our attempts to do so will be grouped in four areas:

- Architecture related
- Loss related
- Training scheme related / Other
- Dataset related

Architecture related improvements

The ResNet50 architecture in the paper seems somewhat arbitrarily chosen. It makes sense to try bigger ResNets (101 for instance) or even other architectures that perform well on ImageNet. Inception v4 comes to mind, but as it performs relatively comparably to ResNet101, this effort will be left for future improvements

The table below compares MAE for the different architectures we tried out (all trained with alpha = 2 and all other default parameters from the paper, for simplicity):

Architecture	Yaw	Pitch	Roll	MAE
ResNet50	6.3697	6.4574	5.3515	6.0595
ResNet101	**6.3562**	**6.2870**	**4.9182**	**5.8538**

Loss related improvements

It's interesting that the bin width seems to have been chosen arbitrarily in the paper, yet it makes sense for that decision to be one of the most important ones. For instance, choosing 3 degrees as bin width, we cannot theoretically get below MAE = 1.5, and practically anything below MAE = 3 would be surprising.

It makes sense to try different bin sizes and see which one works best for 300W_LP dataset and chosen architecture (ResNet18 here, for faster training). The table below summarizes the results of this part of research.

Bin width	Yaw	Pitch	Roll	MAE
6	8.2892	7.1025	6.1938	7.1952
3	6.1019	7.0954	5.5514	6.2496
1	7.5849	6.7592	5.6159	6.6534

By the results of this table, it looks like there is not enough data for the model to be able to learn the 198-class classification problem well and/or the measurement error is bigger than 1 degree in the training set. It would be interesting, though, to try the bin_width of 1 degree with one of the bigger architectures on a bigger / heavily augmented dataset. We'll leave that as a potential future improvement.

Training improvements

Improving the training scheme sometimes leads to improving accuracy. Here, we will try the following things:

1. Two-step training
2. Using a better learning rate scheme

Two step training

The authors of the paper load a pretrained ResNet50 model and further train it on 300W_LP dataset, but they keep the first convolutional layer frozen throughout the whole training. We propose to add an additional phase at the end of the training. So, after the 25 epoch authors

propose, we add additional phase of fine tuning where we unfreeze the whole network. For this, we reduce the learning rate to 3e-6, instead of the regular 1e-5.

The results of this approach are shown in the following table, on various ResNet architectures.

Architecture	Yaw	Pitch	Roll	MAE
ResNet18	6.5513	7.2798	5.8554	6.5622
ResNet18 2-step	**6.3220**	**6.9047**	**5.4218**	**6.2161**
ResNet50	7.1912	6.4521	5.1314	6.2582
ResNet50 2-step	**6.7607**	**6.2396**	**4.9781**	**5.9928**
ResNet101	6.3562	6.2870	4.9182	5.8538
ResNet101 2-step	**6.2089**	**6.2477**	**4.7987**	**5.7518**

Tracking validation loss

The authors of the paper and the code also haven't been splitting data into train and validation sets. We did this to understand the learning process better.

We randomly split the dataset in 90:10 train/val splits and track validation loss and MAE at the end of each epoch. Using these metrics, we have concluded that it's good to train for 25 epochs with the first conv layer frozen (as authors suggest) and add an additional phase of 5 epochs with the mentioned layer unfrozen. After these numbers, validation MAE starts to saturate so we found that it's not meaningful to continue training.

Learning rate decay

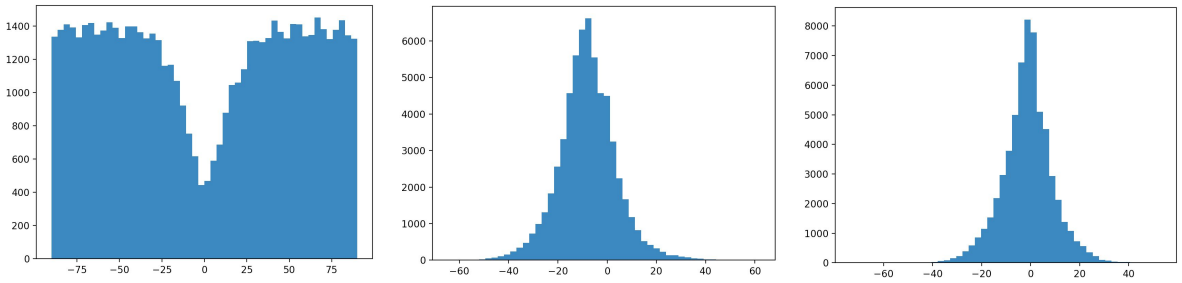
We have also observed that validation loss is pretty unstable towards the end of the training - that means that the learning rate was to big at that point. That's why we employed a simple strategy of reducing the learning rate by 5% on each epoch. That lead to more stable validation losses towards the end of the training and, thus, better optima.

Dataset related improvements

The networks trained in section 4.1 of the paper - the one we're focusing on - are trained on 300W_LP dataset. 300W_LP is a relatively large dataset (61225 images). It contains images from the 300W and their synthetically generated derivatives.

The augmentation is done by fitting a 3D model to the images from 300W and rotating them around yaw axis. This means that for the yaw angle, the 300W_LP dataset provides a big variety, and pitch and roll remain relatively modest and with narrow distributions. Let's take a

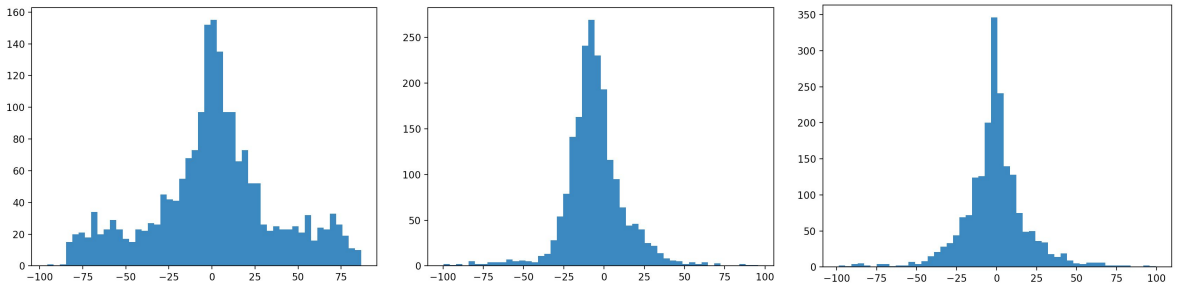
look at what the distributions are. The images below show yaw, pitch and roll distributions, respectively:



As we can see, pitch and roll distributions are fairly narrow. We can also observe this in the following table, where we show 5th and 95th percentiles, in degrees, for yaw, pitch and roll:

Percentile	Yaw	Pitch	Roll
5th	-81.8	-27.0	-18.9
95th	81.8	11.0	15.7

The crucial part for reducing MAE is how these distributions compare to the distributions from the AFLW2000 dataset. If AFLW2000 has a significantly wider spread, no ML model can solve that well. So, let's take a look at the distributions from AFLW2000 in the same fashion:



And the table with percentiles:

Percentile	Yaw	Pitch	Roll
5th	-68.1	-31.0	-33.5
95th	68.4	26.3	33.7

By observing the values from the two tables above, we can definitely say that the test dataset has a significantly wider distribution of pitch, and particularly roll, than the training set.

The asymmetry of the training set is also a problem for pitch, as flipping around y axis doesn't help. With roll it does help solve the asymmetry, but still, angles in the 20-30 degree range are very poorly covered by the training set. It looks like there is a lot of room for improvement there.

So, to tackle this, we will try to further augment the 300W_LP dataset to have wider pitch and roll distributions with a nice trick. Namely, by rotating the images from the dataset by relatively small angles (±20 degrees), we can get a much richer distribution of pitch and roll.

To be able to do this, we just need to iterate through images from the dataset, perform the transformation of the yaw, pitch and raw for the performed rotation, and we can get arbitrarily many new images. The rotation angles shouldn't be too big, though, to avoid losing too much of the original image. The details on how we did this are located in the Appendix at the end of the document.

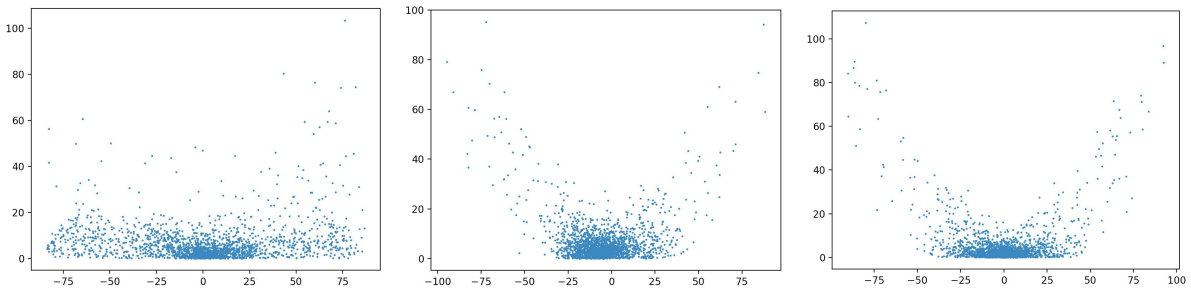
We choose to rotate by -20, -10, 10 and 20 degrees respectively. We augment only a small, randomly chosen, fraction of 10% of the original 300W_LP, to keep the size of the augmented dataset in the same order of magnitude as the initial one. We then add those images to the 300W_LP dataset, obtaining what we call the **300W_AUG** dataset, whose distribution we show below.

Percentile	Yaw	Pitch	Roll
5th	-78.8	-45.2	-44.9
95th	78.6	35.2	42.2

We can see that with this simple trick, we got a distribution that is way wider, which will hopefully lead to better learning and generalization to AFLW2000.

Let's take a look at one more indicator that suggests that the augmentation trick should work well. The three charts below show the correlation between true yaw, pitch and roll angles with the errors on those angles made by the ResNet50 model trained on non-augmented data (hopenet_alpha2 model provided by the authors).

Note: Each point is an example from AFLW2000, X axis values are true values and Y axis are *abs(pred_value - true_value).*



The errors on tails are huge. The errors when absolute angles are smaller that 30 degrees are shown in the table below:

Yaw	Pitch	Roll	MAE
4.4949	4.9996	3.2787	4.2578

So, this indicates, that with proper data augmentation, we can possibly reduce the MAE of the ResNet50 Hopenet even below 5 degrees.

In the next section, we will show the best results we got training on 300W_AUG datasets (with all other improvements from the paper so far included in those models).

The Master Models

Now, we will train three models that employ all the improvements we have been able to observe above, trying to move the state of the art needle for this problem as far as possible.

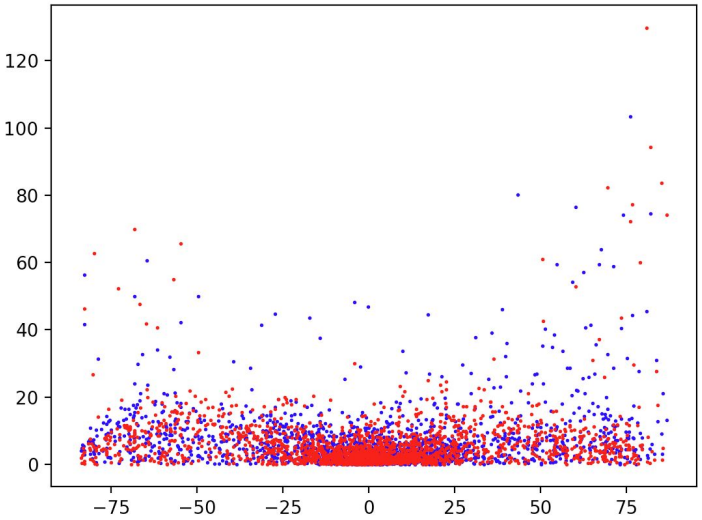
So, the models have the following features:

- ResNet 18, 50, 101, which we call **Master 18, 50, 101**, respectively
- Trained on 300W_AUG
- Two step training, 25 epochs with frozen first conv layer + 5 epochs with unfrozen
- Decaying learning rate, starting from 1e-5, being reduced by 5% at each epoch
- Bin width set to 3

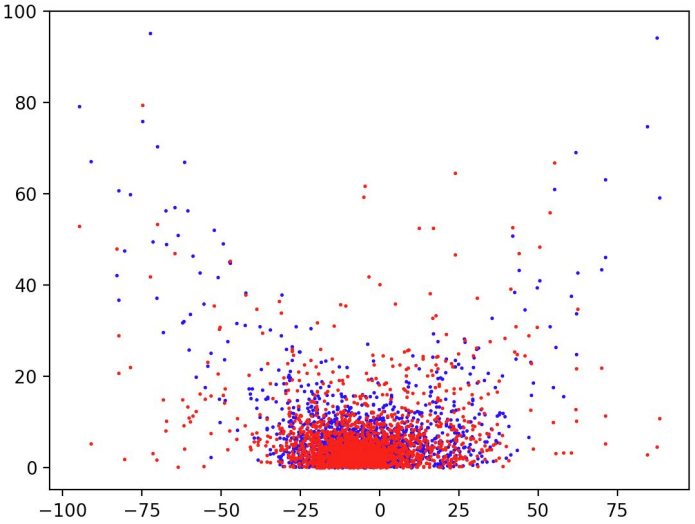
Arch + DS	Yaw	Pitch	Roll	MAE
Master18	5.7212	6.4494	5.0046	5.7251
Master50	5.7294	6.0188	4.8056	5.5179
Master101	**5.5869**	**5.8536**	**4.3402**	**5.2602**
hopenet_alpha2	6.4700	6.5590	5.4358	6.1549

Let’s now take a look back at how error scatter plot looks for our Master50 (red) vs the hopenet_alpha2 model (blue) that authors of the paper have provided. These scatter plots show nicely how our data augmentation technique has significantly improved the performance for higher angles.

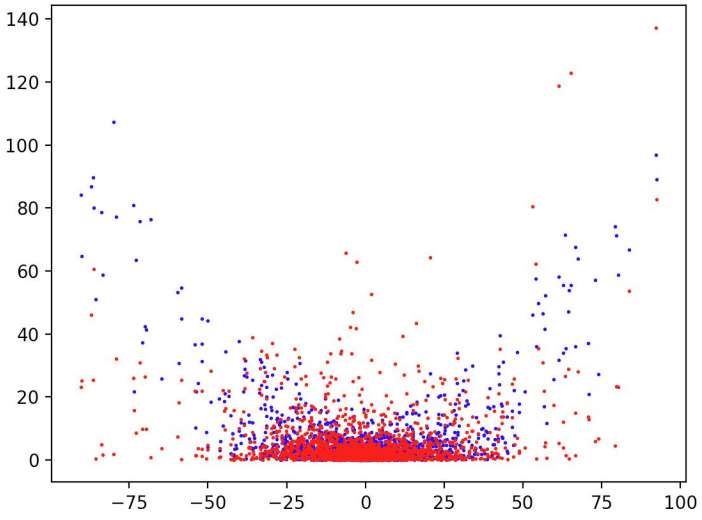
Yaw:



Pitch:



Roll:



Task 2 - Improving inference time

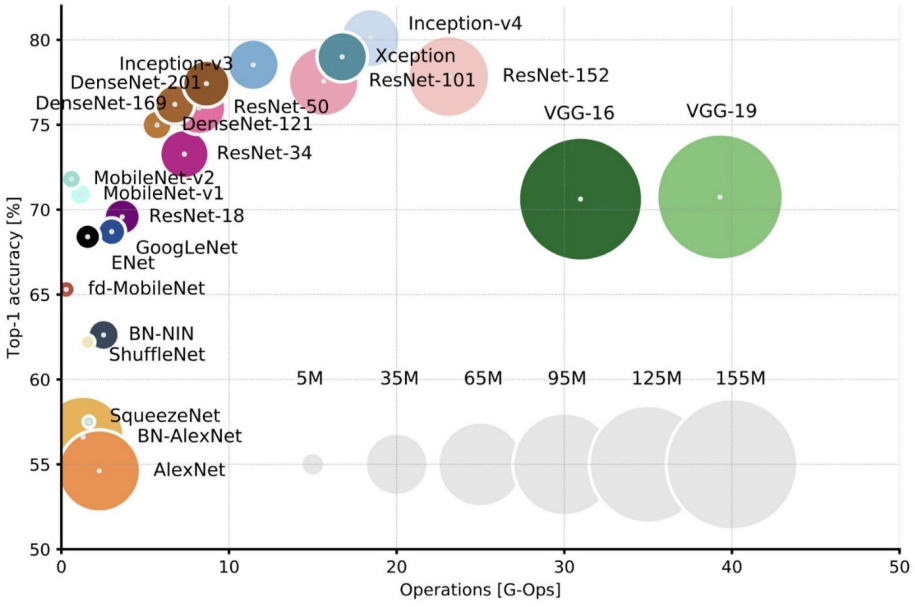
Architecture related improvements

Inference time is heavily determined by network architecture. If we're able to get a relatively similar MAE with a significantly smaller network (with less floating point operations), that would be great for many applications where speed is important.

We're going to compare a couple of different architectures. For each of them, we're going to observe MAE and inference time. The networks are:

- ResNet50
- ResNet18
- MobileNetV2

The following image describes the reasoning behind these architectures. Resnet18 is a logical choice, as ResNet50 is already proven to work - so ResNet18 is faster and straightforward to try, both from engineering perspective (changing a couple of lines of code) and from the perspective of learning subtleties (similar hyperparameters, similar training scheme should work). Mobilenet v2 is also very fast, while retaining decent accuracy.



The table below shows MAE and inference times on a single Tesla V100 for the three architectures we are observing.

Architecture	MAE	Inference time (ms)
ResNet50	5.5179	14.00
ResNet18	5.7251	5.73
MobileNetV2*	6.2748	10.01

*We didn't deal with optimizing hyperparameters for MobileNetV2, thus the bigger MAE

As we can see from the table, by using ResNet18, along with all the improvements listed in this paper, we can reduce inference time by almost 2.5x, while having even better MAE than the one listed in the paper with ResNet50.

Other improvements

This model can be further sped up for use in embedding systems, using techniques like quantization and/or weight sharing, but this was out of the scope of this work.

Conclusions

In this work, we have analyzed the work presented in the deep-head-pose project, found some areas for potential improvement and implemented some of the ideas to improve these areas.

We were able to observe a significant decrease in MAE (~15% decrease on ALFW2000) by architecture, training scheme and dataset augmentation changes.

We were also able to decrease training time by roughly 2.5x, while still decreasing MAE by 7% compared to the original work.

Appendix

Code changes

The following changes have been made to the code:

1. Changes to the original code:
    a. Additional MobileNetV2 model is added to the `hopenet.py`, along with `train_mobilenet.py` that runs training for MobileNetV2.
    b. Parameters that enable changing bin size (and so the number of classes) and unfreezing first conv layer are added and their impact implemented (`--bin_width_degrees` and `--unfreeze`)
    c. Learning rate decay is implemented
    d. Dataset is now split into train and validation sets
    e. Different ResNets are now supported, by changing `--model_type` to "`ResNetXX`", and links to their pretrained state graphs added to the code
    f. Error tracking is added to `test.py`, exporting stats to `error_stats.json`
    g. Tracking inference time is added to `test.py`
2. Additional code used side jobs
    a. Script that analyzes dataset distributions - `analyze_params.py`
    b. Script that augments the dataset by rotating images and re-calculating Euler angles and landmark positions - `rotate_images.py`
    c. Scripts that make the scatter plots we used in this paper - `analyze_errors.py` and `compare_errors.py`

Trained models

We also provide trained Master models for easier validation of results in this work. They are uploaded to Google Drive and links are shared in the README.md file.

Augmenting the 300W_LP

The idea behind the augmentation is, that when we rotate an image in the 2D image space, we are able to determine the new yaw, pitch and roll angles.

We know that the transformations for the projections of x, y and z axis in the head coordinate system to the 2D image plane are (as shown in the draw_axis function in utils.py):

```
x1 = cos(yaw) * cos(roll)
y1 = cos(pitch) * sin(roll) + cos(roll) * sin(pitch) * sin(yaw)

x2 = -cos(yaw) * sin(roll)
y2 = cos(pitch) * cos(roll) - sin(pitch) * sin(yaw) * sin(roll)
```

```
x3 = sin(yaw)
y3 = -cos(yaw) * sin(pitch)
```

And these vectors rotate together with the image, as they're in the same 2D space. So, we rotate them in the following fashion:

```
new_x = cos(angle) * x - sin(angle) * y
new_y = sin(angle) * x + cos(angle) * y
```

And perform an inverse transformation to the one above, to obtain back the yaw, pitch and roll intrinsic angles:

```
yaw = asin(x3)
pitch = asin(y3 / (- cos(yaw)))
roll = asin(x2 / (- cos(yaw)))
```

These are the yaw, pitch and roll angles of the rotated image. We store them along with landmarks into .mat files and use for training in the same way as the images from the original 300W_LP dataset.