Few-Shot Adaptive Gaze Estimation

Final Paper

Navneet Singh Arora and Diana Rueda

28.01.2021

Abstract

Gaze estimation is done usually in controlled environments with specialized equipment. These gaze estimation techniques also, do not take into account person-independent gaze. Thus, making it difficult to generalize and use those applications in real-time. Even after considering them, it requires a considerable amount of iterations and calibrations (in the order of thousands) to achieve minimal improvement in the overall estimation, making it inconvenient [3]. Few-shot Gaze estimations help to bring in person-specific gaze estimation that can then be applied to a new person with calibrations as low as 3 samples. This also takes into fact that the models are not over-fitting, yielding significant performance improvement [1].

1 Introduction

Eye orientation differs for every person, having specific eye-ball model, optical axis among others [2]. Attaining high accuracy considering these specific orientations for gaze estimation is difficult and achieving it in an uncontrolled environment remains far from being robust and efficient. Even though the Convolutional Neural Networks have brought down gaze estimation errors to its lowest levels, these are not low enough for public interaction applications [4] or crowd-sourced attention [5]. Additionally, personalizing CNN's to few-shots leads to over-fitting in most of the scenarios [1].

These challenges are handled through the deployment of the proposed FAZE framework (Figure-1), using disentangling encoder-decoder architecture to learn rotation aware gaze estimation. This framework yields higher performance with fewer samples.

2 Related Work

Gaze Estimation. Earlier approaches cropping up from the classical models tend to capture the images in a controlled laboratory environment [13]. These approaches are modelled with the regression or through random forest synthesis learning to generate head-pose clusters. Multi-modal networks like VGG-16 [14] applying LeNet-5 architecture have progressively decreased the gaze errors to as low as 4.3°. These models further were assisted by the increasing availability of large data-sets like MPIIGaze and GazeCapture [15].

Although plenty of advancements have made through complex CNNs allowed for improved learning, person-independent gaze errors are still far from robust. Furthermore, person-specific models require thousands of samples per subject for any significant improvement [16].

Few-shot Learning. Learning from a very few samples poses a non-trivial problem which leads to over-fitting because of highly over-parameterized networks. On the flip-side, few-shot learning along with meta-learning technique has proved to be promising [17, 18, 19, 20, 21, 22, 23] and successful in tasks like object recognition [21].

3 FAZE Framework

Few Shot Adaptive Gaze Estimation (FAZE) Framework, as the name suggests adapts to new persons from person-specific gaze estimation learning using multiple features. It leverages insights from observing the eye-region appearance variations across a multitude of people with different head pose and gaze direction configurations [1].

The framework is based on an encoder-decoder architecture with rotation equivariant functions to learn rotation aware gaze representation. The learning is achieved by extracting appearance, gaze and head pose features, further comparing them to the images of the same person having different gaze direction. It, in turn, returns an embedded loss term which further minimizes the intra-person differences. Moreover, it is used for gaze estimation with fewer samples.

The framework leverages meta-learning, where each subject is seen as a new task for the meta-learner [1] and thus, improving performance. Evidence shows that this FAZE framework outperforms all the other state of the art methods by significant margins.

FAZE Framework consists of two specific stages:

- The first stage makes use of the Disentangling Transforming Encoder-Decoder (DT-ED) as depicted in (Figure 2) for obtaining the appearance, gaze direc-

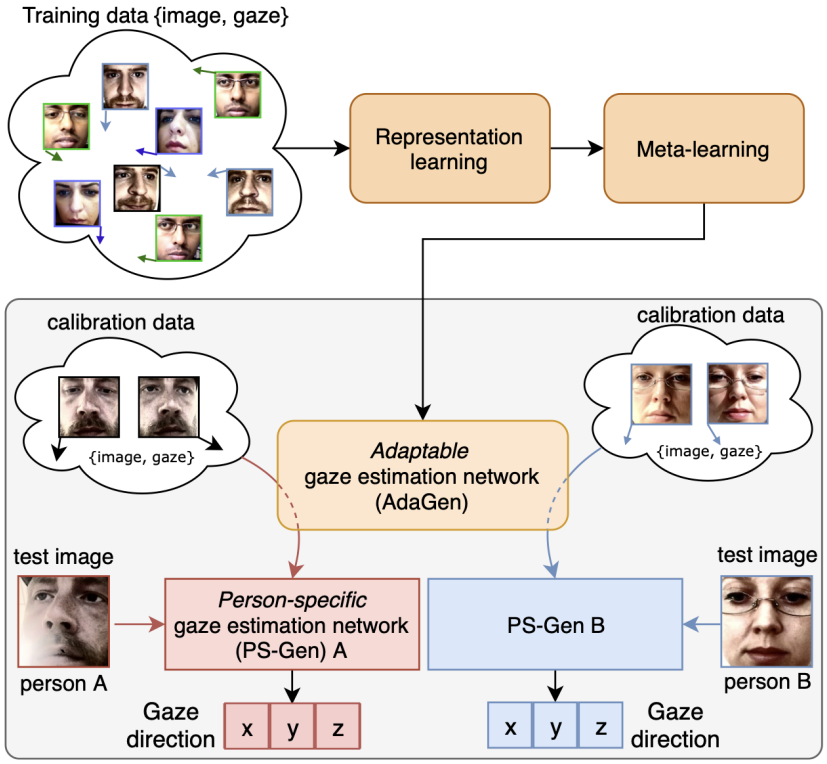


Figure 1: Overview of FAZE Framework. Training images with ground-truth gaze direction information as the source and Gaze Direction as the result through AdaGEN, adaptive gaze estimation network and person-specific gaze estimation network (PS-GEN).

tion and head rotation features, getting a robust feature extractor in succession.

- These features feed the second stage that seeks to adapt the findings for specific persons, making use of the adaptable gaze network (AdaGEN) that applies a meta-learning method (MAML) and finalizing with the produced person-specific model (PS-GEN).

4 Architecture and Model Analysis

Architectural overview considers three factors in problem setting namely gaze direction, head orientation and appearance of eye. All these factors are related to the appearance of the eye region in the input images. These three factors are disentangled explicitly by known unique differences in rotations to their respective codes. It is referred to as an architecture called Disentangling Transforming Encoder-Decoder (DT-ED) [1].

Before diving deep, the current paper starts with the main components of the whole architecture.

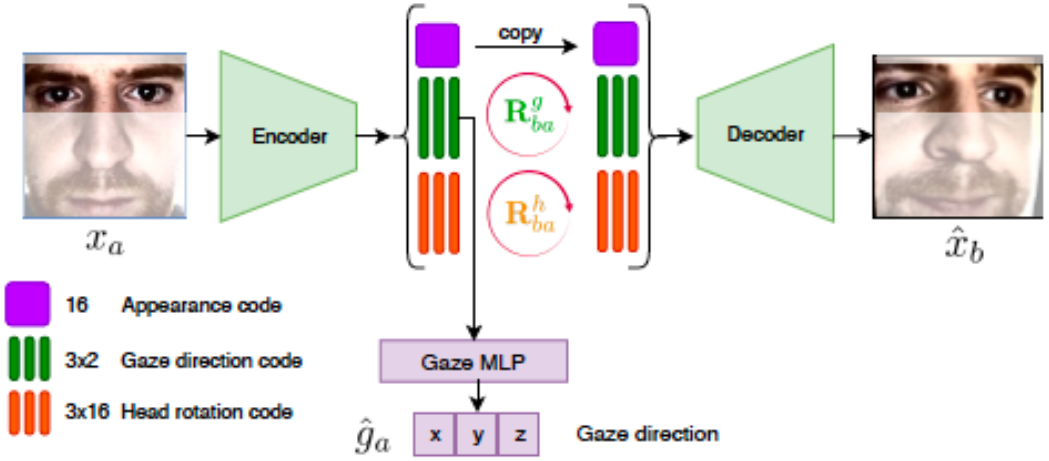


Figure 2: Disentangling appearance, gaze and head pose variations from an image with our Disentangling Trans- forming Encoder-Decoder (DT-ED).

MAML Meta Learning. A model-agnostic meta-learning algorithm is compatible with any model trained with gradient descent and applies to various learning problems, including classification, regression and reinforcement learning [6]. It enables learning new tasks with a small number of training samples by training the model on several learning tasks.

Disentangling Transforming Encoder-Decoder (DT-ED). For getting input information that can be generalized, this method is devised by extricating explicitly the appearance, head rotation and gaze direction by applying matrices rotations.

Additionally, we represent the eyes and heads orientations with Euler angles in rotated matrices form and calculate rotations differences between persons by comparing with the ground truth values. Finally, this Encoder-Decoder architecture is trained with a multi-objective loss function that integrates three defined losses (reconstruction, embedding consistency, and gaze direction) including additional parameters.

Encoder-Decoder architecture is implemented using a CNN DenseNet architecture.

Adaptive Gaze Estimation Network (AdaGEN). The network is trained with the aid of MAML meta-learning method, personalizing the findings of the DT-ED. The few-shot of the title is coined from the advances introduced by a better generalization through iteration to find optimal values of the AdaGEN weights, extracting more information from "tasks" (subjects) in turn needing fewer samples or "shot" to produce accurate results.

It further deploys a multi-layered perceptron, for the AdaGEN and the Gaze estimation function.

Person Specific Model (PS-GEN). For obtaining the final gaze estimations values per person, it's proposed to fine-tune the resulting AdaGEN model with k number of calibration images (k is less than or equal to 20).

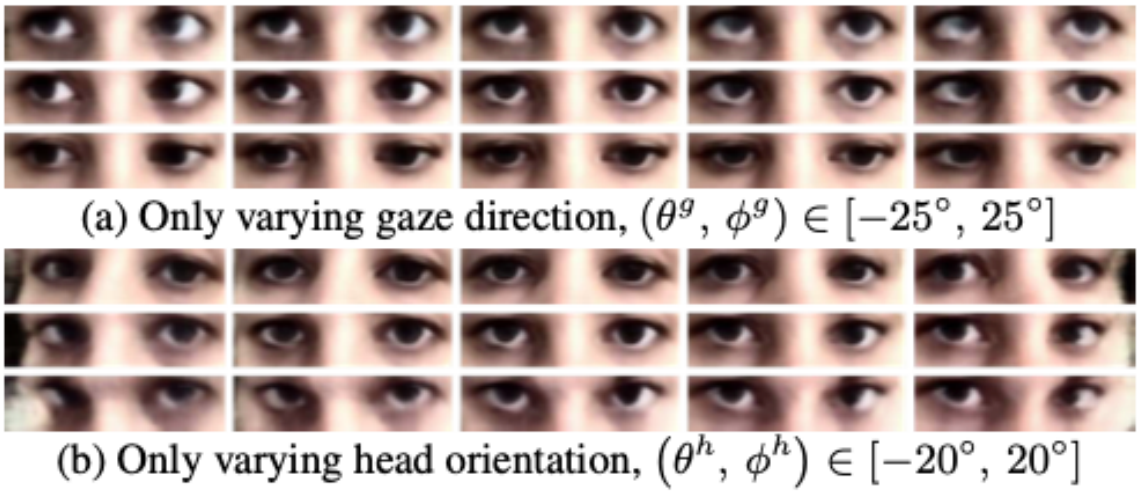


Figure 3: Disentangled rotation-aware embedding spaces for gaze direction and head pose demonstration

Overall, all the components work hand in hand to achieve the few-shot gaze estimation. The latent space embedding (appearance, gaze direction and head-pose) is converted into dimensions which are further rotated through rotation matrices via DT-ED in order to represent the dimensions as Euler angles (describing the orientation in 3D space). These dimensions working as ground truth for the input samples are then used to find out the rotational change for the training images provided as an input.

This approach is equally applicable to real-world noisy imagery (Figure-3). It takes into account the person-specific differences but still making sure that the identical gaze direction is marked to similar features.

After learning the person-specific features, and using the fewest samples as will be available in real-world, needed to train the DT-ED, fine-tuning is further applied to avoid over-fitting. To do so, MAML meta-learning method is considered which in turn help in the creation of AdaGEN. This benefits from the stochastic gradient approach to minimize the generalization loss, therefore, learning the closely related features for a successful few-shot learner of any new task.

5 Implementation and Performance

Pre-requisites. Before moving towards any execution, there are required pre-requisites which needs to be fulfilled, to be able to execute the code. It includes fetching the datasets including both MPIIFaceGaze as well GazeCapture.

Apart from this, it is also required to fetch the supplementary information about these datasets, including the labelled information containing the head position, gaze origin, gaze target and virtual camera reference points using the camera matrix.

Data-sets. The model uses tow largely available data-sets, GazeCapture and MPIIGaze. GazeCapture is the largest gaze dataset. On the other hand, MPIIGaze is the largest benchmark data-set for gaze estimation and in comparison to GazeCapture, it has higher within-person variations including illumination, make-up and facial hair, potentially making it more challenging.

1.7 million samples of GazeCapture, including data from 993 people with 1766 samples per person on an average were used for training the model. Subjects with more than 400 samples are chosen to maintain within-subject diversity. For evaluation, 15 subjects with 2500 samples each are chosen, keeping last 500 samples fixed for every subject and randomly sampling from all the other samples.

Data pre-processing. Once the required information is available through the data-sets, a normalized procedure [8] is used which utilizes multiple markup points using 4 eye corners and 9 nose landing marks [9] [10] [11]. The normalized function ensures upright head position (Figure-4.) through these referenced markup points, before doing head pose estimations. In contrast to other implementations which use the mouth position marks instead of the nose landmarks, this marking procedure is more robust as mouth marks are not sufficiently static due to the changing facial expressions.

Training, Configuration Tuning and Meta Learning. The DT-ED model was originally trained using a batch size of 1536 on 10^6 training samples with 50 epochs using 8 GPU's in parallel using 32GB GPU model. On top of this 64 layers neural network is used for precise feature learning.

As the training process requires a lot of GPU computation and parallel processing, we in our execution rely on low end GPU available through Google Collaboratory. These samples are then supplied to process, reducing parallel GPU nodes from 8 to 2. Also, the batch size is reduced significantly from 1536 to 64 with 20 epochs. We also reduce the parallel script execution from 18 to 6 and only using 9 calibration samples. This is necessary in order to execute the code and get the outcome successfully. For the final tuning to speed up the process further, we change the neural network from 64 layers to 32 layers and also reduce the learning steps from

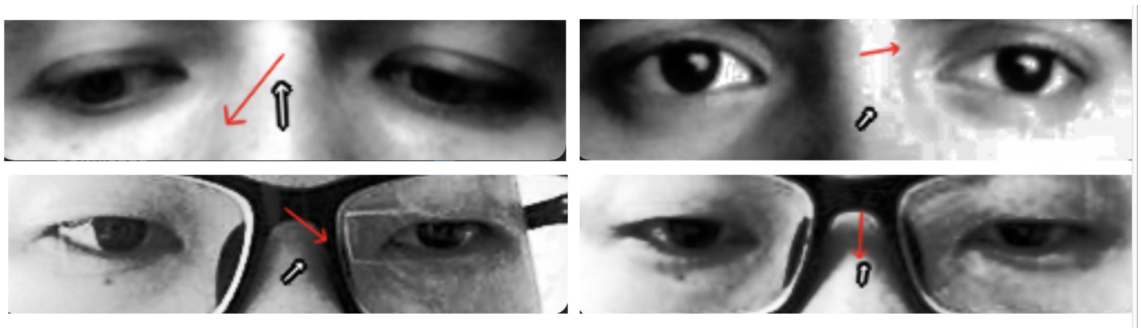


Figure 4: Normalized patch of MPIIFaceGaze Dataset used to create the normalized matrix for the final execution using MPIIFaceGaze Supplementary Labels of Tilt, Gaze Origin, Gaze Target and Virtual Camera reference points. Red arrow points towards the gaze target and broader white arrow depicts the camera reference points. This images have been captured and combined during the implementation phase.

Parameters	**Original Model**	**Experimentation**
Neuron Layers	64	32
Epochs	50	20
Batch Size	1536	64
Parallel Computation	16 Parallel Threads	6 Parallel Threads
Training Iterations	1000	100
Number of GPU Nodes	8	2
Meta-Learning Samples - k	18	6

Table 1: Comparative table showing the change parameters for the experimentation

10000 to 1000.

Finally, we execute the code in 2 different ways. Firstly, by using the pre-trained weights available which helps skip the entire training process and directly execute the meta learning. Secondly, by training the model and then executing the meta learning process.

Performance. As stated above, the hardware requirements for this implementation are resource-intensive, considering the complexity of the operations, the parameters detailed above (Table-1) were changed under the following assumptions:

- The neuron layers were reduced, expecting that by having a not so deep CNN, the operation would run in a realizable time, yet the capacity of the algorithm to generalize is diminished.
- The epochs were also reduced looking to test how well would the algorithm perform with fewer optimization rounds, and aiming to prevent over-fitting.

- To optimize resources, the batch size was decreased as well, to allow for the running implementation to not take as much RAM in the simulation. In this way, the run time was longer, yet it is guaranteed that it runs on the current system's conditions.
- Lowering the parallel computation parameters was thought to have the same effect as reducing the batch size.
- By reducing the training iterations, the intention was to test if the algorithm returned a mean error low enough for it to be acceptable.
- The number of GPU nodes were decreased, to be able to simulate the google collaboratory set-up.
- Finally, meta-learning samples, or k parameter, was changed from 18 to 6, to gather just as with the training iterations, if the algorithm was able to learn enough in these limited conditions while maintaining an acceptable mean error.

6 Results

For all methods, person specific gaze estimation errors are reported with k calibration samples which are randomly chosen from the set of available samples.

Ablation Study and Loss Terms. Firstly, different settings are chosen to understand and compare the impact. The study is done using few-shot gaze estimation using MAML and is compared using fine-tuning, vanilla auto-encoder and DT-ED. The MAML (DT-ED) model outperforms all the other estimation se-

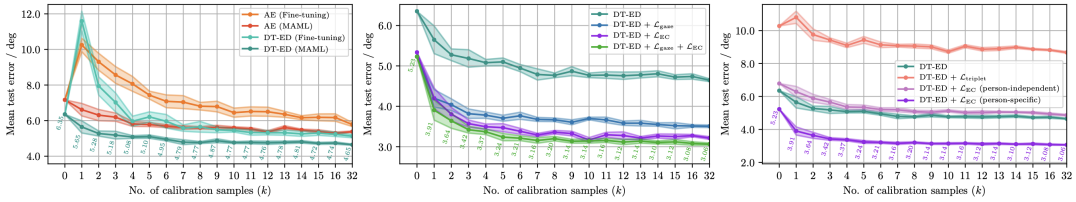
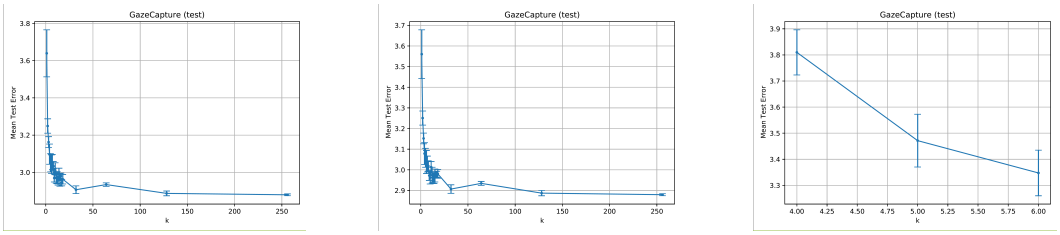


Figure 5: Ablation Study: Impact of (a) learning the few-shot gaze estimator using MAML (Sec. 3.3) and using the trans- forming encoder-decoder for feature learning (Sec. 3.2); (b) different loss terms in Eq. (2) for training the transforming encoder-decoder; and (c) comparison of the different variants of embedding consistency loss term (Eq. (4)).

tups, significantly decreasing the error e.g., 4.87° vs. 5.62° with k = 9 for DT-ED (MAML) and AE (MAML), respectively and the gains are consistent throughout the chosen calibration samples. The only difference between DT-ED and AE is that the latent codes are rotated in DT-ED before decoding [1] and despite this

more difficult task, DT-ED code clearly outperforms person-specific gaze estimation as it incorporates gaze supervision to obtain features that are more informed of the ultimate task of gaze-estimation giving an improvement of 26% from 4.87° to 3.60°. Further adding the consistency loss terms, the combined method improves the performance even further to 3.14°.

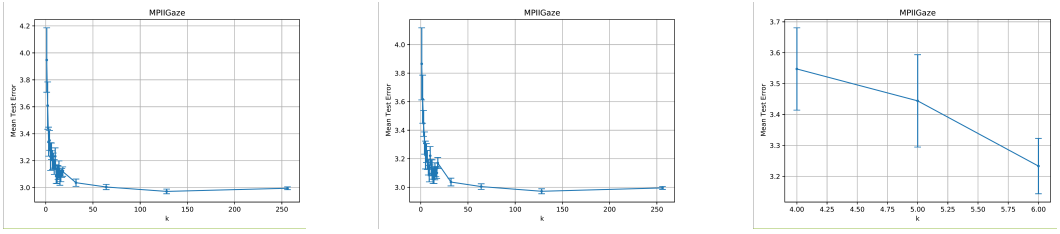
Further Experimentation: Pre-trained Weights v/s Concurrent Training. The model was further executed, trained, tested and compared with the original paper using the pre-trained weights as well as re-training the model with changed parameters to evaluate the learning outcomes of the model.



(a) (b) (c)

Figure 6: GazeCapture Results depicted in Mean Test Error against different values of k (a) Original Result (b) Result using Pre-Trained Weights (c) Result using Concurrent Training with lower calibration samples

On further evaluation of the model, we executed the code using the pre-trained weights to avoid the overhead of training the model and then compare it to the original model. As it can be seen through Figure-6 (a and b), we could notice the performance being equivalent to the original model with similar decrease in the mean error. As per the authors , the model performs well with lower calibration samples as well. We try to test it with low calibration samples by applying the set of changes mentioned in Table-1.



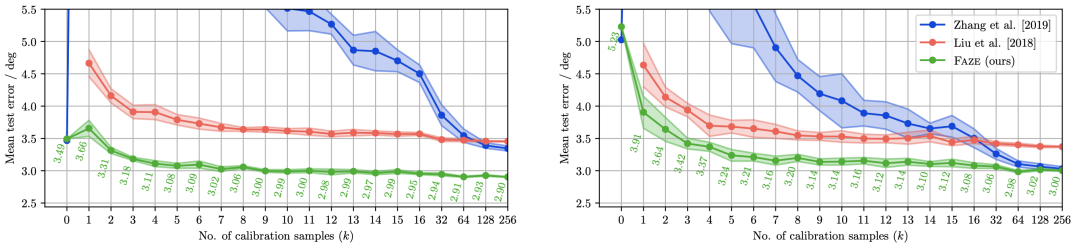
(a) (b) (c)

Figure 7: MPIIGaze Results depicted in Mean Test Error against different values of k (a) Original Result (b) Result using Pre-Trained Weights (c) Result using Concurrent Training with lower calibration samples

It is worth noting that the model performs significantly well, reducing the mean error significantly with performance comparable to the original model. Similar

outcome is seen through Figure-7 where the model is evaluated against the MPI-IGaze data-set, yielding 3.35° v/s 3.1° mean error.

Overall Comparison. Overall, the results show significantly better mean errors



(a) GazeCapture (test) (b) MPIIGaze

Figure 8: Comparison of FAZE against state-of-the-art person-specific gaze estimation methods.

as compared to the other methods. In addition, performance is more consistent, indicating robustness irrespective of value of k calibration samples. The results are monumental as even with k as low as 4, it achieves equivalent to best performance of other state-of-the-art methods.

7 Evaluation

Limitations. The hardware requirements for this model are demanding for a dataset which is not extremely large, demanding to use of 8 core GPU's, processing with parallel GPU system. In real time, most of the images have to be extracted from raw video in order to get the complete latent space orientation; thus, requiring heavy computing.

Also, this approach takes into account perfect healthy human efficient eye [7]. People will Squint, will prove out to be an interesting use case to test out this approach and may even lead to failure. Here, the features (gaze direction, appearance and head position) will tend to match non-similar features.

This does not even take into account various artifacts which can affect the gaze in real time and even in a crowded space; for example, direction of sunlight, accessories like spectacles and so on.

Advantages. Despite of hardware limitations for optimum performance, the model is robust enough to produce comparable results with lower configurations. It outperforms all other state of the art methods, that too significantly. The biggest advantage however is that the model is able to capture latent rotation which otherwise is difficult to handle. Having this capability, enables the model to perform relatively well and achieve an optimum value with k = 9.

Outlook. A lot improvements still need to be made in order to compute and apply this learning on a street or every day life. As the factors affecting the gaze are not considered, it can lead to wrong prediction, for example in Autonomous driving, where sunlight direction can change the overall orientation, thus wrongly apply the brakes as a result.

8 Conclusion

The Few-Shot Gaze Estimation is a personalized gaze estimation method, which using very few calibration samples yields high accuracy outputs. The framework takes into account not only the hard pose and appearance but also the latent embedding of the gaze representation. It tries to do the estimation using various marking points on the face of the subject which are static as well as the virtual camera reference points for better learning. The model tends to outperform other models and reach the best outcome from these models which are achieved with 9 calibration samples as compared to 128 or 256 calibration samples for the other models.

Same observation was made while implementing the algorithm with k=6, using both pre-trained and involving concurrent training. The model is one of its own, providing a near practical solution to something considered difficult to handle, achieving 36% improvement at as low as 9 calibration samples.

However, despite of all the positives, the model is computation heavy and requires dedicated resources to reach the optimum level of training and validation. In the absence of these hardware requirements, the model just takes too long to process the same inputs. Therefore, making it harder to apply in real-time applications. Nevertheless, the model is robust and efficient with high accuracy output.

References

[1] Seonwook Park, Shalini De Mello, Pavlo Molchanov, Umar Iqbal1, Otmar Hilliges, Jan Kautz1, NVIDIA and ETH Zürich. *IEEE Few-Shot Adaptive Gaze Estimation, ICCV 2019.*

[2] Elias Daniel Guestrin and Moshe Eizenman. General theory of remote gaze estimation using the pupil center and corneal reflections. IEEE Transactions on biomedical engineering, 53(6):1124–1133, 2006.

[3] Xucong Zhang, Yusuke Sugano, Mario Fritz, and Andreas Bulling. Mpiigaze: Real-world dataset and deep appearance- based gaze estimation. TPAMI, 2019

[4] Yanxia Zhang, Joürg Müller, Ming Ki Chong, Andreas Bulling, and Hans Gellersen. Gazehorizon: Enabling passers-by to interact with public displays by gaze. In ACM UbiComp, 2014.

[5] Alexandra Papoutsaki, James Laskey, and Jeff Huang. Searchgazer: Webcam eye tracking for remote studies of web search. In CHIIR, 2017.

[6] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model- agnostic meta-learning for fast adaptation of deep networks. In ICML, 2017.

[7] Muhammad Qasim Khan and Sukhan Lee, Department of Electrical and Computer Engineering, Intelligent Systems Research Institute, Sungkyunkwan University, Suwon 440-746, Korea.

[8] Xucong Zhang, Yusuke Sugano, and Andreas Bulling. Re- visiting data normalization for appearance-based gaze esti- mation. In ETRA, 2018.

[9] Ralph Gross, Iain Matthews, Jeffrey Cohn, Takeo Kanade, and Simon Baker. Multi-pie. IVC, 28(5):807–813, May 2010.

[10] Patrik Huber, Guosheng Hu, Rafael Tena, Pouria Mortaza- vian, P Koppen, William J Christmas, Matthias Ratsch, and Josef Kittler. A multiresolution 3d morphable face model and fitting framework. In VISIGRAPP, 2016.

[11] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. Epnp: An accurate o (n) solution to the pnp problem. Inter- national journal of computer vision, 81(2):155, 2009.

[12] Seonwook Park, Shalini De Mello, Pavlo Molchanov, Umar Iqbal1, Otmar Hilliges, Jan Kautz1, NVIDIA and ETH Zürich, Few-Shot Adaptive Gaze Estimation (Supplementary Material), *IEEE Few-Shot Adaptive Gaze Estimation, ICCV 2019.*

[13] Feng Lu, Yusuke Sugano, Takahiro Okabe, and Yoichi Sato. Inferring human gaze from appearance via adaptive linear regression. In ICCV, 2011.

[14] Tobias Fischer, Hyung Jin Chang, and Yiannis Demiris. RT- GENE: Real-Time Eye Gaze Estimation in Natural Environ- ments. In ECCV, 2018.

[15] Xucong Zhang, Yusuke Sugano, Mario Fritz, and Andreas Bulling. Appearance-based gaze estimation in the wild. In CVPR, 2015.

[16] Xucong Zhang, Yusuke Sugano, Mario Fritz, and Andreas Bulling. Real-world dataset and deep appearance based gaze estimation. TPAMI, 2019.

[17] Jake Snell, Kevin Swersky, and Richard Zemel, Prototypical netwroks for few-shot learning. In *Neur*IPS, 2017.

[18] Oriol Vinyals, Charles Blundell, Tim Lillicrap, Daan Wierstra, et al. Matching netwroks for one shot learning. In *Neur*IPS, 2016.

[19] Danilo Jimenez Rezende, Shakir Mohamed, Ivo Danihelka, Karol Gregor, and Daan Wierstra. One-shot generalization in deep generative models. JMLR, 48, 2016

[20] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In ICML, 2016.

[21] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model- agnostic meta-learning for fast adaptation of deep networks. In ICML, 2017.

[22] Alex Nichol, Joshua Achiam, and John Schulman. On first- order meta-learning algorithms. In arXiv:1803.02999, 2018.

[23] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. In ICLR, 2017.

Appendix

Code implementation starts with the pre-requisite steps which are necessary in order to execute the Few-Shot Gaze Algorithm. First step involves fetching the required pre-processing labels for both MPIIFaceGaze and GazeCapture.

grab_prerequisites.bash

```
#!/bin/bash

# Install Python Dependencies
pip3 install --user -r requirements.txt

# Download required additional labels
wget -N https://ait.ethz.ch/projects/2019/faze/downloads/preprocessing/MPIIFaceGaze
wget -N https://ait.ethz.ch/projects/2019/faze/downloads/preprocessing/GazeCapture_
```

After fetching the pre-requisite files for both the datasets, MPIIFaceGaze and GazeCapture, eye-pose angles and head-pose angles are drawn using the datasets and then matrix normalization is performed on the gaze orientation which could then be used as an input for the meta-learning.

create_hdf_files_for_faze.py

```
"""
Copyright 2019 ETH Zurich, Seonwook Park

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
"""
```

```
import os

import cv2 as cv
import h5py
import numpy as np

face_model_3d_coordinates = None

normalized_camera = {
    'focal_length': 1300,
    'distance': 600,
    'size': (256, 64),
}

norm_camera_matrix = np.array(
    [
        [normalized_camera['focal_length'], 0, 0.5*normalized_camera['size'][0]],
        [0, normalized_camera['focal_length'], 0.5*normalized_camera['size'][1]],
        [0, 0, 1],
    ],
    dtype=np.float64,
)


class Undistorter:

    _map = None
    _previous_parameters = None

    def __call__(self, image, camera_matrix, distortion, is_gazecapture=False):
        h, w, _ = image.shape
        all_parameters = np.concatenate([camera_matrix.flatten(),
                                         distortion.flatten(),
                                         [h, w]])
        if (self._previous_parameters is None
                or len(self._previous_parameters) != len(all_parameters)
                or not np.allclose(all_parameters, self._previous_parameters)):
            print('Distortion map parameters updated.')
            self._map = cv.initUndistortRectifyMap(
                camera_matrix, distortion, R=None,
                newCameraMatrix=camera_matrix if is_gazecapture else None,
                size=(w, h), m1type=cv.CV_32FC1)
            print('fx: %.2f, fy: %.2f, cx: %.2f, cy: %.2f' % (
                    camera_matrix[0, 0], camera_matrix[1, 1],
                    camera_matrix[0, 2], camera_matrix[1, 2]))
```

```
            self._previous_parameters = np.copy(all_parameters)

        # Apply
        return cv.remap(image, self._map[0], self._map[1], cv.INTER_LINEAR)


undistort = Undistorter()


def draw_gaze(image_in, eye_pos, pitchyaw, length=40.0, thickness=2,
              color=(0, 0, 255)):
    """Draw gaze angle on given image with a given eye positions."""
    image_out = image_in
    if len(image_out.shape) == 2 or image_out.shape[2] == 1:
        image_out = cv.cvtColor(image_out, cv.COLOR_GRAY2BGR)
    dx = -length * np.sin(pitchyaw[1])
    dy = -length * np.sin(pitchyaw[0])
    cv.arrowedLine(image_out, tuple(np.round(eye_pos).astype(np.int32)),
                   tuple(np.round([eye_pos[0] + dx,
                                   eye_pos[1] + dy]).astype(int)), color,
                   thickness, cv.LINE_AA, tipLength=0.2)
    return image_out


def vector_to_pitchyaw(vectors):
    """Convert given gaze vectors to yaw (theta) and pitch (phi) angles."""
    n = vectors.shape[0]
    out = np.empty((n, 2))
    vectors = np.divide(vectors, np.linalg.norm(vectors, axis=1).reshape(n, 1))
    out[:, 0] = np.arcsin(vectors[:, 1])  # theta
    out[:, 1] = np.arctan2(vectors[:, 0], vectors[:, 2])  # phi
    return out


def data_normalization(dataset_name, dataset_path, group, output_path):

    # Prepare methods to organize per-entry outputs
    to_write = {}
    def add(key, value):  # noqa
        if key not in to_write:
            to_write[key] = [value]
        else:
            to_write[key].append(value)

    # Iterate through group (person_id)
```

```
    num_entries = next(iter(group.values())).shape[0]
    for i in range(num_entries):
        # Perform data normalization
        processed_entry = data_normalization_entry(dataset_name, dataset_path,
                                                   group, i)

        # Gather all of the person's data
        add('pixels', processed_entry['patch'])
        add('labels', np.concatenate([
            processed_entry['normalized_gaze_direction'],
            processed_entry['normalized_head_pose'],
        ]))

    if len(to_write) == 0:
        return

    # Cast to numpy arrays
    for key, values in to_write.items():
        to_write[key] = np.asarray(values)
        print('%s: ' % key, to_write[key].shape)

    # Write to HDF
    with h5py.File(output_path,
                   'a' if os.path.isfile(output_path) else 'w') as f:
        if person_id in f:
            del f[person_id]
        group = f.create_group(person_id)
        for key, values in to_write.items():
            group.create_dataset(
                key, data=values,
                chunks=(
                    tuple([1] + list(values.shape[1:]))
                    if isinstance(values, np.ndarray)
                    else None
                ),
                compression='lzf',
            )


def data_normalization_entry(dataset_name, dataset_path, group, i):

    # Form original camera matrix
    fx, fy, cx, cy = group['camera_parameters'][i, :]
    camera_matrix = np.array([[fx, 0, cx], [0, fy, cy], [0, 0, 1]],
                             dtype=np.float64)
```

```
# Grab image
image_path = '%s/%s' % (dataset_path,
                        group['file_name'][i].decode('utf-8'))
image = cv.imread(image_path, cv.IMREAD_COLOR)
image = undistort(image, camera_matrix,
                  group['distortion_parameters'][i, :],
                  is_gazecapture=(dataset_name == 'GazeCapture'))
image = image[:, :, ::-1]  # BGR to RGB

# Calculate rotation matrix and euler angles
rvec = group['head_pose'][i, :3].reshape(3, 1)
tvec = group['head_pose'][i, 3:].reshape(3, 1)
rotate_mat, _ = cv.Rodrigues(rvec)

# Take mean face model landmarks and get transformed 3D positions
landmarks_3d = np.matmul(rotate_mat, face_model_3d_coordinates.T).T
landmarks_3d += tvec.T

# Gaze-origin (g_o) and target (g_t)
g_o = np.mean(landmarks_3d[10:12, :], axis=0)  # between 2 eyes
g_o = g_o.reshape(3, 1)
g_t = group['3d_gaze_target'][i, :].reshape(3, 1)
g = g_t - g_o
g /= np.linalg.norm(g)

# Code below is an adaptation of code by Xucong Zhang
# https://www.mpi-inf.mpg.de/departments/computer-vision-and-multimodal-compu

# actual distance between gaze origin and original camera
distance = np.linalg.norm(g_o)
z_scale = normalized_camera['distance'] / distance
S = np.eye(3, dtype=np.float64)
S[2, 2] = z_scale

hRx = rotate_mat[:, 0]
forward = (g_o / distance).reshape(3)
down = np.cross(forward, hRx)
down /= np.linalg.norm(down)
right = np.cross(down, forward)
right /= np.linalg.norm(right)
R = np.c_[right, down, forward].T  # rotation matrix R

# transformation matrix
W = np.dot(np.dot(norm_camera_matrix, S),
```

```
                np.dot(R, np.linalg.inv(camera_matrix)))

    ow, oh = normalized_camera['size']
    patch = cv.warpPerspective(image, W, (ow, oh))  # image normalization

    R = np.asmatrix(R)

    # Correct head pose
    h = np.array([np.arcsin(rotate_mat[1, 2]),
                  np.arctan2(rotate_mat[0, 2], rotate_mat[2, 2])])
    head_mat = R * rotate_mat
    n_h = np.array([np.arcsin(head_mat[1, 2]),
                    np.arctan2(head_mat[0, 2], head_mat[2, 2])])

    # Correct gaze
    n_g = R * g
    n_g /= np.linalg.norm(n_g)
    n_g = vector_to_pitchyaw(-n_g.T).flatten()

    # Basic visualization for debugging purposes
    if i % 50 == 0:
        to_visualize = cv.equalizeHist(cv.cvtColor(patch, cv.COLOR_RGB2GRAY))
        to_visualize = draw_gaze(to_visualize, (0.5 * ow, 0.25 * oh), n_g,
                                 length=80.0, thickness=1)
        to_visualize = draw_gaze(to_visualize, (0.5 * ow, 0.75 * oh), n_h,
                                 length=40.0, thickness=3, color=(0, 0, 0))
        to_visualize = draw_gaze(to_visualize, (0.5 * ow, 0.75 * oh), n_h,
                                 length=40.0, thickness=1,
                                 color=(255, 255, 255))
        cv.imshow('normalized_patch', to_visualize)
        cv.waitKey(1)

    return {
        'patch': patch.astype(np.uint8),
        'gaze_direction': g.astype(np.float32),
        'gaze_origin': g_o.astype(np.float32),
        'gaze_target': g_t.astype(np.float32),
        'head_pose': h.astype(np.float32),
        'normalization_matrix': np.transpose(R).astype(np.float32),
        'normalized_gaze_direction': n_g.astype(np.float32),
        'normalized_head_pose': n_h.astype(np.float32),
    }


if __name__ == '__main__':
```

```
# Grab SFM coordinates and store
face_model_fpath = './sfm_face_coordinates.npy'
face_model_3d_coordinates = np.load(face_model_fpath)

# Preprocess some datasets
output_dir = './outputs/'
if not os.path.isdir(output_dir):
    os.makedirs(output_dir)
datasets = {
    'MPIIGaze': {
        # Path to the MPIIFaceGaze dataset
        # Sub-folders names should consist of person IDs, for example:
        # p00, p01, p02, ...
        'input-path': './MPIIFaceGaze',

        # A supplementary HDF file with preprocessing data,
        # as provided by us. See grab_prerequisites.bash
        'supplementary': './MPIIFaceGaze_supplementary.h5',

        # Desired output path for the produced HDF
        'output-path': output_dir + '/MPIIGaze.h5',
    },
    'GazeCapture': {
        # Path to the GazeCapture dataset
        # Sub-folders names should consist of person IDs, for example:
        # 00002, 00028, 00141, ...
        'input-path': '/media/wookie/WookExt4/datasets/GazeCapture',

        # A supplementary HDF file with preprocessing data,
        # as provided by us. See grab_prerequisites.bash
        'supplementary': './GazeCapture_supplementary.h5',

        # Desired output path for the produced HDF
        'output-path': output_dir + '/GazeCapture.h5',
    },
}
for dataset_name, dataset_spec in datasets.items():
    # Perform the data normalization
    with h5py.File(dataset_spec['supplementary'], 'r') as f:
        for person_id, group in f.items():
            print('')
            print('Processing %s/%s' % (dataset_name, person_id))
            data_normalization(dataset_name,
                               dataset_spec['input-path'],
                               group,
```

```
                                        dataset_spec['output-path'])
```

Once the normalized rotation matrix has been created from the gaze origin, gaze target and virtual camera points, the main script is called. This file takes in the previously created hdf file for both MPIIGaze and GazeCapture as the input and therefore is used for meta-learning.

full_train_test_and_plot.bash

```
#!/bin/bash

############################
# Necessary Configurations

# We skip DT-ED training by default, such that the pre-trained weights
# can be used as-is. Please make sure that you have followed the README.md
# instructions to acquire these weights.
#
# Note: This is different to the `--skip-training` argument to
#       `1_train_dt_ed.py` in that it skips the script completely.
#
# Set to 0 to perform DT-ED training and inference for the HDF output.
SKIP_DTED_TRAINING=0

# NOTE: please make sure to update the two paths below as necessary.
MPIIGAZE_FILE="../preprocess/outputs/MPIIGaze.h5
GAZECAPTURE_FILE="../preprocess/outputs/GazeCapture.h5

# This batch size should fit a 11GB single GPU
# The original training used 8x Tesla V100 GPUs.
BATCH_SIZE=64

# Set the experiment output directory.
# NOTE: make sure to change this if you do not intend to over-write
#       previous outputs.
OUTPUT_DIR="outputs_of_full_train_test_and_plot"


if [[ $SKIP_DTED_TRAINING -eq 0 ]]
then
        ############################
        # 1. Perform DT-ED training
        #
        #    The original setup used here was with:
        #    > Batch size:  1536
        #    > # of epochs: 20
```

```
        #     > # of GPUs:    8
        #     > GPU model:    Tesla V100 (32GB)
        #     > Mixed precision training with apex -O1

    TRAIN_CMD=""
        TRAIN_CMD="1_train_dt_ed.py \
                --mpiigaze-file ${MPIIGAZE_FILE} \
                --gazecapture-file ${GAZECAPTURE_FILE} \
                \
                --num-training-epochs 20 \
                --batch-size $BATCH_SIZE \
                --eval-batch-size 1024 \
                \
                --normalize-3d-codes \
                --embedding-consistency-loss-type angular \
                --backprop-gaze-to-encoder \
                \
                --num-data-loaders 16 \
                \
                --save-image-samples 20 \
                --use-tensorboard \
                --save-path ${OUTPUT_DIR} \
        "
    eval "python3 -m torch.distributed.launch --nproc_per_node=8 $TRAIN_CMD --distr
    eval "python3 $TRAIN_CMD --skip-training --generate-predictions; "



fi

############################
# 2. Perform Meta Learning
#
#    This step processes 6 experiments at a time because a single experiment
#    does not make use of the GPU capacity sufficiently well.
#
#    Please note, that you need output HDF files from the previous step to
#    proceed to the next step. These HDF files are provided to you by default
#    in this specific example pipeline.
#
#    In this example script, we use pre-trained MAML weights that we provided on
#    20th January 2020. This can be done by providing the `--skip-training`
#    command line argument. Note that the full testing procedure is still very
#    time consuming, as 1000-step fine-tuning must occur for each participant
#    in GazeCapture and MPIIGaze.
```

```
ML_COMMON=" --disable-tqdm --output-dir ./"

python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR}  1 &
python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR}  2 &
python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR}  3 &
python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR}  4 &
python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR}  5 &
python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR}  6 &
wait

python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR}  7 &
python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR}  8 &
python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR}  9 &
python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR} 10 &
python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR} 11 &
python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR} 12 &
wait

python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR} 13 &
python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR} 14 &
python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR} 15 &
python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR} 16 &
python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR} 17 &
python3 2_meta_learning.py ${ML_COMMON} ${OUTPUT_DIR} 18 &
wait


########################################################################
# 3. Collect all of the individual meta-learning experiment results

python3 3_combine_maml_results.py ${OUTPUT_DIR}
```

The main script in turn call the meta-learning script as well as the result script to combine the various outputs.

meta_learning.py

```
#!/usr/bin/env python3

# ------------------------------------------------------------
# Copyright (C) 2019 NVIDIA Corporation. All rights reserved.
# NVIDIA Source Code License (1-Way Commercial)
# Code written by Seonwook Park, Shalini De Mello.
# ------------------------------------------------------------
```

```
import argparse
import os
import pickle
import random
from collections import OrderedDict

import h5py
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable as V
from tensorboardX import SummaryWriter
from tqdm import tqdm

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

"""
    Utility functions
"""


def angular_error(a, b):
    """Calculate angular error (via cosine similarity)."""
    a = pitchyaw_to_vector(a) if a.shape[1] == 2 else a
    b = pitchyaw_to_vector(b) if b.shape[1] == 2 else b

    ab = np.sum(np.multiply(a, b), axis=1)
    a_norm = np.linalg.norm(a, axis=1)
    b_norm = np.linalg.norm(b, axis=1)

    # Avoid zero-values (to avoid NaNs)
    a_norm = np.clip(a_norm, a_min=1e-6, a_max=None)
    b_norm = np.clip(b_norm, a_min=1e-6, a_max=None)

    similarity = np.divide(ab, np.multiply(a_norm, b_norm))
    similarity = np.clip(similarity, a_min=-1.0 + 1e-6, a_max=1.0 - 1e-6)

    return np.degrees(np.arccos(similarity))


def nn_angular_error(y, y_hat):
    sim = F.cosine_similarity(y, y_hat, eps=1e-6)
    sim = F.hardtanh(sim, -1.0 + 1e-6, 1.0 - 1e-6)
```

```
    return torch.acos(sim) * (180 / np.pi)


def nn_mean_angular_loss(y, y_hat):
    return torch.mean(nn_angular_error(y, y_hat))


def nn_mean_asimilarity(y, y_hat):
    return torch.mean(1.0 - F.cosine_similarity(y, y_hat, eps=1e-6))


def pitchyaw_to_vector(pitchyaws):
    n = pitchyaws.shape[0]
    sin = np.sin(pitchyaws)
    cos = np.cos(pitchyaws)
    out = np.empty((n, 3))
    out[:, 0] = np.multiply(cos[:, 0], sin[:, 1])
    out[:, 1] = sin[:, 0]
    out[:, 2] = np.multiply(cos[:, 0], cos[:, 1])
    return out


"""
    Tasks class for grabbing training/testing samples
"""


class Tasks(object):
    def __init__(self, hdf_path, x_keys=['z_gaze']):

        # Select tasks for which min. 1000 entries exist
        self.data = h5py.File(hdf_path, 'r')
        previous_len = len(self.data.keys())
        self.selected_tasks = [k for k in self.data.keys()
                               if self.data[k + '/gaze'].len() > 1000]
        self.num_tasks = len(self.selected_tasks)

        # Now load in all data into memory for selected tasks
        self.processed_data = []
        for task in self.selected_tasks:
            num_entries = self.data[task + '/gaze'].len()
            xs = np.concatenate([
                np.array(self.data[task + '/' + key]).reshape(num_entries, -1)
                for key in x_keys
            ], axis=1)
```

```
            ys = pitchyaw_to_vector(np.array(self.data[task + '/gaze']).reshape(-1,
            self.processed_data.append((xs, ys))
        print('Loaded %s (%d -> %d tasks)' % (os.path.basename(hdf_path),
                                              previous_len, self.num_tasks))

        # By default, we just sample disjoint sets from the entire given data
        self.all_indices = [list(range(len(entries[0])))
                            for entries in self.processed_data]
        self.train_indices = self.all_indices
        self.test_indices = self.all_indices

    def create_sample(self, task_index, indices):
        """Create a sample of a task for meta-learning.

        This consists of a x, y pair.
        """
        xs, ys = zip(*[(self.processed_data[task_index][0][i],
                        self.processed_data[task_index][1][i])
                       for i in indices])
        xs, ys = np.array(xs), np.array(ys)
        return (torch.Tensor(xs).to(device),
                torch.Tensor(ys).to(device))

    def sample(self, num_train=4, num_test=100):
        """Yields training and testing samples."""
        picked_task = random.randint(0, self.num_tasks - 1)
        return self.sample_for_task(picked_task, num_train=num_train,
                                    num_test=num_test)

    def sample_for_task(self, task, num_train=4, num_test=100):
        if self.train_indices[task] is self.test_indices[task]:
            # This is for meta-training and meta-validation
            indices = random.sample(self.all_indices[task], num_train + num_test)
            train_indices = indices[:num_train]
            test_indices = indices[-num_test:]
        else:
            # This is for meta-testing
            train_indices = random.sample(self.train_indices[task], num_train)
            test_indices = self.test_indices[task]
        return (self.create_sample(task, train_indices),
                self.create_sample(task, test_indices))


class TestTasks(Tasks):
    """Class for final testing (not testing within meta-learning."""
```

```
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.train_indices = [indices[:-500] for indices in self.all_indices]
        self.test_indices = [indices[-500:] for indices in self.all_indices]


"""
    Replacement classes for standard PyTorch Module and Linear.
"""


class ModifiableModule(nn.Module):
    def params(self):
        return [p for _, p in self.named_params()]

    def named_leaves(self):
        return []

    def named_submodules(self):
        return []

    def named_params(self):
        subparams = []
        for name, mod in self.named_submodules():
            for subname, param in mod.named_params():
                subparams.append((name + '.' + subname, param))
        return self.named_leaves() + subparams

    def set_param(self, name, param, copy=False):
        if '.' in name:
            n = name.split('.')
            module_name = n[0]
            rest = '.'.join(n[1:])
            for name, mod in self.named_submodules():
                if module_name == name:
                    mod.set_param(rest, param, copy=copy)
                    break
        else:
            if copy is True:
                setattr(self, name, V(param.data.clone(), requires_grad=True))
            else:
                assert hasattr(self, name)
                setattr(self, name, param)

    def copy(self, other, same_var=False):
```

```
        for name, param in other.named_params():
            self.set_param(name, param, copy=not same_var)


class GradLinear(ModifiableModule):
    def __init__(self, *args, **kwargs):
        super().__init__()
        ignore = nn.Linear(*args, **kwargs).to(device)

        nn.init.normal_(ignore.weight.data, mean=0.0, std=np.sqrt(1. / args[0]))
        nn.init.constant_(ignore.bias.data, val=0)

        self.weights = V(ignore.weight.data, requires_grad=True).to(device)
        self.bias = V(ignore.bias.data, requires_grad=True).to(device)

    def forward(self, x):
        return F.linear(x, self.weights, self.bias).to(device)

    def named_leaves(self):
        return [('weights', self.weights), ('bias', self.bias)]


"""
    Meta-learnable fully-connected neural network model definition
"""


class GazeEstimationModel(ModifiableModule):
    def __init__(self, activation_type='relu', layer_num_features=[48, 64, 3], make
        super().__init__()
        self.activation_type = activation_type

        # Construct layers
        self.layer_num_features = layer_num_features
        self.layers = []
        for i, f_now in enumerate(self.layer_num_features[:-1]):
            f_next = self.layer_num_features[i + 1]
            layer = GradLinear(f_now, f_next)
            self.layers.append(('layer%02d' % (i + 1), layer))

        # For use with Meta-SGD
        self.alphas = []
        if make_alpha:
            for i, f_now in enumerate(self.layer_num_features[:-1]):
                f_next = self.layer_num_features[i + 1]
```

```
                alphas = GradLinear(f_now, f_next)
                alphas.weights.data.uniform_(0.005, 0.1)
                alphas.bias.data.uniform_(0.005, 0.1)
                self.alphas.append(('alpha%02d' % (i + 1), alphas))

    def clone(self, make_alpha=None):
        if make_alpha is None:
            make_alpha = (self.alphas is not None and len(self.alphas) > 0)
        new_model = self.__class__(self.activation_type, self.layer_num_features,
                                   make_alpha=make_alpha)
        new_model.copy(self)
        return new_model

    def state_dict(self):
        output = {}
        for key, layer in self.layers:
            output[key + '.weights'] = layer.weights.data
            output[key + '.bias'] = layer.bias.data
        return output

    def load_state_dict(self, weights):
        for key, tensor in weights.items():
            self.set_param(key, tensor, copy=True)

    def forward(self, x):
        for name, layer in self.layers[:-1]:
            x = layer(x)
            if self.activation_type == 'relu':
                x = F.relu_(x)
            elif self.activation_type == 'leaky_relu':
                x = F.leaky_relu_(x)
            elif self.activation_type == 'elu':
                x = F.elu_(x)
            elif self.activation_type == 'selu':
                x = F.selu_(x)
            elif self.activation_type == 'tanh':
                x = torch.tanh_(x)
            elif self.activation_type == 'sigmoid':
                x = torch.sigmoid_(x)
            elif self.activation_type == 'none':
                pass
            else:
                raise ValueError('Unknown activation function "%s"' % self.activati
        x = self.layers[-1][1](x)  # No activation on output of last layer
        x = F.normalize(x, dim=-1)  # Normalize
```

```
        return x

    def named_submodules(self):
        return self.layers + self.alphas


class GazeEstimationModelPreExtended(ModifiableModule):
    def __init__(self):
        super().__init__()

        # Construct layers
        self.layer00 = GradLinear(640, 118)  # 64 + 2*3 + 16*3
        self.layer01 = GradLinear(118, 64)
        self.layer02 = GradLinear(64, 3)
        self.layers = [('layer00', self.layer00),
                       ('layer01', self.layer01),
                       ('layer02', self.layer02)]

    def clone(self, make_alpha=None):
        new_model = self.__class__()
        new_model.copy(self)
        return new_model

    def forward(self, x):
        x = self.layer00(x)
        x = x[:, 64:70]  # Extract at hardcoded z_gaze indices
        x = F.selu_(x)
        x = self.layer01(x)
        x = F.selu_(x)
        x = self.layer02(x)
        x = F.normalize(x, dim=-1)  # Normalize
        return x

    def named_submodules(self):
        return self.layers


"""
    Meta-learning utility functions.
"""


def forward_and_backward(model, data, optim=None, create_graph=False,
                         train_data=None, loss_function=nn_mean_angular_loss):
    model.train()
```

```
    if optim is not None:
        optim.zero_grad()
    loss = forward(model, data, train_data=train_data, for_backward=True,
                   loss_function=loss_function)
    loss.backward(create_graph=create_graph, retain_graph=(optim is None))
    if optim is not None:
        # nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optim.step()
    return loss.data.cpu().numpy()


def forward(model, data, return_predictions=False, train_data=None,
            for_backward=False, loss_function=nn_mean_angular_loss):
    model.train()
    x, y = data
    y_hat = model(V(x))
    loss = loss_function(y_hat, V(y))
    if return_predictions:
        return y_hat.data.cpu().numpy()
    elif for_backward:
        return loss
    else:
        return loss.data.cpu().numpy()


"""
    Inference through model (with/without gradient calculation)
"""


class MAML(object):
    def __init__(self, model, k, output_dir='./outputs/',
                 train_tasks=None, valid_tasks=None, no_tensorboard=False):
        self.model = model
        self.meta_model = model.clone()

        self.train_tasks = train_tasks
        self.valid_tasks = valid_tasks
        self.k = k

        self.output_dir = None
        self.tensorboard = None
        if output_dir is not None:
            self.output_dir = '%s/%s_%02d' % (output_dir, self.__class__.__name__,
            if not os.path.isdir(self.output_dir):
```

```
            os.makedirs(self.output_dir)

        if not no_tensorboard:
            self.tensorboard = SummaryWriter(self.output_dir)

@property
def model_parameters_path(self):
    return '%s/meta_learned_parameters.pth.tar' % self.output_dir

def save_model_parameters(self):
    if self.output_dir is not None:
        torch.save(self.model.state_dict(), self.model_parameters_path)

def load_model_parameters(self):
    if os.path.isfile(self.model_parameters_path):
        weights = torch.load(self.model_parameters_path)
        self.model.load_state_dict(weights)
        print('> Loaded weights from %s' % self.model_parameters_path)

def train(self, steps_outer, steps_inner=1, lr_inner=0.01, lr_outer=0.001,
          disable_tqdm=False):
    self.lr_inner = lr_inner
    print('\nBeginning meta-learning for k = %d' % self.k)
    print('> Please check tensorboard logs for progress.\n')

    # Outer loop optimizer
    optimizer = torch.optim.Adam(self.model.params(), lr=lr_outer)

    # Model and optimizer for validation
    valid_model = self.model.clone()
    valid_optim = torch.optim.SGD(valid_model.params(), lr=self.lr_inner)

    for i in tqdm(range(steps_outer), disable=disable_tqdm):
        for j in range(steps_inner):
            # Make copy of main model
            self.meta_model.copy(self.model, same_var=True)

            # Get a task
            train_data, test_data = self.train_tasks.sample(num_train=self.k)

            # Run the rest of the inner loop
            task_loss = self.inner_loop(train_data, self.lr_inner)

        # Calculate gradients on a held-out set
        new_task_loss = forward_and_backward(
```

```
                self.meta_model, test_data, train_data=train_data,
            )

            # Update the main model
            optimizer.step()
            optimizer.zero_grad()

            if (i + 1) % 100 == 0:
                # Log to Tensorflow
                if self.tensorboard is not None:
                    self.tensorboard.add_scalar('meta-train/train-loss', task_loss,
                    self.tensorboard.add_scalar('meta-train/valid-loss', new_task_l

                    # Validation
                    losses = []
                    for j in range(self.valid_tasks.num_tasks):
                        valid_model.copy(self.model)
                        train_data, test_data = self.valid_tasks.sample_for_task(j, num
                        train_loss = forward_and_backward(valid_model, train_data, vali
                        valid_loss = forward(valid_model, test_data, train_data=train_d
                        losses.append((train_loss, valid_loss))
                    train_losses, valid_losses = zip(*losses)
                    if self.tensorboard is not None:
                        self.tensorboard.add_scalar('meta-valid/train-loss', np.mean(tr
                        self.tensorboard.add_scalar('meta-valid/valid-loss', np.mean(va

        # Save MAML initial parameters
        self.save_model_parameters()

    def test(self, test_tasks_list, num_iterations=[1, 5, 10], num_repeats=20):
        print('\nBeginning testing for meta-learned model with k = %d\n' % self.k)
        model = self.model.clone()

        # IMPORTANT
        #
        # Sets consistent seed such that as long as --num-test-repeats is the
        # same, experiment results from multiple invocations of this script can
        # yield the same calibration samples.
        random.seed(4089213955)

        for test_set_name, test_tasks in test_tasks_list.items():
            predictions = OrderedDict()
            losses = OrderedDict([(n, []) for n in num_iterations])
            for i, task_name in enumerate(test_tasks.selected_tasks):
                predictions[task_name] = []
```

```
            for t in range(num_repeats):
                model.copy(self.model)
                optim = torch.optim.SGD(model.params(), lr=self.lr_inner)

                train_data, test_data = test_tasks.sample_for_task(i, num_train
                if num_iterations[0] == 0:
                    train_loss = forward(model, train_data)
                    test_loss = forward(model, test_data, train_data=train_data
                    losses[0].append((train_loss, test_loss))
                for j in range(np.amax(num_iterations)):
                    train_loss = forward_and_backward(model, train_data, optim)
                    if (j + 1) in num_iterations:
                        test_loss = forward(model, test_data, train_data=train_
                        losses[j + 1].append((train_loss, test_loss))

                # Register ground truth and prediction
                predictions[task_name].append({
                    'groundtruth': test_data[1].cpu().numpy(),
                    'predictions': forward(model, test_data,
                                           return_predictions=True,
                                           train_data=train_data),
                })
                predictions[task_name][-1]['errors'] = angular_error(
                    predictions[task_name][-1]['groundtruth'],
                    predictions[task_name][-1]['predictions'],
                )

            print('Done for k = %3d, %s/%s... train: %.3f, test: %.3f' % (
                self.k, test_set_name, task_name,
                np.mean([both[0] for both in losses[num_iterations[-1]][-num_re
                np.mean([both[1] for both in losses[num_iterations[-1]][-num_re
            ))

        if self.output_dir is not None:
            # Save predictions to file
            pkl_path = '%s/predictions_%s.pkl' % (self.output_dir, test_set_nam
            with open(pkl_path, 'wb') as f:
                pickle.dump(predictions, f)

            # Finally, log values to tensorboard
            if self.tensorboard is not None:
                for n, v in losses.items():
                    train_losses, test_losses = zip(*v)
                    stem = 'meta-test/%s/' % test_set_name
                    self.tensorboard.add_scalar(stem + 'train-loss', np.mean(tr
```

```
                        self.tensorboard.add_scalar(stem + 'valid-loss', np.mean(te

                # Write loss values as plain text too
                np.savetxt('%s/losses_%s_train.txt' % (self.output_dir, test_set_na
                           [[n, np.mean(list(zip(*v))[0])] for n, v in losses.items
                np.savetxt('%s/losses_%s_valid.txt' % (self.output_dir, test_set_na
                           [[n, np.mean(list(zip(*v))[1])] for n, v in losses.items

            out_msg = '> Completed test on %s for k = %d' % (test_set_name, self.k)
            final_n = sorted(num_iterations)[-1]
            final_train_losses, final_test_losses = zip(*(losses[final_n]))
            out_msg += ('\n  at %d steps losses were... train: %.3f, test: %.3f +/-
                        (final_n, np.mean(final_train_losses),
                         np.mean(final_test_losses),
                         np.mean([
                             np.std([
                                 data['errors'] for data in person_data
                             ], axis=0)
                             for person_data in predictions.values()
                         ])))
            print(out_msg)

    def inner_loop(self, train_data, lr_inner=0.01):
        # Forward-pass and calculate gradients on meta model
        loss = forward_and_backward(self.meta_model, train_data,
                                    create_graph=True)

        # Apply gradients
        for name, param in self.meta_model.named_params():
            self.meta_model.set_param(name, param - lr_inner * param.grad)
        return loss


class FOMAML(MAML):
    def inner_loop(self, train_data, lr_inner=0.01):
        # Forward-pass and calculate gradients on meta model
        loss = forward_and_backward(self.meta_model, train_data)

        # Apply gradients
        for name, param in self.meta_model.named_params():
            grad = V(param.grad.detach().data)
            self.meta_model.set_param(name, param - lr_inner * grad)
        return loss
```

```
class MetaSGD(MAML):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.model = self.model.clone(make_alpha=True)
        self.meta_model = self.model.clone()

    def inner_loop(self, train_data, lr_inner=0.01):
        # Forward-pass and calculate gradients on meta model
        loss = forward_and_backward(self.meta_model, train_data,
                                    create_graph=True)

        # Apply gradients
        named_params = dict(self.meta_model.named_params())
        for name, param in named_params.items():
            if name.startswith('layer'):
                alpha = named_params['alpha' + str(name[5:])]
                self.meta_model.set_param(name, param - lr_inner * alpha * param.gr
        return loss


class NONE(MAML):
    def train(self, steps_outer, steps_inner=1, lr_inner=0.01, lr_outer=0.001,
              disable_tqdm=False):
        self.lr_inner = lr_inner

        # Save randomly initialized MLP parameters
        self.save_model_parameters()


"""
    Actual run script
"""

if __name__ == '__main__':

    # Available meta-learning methods
    meta_learner_classes = {
        'MAML': MAML,
        'FOMAML': FOMAML,
        'Meta-SGD': MetaSGD,
        'NONE': NONE,
    }

    # Define and parse configuration for training and evaluations
    parser = argparse.ArgumentParser(description='Meta-learn gaze estimator from Ro
```

```
parser.add_argument('input_dir', type=str,
                    help='Input directory for experiment data')
parser.add_argument('--output-dir', type=str, default='./',
                    help='Output directory for tensorboard log relative to inpu
parser.add_argument('--no-tensorboard', action='store_true',
                    help='Log training and validation progress to tensorboard.'
parser.add_argument('--disable-tqdm', action='store_true',
                    help='Disable progress bar from tqdm (in particular on NGC)

parser.add_argument('--maml-use-pretrained-mlp', action='store_true',
                    help='Even for MAML, use pre-trained MLP paramters.')

# Gaze estimation neural network configuration
parser.add_argument('--select-z', type=str, default='z_gaze',
                    help='Embeddings/features to select for using as input to M
                         + '(default: z_gaze)')
parser.add_argument('--layer-num-features', type=str, default='64',
                    help='Network configuration, number of FC features delimite
                         + '(default: 64)')
parser.add_argument('--activation', type=str, default='selu',
                    choices=['sigmoid', 'relu', 'leaky_relu', 'elu', 'selu', 't
                    help='Neural network activation function.')

# Parameters for meta-learning
parser.add_argument('--meta-learner', type=str, default='MAML',
                    choices=list(meta_learner_classes.keys()),
                    help='Meta-learning algorithm')
parser.add_argument('--steps-meta-training', type=int, default=100000,
                    help='Number of steps to meta-learn for (default: 100000)')
parser.add_argument('--tasks-per-meta-iteration', type=int, default=5,
                    help='Tasks to evaluate per meta-learning iteration (defaul
parser.add_argument('--lr-inner', type=float, default=1e-5,
                    help='Learning rate for inner loop (for the task) (default:
parser.add_argument('--lr-outer', type=float, default=1e-3,
                    help='Learning rate for outer loop (the meta learner) (defa

# Evaluation
parser.add_argument('--skip-training', action='store_true',
                    help='Skips meta-training')
parser.add_argument('k', type=int,
                    help='Number of calibration samples to use - k as in k-shot
parser.add_argument('--num-test-repeats', type=int, default=10,
                    help='Number of times to repeat drawing of k samples for te
                         + '(default: 10)')
parser.add_argument('--steps-testing', type=int, default=1000,
```

```
                    help='Number of steps to meta-learn for (default: 1000)')

args = parser.parse_args()

# Define data sources (tasks)
x_keys = args.select_z.split(',')
meta_train_tasks = Tasks(args.input_dir + '/gc_train_predictions.h5', x_keys=x_
meta_val_tasks = Tasks(args.input_dir + '/gc_val_predictions.h5', x_keys=x_keys
meta_test_tasks = [
    ('gc', TestTasks(args.input_dir + '/gc_test_predictions.h5', x_keys=x_keys)
    ('mpi', TestTasks(args.input_dir + '/mpi_predictions.h5', x_keys=x_keys)),
]

# Construct output directory path string
output_dir = None
if args.output_dir is not None:
    output_dir = (os.path.realpath(args.input_dir + '/' + args.output_dir)
                  if args.output_dir[0] != '/' else args.output_dir)
    output_dir += '/'
    output_dir += 'Zg'
    output_dir += '_OLR%.0e' % args.lr_outer
    output_dir += '_IN%d' % args.tasks_per_meta_iteration
    output_dir += '_ILR%.0e' % args.lr_inner
    # output_dir += '_OutN%e' % args.steps_meta_training
    output_dir += '_Net%s' % args.layer_num_features.replace(',', '-')

# Get an example entry to design gaze estimation model
sample_train, _ = meta_train_tasks.sample(num_train=1, num_test=0)

# Training configuration
layer_num_features = [int(f) for f in args.layer_num_features.split(',')]
layer_num_features = [sample_train[0].shape[1]] + layer_num_features + [3]
if not args.select_z == 'before_z':
    model = GazeEstimationModel(activation_type=args.activation,
                                layer_num_features=layer_num_features)
else:
    assert args.maml_use_pretrained_mlp is True
    assert args.layer_num_features == '64'
    assert args.activation == 'selu'
    model = GazeEstimationModelPreExtended()
meta_learner_class = meta_learner_classes[args.meta_learner]
meta_learner = meta_learner_class(model, args.k, output_dir,
                                  meta_train_tasks, meta_val_tasks,
                                  no_tensorboard=args.no_tensorboard)
```

```
# If doing fine-tuning... try to load pre-trained MLP weights
if args.meta_learner == 'NONE' or args.maml_use_pretrained_mlp:
    import glob
    checkpoint_path = sorted(
        glob.glob('%s/checkpoints/at_step_*.pth.tar' % args.input_dir)
    )[-1]
    weights = torch.load(checkpoint_path)
    try:
        state_dict = {
            'layer01.weights': weights['module.gaze1.weight'],
            'layer01.bias': weights['module.gaze1.bias'],
            'layer02.weights': weights['module.gaze2.weight'],
            'layer02.bias': weights['module.gaze2.bias'],
        }
        if args.select_z == 'before_z':
            state_dict['layer00.weights'] = weights['module.fc_enc.weight']
            state_dict['layer00.bias'] = weights['module.fc_enc.bias']
    except:  # noqa
        state_dict = {
            'layer01.weights': weights['gaze1.weight'],
            'layer01.bias': weights['gaze1.bias'],
            'layer02.weights': weights['gaze2.weight'],
            'layer02.bias': weights['gaze2.bias'],
        }
        if args.select_z == 'before_z':
            state_dict['layer00.weights'] = weights['fc_enc.weight']
            state_dict['layer00.bias'] = weights['fc_enc.bias']
    for key, values in state_dict.items():
        model.set_param(key, values, copy=True)
    del state_dict
    print('Loaded %s' % checkpoint_path)

if not args.skip_training:
    meta_learner.train(
        steps_outer=args.steps_meta_training,
        steps_inner=args.tasks_per_meta_iteration,
        lr_inner=args.lr_inner,
        lr_outer=args.lr_outer,
        disable_tqdm=args.disable_tqdm,
    )

# Perform test (which entails the repeated training of person-specific models
if args.skip_training:
    meta_learner.load_model_parameters()
meta_learner.lr_inner = args.lr_inner
```

```
    meta_learner.test(
        test_tasks_list=OrderedDict(meta_test_tasks),
        num_iterations=list(np.arange(start=0, stop=args.steps_testing + 1, step=20
        num_repeats=args.num_test_repeats,
    )
```

3_combine_maml_results.py

```
#!/usr/bin/env python3

# ---------------------------------------------------------
# Copyright (C) 2019 NVIDIA Corporation. All rights reserved.
# NVIDIA Source Code License (1-Way Commercial)
# Code written by Seonwook Park, Shalini De Mello.
# ---------------------------------------------------------

import argparse
import os
import pickle
import re
from collections import OrderedDict

import matplotlib.pyplot as plt
import numpy as np
from tensorboardX import SummaryWriter

pickles_to_process = OrderedDict([
    ('GazeCapture (test)', 'predictions_gc.pkl'),
    ('MPIIGaze', 'predictions_mpi.pkl'),
])


def process_dir(input_dir, meta_learner_identifier):
    """Process experiment directory."""
    selected_dirs = OrderedDict()
    candidate_dirs = sorted([
        d for d in os.listdir(input_dir) if os.path.isdir(input_dir + '/' + d)
    ])
    for exp_dir in candidate_dirs:
        maml_dirs = sorted([
            p for p in os.listdir('%s/%s' % (input_dir, exp_dir))
            if re.match(r'^%s_\d{2,4}$' % meta_learner_identifier, p)
        ], key=lambda x: int(x.split('_')[-1]))
        if len(maml_dirs) > 0:
            selected_dirs[exp_dir] = [  # get full paths
                '%s/%s/%s' % (input_dir, exp_dir, p)
```

```
                for p in maml_dirs
            ]

    for exp_dir, maml_dirs in selected_dirs.items():
        for dataset, pkl_fname in pickles_to_process.items():
            data = get_all_data(maml_dirs, fname=pkl_fname)

            output_path = '%s/%s %s %s.pdf' % (input_dir, exp_dir,
                                               meta_learner_identifier, dataset)
            plot_mean_error_with_bars(dataset, data, output_path,
                                      meta_learner_identifier)


def get_all_data(all_dirs, fname):
    """Process individual outputs for different k."""
    all_data = OrderedDict()
    for d in all_dirs:
        k = int(d.split('_')[-1])
        ifpath = '%s/%s' % (d, fname)
        if os.path.isfile(ifpath):
            with open(ifpath, 'rb') as f:
                all_data[k] = pickle.load(f)
        else:
            print('Skipping %s' % ifpath)
    return all_data


def common_post(dataset, output_path):
    plt.title(dataset)
    plt.xlabel('k')
    plt.ylabel('Mean Test Error')

    plt.grid()
    plt.tight_layout()

    plt.savefig(output_path)
    print('> Wrote to %s' % output_path)


def plot_mean_error_with_bars(dataset, data, output_path,
                              meta_learner_identifier):
    """Plot standard deviation of mean errors over trials."""
    # Pick out errors from people into single list
    errors = [
        (
```

```
            k,
            np.concatenate([
                np.concatenate([
                    trial_data['errors'].reshape(-1, 1)
                    for trial_data in person_data
                ], axis=1)
                for person_data in k_data.values()
            ], axis=0),
        )
        for k, k_data in data.items()
    ]

    ks = [k for k, _ in errors]
    ys = [np.mean(y.reshape(-1)) for _, y in errors]
    es = [np.std(np.mean(y, axis=0)) for _, y in errors]
    print('means:  ', ys)
    print('stddev: ', es)

    plt.clf()
    plt.errorbar(ks, ys, yerr=es, fmt='.-', capsize=5)
    common_post(dataset, output_path)

    # Write means to file
    np.savetxt(output_path[:-3] + 'txt', np.vstack([
        np.array(ks).reshape(1, -1),
        np.array(ys).reshape(1, -1),
        np.array(es).reshape(1, -1),
    ]), fmt='%f')

    # Write means to tensorboard
    tensorboard = SummaryWriter(os.path.dirname(output_path))
    for k, e in zip(ks, ys):
        tensorboard.add_scalar(
            'meta-test-final/%s/%s' % (meta_learner_identifier, dataset), e, k)
    tensorboard.close()


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Merge MAML outputs on NGC')
    parser.add_argument('input_dir', type=str,
                        help='Training output directory to source MAML predictions
    parser.add_argument('--meta-learner', type=str, choices=['MAML', 'NONE'],
                        default='MAML', help='Select meta learning output to use')
    args = parser.parse_args()
    process_dir(args.input_dir, args.meta_learner)
```