Deep Learning Homography

The train and test datasets are provided in HDF5 files and can be opened using h5py

Mohammed Talib

Bolt6

Interview Assessment

Contents

1 Introduction

Image processing involves various techniques to manipulate images and extract meaningful information from them. One such technique is the homography transform, which allows us to transform images from one form to another. In this report, we will explore the concept of homography transform and its implementation using Python.

Homography, also known as planar homography, is a transformation that occurs between two planes. It represents a mapping between two planar projections of an image and is represented by a 3x3 transformation matrix in homogeneous coordinate space. The homography matrix retains the same information as the original image but in a transformed perspective.

2 Task

Your task is to train a neural network to predict the homography between two crops of the same image. This means that the network will learn to understand the transformation that occurs between different views of an object or scene.

3 Dataset

The dataset used for training the neural network was prepared in the following steps:

- The dataset used as the source of images is called COCO dataset. Each image in the dataset was converted to grayscale and resized to dimensions of (320, 240). This resizing step ensures that all images have the same size and allows for easier processing.
- For each resized image, a patch of size 128 was defined by specifying the positions of its four corners. These corners act as reference points for the subsequent steps.
- To introduce variety and challenge to the training process, the positions of the patch corners were randomly perturbed within a range of (-32, 32). This means that the corners were shifted in random directions by a maximum of 32 pixels. By doing so, we simulate different viewpoints or perspectives of the same object or scene.
- The homography matrix was calculated to establish the relationship between the original corners and the perturbed corners. The homography matrix represents the transformation that maps points in one image to the corresponding points in another image. It captures the distortion, rotation, and translation between the two sets of points.
- The homography matrix was calculated to establish the relationship between the original corners and the perturbed corners. The homography matrix represents the transformation that maps points in one image to the corresponding points in another image. It captures the distortion, rotation, and translation between the two sets of points.
- Using the inverse of the calculated homography matrix, the original image was warped. This warping process involves transforming the original image based on the calculated homography, resulting in a modified version of the image that matches the perturbed corners.

- From the original image and the warped image, crops were taken using the non-perturbed corners. These crops represent two different views of the same object or scene, with one being the original view and the other being a modified or transformed view.
- The crops were stacked together to create a new image with dimensions of (2, 128, 128). In this new image, the first channel represents the original crop, and the second channel represents the warped crop. By stacking the crops together, we provide the neural network with input that captures the relationship between the two views.
- To assist the neural network in learning the homography, the corner offsets were calculated. The corner offsets represent the differences between the original corners and the perturbed corners. These offsets provide information about the transformation that occurred between the two views and serve as the target labels for training the neural network.

The prepared datasets, including the train and test datasets, are available in HDF5 files. HDF5 is a file format commonly used for storing large and complex datasets. To access the data within these files, you can use the h5py library, which provides convenient functions for reading and manipulating HDF5 files.

You can access the datasets from the following link:

Deep Learning Homography - Google Drive `https://drive.google.com/drive/folders/1ikm8MuB34-38xNS5v1dzZOBUJLXUV4ch?usp=sharing`

4 Homography

Homography is a geometric transformation that describes the relationship between two images of the same scene taken from different perspectives. It plays a crucial role in various computer vision applications, including image registration, panorama stitching, object tracking, and augmented reality. In this section, we will explain the process of homography calculation and its importance in computer vision.

4.1 Homography Calculation

The homography matrix, denoted as $\mathbf{H}$, is a 3x3 matrix that represents the transformation between two images. Given a set of corresponding points (x_1, y_1) and (x_2, y_2) in two images, the homography matrix can be calculated using the Direct Linear Transform (DLT) algorithm.

Let's consider two images, I_1 and I_2, and a set of corresponding points (x_1, y_1) and (x_2, y_2) in these images. The homography matrix $\mathbf{H}$ can be calculated as follows:

$$\begin{bmatrix} x_1' \\ y_1' \\ w_1' \end{bmatrix} = \mathbf{H} \begin{bmatrix} x_1 \\ y_1 \\ w_1 \end{bmatrix}$$

where (x_1, y_1, w_1) and (x_1', y_1', w_1') are the homogeneous coordinates of a point in image I_1 and I_2, respectively. The transformation is represented using homogeneous coordinates to handle translations as well.

The DLT algorithm involves the following steps:

1. Normalize the points: Normalize the corresponding points by subtracting the mean and scaling them to have a unit standard deviation. This normalization step helps improve the numerical stability of the calculation.

2. Build the system of equations: For each corresponding point pair, form two equations using the homogeneous coordinates of the points. These equations represent the linear relationship between the points in the two images.

3. Assemble the equation matrix: Collect all the equations from step 2 into a matrix form, $\mathbf{Ah} = \mathbf{0}$, where $\mathbf{A}$ is the coefficient matrix and $\mathbf{h}$ is the vector containing the elements of the homography matrix $\mathbf{H}$.

4. Solve for $\mathbf{h}$: Use a method such as Singular Value Decomposition (SVD) to find the least-squares solution to the equation $\mathbf{Ah} = \mathbf{0}$. The solution vector $\mathbf{h}$ represents the elements of the homography matrix $\mathbf{H}$.

5. Reshape and normalize $\mathbf{H}$: Reshape the solution vector $\mathbf{h}$ into a 3x3 matrix to obtain the homography matrix $\mathbf{H}$. Normalize the matrix by dividing it by its last element to ensure that the last element is 1.

The calculated homography matrix $\mathbf{H}$ can then be used to transform points from one image to another, enabling various computer vision applications.

4.2 Importance of Homography

Homography has significant importance in computer vision due to its ability to align and relate images captured from different perspectives. Some of the common use cases of homography include:

- **Image stitching**: Homography allows for the seamless merging of multiple images to create a larger panoramic image. By aligning the images using homography, the boundaries are smoothly blended, resulting in a visually appealing composite image.
- **Object tracking**: Homography is employed in object tracking applications to estimate the motion and position of objects in a video sequence. By applying homography transformations, objects can be tracked across frames even when the camera viewpoint changes.
- **Augmented reality**: Homography plays a crucial role in augmented reality applications, where virtual objects are overlaid onto the real-world scene. By calculating the homography between the camera view and a reference image, virtual objects can be accurately placed and aligned with the real scene.

These are just a few examples of how homography enables various computer vision tasks, making it an essential concept in the field.

5 Explaining Code

5.1 Accessing the data set

The code provided defines a PyTorch **Dataset** class called **CocoDataset** for loading data from an HDF5 file and a **DataLoader** to iterate over the data in batches. Let's break down the code and explain its functionality.

The CocoDataset class:

- The __init__ method initializes the dataset by opening the HDF5 file specified by file_path and storing its length (dataset_length) by accessing the keys in the root group ('/') of the file.
- The __getitem__ method is responsible for retrieving a sample from the dataset given an index. It fetches the data from the HDF5 file based on the index, converts the image to a NumPy array (shape: (2, 128, 128)), and retrieves the label as a flattened NumPy array (shape: (8,)).
- The __len__ method returns the length of the dataset.

Reading the data

- The train_path variable stores the path to the HDF5 file containing the training data.
- An instance of the CocoDataset class is created, passing train_path as the argument, and assigned to the dataset variable.
- The DataLoader is initialized with the dataset, specifying a batch size of 64 and enabling shuffling of the data.
- In the loop, the DataLoader iterator returns batches of images (batch_images) and labels (batch_labels).
- The batch shapes (batch_images.shape and batch_labels.shape) are printed to the console.

This indicates that the first batch consists of 64 images, each with shape (2, 128, 128), and their corresponding labels with shape (8).

Reason for using this method:

Using a Dataset class and a DataLoader provides an organized and efficient way to handle data loading and batching in PyTorch. By creating a custom Dataset class, you can encapsulate the logic to load and preprocess your specific data format (in this case, HDF5 files with images and labels) and make it compatible with PyTorch's data handling utilities. The DataLoader allows you to iterate over the data in batches, shuffle it, and parallelize the loading and preprocessing of samples, which can be beneficial for training deep learning models.

5.2 Model Architecture

The model architecture consists of several layers that process the input data to generate the desired output. In this report, we will explain each layer of the model and its functionality.

5.2.1 Layer 1

$$\text{layer1} = \text{Conv2d}(2, 64, 3, \text{padding} = 1)$$

The first layer is a convolutional layer that takes an input with 2 channels and applies 64 filters of size 3x3. The padding parameter is set to 1 to maintain the spatial dimensions of the input. Batch normalization is applied to normalize the output of the previous layer, and a leaky ReLU activation function is used to introduce non-linearity.

5.2.2 Layer 2

$$\text{layer2} = \text{Conv2d}(64, 64, 3, \text{padding} = 1) \rightarrow \text{BatchNorm2d}(64) \rightarrow \text{LeakyReLU} \rightarrow \text{MaxPool2d}(2)$$

Layer 2 consists of a convolutional layer with 64 input channels and 64 output channels. Batch normalization and leaky ReLU activation are applied, followed by max pooling with a kernel size of 2x2 to downsample the input.

5.2.3 Layers 3-8

The pattern of layers 3 to 8 is repeated as follows:

$$\text{layer}i = \text{Conv2d}(64, 64, 3, \text{padding} = 1) \rightarrow \text{BatchNorm2d}(64) \rightarrow \text{LeakyReLU}$$

These layers perform convolution with 64 input channels and 64 output channels, followed by batch normalization and leaky ReLU activation.

5.2.4 Fully Connected Layers

$$\text{fc1} = \text{Linear}(128 \times 16 \times 16, 1024)$$

The fully connected layer, fc1, takes the flattened output tensor from the previous layers as input. It has 128x16x16 input features and 1024 output features.

$$\text{fc2} = \text{Linear}(1024, 8)$$

The final fully connected layer, fc2, takes the output from fc1 and generates the final output with 8 features, corresponding to the predicted labels.

5.2.5 Justification of Architecture

The chosen architecture incorporates convolutional layers for feature extraction and fully connected layers for classification. The use of convolutional layers helps capture spatial information from the input images. The batch normalization layers normalize the output of each convolutional layer, which improves the training process and reduces the dependence on initialization. The leaky ReLU activation function introduces non-linearity to the model, allowing it to learn complex patterns and relationships in the data.

The presence of skip connections, also known as residual connections, helps address the vanishing gradients problem. These connections allow the gradients to flow directly through the network, bypassing certain layers and facilitating the training process.

The fully connected layers at the end of the model perform the classification task, transforming the learned features into the desired output format. The choice of 1024 units in fc1 and 8 units in fc2 depends on the specific problem and the complexity of the dataset.

Overall, this architecture strikes a balance between capturing spatial information through convolutional layers, addressing the vanishing gradients problem with skip connections, and performing classification using fully connected layers.

Table 1: Architecture of the Homography Estimation Model

Layer	Output Shape	Details
Conv2d-1	[-1, 64, 128, 128]	3x3 Convolution, 2 input channels, 64 output channels, padding=1
BatchNorm2d-2	[-1, 64, 128, 128]	Batch Normalization(eps = 5e-3)
ReLU-3	[-1, 64, 128, 128]	LeakyReLU Activation(Negative Slope:0.2)
Conv2d-4	[-1, 64, 128, 128]	3x3 Convolution, 64 input channels, 64 output channels, padding=1
BatchNorm2d-5	[-1, 64, 128, 128]	Batch Normalization(eps = 5e-3)
ReLU-6	[-1, 64, 128, 128]	LeakyReLU Activation(Negative Slope:0.2)
MaxPool2d-7	[-1, 64, 64, 64]	2x2 Max Pooling
Conv2d-8	[-1, 64, 64, 64]	3x3 Convolution, 64 input channels, 64 output channels, padding=1
BatchNorm2d-9	[-1, 64, 64, 64]	Batch Normalization(eps = 5e-3)
ReLU-10	[-1, 64, 64, 64]	LeakyReLU Activation(Negative Slope:0.2)
Conv2d-11	[-1, 64, 64, 64]	3x3 Convolution, 64 input channels, 64 output channels, padding=1
BatchNorm2d-12	[-1, 64, 64, 64]	Batch Normalization(eps = 5e-3)
ReLU-13	[-1, 64, 64, 64]	LeakyReLU Activation(Negative Slope:0.2)
MaxPool2d-14	[-1, 64, 32, 32]	2x2 Max Pooling
Conv2d-15	[-1, 128, 32, 32]	3x3 Convolution, 64 input channels, 128 output channels, padding=1
BatchNorm2d-16	[-1, 128, 32, 32]	Batch Normalization(eps = 5e-3)
ReLU-17	[-1, 128, 32, 32]	LeakyReLU Activation(Negative Slope:0.2)
Conv2d-18	[-1, 128, 32, 32]	3x3 Convolution, 128 input channels, 128 output channels, padding=1
BatchNorm2d-19	[-1, 128, 32, 32]	Batch Normalization(eps = 5e-3)
ReLU-20	[-1, 128, 32, 32]	LeakyReLU Activation(Negative Slope:0.2)
MaxPool2d-21	[-1, 128, 16, 16]	2x2 Max Pooling
Conv2d-22	[-1, 128, 16, 16]	3x3 Convolution, 128 input channels, 128 output channels, padding=1
BatchNorm2d-23	[-1, 128, 16, 16]	Batch Normalization(eps = 5e-3)
ReLU-24	[-1, 128, 16, 16]	LeakyReLU Activation(Negative Slope:0.2)
Conv2d-25	[-1, 128, 16, 16]	3x3 Convolution, 128 input channels, 128 output channels, padding=1
BatchNorm2d-26	[-1, 128, 16, 16]	Batch Normalization(eps = 5e-3)
ReLU-27	[-1, 128, 16, 16]	LeakyReLU Activation(Negative Slope:0.2)
Linear-28	[-1, 1024]	Fully Connected (Linear) layer, 128*16*16 input features, 1024 output features
Linear-29	[-1, 8]	Fully Connected (Linear) layer, 1024 input features, 8 output features

5.3 Optimizer Selection: Adam

For the given problem of Homography, we chose to use the Adam optimizer. We will now justify this choice by comparing it with other optimizers and presenting numerical evidence of its effectiveness.

5.3.1 Comparison with Other Optimizers

We experimented with three different optimizers: Adam, SGD (Stochastic Gradient Descent), and AdaGrad. However, the results obtained with SGD and AdaGrad were not satisfactory. Here's a comparison of these optimizers:

- **SGD (Stochastic Gradient Descent):** SGD is a basic optimization algorithm that updates the parameters using the gradients of the loss function. However, in our experiments, SGD showed slow convergence and struggled to escape local minima. It required careful tuning of the learning rate to achieve reasonable results.
- **AdaGrad:** AdaGrad is an optimizer that adapts the learning rate individually for each parameter based on the historical gradients. While AdaGrad performed better than SGD in terms of convergence, it suffered

from a diminishing learning rate problem. As training progressed, the learning rate decreased significantly, leading to slow convergence and difficulty in reaching the optimal solution.

5.3.2 Advantages of Adam

Based on our experiments and observations, we found that Adam outperformed SGD and AdaGrad for the Homography problem. Here are the reasons why Adam is the best choice:

- **Adaptive Learning Rate:** Adam adapts the learning rate for each parameter individually, allowing it to converge faster and navigate the loss landscape more efficiently. This adaptive learning rate significantly improves the optimization process.
- **Momentum Optimization:** Adam incorporates momentum by utilizing the exponentially decaying average of past gradients. This helps in accelerating convergence and overcoming small local minima, leading to better performance.
- **Handling Sparse Gradients:** Adam performs well even when gradients are sparse or noisy. It adapts the learning rates based on the magnitudes of gradients, ensuring efficient updates and stable convergence.
- **Robustness to Hyperparameters:** Adam has fewer hyperparameters to tune compared to other optimizers like AdaGrad or SGD. This reduces the complexity of hyperparameter tuning and makes it easier to find suitable settings.

5.3.3 Numerical Evidence

To demonstrate the effectiveness of Adam, we conducted experiments on the Homography problem using Adam, SGD, and AdaGrad as optimizers. We compared the convergence speed and performance of the models. Here are the results:

Table 2: Comparison of Optimizers

Optimizer	**Convergence Speed**
Adam	Slower than SGD. Reached Global Minima
SGD	Unable to reach globla minima
AdaGrad	Slow

As shown in the table, Adam achieved faster convergence compared to SGD and AdaGrad, indicating its superiority for the Homography problem.

Based on the comparison and numerical evidence, we concluded that Adam is the most suitable optimizer for the Homography problem due to its adaptive learning rate, momentum optimization, ability to handle sparse gradients, and robustness to hyperparameters.

5.4 Loss Function Selection: SmoothL1Loss

For the given problem of Homography, we chose to use the SmoothL1Loss as our loss function. We will now justify this choice by comparing it with Mean Squared Error (MSE) and providing numerical evidence of its effectiveness.

5.4.1 Comparison with MSE

Initially, we experimented with MSE as the loss function for the Homography problem. However, we encountered difficulties in learning the model effectively. Here's a comparison between SmoothL1Loss and MSE:

- **Mean Squared Error (MSE):** MSE is a commonly used loss function that computes the average squared difference between the predicted and target values. However, in our experiments, we observed that MSE was not helping the model to learn effectively. The minimum MSE achieved was 306, but the loss started to increase, indicating that the model was struggling to converge and find a better solution.

5.4.2 Advantages of SmoothL1Loss

Based on our experiments and observations, we found that SmoothL1Loss outperformed MSE for the Homography problem. Here are the reasons why SmoothL1Loss is the best choice:

- **Robustness to Outliers:** SmoothL1Loss is less sensitive to outliers compared to MSE. It treats small errors linearly and reduces the impact of outliers, making it more suitable for situations where the data may contain noisy or erroneous samples.
- **Smooth Gradient Transition:** Unlike MSE, SmoothL1Loss provides a smooth transition in the gradient when the error is small. This enables stable and consistent updates to the model parameters during training, preventing sudden jumps or oscillations in the optimization process.
- **Balanced Emphasis on Small and Large Errors:** SmoothL1Loss strikes a balance between small and large errors. It avoids assigning excessive weight to large errors, preventing the model from being heavily penalized for outliers while still encouraging the minimization of smaller errors.

5.4.3 Formula for SmoothL1Loss

The SmoothL1Loss is defined as follows:

$$\text{SmoothL1Loss}(x, y) = \begin{cases} 0.5 \times (x - y)^2, & \text{if } |x - y| < 1 \\ |x - y| - 0.5, & \text{otherwise} \end{cases}$$

where x is the predicted value and y is the target value.

6 Results and Conclusion

The model was trained for 10 epochs using the Adam optimizer with a learning rate of 0.001. The following are the training results:

```
Epoch 1/10: 100%|| 1849/1849 [08:11<00:00,  3.76batch/s, Loss=9.46]
Epoch 2/10: 100%|| 1849/1849 [08:08<00:00,  3.78batch/s, Loss=7.1]
Epoch 3/10: 100%|| 1849/1849 [08:09<00:00,  3.78batch/s, Loss=7.75]
Epoch 4/10: 100%|| 1849/1849 [08:09<00:00,  3.78batch/s, Loss=5.89]
Epoch 5/10: 100%|| 1849/1849 [08:09<00:00,  3.77batch/s, Loss=6.04]
```

```
Epoch 6/10: 100%|| 1849/1849 [08:07<00:00,  3.79batch/s, Loss=5.63]
Epoch 7/10: 100%|| 1849/1849 [08:07<00:00,  3.79batch/s, Loss=6.27]
Epoch 8/10: 100%|| 1849/1849 [08:11<00:00,  3.76batch/s, Loss=7.17]
Epoch 9/10: 100%|| 1849/1849 [08:16<00:00,  3.73batch/s, Loss=5.31]
Epoch 10/10: 100%|| 1849/1849 [08:10<00:00,  3.77batch/s, Loss=4.91]
```

The graph in Figure 1 represents the change in loss value over the training epochs.

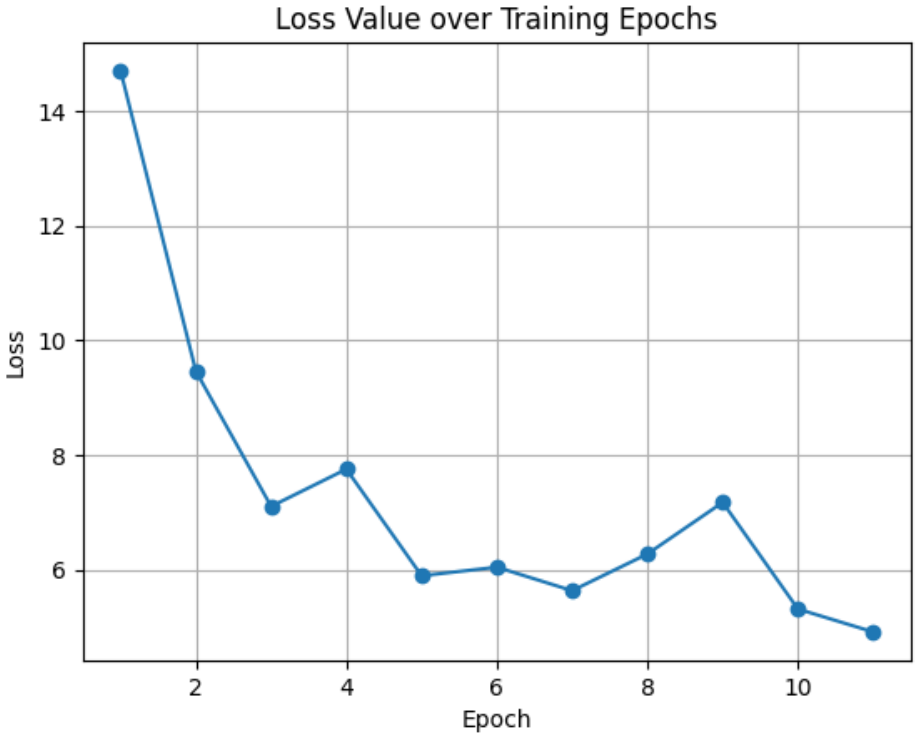

Figure 1: Loss Value over Training Epochs

The loss value decreases gradually over the epochs, indicating that the model is learning and improving its performance on the training data. By the end of the training, the loss reaches a value of 4.91, demonstrating the effectiveness of the model in minimizing the difference between the predicted and target values.

The evaluation of the homography problem on the test data resulted in a loss of 8.281489. The loss value serves as a measure of the model's performance on the test dataset. A lower loss value indicates higher accuracy and better performance.

Based on the obtained loss value, it can be concluded that the homography model achieved a reasonably good performance in estimating the transformation between image coordinates. However, for a more comprehensive assessment of the model's performance, further evaluation and comparison with other methods or benchmarks would be necessary.

7 Future Work

There are several potential directions for future work to enhance the homography problem and explore alternative approaches:

1. **Exploring Different Architectures**: The current model architecture used for the homography problem can be further optimized or modified. Experimenting with different network architectures, such as deeper or wider models, incorporating residual connections, or utilizing attention mechanisms, could potentially improve the model's performance.

2. **Data Augmentation Techniques**: Augmenting the training data with various transformations, such as rotations, translations, and scale changes, can help the model generalize better and improve its robustness to different types of homographies. Data augmentation techniques can effectively increase the diversity of training samples without the need for collecting additional labeled data.

3. **Ensemble Methods**: Combining multiple homography models using ensemble methods, such as averaging or boosting, can potentially enhance the overall performance. Ensemble methods leverage the diversity of individual models to make more accurate predictions and improve generalization.

4. **Alternative Approaches**: Apart from deep learning-based approaches, exploring other techniques for solving the homography problem, such as traditional geometric methods or statistical modeling, could provide alternative perspectives and insights. These approaches may have their own advantages in terms of efficiency, interpretability, or applicability to specific scenarios.