Project Report

Augmented Reality for Inspection

Johannes Merz, Ali Mahdavi-Amiri, Richard Zhang

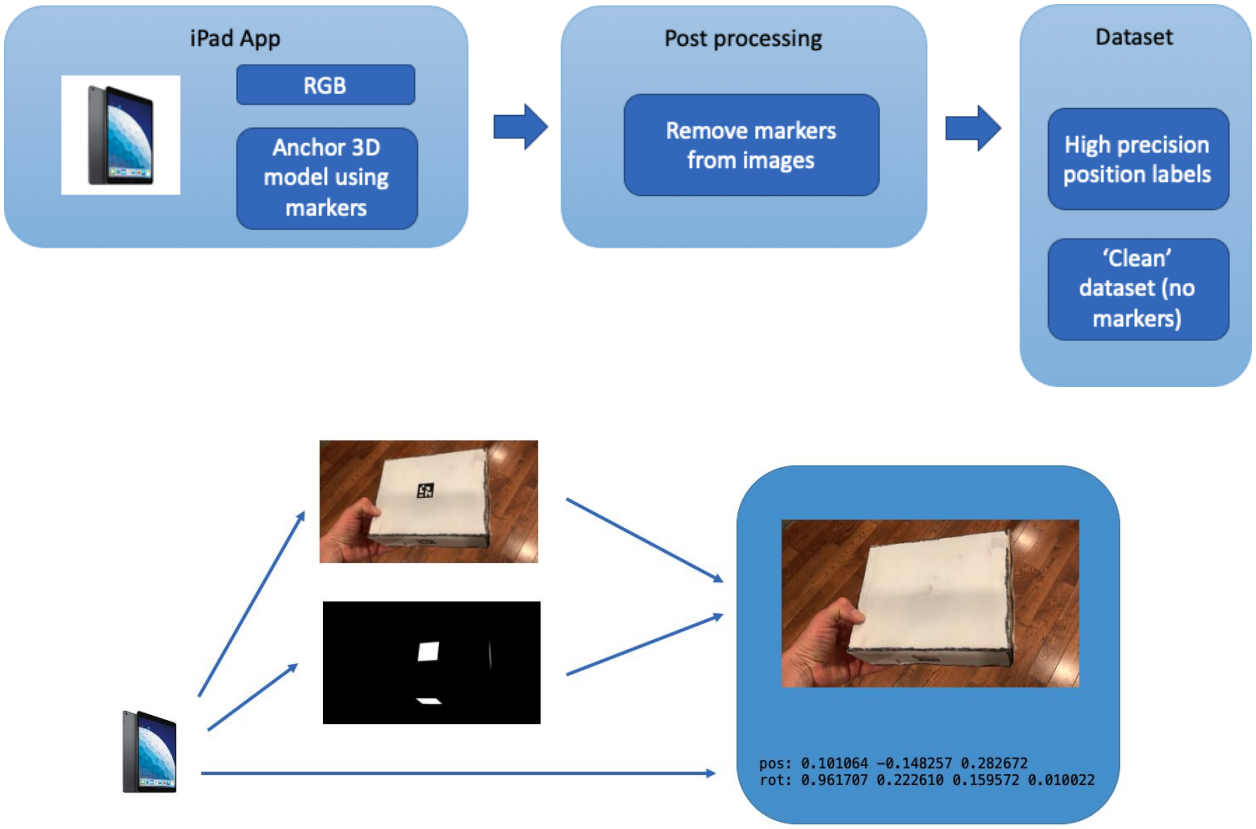
This project is about anchoring a 3D aircraft on a given image using deep learning networks. In the following, we summarize each step of our contribution to this project.

Literature Research

There are two classical computer vision problems that are a dual of each other: pose estimation and camera localization. In pose estimation, the location and coordinate system of the camera is known but the pose (location and orientation) of a given object is needed. In camera localization, the position and orientation of the object is known but the location and orientation of the camera are not known. According to our initial literature research, we have found two related works very relevant to our problem. The first one is PoseNet that is a convolutional neural network trying to find the pose of a given object on a given image. The other work is VidLoc that has LSTM components. LSTM can remember the previous inputs which is a good property when we deal with videos as the information of the previous frames are relevant to the next frames' predictions.

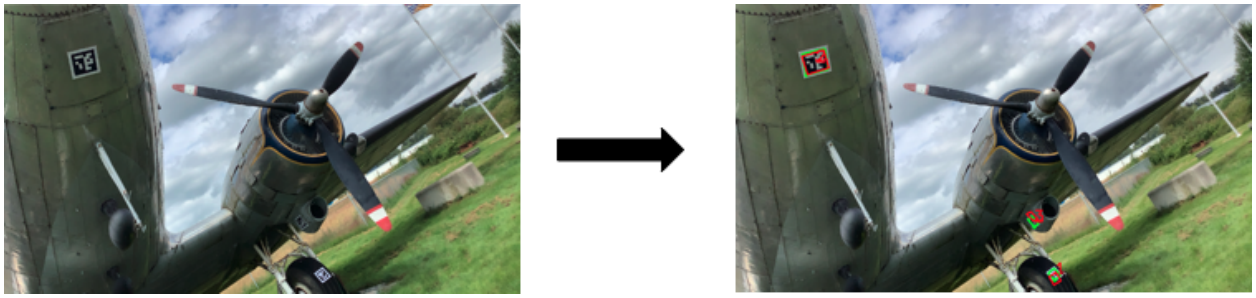
Data Collection

Since having a large clean dataset for training a neural network is a necessity, we started this work by collecting relevant datasets. Unfortunately, due to Covid-19, we did not have access to the Boeing aircraft at BCIT. Therefore, we instead collected our datasets first on a box, then a car, and eventually on a DC3 aircraft available in Canadian Museum of Flight.
To collect datasets, we attached ArUco markers on the body of the aircraft as they can be used in an algorithm called PnP to find the right pose of an object. We used this mechanism to generate our ground truth dataset. These ArUco markers are later removed through an inpainting algorithm (DeepFlow) to make a clean dataset for training. The below figure visualizes the necessary steps during data collection:



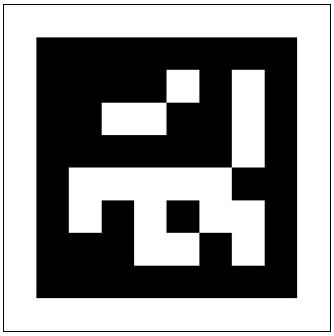
In order to realize the pipeline, different technologies on different platforms were used. The core of the data collection is a Swift and Objective-C++ based iOS app intended to run on an iPad. It uses the iPad camera to gather a video stream and convert individual frames to RGB. Moreover, it collects the per frame internal camera parameters. These parameters describe physical properties of the camera that are essential to estimating 3D world coordinates from 2D positions in the video frames. Traditionally, these values can be estimated in a camera calibration step, however the autofocus that is used by the iPad camera might lead to changing values between frames. Thus, receiving up to date camera parameters for each individual frame prevents possible errors. Unfortunately, we were unaware that this is possible for a long time, because the documentation provided by Apple is less than informative. This proved to be a problem at multiple points in the project. Another issue that we encountered was due to the fact that the video stream coming from the iPad camera was encoded in YCbCr, a color space that separates the luma (brightness) component (Y) of the image from the blue and red chroma components (e.g. Cb = B (Blue color) - Y (luma)). OpenCV however expects RGB as input. The frameworks provided by Apple do not provide any straightforward way to convert these color spaces. Again, Apple's rather compact documentation led to unnecessary delays.
Both the current frame and the camera parameters are then handed to an Objective-C++ submodule that runs an OpenCV based algorithm.

In a first step, the algorithm detects the ArUco markers that are visible in the frame.

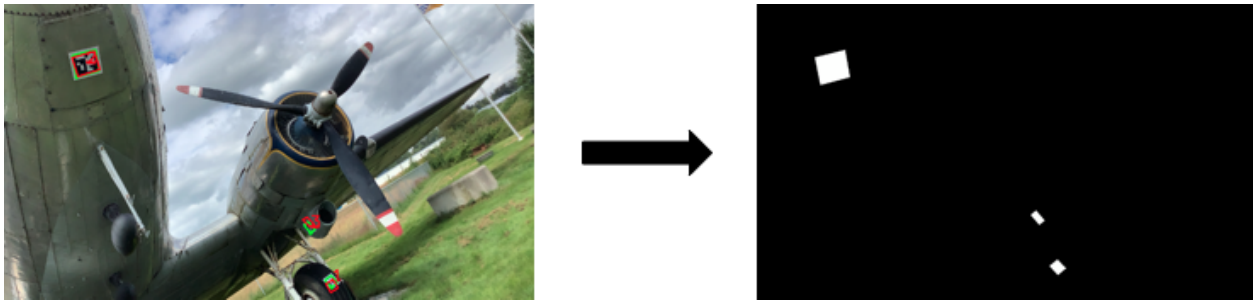


In addition to the above mentioned inputs, the algorithm also receives a list with the actual 3D coordinates of the physical markers in the real world (more on that in the following paragraph). Since each ArUco marker encodes a unique integer ID (similar to how a QR code encodes arbitrary information), the algorithm can now build correspondences between the detected 2D position of the marker in the frame and its real world 3D position. Using an algorithm known as Perspective-n-Point (PnP), we can now estimate the position and rotation of the marker in relation to the camera. If we detect more than one marker at the same time, this algorithm then finds the optimal pose that minimizes the error for all markers. The accuracy of this approach scales with the number of markers that are visible in each frame. Finally, the system stores the frame, together with an xyz-position and a wpqr-quaternion that describes the rotation of the whole airplane.

ArUco markers can easily be generated with simple tools available online in different resolutions and sizes. For our data collection, we generated markers with a resolution of 6x6 blocks and a side length of 150mm. Experiments also showed that adding an additional white border with the size of one block around the markers increases the detection possibility. For easy printing of the markers including the white border, we provided a GIMP project.



Once the system has detected the markers, it also stores this information in a binary mask that has pixel value zero wherever there is no marker, and pixel value one if it is part of a marker in the original frame. These masks are used in the image inpainting step. To generate these masks, the system enlarges the regions of the detected markers, as experiments showed that this improves the performance of the inpainting network (see paragraph on inpainting). The masks are stored together with the original frames.



Once the system is able to detect the pose of the plane, it might still be that it has failed detecting all visible markers. This is not a problem for the pose estimation to generate ground truth data as long as it has detected some of the markers, however the subsequent inpainting algorithm relies on the mask image as a guide to know where markers need to be removed from the image. For this reason, the system now uses the just detected pose and the knowledge of all markers in 3D space to project the positions of not detected markers into the mask image artificially. To make sure that it only projects markers into the mask image that are facing the camera, it checks the orientation of the markers using the cross product of the vector from the camera to the marker and the surface normal of the marker itself.

In a separate module, the same app provides the possibility to visualize all data points and overlay the 3D model of the airplane based on the generated ground truth pose. It then offers the user the possibility to decide, if the pose is correct or not. This module is intended to help clean the collected dataset and make sure that it is of high quality.

All the previously described parts of the algorithm are based on the OpenCV library and the ArUco module. Unfortunately, no precompiled library for iOS that includes the ArUco module is provided by the developers. This meant that we needed to compile OpenCV including the

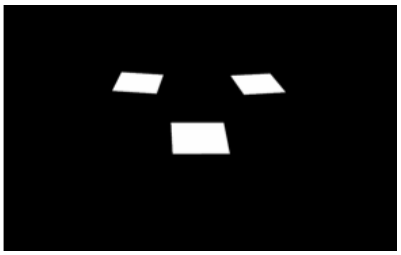
module from source before being able to develop the iPad app. The compiled library can be found under “opencv_aruco/opencv2.framework”.

It was mentioned before that the detection algorithm receives the physical 3D coordinates of the ArUco markers. These could be defined by hand, however this approach is very tedious on any object larger and more complex than a simple box. Thus, another iPad app was used to help in the definition of the positions. This app was based on an ARkit prototype provided by Boeing. In this prototype, a 3D model of a plane could be loaded and visualized. The user was then able to tap on the plane to interact with it. We modified this app so that each tap on the plane defines the position of the center of a marker on the surface of the plane. Using this information and the surface normal of the 3D model of the plane at this position, the four corners of the marker are calculated. The corners are being generated to align the marker horizontally with the ground. Also, each defined marker has a unique ID assigned to it that can also be changed on the fly while placing the markers on the actual plane.

After finalizing all the tools, we went to the Canadian Museum of Flight for a first visit to the DC3. While testing the software, it quickly became apparent that the 3D model of the plane did not align completely with the physical plane. This was due to the age of the suspension of the rear wheel. It was significantly lower than in the 3D model, resulting in a different angle of the plane while standing on the ground. This could be resolved by manually editing the model to fit as best as possible to the physical plane before coming for a second visit.

Another feature that the tool provides is that it helps the user while physically placing the markers. As soon as the system detects a single marker and estimates an initial pose of the 3D model, it uses the 3D knowledge of all other 3D markers to visualize red squares where the user has to place more markers together with the respective IDs. This works similar to how the system adds masks for missed markers, as described before.





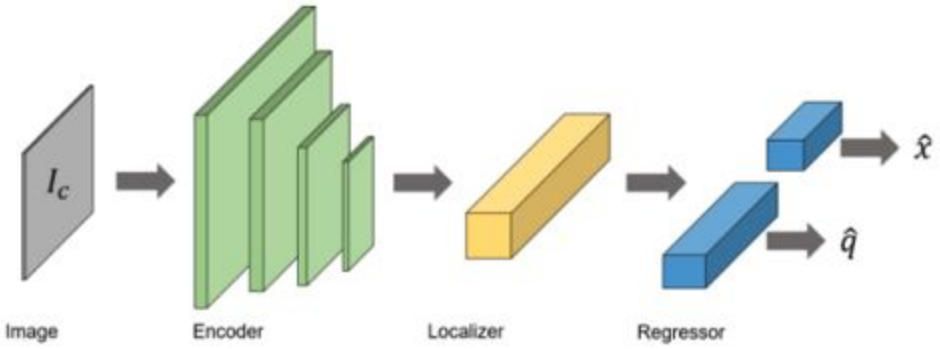
During a second visit, we were able to collect 44 sequences of images. After a cleaning pass over the data, the final dataset consisted of 1981 images and their respective ground truth poses and mask images.

After the data was recorded, it needed to be processed. This was done in two steps. First, the original frames were inpainted using the pretrained DeepFlow neural network. Inpainting is a technique to fill holes in an image by estimating what should be there. Using the mask images that define the positions of the enlarged markers, the inpainting network could remove the

ArUco markers in the images and fill in what “is behind the markers” in the image. The result was a dataset of images of a plane without markers and additional ground truth position and rotation information. The inputs to the network were all the original frames in a resolution of 1920x1080, while the output was of resolution 1280x720 pixels. This was due to GPU memory restrictions. The inpainting was performed on a Titan RTX GPU with 24GB of video memory.

PoseNet

Following the data collection, we could train a preliminary neural network. The initial model was based on the PoseNet architecture. PoseNet is a simple implementation of a neural network that uses a deep CNN to learn features of images. Then it adds fully convolutional layers to regress a 3D vector (xyz position) and a 4D vector (wpqr rotation) from the features. The CNN backend of the network is based on Inception modules with 23 layers.

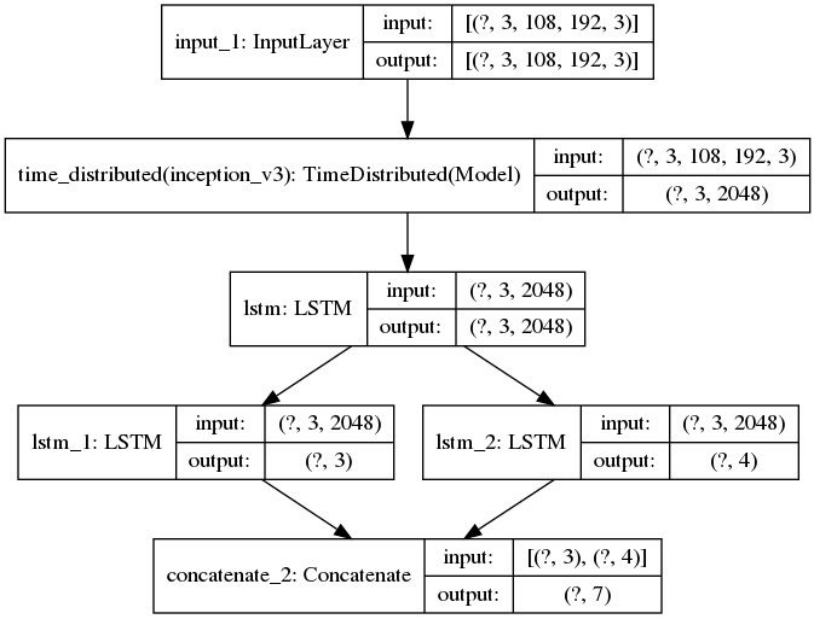


This network does not include temporal information, it analyzes each frame individually. To prepare the training, we split the dataset into a training set (90%) and a testing set (10%), which was set aside and not used during training. The network was trained for 100.000 iterations on the training set. After training, evaluating the model on the testing set showed a median error 0.14546m for the position and 1.411 degrees for the rotation. Visualizing the evaluated poses showed promising results for a preliminary model:

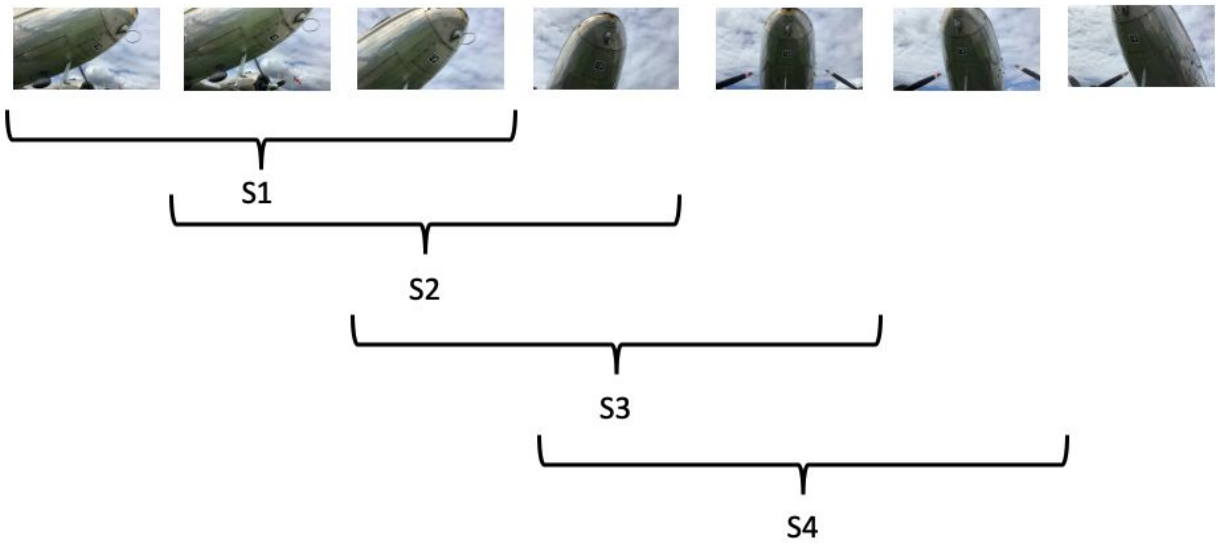
To be able to test the model in a live setting, we now needed to convert the model to CoreML to be able to deploy it on the iPad. Unfortunately this proved to be a challenge. Initially, we used a reference implementation of PoseNet that was implemented in PyTorch. To convert PyTorch to CoreML, it is necessary to first convert the model to the ONNX format and then to CoreML. Many attempts led to a multitude of different errors at different stages of the conversion. We finally decided to port the code to Tensorflow and retrained the model. Luckily, converting the Tensorflow model was less of a challenge and we were able to deploy the model on the iPad. An additional issue that needed to be addressed was the fact that the Vision preprocessing framework on iOS does not provide complete control over how the data is transformed before being fed to the network. This problem was solved on Boeing's side by circumventing the framework and manually processing the data using OpenCV on the iPad.

VidLoc

To be able to leverage the temporal information in the sequences of the dataset, we implemented a modified version of PoseNet, similar to VidLoc. This model is based on the same fundamentals als PoseNet. It uses an Inception based CNN as a backend to extract features of the images followed by a global average pooling step. However this CNN is distributed over the whole length of the sequence, meaning that features will be extracted for each frame in a sequence. Following this sequenced CNN, we now do not use fully connected layers anymore to regress the pose. Instead we use an LSTM module as the bottleneck layer and two more LSTM modules to regress both the position and the rotation. LSTMs are modules in neural networks that are able to remember data from previous timesteps in a sequence and adapt their prediction based on the history of seen data. We used stateless LSTMs, because the data was fed in well defined sequences that were all of the same length.



This was ensured by processing all the input sequences using a sliding window approach. This means that each sequence of the dataset was cut into a number of shorter sequences (length defined by a parameter), where the first frame of the sequence was shifted by a specific number of frames (sampling rate parameter). The below example uses a sequence length of three and a sampling rate of one:



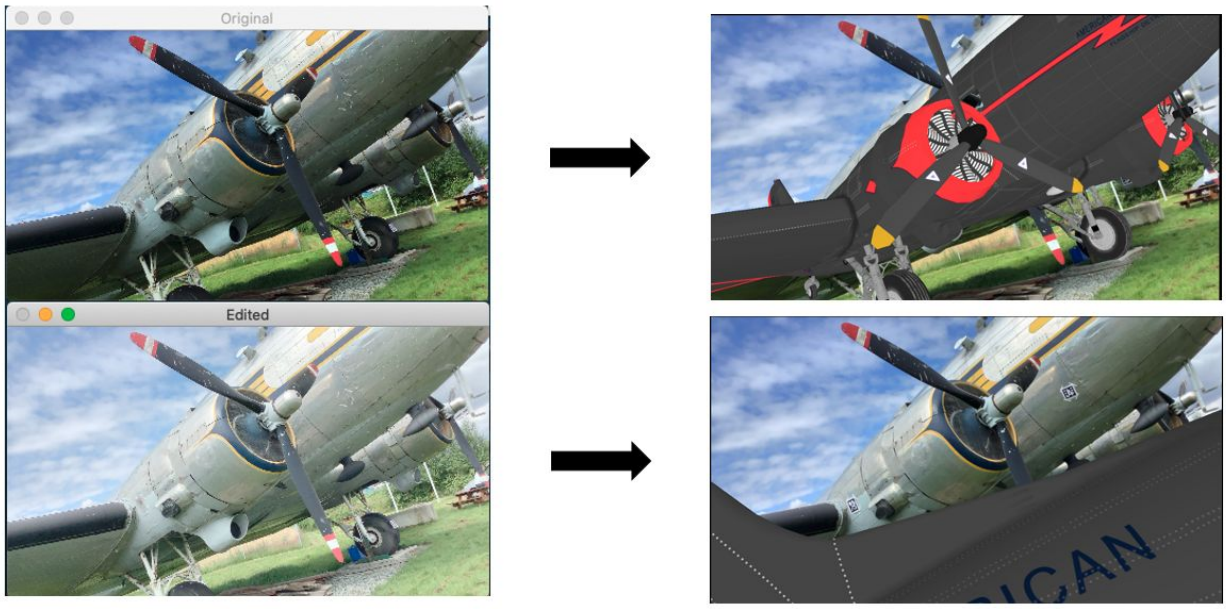
As with PoseNet, we encountered problems while converting the model to CoreML. LSTMs are often represented as cycles to decrease memory usage, which was not supported by the conversion tools in combination with the version of Tensorflow we used (1.15). These issues could be solved by reimplementing the model in Tensorflow 2 and using the unified conversion API provided by coremltools 4.

Training this model is really slow due to the large size of the network. It requires a huge amount of memory and also time. Unfortunately we did not have enough time and processing resources available to fully train this network. However, we made sure to implement it in a way that would automatically distribute the training over all available GPUs in a cluster, to make it easy for Boeing to use their hardware to finish the training.

Data Augmentation

After training, the model (preliminary PoseNet model) provided promising results on the test data, we went for another visit at the DC3 to collect new field test data and test the system on the iPad. Unfortunately the results on the field test data were not satisfactory.

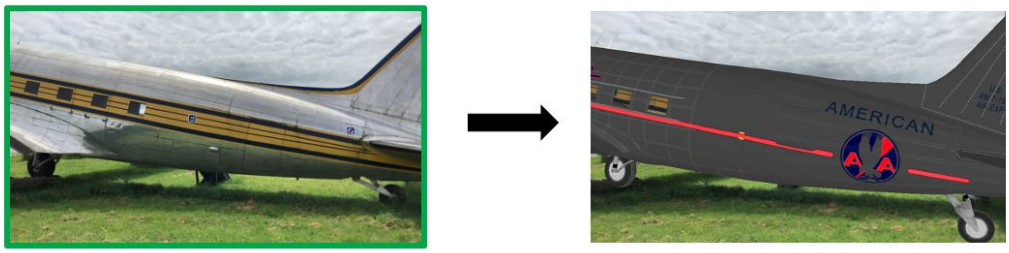
We thus studied the possible causes. After applying preliminary tests, we concluded that the results are dependent on the lighting, and background. To verify the influence of the lighting, we changed the brightness of the initial working test images using gamma correction and tested them on the trained network and the results were completely off.



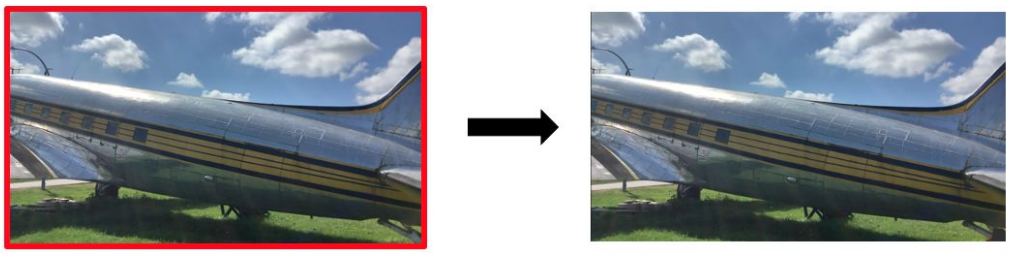
Therefore, we decided to perform a data augmentation to resolve this issue. Gamma correction is a technique to change the perceived brightness of an image by adapting the scaling of the intensities using the formula $I_{out} = I_{in}^{1/\gamma}$. While training the model, we augmented the training images by generating five additional versions of each image with changed brightnesses. The values for γ were randomly sampled between 0.5 and 2.0.

In addition to differences in brightness between the two days of data collection, another difference was the cloudiness of the sky. We tested the influence of this by taking an image of the plane from either day with a similar pose. Evaluating them showed that the pose estimation for the image from the initial data collection worked as expected, while it was not working with the image taken during the field test day.

Test Data

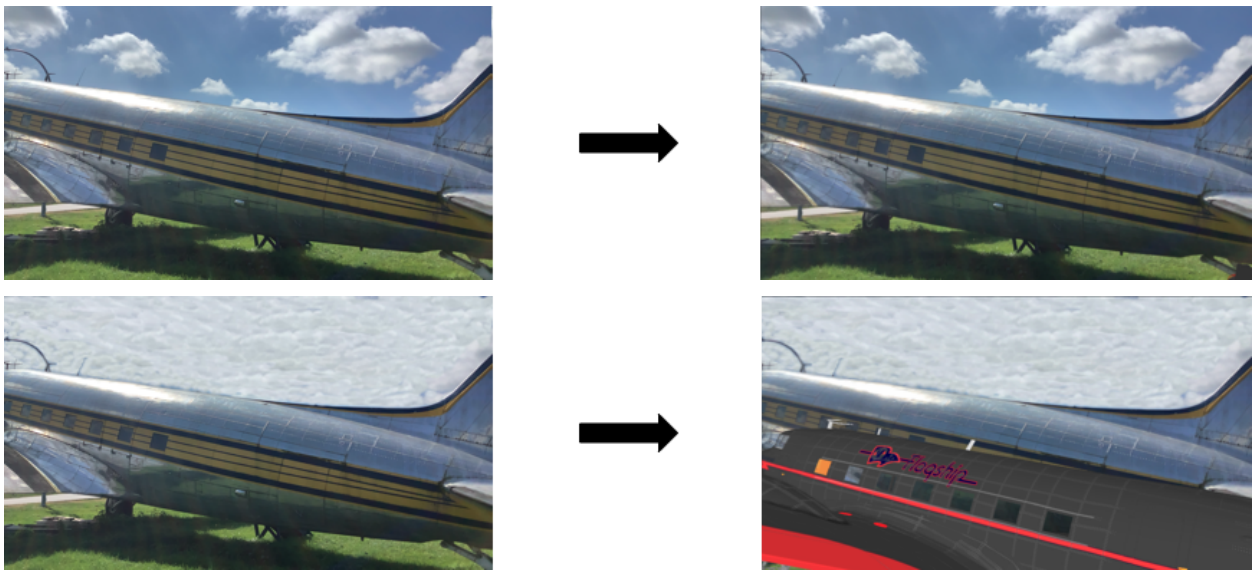


Field Test



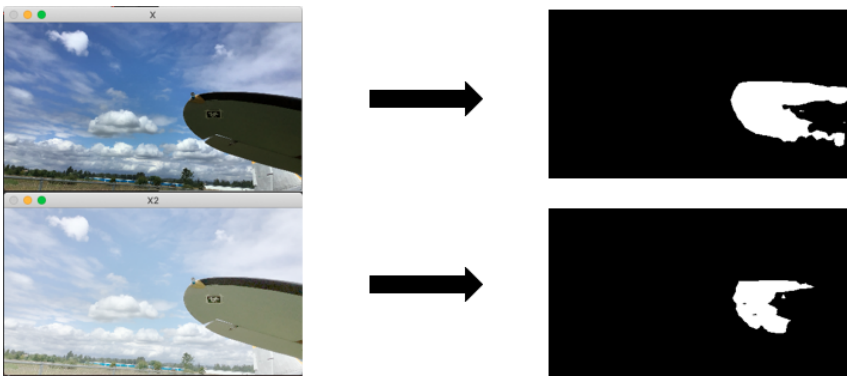
We then manually inserted the sky from the working image into the non-working image as shown below:

Finally we compared the result of evaluating the model with the unchanged image taken during the field testing day to the result of evaluating it with the manually changed image. This simple change led to significant improvements.

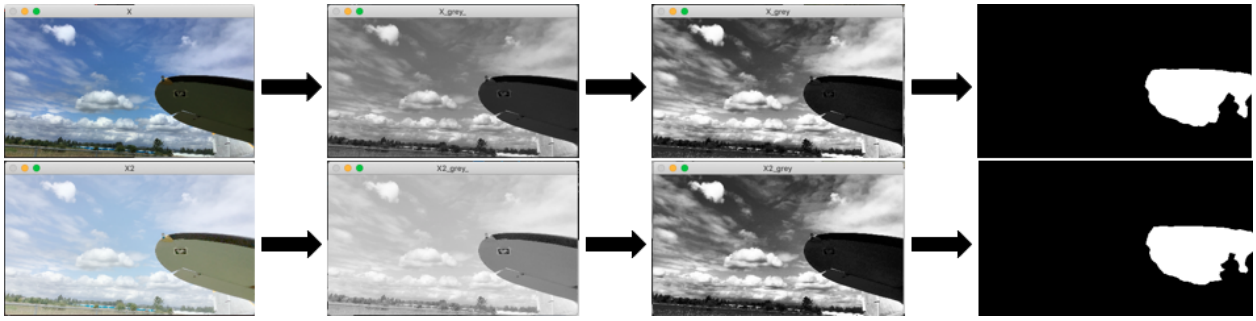


Both problems can be solved by collecting more training data on different days with different weather conditions. Unfortunately this was not possible, due to time and access constraints.
As an alternative approach, we decided to make use of DeepLab, a pretrained segmentation network. It is able to segment background from foreground and was trained on the PASCAL VOC 2012 dataset, which includes “aeroplane” as a category. The goal is to identify the pixels in the images that belong to the airplane. This information can then be passed to the neural network during training alongside the images as an attention prior. The network can use this hint to understand which parts of the image are actually relevant for the pose.

During the implementation of this approach, we encountered the problem that the DeepLab network itself is also sensitive to brightness changes. This means that the same image on a brighter day might result in a different segmentation result.

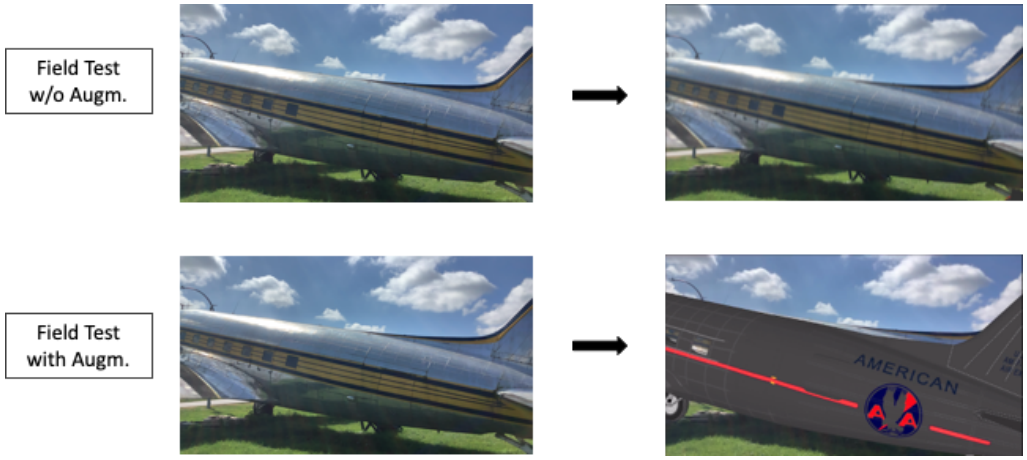


We devised the following preprocessing pipeline to normalize the inputs to the segmentation network and achieve a similar output. First, the input image will be converted to a single channel grayscale image. On this grayscale image, we apply a histogram equalization technique. This technique first counts the number of pixels with any given intensity to build a histogram. Then the intensity values are modified in a way to stretch them over the whole intensity scale [0, 255]. Now we convert the resulting single channel grayscale images to a three channel image, as the DeepLab model expects this input shape. This is done by simply duplicating the intensity values over three channels. The output of the model is an image with integer label values. The value ‘1’ stands for the category ‘aeroplane’, so we convert all values not equal to ‘1’ to ‘0’, which stands for background. Thus, we generate a binary segmentation mask that we convert back to a three channel image. The final outputs of the model for different intensities are now similar to each other.



The resulting segmentation masks can now be fed as a separate three channel image together with the corresponding RGB input frame. This is done by concatenating the image and the segmentation mask along the color channel axis. The input to the model during training and inference is now an image with six channels, instead of just three.

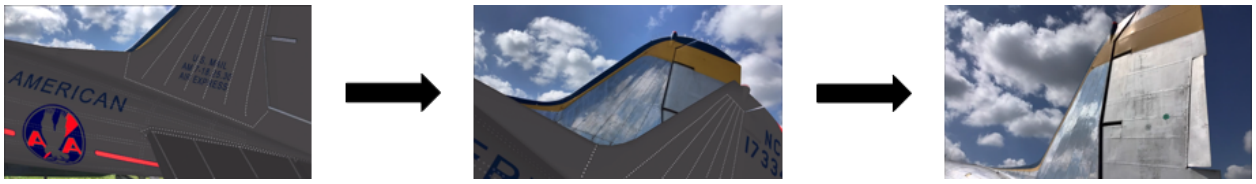
After applying the above mentioned data augmentation techniques during the training of the PoseNet model, the results did improve significantly.



Limitations

The results are promising and work very well on the field test data, as long as the viewpoint that the user is trying to capture was represented in the training dataset. During the data collection there were a number of viewpoints that were not recorded for the training dataset, due to issues of access or marker placements. This results in the model having difficulties to estimate the pose correctly in these cases.

For example the below progression of images shows a shot of the backwing of the DC3 with an accurate pose estimation. However, while the user moves the camera upwards, the pose estimation gets increasingly more incorrect. The training dataset contains images similar to the first image, but no images showing only the top of the backwing. The reason is that it was hard to get access to this area to place a marker at the top of the wing.



A similar result can be seen when looking at the wings from the back:

These problems can easily be solved by spending more time to collect a more thorough dataset, or possibly by adding synthetic data to the dataset. The temporal knowledge of the VidLoc model will also improve these cases by learning a pose progression on sequences of images.

Future Work

There are many future research avenues that can be pursued. The first line of work is to train the same networks on the synthetic data. We suggest training the PoseNet with a synthetic dataset with similar livery as the actual DC3 from which we collected data. Different combinations of training are doable as 100% synthetic, or mixtures between synthetic and real data in ratios of e.g. 70:30 and 30:70. First the network should be tested on the synthetic data to make sure that it actually produces good results and then tested on the real collected data.

After training PoseNet and verifying that the approach is feasible, we suggest training VidLoc, as VidLoc needs significantly more resources and time to complete the training. The next step would be to train the network on synthetic data with a variety of liveries but the same 3D model (same aircraft) and see if the network can learn to ignore the liveries.

Adding/removing weathering artifacts to augment the data could be of use, too. One possible way to do this is to use the segmentation results generated for data augmentation to copy the airplane into a different image with a different weather setting.

It is also worth testing an alternative approach to the data augmentation using gamma correction. To generate the segmentation maps, we used an approach to equalize the images to reduce the influence of different brightnesses on the segmentations. Training the model on these equalized grayscale images, instead of the original images and additional gamma augmentation, might possibly reveal benefits in performance and training speed. However, this needs to be tested.