Plant Tracer:
Tracking plant movement

Deep Learning Approach

Report for Advanced Project III

Aakaash Jois

Electrical and Computer Engineering Department

Tandon School of Engineering

New York University

Abstract

Plant tracker uses motion-capture technology to quantify the time-lapse videos of movement of plants. Here, we show that the movement of the plant can be traced using a deep learning model which makes it robust and work on videos with occlusion while tracking plants which exhibit *circumnutation* type of movement.

Introduction

Plant Tracer is a software application designed to track and quantify the movement of a plant as it grows. It does this by using time-lapse videos of plants captured using cellphones [1]. The goal of Plant Tracer is to stimulate interests in plants by visualizing the complexity in plant development and growth. Plant Tracer is designed not only to track but to analyze the plant movements. These methods are currently restricted to academic research laboratories and make use of expensive camera and domain-specific knowledge and code.

Plant Tracer has been designed to work with two types of plant movement, *gravitropism* and *circumnutation*. *Gravitropism* is differential growth of plants in response to gravity pulling on it [2]. The roots of the plant show positive *gravitropism* (grow towards the pull of gravity) and the stems show negative *gravitropism* (grow away from pulling of gravity). *Circumnutation* is a type of plant movement which is common to all types of plants [3]. *Circumnutation* is observed as the back and forth swaying of plant, but little is understood as to why this takes place. Plant Tracer provides a method for

the users to observe and quantify this behavior in plants and try to develop a hypothesis to better understand this behavior.

This project will focus on tracking the movement of plants which exhibit the *circumnutation* movements using Deep Learning techniques.

Previous work

Plant Tracer is implemented as a *Matlab* program as well as an *iOS* application. It can be downloaded and executed on *Windows*, *MacOS* and *iOS*. Plant Tracer uses a block matching algorithm to track the movement of the apex of the plant. While the block matching algorithm is able to track the plant when it exhibits *gravitropism* type of movement, it fails to efficiently track the movement of the plant when it is exhibiting the *circumnutation* type of movement. One of the observed occurrences in the occlusion of the target which is being tracked. Since the apex of the plant rotates when exhibiting *circumnutation* type of movement, the target being is hidden for multiple video frames consecutively. This leads to the block matching algorithm to lose the target.

Problem statement

In the previous section, it could be seen why the block matching algorithm has difficulty tracking the plant. To tackle this problem, we try to introduce a deep learning based tracker to better track the plant, especially when the

target is occluded.

The purpose of using deep learning is to help the tracking algorithm learn the representation of the object which is needed to be tracked. Since the tracking model is trained offline with multiple videos, the objective is to make it learn to extract the features and obtain a deeper understanding of what the apex of the plant looks like even when it is hidden behind leaves.

Understanding data

The data to train the deep learning model contained videos of plant with *circumnutation* movement. Since the block matching algorithm is able to perform well on *gravitropism* type of movement, just the *circumnutation* was focused here. The video was captured using a smart electronic device such as a smartphone or a tablet. The plant was grown in a plastic pot placed in front of a solid black background with a metric ruler placed next to the plant. The metric ruler had white lettering and white increments to have good contrast with the background. There also is a label placed next to the plant identifying the strain of the recording. The recording was created using a publicly available and free app called Lapse-It [4]. A frame is captured every two minutes and rendered at 20 frames per second.

Annotating data

A total of 12 videos were obtained using the procedure in the previous section. If a video contains more than one plant in the frame, the video is cropped and made into multiple videos isolating the plants. The rest of the area in the video is filled with a black solid background. This also helped in augmenting the data to make the model more robust.

The video frames obtained were unlabelled data and the area to track were not annotated. To annotate the data, the site *vatic.js* [5] was used. The *vatic.js* is a JavaScript-based video annotation tool which provides tools to annotate videos and export a *Vatic* [6] format XML document for each video. *vatic.js* provides a simple matching algorithm which makes it easier to annotate each frame of the video by automatically tracking the object from the previous frame. All the videos were annotated using this tool on *vatic.js*.

Each video in the data is of varying lengths and a varying number of frames. The shortest video contains a total of 1064 frames while the longest video contains a total of 4861 frames. Each video also contains a single plant with a single apex which needs to be tracked. Each frame is annotated with a single bounding box which contains the apex of the plant. The XML document exported from *vatic.js* contains the frame number and the location of the 4 points of the bounding box. It also includes a title for each object. Since the only object in each frame is the apex of the plant, every bounding box will have the title as *apex* with the associated number as *0*.

Previous Architectures

Since the goal of Plant Tracer is to track the movement of a given plant, it needed a robust tracking algorithm. One of the criteria to keep in mind was that the model is going to be deployed desktop but also on mobile. For this, the tracking model should not be very complicated and heavy.

While there are many tracker architectures available, a majority of them rely on a detector module. These detector modules are typically trained to work for categories in *ImageNet* [7] dataset. One of the architectures which are shown to perform very good at tracking single objects in a video is *ROLO* [8]. However, this module contains a *YOLO* [9] object detector as the first stage of the architecture. If the *ROLO* has to be implemented in Plant Tracer, the *YOLO* object detector needs to be first trained to detect plant apex and once this training is complete, the *YOLO* object detector has to be exported and implemented into the *ROLO* model. After this step, the *ROLO* has to be trained to track the plant apex. This architecture was not chosen here because of two reasons. Firstly, the steps to train the model is complicated and it will be tedious in the future if different plant species have to be included. Second, the number of videos needed to train this model is much higher than what is currently available because of the complicated model structure. Also, the *ROLO* model included a very deep neural network and the use of RNNs [10] which need a huge amount of memory to perform predictions.

The architecture which was selected for Plant Tracer was *GOTURN* [11].

GOTURN stands for Generic Object Tracking Using Regression Network. This architecture was much simpler and did not rely on any object detector or any complicated RNN based model for tracking. The *GOTURN* architecture takes two input images and predicts a bounding box. The two input images are a crop around the target in the previous frame and the same crop in the current frame. Since the *GOTURN* model needs the initial location, the first frame's ground truth needs to be provided to the model. *GOTURN* has been trained such that it has learned to identify the object which is present in the center of the previous frame and predict the location of the same object in the current frame.

Model creation

The original *GOTURN* architecture was implemented in *Caffe* [12] on C++. Since the original implement was implemented in C++, it had to be ported to Python before it could be integrated with the rest of the project. Since a majority of deep learning architectures support Python, and many libraries are available to help with the support tasks, it was a better option to implement this project purely in Python. *Caffe* models always come with **model.prototxt** and **model.caffemodel** files. The prototxt file contains the information on how the model has to be built and the caffemodel file contains the weights and biases. The appropriate **goturn.prototxt** and **goturn.caffemodel** was obtain from the *GitHub* repository of the original implementation. Based on the layers used in the implementation, the *Py-*

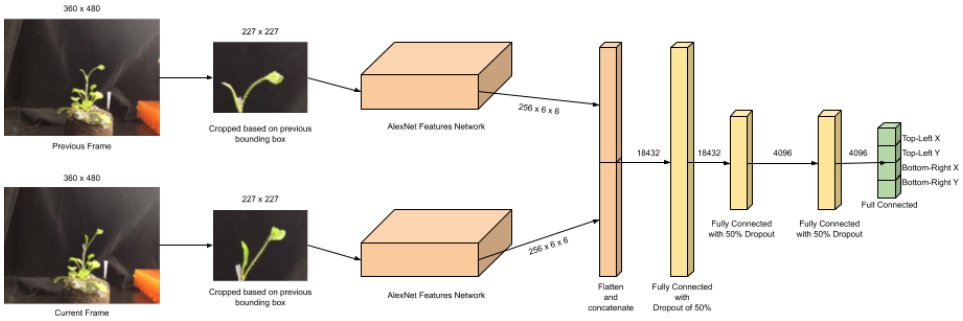


Figure 1: Architecture of the implemented *GOTURN* model.

Torch [13] framework was chosen to reimplement the model. *PyTorch* had a very specific layer like *Local Response Normalization* (LRN) and Convolution using filter groups which were used in the original implementation of GOTURN. Once the framework was determined, the model was implemented in *PyTorch* and also open-sourced as a library on *GitHub* [14]. The architecture in Figure 1 shows the complete architecture of the model implemented.

Training procedure

The training procedure is similar to the training procedure used to turn on the original *GOTURN* model. In each training step, a random video is selected and from this video, a random frame is selected. Let us call this frame f_t. The previous frame (f_{t-1}) and its ground truth is obtained and frames f_t and f_{t-1} is cropped to a crop window of size $2 \times (k \times max(height, width)$ of the frame, where k is an integer multiplier which is set to a fixed quantity. The center of the crop window and the center of the bounding box is the

same. k is initially set to 2 as described in the *GOTURN* paper. The ground truth is also modified to match the crops. The ground truth bounding boxes are scaled to be in a range of 0 to 10. The value 10 has been picked based on the testing done in the *GOTURN* literature. The model is trained for this pair of frames f_t and f_{t-1}. This procedure is repeated for N number of steps where n is the total number of frames available. The L1 Loss was used to calculate the loss at the end of each training epoch which is given by equation 1. In equation 1, x_1 and y_1 correspond to the top-left corner of the ground truth bounding box. x_2 and y_2 correspond to the bottom-right corner of the ground truth bounding box. $\hat{x_1}$ and $\hat{y_1}$ correspond to the top-left corner of the predicted bounding box. $\hat{x_2}$ and $\hat{y_2}$ correspond to the bottom-right corner of the predicted bounding box.

$$L1 = \frac{|x_1 - \hat{x_1}| + |y_1 - \hat{y_1}| + |x_2 - \hat{x_2}| + |y_2 - \hat{y_2}|}{4} \tag{1}$$

The *GOTURN* paper had shown that L1 Loss works better than L2 Loss for training the model, hence that was picked. The network was trained using *Stochastic Gradient Descent* (SGD) optimizer with a learning rate of $1e-5$ and an input image size of 227×227. Here, the input size is the size of the image provided to the model. Once the cropping is performed, f the size of the crop does not match the size of the input size, it is rescaled to the input size before providing it to the model. The model was initially trained for 100 epochs. The training and validation loss versus epoch is shown in Figure 2. The model starts to overfit after 50 epochs. Hence, all the future models are trained for 50 epochs indicated by the red line in Figure 2.

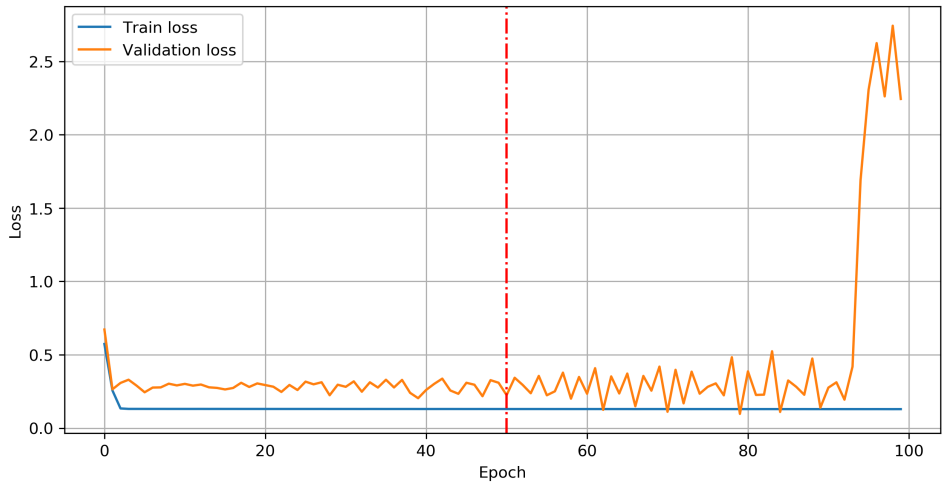


Figure 2: Training and validation loss versus epoch.

The model was trained again, but this time, data augmentation was performed. Once the frames are cropped, augmentation is performed at this point. 3 different augmentation techniques were used using the *albumentations* [15] library. All the augmentations were performed after creating the crops using the steps mentioned previously. It should also be noted that the same augmentation was applied to both the images which input to the model. The first augmentation was to randomly perform horizontal flips. This augmentation easily helps increase the amount of data available to train. This can also help the model become more robust towards plant movement on either side of the center of the frame. The second one was to apply random affine transforms to the image. These affine transforms included random shifts, random scaling, and random rotations. The random shifts were in the range of $[-15, 15]$ pixels. The random scaling was in the range of $[-0.1, 0.1]$ times the image size. Using the scale can help the

model generalize the scale of the target it is tracking. From the dataset, we can see that a couple of videos are recorded with a view closer to the plant and some are recorded from further away. Using the scaling should help the model get more robust towards these variations. The random rotations were in the range of $[-45°, 45°]$. The rotation was performed with respect to the center of the frame. So the bounding box will still remain at the center for frame f_{t-1}. The plant apex in the f_t will be moved from its original position. Creating random rotations should help the model get more robust towards different views of the target it is tracking. Since the apex does not maintain the same orientation in all the frames, by providing the random rotations, this should make the model get more examples towards frames were the target has different orientations. Random change in brightness and contrast was also performed on the images as an augmentation technique. These should help the model get more robust towards variations in the change of lighting in the room when the video is recorded.

Evaluation procedure

Out of the 14 videos available, one of the videos is reserved for validating and testing the models and is not used during the training. For validating the model, when pairs of video frames are generated, the location of the previous frame f_{t-1} is assumed to be predicted correctly and is taken as the ground truth. The frames are provided in the same sequence as they appear in the video. For testing the model, the prediction of the previous frame

f_{t-1} is used for creating the crops of the current frame f_t. If the model predicted the bounding box to be outside the bounds of the image, these values were clipped to fit inside the bounds of the image before creating the crops. The frames are provided in the same sequence as they appear in the video during testing as well.

Model tuning

Multiple parameters were tuned to find the best results for the model. The model which used the weights from the *GOTURN* model had a bad performance when trained on the Plant Tracer data. The model was unable to learn the representation of the plant apex and was not able to track. It could be because the model was trained extensively to be a generic object tracker and it was unable to lose that feature.

To tackle this problem, the weights of the regression network was initialized to random and the feature extractor was frozen to prevent any change in weights. The network was trained again using the same training procedure described above. At the end of the training, an improvement in the tracking could be seen. The model was able to track the plant apex on most of the frames unless it encountered a large movement. During the tracking, the model was able to recover itself when it lost track of the plant apex. This could also be because, since the plant movement is circular in the horizontal fashion, the plant apex comes back into the crop window which is fed to the model a few frames later. This means that the model was able to learn what

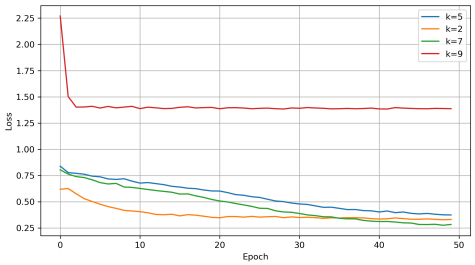
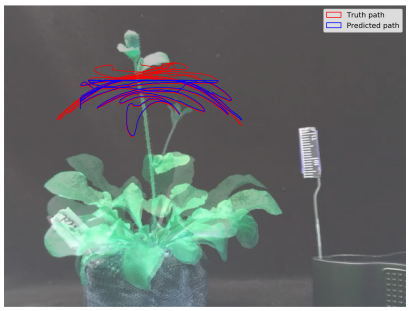


Figure 3: Training loss vs epoch for different values of k

the apex looks like and identify it. To combat the problem of losing track when there is a large movement, the window size multiplier k was increased. k was set to values of 2, 5, 7 and, 9. The complete details on how k is used are described in the training procedure section. Figure 3 shows the training loss for the different values of k.

Result

A window of size 7 shows the best for the current dataset. Figure 4 shows the ground truth path of the plant and the predicted path of the plant. Figure 5 shows the L1 Loss calculated for each frame for the X coordinates, Y coordinates and the mean L1 loss in each frame. The loss values here are much higher than the ones seen during training. This is because the loss is calculated after the predicted boxes are resized back to the original video size (640×480). We also notice that, during the testing procedure, the loss is higher and lasts for more number of frames as compared to the validation. This happens because the error propagates over the future frames as the crop window is dependent on the previous frame predictions. Even a small



(a) Ground truth path of the plant apex vs predicted path by the model when validating

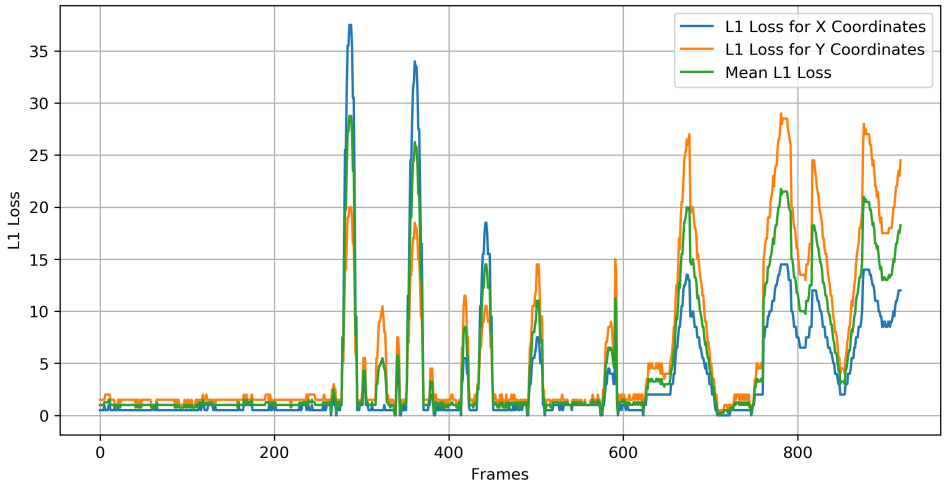
(b) Ground truth path of the plant apex vs predicted path by the model when testing

Figure 4

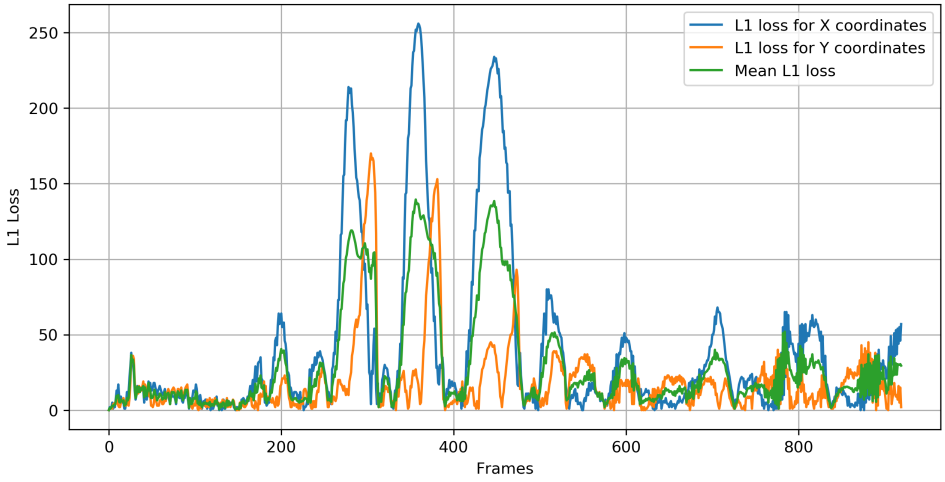
error in the previous frame will offset the crop window in the future frames and this starts to accumulate.

Discussions

The model is able to learn the representation of the plant apex within the first 30 epochs after which the validation loss stays almost the same. It can be noticed that the window size plays a critical role in the model to train well. Increasing the window size benefits the model but a window size too large will stop the model from learning. From Figure 4, we can see that a strong movement in the horizontal direction makes the model lose track of the plant apex. Based on the limited training data available, the model was able to understand what a plant apex looks like. It was able to predict the location of the apex even during some of the frames where it was occluded. From Figure 6 we can see that the apex is hidden in frames 100, 400 and



(a) The L1 loss for X coordinates, Y coordinates and the mean L1 Loss for each frame when validating



(b) The L1 loss for X coordinates, Y coordinates and the mean L1 Loss for each frame when testing

Figure 5

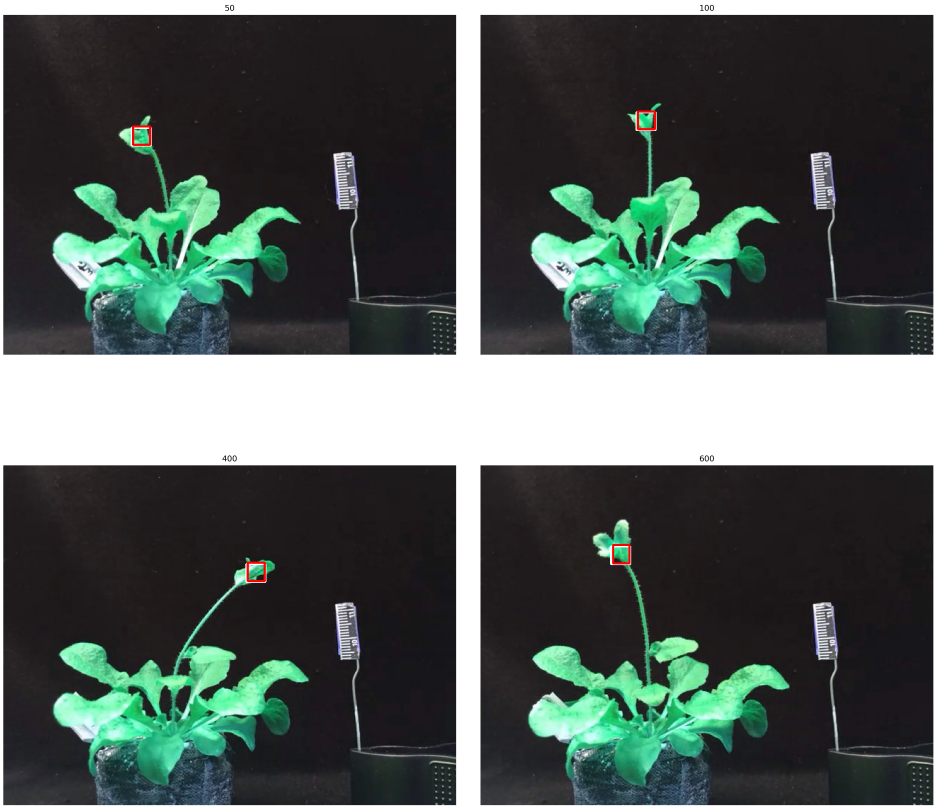


Figure 6: The apex is seen in frame 50. The apex is hidden behind a leaf in frames 100, 400 and 500

600. The model is able to identify the location of the apex in the frames 100 and 400 accurately. It was also able to get a close prediction of the apex in frame 500.

While the model was not able to track the plant movement when the movement was very large, with more examples of that situation, it should be able to learn it. In Figure 8 we can see that almost every alternate frame of the video shows 0 movements in the plant apex. Based on this, the model would have learned that most of the frames do not contain any movement and this

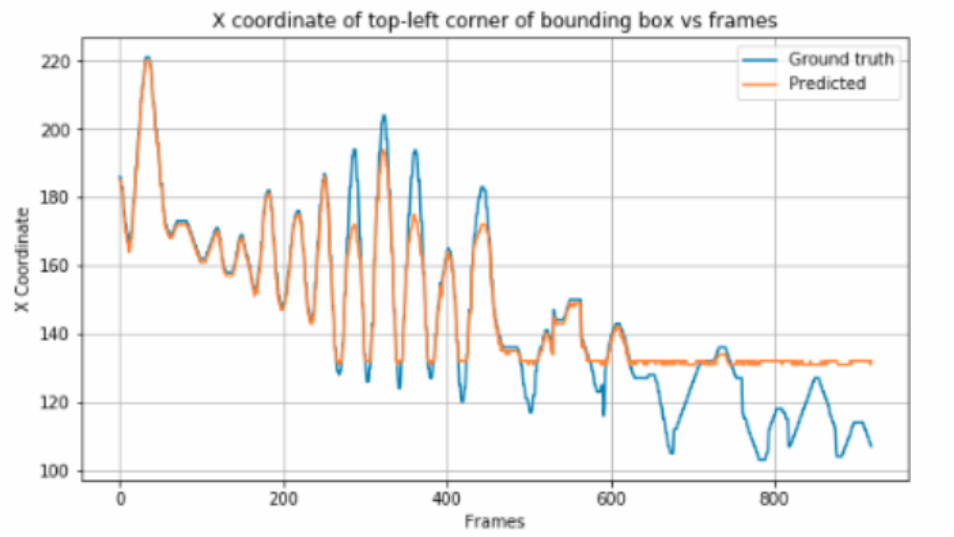


Figure 7: Prediction and Ground truth X coordinate of top-left corner of bounding box against frames of the video

further decreases the model from learning about larger movements in the video. From Figure 5, we can notice that the testing loss has more jitters when compared to the validation loss. The same can be observed from Figure 4 where the testing predictions path is more jittery when compared to the predictions made during validation. When both validating and testing, we can clearly notice that the loss value increases at the same frames of the video. Figure 7 shows how the prediction bounding box drifts away from the ground truth. Figure 8 shows the movement of the bounding box for each pair of frames. We can clearly notice that the bounding box shows 0 movements in almost every 3rd to the 4th frame. While the figure shows for a small subset of frames, this has been noticed throughout the dataset. One of the reasons this is happening could be because of the way LapseIt makes the timelapse videos. This could be handled in the future by training the

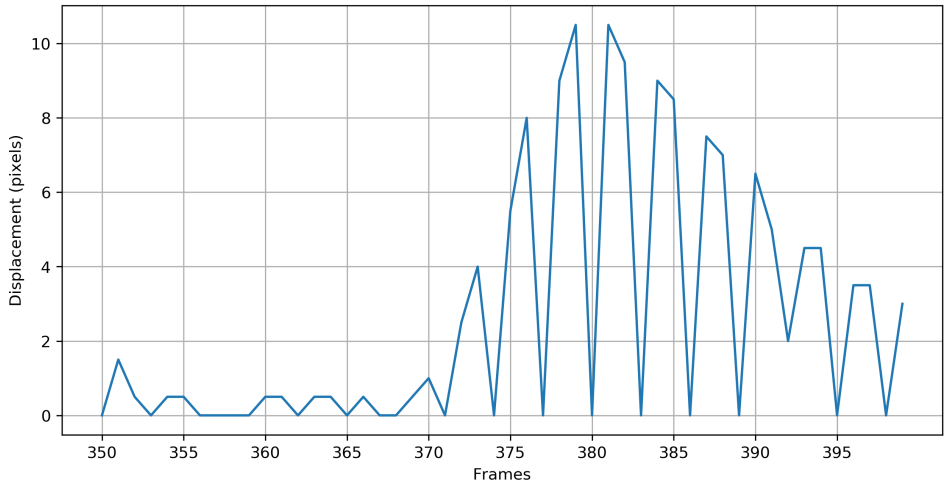


Figure 8: Pairwise difference in the movement of bounding box for the ground truth for the frames 350 to 400.

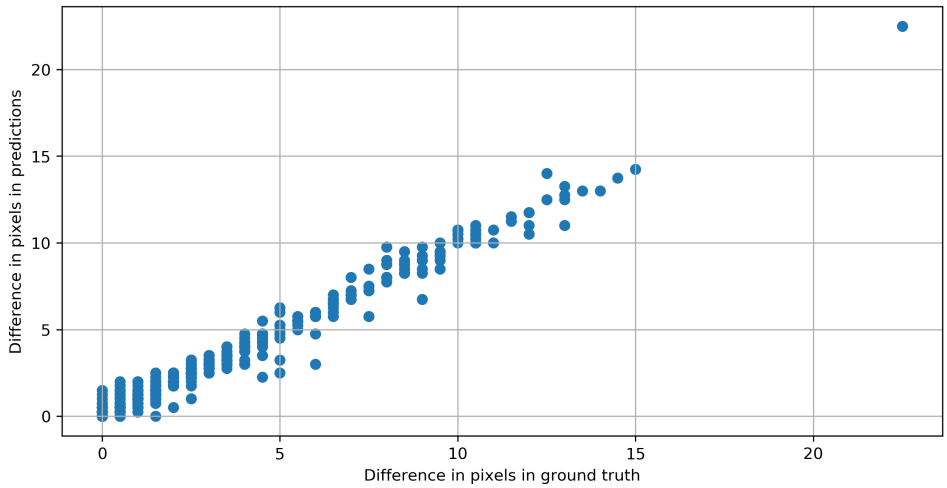


Figure 9: Scatter plot showing the relation of the amount of movement between a pair of frames for ground truth and predictions during validation

network on frames where there is a certain amount of movement.

From Figure 9 we can see that the model is able to make a prediction of the bounding box and follow the same trend as the ground truth. We can notice a linear relationship between the shift in pixels in a pair of frames in ground truth and model predictions. This reinforces the point made in the previous paragraph.

Conclusion

Using the *GOTURN* architecture, we were able to create a neural network model which can track the movement of a plant by tracking the apex with a reasonable success rate. The model was able to track the apex of the plant in the majority of the frames. We could see that the model was able to identify the apex even when it was occluded in certain frames and the model was able to continue recognizing the apex as the apex changed in shape and size. Further, we also saw that the pretrained weights of *GOTURN* were not able to learn the representation of the plant apex and the classification layers had to be trained from scratch to learn those features. The window size of the crops given to the model also plays a significant role. And this size depends on the data available when training the model. The window needs to accommodate for the movement of the plant but should not be too wide because this can include other apexes. This architecture is also simple to implement and export if it has to be deployed on mobile devices which makes it an ideal model.

Future work

Currently, the amount of data available to train the model is limited to 13 videos. Once more data is available, it can be annotated and the model can be trained further. It should also be noted that the model will be able to learn better if the videos are able to show different types of examples for the model to learn. The current implementation of the model uses an *AlexNet* [16] as the feature extractor. A modern feature extractor can be used to improve the performance. The modern feature extractors can be more robust to changes such as lighting, changes in shapes, and stronger movements. Another approach can also be to compute the delta changes in the predictions from the previous frame. This can help the model learn that the output values need to be small and might help in giving better predictions.

The current implementation does not make use of any attention layers. The current architecture can be modified to include attention based architecture might help the model keep track of previous predictions and use the trajectory of the movement to better predictions.

References

[1] E. D. Brenner, "Smartphones for teaching plant movement," *The American Biology Teacher*, vol. 79, no. 9, pp. 740–745, 2017.

[2] C. Darwin, *The movements and habits of climbing plants.* J. Murray, 1875, vol. 9.

[3] ——, *The power of movement in plants.* Appleton, 1897.

[4] L. It, "Lapse it." [Online]. Available: http://www.lapseit.com/

[5] D. Bolkensteyn, "vatic.js a pure javascript video annotation tool." [Online]. Available: https://dbolkensteyn.github.io/vatic.js/

[6] C. Vondrick, D. Patterson, and D. Ramanan, "Efficiently scaling up crowdsourced video annotation," *International Journal of Computer Vision*, pp. 1–21, 2012, 10.1007/s11263-012-0564-1. [Online]. Available: http://dx.doi.org/10.1007/s11263-012-0564-1

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[8] G. Ning, Z. Zhang, C. Huang, Z. He, X. Ren, and H. Wang, "Spatially supervised recurrent convolutional neural networks for visual object tracking," *arXiv preprint arXiv:1607.05781*, 2016.

[9] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015. [Online]. Available: http://arxiv.org/abs/1506.02640

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol.

323, no. 6088, p. 533536, Oct 1986. [Online]. Available: http://dx.doi.org/10.1038/323533a0

[11] D. Held, S. Thrun, and S. Savarese, "Learning to track at 100 fps with deep regression networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 749–765.

[12] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[13] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[14] A. Jois, "Pytorch-goturn." [Online]. Available: https://github.com/aakaashjois/PyTorch-GOTURN

[15] E. K. V. I. I. A. Buslaev, A. Parinov and A. A. Kalinin, "Albumentations: fast and flexible image augmentations," *ArXiv e-prints*, 2018.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf