Report: Block B

Author

Affiliation

Report: Block B

Introduction

Project Overview

The primary objective of this project is to develop a comprehensive pipeline that leverages machine learning and robotics to automate the analysis and manipulation of specialized imagery data. Using a dataset exclusively obtained from a specific company, which employs distinct imaging devices, this pipeline aims to systematically preprocess, analyze, and utilize image data to guide robotic actions effectively. The process encompasses several key phases: image preprocessing, patch creation, model training, post-processing, landmark and root length extraction, and robotic arm manipulation using both reinforcement learning (RL) and proportional-integral-derivative (PID) controllers.

Goals and Scope

The overarching goal of this project is to create a seamless integration of advanced image processing techniques with robotic control systems to address specific operational needs. This involves:

Image Preprocessing and Patch Creation:

The raw images obtained from the specified devices are initially processed to standardize their size and quality. They are then segmented into manageable patches to facilitate more efficient processing, accommodating the constraints imposed by GPU memory limits and enhancing batch processing capabilities.

Model Training:

Leveraging deep learning, specifically convolutional neural networks (CNNs), the project aims to train models capable of interpreting the preprocessed images. These models are designed to identify and categorize various features within the images, crucial for subsequent analysis stages.

Post-Processing and Analysis:

After the initial model predictions, the data undergoes a series of post-processing steps to refine the results. This phase corrects any anomalies and enhances the clarity of the output, preparing the data for precise quantitative analysis, including the extraction of critical landmarks and the measurement of root lengths from the images.

Robotic Manipulation:

The processed data, now enriched with spatial coordinates and other relevant metrics, is used to control a robotic arm. Two distinct control schemes are employed and evaluated: a reinforcement learning-based controller and a traditional PID controller. These systems interpret the processed image data to perform precise movements and actions, mimicking or augmenting human capabilities in specific operational settings.

See Figure 1 at the bottom of this page for more details.

Dataset

Source and Device Specificity

The dataset used in this project was sourced exclusively from a single company, utilizing specialized devices that are not commonly available or used elsewhere. This specificity inherently introduces a certain degree of bias, as the data is representative only of the conditions under which it was collected. The devices employed likely have unique imaging characteristics, such as specific resolution settings, optical properties, and sensor configurations, which are not standard across other platforms or environments. Consequently, the models developed from this dataset are optimized for inputs that closely resemble those produced by these particular devices. This limits the generalizability of the models, as they are less likely to perform with the same level of accuracy if applied to data collected from different hardware or under different operational conditions.

Labeling Process

The labeling of the dataset was performed manually, utilizing ImageJ software. Each student was assigned to either three or four images, who was responsible for annotating it according to specified criteria. This manual process introduces several potential sources of bias:

Inter-rater Variability:

Given that multiple individuals were involved in the annotation process, there is a natural variability in how each person interprets what constitutes a relevant feature within the images. Despite training and guidelines, personal judgment plays a significant role, which can lead to inconsistencies across the labeled dataset.

Subjectivity in Interpretation:

Each annotator's understanding and subjective interpretation of the images can lead to differences in the labeling outcomes. What one annotator might consider a feature of interest, another might overlook or judge as irrelevant. This subjectivity can affect the reliability and uniformity of the training data.

Labeling Accuracy and Exhaustiveness:

The accuracy of labels and the completeness of features annotated can vary significantly, influenced by the annotator's level of expertise, familiarity with the domain, and even fatigue. This variability can lead to incomplete or inaccurate labels, which in turn can impact the training of machine learning models, potentially introducing errors or biases in the models' ability to generalize from the training data.

Early Milestones: Image Preprocessing and Patch Creation

The script's initial focus is on image preprocessing - a step I recognized as vital to ensure the uniformity of data. I apply various traditional cv techniques to cut all the images to the same size and exclude the unnecessary and even disturbing borders, I do this by looking at all connected components in the binary image and choosing the biggest one.

Following this, I create and save image patches from those, before we can transform them we have to pad the images so that the image is dividable by the patches in both width and height, i did this with my own padder function. This step was crucial for handling large images by breaking them down into smaller, more manageable segments to stay in the bounds of my gpu memory and be able to pass them as batches.

Deep Dive: Patch Analysis and Model Building

Next, I delved into the world of deep learning. I defined a weight map for my images, focusing on important regions within the image during model training, which is the middle. So i just defined the weightmap as a gradient which is lowest on the borders and highest in the middle, this was crucial to get the model to stop focusing on edges or other disturbances on the corners. Next i wrote my custom metrics classes F1 Score and IoU for model performance. For the model architecture i chose the U-Net with a ResNet50 backbone for effective image segmentation. Here i can import my weighted loss function with the map created earlier by adding a new function which calculates the binary crossentropy and multiplicates it to all my weights from he map and then take the average, adding a layer of specificity to the learning process. Important is also to mention that i initialized my model with the weights of the pre-trained ResNet50 backbone, which is a common practice in deep learning. I then compile the model with the Adam optimizer and the custom metrics.

Enhancing Model Specifics: The Power of ResNet50 and U-Net in Agricultural Image Processing

Building upon the earlier sections, it's important to highlight why the combination of ResNet50 and U-Net architectures forms a potent solution for our plant image processing task. ResNet50, known for its deep residual learning framework, excels in feature extraction even from very deep networks, which is important for accurately identifying the patterns in agricultural images. By integrating it with the U-Net architecture, renowned for its efficiency in image segmentation tasks, the model gains an exceptional ability to not

only recognize but also precisely delineate our plant roots and leaves.

```
def resnet_unet(input_size, num_classes):
    base_model = ResNet50(weights='imagenet', include_top=False, input_shape=
```

Leveraging ResNet50 Layers:

In the construction of the ResNet50 U-Net model, different layers of the pre-trained ResNet50 serve as the backbone. By utilizing layers like conv4 block6 out and conv3 block4 out, the model can extract rich, hierarchical feature representations. This is particularly valuable in our task, where different scales of features (like minor root tips or larger plant structures) are essential for accurate analysis.

```
    conv4 = base_model.get_layer('conv4_block6_out').output
    conv3 = base_model.get_layer('conv3_block4_out').output
    conv2 = base_model.get_layer('conv2_block3_out').output
    conv1 = base_model.get_layer('conv1_relu').output
```

UpSampling and Concatenation:

The process of UpSampling and concatenating feature maps from ResNet50 with the U-Net architecture is a critical step. It ensures that the model does not just analyze high-level features but also retains finer details lost during downsampling.

```
    up4 = UpSampling2D((2, 2))(conv4)
    up4 = concatenate([up4, conv3])
    up4 = Conv2D(256, (3, 3), activation='relu', padding='same')(up4)
    up4 = BatchNormalization()(up4)
    up4 = Dropout(0.2)(up4)

    up3 = UpSampling2D((2, 2))(up4)
    up3 = concatenate([up3, conv2])
    up3 = Conv2D(128, (3, 3), activation='relu', padding='same')(up3)
```

```
up3 = BatchNormalization()(up3)
up3 = Dropout(0.2)(up3)

up2 = UpSampling2D((2, 2))(up3)
up2 = concatenate([up2, conv1])
up2 = Conv2D(64, (3, 3), activation='relu', padding='same')(up2)
up2 = BatchNormalization()(up2)
up2 = Dropout(0.2)(up2)

up1 = UpSampling2D((2, 2))(up2)
up1 = Conv2D(32, (3, 3), activation='relu', padding='same')(up1)
up1 = BatchNormalization()(up1)
up1 = Dropout(0.2)(up1)
```

Incorporation of Dropout and Batch Normalization:

My use of Dropout and Batch Normalization in each UpSampling step addresses two key challenges in deep learning: overfitting and internal covariate shift. Dropout reduces the model's reliance on any single pattern, promoting generalization, while Batch Normalization helps in stabilizing the learning process. This combination enhances the model's robustness, a necessary feature for the varied and unpredictable nature of our plant images.

Sigmoid Activation for Final Layer:

Choosing a sigmoid activation function for the final layer aligns perfectly with the need for binary segmentation in our images, such as distinguishing between plant and non-plant elements.

```
outputs = Conv2D(num_classes, (1, 1), activation='sigmoid')(up1)
model = Model(inputs=base_model.input, outputs=outputs)
```

Model Training

During the training i tried many different architectures, also different backbones like the EfficientNet which performed a little worse. All of my initial models were multiclass because thats the logical thing to do when given multiple mask classes to detect. It turned out that we only needed a binary one with only the roots so i changed my entire model two days before the kaggle competition to be optimized to just roots but i ended up screwing up my own model with that and got a very bad placement on kaggle. I wanted to try some different models still like Inception but i didnt have enough time for that.

Predictive Insights: Image Processing for Prediction

Now we get to the exciting part, predicting the masks for the image patches. When predicting the patches it is important to keep the order of the patches according to the original image. I then use my own written unpatchify function because the imported one is too much of a pain to use.

Refinement and Detailing: Post-Processing of Predicted Masks

Post-prediction, the part were we cover up our models weakpoints. This step has been redone so many times by me because of the ever changing model. The issue is that everytime i change something in my model architecture the predicted images have different types of noise or things they do wrong or wright, this makes this step particularly difficult because it always needs to be readjusted to the model. In my final version i am using binary threshholding and morphological operations to remove the noise and fill in the gaps and connect some roots. Another important step i implemented is to check for each connected component its pixel density. I do this because roots tend to be very thin and spread, this makes the density of the object very low. A water droplet on the other hand is never large in area but filled out almost completly, which means high pixel density. This enables me to remove water droplets or other disturbances which are denser thasn the roots. Together with this i also checked its circularity just to be sure to remove unwanted objects. Using this i am left with only the roots and can then save the masks. The output

looks like this, it is the prediction we get from the model with the postprocessing so we see the selected parts in green:

See Figure 2 at the bottom of this page for more details.

A New Angle: Skeletonization and Analysis

Next up is the skeletonization which is a very important step to extract root ends and branch points. It was used to calculate root lengths as well but that is not needed in the pipeline. I then get all my endpoints for each skeleton and look for the lowest point in a skeleton, this leaves me with the main root ends. I would say this maybe does not work on all of the images but in case it wouldnt i can always just use the length calculator. This process is shown below. I then get the main root ends and save them.

```
def draw_main_root(image, skeleton_branch_data, u):
    info_image = np.copy(image).astype(np.uint8)  # Convert to uint8 if neede
    info_image *= 255
    main_root_lengths = {}
    lateral_root_lengths = {}

    for skeleton_id, group in skeleton_branch_data.groupby('skeleton-id'):
        G = nx.Graph()
        lateral_root_lengths[skeleton_id] = []

        # Add edges based on branch data, with actual lengths as weights
        for index, row in group.iterrows():
            src = (int(row['image-coord-src-1']), int(row['image-coord-src-0'
            dst = (int(row['image-coord-dst-1']), int(row['image-coord-dst-0'
            length = row['branch-distance']  # The actual length of the branc
            G.add_edge(src, dst, weight=length)
```

```
endpoints = [node for node, degree in G.degree() if degree == 1]
topmost_point = min(G.nodes, key=lambda point: point[1])
max_path_length = 0
# Find the longest path based on actual branch lengths
for end_point in endpoints:
    if topmost_point != end_point:
        try:
            length, path = nx.single_source_dijkstra(G, topmost_point
            if length > max_path_length:
                longest_path = path
                max_path_length = length
        except nx.NetworkXNoPath:
            continue
plant_id = assign_plant_id(topmost_point[0])
main_root_lengths[plant_id] = max_path_length
for edge in G.edges(data=True):
    if not (edge[0] in longest_path and edge[1] in longest_path):
        lateral_length = edge[2]['weight']
        lateral_root_lengths[skeleton_id].append(lateral_length)
```

This is how i get the root lengths, lateral root lengths, landmarks and endpoints. I am using all points from my summarization of the skeleton and load them into a graph with all the edges. Here it is important to add the branch lengths as weights so that it doesnt calculate the distance with the shortest path but rather the accurate root length. Next i am extracting the start point in the skeleton by looking at all nodes y axis and picking the highest (y axis is inverted so i search for the lowest y value). Next i am looking for all paths from this points until i find the longest possible, this will be my main root, important here again to add the weights for accurate lengths. This is how an image with

landmarks looks: See Figure 3 at the bottom of this page for more details.

Bridging Digital and Physical: Point Extraction and Coordinate Transformation

Last part of the pipeline is the robotics part. Here i am using the coordinates of the root ends, which i transorm to the real world coordinates using my get meter coordinates function. This is how it works: i get the pixel coordinate on the screen convert those into meters with a given conversion rate (24pixels/mm) and then due to the robots workarea being much bigger than the space my plants are on i have to add the offset in the code set as start point, important here is to swap the points x and y axis because the robots axis are inverted.

```
def get_meter_coordinates(point):
    start_point = np.array([0.10775, 0.088])
    pixels_per_meter = 24.52 * 1000
    point_in_meters = np.array([point[1 - i] / pixels_per_meter for i in rang



    return np.array([start_point[i] + point_in_meters[i] for i in range(2)])
```

The Climactic Integration: Reinforcement Learning Environment Interaction

In the final stage, I implemented my reinforcement learning. This was trained over multiple days with many different hyperparameters and reward functions. The most important part about a reinforcement learning training is the reward function. For me it looks like this:

```
def _calculate_reward(self, pipette_pos):
        cur_distance_to_goal = np.linalg.norm(pipette_pos - self.goal_positio
        prev_distance_to_goal = np.linalg.norm(self.prev_pipette_pos - self.g
```

```
# Calculate improvement in distance
distance_improvement = prev_distance_to_goal - cur_distance_to_goal

# Define constants for rewards, penalties, and scaling factors
GOAL_REACHED_REWARD = 200.0
SCALER = 10



# Update penalty/reward based on direction of movement
if distance_improvement > 0:  # Moving towards the goal
    self.consecutive_wrong_direction = 0
    self.consecutive_right_direction += 1
    distance_reward = distance_improvement * self.consecutive_right_
else:  # Moving away from the goal
    self.consecutive_right_direction = 0
    self.consecutive_wrong_direction += 1
    distance_reward = distance_improvement * self.consecutive_wrong_

# Check if the goal is reached
termination_threshold = 0.001
if cur_distance_to_goal < termination_threshold:
    terminated = True
    reward = GOAL_REACHED_REWARD + distance_reward
else:
    terminated = False
    reward = distance_reward
```

```
        # Check for truncation
        if self.steps >= self.max_steps:
            truncated = True
        else:
            truncated = False

        # Update the previous position for the next step
        self.prev_pipette_pos = pipette_pos.copy()

        return reward, terminated, truncated
```

Here i calculate the current distance from pipette to goal and the distance of the previous step and subtract them. If i got closer to the goal, this is a positive value of the right distance travelled and if he is going away from the goal it is a negative reward and also as big as the distance travelled. This assures that the agent tries to get the the goal as fast as possible. Also for reaching the goal he recieves an extra 200 points to assure thats the main goal. Lastly i implemented consecutive rights or wrongs so that the agent is motivated to to keep getting closer, because it is a linear increment of the reward he gets, same for negative. I performed my training on a remote server so that it could run independenty since this process takes multiple days and used weights and biases to keep track of my agents metrics and to save them later on. The remote server was also essential for the possibility to train multiple agent at the same time. Using my trained PPO model, I set out to make goal-oriented decisions based on the insights gained from image analysis. This part of the script marked the transition from computer vision to robotics.

PID-controller

Opposing to the reinforcement learning solution we have another one, the PID-controller. This is a commonly used controller for any regulation tasks like temperature, presssure or in our case speed/position. I implemented both in my pipeline

and benchmarked the results

See Figure 4 at the bottom of this page for more details.

We can see that the PID-controller is a little bit faster on each plant which added up on hundreds of samples makes a big difference. We can see that the PID just rushes from the start on with max speed towards the goal and slowly decellerates as it gets closer, the rl however starts off in a little curve and then just accellerates twards the goal which most likely makes him overshoot his target position as well, but that is not captured here. First i thought this might be due to him checking with short distances which is the right direction so that he will just get a very little loss while figuring out the direction, but after careful inspection of the reward function i think that this is due to him actively exploiting a weakness in my reward system. In order to achieve the maxium reward for the model he does not need to complete it in the least amount of steps rather as long as he goes the right direction the maximum reward will always be the same, and secondly i think he exploited my consecutive right multiplier. Because if you want to maximize the reward you gotta take some really small steps in the beginning to get the consecutive steps multiplier up and then every distance he takes counts as more. To fix this issue i shouldve implemented a small reward loss for every step he didnt reach the goal, which i had in earlier versions of my reward.

Conclusion

Reflecting on my journey with the Full pipeline script, I am genuinely excited and proud of what I've achieved. This project wasn't just coding for me; it was about diving into the fascinating world of AI and combining it with my background in robotics, which I find super interesting.

Working on the Full pipeline script was like piecing together a complex puzzle. Each part, from handling image data to making the AI learn and adapt, was challenging but really rewarding. What stood out for me was the computer vision part. It's amazing how you can make a computer "see" and understand images, kind of like teaching it to look at

the world like we do, but even better in some ways.

Then there's the robotics part. I've always been into robots and how they work. Putting AI into the mix opens up so many possibilities. Imagine robots that don't just do what they're programmed to do, but can learn and adapt by themselves. That's the kind of future I'm excited about, and this project was a small step in that direction.

I learned a lot about different things like image processing, how to split images into smaller parts and work on them, and then put them back together. Then there was the whole thing about training the AI model, which was tough but super cool. Seeing the AI get better at its job over time was like watching something come to life.

The real kicker was when I got to combine all this with an environment I created, the 'OT2Env'. This was where the AI had to make decisions in real-time, learn from what's happening around it, and figure out the best move. It was a bit like teaching a robot to think on its feet.

And there was this interesting comparison between how the AI model and a PID controller worked in the same environment. It's like looking at two different ways to solve the same problem, and seeing which one does better.

In the end, seeing everything come together, the AI making smart moves, the robot performing tasks accurately, it felt like a big win. It wasn't just about getting the code to work; it was about creating something that combined two fields I'm passionate about - AI and robotics. It's projects like these that make me excited about where technology is headed and how I can be a part of that journey.

So, yeah, this project was a big deal for me. It was challenging, but it was also a lot of fun and super rewarding. It's definitely something I'm going to remember and build on as I explore more in the worlds of AI and robotics.

Acknowledgments

The author would like to acknowledge the assistance provided by OpenAI's ChatGPT, which was instrumental in the formulation of this paper. The insights provided

by ChatGPT were derived from its capabilities as a large language model developed by OpenAI.

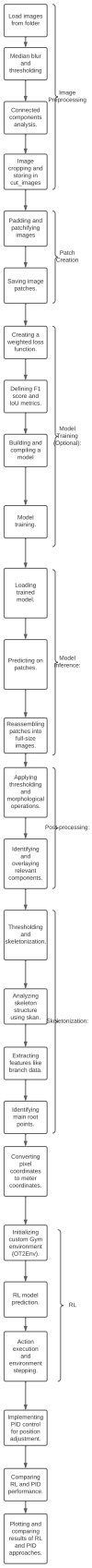
Load images from folder
Median blur and thresholding
Connected components analysis.
Image cropping and storing in cut_images
Image Preprocessing
Padding and patchifying images
Saving image patches.
Patch Creation
Creating a weighted loss function.
Defining F1 score and IoU metrics.
Building and compiling a model
Model training.
Model Training (Optional):
Loading trained model.
Predicting on patches.
Reassembling patches into full-size images.
Model Inference:
Applying thresholding and morphological operations.
Identifying and overlaying relevant components.
Post-processing:
Thresholding and skeletonization.
Analyzing skeleton structure using skan.
Extracting features like branch data.
Identifying main root points.
Skeletonization:
Converting pixel coordinates to meter coordinates.
Initializing custom Gym environment (OT2Env).
RL model prediction.
Action execution and environment stepping.
RL
Implementing PID control for position adjustment.
Comparing RL and PID performance.
Plotting and comparing results of RL and PID approaches.

Figure 1

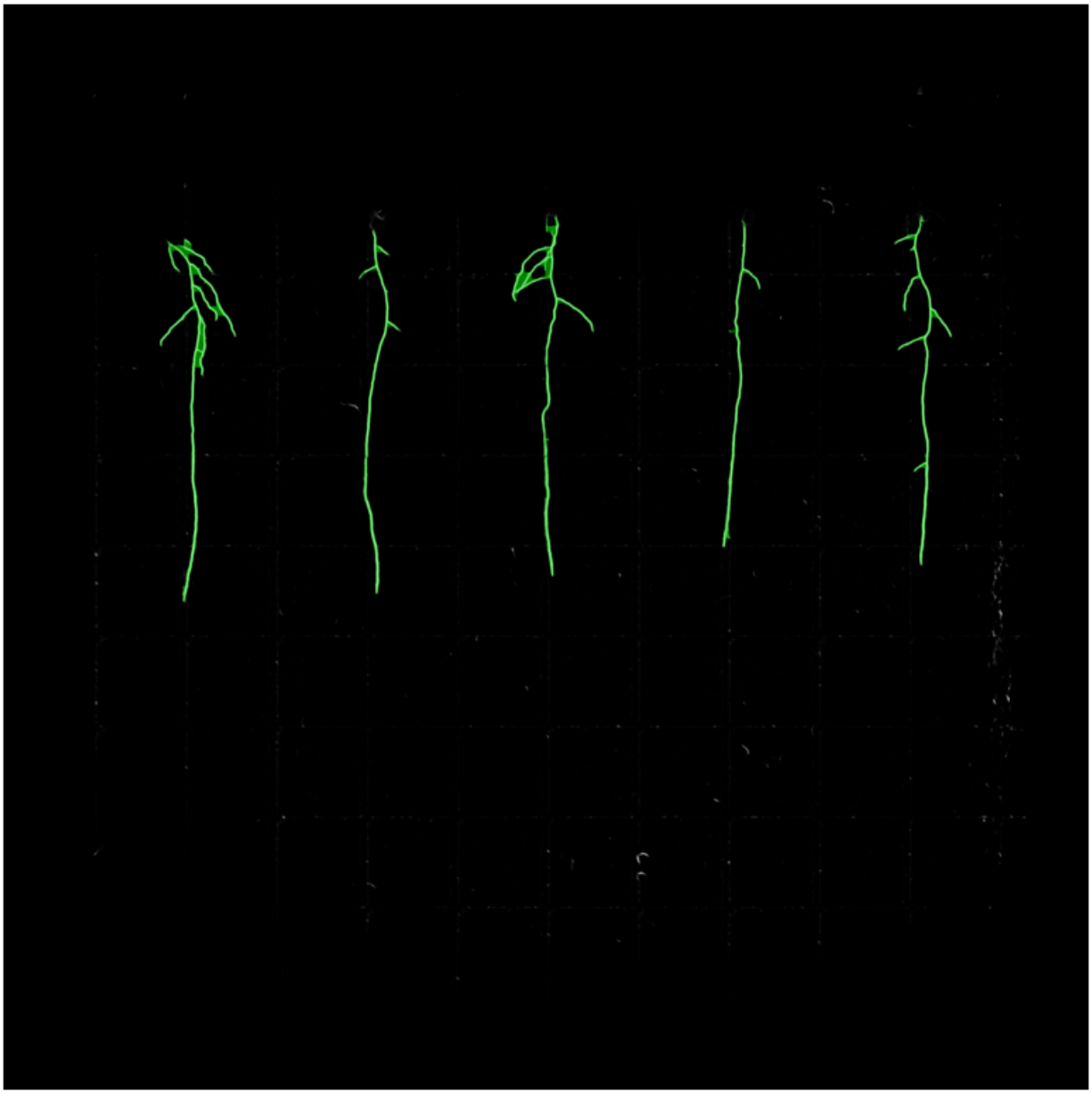

Figure 2

Post-processing result

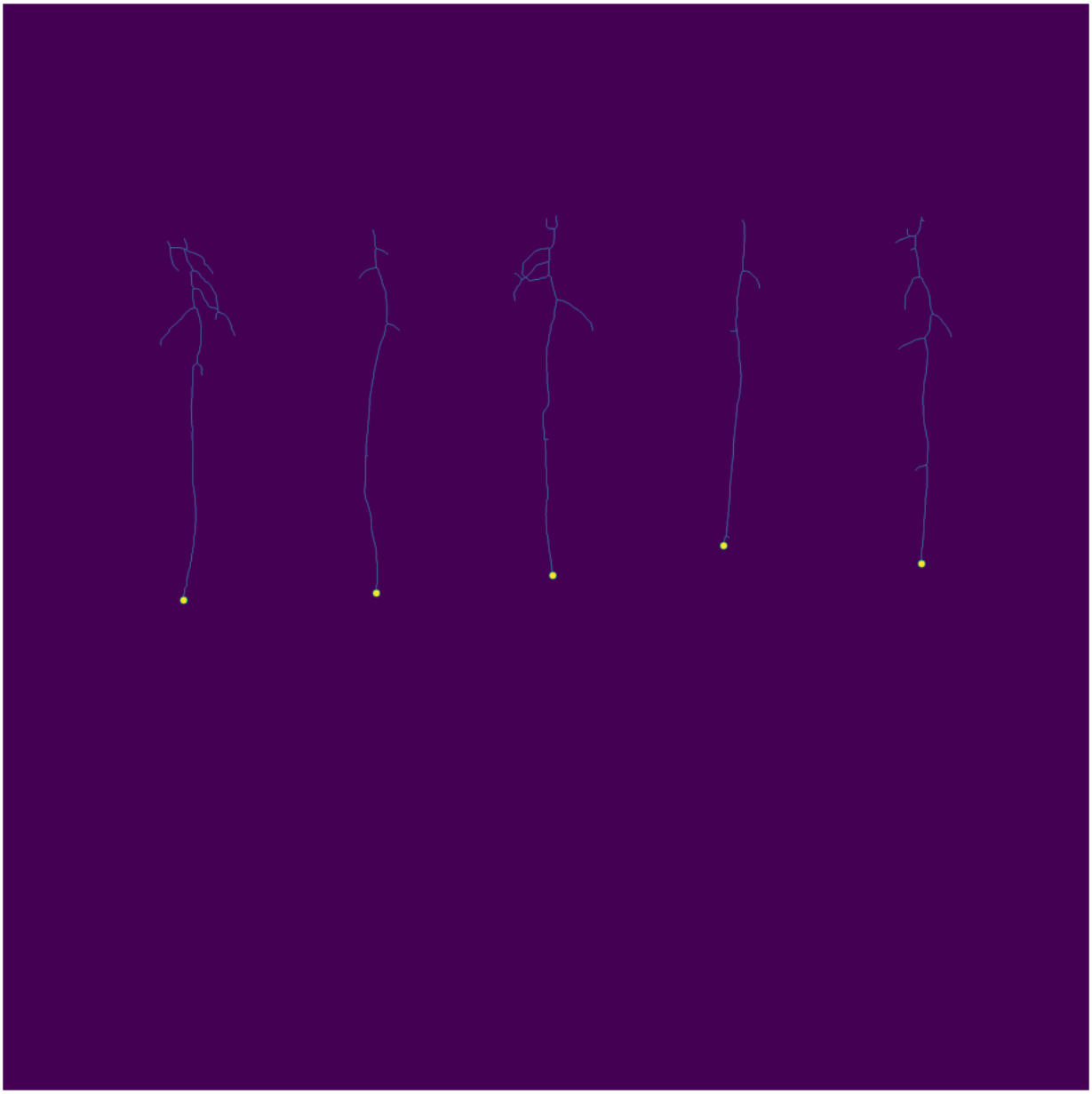

Figure 3

Landmarks

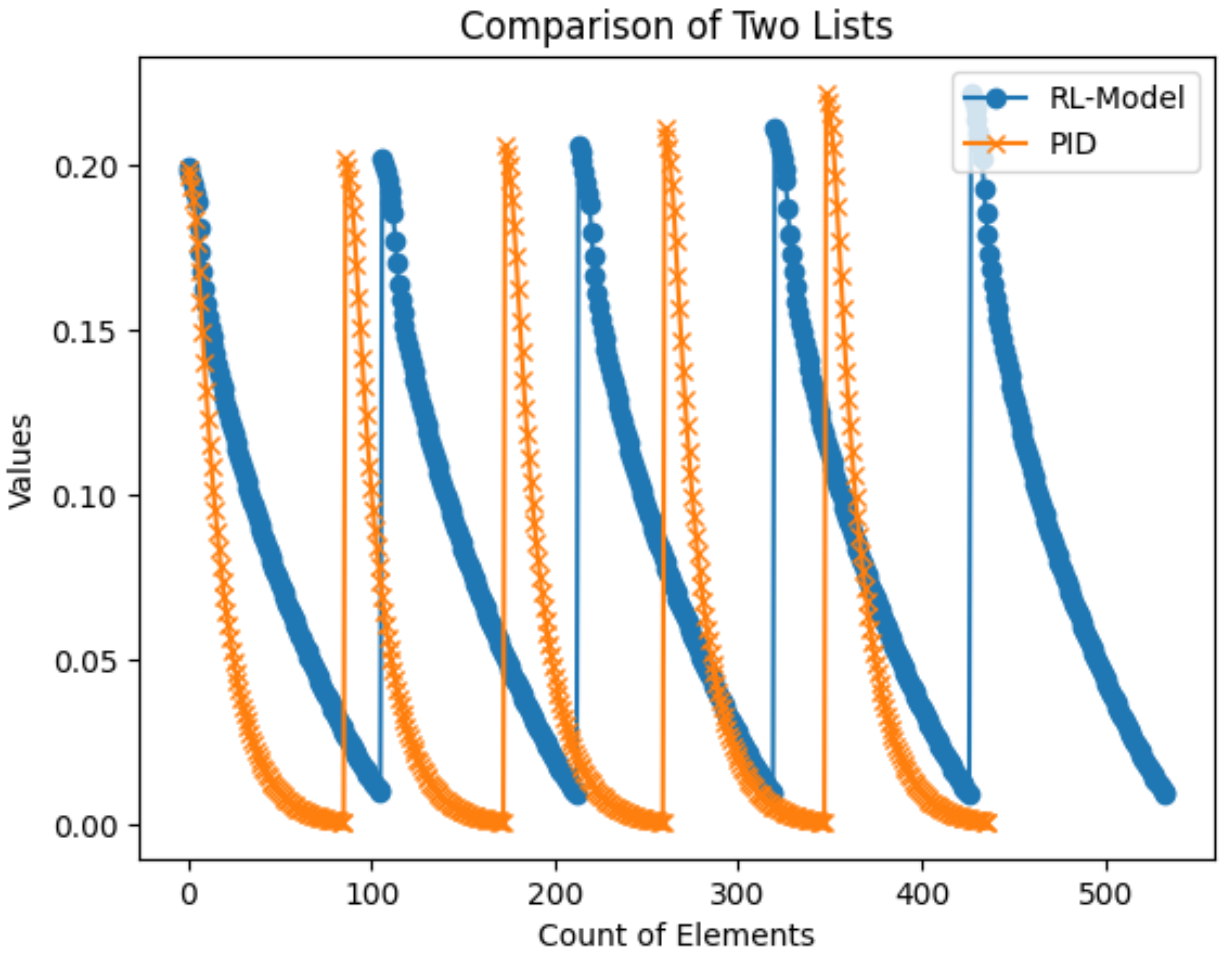

Figure 4

Benchmark