Technical Proposal

Robomaster@UC Berkeley, Team TensorBot

December 25, 2019

1 Software

1.1 Computer Vision

1.1.1 Overview

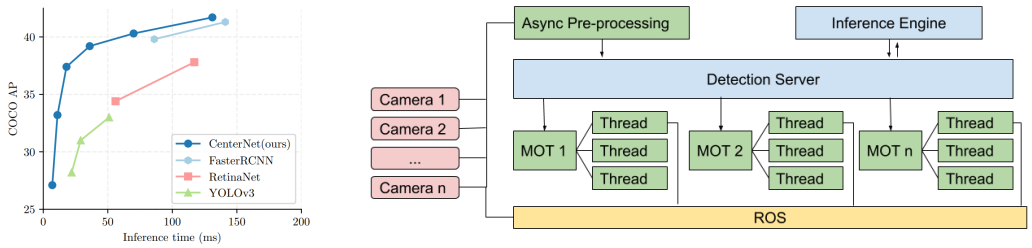
We have three major tasks for computer vision. First, we have general robot detection for multiple on-robot cameras to detect the relative position and pose of enemy robots. Second, we detect armor plates at low latency for aiming and shooting. Third, outpost cameras detect enemy 3D global coordinates for decision making. We present both algorithm choices and latency optimizations.

1.1.2 General Robot Detection

A general robot detector will be run on 4 cameras covering 360° view around the robot and output relative positions and orientations of enemies. Position and orientation helps determine the threat of an enemy robot and build our cost map in planning. The general robot detector will also be able to detect partially occluded robots so we can do pre-aiming before we see the armor plate.

Last season we trained the MobileNet SSD. Thanks to our efficient synthesized data (Figure 3b), we received very impressive results as we showed in our demo (link). This year we add orientation in addition to bounding box. However, SSD is known to completely fail on small or distant objects. Thus we will switch to CenterNet, a much faster, more accurate, state-of-the-art network (Figure 1a) as described in Object as Points by Zhou et al. [2019].

We want to especially emphasize that we have secret optimization. After the competition from last season, we have updated our inference pipeline with Berkeley's latest research MCRDT framework in Figure 1b such that we can run detection on an i5 NUC at 15fps with less than 10ms latency on 4 cameras simultaneously. This method is achieved with special optimization for batch size 1 to reduce latency with OpenVINO, using highly paralleled trackers between inference, and using low-latency ZeroMQ instead of ROS. The GPU performs inference for each camera sequentially and multi-object trackers handle the gap between each detection for each camera as shown in Figure 2. Camera reading is implemented in separate thread and uses an inference request queue so that the GPU pipeline is never interrupted.



(a) (b)

Figure 1: a) CenterNet is now the state of art, and we will code an optimized version b) Our research MCRDT framework

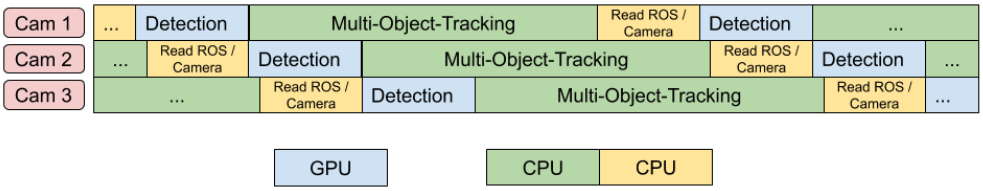


Figure 2: MCRDT pipeline

1.1.3 Armor Plate detection

We detect the 3D coordinate of enemy armor plate centers on a stereo camera. Armor plate detection is easier than extracting meta information of robots due to the LED light bars. Thus we use more traditional computer vision techniques, such as first filtering out everything except LEDs with HSV filtering, and then perform parallel line detection. Because we already have ultra low latency detection of the robot, we can perform this step within the bounding box of detected robots, so our parallel line detector will be very robust. With RGBD, we can easily map 2D coordinates in an image to 3D coordinate of the center of the armor plate.

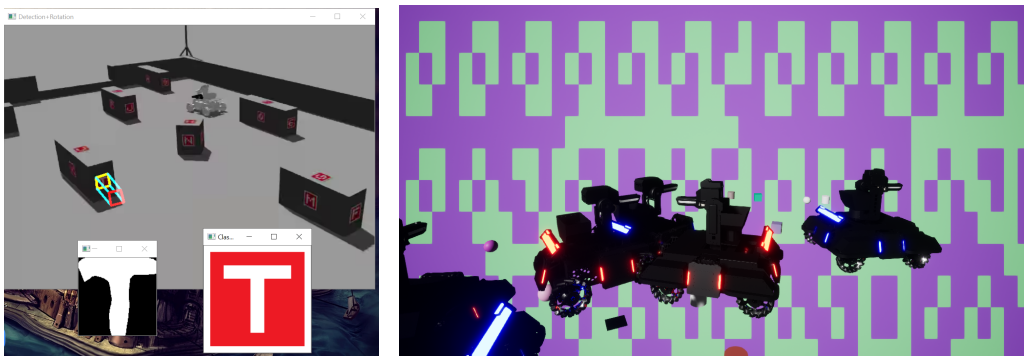
1.1.4 Outpost Camera Detection

Since we are allowed to install outpost cameras, we will use them to locate all robots and their orientations in the field.

First, since the officials haven't release a standard camera mount, we assume we need automatic calibration to tolerate actual field uncertainty. Upon initialization, the outpost cameras will determine their 6D pose in the field with Vision Markers. Based on the real-world coordinates and size of the markers, and the camera's intrinsic matrix, we can calculate the coordinates of the sentries with regression.

As in Figure 3a, we have already developed a detection algorithm that can accurately detect and determine the 6D pose of markers in an image, from which we can derive our outpost cameras' global pose.

Since the robots are always on the same xy plane (ie always at the same height), we can use back-projection to localize the robots after detecting their location in the image. Assuming the robots are always on the same xy plane, each pixel in the sentries' images



(a) (b)

Figure 3: a) Our program to calibrate camera pose in actual field b) Synthesized data generated using NVIDIA's Dataset Synthesizer in Unreal Engine 4

has a one-to-one correspondence with a position in that xy plane. These positions can be computed using the previously calculated values of the camera's intrinsic and extrinsic parameters. With this, all we have to do is perform detection, which we can do with the same architecture as we use for the robots' cameras, except trained on different data from a different angle. Figure 3b is sample data we have generated including both last year's robots and this year's robots in NVIDIA's dataset synthesizer. We can also easily generate the view from outpost viewpoint. The models are close to photorealistic, and since the images are created in simulation, they come with the 6D poses pre-labeled, removing the need for hand labeling real-world data. This detection is running on a laptop outside of the robots, so we can use an NVIDIA GPU and TensorRT for best performance.

1.2 Localization

We use the AMCL Localizer from RoboRTS for extracting information and statistics regarding the distance between our own robots and their surrounding environments. We further use the outpost cameras to handle the random initialization, as in Figure 4.

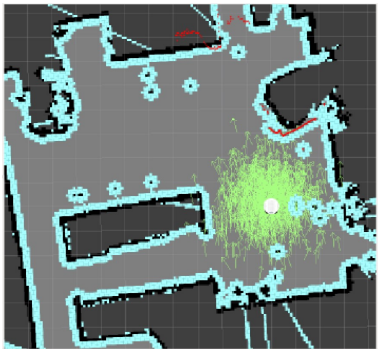


Figure 4: Using particle filter to determine the location of the robot

1.3 Strategy and Decision

1.3.1 Decision Overview

The strategy module takes in the state observations and outputs commands to the execution module. Given the large amount of information available in combat and the advantages of the algorithms and hardware mentioned above, we assume our robots will have near ground-truth state with Gaussian noises at all times. We assume that our gimbal always points to the closest armor plate of any enemy robot in sight. An overview of our system is shown in Figure 5.

We implement the game strategy into two parts: Maneuver and Shooting. Shooting only concerns whether the robot should shoot, and, if possible, how frequently it shoots. Maneuver concerns everything else. Since we have changed the code for the STM32 developer board, these two parts' control can be disentangled. Assuming the controls to them are orthogonal, we propose the following strategies separately. In addition, we break maneuver into lower level planning policies and upper level decision policies. The upper level decision outputs target waypoints while lower level planners use optimization and decision rules to plan path and rotation.

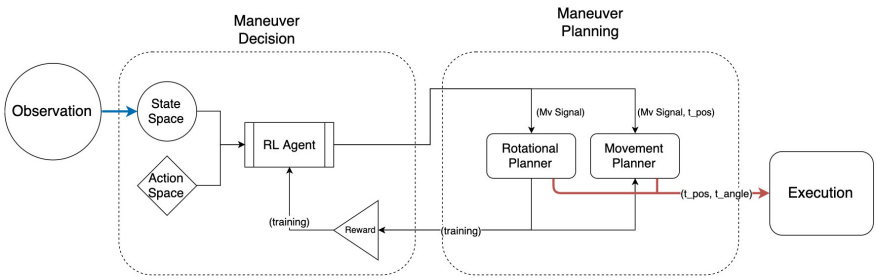


Figure 5: Decision pipeline. Pipes for training included.

1.3.2 Shooting

To make a shooting decision, we need distributions of two parameters: damage probability (the chance that each shot deals damage to a detected enemy armor plate in a given position) and bullet initial velocity. To determine these two parameters, we will conduct the shooting experiment specified below: For a set of distances, for a set of angles, place the enemy armor plate at the distance and angle. Aim the gimbal at the center of the enemy plate. Count shots that dealt damage to the enemy robot (registered to referee system) out of 20 shots, and use the ratio as an estimate of damage probability in the specific setup. Fit Effectiveness to a distribution conditioned on distance and angle. Also read the bullet initial speeds and find their maximum.

The robot detector in Computer Vision provides the enemy robot pose. We can thus infer the pose of the armor plate that we detected. We then estimate the damage probability using our predefined distribution and multiply it by the damage on the armor plate to obtain expected damage dealt. We issue a shoot command if the expected damage dealt is above a threshold, which we will empirically determine later.

We override a shoot command if our robot will become overheated—that is, we override if current heat + maximum initial speed ≥ 120. Assuming the maximum initial speed is lower than $25m/s$, such a conservative policy still allows 5 shots per second per robot and 10 shots per second for robots under 400 HP. If we are able to control the initial speed of firing, we will simply keep it constant around $20m/s$ for simplicity.

1.3.3 Maneuver (Decision)

The maneuver decision is used to output waypoints for the robot to follow. We have decided to continue trying a deep reinforcement learning approach. We improve upon our OpenAI Gym simulator from last year by replacing dynamics with a Gazebo back-end, outputting */cmd_vel* from neural network to a cloned non-ROS version of ROS navigation for training.

Now, we define an RL problem for waypoint selection:

- State Space: a combination of: a) Two identically-structured (robot pos, robot angle, robot health, robot ammo), each tuple representing one robot; b) Two identically-structured (enemy pos, enemy health, enemy estimated ammo), each representing one enemy robot; c) Game meta information

- Action Space: A concatenation two of identically-structured tuples of (movement signal, target pos), each tuple representing one robot. a) Movement signal: 0 indicates staying at the same spot and the rotation planner acting. 1 indicates moving towards target pos and the rotation planner being inactive. b) Target pos: significant if and only if movement signal = 1. This is the coordinate we are feeding to the Movement Planner. c) Target angle: significant if and only if movement signal = 0. This is the angle that the Rotation Planner outputs.

- Reward: For every duration which we plan a new waypoint and actually get there in simulator, we gain positive reward proportional to the damage we deal to enemies and negative reward proportional to the damage enemies deal to us.

1.3.4 Maneuver (Planning)

The maneuver planner takes in planned waypoints from the RL decision and uses optimization and a decision tree to plan a path and rotations. Since the holonomic robot can disentangle rotation and translation, we can have separate planners.

- Movement Planner: Movement Planner takes in a target position, passed by the RL agent, and feeds that along with the field map to ROS Navigation. The Field map parameter is dynamically altered:

  (a) We place heavy weights on the active debuff region and emerging buff/debuff zones in the next 5 seconds.

  (b) We will experiment with two ways to compute the field map:

    (i) Dynamically compute the field map (a superposition of: fixed base field map, dynamic buff / debuff zones, dynamic robot positions) on each minute mark.

(ii) Precompute all possible layouts of the field map (a superposition of: fixed base field map, dynamic buff/debuff zones, and disregard robot positions). Even though buff/debuff zones are dynamically changing, their symmetrical pattern of emergence makes it possible to compute all possible layouts of the map that arise.

(c) Dynamic waypoint update: We choose the target position among the union of two sets of waypoints:

(i) Strategic locations: fixed coordinates that are made up of locations inside buff / debuff zones, near obstacles, as well as other points that capture the skeleton of the field's layout. These points will assist in global path planning.

(ii) Tactical locations: dynamic coordinates that enrich the area surrounding each robot. These points will assist local path planning under situations such as neck-to-neck close combat.

(iii) Further optimization: to even further limit the action space, we will experiment pruning the set of strategic locations. For example, if under current state, reaching point A has to pass through point B, then point A can be removed from the strategic locations, since a convergent RL agent is expected to choose B under current state, and choose A once the robot has reached B.

- Rotational Planner:

(a) If the RL agent signals a rotation, we will compute the target angle to turn. This can be deterministically calculated by these methods:

(i) A decision tree will optimize between expected damage dealt and expected damage taken when enemies are in range, or turn in preparation for shooting if no enemies are in range.

(ii) Nearby obstacles and gimbal rotation range will be plotted on a [0,2pi) domain. According to the function determined in the Shooting section, local optimum will be found that are within gimbal range, between obstacles and minimize expected damage by the enemy.

(b) We expect movement signal to be 1 under (but not limited to) these situations:

(i) If there's only one enemy around us and can only be included in our line of sight through turning

(ii) If we detect enemies behind us and we are not shooting enemies in front of us

(c) We expect movement signal to be 0 in (but not limited to) these situations:

(i) If it's impossible to include any enemy in our range of sight through rotation

(ii) If both enemies are around the robots and it's possible for us to escape. In such a case we should quickly move away.

(d) To ensure the RL agent only outputs the correct rotational signal, we penalize it if constraints described in (b), (c) are violated.

1.3.5 RL training

We will train our RL agent with curriculum learning followed by PPO. The process can be broken down into the following:

- Create a decision tree based on baseline agents, and use imitation learning to quickly pick up useful skills.
- Train against baseline strategies, employing small-scale game training to speed up the emergence of skills. Some proposed small-scale games:
  (a) Close quarter combat: A random square within the real playing field is set as game boundaries. Two opposing agents will cache reward by damaging their opponent but only earn this reward if it leaves the playing area. If the agent dies, it will gain no points.
  (b) Stationary Cover: Over the entire playing field, the opponent will be stationary but appear at random locations while the agent will act normally. The game terminates after set episode length. The agent will receive positive reward for damage done to the opponent and negative reward for damage taken. This encourages the agent to identify advantageous locations.
  (c) Unfair resources: In one version the agent will have a third of the real capacity of ammo, and in another version, the opponent will have a third. The game will proceed as normal.
- Train with a combination of self-play and human exploitation with PPO (proximal policy optimization).

1.3.6 Potential Challenges in Real-World Integration

Our approaches suffer from some drawbacks of ROS Navigation. ROS Navigation gives a segmented path which naive PID control does not follow well. Simulating movement with ROS Navigation also by default uses its built-in controller, which behaves differently from our own control scheme described in the following Executing section. ROS Navigation also ignores slipping on Mecanum wheels. These inaccuracies all contribute to the error between simulation environment and the real world.

Controls for Maneuver and Shooting are not fully orthogonal. Fast Movement and/or Rotation could hurt the accuracy of Shooting. We assume this to be a relatively small source of error.

The Separation approach limits our robots' mobility by forbidding Movement and Rotation at once. Hand-picked waypoints and discretized angle space could also limit their full potential. Such limitations will make our strategy suboptimal to an equivalent full-space strategy.

1.4 Execution

1.4.1 Chassis Control

The holonomic chassis will be controlled with PID with the data from the built-in encoders at 20Hz. We are also planning to change the source code of the STM32, decoupling the yaw control of the turret from the chassis, thus providing more passive defense options that are baked into the low-level controller.

1.4.2 Turret Control

In order to disentangle turret and chassis control, we changed source code from STM32 developer board to incorporate a self-stabilization control with offset from chassis control signals. Since we already have the 3D coordinate of the target armor plate from computer vision at high FPS, we can already have decent aiming result by pointing to that coordinate. However if we find shooting in motion be a problem, we calculate a trajectory offset. The calculation is shown below: The short-term layer will provide motion data, including movement and pose, and we can retrieve the current state via the onboard gyroscope and encoder. When the fire event is triggered, we need to calculate the pitch angle and lateral velocity, assuming an unaggressive rotation of the chassis. Let α_1 = our gimbal relative to the enemy armor, α_2 = enemy gimbal relative to our armor, we solve $\min(\frac{\pi}{2} - \alpha_1)^2 + \alpha_2^2$. We can calculate the offset using the information of enemy motion provided by the computer vision result in the IO layer. To get a more accurate result, we take the processing delay of the computer vision program into account and have added a proper offset to it.

1.5 Communication

In order to exchange complex data between two robots and outputs over WiFi, we will use ZeroMQ, which is similar to the DDS that ROS2 is built on. ZeroMQ can exchange data, such as text or images with much lower latency and higher bandwidth and reliability than ROS. By utilizing this module, we will have a superb communication protocol between the robots, the referee system, and the outposts. Each robot will individually send meta-information to our outpost computer, which will compute the action produced by the decision-making agent and feed that action back to both robots. Both robots are controlled by one single agent under this circumstance. We also implement separate agents on each robot to ensure robustness against unstable connection. When there is a stable communication between the robot and the outpost computer, the independent agent on each robot remains idle. However, once a robot detects that it has not received messages from the outpost computer for a certain period of time, it will activate its independent agent, which uses only information collected by its own sensors and make decisions independently, while attempting to reestablish communication with the outpost computer.

2 Hardware

2.1 Sensors

2.1.1 LiDAR

We are using the Slamtec RPLIDAR A1 for localization. It has a scanning frequency that is adjustable from 5 to 10 Hz, a sample rate of 8000 samples per second, 1 degree angular resolution, and up to a 12 meter range with 0.5 centimeter resolution. This range and resolution is enough for the field and they are cheap.

2.1.2 Robot Cameras

For our robot cameras, we are choosing to have 1 on the turret, 1 in front on the base, and 2 rear-facing. The front camera gives us reliable coverage of the 180° range of the turret, and the turret camera gives redundant coverage and aiming feedback control. The two rear-facing cameras will face diagonally to the left and right, and allow us to know if enemies are at our back, which is important since the back armor plate takes the most damage.

For the turret and front-facing cameras, we are choosing to use an Intel Realsense D435 and a Mynt Eye Standard stereo camera. These stereo cameras are good for their high framerates (90 fps, 60 fps respectively), field-of-views (95°, 146°), and depth-sensing range (0.1m to 10m, 0.5m to 18m). Furthermore, they both have global shutters, which will help us reduce distortion in images while moving and tracking moving objects.

We chose to use stereo cameras over monocular cameras for the front-facing cameras because we can use the depth information to get the 3D pose of enemy armor plates. This information is important for decision-making and shooting control, as the exact position of the plates can aid in the accuracy of their algorithms. With only the RGB image from a monocular camera, this information would have to be inferred by the size of the armor plate in the image, which can be heavily affected by the yaw orientation of the plates.

Since we cannot shoot robots until they are in front of us, we do not need depth information on the back cameras. Because of this, we chose the Logitech C270. It has decent framerate (30 fps), field-of-view (60°), and resolution (1280 by 720), which should be good enough for performing robot detection on.

2.1.3 Outpost Cameras

Our choice for our outpost cameras is the FLIR Pointgrey Blackfly BFS-U3-16S2C-CS sensor paired with the Computar A4Z2812CS-MPIR lens. We chose the sensor for its combination of high frame rate (up to 226fps) and high resolution (1440 by 1080), which will be necessary since it needs to be able to see across the whole field, which is nearly 9.5 meters from corner to corner. Furthermore, it has a global shutter, which has the advantage of reducing blur that a rolling shutter would have when capturing fast-moving objects, such as the competition's robots. We can also set white balance for these industry cameras so detection is stable. We chose the lens for its adjustable focal length, which allows us to have a variable field-of-view that ranges from 47° to 115°. This high FOV allows us to view the entire field without any corners out of our line-of-sight. In addition, the lens has an adjustable aperture, which

allows us to adjust for the bright lights coming from the robots' armor plates and health bars.

2.2 Computing Devices

2.2.1 Robot Computers

We plan on using the Intel NUCi5BEK as computing device on robots. It features an i5-8259U quad-core 8-thread CPU that boosts up to 3.80 GHz is much more powerful than the Jetson and Manifold, and has an Iris Plus for tasks that require graphics.

Although it lacks CUDA graphics, the NUC actually outperforms a Jetson on batch size 1 inference using Intel's OpenVINO on the Iris Plus Graphics 655, with 1.8 Tflops of FP16 compared to the 1.5 Tflops of FP32 on the Jetson. The Intel NUC also supports faster CPU instruction sets than the ARM processor found in the Jetson and Manifold. In particular, AVX and SIMD give significant performance boosts in OpenCV.

2.2.2 Outpost Computer

For our Sentry computing device, we are considering a laptop with an i7-9750H and an RTX 2060 (Laptop). This hardware configuration is a great option for us because of its combination of performance-for-price and portability. The 240 Tensor cores can be leveraged by NVIDIA TensorRT for faster TensorFlow inference, which we will be using for computer vision for outposts.

2.3 Communication

2.3.1 Sensor-Computer Communication

The sensor-computer communication on the robots will be done through the four USB 3.1 Gen 2 ports of the NUC which all support speeds of up to 10 Gbps, on top of extra ports from a Thunderbolt 3 hub that supports up to 40 Gbps. This is more than enough for the sensors we have onboard (LiDAR, 2 stereo cameras, 2 webcams). The sensor-computer communication on the sentries will be done over USB 3.1 Gen 1, which offers speeds of up to 5 Gbps. This is enough since the Blackfly BFS-U3 camera sensor only supports up to USB 3.1 Gen 1 speeds, not Gen 2 or Thunderbolt. Furthermore, our bottleneck for the sentries will likely be the data processing speeds, rather than the data transfer speed between the sentry cameras and the computing device.

2.3.2 Inter-Computer Communication

Inter-Computer Communication will be over WiFi, but with optimization as described in communication in software section.

References

Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points, 2019.