Artificial Intelligence & Machine Learning Final Project: Wumpus World 3D!

CS152, CS156, Yoav Rabinovich, December 2018

Introduction

In this assignment I implemented a simple 3D video game in C#. It simulates a robot controlled using PID, exploring the dungeon of Wumpus World using a camera, and playing the game. I've first simulated the world in 3D, then implemented the PID controller to control the robot and the ability for the player to set PID parameters while playing. Finally, I trained a convolutional neural network using TensorFlow in Python to allow the robot to translate his visual precept into a knowledge base about the features of each room it visits.

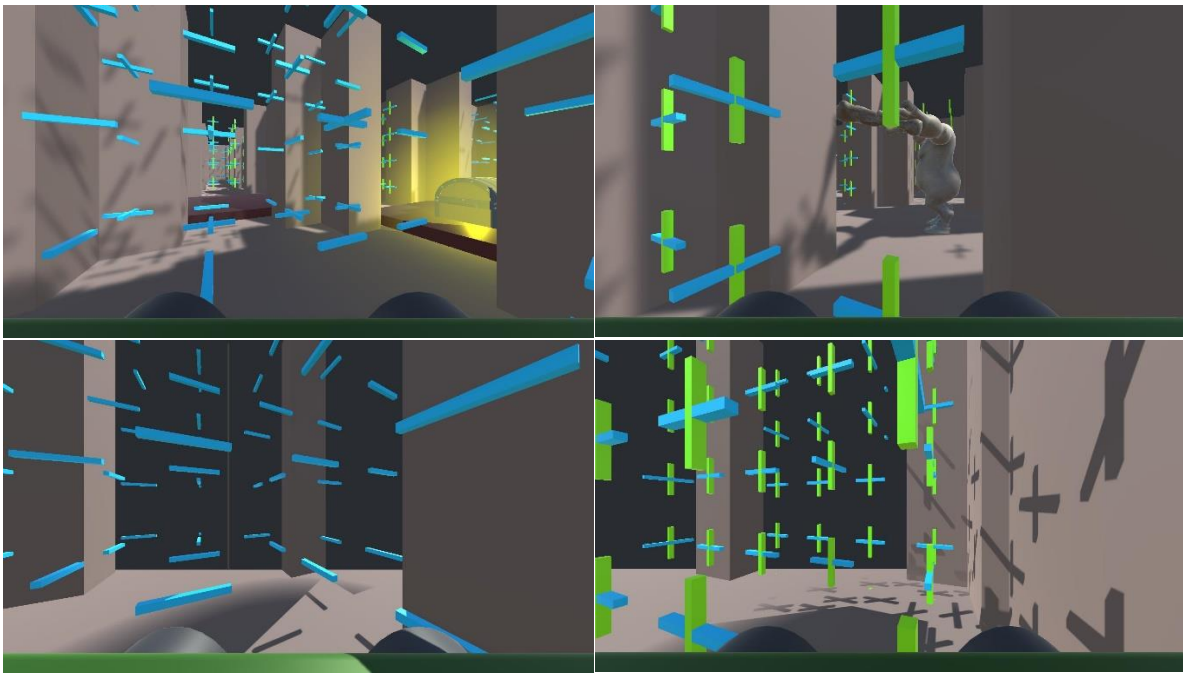

Figure 1. Screenshots from Wumpus World 3D! From top left: (1) A tile with a breeze between two pits. The treasure and glow are visible. (2) A tile with stench and a breeze. The Wumpus is visible. (3) A limited range view available to the player in a breezy tile. (4) A limited range view of a tile with stench and a breeze.

General

World Generation

I've used the Unity3D engine for handling graphics and physics and implemented *"GenerateBoard.cs"* that constructs an *n*n* board of a *Wumpus World 3D!* game using basic 3D models. The code generates tiles and walls to build the game world and populates it with the relevant game objects as well as a predetermined number of pits. It then appropriately generates visual cues to signal environmental effects to be interpreted as precepts by the computer vision network. There are also functions to convert between cartesian world-space and board coordinates.

CS152

Robot Control

Problem Definition: A simulated robot is faced with desired velocity input changing from moment to moment and has only limited thrust with which to respond. This is the common problem of many mobile robots which are controlled in real time: The operator thinks in terms of velocity while the robot has a mechanical motor capable only of exerting impulses of force. While a move-to-point feature hasn't been implemented in this case, the same problem applies to any robot using kinematic planning. Dynamic models for computing complex trajectories using forces is computationally infeasible for many use cases, which is why solutions are needed for kinematic planning.

The robot simulated has 6 degrees of freedom, able to apply forces horizontally and vertically as well as control yaw, and is equipped with a camera as a sole sensor apart from an ability to measure its own velocity.

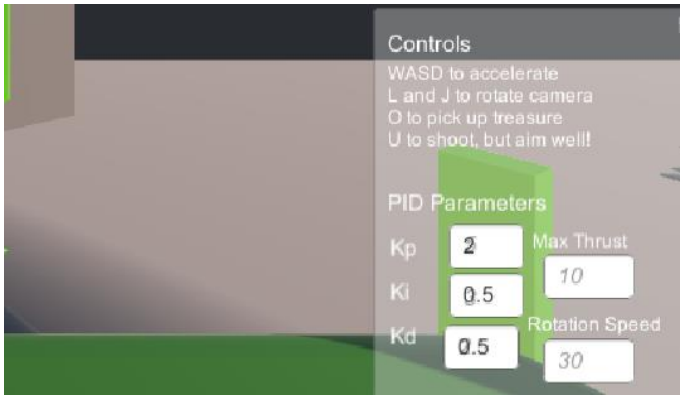

Figure 2. User input of PID parameters in Wumpus World 3D!, which is capable of updating live while playing.

Solution Specification: I've decided to implement a PID controller to control the robot in the simulation. "*RobotController.cs"* contains the entire system, capable of reading live input to change PID parameters and use them to calculate an output force vector for the motor to apply. The algorithm operates each frame of the simulation: First it reads a desired velocity from the controller input, and calculates the error between it and the robot's current velocity vector. It then computes an integral and derivative based on the time passed since the previous frame. Finally, it scales these values by the input weights and translates them to an output vector proportional to the available thrust of the motor. This code also includes game rules such as shooting, picking up treasure, falling in pits, dying to the Wumpus and winning the game, making *Wumpus World 3D!* a fully playable game.

$$u(t) = K_\mathrm{p} e(t) + K_\mathrm{i} \int_0^t e(t')\,dt' + K_\mathrm{d} \frac{de(t)}{dt}$$

Analysis of Solution: The parameters tune the behavior of the system, as can be experienced by running *Wumpus World 3D!* and exploring parameter values. Increasing proportional gain

decreases the rise time of the function, converging on the desired velocity faster, but makes the robot prone to overshooting. The derivative exists to decrease overshoot by considering the result of an impulse before it occurs. However, it builds up steady state error from the derivation process, which is helped by the integral gain which detects accumulated errors over time. A correct blend of the parameters depends on the system's specification and requirements and is left for the player of *Wumpus World 3D!* to find on their own.

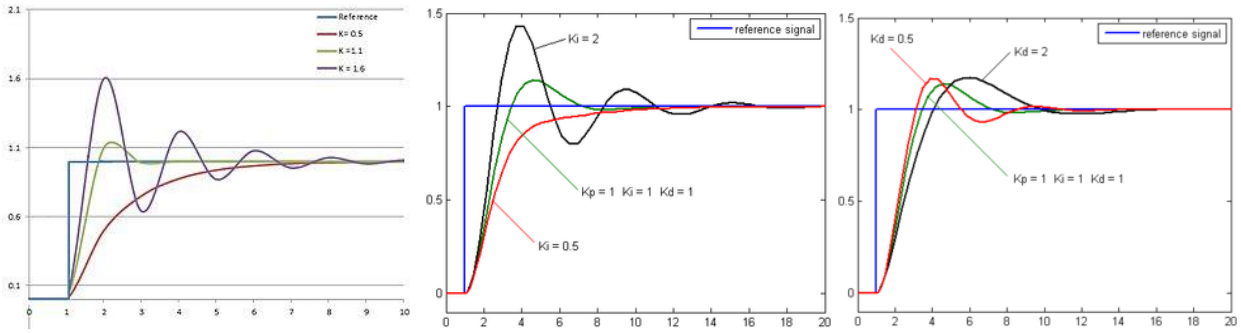

Figure 3. Effects of PID parameters with a set input signal. From the left: (1) An increase in Kp, shortening rise time and resulting in overshooting. (2) An increase in Ki, increasing precision but resulting in overshooting. (3) An increases in Kd, flattening overshooting but introducing steady state error.

CS156

Computer Vision:

Once the robot was operational in the game world, I adapted code to write "*Capture.cs*" to capture around 2500 thumbnail photos of multiple tile conditions. The code captures 108x192 screenshots from the camera on the robot, and the file name includes a binary representation of the classification labels present at the screenshot location: breeze, stench and glitter. Apart from choice of resolution (chosen to preserve "real Wumpus world" ratio) and converting them a 4D array, preprocessing was left to the convolutions. The labels were converted to integer form, since classifiers don't run into the problem of assuming relation between such labels. I have also opted out of performing explicit multilabel classification and pooled the different combinations into different labels. This was done to conserve time since 8 options is not too difficult for a classifier to distinguish.

I've constructed a convolutional neural network in "*ComputerVisionCNN.ipynb*" based on a TensorFlow suggested architecture. The network is deconstructed into several parts: first some custom convolutions, then two sections of mass-convolution layers and pooling to reduce dimensionality. Finally, a dense layer performs the classification based on the features highlighted by the convolution steps and a logits layer outputs 8 values.

My custom convolution layer was separated into 3 parallel columns and summed together at the end. I wanted to capture explicitly the insights from edge detection both vertical and horizontal as well as the overall structure of the image. This is because the visual cues I designed in Unity3D were intentionally stylized as vertical lines for stench, horizontal lines for breeze and bright color light for glitter, knowing that these simple convolutions would be able to detect the patterns.

The mass convolution uses many pre-made convolution kernels optimized by google to detect common patterns in images and didn't require input from me. The pooling layers served to reduce the dimensionality of the data, both to ease computation and to detect patterns at different levels.

Finally, the dense layer used TensorFlow's dropout: each feed forward or backward pass leaves out a proportion of neurons completely. This prevents neurons from developing dependency upon one another and prevents overfitting.

The evaluation was done using softmax with cross-entropy loss. Softmax converts outputs into a normalized array of probabilities for classification into classes.

$$\begin{bmatrix} P(t=1|\mathbf{z}) \\ \vdots \\ P(t=C|\mathbf{z}) \end{bmatrix} = \begin{bmatrix} \varsigma(\mathbf{z})_1 \\ \vdots \\ \varsigma(\mathbf{z})_C \end{bmatrix} = \frac{1}{\sum_{d=1}^{C} e^{z_d}} \begin{bmatrix} e^{z_1} \\ \vdots \\ e^{z_C} \end{bmatrix}$$

Cross-entropy is then used to evaluate the classification. We first compute the likelihood of the model parameters we want to maximize, $L(\theta|t,z) = P(t,z|\theta)=P(t|z,\theta)P(z|\theta)$. Fixing the parameters and ignoring the probability for output z, we get that the probability for each class given the outputs is equal to the cumulative multiplication of the predicted values for each class:

$$P(\mathbf{t}|\mathbf{z}) = \prod_{i=c}^{C} P(t_c|\mathbf{z})^{t_c} = \prod_{i=c}^{C} \varsigma(\mathbf{z})_c^{t_c} = \prod_{i=c}^{C} y_c^{t_c}$$

Now we can finally find the max likelihood (or negative min likelihood), which is done every back propagation setp:

$$-\log \mathcal{L}(\theta|\mathbf{t},\mathbf{z}) = \xi(\mathbf{t},\mathbf{z}) = -\log \prod_{i=c}^{C} y_c^{t_c} = \sum_{i=c}^{C} t_c \cdot \log(y_c)$$

I ran two models, each for around an hour of on a GPU, for 11000 forward and backward steps. Once using my custom convolutional layer section before the recommended architecture, and once without. Both models were able to reduce training loss to about 0.007, but when evaluated on my test set, the accuracy achieved was 93.2% for the default architecture, and only 82.6% on my enhanced version. This is probably because my simple convolutions were already included in TensorFlow's mass-convolution layers, and my attempt at giving them the same weight in training as the original picture masked much of the additional patterns that could be detected.

The Way Forward

Left out of this implementation is code to connect the TensorFlow agent to the robot, letting it evaluate its precepts by its camera feed, and ask the player how to act based on this approximation alone. This proved difficult due to compatibility issues between Unity3D, TensorFlow and the plugins used to make the two interact. However, I believe the sections relevant to AI and Machine

learning have been completed. Perhaps the feature will be included in a future downloadable content pack.

References

Build a Convolutional Neural Network using Estimators | TensorFlow. (2018). *TensorFlow*. Retrieved 22 December 2018, from https://www.tensorflow.org/tutorials/estimators/cnn

Minimize an error with a PID controller in Unity with C# - Stabilize a quadcopter | Habrador. (2018). *Habrador.com*. Retrieved 22 December 2018, from *https://www.habrador.com/tutorials/pid-controller/3-stabilize-quadcopter/*

How to save a picture (take screenshot) from a camera in game? - Unity Answers. (2018). Answers.unity.com. Retrieved 22 December 2018, from https://answers.unity.com/questions/22954/how-to-save-a-picture-take-screenshot-from-a-camer.html

Softmax classification with cross-entropy. (2018). *Peterroelants.github.io*. Retrieved 23 December 2018, from https://peterroelants.github.io/posts/cross-entropy-softmax/

Appendix: Project Proposal

Note: I have decided, based on teachers' advice and time constraints, to drop my ambitious inclusion of a Kalman Filter feature, as well as to change the product from a Wumpus solver to a fun Wumpus game. However, I believe I was still able to apply most of the proposed LOs.

Problem Definition: A simulation of a Wumpus World mobile robot solver, with only visual information to consider, and with simulated uncertainty in the execution of the motion plan.

Proposed Solution: I've already coded a simulation space in a 3D engine (Unity 3D), and set a camera on a mobile agent. I'll now use the neural network LOs (CS156) for computer vision, to process the visual frames into approximations of precepts for the location and features of the current square. I'll then use the #robotics (CS152) and relevant CS156 LOs and construct a PID controller as well as a Kalman Filter model to control the movement of the robot, which will be artificially tempered with by external force. Finally, a simple application of #search and #ailogic (NS152) will be used to construct a representation of the world map and solve the puzzle.

Deliverables: A program simulating a semi-realistic 3D Wumpus World and controls a robot agent to solve it, as well as a paper summarizing the work and including the requirements from both projects.