MAE 598: Final Project
Evaluating Efficiency improvements on RRT* using Deep Learning

June 2023

Yuanhao Chen
Jose Osechas
Instructor: Dr. Ronald Calhoun

Contents

List of Figures

GPT Prompt

"Using Matlab, create a path planning algorithm using RRT*, integrating a Deep Learning Structure in its sampling. Generate a dataset of 2000 random mazes of 35x35. Use the Navigation Toolbox and the Deep Learning Toolbox. Apply the trained Neural Network to control the distribution of the randomized samples (Uniform Samples vs Learned Samples) and evaluate the efficiency in Path Costs and Success Rate between paths with 0% Learned Samples and 50% Learned Samples"

1 Introduction

One of the key points when talking about new opportunities from autonomous vehicles is that riders would be able to use their commute time for various activities other than focusing on the road. This can be compared to how people who currently commute by train have the ability to use their time for reading, working or simply for entertainment. But a key advantage that mass transportation systems have over smaller vehicles is that their route is predetermined, with levels of comfort and accessibility being tied only to its design. AVs on the other hand, have to deal with changes in environmental conditions, moving and stationary obstacles, making optimal path planning an additional problem that needs to be solved to reach that level of commute freedom.

For this end, we can find several types of controllers and algorithms [3][5] that are used for path planning with key factors being safety (collision-free) and comfort (avoiding sudden movements). Just as with human drivers, both these factors require fast and predictive decision making, which becomes harder when dealing with higher speed scenarios and all the complex interactions that occur in real-life driving situations.

During our discussion of PID controllers in class, the idea of using neural networks in conjunction with a controller was brought up. Using the PID controller to prepare the training data would mean that it would not only be labeled but optimized, which in turn means that the efficiency of the neural network would be improved. Aiming to prove this, we set up a sample-based path-planning algorithm in Matlab and implemented its results to train a Neural Network to changed the sampling method. We then evaluated its efficiency, using a Success Rate threshold of 90% and observing Path Cost improvements when we use *Learned Samples*. Section 2 describes the algorithm used, *Rapidly-exploring random tree*, and which modifications were applied to it. Section 3 details the methods used, mainly which Matlab modules were used and the process to randomly generate the maps for the data-set. Section 4 summarizes the results and concludes.

2 Sampling-Based Path Planning Algorithm

Sampling-based path planners randomly connect points in the state space and construct graphs to create obstacle-free paths [1], with the difference being on how these graphs are generated. For this project, we used the *Rapidly-Exploring Random Tree* algorithm.

2.1 RRT

This type of data structure, first presented by LaValle in 1998[2], the basic RRT algorithm builds a tree outward from the start, the root of the tree, to explore the state space until it reaches the target (goal) [4]. Obstacles are not explicitly represented, but their regions are set to be avoided, making the vertices, and the edges between these, be set in the free space.

From the initial state, the algorithm samples a random vertex that is at a distance ρ then it selects an input that minimizes the distance between vertices. It continues exploring the space in this way until it reaches the goal or the number of iterations equals the threshold [4]. This method carries an advantage that the expansion of the RRT is heavily biased toward unexplored portions of the state space. [2].

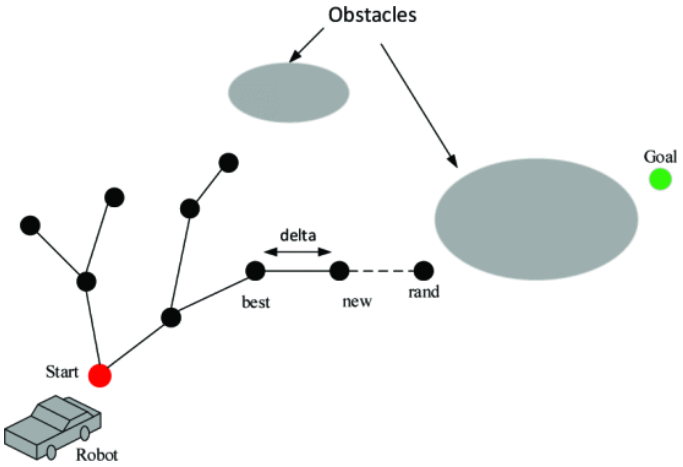


Figure 1: Basic Form of RRT[4]

2.2 RRT-Star

Even if the RRT algorithm is shown that is probabilistic complete, there is no feature that allows us to analyze the quality of the solution, especifically as a function of the number of samples. Through analysis of this algorithm, Karaman & Frazzoli were able to present a new algorithm RRT*, which can be proven to converge to the optimal solution [1].

At the moment RRT* connects the nearest node to a new sample, it checks for all other vertices in the vertex set V that are within a sphere of a defined radius and creates edges between them. The main difference is that after moving to the new vertex x_{new}, it will check all vertices in the set that are in the previous state, picking the one with the lowest cost and deleting all others. This way it maintains the tree structure in its most optimal way.

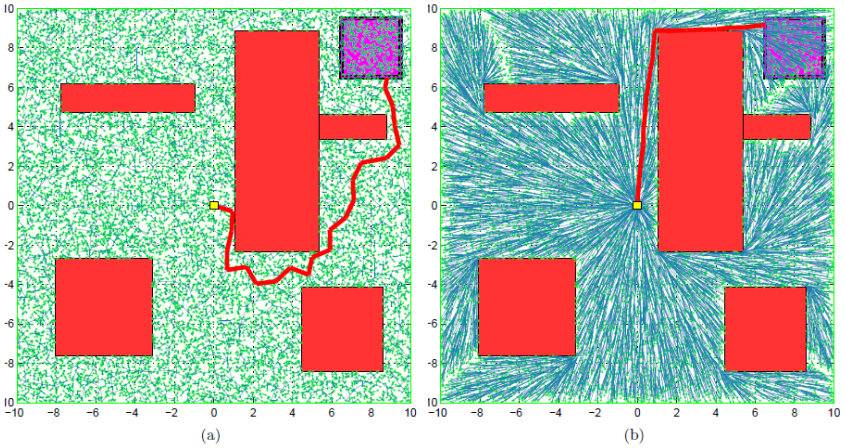


Figure 2: Comparison of RRT (a) and RRT* (b)[1]

3 Methodology

3.1 Generating Maps

In order to analyze the effects of deep-learning in path planning, we decided to use an algorithm to plan a path that solves randomly generated mazes. The mazes were set to be of $35m$ by $35m$ with a cell resolution of 2 cells per meter, a wall thickness of 1 cells and lanes of 5 cells. These *binaryOccupancyMaps* are randomly generated by setting a random seed and using the function mapMaze:

Listing 1: Code Snippet for map generation

```
% Set random seed
rng("default");
% Number of maps
numMaps = 2000;
% Map size in metres (assume height = weight)
mapSize = 10;
% Number of grid cells (assume height = weight)
gridSize = 35;
% Passage width in cells
passageWidth = 5;
% Wall thickness in cells
wallThickness = 1;
% Map resolution (cells per meter)
mapRes = gridSize/mapSize;
% Generate maps
for k=1:numMaps
    maps{k} = mapMaze(passageWidth, wallThickness,...
        MapSize=[mapSize,mapSize], MapResolution=mapRes);
end
```

A key step in the map generation process is that, to ensure there are no collisions, we would need to consider the radius of the robot and we *inflate* the walls of the maze. This limits the path generated so the obstacles are completely avoided.

From the MATLAB Navigation toolbox, we use the *plannerRRTstar* function, which ensures that the path converges to an optimal solution:

Listing 2: RRTstar Planner

```
    planner = plannerRRTStar(stateSpace, stateValidator);
    planner.ContinueAfterGoalReached = true; % optimize
    planner.MaxConnectionDistance = 1;
    planner.GoalReachedFcn = @examplerHelperCheckIfGoalReached;
    planner.MaxIterations = 2000;
```

This path planner will obtain the optimal path from Start to Goal states, if it is reachable within the max iterations set.

3.2 Preparing Data

Taking advantage that maps are usually relatively sparse, the data is compressed using the *trainAutoencoder* 4 function from the MATLAB Deep Learning Toolbox, which will help in the computational load when training the network.

Afterwards, the scaling used for the dataset is set to the ranges of $[0, 1]$ and $[-1, 1]$. This data then needs to be divided in multiple dependent sets, making sure that they are well dispersed. We also split the dataset, to store a subset which will be used for testing the trained network.

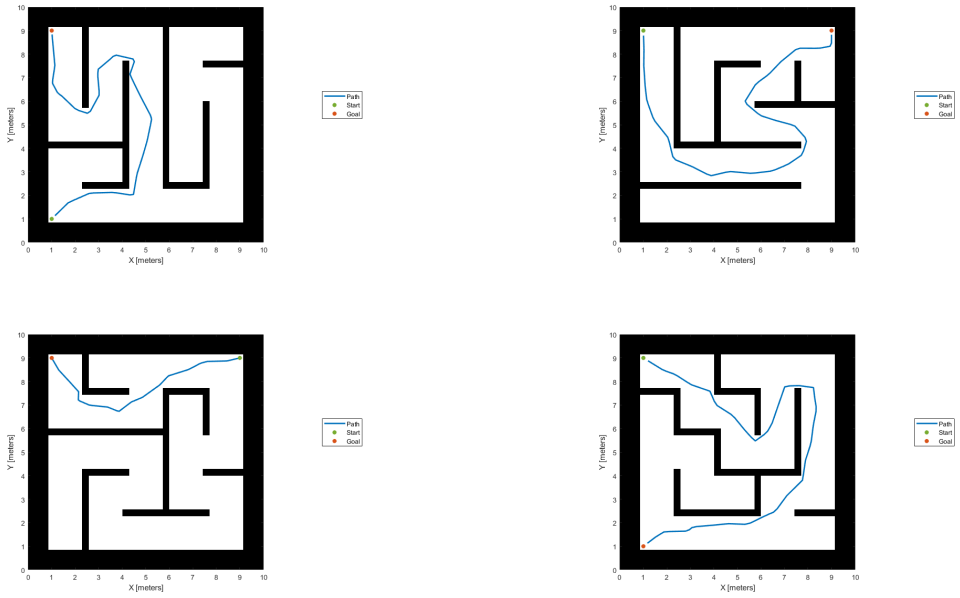


Figure 3: Example of solved mazes

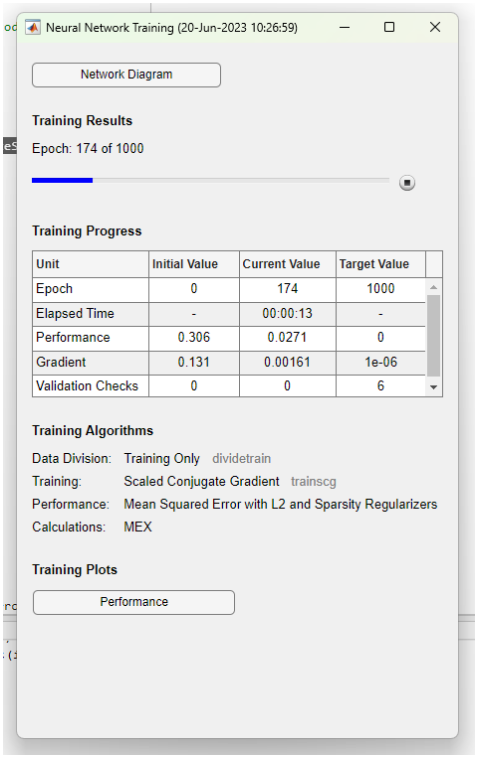


Figure 4: Compressing Map Data

3.3 Defining Network Architecture

For this step, a template for a Conditional Variational Autoencoder (CVAE) network is picked from MATLAB's toolbox. This is a variationon a Variational Autoencoder which takes an additional input called "condition", so that the data is generated from a conditional probability distribution.

The CVAE works differently during training and prediction phases. For training, the encoder takes an input state and an input condition to compute the latent state. This feeds into the decoder which will use the latent state to calculate the predicted state: During this phase, losses are calculated as well, trying to match the divergence loss

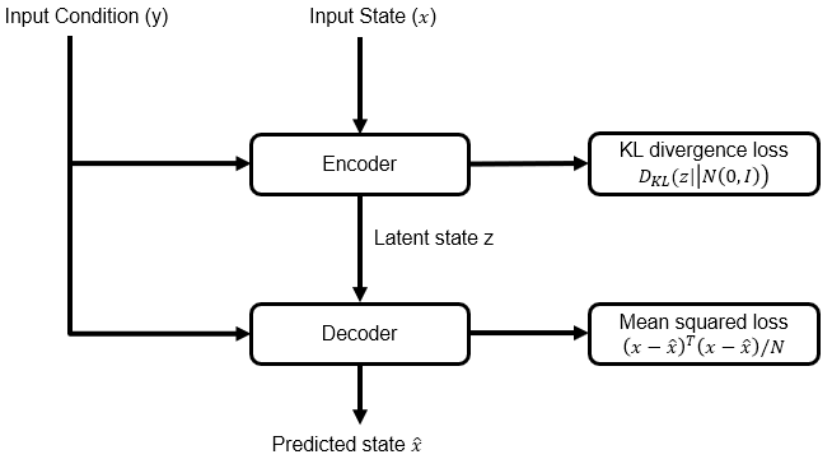


Figure 5: Training Phase Schematic

to the normal distribution $N(0, I)$ of the latent state. The mean squared loss is used to try to match the predicted state to the input state, so the data is labelled correctly.

For the prediction phase, only the decoder is used. It uses the normal distribution $N(0, I)$, the input condition and the input latent state. This gives us the predicted state, using the previously learned samples:

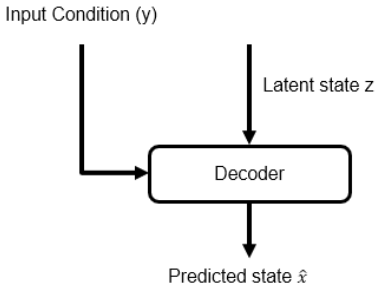


Figure 6: Prediction Phase Schematic

Both encoder and decoder network use fully connected layers with ReLU activation function and dropout layers in between.

Listing 3: Encoder Network Creation

```
% Hidden sizes of fully connected layers in the encoder network
encoderHiddenSizes = [512, 512];
% Probability values for the dropout layers
prob = [0.10, 0.01];
```

```
% Create layers
encoderLayers = featureInputLayer(numDependentSets*stateSize+conditionSize, Name
    ="encoderInput");
for k=1:length(encoderHiddenSizes)
    encoderLayers(end+1) = fullyConnectedLayer(encoderHiddenSizes(k)); %#ok<*
        SAGROW>
    encoderLayers(end+1) = reluLayer;
    encoderLayers(end+1) = dropoutLayer(prob(k));
end
encoderLayers(end+1) = fullyConnectedLayer(2*latentStateSize);
encoderLayers(end+1) = exampleHelperSamplingLayer(Name="encoderOutput");
% Create layer graph and dlnetwork object
encoderGraph = layerGraph(encoderLayers);
% Create this network only when doTraining=true
if doTraining
    encoderNet = dlnetwork(encoderGraph);
end
```

Listing 4: Decoder Code Creation

```
% Hidden sizes of fully connected layers in the decoder network
decoderHiddenSizes = [512 512];
% Probability values for the dropout layers
prob = [0.10 0.01];
% Create layers
decoderLayers = featureInputLayer(conditionSize+latentStateSize,Name="
    decoderInput");
for k=1:length(decoderHiddenSizes)
    decoderLayers(end+1) = fullyConnectedLayer(decoderHiddenSizes(k)); %#ok<*
        SAGROW>
    decoderLayers(end+1) = reluLayer;
    decoderLayers(end+1) = dropoutLayer(prob(k));
end
decoderLayers(end+1) = fullyConnectedLayer(numDependentSets*stateSize,Name="
    decoderOutput");
% Create layer graph
decoderGraph = layerGraph(decoderLayers);
% Create this network only when doTraining=true
if doTraining
    decoderNet = dlnetwork(decoderGraph);
end
```

3.4 Training Network

For training this network, the following parameters are used:

- Number of epochs: 100
- Mini-batch size for training: 32
- Learning Rate: $1x10^{-3}$
- Beta weight for KL divergence loss: $1x10^{-4}$
- Weight for mean squared error loss: $[1, 1, 0.1]$

Listing 5: Training Network

```
% For reproducibility
rng("default")
% Create mini-batch queue for training
trainData = combine(arrayDatastore(trainCondition),arrayDatastore(
    trainStates));
mbqTrain = minibatchqueue(trainData,MiniBatchSize=options.TrainBatchSize,
    ...
                OutputAsDlarray=[1,1],MiniBatchFormat={'BC','BC'});
% Train the CVAE sampler model
figure(Name="Training Loss");
[encoderNet,decoderNet] = exampleHelperTrainCVAESampler(encoderNet,
    decoderNet, ...
         @lossCVAESampler,mbqTrain, ...
                              options);
```

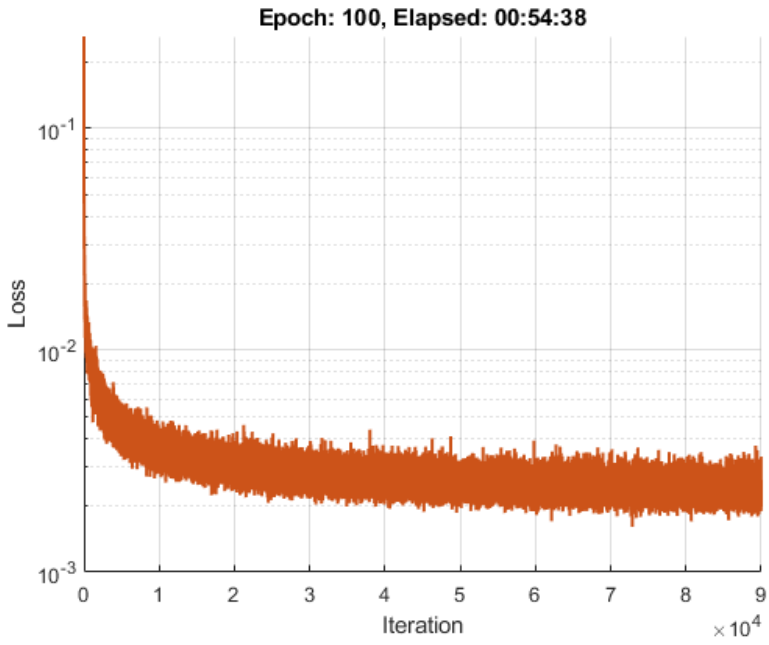


Figure 7: Loss Calculation during training

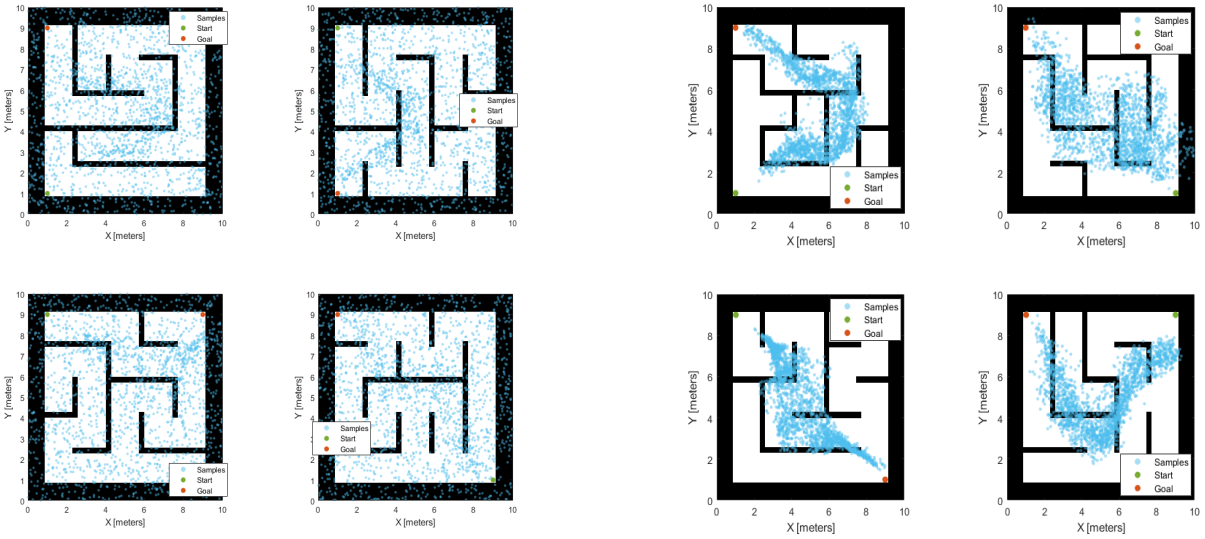
3.5 Prediction

After training, the subset we set aside was used for prediction testing purposes.

Listing 6: Prediction Code

```
% For reproducibility
rng("default")
% Prepare test mini-batches
testData = combine(arrayDatastore(testCondition),arrayDatastore(testStates));
mbqTest = minibatchqueue(testData, MiniBatchSize=1,...
                        OutputAsDlarray=[1,1],MiniBatchFormat={'BC','BSC'});
shuffle(mbqTest)
```

We can then generate the paths using a combination of uniform and learned samples, with uniform samples coming straight from a pure RRT* algorithm and learned samples from the trained NN. By setting λ as the proportion between uniform and learned samples, we can control the distribution of the vertices.



(a) Prediction with 80%-20% distribution

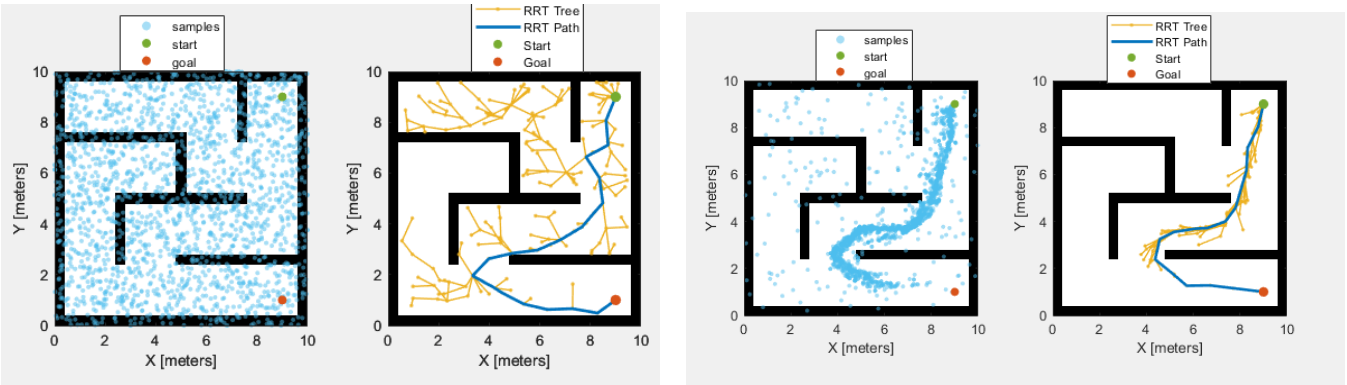
(b) Prediction with 100% learned samples

Figure 8: Comparison between different distributions

3.6 Path Planning with Improved RRT

With a trained model that gives us learned samples, we applied the original path-planning RRT* method to the testing subset. For this, we redefined the original State Space as a Custom State Space, adding the variables created in the NN process, most important being the addition of *Lambda* as a method so we can control the distribution of Learned vs Uniform samples.

Running the RRT* planner will create the most optimal path, as it did before, but having more control of the distribution of the random samples directly affects its efficiency.



(a) RRT* with Uniform Sampling

(b) RRT* with Learned Sampling

Figure 9: Uniform vs Learned Sampling Paths

3.7 Evaluation

In order to have measurable metrics, we evaluated the results by calculating the *Success Rate* and the *Path Costs* with a small set of 5 maps. We also selected maps from the test data that have at least two left/right turns to have a marked difference between learned and uniform sampling, due to that type being the most complex to solve for uniform sampling.

Due to the random nature of the algorithm, predicted paths will have slight differences, even if the path is optimized. Because of this, we set the number of runs to 100, so each map would be solved 100 times.

The success rate is defined as the fraction of total runs for which the paths are found, which was used to compare for lambda values of $\lambda = 0$ and $\lambda = 0.5$. For Path Costs, we compared the average values per 100 runs, between the same two distributions. We also set a *Max Connection Distance*, to limit the segments of the paths between sample points.

Listing 7: Calculating Success Rates

```
% Set optimize to false to exit from planner if path found.
optimize = false;

maxIterations = [5:50:500 1000:500:2500];
maxConnectionDistance = 1;

lambda = 0.5;
nRunTime = 100;

seed = 100;

% ...

for i=1:numel(maxIterations)
    for j=1:nRunTime
        [successRateLearned(j,i),successRateUniformed(j,i)] =
           exampleHelperSuccessRateComputation(...
            maps,optimize,maxConnectionDistance,maxIterations(i),startStates,
               goalStates,network,lambda,seed+j*10);
    end
end
```

Listing 8: Calculating Path Costs

```
% Set optimize to false to exit from planner if path found.
optimize = true;

maxIterations = 500:500:2500;
maxConnectionDistance = 1;

lambda = 0.5;
nRunTime = 2;

seed = 100;

%...

for i=1:numel(maxIterations)
    for j=1:nRunTime
        [pathCostLrndOpt(j,i),pathCostUniOpt(j,i)] =
           exampleHelperPathCostComputation(maps,optimize, ...
```

```
                maxConnectionDistance,maxIterations(i),startStates,goalStates,
                    network,lambda,seed+j*10);
    end
end
```

Of note, the value for *optimize* when calculating Path Costs was set to true. This way we ensured that the path keeps getting optimized for the fixed number of samples, even if the goal is reached earlier.

4 Results

We can see how for untested maps in the dataset, the distribution of the samples can be modified by changing the value of λ. This in turn allows us to set up more efficient path-planning, which reduces the computational load.

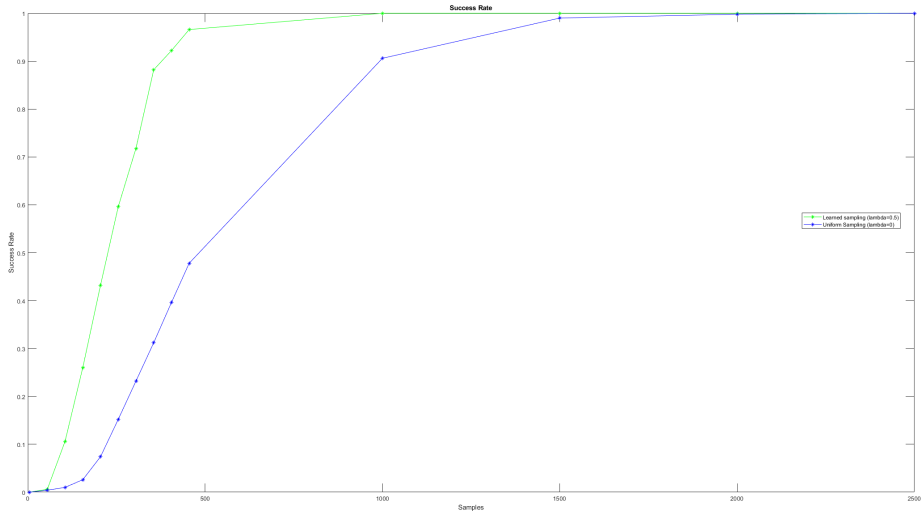


Figure 10: Success Rate Comparison

We can see how by using learned sampling at $\lambda = 0.5$ we can find a solution for 90 out of 100 runs with ≈ 400 samples. Using uniform sampling requires > 1000 runs to reach that level of accuracy.

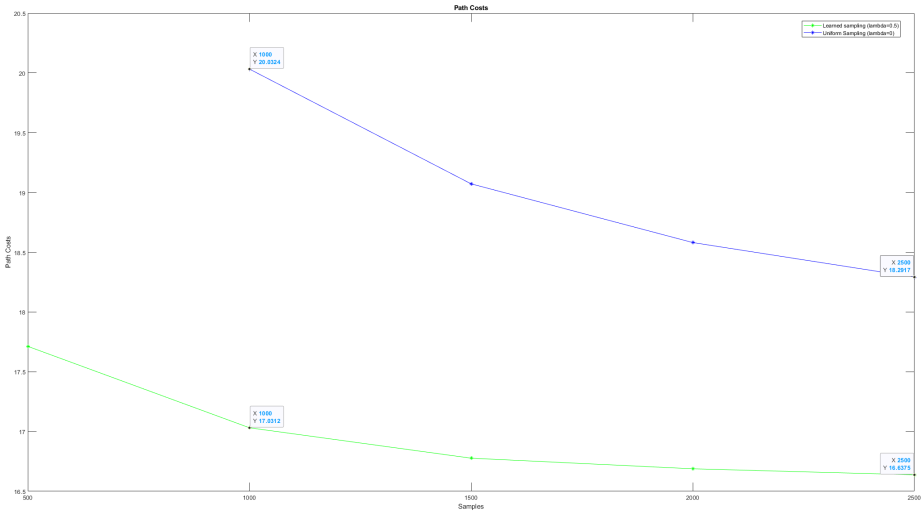


Figure 11: Path Cost Comparison

Between Path Costs, similarly matching what we see in the success rates, we can only compare when handling more than 1000 samples, due to uniform sampling having such a low success rate with lower sample numbers. There is a marked reduction of path costs by $\approx 15\%$ when using $\lambda = 0.5$. We estimate that with larger training sets, or having a more biased distribution towards the learned samples, these costs will be lower.

5 Conclusion

The goal of this project was to evaluate how applying deep learning into various parts of algorithms can be an improvement in their efficiency. Just as it was discussed in class, starting with a base of a path-planning algorithm, we feed the information into a neural network during its training phase to then allow it to make learned choices. We focused the deep learning aspect into obtaining a more pinpointed sampling, due to the nature of the RRT* algorithm, but this can be used with other types of controllers and models, since they would allow the Neural Network to receive highly accurate labeled data, which in turn will improve its efficiency.

References

[1] Sertac Karaman and Emilio Frazzoli. "Sampling-based algorithms for optimal motion planning". In: *The International Journal of Robotics Research* 30.7 (2011), pp. 846–894. DOI: 10.1177/0278364911406761. eprint: https://doi.org/10.1177/0278364911406761. URL: https://doi.org/10.1177/0278364911406761.

[2] Steen M LaValle. "Rapidly-Exploring Random Trees: A New Tool for Path Planning". In: *Computer Science Dept.* (). URL: http://lavalle.pl/papers/Lav98c.pdf.

[3] Jiangdong Liao et al. "Decision-Making Strategy on Highway for Autonomous Vehicles Using Deep Reinforcement Learning". In: *IEEE Access* 8 (2020), pp. 177804–177814. DOI: 10.1109/ACCESS.2020.3022755.

[4] Aisha Muhammad et al. "Simulation Performance Comparison of A*, GLS, RRT and PRM Path Planning Algorithms". In: *2022 IEEE 12th Symposium on Computer Applications & Industrial Electronics (ISCAIE)*. 2022, pp. 258–263. DOI: 10.1109/ISCAIE54458.2022.9794473.

[5] Dong-Sung Pae et al. "Path Planning Based on Obstacle-Dependent Gaussian Model Predictive Control for Autonomous Driving". In: *Applied Sciences* 11.8 (2021). ISSN: 2076-3417. DOI: 10.3390/app11083703. URL: https://www.mdpi.com/2076-3417/11/8/3703.

6 Appendices

6.1 Map Generating Code

Listing 9: Map Generating Code

```
%% Generate maps
% load("MazeMapDataset","dataset","mapParams")
% Set random seed
rng("default");

% Number of maps
numMaps = 1;
% Maze map parameters
mapSize = 10; % Map size in meters (assume height = weight)
gridSize = 45; % Number of grid cells (assume height = weight)
passageWidth = 5; % in cells
wallThickness = 1; % in cells
mapRes = gridSize/mapSize; % map resolution (cells per meter)
% Generate maps
for k=1:numMaps
    maps{k} = mapMaze(passageWidth,wallThickness, ...
                      MapSize=[mapSize,mapSize], ...
                      MapResolution=mapRes);
    mapMatrix = maps{k};

    % Create state space
    stateSpace = stateSpaceSE2([-1, 1; -1, 1; -pi, pi]);

    % Create binary occupancy map
    map = binaryOccupancyMap(mapMatrix, mapRes);

    % Create state validator
    stateValidator = validatorOccupancyMap(stateSpace);
    stateValidator.Map = map;
    stateValidator.ValidationDistance = 0.01;

    % Create planner
    planner = plannerRRTStar(stateSpace, stateValidator);
    planner.ContinueAfterGoalReached = true; % optimize
    planner.MaxConnectionDistance = 1;
    planner.GoalReachedFcn = @examplerHelperCheckIfGoalReached;
    planner.MaxIterations = 2000;
end
% Randomly sample two different start and goal states from this
startGoalStates = [1, 1, 0;
                   9, 9, 0;
                   9, 1, 0;
                   1, 9, 0];
show(map)
% save(fullfile(getenv('HOMEPATH'), 'Desktop', 'maps.mat'), 'maps');
```

6.2 Neural Network Code

Listing 10: Neural Network Code

```
% doTraining=true;
% mapsFolder = fullfile(getenv('USERPROFILE'), 'Desktop', 'map3');
doTraining=false;
if ~doTraining
    load("CVAESamplerTrainedModel","encoderNet","decoderNet")
end
load("MazeMapDataset","dataset","mapParams")
% load("maps.mat","maps")

%% Load the maps from the "map3" folder
%{
filePaths = dir(fullfile(mapsFolder, '*.png'));  % Adjust the file extension if
   your maps are not PNG files
numMaps = length(filePaths);

% Initialize the dataset
dataset = struct('maps', [], 'startStates', [], 'goalStates', [], 'pathStates',
   []);

for i = 1:numMaps
    % Load the map
    mapImage = imread(fullfile(mapsFolder, filePaths(i).name));
    % Convert the map image to a binary matrix
    mapMatrix = imbinarize(rgb2gray(mapImage));
    % Flip the map matrix vertically because images are read from top to bottom,
    % but matrices are indexed from bottom to top
    mapMatrix = flipud(mapMatrix);

    % TODO: Set the start and goal states for each map
    startState = [1, 1];  % Placeholder, adjust as needed
    goalState = [size(mapMatrix, 2), size(mapMatrix, 1)];  % Placeholder, adjust
        as needed
    pathState = [startState; goalState]; % Placeholder, adjust as needed

    % Add the map and states to the dataset
    dataset(i).maps = mapMatrix;
    dataset(i).startStates = startState;
    dataset(i).goalStates = goalState;
    dataset(i).pathStates = pathState;
end
%}
%% Continue with existing code to display maps and train the model
figure
for i=1:4
    subplot(2,2,i)
    % Select a random map
    ind = randi(length(dataset));
    exampleHelperPlotData(dataset(ind).maps,dataset(ind).startStates,dataset(ind
       ).goalStates, ...
                          navPath(stateSpaceSE2,dataset(ind).pathStates));
end
load("MazeMapAutoencoder","mapsAE")
split = 0.9;
```

```
numDependentSets = 5;
[trainCondition,trainStates,testCondition,testStates] = exampleHelperProcessData
    (dataset,mapsAE,numDependentSets,split);
stateSize = 3;
workspaceSize = 50;
latentStateSize = 4;
conditionSize = workspaceSize + 2*stateSize;
% Hidden sizes of fully connected layers in the encoder network
encoderHiddenSizes = [512, 512];
% Probability values for the dropout layers
prob = [0.10, 0.01];
% Create layers
encoderLayers = featureInputLayer(numDependentSets*stateSize+conditionSize, Name
    ="encoderInput");
for k=1:length(encoderHiddenSizes)
    encoderLayers(end+1) = fullyConnectedLayer(encoderHiddenSizes(k)); %#ok<*
        SAGROW>
    encoderLayers(end+1) = reluLayer;
    encoderLayers(end+1) = dropoutLayer(prob(k));
end
encoderLayers(end+1) = fullyConnectedLayer(2*latentStateSize);
encoderLayers(end+1) = exampleHelperSamplingLayer(Name="encoderOutput");
% Create layer graph and dlnetwork object
encoderGraph = layerGraph(encoderLayers);
% Create this network only when doTraining=true
if doTraining
    encoderNet = dlnetwork(encoderGraph);
end
% Hidden sizes of fully connected layers in the decoder network
decoderHiddenSizes = [512 512];
% Probability values for the dropout layers
prob = [0.10 0.01];
% Create layers
decoderLayers = featureInputLayer(conditionSize+latentStateSize,Name="
    decoderInput");
for k=1:length(decoderHiddenSizes)
    decoderLayers(end+1) = fullyConnectedLayer(decoderHiddenSizes(k)); %#ok<*
        SAGROW>
    decoderLayers(end+1) = reluLayer;
    decoderLayers(end+1) = dropoutLayer(prob(k));
end
decoderLayers(end+1) = fullyConnectedLayer(numDependentSets*stateSize,Name="
    decoderOutput");
% Create layer graph
decoderGraph = layerGraph(decoderLayers);
% Create this network only when doTraining=true
if doTraining
    decoderNet = dlnetwork(decoderGraph);
end
options = struct;
options.NumEpochs = 100;
options.TrainBatchSize = 32;
options.LearningRate = 1e-3;
options.Beta = 1e-4;
```

```
options.Weight = [1,1,0.1];
if doTraining
    % For reproducibility
    rng("default")
    % Create mini-batch queue for training
    trainData = combine(arrayDatastore(trainCondition),arrayDatastore(
       trainStates));
    mbqTrain = minibatchqueue(trainData,MiniBatchSize=options.TrainBatchSize,
       ...
                              OutputAsDlarray=[1,1],MiniBatchFormat={'BC','BC'})
                                 ;
    % Train the CVAE sampler model
    figure(Name="Training Loss");
    [encoderNet,decoderNet] = exampleHelperTrainCVAESampler(encoderNet,
       decoderNet, ...
                                                            @lossCVAESampler,
                                                               mbqTrain, ...
                                                            options);
end
% For reproducibility
rng("default")
% Prepare test mini-batches
testData = combine(arrayDatastore(testCondition),arrayDatastore(testStates));
mbqTest = minibatchqueue(testData, MiniBatchSize=1,...
                         OutputAsDlarray=[1,1],MiniBatchFormat={'BC','BSC'});
shuffle(mbqTest)
% Press Run button to visualize results for new maps
% RUN Botton here

%% Vary lambda to visualize results for different ratios of learned samples to
   total samples
lambda = 1; % From 0 to 1

% Number of samples to be generated
numSamples = 500; % It was 2000

if ~hasdata(mbqTest)
    reset(mbqTest)
end

% Generate samples for different test maps
figure(Name="Prediction");
for k = 1:4
    [mapMatrix,start,goal,statesLearned] = exampleHelperGenerateLearnedSamples(
       encoderNet, ...
                                           decoderNet,mapsAE,mbqTest,
                                              numDependentSets, ...
                                           mapParams.mapSize,numSamples,
                                              lambda);
    % Visualize the samples
    map = binaryOccupancyMap(mapMatrix,mapParams.mapRes);
    subplot(2,2,k)
    exampleHelperPlotData(map,start,goal,statesLearned);
end
```

```
function [loss,gradientsEncoder,gradientsDecoder] = lossCVAESampler(encoderNet,
    decoderNet,condition,state,beta,weight)
% lossCVAESampler Define losses for the CVAE network

% Predict latent states from encoder
[z,zMean,zLogVarSq] = forward(encoderNet,vertcat(state,condition));

% Predict state from decoder
statePred = forward(decoderNet,vertcat(condition,z));

%% KL diveregence loss
klloss = exp(zLogVarSq) + zMean.^2 - zLogVarSq -1;
% Reduce sum over zdim
klloss = sum(klloss,1);
% Reduce mean over batch
klloss = mean(klloss);
% Weighting term for KL loss
klloss = klloss*beta;

%% Reconstruction loss
reconLoss = (state-statePred).^2;
% Apply weight vector to state vector
numSets = size(reconLoss,1)/length(weight);
weight = repmat(weight,numSets,1);
reconLoss = reconLoss.* weight;
% Reduce mean over batches
reconloss = mean(reconLoss,1);
% Reduce mean over state vector dimensions
reconloss = mean(reconloss);

% Total loss
loss = klloss + reconloss;

% Gradients
[gradientsEncoder,gradientsDecoder] = dlgradient(loss,encoderNet.Learnables,
    decoderNet.Learnables);

% Convert loss to double
loss = double(loss);

end
```

6.3 RRT* Maze Solver

Listing 11: RRT* Maze Solver

```
folderName = fullfile(getenv('USERPROFILE'), 'Desktop', 'map3');
if ~exist(folderName, 'dir')
    mkdir(folderName);
end

startPositions = zeros(500, 2);
goalPositions = zeros(500, 2);
```

```
% Preprocessing and data preparation (map generation, resizing, etc.)
% Add your map generation and preprocessing logic here

% Continue with your training code, use trainCondition and trainStates
% Add your network training logic here

% Substitute the "dataset" variable in your existing code with your own data
dataset = struct('trainCondition', trainCondition, 'trainStates', trainStates);

% Load pre-trained model and autoencoder
load("CVAESamplerTrainedModel","decoderNet")
load("MazeMapAutoencoder.mat","mapsAE")

figure(Name="Maps");
for i=1:4
    subplot(2,2,i)
    testInd = randi(size(dataset.trainCondition, 1));

    % Extract the map from the trainCondition
    mapData = dataset.trainCondition(testInd, 5:end);
    % Reshape and convert to logical
    mapData = reshape(mapData, [50, 50]);
    mapData = logical(mapData);

    % Create an occupancyMap object
    map = binaryOccupancyMap(mapData, 2); % 2 is the MapResolution you used when
        creating the maps

    % Extract start and goal positions
    start = dataset.trainCondition(testInd, 1:2);
    goal = dataset.trainCondition(testInd, 3:4);

    exampleHelperPlotData(map, start.*25, goal.*25);  % Scale the start and goal
        positions by the map resolution
end

% Load autoencoder network that encodes maze maps
load("MazeMapAutoencoder.mat","mapsAE")

% Prepare network
network = struct("DecoderNet", decoderNet, "MapsAutoEncoder", mapsAE);

% Select test data index (1-500)
testInd = randi(size(dataset.trainCondition, 1));

% Get map, start, goal for current test index
mapData = dataset.trainCondition(testInd, 5:end);
mapData = reshape(mapData, [50, 50]);
mapData = logical(mapData);
map = binaryOccupancyMap(mapData, 2); % 2 is the MapResolution you used when
   creating the maps
start = dataset.trainCondition(testInd, 1:2);
goal = dataset.trainCondition(testInd, 3:4);
```

```
% Select the learned sampling proportion 0 = pure uniform, 1 = pure learned
lambda = 1;

%Set max Iterations
maxIters = 1000;
% Set random seed
rng("default");

% Create ExampleHelperCustomStateSpaceSE2
customSE2 = ExampleHelperCustomStateSpaceSE2(start.*25,goal.*25,map,maxIters,
   network); % Scale the start and goal positions by the map resolution
customSE2.StateBounds = [map.XWorldLimits; map.YWorldLimits; [-pi, pi]];
customSE2.Lambda = lambda;

% Create stateValidator
sv = validatorOccupancyMap(customSE2);
sv.Map = map;
sv.ValidationDistance = 0.1;

% Create plannerRRTStar
planner = plannerRRTStar(customSE2,sv);
planner.MaxConnectionDistance = 1;
planner.MaxIterations = maxIters;

% Run the planner
[pathObj, solnInfo] = plan(planner, start.*25, goal.*25); % Scale the start and
   goal positions by the map resolution

% Visualize the results
figure(Name="RRT results")
exampleHelperPlotData(map, start.*25, goal.*25, pathObj, solnInfo);  % Scale the
    start and goal positions by the map resolution
```