UNIVERSITY OF CALIFORNIA,
IRVINE

Optimal Shortest-Distance Motion Planning through a Field of Circular Obstacles: A Classifier-Enhanced Approach

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Mechanical Engineering

by

Robert Palfini

Thesis Committee:
Associate Professor Solmaz Kia, Chair
Professor Athanasios Sideris
Associate Professor Yasser Shoukry
Assistant Professor of Teaching David Copp

2023

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express great thanks to my advisor, Professor Solmaz Kia, for giving me the opportunity to do this work. Her guidance and enthusiastic support throughout the project helped make this result possible.

I would like to thank my committee for their guidance and support in my research work.

I wish to thank Changwei Chen, and my lab members for their help and insight into this project, and patience in answering my questions.

I wish to thank my wife, Eve, for her continuous, loving support that helped me get this done.

Finally, I wish to thank my parents and parents-in-law who's love and support helped me throughout my study.

ABSTRACT OF THE THESIS

Optimal Shortest-Distance Motion Planning through a Field of Circular Obstacles: A Classifier-Enhanced Approach

By

Robert Palfini

Master of Science in Mechanical Engineering

University of California, Irvine, 2023

Associate Professor Solmaz Kia, Chair

Global shortest distance motion planning through a field of obstacles using gradient based optimization techniques is a nonlinear and non-convex problem which is very challenging to optimize. Exact motion planning techniques such as visibility graphs scale poorly with the number of obstacles, and sample based methods find optimal paths asymptotically with samples, making them not ideal for online planning applications. Inspired by the the fact that visibility graphs can find shortest path solutions in fields of circular obstacles, and that these solutions can be characterized by the direction of the path around the obstacles, we look to train an Artificial Neural Network to predict the initialization needed for locally converging to the global minimum in our shortest-distance optimization. We found this method works well for obstacle courses with low numbers of obstacles, achieving 90% test accuracy for unseen obstacle courses, but has issues scaling due to an exponential increase in the amount of data needed for generalization across the entire feature space. This technique is meant for online planning applications that require many path queries, in a configurable environment, as it leverages the O(1) time complexity of a classifier, combined with the interior point algorithm for finding optimal paths rapidly.

Chapter 1

Introduction

1.1 Challenges of Non-Convex Optimization

A common issue when solving nonlinear optimization problems, is when our problem formulation is non-convex. There are many challenges associated with non-convex optimization such as potentially many local minima, saddle points, very flat regions, or a combination of all three. When utilizing gradient descent, all of these challenges make it extremely difficult to find the global minimum. When optimizing these problems using gradient descent, the quality of our solution is dependent on the initialization used for the problem. As seen in Figure 1.1 the initialization chosen resulted in our minimization only finding a local minimum instead of the global minimum. In fact, finding the global minima is known to be NP-Hard for non-convex optimization problems [3].

When approaching non-convex optimization problems, we can still use methods from convex optimization such as gradient descent to quickly find local minimum. These methods are great for finding local minimum quickly in convex settings. As we can see in Figure 1.1, our non-convex objective can be divided into convex intervals. These convex intervals

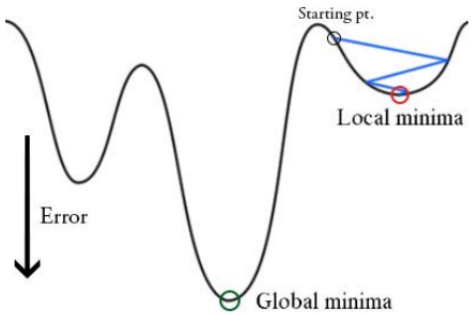


Figure 1.1: Example of nonlinear, non-convex, objective function taken from [1].

explain why gradient descent can find local minima even on a non-convex function. We can reinitialize our solver with different initializations and choose the best minimum from these solutions, but this provides no guarantees that the point found is a global minimum as we don't have a connection between initialization and locally convex region until we solve the optimization problem. Additionally we do not know how many convex regions our objective has.

There are other methods that can escape shallow minimum such as stochastic gradient descent, mini-batching, momentum, and variance reduction [4]. While these methods have been met with success in applications such as optimizing the weights of a neural network, they do not provide a robust method for finding the global minimum for non-convex optimization.

However, for some non-convex optimization problems, such as spatial path planning, there exists other methods outside of pure optimization, which can find the global minimization of the equivalent non-convex optimization problem. Typically, however, these methods are computationally expensive to perform. This leads to the motivation for this Thesis. What if we could leverage these other methods to aid our gradient-based optimizer in finding the global minimum of our optimization problem? More specifically, what if we could use these other methods to generate data offline, to train a neural network that can predict initializations for our optimization problem online, that will lead us to the global minimum.

This type of method would combine the best-of-both-worlds as it leverages the ability of other methods to find global minima with the speed of gradient-based optimization. To explore this question, we examine the shortest distance path planning problem through a field of circular obstacles as shown in Figure 1.2.

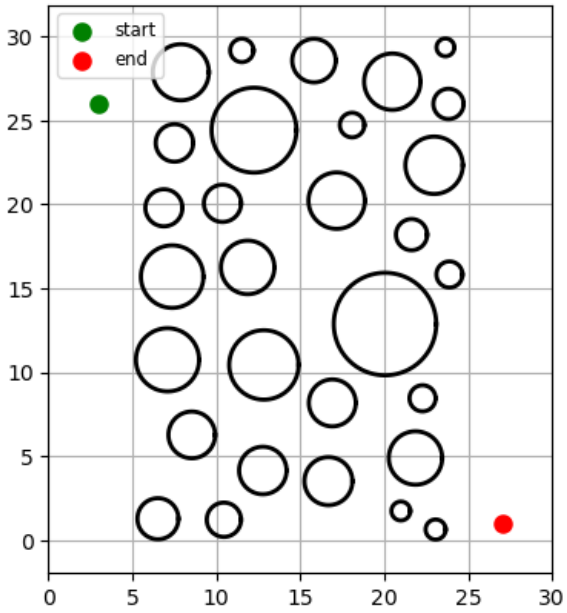


Figure 1.2: Example Path Planning Environment.

1.2 Shortest Distance Path Planning Methods

In shortest distance path planning, our goal is to find a trajectory vector for the shortest distance path that goes from our start to goal coordinate, without intersecting any of the obstacles. This problem is seen in many navigation applications such as path planning for automated robots in warehouses, hospitals, agriculture, and more. This problem is well-known and there are many methods that are used to solve it outside of formulating it as an optimization problem. These methods can be classified as exact, or sampling-based [5]. Exact methods involve calculating the path in the continuous configuration space. One method for doing this is to use visibility graphs [6]. Visibility graphs are created by

making a graph of all the points of visibility between obstacles in the environment. When building a graph with this method, it is shown that the shortest path through the field of obstacles is contained in this graph [6]. The drawback to this method is that the time complexity of computing the visibility graph scales with the number of obstacles, making it undesirable in online applications. Another issue is that visibility graphs work in the continuous configuration space [7] of an obstacle environment. This is undesirable as the time complexity for computing the continuous, configuration space of an obstacle environment is exponential in the dimension of the robot's configuration space [8].

The other class of methods for path planning, which are generally considered state-of-the-art [9], are sample-based methods. In these methods, we create samples of our environment and search the sampled states for our path. These methods are popular as they circumvent the need to calculate the continuous, configuration space, and rather sample individual points from the configuration space [5], reducing our computational costs. Rapidly Exploring Random Trees (RRT), is a sample-based method that builds a graph of our path planning environment by randomly sampling points from the environment and connecting them to the nearest node in the graph [10]. One disadvantage of RRT is that it does not have any guarantee for finding the shortest distance path. To find the shortest distance path, many variants have been created based on this method such as RRT* [11], informed-RRT* [12], or Batch Informed Trees (BIT*) [13]. In these variants, they allow modification of the graph created as new points are sampled, which allows it to find optimal paths asymptotically [11–13]. The issue with these methods for our application is that they still can take a long time to find the optimal path, which is undesirable in online applications.

1.3 Solution Introduction

In this Thesis, we develop a data-based path planning method, which utilizes a classifier trained on offline data generated from visibility graphs, to predict initializations for a shortest distance, optimization problem that will lead us to finding the global minimum of our objective. We use visibility graphs instead of a sampling-based method as this method finds exact solutions for our global shortest path and is proven to find the global minimum. This technique is ideal for online, shortest distance, path planning applications where we wish to perform multiple queries for many robots that are navigating a highly configurable obstacle environment. The reason this method is ideal for online applications, is that it utilizes a combination of a neural network which has time complexity of O(1) with a gradient-based optimization which can find local minimum quickly. Additionally, since the neural network is trained from data created by visibility graphs, it can leverage that data to improve the chances that the solution found via our optimization problem will be a global minimum. The reason this method is ideal for multiple-query problems is due to the time it takes to generate offline data with visibility graphs. If we invest the time to create an offline dataset, we want it to be for an environment that will receive many queries so we can see the benefit of our method versus exact and sampling-based methods. Lastly, the reason why this method is meant for highly configurable obstacle environments, is that visibility graphs provide a better solution for static environments as it only needs to calculate the obstacle visibility graph once.

In the following chapters, Chapter 2 covers the preliminaries from the literature that are used to establish our result. In Chapter 3, we cover the methodology of our path finding solution. In Chapter 4, we go over the results found from training our neural network. Finally, Chapter 5 gives our conclusions and avenues for future research.

Chapter 2

Preliminaries

In this chapter we examine the concepts and material from the literature that is used to develop our method. We leverage concepts from computational geometry, optimization, and machine learning.

2.1 Notations

For describing path planning problems, we use the following notation. x_{ck} and y_{ck} are the coordinates to the center of obstacle k of n obstacles, with R_k corresponding to obstacle k's radius. Points along the discrete trajectory of the path are given by (x_i, y_i) where this is the i^{th} point our of L along the trajectory vector $(\mathbf{x}, \mathbf{y})$ with $\mathbf{x}, \mathbf{y} \in \mathbb{R}^L$. (x_{start}, y_{start}) refers to our start location and (x_{goal}, y_{goal}) refers to our goal location. These values are shown in Figure 2.1 and are used later in (3.1). When referring to optimal path, we are referring to the shortest-distance path.

For optimization problems, an initialization refers to the initial values of our variables of optimization. We require these values to be able to perform optimization such as gradi-

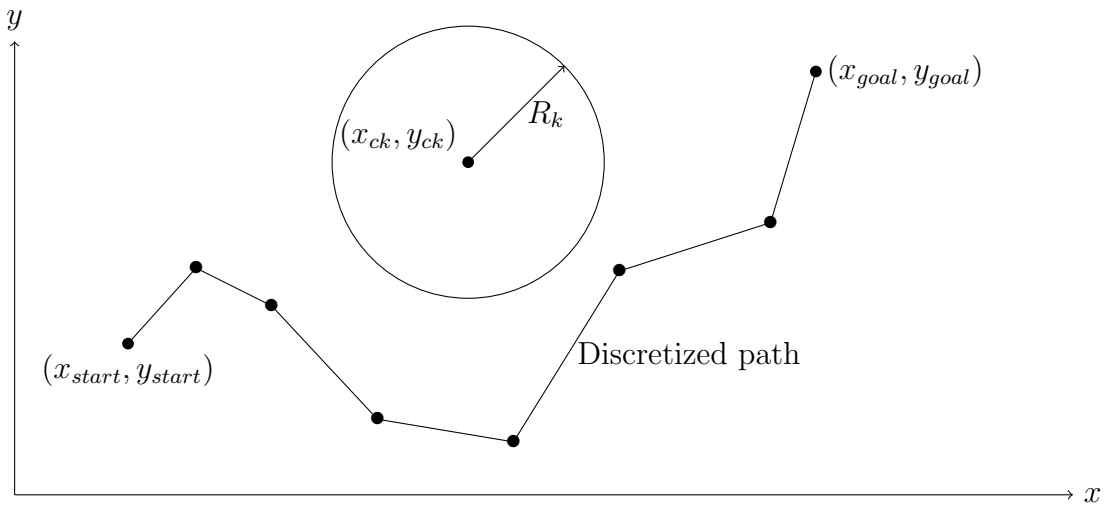


Figure 2.1: Components of environment and path description.

ent descent. Re-initialization refers to solving the optimization problem again, but with a different set of initial values.

When referring to our artificial neural network, we refer to it as a neural network or a classifier. When referring to neural network architecture, we are referring to the number of hidden layers, the number of neurons per hidden layer, the activation functions used in the hidden layers, the optimizer chosen, and any hyperparameters used such as batch size, learning rate, and number of epochs.

When referring to the axes of plots in this Thesis, x and y will refer to the directions as shown in Figure 2.2.

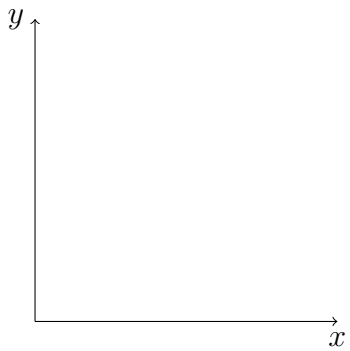


Figure 2.2: Axis labels for obstacle environment.

2.2 Circle Direction Label Notation

In this work we are applying a binary label to each obstacle in our problem. These labels are "above" and "below", or "up and "down". In Figure 2.3, we provide visual examples of what these labels correspond to visually.

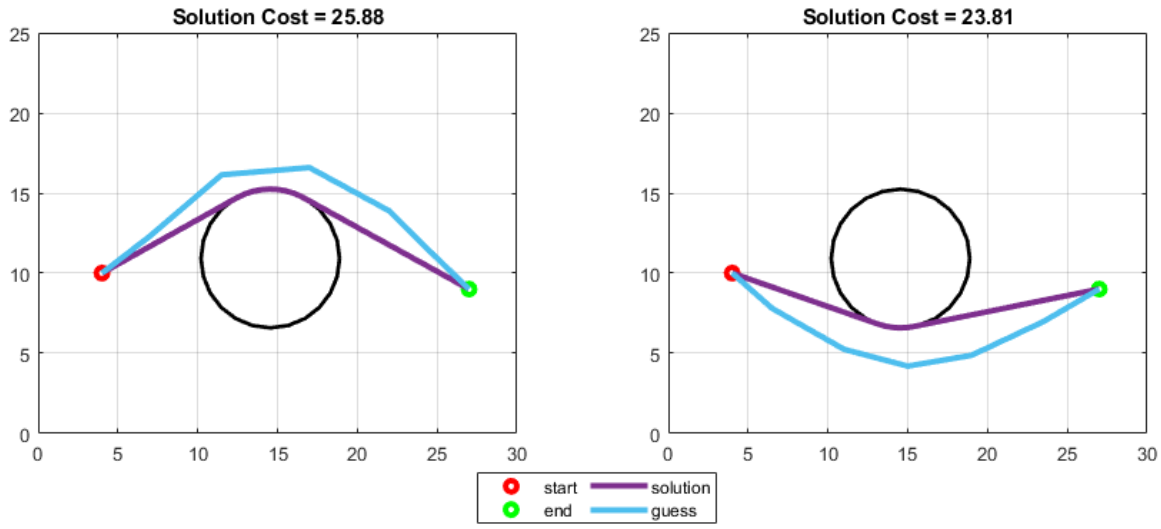


Figure 2.3: Example of obstacle with up and down labels.

For the obstacle on the left, we can see it has an "up" label as our initialization, and final solution go "above" the circle's center on the y-axis. Similarly, on the right we have a "down" label as our initialization goes "below" the circle's center on the y-axis. Note that the initialization and solution found will match the direction label of the circle.

2.3 Shortest Distance Path Planning

Shortest distance path planning consists of finding a trajectory in an environment that provides the shortest distance from the start to the goal. In shortest distance path planning, an initialization refers to a vector of points that is used as the initial solution of the problem.

With an initial solution we can perform gradient descent on our optimization problem to find the shortest distance trajectory.

2.4 Dubins Path

In geometry, a Dubins path refers to the curve of minimal length between a start and end goal with a constraint on average curvature defined in Definition 1. We give a short view of Dubins Path as this is the basis for why our visibility graphs discussed in the next section, contain the shortest path through a field of circles.

Definition 1 *For a curve $\mathcal{X}$, in real n-dimensional Euclidean space, parameterized by arc length s, for which $\ddot{\mathcal{X}}(s)$ exists everywhere, we define its average curvature, $f(s_1, s_2)$ by* (2.1) *[14].*

$$f(s_1, s_1) = \frac{\|\dot{\mathcal{X}}(s_1) - \dot{\mathcal{X}}(s_2)\|}{|s_1 - s_2|} \tag{2.1}$$

We say that a curve $\mathcal{X}$ in real Euclidean n-space parameterized by arc length has average curvature always less than or equal to R^{-1} provided that its first derivative $\dot{\mathcal{X}}$ exists everywhere and satisfies the Lipschitz condition (2.2) [14].

$$\|\dot{\mathcal{X}}(s_1) - \dot{\mathcal{X}}(s_2)\| \leqq R^{-1}|s_1 - s_2| \tag{2.2}$$

In [14], it is proven that the shortest distance path, or R-geodesic, consists of pieces, each of which are either a straight-line segment, or an arc of a circle of radius R or less. We note in (2.2), that for a field of circles with varying radii, R is equal to the radius of the largest radius circle.

2.5 Visibility Graphs

In this section we cover graph notation, tangent circular visibility graphs, and how to find the shortest distance path with them.

2.5.1 Graph Notation

A graph $\mathcal{G}$, is a collection of nodes and edges, $(\mathcal{V}, \mathcal{E})$, respectively. For our visibility graphs, a node is a point in our free space, and an edge refers to lines of visibility between the nodes. We take lines of visibility to mean lines of sight between nodes with no obstruction from obstacles. The edges can also refer to edges connecting two nodes on an obstacle.

2.5.2 Tangent Circular Visibility Graphs

A graph is considered a visibility graph if its vertices $v_1, \ldots, v_n$ can be associated with disjoint, horizontal line segments $s_1, \ldots, s_n$ in the plane such that v_i and v_j are joined by an edge if and only if there is a vertical line segment connecting s_i to s_j without intersecting any other s_k [15]. Visibility graphs have many uses such as circuit planning, motion planning, and are fundamental for use in computational geometry and geometric graph theory [16]. For navigation through a field of obstacles, we can utilize visibility graphs as a way of simplifying our configuration space. With this graph built based on points of visibility between obstacles, we can use graph search techniques to find the shortest paths.

To construct a visibility graph for circles, we first must find the bitangents between our obstacles. We must find these bitangents as the graph built from them will contain the Dubins path from our start to goal. Between two non-intersecting circles, there exists four bitangents. A bitangent is a line that is tangent to two obstacles. There are two types of

bitangents as shown in Figure 2.4. These are the first type of edge that connects nodes in our visibility graph. These types of edges are referred to as surfing edges.

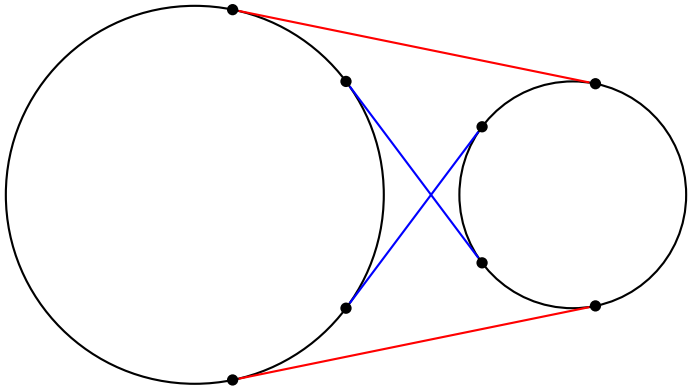


Figure 2.4: External (red) and internal (blue) bitangents.

The second type of edge are edges between nodes on the same obstacle, which are referred to as hugging edges as shown in Figure 2.5.

2.5.3 Bitangents Between Two Circles

To calculate the bitangents needed for our graph we refer to the solution of the belt and pulley problem [17].

To calculate the endpoints of our external bitangents, we find θ with equation (2.3), where

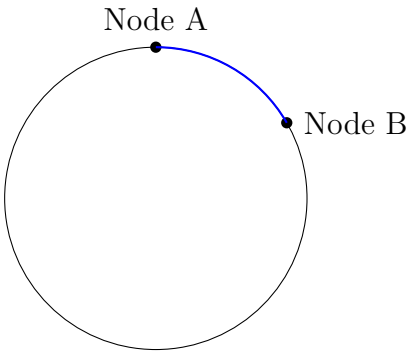


Figure 2.5: Hugging edge of circle visibility graph.

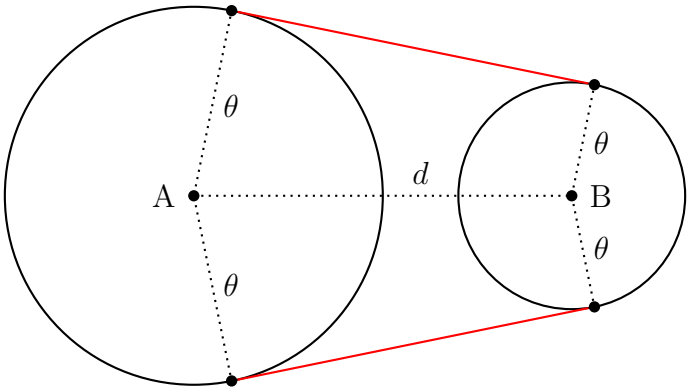


Figure 2.6: External bitangent diagram.

r_A is the radius of circle A, and r_B is the radius of circle B.

$$\theta = \arccos\left(\frac{r_A + r_B}{d}\right) \tag{2.3}$$

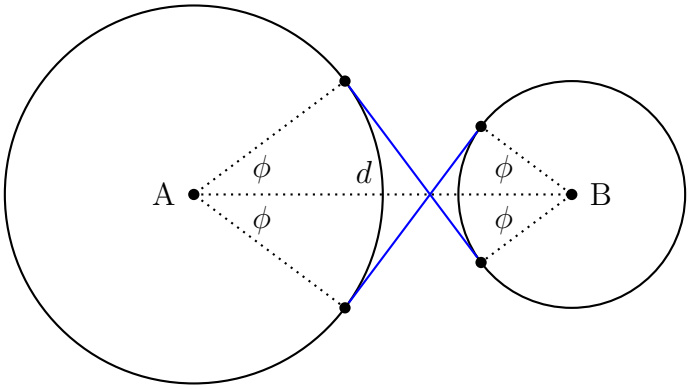


Figure 2.7: Internal bitangent diagram.

To calculate the endpoints of our internal bitangents, we find ϕ with equation (2.4), where r_A is the radius of circle A, and r_B is the radius of circle B.

$$\phi = \arccos\left(\frac{|r_A - r_B|}{d}\right) \tag{2.4}$$

We have shown how to find the nodes and edges of the bitangents between circles. In the next section we will explain how to check if these bitangents are lines of visibility or if another obstacle obstructs this line of visibility.

2.6 Collision Primitives

There are a variety of collision primitives that we need to use to build a visibility graph. For example, how do we check if there is an intersection between a line of visibility and an obstacle? Or how can we check if two circular obstacles are intersecting? The primitives in the following section provide routines for answering these questions.

2.6.1 Line Segment Intersection Primitive

To detect a line intersection, we use the primitive described in [18]. This is used to build our visibility graph.

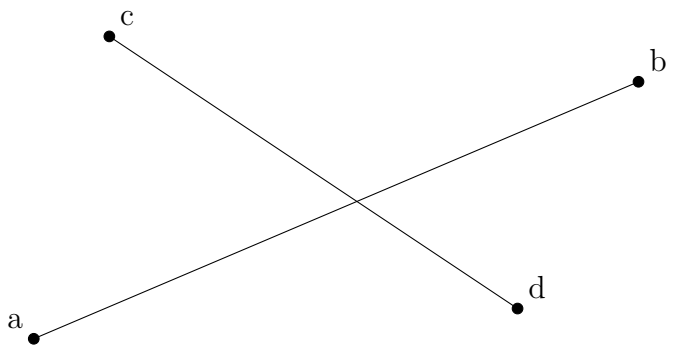


Figure 2.8: Intersection of two line segments

For two line segments $\overline{ab}$ and $\overline{cd}$ we wish to determine if there is a single point of intersection. Another way to describe this problem is if there exists a point p_1, on $\overline{ab}$ and a point p_2, on $\overline{cd}$ such that $p_1 \equiv p_2$. We can accomplish this by first checking if the lines, $\overleftrightarrow{ab}$ and $\overleftrightarrow{bc}$ intersect. If they intersect, we then check if the point of intersection lies within the segments, $\overline{ab}$ and $\overline{cd}$.

To check if the two lines intersect, we first note that we can describe any point along each

line segment with:

$$p_1 = a + s_1(b - a) \quad \text{for } s_1 \in [0, 1]$$
$$p_2 = c + s_2(d - c) \quad \text{for } s_2 \in [0, 1]$$

Now if we set p_1 equal to p_2 and describe $a = (x_a, y_a)$, $b = (x_b, y_b)$, $c = (x_c, y_c)$, and $d = (x_d, y_d)$, we get a system of two linear equations with two unknowns, s_1 and s_2:

$$x_a + s_1(x_b - x_a) = x_c + s_2(x_d - x_c)$$
$$y_a + s_1(y_b - y_a) = y_c + s_2(y_d - y_c)$$

We can solve for s_1 with:

$$s_1 = \frac{(x_d - x_c)(y_a - y_c) - (y_d - y_c)(x_a - x_c)}{(y_d - y_c)(x_b - x_a) - (x_d - x_c)(y_b - y_a)}$$

and s_2 with:

$$s_2 = \frac{(x_b - x_a)(y_a - y_c) - (x_a - x_c)(y_b - y_a)}{(y_d - y_c)(x_b - x_a) - (x_d - x_c)(y_b - y_a)}$$

Having solved for s_1 and s_2 we can determine if the lines $\overleftrightarrow{ab}$ and $\overleftrightarrow{bc}$ intersect, by examining the values of the numerator and denominator of either s_1 or s_2. If the denominator is nonzero, then the lines are not parallel and intersect at a single point. If the denominator equals zero, then the lines do not intersect at a single point.

If the lines $\overleftrightarrow{ab}$ and $\overleftrightarrow{bc}$ do intersect, we can now check if the line segments $\overline{ab}$ and $\overline{cd}$ intersect by checking if s_1 and s_2 belong to the interval $[0, 1]$.

2.6.2 Line Segment and Circle Intersection Primitive

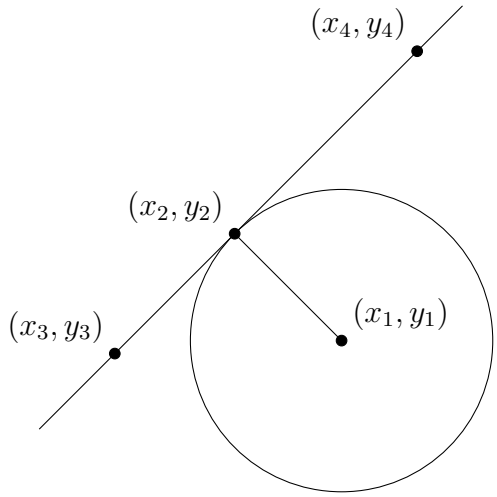
When checking if a line segment intersects a circle, there are three cases. The first case is the line does not intersect the circle. The second case is the line intersects the circle once, and the third case is the line intersects the circle twice. We first check if the line defined by two points intersects the circle. Next, we check if this intersection occurs within the line segment. To perform this task, we use the following primitive. First, we find the distance d, of the line $ax + by + c = 0$ from the center of the circle (x_1, y_1) using the following equation from [19]:

$$d = \frac{|ax_1 + by_1 + c|}{\sqrt{a^2 + b^2}}$$

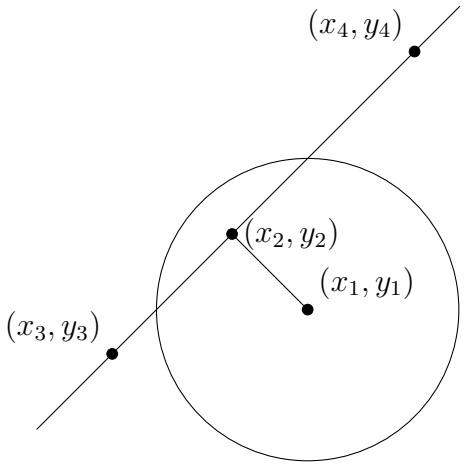
From here we compare d to the radius of the circle, r. If $d > r$, the circle and line do not intersect. If $d \leq r$, the circle and line intersect, so we now need to check if the line segment intersects the circle. We can do this by checking if the line segment defined by (x_3, y_3) and (x_4, y_4) intersects the line segment defined by (x_1, y_1) and (x_2, y_2) as seen in Figure 2.9. (x_2, y_2) is the point on the line segment defined by (x_3, y_3) and (x_4, y_4) that is closest to the center of the circle (x_1, y_1). Let the line segment defined by (x_3, y_3) and (x_4, y_4) be described by $ax + by + c = 0$. Then, from [20], we can find (x_2, y_2) with equation (2.5).

$$\begin{aligned} x_2 &= \frac{b(bx_1 - ay_1) - ac}{a^2 + b^2} \\ y_2 &= \frac{a(ay_1 - bx_1) - ac}{a^2 + b^2} \end{aligned} \tag{2.5}$$

Now we can use the primitive defined in section 2.6.1 to check if these two line segments intersect, which indicates if the line segment defined by (x_3, y_3) and (x_4, y_4) intersects the circle.



(a) Circle intersection with one point of intersection.



(b) Circle intersection with two points of intersection.

Figure 2.9: Possible intersections of a line segment and circle.

2.6.3 Circle Intersection Primitive

To check if two circles are intersecting, we use the following primitive. The circles are considered colliding if they either intersect, or one circle is completely contained within the other.

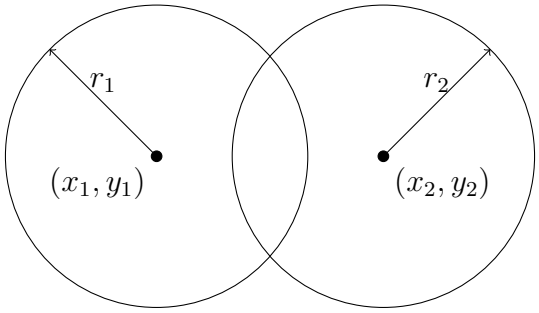


Figure 2.10: Two intersecting circles.

First we calculate the distance d, between (x_1, y_1) and (x_2, y_2).

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

If $d > (r_1 + r_2)$, then the circles do not intersect. If $d \leq (r_1 + r_2)$, then the circles are intersecting.

2.7 Machine Learning

Machine learning is a field of artificial intelligence that answers the question: can we devise algorithms that enable computers to learn from data? The basic premise of learning from data is whether we can use a set of observations to uncover an underlying process [21]. Since the problem of learning from data is a very general problem, machine learning is considered to have three subfields: supervised learning, unsupervised learning, and reinforcement learning. In this thesis we use techniques from the field of supervised learning.

A dataset consists of observations from an underlying process. For example, a dataset could consist of the attributes of a sample of cars such as engine type, car brand, miles driven, and their monetary value. Now consider we wish to make a prediction based on this data, for example given a car's engine type, brand, and miles driven can we predict the car's monetary value? This is an example of a supervised learning problem. This falls under supervised learning as our dataset consists of features x, i.e. engine type, brand, and miles driven, to describe the car, as well as labels y, i.e. the monetary value, which explains the correct answer for a given sample of features. The goal of our learning problem is to find a function g, that closely approximates an unknown target function f, that correctly maps our feature space $\mathcal{X}$, to our label space $\mathcal{Y}$. To turn this problem into a supervised learning problem, we use the learning framework shown in 2.11.

First we consider a hypothesis space $\mathcal{H}$ which consists of the set of parametric functions $h(x; w)$, that maps our feature space to our output space, i.e. $h(x; w) : \mathcal{X} \to \mathcal{Y}$. There are many different options for $\mathcal{H}$, but in this work, we use neural networks for our functions

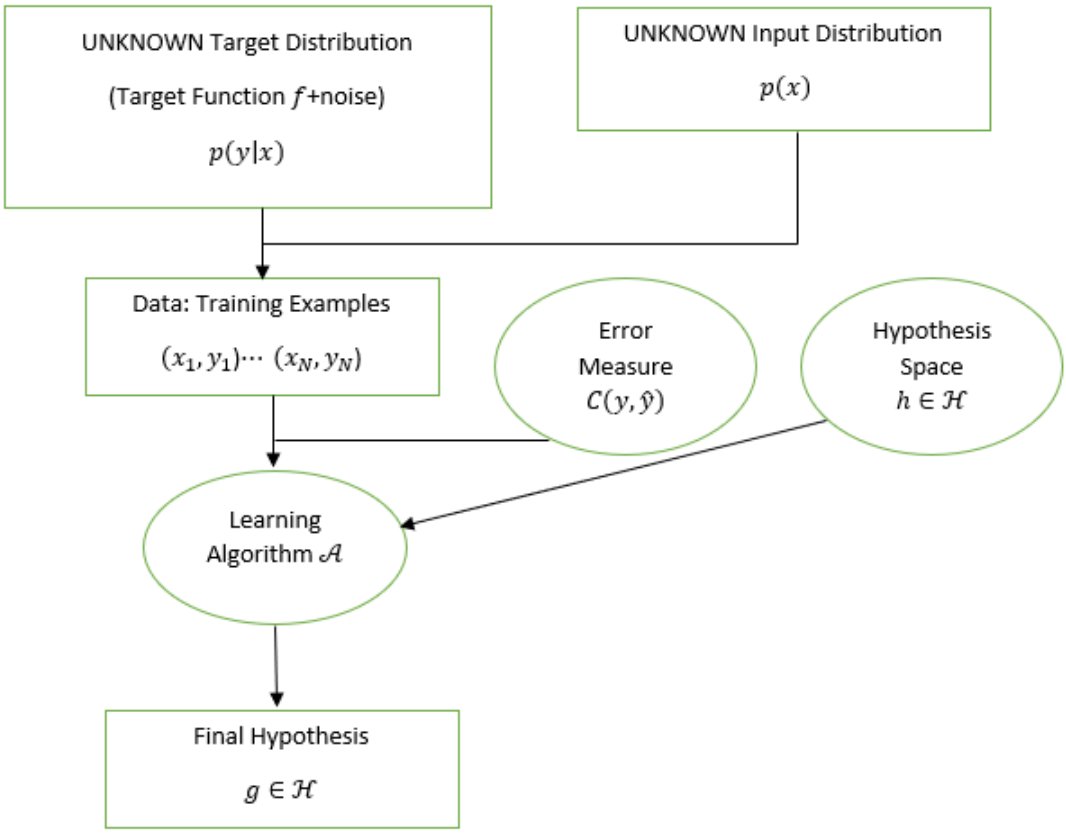


Figure 2.11: Overview of Machine Learning Process

$h(x; w)$, where w is a vector of parameters defining a hypothesis in $\mathcal{H}$. Next we present our data and $\mathcal{H}$ to our learning algorithm $\mathcal{A}$, which is a procedure that seeks to pick optimal parameters w^* such that $f(x) \approx g(x) \equiv h(x; w^*) \in \mathcal{H}$.

But how do we determine if $g \approx h$? To do this we define an error or cost function $C(y, \hat{y})$, where we compare our labels $y = f(x)$ from our dataset, with the predictions $\hat{y} = g(x) \equiv h(x; w^*)$. Using the cost $C(y, \hat{y})$, our learning algorithm $\mathcal{A}$, picks our w^*, usually found as the solution to an optimization problem of $C(y, \hat{y})$. In the following sections we elaborate on the specifics used for $\mathcal{H}$, $C(y, \hat{y})$, and $\mathcal{A}$ in our classification problem. For more information on the field of machine learning, please consult [21].

2.7.1 Artificial Neural Networks

An Artificial Neural Network, or neural network, is a type of parametric function that can be characterized by a vector of weights w. The basic building block of a neural network is a neuron that is connected to other neurons via links. [22]. Each link has a numeric weight associated with it. Weights are the primary means of long-term storage in neural networks, and learning takes place by updating these weights. As shown in Figure 2.12, a neuron creates a weighted sum of its input links, $\mathbf{x} \in \mathbb{R}^r$ with its weights $\mathbf{w} \in \mathbb{R}^n$ plus a bias b. Next the neuron takes this sum as the input to an activation function f. The output of f is then taken as the neuron's output. f is a function that can be set for each neuron.

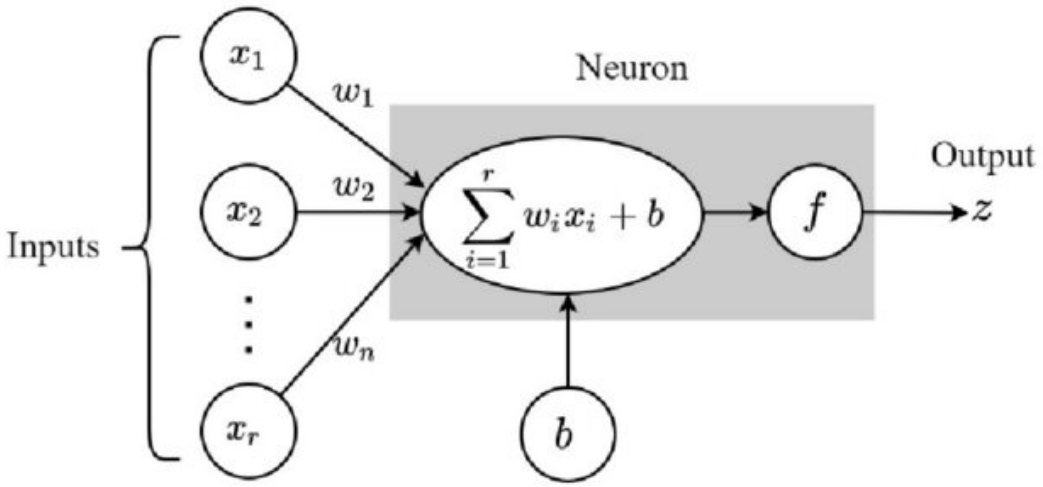


Figure 2.12: Structure of a neuron, taken from [2].

If we form a network of multiple neurons arranged in layers, with the outputs of the neurons in the previous layer as the inputs to the next layer, we get a feedforward neural network (FFNN), as shown in Figure 2.13. When a FFNN has multiple hidden layers, it becomes a deep neural network (DNN). In this work we utilize deep feedforward neural networks but will refer to them as FFNN.

The motivation for using FFNN is their ability to be used as universal approximators for

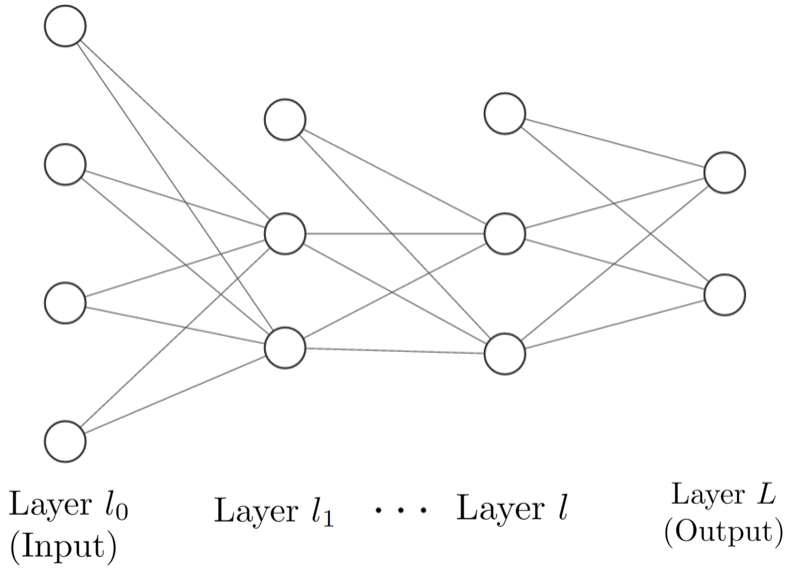


Figure 2.13: Structure of a deep feedforward neural network. Layers $l_1, \ldots, l$ are called hidden layers.

linear and nonlinear functions. Specifically, the universal approximation theorem [23–25] states that a feedforward network with a linear output layer and at least one hidden layer with a "squashing" activation function can approximate any Borel measurable function from one finite-dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available. In [26], extended universal approximation to hold for networks built with ReLU activation functions as well. Borel measurability is beyond the scope of this thesis, but for our purposes, we can say a continuous function on a closed and bounded subset of $\mathbb{R}^n$ is Borel measurable and therefore may be approximated by a neural network [27]. In the next section, we will explain how the weights of FFNN are updated.

2.7.2 Learning Algorithms

A learning algorithm is the mechanism by which the weights of a neural network are updated. To create this formulation, we first need to define the Risk $R(w)$, which is the objective we

attempt to minimize.

$$R(w) = \int C(y, h(x; w))p(x, y)dxdy \tag{2.6}$$

In equation (2.6), $p(x, y)$ is the joint probability density function of our input features x, and output labels y. We wish to solve $\min_w R(w)$ to obtain w^* and $h(x, w^*) \in \mathcal{H}$. However, $p(x, y)$ is unknown and we only have our independent samples $(x_k, y_k) \approx p(x, y)$.

To estimate $R(w)$, we examine the empirical risk R_{emp} shown in (2.7) which is based on our independent samples.

$$R_{emp} = \frac{1}{N}\sum_{n=1}^{N} C(y_n, h(x_n, w)) \to R(w),\ N \to \infty \tag{2.7}$$

The law of large numbers implies that $R_{emp} \to R(w),\ N \to \infty$ for independent samples (x_k, y_k) [28].

We can update our weights w, with Gradient Descent shown in equation (2.8), where η is our learning rate.

$$w(t+1) = w(t) - \eta\frac{\partial R_{emp}}{\partial w} \tag{2.8}$$

We can efficiently calculate $\frac{\partial R_{emp}}{\partial w}$ by using backpropagation. See [28] for further details on this method.

To improve convergence, there are other gradient based methods which utilize momentum and adaptive learning rates to improve our results. One of the methods is ADAM and is primarily used in this work.

2.7.3 Loss Function Binary Cross-Entropy

For our loss function, we use binary cross-entropy as this loss is suitable for training binary classification problems [28].

2.7.4 ReLU, Sigmoid, and Tanh Activation Functions

An activation function maps an input to a new domain depending on the function used. These functions are typically nonlinear, although linear functions can be used. We use nonlinear functions as this allows our neural network to better approximate nonlinear processes. There are two activation functions that are used in this work, the Rectified Linear Unit (ReLU), Sigmoid functions, and Tanh functions. There are many more options for activation function, but we will only cover the ones used in this work. In the following diagrams, we will define s for a given neuron in (2.9) using the variables established in Figure 2.12.

$$s = \sum_{i=1}^{n} w_i x_i + b \tag{2.9}$$

The ReLU activation function shown in Figure 2.14, linearly maps positive values of s while zeroing negative values. This is a popular function for hidden layers as this function does not suffer as severely from the exploding/vanishing gradient problem.

The sigmoid activation function shown in Figure 2.15, takes the weighted sum s and maps the values between zero and one. We note that sigmoid has the following asymptotic behavior:

$$\sigma(s) = \begin{cases} 1, & \text{for } s \to +\infty \\ 0, & \text{for } s \to -\infty \end{cases}$$

The sigmoid function is useful for classification problems with a binary decision. For example

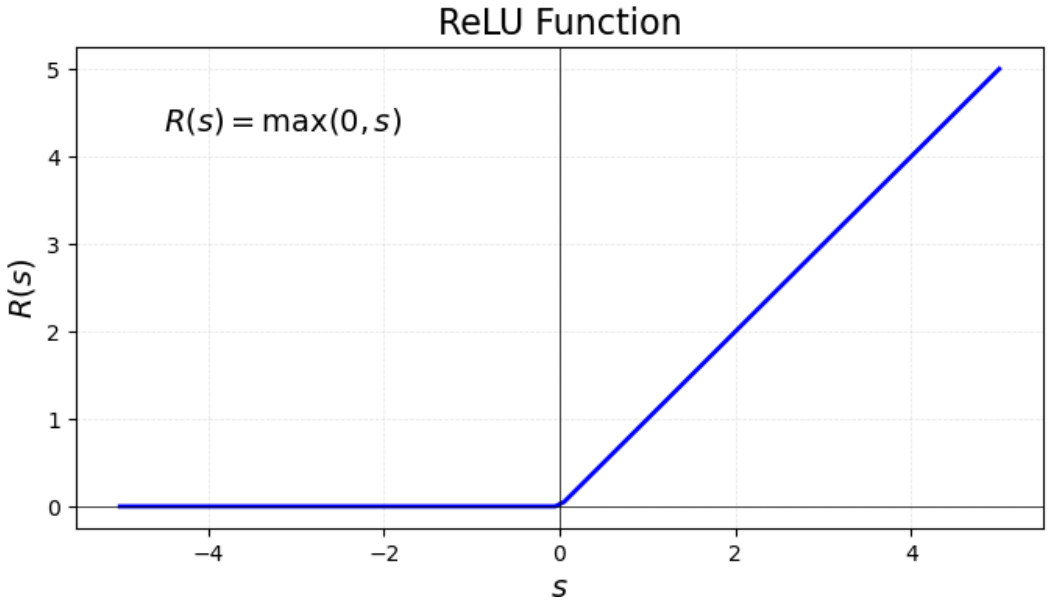


Figure 2.14: The ReLU activation function.

it could be used in the decision of whether a picture shows a cat or does not show a cat. The output of the sigmoid function represents the probability of the input belonging to the positive class. We use a cutoff of 0.5 in mapping the decision class to either the positive or the negative decision, e.g. if the picture does or does not contain a cat.

Lastly, the tanh function shown in Figure 2.16, takes the weighted sum s and maps the values between negative one and one. We note that tanh has the following asymptotic behavior:

$$\phi(s) = \left\{ \begin{array}{ll} 1, & \text{for } s \to +\infty \\ -1, & \text{for } s \to -\infty \end{array} \right\}$$

This output range of negative one to one is suitable for applications for which we need to capture both positive and negative relationships in the data. Although this function is typically not chosen due to the vanishing/exploding gradient problem, our best performing networks typically used one layer where the effect of the vanishing/exploding gradient is not as much of a problem.

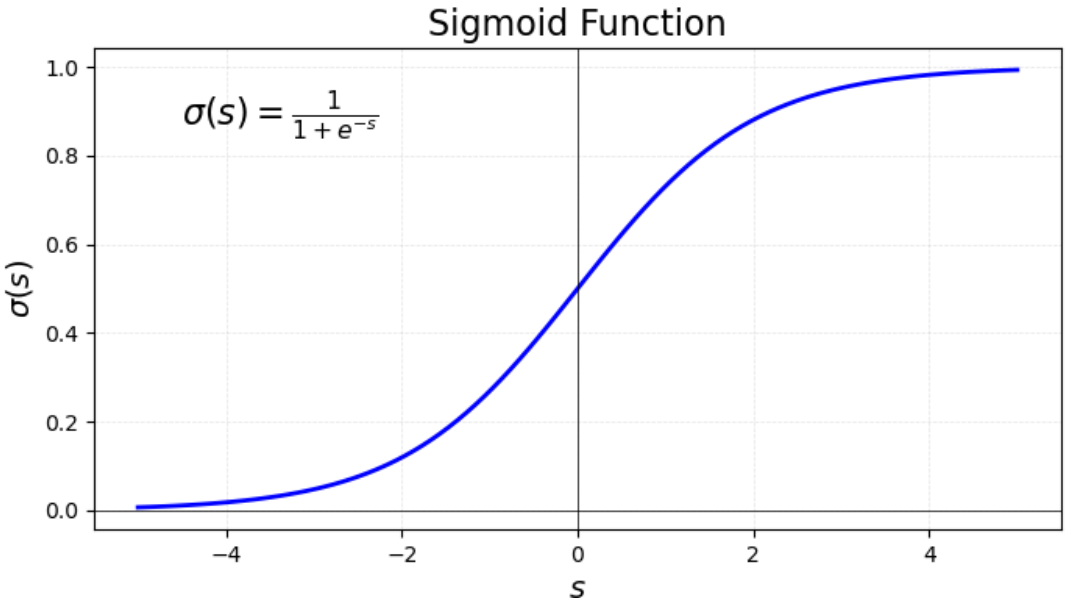


Figure 2.15: The sigmoid activation function.

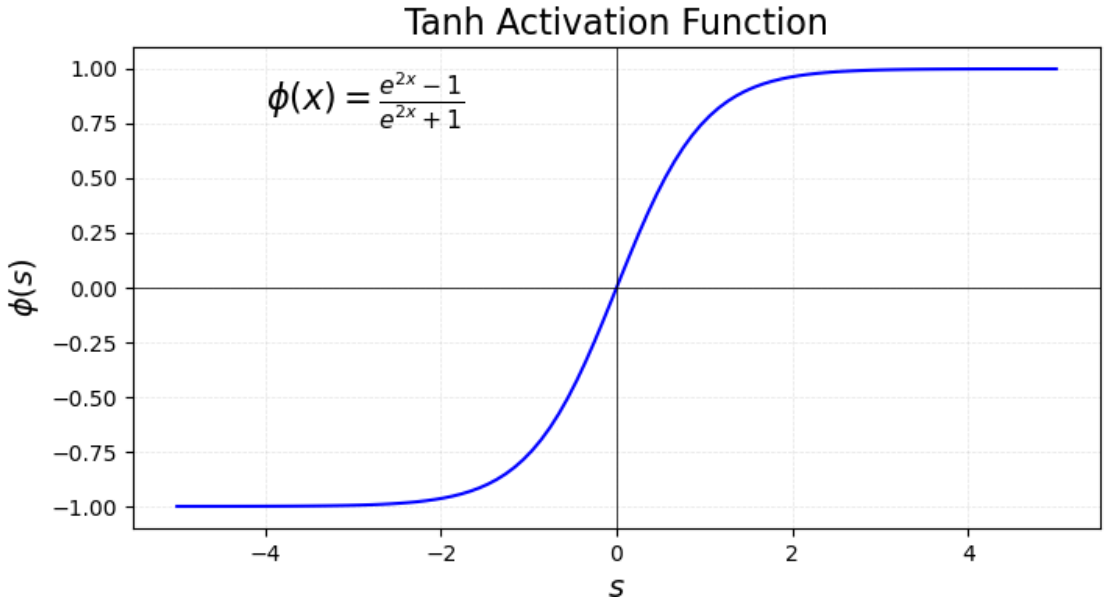


Figure 2.16: The tanh activation function.

Chapter 3

Methodology

In this chapter, we will cover the three parts of our path planning solution. The first section will describe the formulation of our shortest distance optimization problem. The second section will cover data generation for our classifier. The third section will cover our neural network design, and the last section will explain how we integrated our classifier with our optimization problem.

3.1 Shortest Path Formulation

Let us consider the following motion planning problem. We desire to determine the shortest path between a start and goal point in a field of circular obstacles. We use circular obstacles as this is a conservative approximation for all obstacles if the circle fully contains the obstacle. We also choose circular obstacles for their simplification of our neural network features, as a circle can be described by just three parameters. We wish to find the shortest distance, collision-free path. For a given environment consisting of N obstacles, and by describing the trajectory as a discrete vector of points of length $L + 1$, starting at the start coordinate

and ending at the goal, we can formulate finding the shortest distance path as a constrained minimization as shown in equation (3.1).

$$\begin{aligned} \min_{x,y} \quad & \sum_{i=0}^{L} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2} \\ \text{s.t} \quad & (x_j - x_{ck})^2 + (y_j - y_{ck})^2 > R_k^2, \qquad \forall j \in \{0, \ldots, L\}, \forall k \in \{1, \ldots, N\} \\ & (x_0, y_0) = (x_{start}, y_{start}) \\ & (x_L, y_L) = (x_{goal}, y_{goal}) \end{aligned} \tag{3.1}$$

In this formulation, our objective function is our total path length calculated by summing the L2-norm between adjacent points in the trajectory vector. For our inequality constraints, we use the equation of a circle to enforce that our trajectory vector is outside the circle. We check every point of the trajectory against every circle. For our equality constraints, we require the first point of our trajectory to be our start location, and the final point of our trajectory to be our goal location. Although our objective function is convex, our constraint set is non-convex resulting in our optimization formulation being non-convex. To minimize this problem, we utilize the interior-point algorithm.

3.1.1 Connecting Unique Local Minima to Optimization Initializations

For each local minimum, there are an infinite number of initialization trajectories that will result in our optimizer finding the same local minimum. However, due to the circles being the source of non-convexity in our optimization problem, there is a pattern in the relative direction of the path around an obstacle, and the local minimum found. We make the following conjecture. Each local minimum can be characterized by the relative direction of the initialization around an obstacle. For a field of N obstacles, there is up to 2^n local

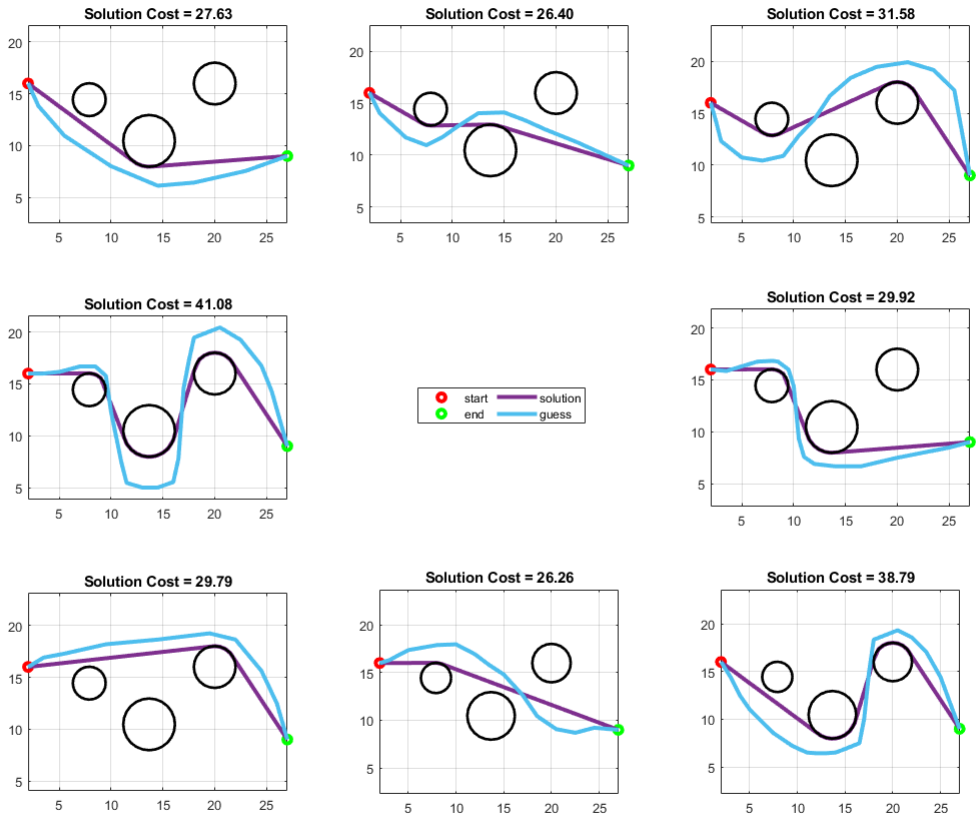


Figure 3.1: Visual of All Unique Initializations for Three Obstacle Course.

minima. Each of these local minima can be found by utilizing an initialization that obeys the specified directions around the obstacles. For example in Figure 3.1, we can see there are eight patterns of "above" and "below" for the initializations of a three-obstacle course. Any initialization for equation (3.1) that matches a given pattern of "above" and "below" labels will find the same local minimum. This observation is imperative for building a connection between our optimization formulation, and the classifier we will create. This is because our classifier will communicate a given initialization to our optimization problem by specifying the relative directions around obstacles our initialization must follow. This initialization will set up our optimization problem to have local convergence to the global minimum if the neural network prediction is correct.

3.2 Neural Network Design for Classification

In this section we outline the design of our Neural Network classifier. First, we will describe the features and labels chosen, and then we will describe the architecture of the classifier.

3.2.1 Features and Labels

When choosing features for the input layer of our Neural Network, we want to use parameters from the problem that can help the classifier identify the correct obstacle labels. The features chosen to describe our problem environment are shown in Figure 3.2. The first four features

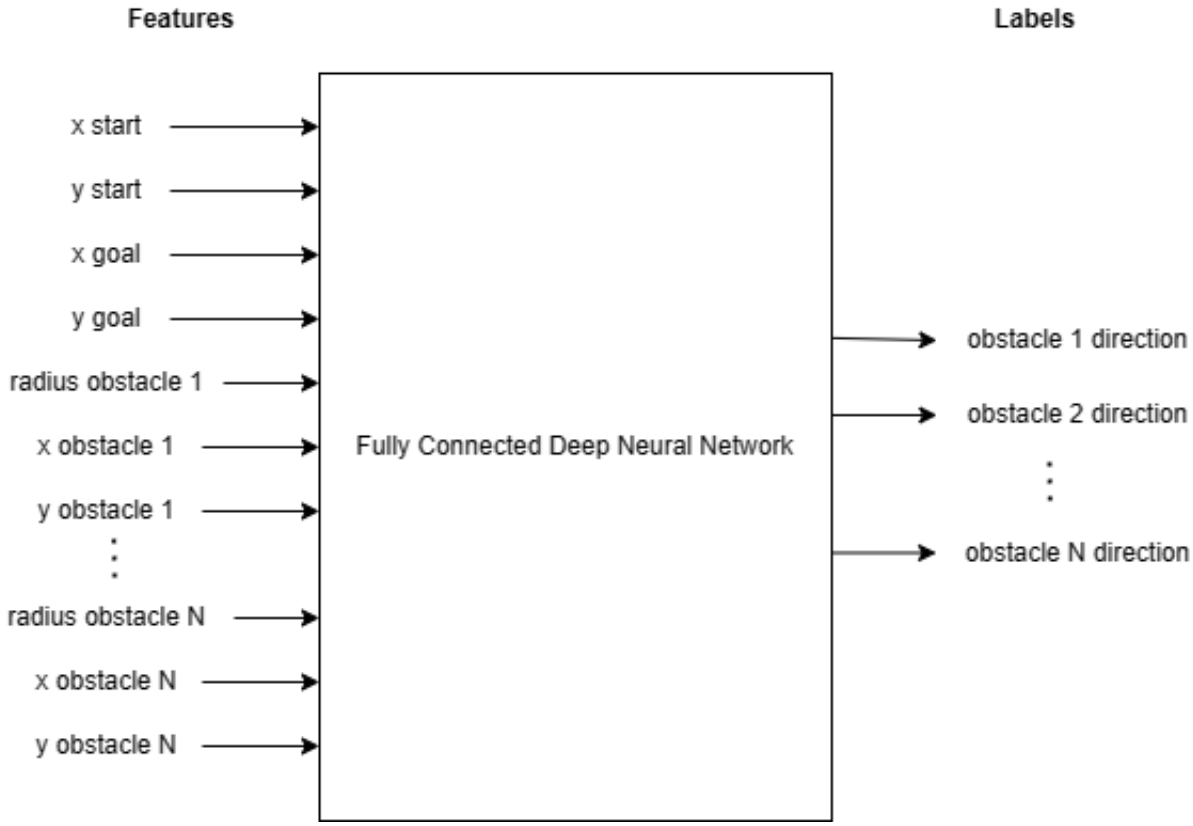


Figure 3.2: Features and Labels used for Neural Network

x start, y start, x goal, and y goal, are real numbers that correspond to the coordinates of our start point and goal point. The next features are real numbers that describe our

circular obstacles. For each obstacle there are three features, the radius, the x, and the y coordinate of the circle's center, going from obstacle 1 to obstacle N. To calculate the number of features, f, we can use equation (3.2). In this study we examined classifiers for obstacle environments with up to twenty obstacles.

$$f = 3N + 4 \tag{3.2}$$

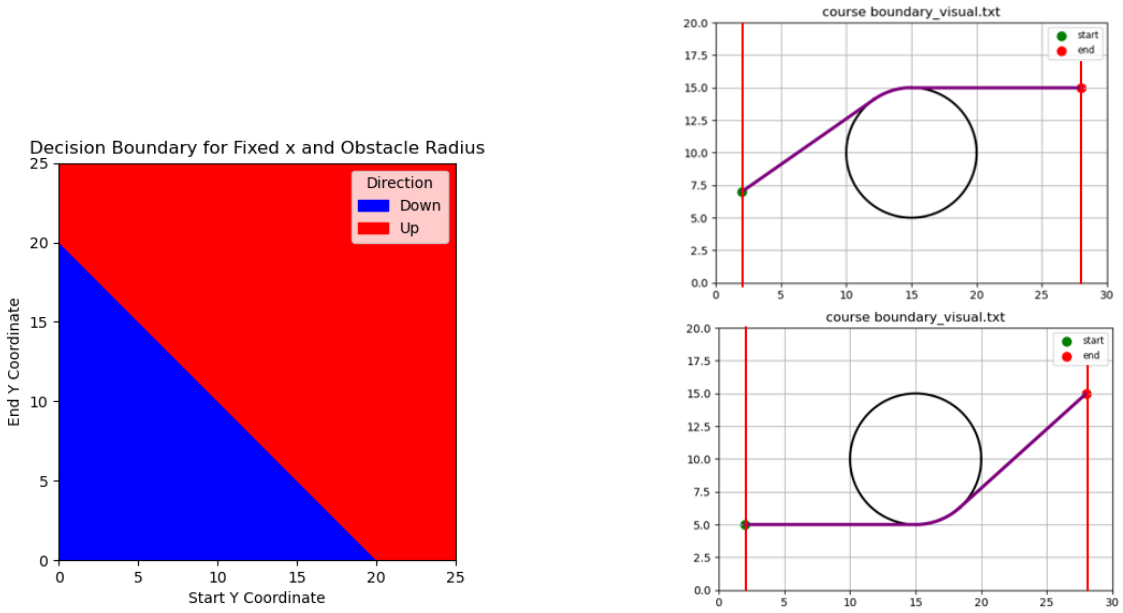
The labels used for our classifier are "down" or "up", or zero and one, to indicate the direction the optimal path takes around an obstacle. There are N binary outputs that each correspond to one of the N obstacles in our environment.

Due to neural networks having fixed number of features and labels, this implies that a given classifier can only make predictions on courses with N number of obstacles. To overcome this issue, we came up with the following solution to make an N obstacle environment able to predict on an $n < N$ obstacle environments. For features referring to non-existent obstacles in our environment, we zero these inputs and ignore the predicted labels. We present the results of this method in section 4.4.

3.2.2 Is this Learning Feasible?

An important question in learning is do our features provide the information needed to learn the underlying process? To examine if learning on our problem is feasible, we examined in reduced feature cases the decision boundary. For example, in Figure 3.3a, we examine a simplified version of our problem where all our features are fixed except for the features y_{start} and y_{goal}. We see that based on these features we form a continuous decision boundary which a neural network could learn. In Figure 3.3b, the red lines indicate the axis on which the features shown in the plot were varied. A start end pair in the red region of Figure 3.3a

corresponds to the top plot in 3.3b, while the blue region corresponds to the bottom plot in 3.3b.

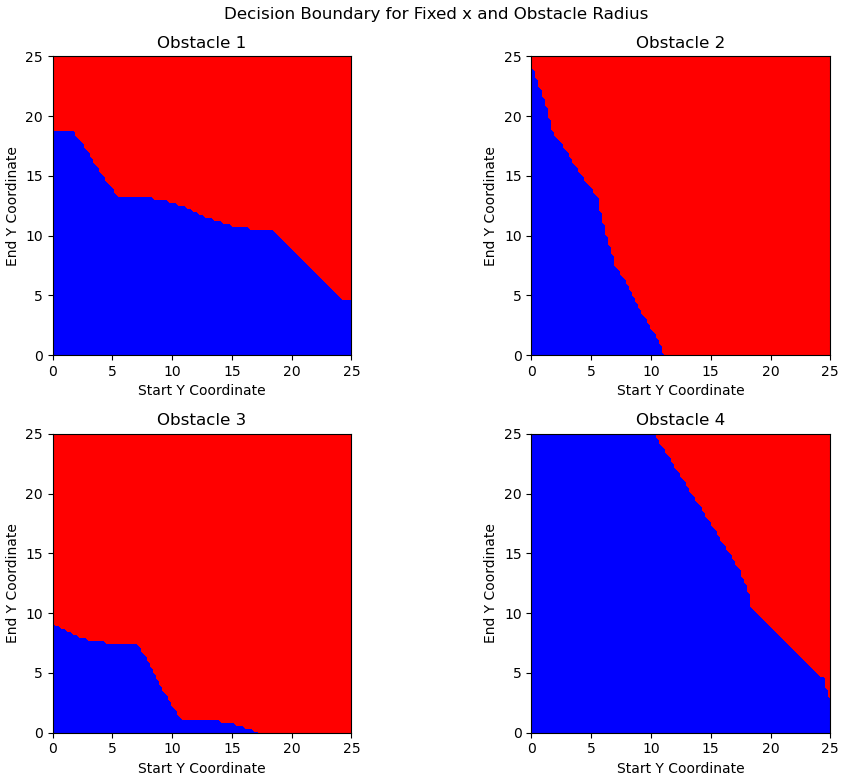


(a) One obstacle decision boundary

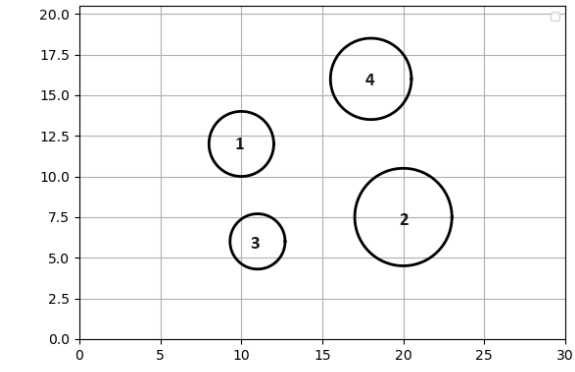
(b) One obstacle environment.

Figure 3.3: Decision Boundary Visual for One Obstacle.

To examine our other features, we performed the same problem but added more obstacles. In Figure 3.4a, we plot the decision boundary for each obstacle with the same features y_{start} and y_{goal}. As we can see, we still have a continuous decision boundary albeit more complex, which a neural network could learn.



(a) Four obstacle decision boundary.



(b) Four obstacle environment.

Figure 3.4: Decision boundary visual for four obstacles.

3.2.3 Network Architecture and Activation Functions

For our network architecture we chose to use a fully connected Deep Neural Network (DNN). We tested a variety of different DNN configurations and found the best performance with networks using layer shapes with one hidden layer using the activation function tanh on the hidden layer, and sigmoid on the output layer. This architecture was found through trial and error and had the best accuracy when classifying our obstacles. The results of training is presented in Chapter 4.

3.2.4 Neural Network Loss, Optimizer and Hyperparameters

For neural networks solving binary classification problems, we use the binary cross-entropy loss. This loss is ideal for classification with two classes [28]. Since each of our outputs are attempting to distinguish a binary class for an individual obstacle, we use this loss. For our optimizer we tried using Adam, rmsProp, and SGD. We have found the best results using Adam. For hyperparameters, we used the learning rate = 0.0001 and batch size = 64. For epochs we tested for at most 200 epochs. For our initial weights, we used default initialization from Keras [29].

For training our classifier, we implemented training code with Keras [29]. The metrics we used to analyze performance were sample and binary accuracy which we define in the following sections.

3.2.5 Sample Accuracy and Binary Accuracy

To analyze the performance of our neural network model, we use the following two metrics, sample accuracy and binary accuracy. We define these accuracies in equations (3.3) and (3.3),

respectively, and how they are used for neural network model analysis. In the following we define one data sample as one set of features and associated labels.

Sample Accuracy

We define sample accuracy by (3.3) and indicates the percentage of our samples, where our neural network model correctly identified the initialization that will lead to the global minimum shortest distance. For a sample to be considered identified correctly, all model outputs, corresponding to our obstacles' labels, had to be predicted correctly. This is the primary measure for the performance of our classifier as it tells us how often our neural network can predict the optimal initialization. Since there are N outputs for an N-obstacle neural network, the difficulty of predicting the optimal initialization scales with the number of obstacles.

$$\text{sample accuracy} = \frac{\text{samples identified correctly}}{\text{total samples}} \tag{3.3}$$

Binary Accuracy

For binary accuracy, we define it in (3.4). This accuracy is based on the percentage of outputs correct over total outputs. This differs from sample accuracy as binary accuracy measures the fraction of correct neural network outputs in terms of each of the N obstacles in our environment. We use this as another indicator to understand how our neural network is performing. Since it gauges the accuracy based on each output instead of based on the whole sample, we can look at the incremental progress of our neural network during training. This gives us a better idea of if the network is still improving during training. For the case of one obstacle neural networks, the binary accuracy is equivalent to the sample accuracy.

$$\text{binary accuracy} = \frac{\text{number correct outputs}}{\text{total outputs predicted}} \tag{3.4}$$

3.3 Data Generation

To generate the data set for training our classifier, we use circle visibility graphs combined with uniform cost search a.k.a. Dijkstra's method. To create a classifier with good generalization, we desire a large data set with a variety of different obstacle configurations. The following is our overall strategy for data generation. First, we generate an obstacle course with random obstacle placements and radii. Second, we build a visibility graph based on the obstacle configuration, our start, and our goal location. Third, we search our visibility graph using uniform cost search to find a trajectory for the shortest-distance path. Finally, we compare the trajectory of our path to the center coordinate of our obstacles to determine if our obstacle is above or below a given obstacle. This process is outlined in Figure 3.5 and explained in subsections 3.3.1 and 3.3.2.

3.3.1 Random Obstacle Course Data Set Generation

The following sections describe the parameters, assumptions, and methodology used for generating our data set.

Obstacle Course Generation Parameters and Assumptions

This section explains the parameters chosen and assumptions made for generating our data set. To improve the generalization of our classifier, we need to set limits on the parameters of our training environment. By doing this, we ensure all our samples fall into a given range and

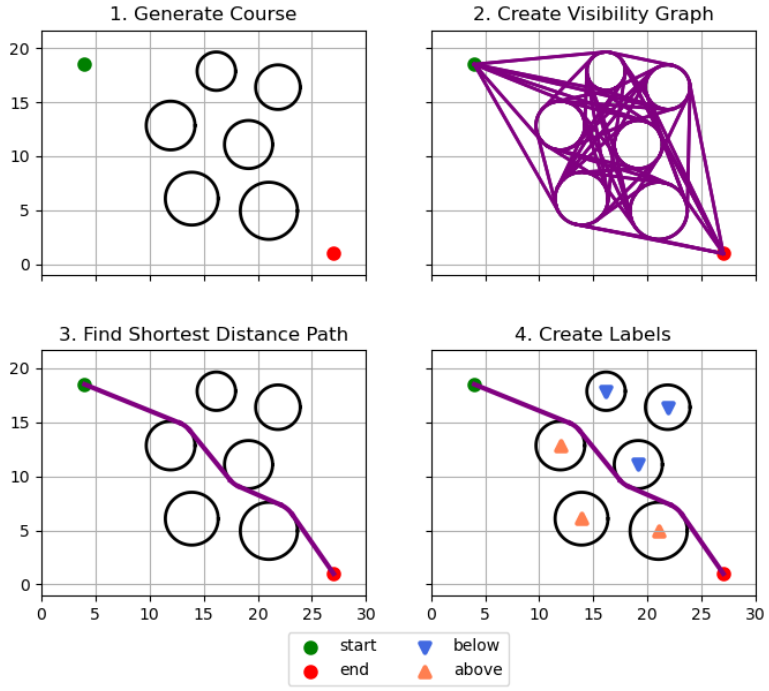


Figure 3.5: Overview of Label Creation Process.

give us a domain for which our classifier can perform. Without these restrictions, it would be impossible to generate enough data as all our features are real numbers. We can justify our classifier will still be able to generalize to problem environments outside the scale of our data generation environment as we can rotate and scale any given problem environment to the scale of our generated environments, and still receive a valid prediction from our classifier.

We first must limit the size of our environment. We have chosen that our environment is a square of thirty meters by thirty meters. This is approximately the size of two basketball courts placed side-by-side with their longer edges adjacent. In this environment we define three regions, the start, obstacle, and goal areas, as shown in Figure 3.6. All start points must be contained with the start area, all obstacles must be within the obstacle area, and all goal points must be contained with the goal area. We can add this restriction as all obstacles that are within the start and goal areas, which are not between the start and goal on the x-axis, do not need to be considered as they do not affect the optimal path. Our

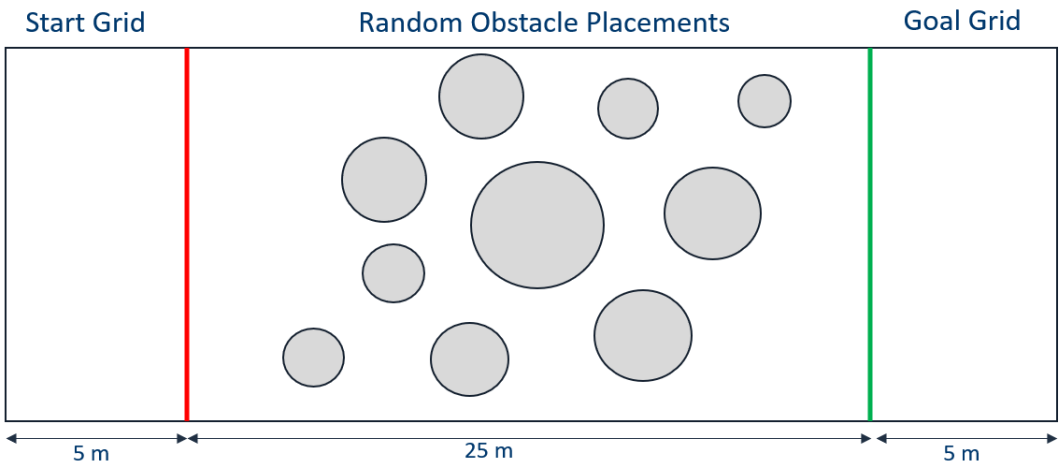


Figure 3.6: Visual of Generated Obstacle Course Parameters.

second restriction is on the size of our obstacles. We assume the size of our obstacles are between one tenth and ten meters. We place a restriction on the size of our obstacles to prevent our obstacles from being larger than our environment, and to increase data diversity by preventing our data set from consisting of one relatively very large obstacle augmented with a large number of relatively very small obstacles. This can also be seen as there is limited area for obstacles in our problem and as soon as a relatively very large obstacle is placed, there is only room for relatively very small obstacles in the remaining obstacle area, resulting in a lack of diversity in the size of most of the obstacles on the field. Another assumption made for our problem is that none of our obstacles are intersecting. Removing this assumption will be examined in future works.

Obstacle Course Generation Methodology

With restrictions on the magnitude of our features established, we can describe the method for generating an obstacle environment used in our data set. To generate our obstacle environment, we use the following obstacle placement procedure:

1. Sample the radius, r of our candidate obstacle from a truncated normal distribution.

Variable	Truncated Normal, $N(\mu, \sigma, a, b)$
r	$N(2, 0.5, 0.1, 10), N(4, 2, 0.5, 6)$
x_c	$N(15, 5, 5, 25)$
y_c	$N(15, 5, 0, 30)$

Table 3.1: Truncated normal distributions used for sampling obstacle center coordinates and radii. μ is the mean, σ is the standard deviation, a is the lower bound, and b is the upper bound.

Variable	Uniform, $U(a, b)$
x_c	$U(5, 25)$
y_c	$U(0, 30)$

Table 3.2: Uniform Distributions used for sampling obstacle center coordinates. a is the lower bound and b is the upper bound.

Details of the distribution are described in Table 3.1.

2. Sample the obstacle's center coordinates x_c, y_c from either a truncated normal distribution or a uniform distribution. Details of the distributions are described in Table 3.1 and 3.2. The tighter bounds on x_c are to prevent obstacles from being placed in the start and goal grids.

3. Check if obstacle placement intersects with any already placed obstacles. If so, re-sample our obstacles center coordinates x_c and y_c.

4. If we re-sample our center coordinates more than our max number of attempts, we return to step one to re-sample our obstacle radius.

5. Repeat procedure until desired number of obstacles have been placed.

Remark 3.3.1 *The use of a variety of distributions and distribution parameters was to increase the diversity of data in our data set.*

Once we have an obstacle course created, we now create a grid of start and goal locations within our start and goal areas to create our data set. The start and goal areas both have

dimensions of five meters by thirty meters. For the x and y coordinates of our start and goal area, we create a grid with size (x_values, y_values). We chose the grid size for both our start and goal with equal probability of size $(10, 30)$, $(10, 25)$, or $(12, 32)$ to promote variety in our data set. Each pair of start and goal locations creates one sample in our data set.

3.3.2 Visibility Graph and Label Creation

With our start, goal, and obstacle courses created, we can now create a visibility graph. We use the following procedure to generate a visibility graph and create our obstacle labels.

1. For each pair of obstacles, calculate the four bitangents to use as edges of the visibility graph.
2. For each bitangent, verify the candidate edge does not intersect any of the remaining obstacles. If it does, throw out the candidate edge.
3. Calculate the two bitangents between the start and every obstacle, and the goal and every obstacle.
4. Calculate the hugging edges between vertices on a given obstacle. This completes our visibility graph for this course.
5. Perform uniform cost search on the visibility graph to find the path of shortest distance from our start to our goal.
6. Compare the center y coordinate, y_c, to our path and assign a label of up if y_c is below the optimal path, and a label of down if y_c is above the optimal path.

3.3.3 Dataset Size

We used approximately twenty million samples to train each of our N-obstacle classifiers. We created twenty of these datasets, with one dataset for each of our number of obstacles from one to twenty.

3.4 Full Stack Combining Classifier and Optimization Solver

In this section we explain how our full stack works to combine our classifier and optimization solver in online, path finding applications. We first scale the obstacle course to match the dimensions of our thirty-by-thirty meter training environment. We pass these scaled features to our classifier which then outputs "up" and "down" labels for each obstacle. Next a human in the loop will create an initialization that matches the outputs of our classifier. Lastly, we pass this initialization to our optimization problem which then outputs the shortest distance trajectory. If our classifier labeled all our obstacles correctly, this output trajectory will be the global shortest path in our environment. This whole process is shown in Figure 3.7.

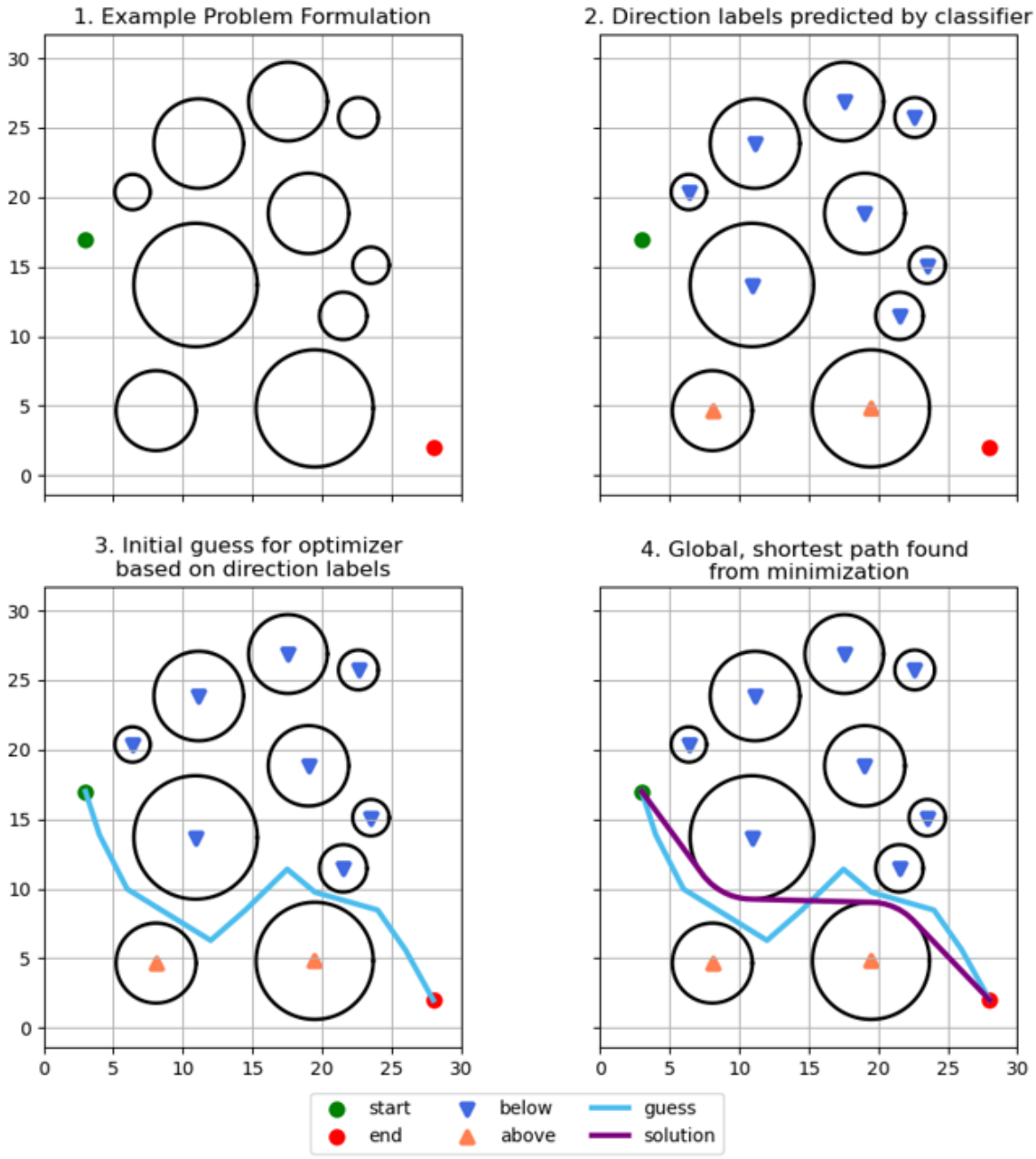


Figure 3.7: Workflow of full stack classifier and optimization solver.

Chapter 4

Results and Analysis

In this section we view the results and analysis from this project. We present results for the accuracy of our classifiers, as well as results on the generalization of our N-obstacle classifier to less than N obstacle courses, the coverage of our feature space by our data, and other results.

4.1 Training Results for One through Twenty Obstacle Classifiers

In this section we present the results from training and testing our N-obstacle classifiers for N equal to one through twenty. We present our sample accuracy and binary accuracy. In all tests, we split our data 90% training, 5% for validation, and 5% for testing. In this section all networks used one hidden layer with three hundred hidden units and the tanh activation function, and we took only our best result with this network design. This was done to limit the number of changed variables when looking at how our classifiers perform for various number of obstacles.

4.1.1 Sample Accuracy Results

In this section we present the sample accuracy results for our N-obstacle classifier. We show the results of our sampling accuracy on training, validation, and test data in Figure 4.1 and Table 4.1. Neither the obstacles courses from our test nor validation data was shown to the classifier during training.

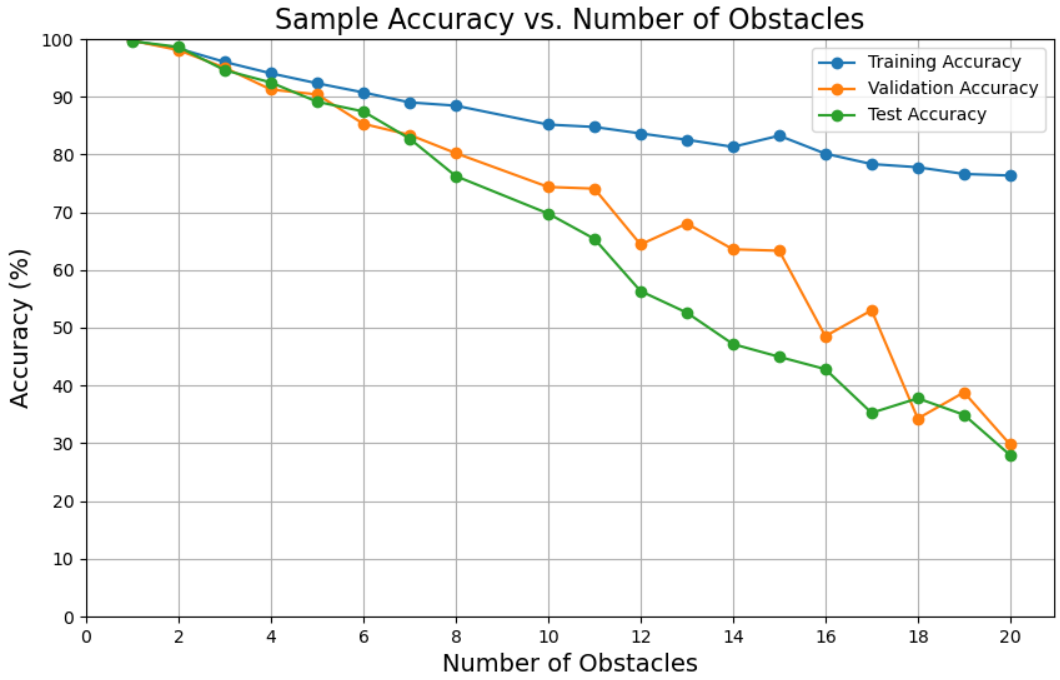


Figure 4.1: Sample Accuracy vs Number of Obstacles Classified.

We can see in Figure 4.1 as the number of obstacles went up, the test accuracy goes down. We believe this is due to the increase in our feature dimension by three for each additional obstacle to our classifier and our relatively constant number of samples in each obstacle dataset. With the increase in our feature space, this corresponds to an exponential increase in the number of samples required for our classifier to have good generalization across the feature space. This scaling in our number of samples generated offline for training is the trade-off we face for having good time complexity in online path planning.

Num Obstacles	Train Sample Acc (%)	Val Sample Acc (%)	Test Sample Acc (%)
1	99.68	99.73	99.61
2	98.36	98.07	98.62
3	96.02	95.03	94.60
4	94.05	91.25	92.47
5	92.33	90.40	89.14
6	90.75	85.31	87.45
7	89.03	83.40	82.76
8	88.46	80.23	76.27
10	85.18	74.38	69.78
11	84.77	74.09	65.39
12	83.64	64.42	56.35
13	82.56	68.04	52.62
14	81.32	63.59	47.17
15	83.26	63.33	44.96
16	80.14	48.54	42.83
17	78.34	53.03	35.26
18	77.79	34.28	37.78
19	76.63	38.82	34.92
20	76.36	29.77	27.93

Table 4.1: Sample Accuracy vs. Number of Obstacles

To examine this, we trained our 20-obstacle classifier on a twenty-obstacle dataset consisting of 66% of the number of samples in our full data set. As we can see in 4.2, as we decrease the number of samples trained on, our test sample accuracy decreases by more than 10%, which indicates a decrease in our generalization ability.

Data Samples	Train Sample Acc (%)	Val Sample Acc (%)	Test Sample Acc (%)
10564605	75.31	20.18	16.62
15845296	76.36	29.77	27.93

Table 4.2: Sample Accuracy for different dataset sizes.

4.1.2 Binary Accuracy Results

In this section we present the binary accuracy results for our N-obstacle classifiers. The results are shown in Figure 4.2 and Table 4.3.

Since this metric references the fraction of tested outputs with accurate predictions, it must be greater or equal to the sample accuracy found for a given N-obstacle classifier. This

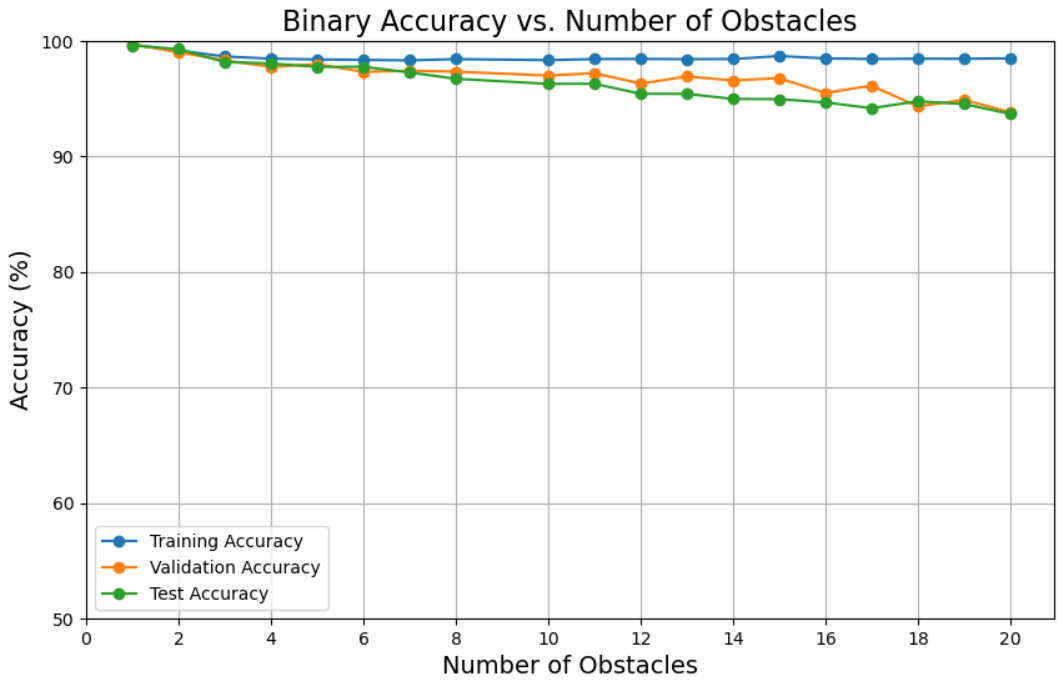


Figure 4.2: Binary Accuracy vs Number of Obstacles Classified.

metric indicates how close our network was to finding the optimal solution. These accuracy values typically correspond to one out of N outputs being incorrectly classified.

4.2 Best 20-Obstacle Classifier

In this section we show our best performing model architecture for a 20-obstacle classifier. The network design used is shown in Figure 4.3. This network utilizes one hidden layer with five hundred hidden units with the tanh activation function. We used the optimizer Adam, a learning rate of 0.0001, a batch size of 64, and 10 epochs. The results of training are shown in Table 4.4 and our loss and accuracy vs epochs is shown in Figure 4.4. Although the test sample accuracy for our model is low at 29.53%, we recognize this is significantly better than random guessing which has a test accuracy of $\frac{1}{2}^{20}$.

Num Obstacles	Train Binary Acc (%)	Val Binary Acc (%)	Test Binary Acc (%)
1	99.68	99.73	99.61
2	99.18	99.01	99.30
3	98.66	98.33	98.18
4	98.47	97.76	98.06
5	98.41	97.99	97.76
6	98.37	97.33	97.77
7	98.32	97.43	97.28
8	98.43	97.35	96.73
10	98.35	97.01	96.29
11	98.44	97.22	96.30
12	98.45	96.32	95.43
13	98.43	96.93	95.42
14	98.44	96.59	94.99
15	98.70	96.78	94.97
16	98.49	95.50	94.68
17	98.44	96.13	94.18
18	98.47	94.33	94.78
19	98.46	94.91	94.54
20	98.50	93.83	93.69

Table 4.3: Binary Accuracy vs. Number of Obstacles

Num Obstacles	Train Sample Acc (%)	Val Sample Acc (%)	Test Sample Acc (%)
20	81.58	30.90	29.53

Table 4.4: Best Accuracy Results for 20-Obstacle Classifier.

4.3 Feature Space Coverage of 20-Obstacle Classifier

For our classifier to have good generalization, we require a dataset that provides examples across the entire feature space. To do this we examine the distributions of the features of our twenty obstacle dataset shown in Figure 4.5. In this figure each row corresponds to the three attributes of an obstacle input. In our random course generation strategy, each obstacle was placed in the course in the order of the rows shown in Figure 4.5. We can see that our strategy provides good coverage of the entire feature space for our first obstacle, and the coverage decreases significantly with each obstacle placed. Thus, our dataset consisted of our first obstacle inputs containing large radii in the center of our course, while as we go progress through the rest of the obstacle course, we can see that the other inputs only have data for very small obstacles that are placed at the edges of the course. This dataset

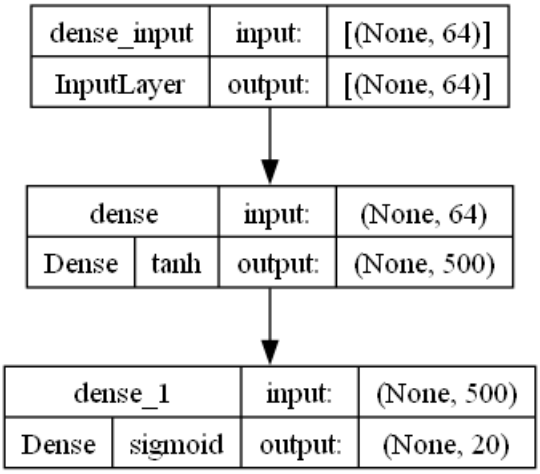


Figure 4.3: Best 20-Obstacle Classifier Architecture.

indicates that we have poor coverage of our feature space, and our predictions likely could be improved by modifying the data generation strategy to create examples from portions of the feature space that has few to no examples.

4.4 Effectiveness of N-Obstacle Classifier on less than N Environments

Due to our neural network having a fixed number of inputs and fixed number of outputs, we wish to examine the effectiveness of an N-Obstacle Classifier on datasets with less than N obstacles. To do this we use the technique from 3.2.1, where we apply zero to all outputs and only examine the first n outputs when measuring sample accuracy. The results are shown in Figure 4.6.

From the data in Figure 4.6, we can see that our 20-obstacle classifier generalizes well to unseen obstacle environments with less than twenty obstacles. Although the accuracy for obstacles higher than nine, was found to be less than 50% on unseen courses, we have to keep in mind that the best performance of this model on unseen twenty obstacle data was

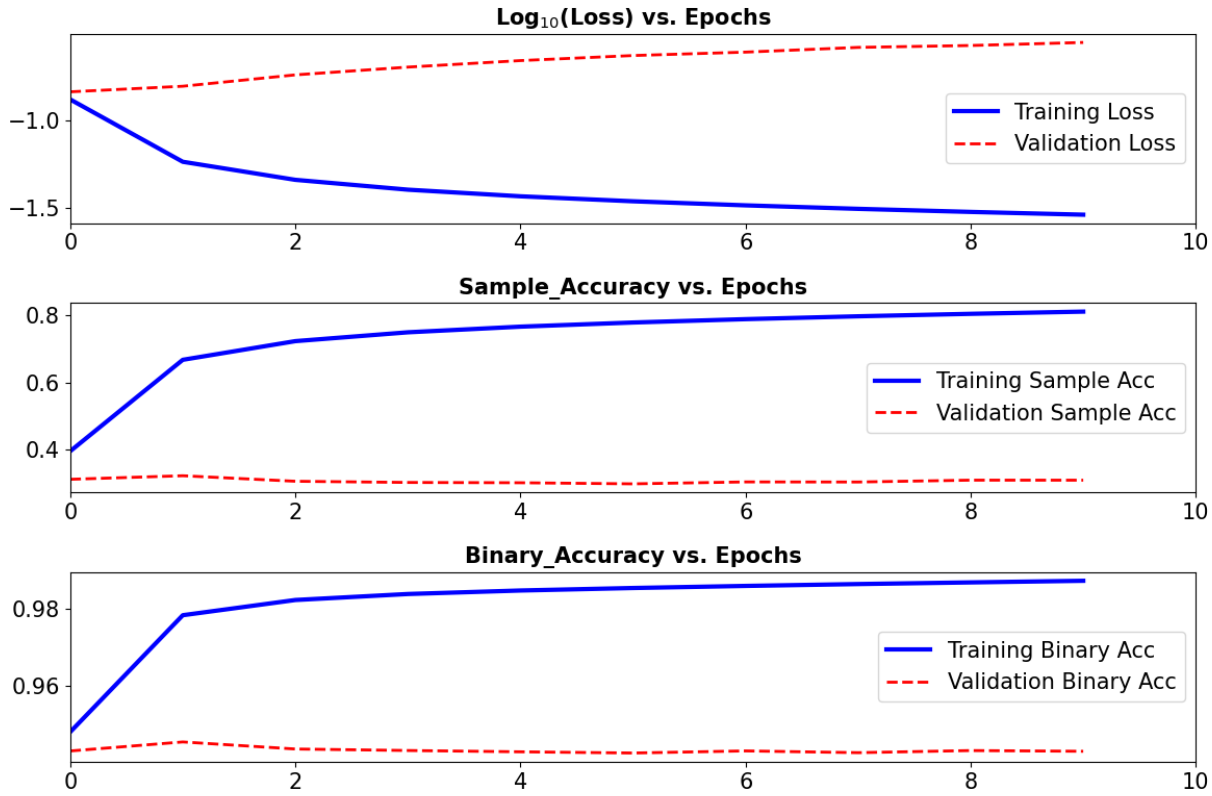


Figure 4.4: Loss and accuracy vs. epochs for best 20-obstacle classifier.

29.53%, so for our network to always be higher than this for less number of obstacles, shows that better sample test accuracy on our twenty obstacle data correlates to better sample test accuracy on courses with less than twenty obstacles.

If we compare the results from 4.6 with 4.1, however, we can see that a neural network designed for the correct number of inputs and outputs for the number of obstacles in the data always performs better than our 20-obstacle classifier. We can see that the percentage difference between the two is less than 10% for obstacle datasets for 15-19, and that the percent difference between the two grows as the number of obstacles in the obstacle dataset decreases. This result indicates that we can use higher number obstacle classifiers on courses with less than the number of obstacles it was trained with.

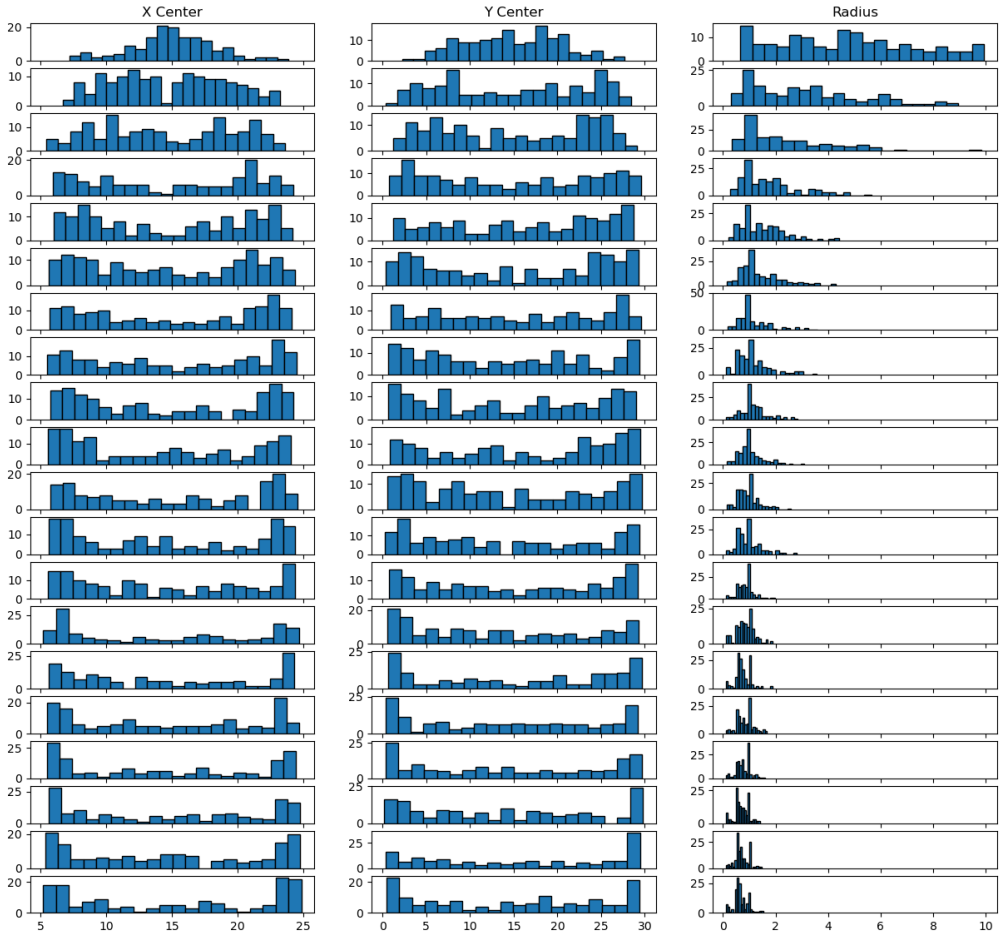


Figure 4.5: Distributions of Feature Data by Obstacle. Each row corresponds to one of twenty obstacles,

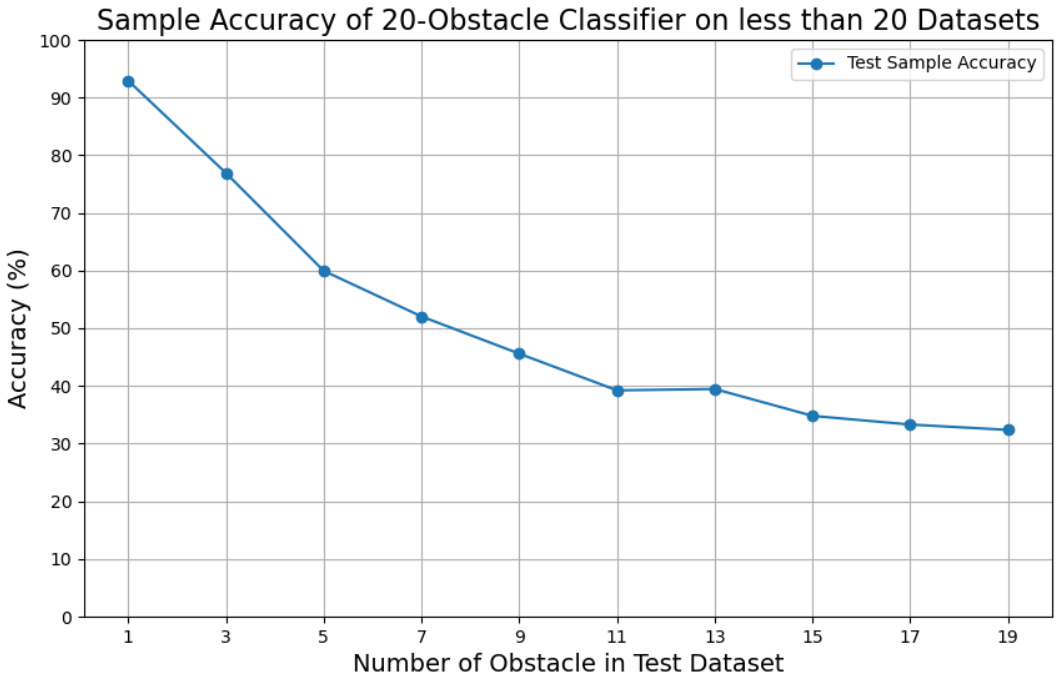


Figure 4.6: Twenty obstacle neural tested on less than 20 obstacle courses.

Chapter 5

Conclusion

In this thesis we presented a study on a data-based method for finding global minimum of nonlinear, non-convex optimization problems. Non-convex functions are hard to minimize due to their having saddle points, unknown number of local minima, very flat regions, or a combination of all three. This method allowed us to predict with high accuracy an initialization in the locally convex region of our problem and use the interior-point method to find the global minimum.

This study was performed on the shortest distance path planning problem. By leveraging that we can find globally shortest distance paths using circle tangent visibility graphs, as well as the knowledge that path planning problems with a restriction on direction and location of the start and goal nodes, have at most 2^n number of local minima, we can train a classifier. This classifier is based off a FFNN that is trained to classify the relative direction, "up" or "down", of the shortest distance path around each obstacle in the environment between the start and goal nodes. Using the output of the FFNN we can create an initialization path that follows the labels of the FFNN, and using a shortest path distance minimization with constraints, we can find the shortest distance path.

We were able to classify courses up to twenty obstacles with test accuracy on never-before-seen samples with 29.53% sample accuracy. We found that this method does not scale well with number of obstacles, with a 68% decrease in test sample accuracy from our 1-obstacle classifier. We attribute this issue to a need for more data samples as our feature space increases by three for each additional obstacle we classify, indicating an exponential increase in the amount of data needed for generalization across the feature space. This technique is recommended for path planning problems involving varied environments which require multiple queries of the shortest distance path from a varied set of start and goal locations, as our classifier has O(1) complexity scaling which leveraged with a fast convex optimizer, which allows for finding shortest distance paths quickly.

In future work, we recommend learning for more obstacle types such as ellipses and intersecting obstacles. We also recommend comparing the full stack of neural network and convex solver, to sampling based methods such as RRT*. We also recommend increasing the number of obstacles and gauge the performance of this network design. Another recommendation we make it to use a variety of distributions for generating data to get better coverage of our feature space. We also recommend trying data augmentation by switching around the feature columns in our dataset to increase the feature coverage of all our inputs. Lastly, we recommend automating our initialization creation from classifier labels to gauge the speed of our full stack solution.

Bibliography

[1] Issam Laradji. Non-convex optimization. `https://www.cs.ubc.ca/labs/lci/mlrg/slides/non_convex_optimization.pdf`. Accessed: 2023-11-01.

[2] Xi Luo, Ran Yan, Shuaian Wang, and Lu Zhen. A fair evaluation of the potential of machine learning in maritime transportation. *Electronic Research Archive*, 31:4753–4772, 07 2023. This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit `http://creativecommons.org/licenses/by/4.0/`.

[3] Marina Danilova, Pavel Dvurechensky, Alexander Gasnikov, Eduard Gorbunov, Sergey Guminov, Dmitry Kamzolov, and Innokentiy Shibaev. Recent theoretical advances in non-convex optimization, 2021.

[4] Chris De Sa. Non-convex optimization. CS6787 Lecture slides, 2017. Cornell University, Fall 2017.

[5] Steven M. LaValle. *Planning Algorithms.* Cambridge University Press, 2006.

[6] Tomás Lozano-Pérez and Michael A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Commun. ACM*, 22(10):560–570, oct 1979.

[7] Tomas Lozano-Perez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, C-32:108–120, 1983.

[8] John Canny. *The complexity of robot motion planning.* MIT press, 1988.

[9] Ahmed H. Qureshi, Anthony Simeonov, Mayur J. Bency, and Michael C. Yip. Motion planning networks, 2019.

[10] Steven M. LaValle. Rapidly-exploring random trees : a new tool for path planning. *The annual research report*, 1998.

[11] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[12] Jonathan D. Gammell, Siddhartha S. Srinivasa, and Timothy D. Barfoot. Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014.

[13] Jonathan D. Gammell, Siddhartha S. Srinivasa, and Timothy D. Barfoot. Batch informed trees (bit*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, May 2015.

[14] L. E. Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 79(3):497–516, 1957.

[15] Thomas Andreae. Some results on visibility graphs. *Discrete Applied Mathematics*, 40(1):5–17, 1992.

[16] Subir Kumar Ghosh and Bodhayan Roy. Some results on point visibility graphs. *Theoretical Computer Science*, 575:17–32, 2015. Special Issue on Algorithms and Computation.

[17] Amit Patel. Circular obstacle pathfinding, 2017. Accessed: November 22, 2023.

[18] F. Bullo and S. L. Smith. *Lectures on Robotic Planning and Kinematics*. 2022.

[19] J. P. Ballantine and A. R. Jerbert. Distance from a line, or plane, to a poin. *The American Mathematical Monthly*, 59(4):242–243, 1952.

[20] R. Larson. *Precalculus: A Concise Course*. Cengage Learning, 2013.

[21] Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data*. AMLBook, 2012.

[22] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[23] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[24] Ken-Ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2(3):183–192, 1989.

[25] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, December 1989.

[26] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993.

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. Book in preparation for MIT Press.

[28] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[29] François Chollet et al. Keras. `https://keras.io`, 2015.

[30] Lars Otten. LaTeX template for thesis and dissertation documents at UC Irvine. `https://github.com/lotten/uci-thesis-latex/`, 2012.

[31] Eric W. Weisstein. Circle-circle intersection. MathWorld–A Wolfram Web Resource.

[32] J.-D. Boissonnat, A. Cerezo, and J. Leblond. Shortest paths of bounded curvature in the plane. In *Proceedings 1992 IEEE International Conference on Robotics and Automation*, pages 2315–2320 vol.3, 1992.

[33] Wikipedia contributors. Dubins path, 2023. Accessed: 11/22/2023.

[34] Abhishek Paudel. Motion primitives based path planning with rapidly-exploring random tree, 2022.