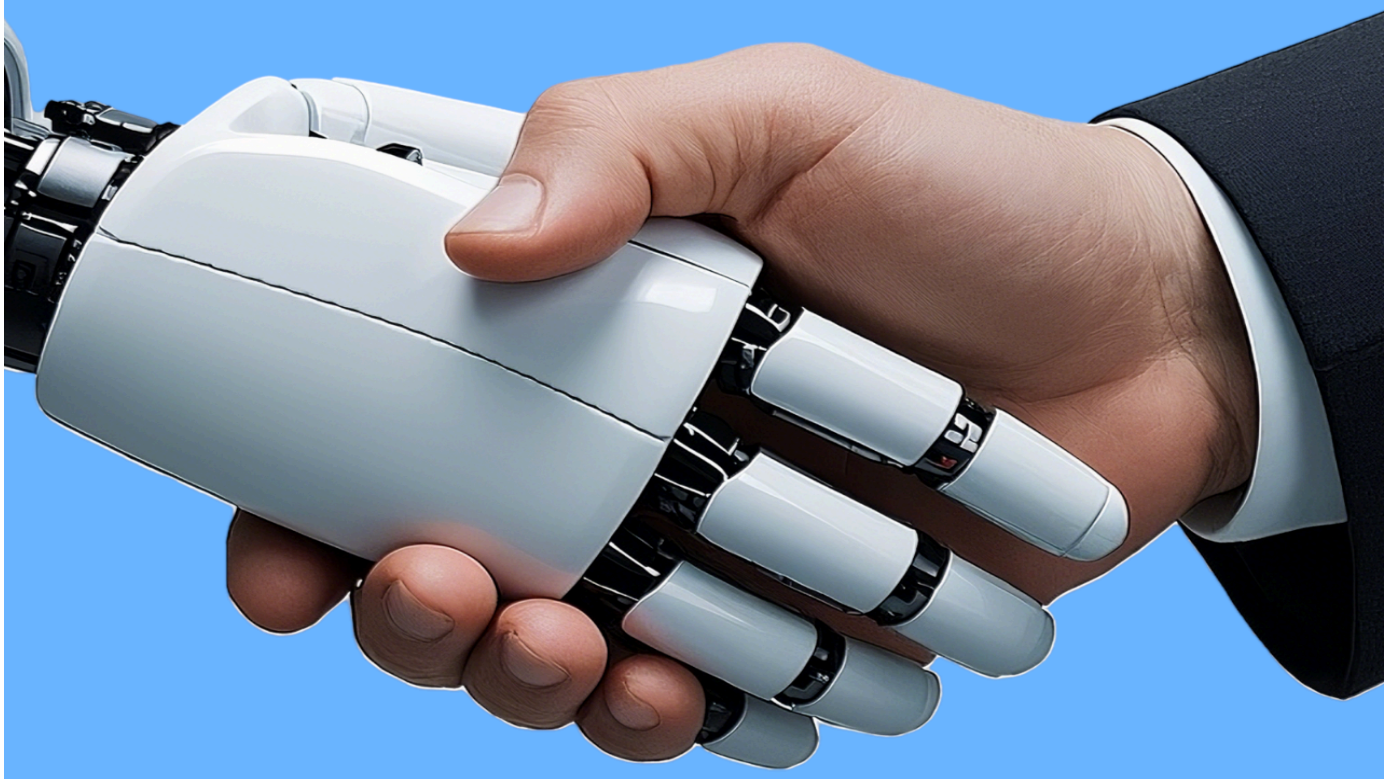
Robotics: Perception & Planning



Samuel Kinstlinger
Andrew Xu
Sameer Chawla
Brian Fugh

Table of Contents

Preface

This textbook is designed to comprehensively cover the core algorithms and paradigms in robotics, with a focus on machine perception and planning. It is intended for both undergraduate and graduate students seeking a deep, principled understanding of robotics.

The central teaching philosophy of this book is to justify each algorithm from first principles, showing when and why it is optimal. Rather than treating techniques as black boxes, we begin with the foundational goals of robotics and derive each algorithm as a solution to those goals. This derivational approach enables students not only to apply algorithms effectively, but also to understand the specific conditions under which they are optimal. It also empowers them to recognize when an algorithm may be suboptimal due to missing or flawed insights. Once students identify these gaps in justification, they are equipped to address them—potentially leading to improved or novel algorithms. In this way, the book not only teaches existing techniques but also cultivates the analytical mindset necessary to advance the field.

Throughout the text, we support this learning process with concrete examples, intuitive explanations, and visualizations that illuminate key insights and clarify the rationale behind each method. Our goal is to equip students with the tools to analyze, apply, and improve robotics algorithms with confidence and rigor.

If at any point while reading this textbook you cannot recall a term's definition, please refer to the glossary at the end. To fully benefit from this textbook, it is essential to have the equivalent of one university class in statistics, multivariable calculus, linear algebra, computer programming, and robotics.

About the Authors

This textbook is a joint effort by four educators and researchers in robotics, machine learning, perception, and planning at the University of Maryland, College Park: Samuel Kinstlinger, Andrew Xu, Sameer Chawla, and Brian Fugh. Each author brings research and industry experience spanning both perception and planning, contributing complementary expertise to the project.

The textbook originated as an internal guide for students joining their research team. Frustrated by the prevalence of black-box explanations—where textbooks often present only the procedural steps of algorithms without conveying the underlying reasoning—the authors set out to create a resource that emphasizes deep, principled understanding. This textbook reflects their commitment to transparent, derivation-driven pedagogy that equips students not only to use robotics algorithms, but to truly understand and improve them.

Section I: Introduction to Cognitive Robotics

Chapter 1: Introduction to Perception & Planning

As humans, we constantly strive to improve our quality of life and sustain the achievements we have made. This often involves maintaining, modifying, or constructing physical entities. Whether in industry, manufacturing, or daily life, these interactions are essential. In factories, people assemble products and transport materials; at home, we cook, clean, and make repairs; and in agriculture, we plant and harvest crops. These activities highlight the widespread human need to engage with and manipulate our environment. Additionally, humans rely on real-world data about ourselves and our environment, such as crop information or fish populations.

Due to our evolutionary history, humans face significant biological and cognitive limitations. We evolved to conserve energy in environments where food and water were scarce, making us prone to fatigue and distraction during prolonged tasks. As generalists, we had to perform many functions moderately well rather than excelling in a single one. Trade-offs between abilities—like strength vs. endurance or speed vs. precision—led to limits in physical power, reaction time, memory, and processing speed. We also evolved in relatively stable environments, leaving us poorly adapted to extremes like deep oceans, outer space, or disaster zones. Finally, even when we can perform a task, we often wish to avoid tasks that are unpleasant or unmotivating.

These limitations highlight the need for alternatives that can match and occasionally exceed human capabilities in certain situations—alternatives that are not constrained by biological limits. This points to inanimate devices designed with specific goals in mind. Specifically, we require machines that quickly perform complex calculations and interact with their environment. These machines must be physical systems capable of applying force, controlling motion, or performing specific tasks using mechanical, electrical, or other forms of energy. The primary goal of these machines is usually to change the position of an object in some way. This could be some external object, such as medical supplies, or some internal/attached object such as an attached camera or person inside a car.

For a machine to interact with its environment, it must apply forces at specific contact points, a process known as **actuation**. At each moment, the machine must precisely control its actuation mechanisms to achieve the desired outcome. These actuation mechanisms are typically motors which input electric signals and output some force or torque. Each unique configuration of these controls is referred to as an **action**. Since these machines operate in a continuous world and motor changes are usually continuous or at least very small, action spaces are generally continuous. To successfully perform its tasks, the machine's design must enable all necessary

movements to interact with external objects. These possible actions form the machine's **action space**. Each action is a sample chosen from the action space. To operate continuously in dynamic environments without requiring excessive human labor for long periods or being constrained by human reaction times, precision, etc., this acting must occur **autonomously**. For example, mechanical arms in manufacturing use motorized joints to weld, assemble, or paint car parts based on pre-planned instructions or autonomous algorithms.

Machines are usually designed so they can configure themselves and interact with their environment to cause the desired outcomes in a variety of situations. While a machine may have many possible actions, some are more advantageous than others and are more likely to align with the stakeholders' goals. For example, prioritizing certain tasks can lead to faster completion, which is often more beneficial. Similarly, different navigation paths affect outcomes such as travel time, crash rates, or energy consumption. In these cases, the machine may need to optimize for speed, safety, or efficiency, depending on the context. To guide the machine's design and programming, and to compare different approaches, it is essential to create a **reward function**. This function assigns numerical values to outcomes such as time, battery life, or damage sustained, which quantify the machine's performance. For instance, an autonomous food stocking machine's reward function might factor in both the time taken to reach a shelf and any obstacles encountered along the way. To provide clear direction, the higher the reward function, the better the performance. In this situation, negative rewards act as punishments. The machine should act to maximize this reward or minimize costs. Since a machine can't know the exact result of its actions, it must instead predict the action that maximizes the expected reward. For example, a self-driving car approaching a four-way stop doesn't know exactly how nearby human drivers will behave, but it must choose whether to proceed, wait, or inch forward. It predicts which action is most likely to result in a smooth and safe crossing—i.e., which one maximizes expected safety and progress.

In different situations, the same action or sequence of actions may lead to varying results. This means that the actions which maximize the reward function differ depending on the context. For instance, factors like the machine's location and velocity, its proximity to nearby obstacles, and the type of terrain it's navigating on all influence the effects of specific actions. To determine the action that has the greatest expected success, it is necessary to gather relevant data. While perfectly predicting the outcomes of different actions requires comprehensive data, we can often make accurate estimations based on key variables. To collect and store this data, the machine relies on electromechanical devices that measure specific physical attributes of the environment—these are known as **sensors**. At any given moment, the data we read from these sensors is referred to as an **observation**, with the full range of possible observations known as the **observation space**. For example, an observation from a machine moving wood pallets might include its position from wheel encoders, its angular velocity from an IMU, a point cloud from a

LiDAR sensor, and the weight of the pallet from a strain gauge or load cell. Similarly, a self-vacuuming machine like a Roomba uses infrared and bump sensors to detect walls, furniture, and dirt. A self-driving car, on the other hand, processes camera images, LiDAR data, and GPS to detect lane markings, nearby vehicles, and pedestrians.

To ensure the machine selects optimal actions, we need a function or algorithm that maps observations to actions. However, much of the relevant data for determining the best action may not be immediately observable due to technological or situational limitations, a concept known as **partial observability**. Additionally, some data has the potential to be represented in a simpler form that is easier for the system to use in decision-making. First, observations are instantaneous snapshots, whereas states often incorporate temporal context (e.g., velocity is derived from changes in position over time). For instance, a sensor cannot predict where a moving object will be in five seconds. To make use of past insights or forecast future outcomes, machines must maintain a model of how the environment evolves. Second, some data may involve human-made labels or categories that do not correspond directly to physical quantities in the real world but derive meaning from their functionality. Examples of this include determining whether an area is occupied by an obstacle or whether a fruit is an apple or an orange. Third, observations can be high-dimensional (e.g., images with millions of pixels or points), particularly with camera and LiDAR data. However, these large data sets can often be represented more efficiently with simplified, lower-dimensional states, which make action calculation computationally feasible. For example, it is less memory-intensive to store the object detected by an image instead of all pixel data, or whether a region is occupied instead of a series of LiDAR points. Lastly, due to technological limitations, some information is not directly observable. For example, an IMU provides data on linear acceleration, angular velocity, and orientation. To derive position and linear velocity, additional mathematical operations are required.

Overall, to process information in a format that is usable for planning algorithms, machines must employ techniques to extract meaningful insights from the data they observe. An individual vector that represents the relevant data to determine the optimal action at a given point in time is known as the **state**. For example, the state of a machine that moves wood pallets from one location to another could include the machine's position, velocity, the location of the nearest obstacle, a boolean indicating whether it is currently carrying a pallet, the pallet's type, data about the floor surface, and whether the target has moved in the last 10 seconds. Similarly, consider a self-driving car navigating through a city. The state could include the car's position (GPS coordinates), speed, steering angle, the distance to the closest vehicle or pedestrian, the status of traffic lights (red, yellow, green), the car's current lane, road conditions (e.g., wet or dry), the car's battery level, and the time of day. The set of all possible states is known as the **state space**.

We assume that changes in each relevant state variable will have physical effects that can be observed. For example, different machine velocities will affect IMU readings. This means that the sensor array of a machine must ensure that there are sufficient physical observations that are influenced by the state such that one can deduce the state from the observations. Due to fundamental physical laws, such as kinematics and dynamics, and the inherent relationships within the system being measured, there are true equations that map certain physical quantities to others. For example, in a car, the relationship between the motor's rotation and the car's forward movement is defined by these physical laws. By understanding these principles, we can derive mathematical functions that relate measurable inputs to the desired outputs represented in the state vector. This means the machine must use algorithms to map observations to states, a process referred to as **information processing** or **machine perception**.

By design, the state contains all relevant information needed to make decisions and predict future states, incorporating all current and past measurements. As a result, past states are not needed directly, since any useful information they held is already encoded in the current state. For example, a self-driving car relies solely on its current position, velocity, and map of nearby obstacles to decide whether to brake or change lanes—without referencing its entire driving history. This allows us to assume that future states depend only on the current state, meaning the optimal action is based solely on the current state. This assumption is known as the **Markov Property**. This means that our planning/acting algorithms will take in the current state and map it to the optimal action. The machine must then employ algorithms to map states to actions, or sequences of actions, that will probabilistically maximize the reward function, a process known as **machine planning**. This means we need electromechanical computational machines that can autonomously sense data from their surroundings, process that data to extract relevant information, use this information to determine the probabilistically optimal actions, and actuate accordingly. This type of machine is known as a **robot**.

The electrical and mechanical aspects of robotics primarily draw from the fields of physics, electrical engineering, and mechanical engineering. Advances in these areas enhance the physical capabilities of robots, shaping their mechanical and electrical components, including sensing and actuation systems. Sensors convert physical data into electrical signals, while motors convert electrical signals into forces or torques. These motors generally actuate according to velocities or forces/torques specified by the user. The unique challenges in robotics require a processor to transform electrical signals from the sensors into signals for the actuators.

However, the unique challenges of robotics arise from the need for complex perception and planning abilities. The processor must then be directed to perform this transformation. To consistently execute and communicate optimal perception and planning processes, humans must develop clear and structured **algorithms** to achieve specific goals. The subfield of robotics focused on these algorithms is known as **cognitive robotics**.

This textbook explores the primary algorithms and areas within cognitive robotics, detailing the goals they aim to fulfill, when and why these goals are important, and how the algorithms effectively achieve them. By the end of this textbook, you should have the algorithmic knowledge required for most basic robotics tasks and the ability to delve deeper into any individual subfield of cognitive robotics as needed. If at any point while reading this textbook you cannot recall a term's definition, please refer to the glossary at the end. To fully benefit from this textbook, it is essential to have the equivalent of one university class in statistics, multivariable calculus, linear algebra, computer programming, and robotics.

Chapter 2: Machine Learning

2.1 - Introduction to Machine Learning

Robots often face situations where not all relevant information is immediately available, due to either technological limitations or the nature of the environment. For instance, a robot may need to predict the likelihood of different future states to make decisions, but since the future cannot be sensed, this information is inherently unavailable. Additionally, certain physical data may remain inaccessible because the necessary sensors either don't exist or are inadequate. This scenario, where the robot operates without full knowledge of its state and environment, is known as **partial observability**.

Physical realities, rules, and relationships create true mathematical connections between different physical quantities. For example, $e = mc^2$, $v = \frac{dx}{dt}$, etc. This implies that the information needed to infer unobserved quantities is often embedded, at least partially, within the observed data. In other words, there are inherent functions that link observed inputs to their corresponding outputs. By leveraging these functions, we can predict the values of unobserved quantities based on the available observed information.

For each situation, there is a desired degree of accuracy or maximum allowable deviation from the true value. A prediction is considered **Approximately Correct** if it stays within this margin of error. For example, a prediction of 4.1 m when the true value is 4 m is approximately correct if the situation warrants a margin of error of .2 m. However, approximations do not always produce outputs that meet this standard. In practice, it is often sufficient for only a certain percentage of predictions to be approximately correct. If an approximation achieves this level of accuracy with a high enough probability, it is referred to as **Probably Approximately Correct (PAC)** (Valiant, 1984). This refines our goal of approximating a function, to finding a PAC approximation of that function.

Due to inherent system randomness or missing pieces of information, we will rarely ever be able to perfectly predict unobserved quantities. As shown by statistics, the optimal predictor is the expected value because it minimizes the posterior variance between the prediction and observed values, and it also leads to errors that tend to cancel out over time. This means that for each input, we simply want to predict the expected value of the output. This formalizes our function approximation problem for an output y given inputs x as follows:

$$y_{pred} = E[y|x] \tag{2.1}$$

To be able to calculate the expected value of y, we must identify the function that maps observed data as inputs to unobserved data as outputs. If x values are continuous, then we must

have some continuous function that works for previously unseen values of x. Typically, we rely on our understanding of physical principles—such as kinematics, statics, and dynamics—to estimate these functions. This process helps us derive observation models and state transition models used in sensor fusion. However, due to limitations in technology or knowledge, it is often not feasible to directly derive a precise function linking inputs to outputs. Additionally, situational limitations often prevent relevant information from being observed, such as which action is optimal or what some state variable will be in the future. In such cases, we must approximate these functions with other methods.

In many situations, humans can manually approximate these functions using intuition or patterns that they see in the data. However, if there is a lot of data, humans do not have a strong intuition about a system, or there are complex patterns in the system, then humans cannot properly calculate sufficiently accurate approximators. In this case, we must defer to mathematics and computation to learn these functions that humans cannot.

A function fundamentally maps inputs to outputs, with each input corresponding to a unique output. In the context of learning, each data point represents a sampled input-output pair, where the input is drawn from an underlying probability distribution, and the output is generated by the function, possibly with noise.

To estimate the expected value of a function empirically, we must use sampled data drawn from the underlying distribution of inputs and their corresponding function outputs. This is the only way to gain insight into the function or evaluate the performance of an approximation. This allows us to construct an algorithm that learns to approximate a function based on observed input-output pairs and then generalizes to unobserved data. For our notion of probable correctness in learning to remain valid, the underlying input distribution must remain stable. Likewise, for our notion of approximate correctness to hold, the function mapping inputs to outputs must remain stable. This assumption is often referred to as **stationarity.**

This data that we learn an approximation for is known as **training data**. To build an effective model, it must learn the underlying complex patterns in the training data rather than oversimplifying them. Failure to meet this requirement will lead to insufficient approximations in the real-world. This issue is known as **underfitting**.

The purpose of finding such a solution is so we can generate outputs on new data for which we only have inputs. This means that the function must perform well on unseen data. This is known as **generalization**. However, the training data may not perfectly reflect the true function. For example, there may be outliers, noise, and/or random fluctuations in the input data. If the model has the ability to perfectly conform to the data, it will learn these patterns that are not reflective of the true function, and will thus perform poorly on unseen data. These issues are particularly significant in regions of the input space with sparse data. In such cases, the limited samples may not accurately represent the true underlying function in that region leading to more

pronounced noise and randomness, increasing the likelihood of inaccurate approximations. Essentially, the fewer the samples, the greater the probabilistic amount that the expected value of the samples deviates from the true expected value. Additionally, in any region, there may be multiple different functions which can approximate the data well. A very complex function may be able to fit the data perfectly, but not in ways that reflect the true function. This overall problem is known as **overfitting**. A graph of a function of $-x^2 + b$ with some noise that a model with the ability to conform perfectly to a function could overfit to is shown below:

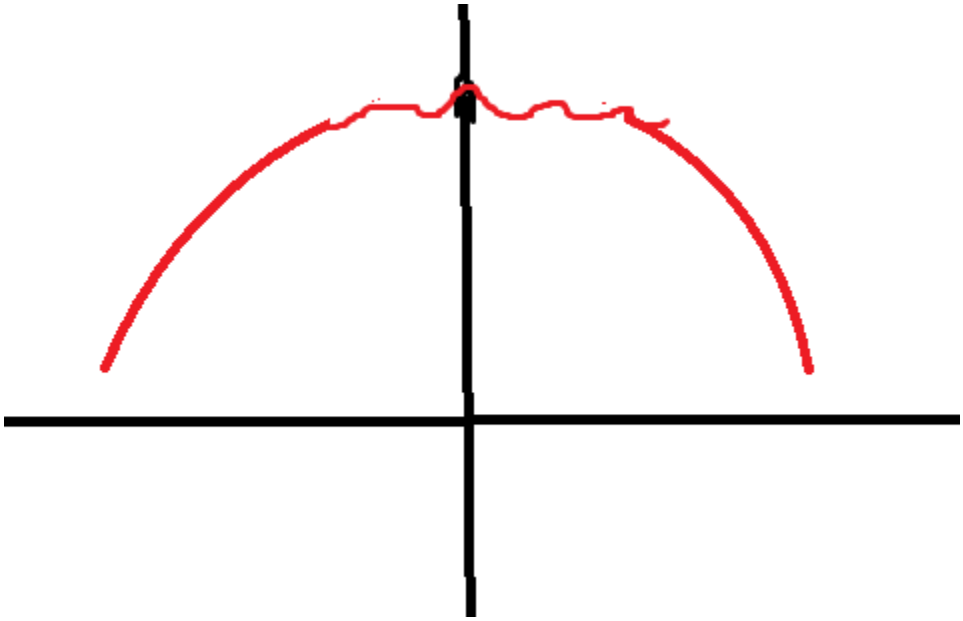


Fig. 2.1: Illustration of overfitting in a regression task. The true underlying function is a smooth downward-opening parabola (e.g., $-x^2+b$), but the red curve represents a highly complex model that conforms to every fluctuation and noise point in the training data. This results in a jagged approximation that fails to generalize to unseen data, especially in regions where data is sparse—demonstrating the classic behavior of an overfit model.

To ensure that any potentially PAC approximation is PAC on unseen data, we usually separate a small portion (~20% - 30%) of our data to not be used on training, and instead for evaluation. The approximation which performs best on the unseen data is then taken as the best available approximation. This unseen data is known as the **testing data**.

The best function is the one that most accurately predicts the ground truth based on the inputs. When working with predicting physical data, the data will be numerical. Predicting such data is known as **regression**. In regression analysis, each data sample is associated with a label that indicates the actual numeric output (y) and the predicted output (y′). Since our goal is to find the expected value, we can simply find the minimizer of the prediction variance. Since we have a finite amount of data, this will be the minimizer of the sum of the squared difference between the prediction and true data. This metric is known as **Mean Squared Error (MSE)** (Gauss, 1809). The formula for MSE is given below:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y'_i - y_i')^2 \quad (2.2)$$

MSE is an example of a function that quantifies the quality of predictions. Such functions are called **Loss Functions**.

Therefore, to find the best function, we must collect examples of inputs and their corresponding outputs and identify the function that best predicts each output from its input. This is part of a broader field involving learning to make predictions and approximate functions from data via optimization known as **Machine Learning (ML)**. The specific task of figuring out true functions from input-output pairs is a subsection of ML known as **Supervised Machine Learning (SML)**. A pictorial representation of of SML for regression is shown below:

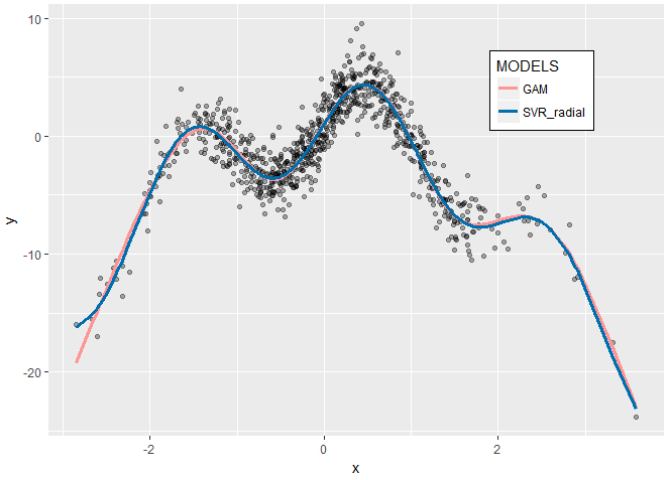


Fig. 2.2: Example of supervised machine learning for regression. The black dots represent training data composed of input-output pairs. Two models attempt to learn a function that maps inputs to outputs. Both curves demonstrate different learned approximations of the underlying pattern in the data, illustrating the central task of supervised regression: identifying a function that generalizes well from noisy examples.

Outside of very special and complex cases, each function can be broken down into individual simple operations on the inputs, such as summation, exponentiation, and trigonometric functions, each having a specific number of inputs and a single output. For example, in the expression $z = \sin(x^2+y)$, the function can be decomposed into three steps: first, squaring the x value, then adding the result to y, and finally taking the sine of the resulting sum to obtain z. As another illustration, if each input is linearly related to the output, the function would consist of scalar multiplication of each input followed by the summation of these products with a constant scalar. The existence of a true function mapping inputs to outputs implies that there are genuine operations performed on the inputs, along with a specific organization and composition of these operations. In order for some function we learn to be used on continuous data we did not exactly

observe, our function must have a continuous structure. This structure is referred to as the function's **format**, also known as its **inductive bias**.

Many operations, such as exponentiation, scalar multiplication, and scalar addition, involve a specific constant numerical value associated with the operation on the input, which is independent of the input itself. This constant is known as a **parameter**. In exponentiation, the parameter represents the exponent to which the input is raised; in scalar multiplication, it is the value by which the input is multiplied; and in scalar addition, it is the constant added to the input.

This implies that each function has a set of parameters. For a linear function, there are coefficients associated with each input and a constant value added to the sum of the products as shown below:

$$y' = \sum_{i=1}^{n} c_i x_i + c_{n+1} \tag{2.3}$$

These scalar coefficients and the added constant constitute the parameter vector. In this case, the number of parameters equals the number of inputs plus one. The existence of a true function implies that there is a corresponding true parameter vector. Each function is fully defined by its format and parameter vector, meaning that to identify a true function, we must determine both its true format and its parameter vector.

We often possess some background knowledge regarding the format of, or the operations that comprise, the true function or an approximation of it. For instance, we may know that each input is linearly related to the output, that there is a polynomial relationship between the inputs and the output, or that these assumptions are at least approximately true. However, we usually have little or no knowledge about the parameter vector. For example, we may not know the true coefficients associated with each input in a linear function or the constant to be added at the end. This refines our goal from finding the true function to identifying the true parameter set. Consequently, given a series of inputs, their corresponding outputs, and the function format, we must determine the parameter vector that minimizes the MSE between the predicted output and the ground truth.

2.2 - Optimization

2.2.1 - Iterative Optimization

For a given function format and set of inputs, the function predictions are a continuous function of the parameter vector. Additionally for a given set of inputs and true outputs, the MSE is a continuous function of the predictions. This means that for a given function format and set of true input-output pairs, the MSE is a continuous function of the parameter vector. This

continuous function inputs the parameter vector and outputs the MSE. This continuous function is known as the **Objective Function**.

This means that we can simply perform multivariable optimization, to find the inputs to the loss function which minimize its output. To solve this form of machine learning analytically, we can find the critical points of the loss function. Taking the partial derivative of this loss function with respect to each parameter and setting it equal to zero yields a system of equations. Substituting in the input data and corresponding ground truth values allows us to solve for the optimal parameters. This approach is known as an **analytical method**, as it relies on having a closed-form expression of the loss function and uses calculus to directly find the global minimum. However, there are some issues with this approach.

- Many true functions have a non-linear format, incorporating exponentiation, logarithms, trigonometric functions, etc. Non-linear models often don't have a solution that can be represented with a finite number of algebraically calculable operations, known as closed-form solutions, making it impossible to solve for the optimal parameters directly. For example, $y = \sin(x)$ does not have a closed-form optimization solution. This means that an alternative to analytical methods must be used that does not directly calculate the optimal points.
- In many optimization problems, especially for nonlinear functions, the variables are interdependent. This means that the optimal value of one variable depends on the values of other variables. When calculating partial derivatives separately, you're treating each variable in isolation, as if changing it doesn't affect the others. However, in reality, the changes in one variable can influence the optimal values of other variables due to their interdependence. This can make solving the system analytically by setting partial derivatives equal to zero challenging or even impossible for complex, multi-variable functions. This shows that our new approach must take into account the effects of changes to each variable on optimality of a specific value of another variable. This means that our new approach must be able to explore changes in the parameters on changes in the cost function.
- When there are a lot of features and examples, analytical methods can be very computationally intense. Solving the optimization problem at once is not feasible in these situations. This shows that our new approach must not solve the optimization problem in one calculation.
- Non-convex cost functions have multiple local minima, which analytical methods can't handle well because they generally assume convexity for global optimization. This means that an analytical method cannot algebraically solve for a single point. This means that our new approach must be able to handle multiple local minima.

- Many cost functions may have one or more stationary points that are not an extremum, known as saddle points. This means that the partial derivatives will be zero but the point will not be a minima. Images of this for 1D inputs and 2D inputs respectively are shown below. Our new approach must be able to handle saddle points.

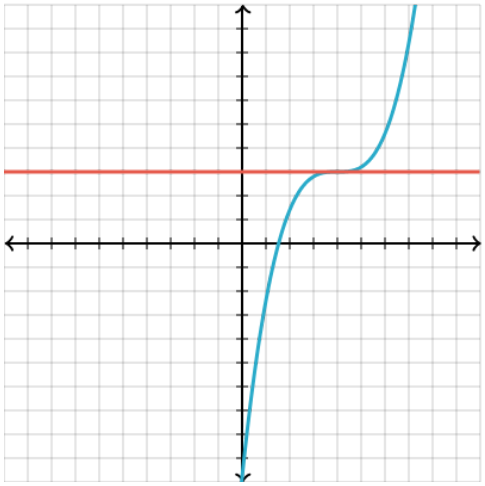


Fig. 2.3: A saddle point in a 1D cost function. The red horizontal line represents the zero-gradient threshold. The blue curve has a flat region where the derivative is zero, but the point is not a minimum. This highlights the challenge of distinguishing true minima from saddle points during optimization.

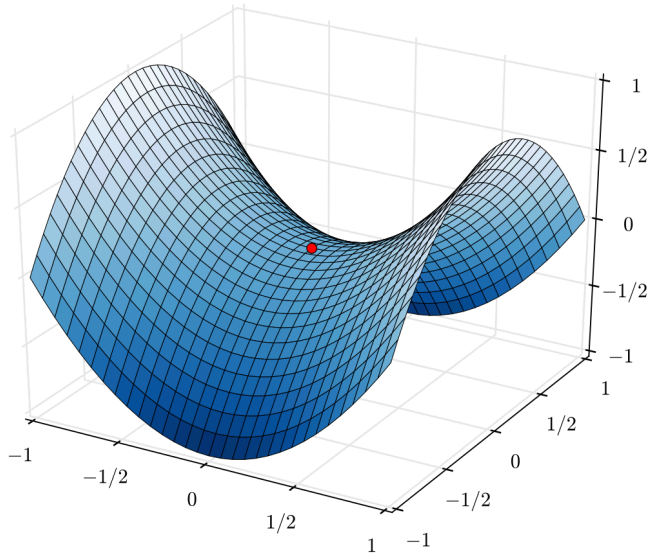


Fig. 2.4: A saddle point in a 2D cost surface. The red dot marks a stationary point where the gradient is zero, but the point is not a local minimum—along one axis the function curves upward, while along the other it curves downward. Such points can trap optimization algorithms if not properly handled.

As discussed above, a viable optimization algorithm may not always be able to find the global minimum in every situation. However, it should at least be capable of identifying a local minimum in the cost function, meaning it can provide an optimal local solution. The challenges mentioned indicate the need for an approach that does not directly compute solutions but instead explores the effects of parameter changes on the cost function. This approach should involve multiple calculations, handle multiple local minima, and address saddle points. The requirements suggest a methodology that involves exploring the cost function in multiple steps, using the effects of parameter changes to refine the parameter vector until a sufficient local minimum is found. This entails starting with an initial guess and incrementally refining it, gradually approaching an optimal or near-optimal solution. This is known as an **iterative (numerical) approach** to optimization, where the solution is gradually improved over multiple steps rather than computed in a single analytical pass.

2.2.2 - Gradient Descent

The optimal adjustment minimizes the cost function. However, as mentioned earlier, we must make small changes to account for input/weight interdependence and the complexity of functions. This involves making an initial guess, then iteratively moving in the direction that reduces the cost function by determining the optimal direction and magnitude for adjusting the parameter vector. If multiple iterations result in little to no adjustment, we have found a local minimum, and the iterations should cease. An example of this is pictorally represented below:

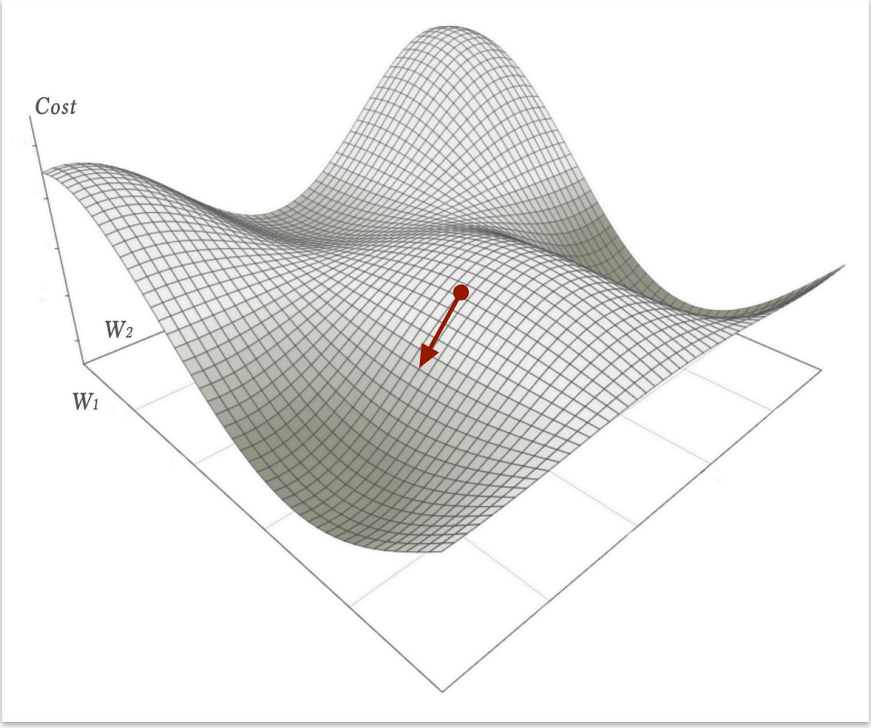


Fig. 2.5: Visualization of gradient descent on a 2D cost surface defined by weights W_1 and W_2. The red arrow shows the direction of steepest descent—indicating how the parameter vector is iteratively updated to reduce the cost. As the updates continue, the model converges toward a local minimum, where subsequent adjustments result in negligible change.

Every directional change to a parameter vector will have an effect on the cost function. However, we must figure out the change which moves the cost function down by the largest amount/in the most efficient way. This is known as steepest descent.

From multivariable calculus, we already know how to figure out the change in a given cost function due to a directional change in a specific parameter vector point. An instantaneous change in a specific parameter from a point will entail an instantaneous change in the output of the cost function. This is definitionally equivalent to the partial derivative of the cost function with respect to the specific parameter, evaluated at the original parameter vector point, multiplied by the instantaneous change in the parameter. This is described mathematically below for a function y = f(x):

$$\delta y = \frac{\delta y}{\delta x}\delta x \tag{2.4}$$

For a multivariable function, changes in each parameter entail a change in the cost function. The total change is equal to the sum of the changes due to a change in each parameter. This is described below for y = f(x) where x is an n-dimensional input vector:

$$\delta y = \sum_{i=1}^{n} \frac{\delta y}{\delta x_i}\delta x_i \tag{2.5}$$

However, we cannot move our parameter vector an infinitesimal amount. However, if we move a small amount, then we can approximate the changes as infinitesimal, and thus the function as linear around the original point. This means that for small changes, the partial derivative of y with respect to x[i] is approximately constant. This means that the change in cost function with respect to the change in each parameter is constant. This is visually represented and mathematically described below:

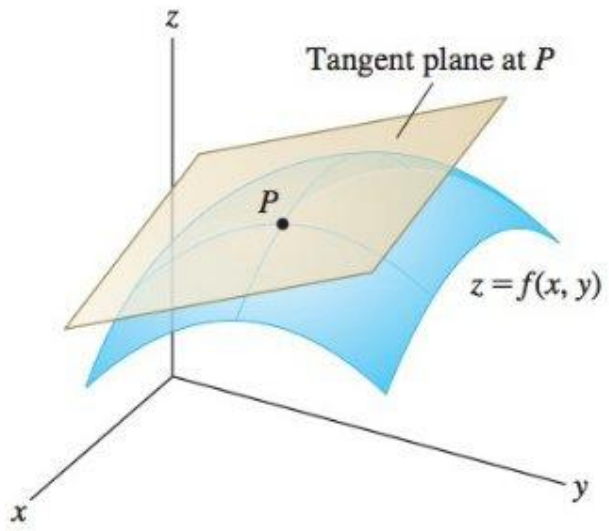


Fig. 2.6: Visualization of local linear approximation using a tangent plane. At point P, the nonlinear surface z = f(x,y) is locally approximated by a tangent plane. For small changes in x and y, the behavior of the function can be considered approximately linear, meaning the partial derivatives remain nearly constant. This approximation underlies gradient-based optimization methods for adjusting parameter vectors in machine learning.

$$\Delta y = \sum_{i=1}^{N} \frac{\delta y}{\delta x_i} \Delta x_i \tag{2.6}$$

This operation is known as the directional derivative. This can be equivalently written by taking the dot product of the change unit vector (v) and the vector of corresponding partial derivatives, known as the gradient vector. This is described below:

$$D_V f(x) = \nabla f(x) \bullet v \tag{2.7}$$

Since the optimal directional change entails the greatest decrease in output, it must entail the minimal (most negative) directional derivative. By the definition of the dot product:

$$D_v f(x) = ||\nabla f(x)|| \cdot ||v|| \cdot cos(\theta) \tag{2.8}$$

Where θ is the angle between the gradient and v. Since v is a unit vector, it has a magnitude of 1. So, the equation simplifies to:

$$D_v f(x) = ||\nabla f(x)|| \cdot cos(\theta) \tag{2.9}$$

Since the gradient, and thus the magnitude of the gradient, is constant for a given point, this reduces to minimizing cos(θ). As we know, the minimum value of cos(θ) is -1, which occurs when θ = π. This means that the unit vector points in the opposite direction of the gradient. This means that v points in the direction of the negative gradient. This is described mathematically below:

$$v = -\nabla f(x) \tag{2.10}$$

This means that we must adjust our current parameter vector by subtracting the gradient vector multiplied by the small step size (η). This is described below:

$$x_{t+1} = x_t - \eta \nabla f(x_t) \tag{2.11}$$

This algorithm is known as **gradient descent** (Cauchy, 1847)

One important note is that the gradient only holds true for that specific point. As we move, the gradient changes. The farther we move, the more the gradient changes from our original calculation. As the programmer/mathematician, you must determine how much the parameters change in each iteration of parameter updating. This is done by multiplying the gradient vector by a constant before subtracting it from the parameter vector. This constant (called the **learning rate** (η or sometimes α)), is chosen by the programmer before training the model. A larger learning rate means the model learns faster, but a smaller learning rate generally leads to better performance. However, since small learning rates can only move the parameters a tiny bit, the model often gets stuck in sub-optimal local minima. Generally, we want the model to converge quickly, but be able to properly fine-tune. So, to leverage the benefits of small and large learning rates while minimizing the downsides, we usually start with a larger learning rate to converge quicker and avoid small local minima, but slowly decrease our learning rate over the iterations to accurately fine-tune our model near the end of the training.

The learning rate is one of many pre-chosen variables, called **hyperparameters**, which dictate certain model attributes. These hyperparameters are generally tuned to maximize the performance of the model through multiple different randomizations and training. Generally, the programmer picks some possible sets of hyperparameters from experience and system knowledge, trains multiple models each with a different set of hyperparameters, and saves the model which performs the best on the testing data.

The weight updating process occurs many times during the model training. We stop training either after a set number of iterations over the dataframe (known as **epochs** (another hyperparameter)), or until the model performance barely changes when the parameters are updated (this minimum threshold is another hyperparameter)

To evaluate the gradient at a point, we must pass in a series of example inputs, generate model predicted outputs, then compare the predictions to the example ground truths. Doing this for one example is known as one forward pass. If there are multiple forward passes in between each iteration gradient descent, to ensure we have the most representative gradient, we use the mean of the partial derivative for each parameter with respect to the samples.

The **batch size** in machine learning refers to the number of forward passes/predictions performed before each optimization step during training. It is a crucial hyperparameter that significantly impacts the performance and efficiency of an optimization algorithm. **Batch Gradient Descent** processes the entire training dataset before performing a single optimization step, providing the most stable gradient estimates. However, this method can be computationally expensive and slow, particularly for large datasets. **Stochastic Gradient Descent (SGD)**, on the other hand, updates the model after each individual sample, making it more efficient and suitable for streaming data. However, the noise in gradient updates can lead to erratic convergence, causing the weights to oscillate around the error minimum rather than settling precisely. To balance stability and efficiency, **Mini-Batch Gradient Descent** is commonly used. In this approach, a subset of the training data (where the batch size is greater than one but smaller than the full dataset) is processed before backpropagation. Mini-batch gradient descent provides a compromise between stability and computational efficiency, making it the most widely used approach in deep learning applications. Batch size should be tuned to the desires of the developer.

2.3 - Neural Networks

2.3.1 - Piecewise Universal Function Approximation

From the above two sections, you now know how to find the optimal parameter set for any function given examples of the function as well as the functions format. However, we often do not know the format of the function. This presents a large issue because assuming an incorrect format means that one cannot capture the true function, and will likely be left with a function that has significant error. You could simply try a bunch of different functions and choose the best one, but this is very computationally intense. Also, many functions will be very complicated. They may involve multiple different algebraic and trigonometric operations on different variables. For example, consider the following equation:

$$y = 3x^2 + 2ln(y + 6) + sin(xy^{1/2}) \tag{2.12}$$

This equation has the following format:

$$y = w_1 x^{w_2} + w_3 ln(y + w_4) + sin(xy^{w_5}) \quad (2.13)$$

This means that you would essentially have to try every possible format to guarantee that you find the correct one. This is impossible. This shows that we must be able to have no prior information about the function format or parameter vector and figure out the optimal combination of function format and parameter vector, thus finding the true function.

However, to use gradient descent to update the parameters, you must have some format with an associated number of parameters. Then, through gradient descent, you find the optimal value for each parameter. This means we must have some format with an associated number of parameters capable of taking on any other format. Similarly to how the true parameter set minimizes error, the true format also minimizes error. This means that we can use gradient descent to find the true parameter vector and format. However, our format is set in gradient descent. We cannot simply switch between functions such as logarithmic and exponential. Also, these are separate formats. It is nonsensical to say that you iteratively get closer to exponential or logarithmic because these are discrete and separate concepts. This means that instead of taking on any possible format, we must have some format with an associated number of parameters capable of approximating functions of any other format. This means we can approximate and format a true parameter vector, meaning we can approximate any function. A function that can approximate any other function is called a **Universal Function Approximator (UFA)**. The goal now becomes to find or create this format.

In order to approximate a function globally, meaning for all potential inputs, you must have a function which always matches the true function irrespective of the inputs. To accomplish this, you simply utilize the true function format. However, we do not have this available to us.

Alternatively, you could use a single continuous function that can take on many other functions, such as a polynomial. However, these would struggle to adapt to regions with different behaviors (e.g., smooth areas vs. sharp transitions). The function may fit well in some regions, but not in others. To achieve the desired accuracy globally, the function would require a large number of parameters.

While high-degree polynomials offer great flexibility, this often leads the model to capture minor fluctuations in the training data—even if those reflect noise rather than true structure. These spurious patterns do not generalize to new data, resulting in poor predictive accuracy. Moreover, polynomial terms like x^d grow rapidly, so small changes in input or coefficients can cause large swings in output, leading to numerical instability. Another drawback is that polynomials are global: adjusting one coefficient affects the entire function, making it difficult to fine-tune behavior in just one region. Finally, because polynomials are smooth and

infinitely differentiable, they perform poorly on functions with sharp transitions, discontinuities, or step-like changes.

These issues show that we cannot have a single function which globally approximates the true function. We must generate a function that can approximate each region in the input space. However, to prevent overfitting, the function should not be able to perfectly fit the data in an individual region. We should not rely on high degree terms because they cause instability and other issues with optimization. We also should allow local changes without changing the rest of the function. These goals show the desire to fit individual regions with simpler functions, and allow optimization algorithms to make local adjustments to the function. Since functions generally abide by simpler relationships within small continuous regions, our goal is feasible as each region can be approximated by a simpler function with fewer parameters or expressive power. However, these simple functions would become highly inaccurate the larger the region that they must approximate because they would have to approximate more complex changes and curvature. To allow for global approximation, we must use multiple simple functions, each locally approximating a small region in the input space. This means the input space is partitioned into a series of non-overlapping regions, where each region has a single function mapping inputs to outputs. These partitions are points for 1D inputs, 2D shapes for 2 inputs, etc. This type of function is known as a **piecewise function**, or more specifically in this case a **piecewise universal function approximator** (Stone, 1948).

2.3.2 - Piecewise Linear Universal Function Approximation

To allow different functions in different sections of the input based on the data, the function format must allow for learned functions within each partitioned, continuous region. To standardize and simplify the representation and optimization of this function, the locally approximating functions should have the same format.

Additionally, the optimal partitioning of the input space is data-dependent. A preset partitioning prevents smaller partitions where more precision is needed, and prevents larger partitions where a single simple function can properly approximate a larger area. In many tasks, the input–output relationship changes across regions of input space, but we don't know where these changes occur. In reality, we do not know the optimal size or location of partitions as they rely on the underlying function. Additionally, the shape of the region over which a single continuous estimator can achieve PAC-learnable performance may vary across the input space. In one region, the optimal estimator may apply cleanly within a circular boundary; in another, it may hold over a hexagonal or irregular region. If all partitions are constrained to a fixed shape—such as axis-aligned rectangles—the model becomes inefficient: it must use more partitions to fit complex areas and may overfit simpler ones. Allowing partition shapes to adapt ensures that each region aligns with the structure required for effective learning. This shows that

we must not only learn the optimal function within each partitioned section of the input space, but also the size, shape, and location of the segments themselves. In other words, we should simultaneously learn the optimal partitions of the input space and the function mapping inputs to outputs for each partitioned segment of the input space. For the same reasons as with the approximators of the functions within a segment, the functions we use to partition the input space should have the same format as each other.

The simplest function we can use to represent a region of an input space is a constant value. As mentioned above, a continuous function is merely an infinite series of points that adhere to the input-output relationship. Each point represents a constant function within the output space, corresponding to an infinitesimally small location in the input space. Consequently, all functions can be approximated by an infinite number of constant functions defined over these infinitesimal intervals. For one-dimensional functions, this results in constant lines; for two-dimensional functions, constant planes; and for three-dimensional functions, constant hyperplanes, etc. However, we obviously cannot have an infinite number of constant functions. This means we must use a finite, but sufficiently large, number of intervals over which we assume the output to be the constant which optimally predicts the output value in that region. With a sufficient number segments, each segment will be approximately correct over its interval. This is known as a **piecewise constant** function.

An example of 1D piecewise constant approximation is shown below:

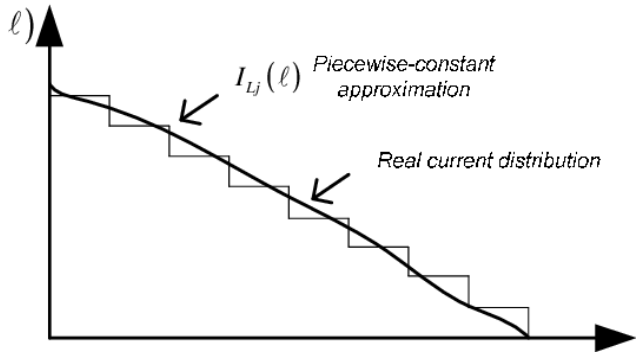


Fig. 2.7: A one-dimensional function (black curve) approximated by a piecewise-constant function (gray step segments). Each interval assumes a constant value that best predicts the output in that region. As the number of intervals increases, the approximation becomes more accurate, illustrating the foundational idea behind piecewise-constant function modeling.

Example of 2D piecewise constant functions are shown below:

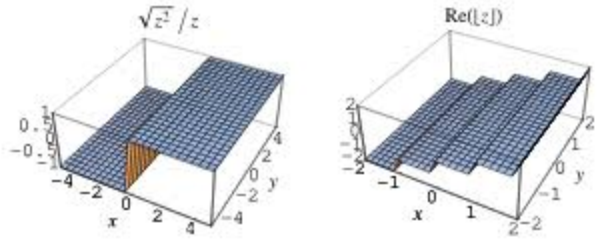


Fig. 2.8: Two visualizations of piecewise-constant approximations in two-dimensional input spaces. On the left, the planar surface is approximated by constant planar segments with one sharp discontinuity (2 segments). On the right, the surface is similarly approximated by horizontal planes with three sharp discontinuities (4 segments).

Fig. 2.8:

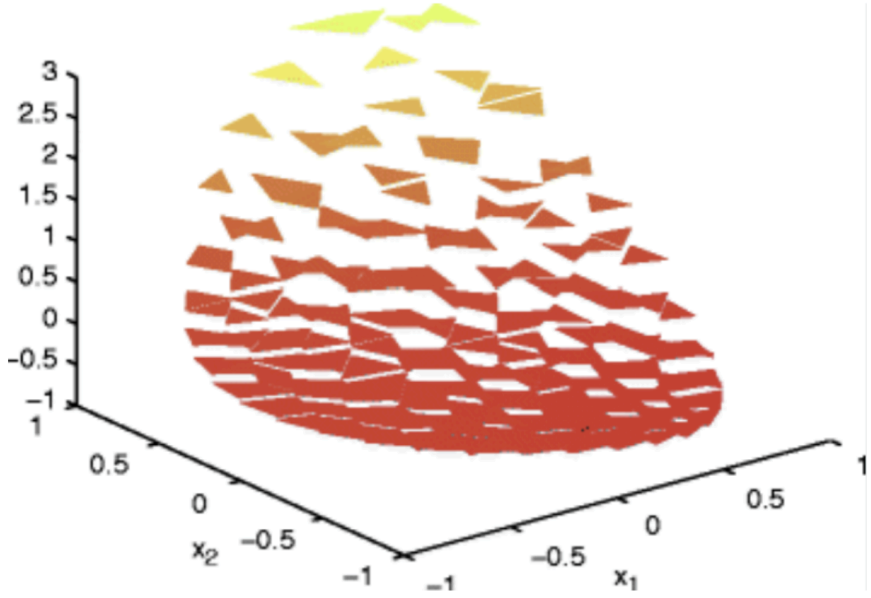


Fig. 2.9: Three-dimensional visualization of a function approximated using a piecewise-constant model over triangularly partitioned regions in the input space. Each colored tile represents a constant output value for a corresponding region defined by coordinates (x_1, x_2).

Note that for a function to be adequately approximated by a finite number of piecewise constants, the input space over which we approximate must be bounded on all sides. This does not present a huge issue as inputs almost always have explicit bounds or become so improbable at the extremes that we do not need to consider extreme values. In this case, we can simply extend our outermost constants to infinity and take caution if we get an input outside of the range of the training data. However, there are a couple problems with using piecewise constants.

However, there are several problems with using piecewise constant functions. First, they are not differentiable at the discontinuities, which makes them incompatible with gradient-based optimization. Even when we approximate these step-like behaviors using smooth functions like sigmoid or tanh, we encounter two issues: near the transition regions, small changes in input can

produce very large gradients, leading to unstable updates during training; and in the flat, saturated regions, small input changes produce very little gradient, causing the optimizer to stall. Both effects make training inefficient or unstable. Lastly, piecewise constants worked for modelling infinitesimally small points, but since we must deal instead with intervals, we must be instead concerned with the best approximator of the function within the interval. Piecewise constants quickly become inaccurate as even simple input-output relationships within a region cause the output to change. This means that a large number of piecewise intervals are required to accurately model functions. This leads to higher computational complexity and a larger probability of learning noise in the data.

This shows that we instead want a continuous function with the capacity to learn simple relationships within each region. We do not want to introduce unnecessary complexity that could be better used for adding more segments, even if each segment is valid for a smaller region of the input space. The function which meets these criteria and has the smallest number of parameters is a linear function. This means that we can approximate a bounded function by a finite series of linear functions. This is known as a **piecewise linear** approximation. Examples of this in 1D and 2D are shown below:

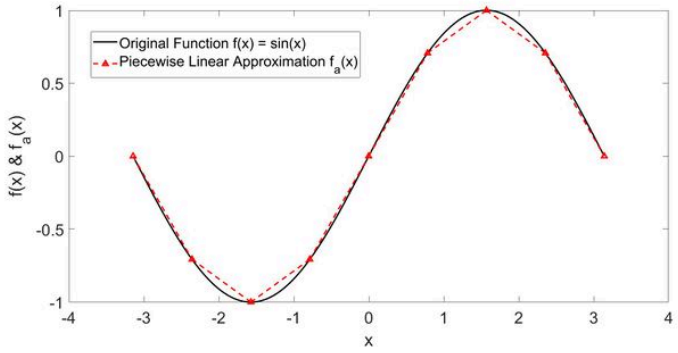


Fig. 2.10: One-dimensional piecewise linear approximation of the sine function. The smooth black curve represents the true function f(x) = sin(x)), while the red dashed line segments show a piecewise linear approximation $f_a(x)$. Each segment is a linear function defined over a finite interval, capturing the shape of the original function using a small number of parameters.

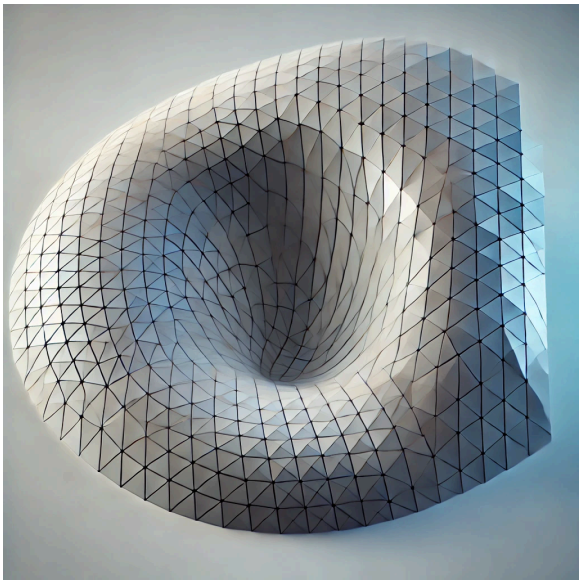


Fig. 2.11: High-resolution piecewise linear surface approximation using triangular mesh patches. Each facet approximates the local shape of the function using planar triangles, and the overall surface captures smooth curvature with increased fidelity as the number of segments grows.

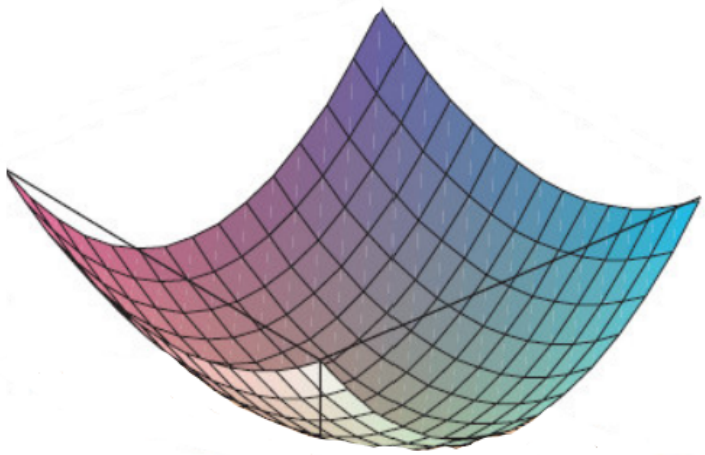


Fig. 2.12: Two-dimensional piecewise linear surface approximation. The surface is divided into a grid of triangles, with each triangle defining a linear plane over its domain.

Each piecewise linear unit entails a linear combination of inputs added to a constant. This means that the output as a function of the input vector x, parameter/weight vector w, and constant scalar (known as the bias) b, is as follows:

$$y = w \bullet x + b \tag{2.14}$$

Since adding a scalar is the same as multiplying a weight of b by an input of 1, to standardize optimization, the bias is generally seen as another weight with a corresponding input value of 1. This means we usually just add a 1 at the beginning of the input vector X. This means that:

$$y = w \bullet x \tag{2.15}$$

2.3.3 - Shallow Neural Networks

The compounding effect of an incorrect slope increases inaccuracy as the input deviates from where the segment is perfectly accurate. We want input space to be partitioned such that each n-dimensional hyperplane segment only maps inputs to outputs for a subset of the input space. The transition between the functions forms a boundary where one function becomes active and another becomes inactive.

We want a continuous input space to be mapped to an individual segment/function, with the segment/function changing upon crossing a boundary. To allow for these boundaries to effectively partition the input space, they must be a function of the inputs. These boundaries should be continuous and smooth, compatible with hyperplane segments, able to express non-linear functions, and simple. For similar reasons as our choice of segment function, we generally choose for each boundary to be linear. We could use other functions for forming the boundaries such as x^3 or curve types, but these would add unnecessary computational intensity that would be better suited for more segments or partition lines.

This means that the change in behavior occurs when the input crosses some hyperplane, making some previously active boundary inactive or vice-versa. For an n-dimensional input vector, each segment is an n-dimensional hyperplane, so each boundary hyperplane is definitionally an (n-1)-dimensional hyperplane. Each hyperplane divides the input space into two half-spaces, creating a boundary for one or more segments.

The equation for an n-1 dimensional hyperplane is represented below:

$$w \bullet x = c \tag{2.16}$$

Since one of our inputs is one to account for adding a constant, we essentially take a linear combination of weights, add a bias, and set it equal to the constant to get the hyperplane. However, adding a constant and then dealing with the constant c is redundant because it adds an unnecessary additional parameter to optimize. The equation above is the same as if we subtracted c from out bias and set w.x equal to 0. Since these two are the same, altering the equation

accordingly does not reduce any expressive power of these hyperplanes. This brings us to the following equation:

$$w \bullet x = 0 \tag{2.17}$$

More specifically in the ‘active’/‘inactive’ paradigm:

$$w \bullet x \leq 0 \rightarrow \text{Inactive} \tag{2.18}$$

$$w \bullet x > 0 \rightarrow \text{Active} \tag{2.19}$$

To ensure that each segment only contributes to the final function when the input is within a specific region, each segment must be bounded on all sides unless it extends out to infinity in some direction. Since the boundaries of each segment are linear, to bound each segment on all sides, each segment’s boundaries must be defined by multiple intersecting hyperplanes. To allow for maximum expressive power in the boundaries, each boundary should be free to change its slope and bias as necessary. These hyperplane equations should not be ‘tied’ to each other, in that certain boundaries must be parallel to each other, perpendicular to each other, etc. This means that we will have a series of unique and free (n-1)-dimensional hyperplane equations partitioning the input space, where each unique combination of active boundaries creates a single continuous linear region.

To properly define a single optimal segment, we would have a series of intersecting hyperplanes that collectively surround the segment. For the purposes of defining a single segment, each hyperplane would only be relevant within specific bounds. For example, with 2D inputs, each line segment forming part of a hexagonal partition would be bounded at both ends, so that it does not extend beyond the hexagon’s edges. If every segment were hexagonal, the partitioning may look something like this:

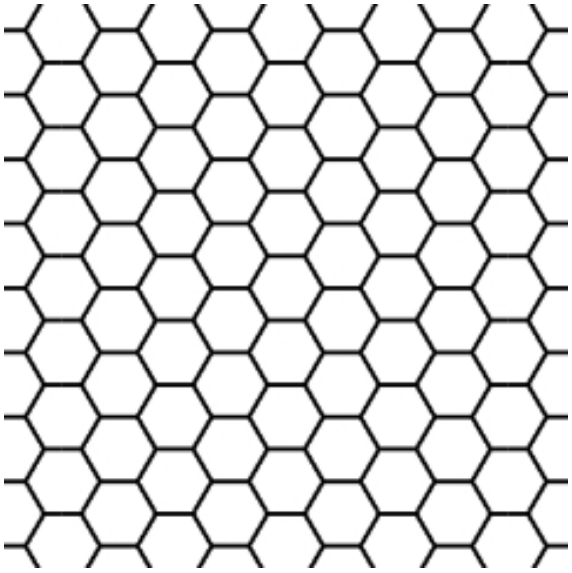


Fig. 2.13: Hexagonal tiling of a two-dimensional input space, illustrating a possible piecewise partitioning approach. Each hexagonal region represents a distinct segment bounded by linear edges, where a locally optimal function could be defined.

To perform this bounding on each line, such as $y = 2x$, we must set two boundaries, such as $0 \leq x$, $x \leq 5$. Past these bounds, another hyperplane boundary takes over the partitioning, and the original hyperplane is no longer relevant for defining that segment.

However, there are some issues with using finite linear boundaries.

- Defining the bounds for these hyperparameters requires many additional parameters to specify and learn.
- Also, a single, finite line segment does not define a complete boundary in the input space on its own because it does not divide the space into separate regions—it only marks a limited edge without enclosing or separating any volume. We cannot simply check whether or not $w.x + b > 0$. A point might be on the "correct side" of the line, but far outside the region the segment actually spans. Having bounds on a partitioning hyperplane means that we would have to define additional mathematical equations to determine whether a boundary is active or inactive for a given point.
- Additionally, the same hyperplane can help separate meaningful patterns in multiple locations of the input space. This reusability is highly efficient: if we used separate, bounded hyperplanes for each region, we would need to learn and store multiple nearly identical equations—wasting capacity. Even if a hyperplane is not helpful in certain areas (e.g., it cuts through a region where the same output applies on both sides), the network can still learn the same function on both sides, rendering the boundary neutral in that region. It does no harm. Crucially, leaving a hyperplane intact across the entire space increases the number of distinct linear regions the network can form. If the hyperplane

intersects other regions, it can combine with other hyperplanes to define new functional partitions. This means a single hyperplane can help shape multiple useful regions, effectively increasing the expressive power per parameter and making the model more powerful without adding neurons.

- Lastly, we want boundaries to extend to unseen inputs to promote generalization. If boundaries were localized or bounded, they wouldn't apply to new input regions during test time. But unbounded hyperplanes generalize their decision logic throughout the input space. For these reasons, we do not bound these hyperplane boundaries, and allow them to extend infinitely. A pictorial representation of this is shown below:

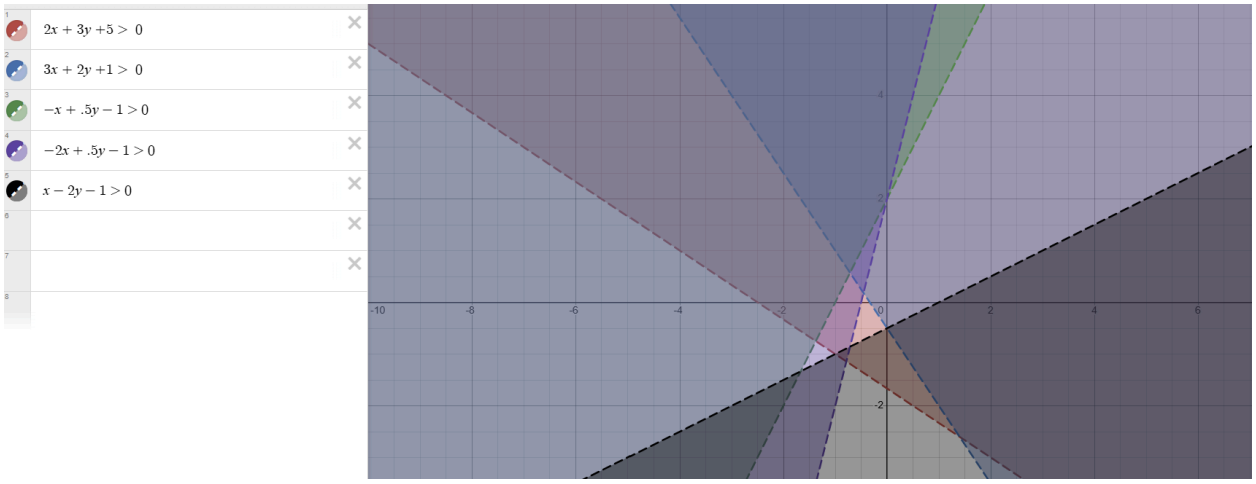


Fig. 2.14: A visualization of multiple unbounded hyperplanes (shown here as 2D linear inequalities) dividing the input space. Each colored region represents the area satisfying a particular inequality. The dashed lines represent the boundaries of these regions—each one extends infinitely, without being clipped to local regions. This illustrates how allowing hyperplanes to generalize across the entire input space enables the model to apply consistent decision boundaries to unseen data during test time.

This means we define a set of linear hyperplane boundaries and a set of linear segment functions, then use gradient descent for optimal input space partitioning and function approximation for each segment. However, there are a few issues with this approach.

If each segment is treated as a completely independent function, there is no connection or shared knowledge between segments. This independence means that training one segment does not affect or improve the behavior of neighboring segments. This hinders generalization, requiring more data for each segment to ensure it learns the true pattern in that region.

Real-world functions often exhibit smoothness and continuity, where derivatives change gradually rather than abruptly. This suggests that relationships between inputs and outputs are not entirely independent across regions. Nearby segments are likely to share similar functional behavior or trends, even if they are defined within distinct boundaries. For example, temperature variation across a geographic region is typically smooth, with gradual changes rather than sharp jumps.

By leveraging the assumption of smoothness, knowledge learned in one segment can improve the accuracy and generalization of neighboring segments. This influence should diminish as the distance from the segment increases. Intuitively, the functional relationship in one region has more relevance to its immediate neighbors than to distant regions. This gradual decay of influence aligns with the principle that local approximations work best in nearby regions and less so in faraway ones.

For data from one segment to influence the updates to a function in another segment, the function in this other segment must be influenced by the function mapping the data to its corresponding output in the original section. This means that the function in this other segment must directly contain all or part of whatever function is utilized in the original function. To allow for this data to be used to train other segments, these other segments must include all or part of the original segment's function in their function. For this original segment to also utilize data from other segments, it must include all or part of the other segment's functions in its function. This means that each segment will include all or part of potentially many other segment's functions in its own function, and part or all of its function will be included in the functions of potentially many other segments.

Each segment's function is linear and the sum of linear functions is a linear function. This means that a simple way for a segment to incorporate the functions of other segments into its own function is to add these other functions to a linear function that the segment itself adds. To facilitate the idea that individual segment functions should have more in common with the functions of their neighbors, each segment function is really just its neighbors segment function with an adjusted slope, and also the idea that functions change at boundaries, each boundary should add a function when activated. This means that for some segment, the segment changes at each boundary based on the activation or inactivation of some linear function. As discussed above, to reduce the complexity of boundaries, avoid gaps in the decision surfaces, minimize the number of boundaries required, and maximize the number of segments created from a finite set of boundaries, we extend these boundaries infinitely. This means that we have a series of linear boundaries, each with their own linear function of the input that they add when they are active. This means that each section's function is fully defined by summing a unique combination of active linear functions.

Another reason for this relationship between boundary functions and segment functions is the necessity to update boundaries via gradient descent. If boundary activity is discrete, then there is no continuous way to update them. Additionally, to update a boundary based on inputs and predictions, the predictions must be continuously tied to the boundaries. This means that the functions mapping inputs to outputs must be a continuous function of each active boundary. This means that each boundary should impart a continuous function that is active when the boundary is active, meaning that each unique combination of boundary contributions combines together to

form a segment. For reasons mentioned above, these functions should be linear, meaning that each boundary adds a linear function. This means that each section's function is fully defined by summing a unique combination of active linear functions.

To ensure that the function is continuous across boundaries to prevent exploding gradients, each boundary linear addition should start at 0 when activated. This means that each boundary is defined by a linear function it adds, once that linear function exceeds 0. This contribution for an individual boundary is given as below:

$$max(0,\ w \bullet x) \tag{2.20}$$

This function which takes the maximum between 0 and a variable is known as the **Rectified Linear Unit (ReLU)** function. Since this determines the activity of the function, it is known as our **activation function**.

A 1D for x^2 is shown below:

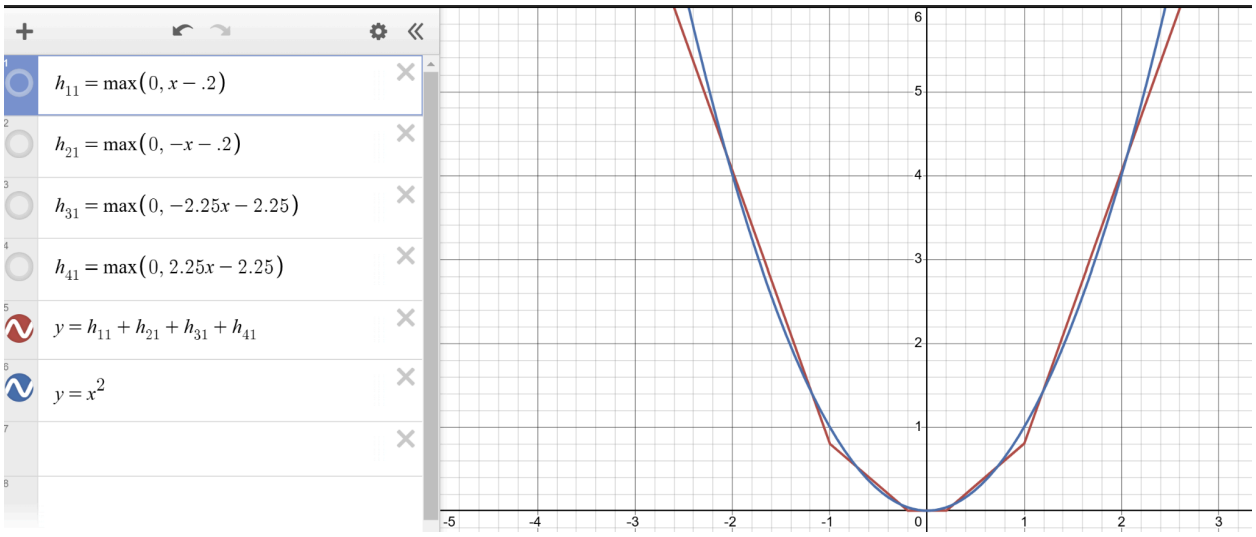


Fig. 2.15: A one-dimensional example illustrating how a quadratic function $y = x^2$ (blue curve) can be approximated using a combination of ReLU functions (red curve). Each ReLU function introduces a linear segment that activates over part of the input space. By summing several of these piecewise linear ReLU components, we approximate the curved shape of the quadratic.

However, ReLU outputs are always non-negative, contributing only positive values to the next segment. This presents an issue because many functions have negative outputs for some input. To allow for the output of a ReLU term to be negative, we must be able to turn certain ReLU outputs negative before the summation.

Additionally, consider the behavior of the function f(x) as you increase a specific input variable x_j. The partial derivative of f(x) with respect to x_j depends on the behavior of the individual ReLU terms. Within a segment (no boundary crossing), For any given input, each

ReLU term is either active or inactive. While x_j increases within the current segment (without crossing a boundary where $w_i x + b_i = 0$), the partial derivative of the active ReLU terms with respect to x_j remains constant. The inactive terms contribute nothing to the gradient. Thus, within a segment, the partial derivative of f(x) with respect to x_j remains constant.

When x_j increases to the point where $w_i x+b_i = 0$, a boundary is crossed. This transition changes the active/inactive status of the corresponding ReLU term. In this situation, there are two possible cases. The first case is that an active ReLU term becomes inactive. This happens if the boundary is crossed from $w_i x + b_i > 0$ to $w_i x + b_i \leq 0$. The term $\text{ReLU}(w_i x + b_i)$ contributes w_{ij} additively to the partial derivative of the function at large with respect to x_j while it is active. When the boundary $w_i x + b_i = 0$ is crossed, this contribution drops to zero. In this case, w_{ij} must be negative in order to reduce $w_i x + b_i = 0$, meaning that the ReLU term becoming inactive increases the partial derivative of the function with respect to x_j.

The other case occurs when an inactive ReLU term becomes active. This occurs if the boundary is crossed from $w_i x + b_i \leq 0$ to $w_i x + b_i > 0$. This occurs when the increasing value of x_j causes $w_i x + b_i$ to cross zero into positive territory. The term now contributes positively to the gradient, increasing the overall partial derivative of f(x) with respect to x_j. The partial derivative with respect to x_j increases by the positive w_{ij}, the weight of the term along x_j.

In any of these situations, these transitions never cause a decrease in the gradient. The constant or increasing nature of the gradient across all transitions ensures the overall function is convex. In other words, the function never bends “downward”, and is “bowl shaped”. Convex functions are characterized by the property that for any two points on the function, the line segment connecting them lies above the function. This property has biconditional equivalence to a non-decreasing slope in any direction. This means that a sum of ReLU functions can only model convex functions.

However, we want to be able to model non-convex functions. To model non-convex functions, we need to allow negative slopes across ReLU activations. More technically, we want the transition of a segment from inactive to active as a variable increases to potentially add a ReLU term that has a negative weight on that input. This means that as a ReLU term becomes positive and is increasing, the overall function is decreasing. Additionally, we want the transition of a segment from active to inactive as a variable increases to potentially remove a ReLU term with a positive weight on that variable. This means that a ReLU term decreases toward 0, causing an increase towards 0 in the overall function. Both of these situations point to taking the negative of a ReLU output.

The primary way to accomplish this in a way that allows for altering the slope and the bias accordingly is to multiply the output by a negative number. To allow each segment to take on a positive or negative value in a continuously changing manner, we multiply each ReLU

function output by a unique numerical value before the summation. This is mathematically described below where w_{io} is the weight on function i in calculating the output:

$$f(x) = \sum_{i=1}^{n} w_{io} * ReLU(w_i \bullet x) \tag{2.21}$$

The four possible types of segments are shown below in 1D:

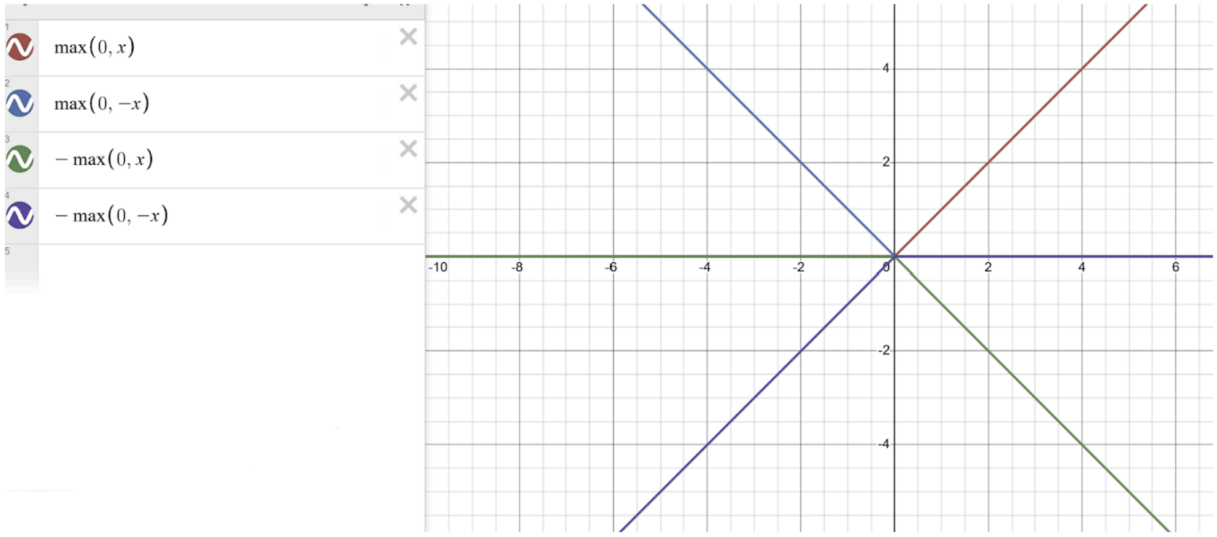


Fig. 2.16: A visualization of the four possible segment types resulting from weighted ReLU functions in a one-dimensional input space. Each function is a variation of the basic ReLU form, scaled or reflected by a negative sign. These four basis segment shapes form the foundation for approximating arbitrary functions.

This means that we initialize our model by choosing some number of boundaries and initializing the weight vectors accordingly. We define our model as a weighted sum of ReLU of each weight vector taking the dot product with the input. We then perform gradient descent on the vector of all weights, simultaneously adjusting the ReLU scaling, as well as the boundaries and segment functions, with the segments closer to the data point having more boundaries in common with the segment that the data point resided in, thus being affected more. Eventually, we end up with the piecewise linear approximations shown above. This is the simplest fully implementable piecewise universal function approximator, specifically a linear one in this case. In fact, this type of network has been proven to be a UFA for a given PAC requirement. This proved conclusion is known as the UFA theorem (Cybenko, 1989).

Since this function format has some resemblances to the firing of neurons in the human mind, nodes are sometimes referred to as neurons. Hence, this format is known as a **Neural Network (NN)** (McCulloch & Pitts, 1943). This type of network is also known as **Multi-Layer Perceptron (MLP)**. A perceptron is merely a function that takes a linear combination of inputs,

passes it through an activation function, and outputs the activation function output. An NN example is mathematically shown below in 1D for sin(x):

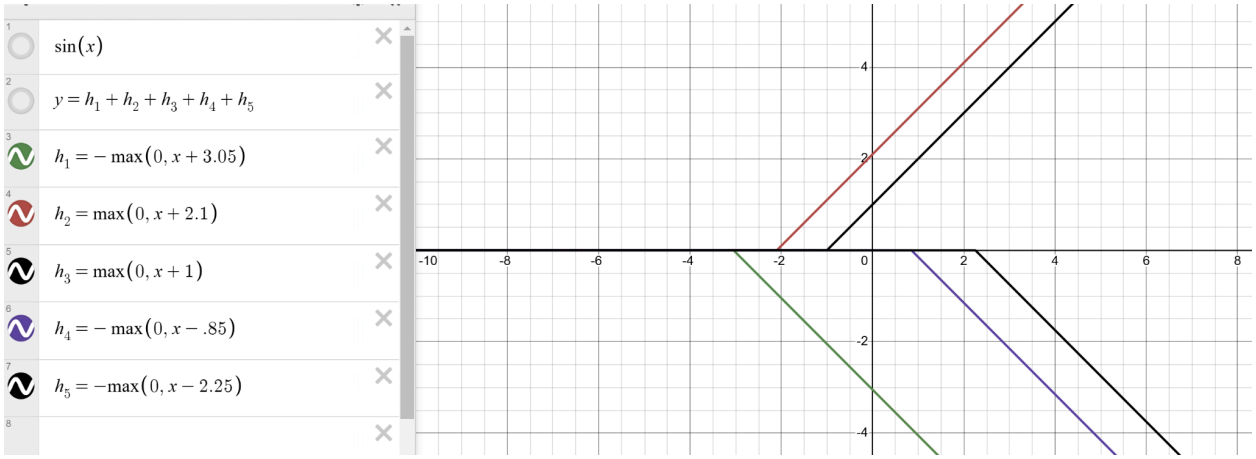


Fig. 2.17: Approximation of the function sin(x) using a 1D neural network composed of five ReLU-based neurons.

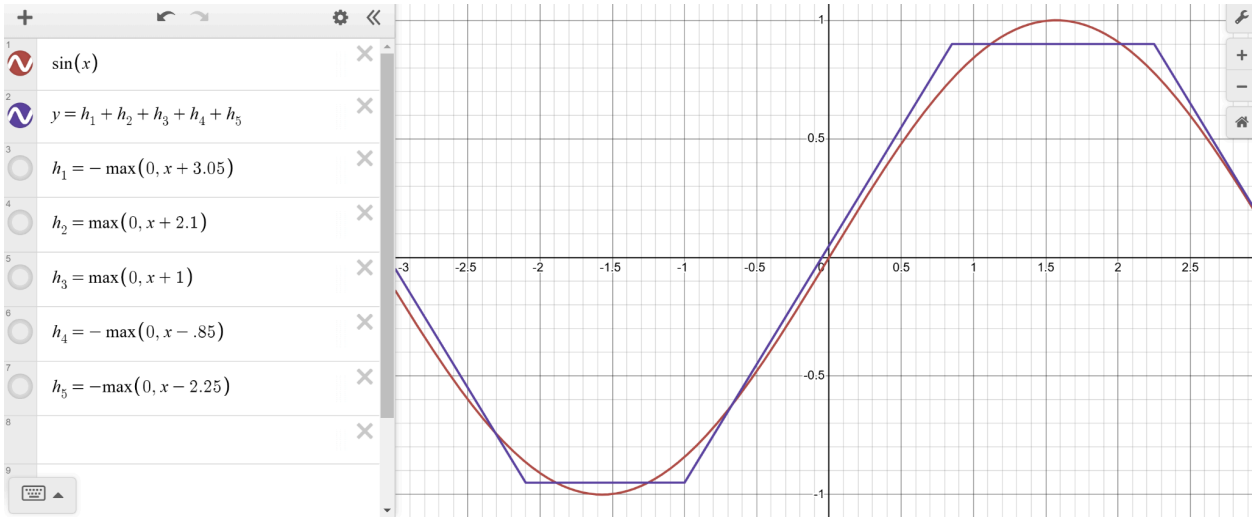


Fig. 2.18: An overlay of the ReLU approximation in blue and sin(x) in red

Note that this type of network can be used in conjunction with activation functions that more closely follow the piecewise constant paradigm, such as sigmoid or tanh. Sigmoid transforms its inputs such that the output can only be between 0 & 1, and tanh does the same but for -1 & 1.

As a communicative and pedagogical tool to convey ideas about this type of model, we often represent this function format as an interconnected network of outputs and weights connecting outputs. This is often pictorially represented as a directed computational graph where the outputs are represented by nodes and weights connecting an input to an output are represented as edges.

In this situation, each set of inputs and outputs is known as a **layer**. We have the **input layer** containing the inputs which is used to generate the middle layer output. From this, we can directly compute the **output layer** containing the output via a linear combination. In this situation, since we only see the inputs and outputs to a function, the layer in between is known as a **hidden layer**.

This is illustrated below:

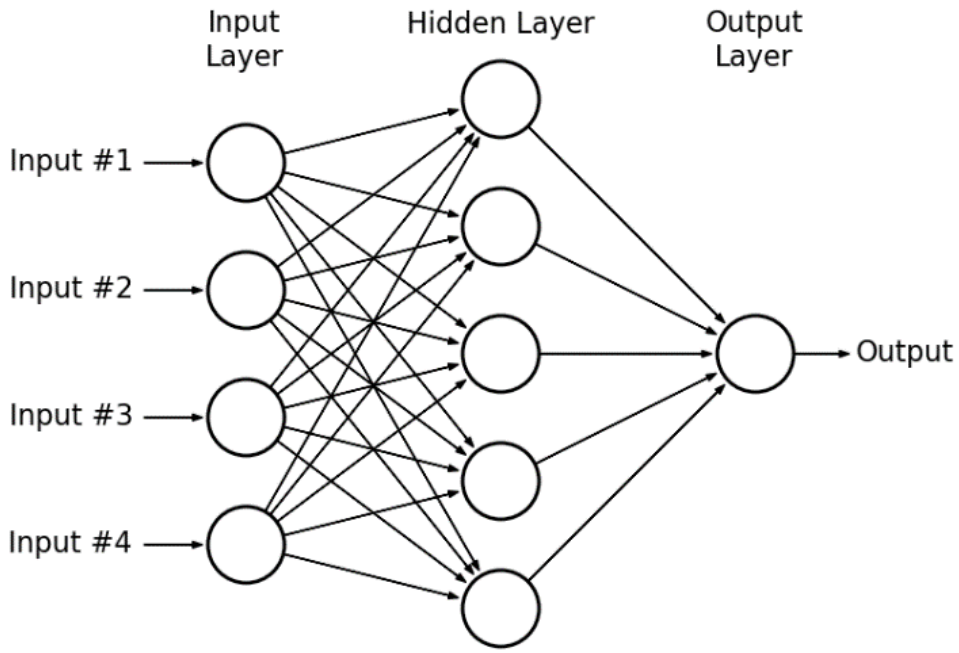


Fig. 2.19: A neural network with an input layer, hidden layer, and output layer.

The specific type of NN with only one hidden layer is known as a shallow neural network (SNN). Note that some definitions of SSN include 0 hidden layers, but we will not be using SNN in this way.

As a developer, you must tune the number of neurons in the hidden layer to enable modeling of complex functions. A limited number of hidden neurons means that the network can only form a small number of decision boundaries in the input space. If these regions are too few, the network cannot sufficiently capture the complexity of the decision boundaries needed for the task, especially when the data is non-linear. Additionally, too few neurons will lead to an insufficient number of segments, leading to individual segments being too large. This means that many segments will likely have to be too large leading to significant portions of those segments where the segment is not PAC. This means that an insufficient number of hidden layer neurons leads to **underfitting**.

Too many neurons provide the network with the ability to create a large number of segments. This means that the network may learn a segment that is very small and reflects noise

or outliers in the training data. For example, a large segment may be a PAC estimator of the true function, but if broken up, the model could learn the outliers, noise patterns, or tiny fluctuations within the data that do not reflect the true function, and thus will lead to lower testing performance. If you take an existing PAC network and add many more boundaries, the network has the potential to configure each boundary function such that it largely resembles the original function, but some sections are broken up to reflect intra-segment patterns. At some point, this leads to the learning of erroneous patterns. This is an example of **overfitting**. A graph of a function of $-x^2 + b$ with some noise that a model with too many segments could overfit to is shown below:

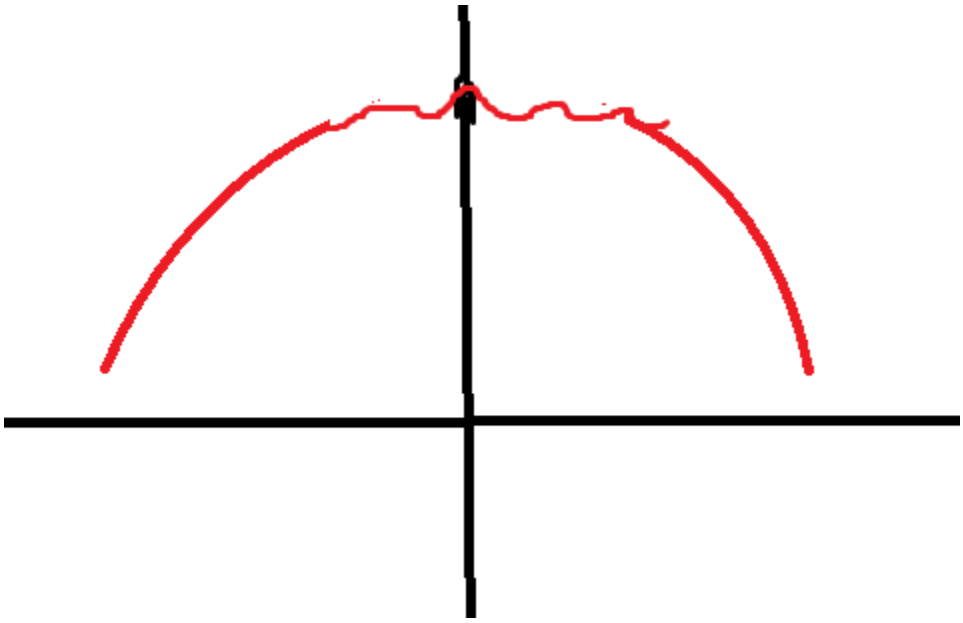


Fig. 2.1: Illustration of overfitting in a regression task. The true underlying function is a smooth downward-opening parabola (e.g., $-x^2+b$), but the red curve represents a highly complex model that conforms to every fluctuation and noise point in the training data. This results in a jagged approximation that fails to generalize to unseen data, especially in regions where data is sparse—demonstrating the classic behavior of an overfit model.

2.3.4 - Deep Neural Networks

However, this approach has several limitations. Directly approximating the full function with linear segments becomes problematic when the function has regions with large 2nd derivatives that are primarily positive or primarily negative for significant portions of these regions. These situations lead to first derivative approximations of a region to become not PAC very quickly in one or more directions. In such cases, the input range for which each linear segment is PAC becomes very small. This necessitates a larger number of segments, each covering a smaller range. This will lead to large computational intensity and increase the risk of overfitting for the reasons described above.

As discussed earlier, almost all functions can be broken down into simple operations. Extensive function composition (or nesting) can amplify the rate of change in the inputs to subsequent functions. This occurs because intermediate functions can transform the inputs in ways that magnify their variation, leading to outputs with larger first and second derivatives. As a result, the second derivative of the final composed function can become greater in magnitude compared to the scenario where the same function is applied directly to the original inputs. As a result, a segment that would have been PAC for a certain input region of a simple function now becomes valid for a much smaller region of the composed function's input. Consequently, the linear segments approximating the final output become valid over increasingly smaller input regions. This issue is exacerbated as additional layers of composition are added and as the input space grows, leading to exponential growth in the complexity of the approximation.

For example, consider $\sin(x^2)$, x^2, and $\sin(x)$ below:

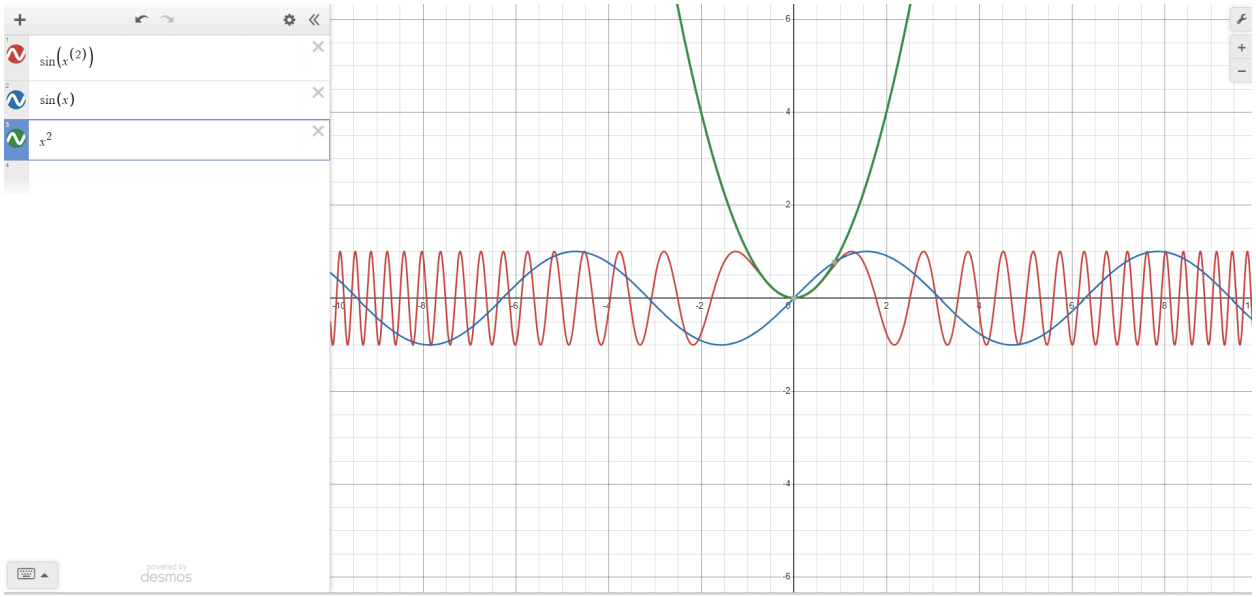


Fig. 2.20: Comparison of $\sin(x^2)$ (red), $\sin(x)$ (blue), and x^2 (green), illustrating how function composition can amplify input variation and increase complexity.

To approximate functions, or other high-dimensional, highly nonlinear functions, our final model must create small linear partitions of the input space. This presents two main issues. First off, learning this many parameters is very computationally intensive. Additionally, each unique summation of active linear functions would be so small that it would likely overfit. This shows that an individual weighted sum of ReLUs, i.e. SNNs, must not have an exceedingly large number of boundaries. This also means that the partitioning of inputs to the overall function must often be finer than the partitioning of the inputs to an individual weighted sum of ReLUs. These statements are equivalent to saying that each unique segment in a weighted sum of ReLUs should not be too small. The desire to avoid computational intensity also means that we must

provide a way for some neurons to provide more functional modeling capabilities than adding one linear segment and transition.

A segment adds a boundary and functional change to the rest of the SNN because of its transition when its defining function equals 0. For a segment to add multiple boundaries to the original function, the function must have multiple lines or regions in the input space where it transitions above and below zero. The overall sum of ReLUs function would not necessarily equal 0, but merely the function of an individual piecewise hyperplane. This allows for multiple disconnected regions of the input space where the segment is active. Specifically, this allows for points in the input space where the segment is active to be connected through a line which passes through a region of inactivity.

If the segment is purely linear with respect to its inputs, this is impossible, as a linear function can only have a single boundary in its input space. To achieve multiple boundaries, the function must introduce nonlinearity in its decision boundaries. More specifically, the linear segment must not be monotonic, meaning that, for each direction, it is always increasing or decreasing. As an example, $x^2 - 4$ crosses 0 at two places, meaning that $\max(0, x^2 - 4)$ would have two transitions. This number of boundaries increases significantly as we add more dimensions and more curves.

As mentioned above, to allow for minimal computational intensity, and an ability to take on any other function at different regions, segments should be linear with respect to their inputs. For such a function to have nonlinear boundaries, its inputs must be nonlinear. Another reason for this is that to prevent overfitting, this linear segment should not be able to alter the existing nonlinearity in the segment. This means that functions which are non-linear with respect to the overall function inputs are passed through these segment functions. This entails a compositional function.

For the first layer of functions to take on any nonlinear function, with differing functions as different points in space, these should be UFAs. This means that these are learned in accordance with the SNN format. For our second layer of functions to be able to alter the slopes of segments from the first function and add new ones in any given manner, with different alterations at different points of the input space, these functions must be UFAs learned in accordance with the SNN format. In this situation, each SNN would have a boundary at places where its inputs had boundaries, or where it introduced new boundaries. The SNN can alter the weights and biases on each segment to both influence its new boundaries, and also learn the added function when its segment is active.

This second layer now has more expressive power without a significant increase in risk of overfitting. To allow for a function to learn even more segments, with even finer nonlinear boundaries, we may want our UFA to have multiple layers of composition. In this sense, each layer of composition is essentially a series of UFAs that learns such that it can make the optimal

decision boundaries for the subsequent layer of UFAs. An equivalent way of viewing this is that each layer learns the optimal function for the subsequent layer to compose.

However, complex data often involves multiple relationships or correlations that are not easily captured by a single function. Each layer of SNNs should generally include multiple SSNs such that many different nonlinear boundaries and functions can be leveraged by the subsequent layer. Since the final function may be a function of multiple different functions each with multiple variables, we want the algorithm to be able to learn multiple different functions of each input, then learn multiple different functions of these values, etc. until obtaining the output.

This means we want to have multiple different SNNs acting as input to multiple different SNNs, etc. with each SNN in a layer representing a different function in that layer, and each layer representing another layer of composition. To do this, we could have each layer learn many different SNNs, but this is very computationally intensive. This is especially true for functions with many layers, where each function would essentially necessitate many separate SNNs in prior layers, each with their own separate prior SNNs, etc.

However, the boundaries learned by one SNN can often be used to represent many functions. The same goes for linear segment functions. If multiple different SNNs each use similar segment functions, then these can be reused. In fact, if two functions use similar boundary functions, then they will also use some similar segment functions.

This reuse of boundaries and functions is especially true if the number boundaries of the SNN are increased to accommodate representing multiple functions, and if these boundaries are learned such that they maximally accommodate other functions. In addition, a large number of segments and boundaries could accommodate segments and boundaries that are used by one or a subsection of functions. Essentially, if multiple similar boundaries are learned by multiple different functions, then this is inefficient, as this boundary could simply be reused for each function that utilizes it.

Additionally, a SNN already requires that each segment is multiplied by a constant before adding to form the final function. Different weighings of these segments create entirely different UFAs with different slope directions and magnitudes.

This means that we can simply have one large SNN which we use the post-ReLU values of to generate nonlinear functions. For each function, we take a different linear combination of the post-ReLU values of the layer. This means that we essentially have a single larger SNN with multiple outputs, learning its weights to optimally partition the input space and generate functions for the subsequent layer functions to compose and utilize for nonlinear boundaries.

Each level of UFA composition can reuse the boundaries and segments created by the previous level, with each function in the current layer composing earlier outputs in distinct ways. As a result, a single function in the current layer can introduce many new transition points—and therefore new boundaries and segments—that differ both from those in the previous layer and

from those generated by other functions in the same layer. Consequently, a weighted sum of functions at the current level can produce significantly more complex structure in the input space.

For subsequent functions, the weighted averages of the prior function's post-ReLU values represent the function outputs. These outputs serve as inputs for the initial weighted sum of the subsequent function, which generates its pre-ReLU segment values. In other words, we are taking a series of linear combinations of the post-ReLU values, and then taking further linear combinations of these values to generate the subsequent function's pre-ReLU values.

However, it's important to note that a linear combination of a linear combination of values is still just a linear combination of those values. To elaborate, if we multiply a linear combination by a constant, the constants in the linear combination simply change, but the structure remains the same. Similarly, adding a value to a linear combination alters the value added to the sum of products, but the format remains unchanged.

This insight suggests that instead of learning two separate linear combinations to combine the segments of one function and then apply them to the next function, we can directly learn the composition of the functions. In practice, this means that the post-ReLU values, which act as the initial segment outputs for a function, will be passed through linear functions, and then through ReLU, generating the initial segment outputs for the subsequent function.

To illustrate this, consider two linear functions, each acting on a single variable:

$$f(x) = c_1 x + c_2 \tag{2.22}$$

$$g(x) = c_3 x + c_4 \tag{2.23}$$

Now, if we compose these two functions:

$$f(g(x)) = c_1(c_3 x + c_4) = c_1 c_3 x + c_1 c_4 = c_5 x + c_6 \tag{2.24}$$

We observe that the composition of the two linear functions results in a single linear function:

$$f(g(x)) = c_5 x + c_6 \tag{2.25}$$

This demonstrates that two linear combinations are redundant and can be represented as a single linear function. Hence, instead of learning two separate linear combinations for combining

the segments of one function and then applying them to the next function, we can equivalently learn c_5 and c_6 directly through gradient descent.

This reality shows that we can add a set of functions at a new level/depth of composition by adding a single hidden layer linearly combining the values of the nodes from the previous layer and outputting the initial post-ReLU segment outputs. In this final function format, each hidden layer adds an additional function composition, with the post-ReLU node values of each layer representing the values of the different initial segments of the function acting on the output values from the previous functions.

In this final function format, each hidden layer represents an additional level of function composition. Additionally, each additional neuron in a hidden layer represents an additional segment for the functions of that layer.

For n layers, we have n-1 layer connections. This means that we will have n-1 weight matrices. The result is a weight array (W) where W^i contains the matrix connecting layer i-1 to layer i. This array of matrices is a type of algebraic object called a **tensor**. Since the structure of the network is decided ahead of time, so will the full size of the tensor. This tensor containing each individual weight is the entirety of the network. In fact, when saving and transmitting these algorithms, we often merely save and transmit this tensor.

This means that the forward propagation of inputs (**forward pass**), where a^i is the vector of values at layer i, can be written as:

$$y' = a^n = \text{ReLU}(W^n a^{n-1}) = \text{ReLU}(W^n \text{ReLU}(W^{n-1} \text{ReLU}(\dots W^2 \text{ReLU}(W^1 a^0) \dots))) \qquad (2.26)$$

As with the SNN, we can represent this function in a directed graph with nodes as input/outputs and edges as weights. This is represented pictorially below:

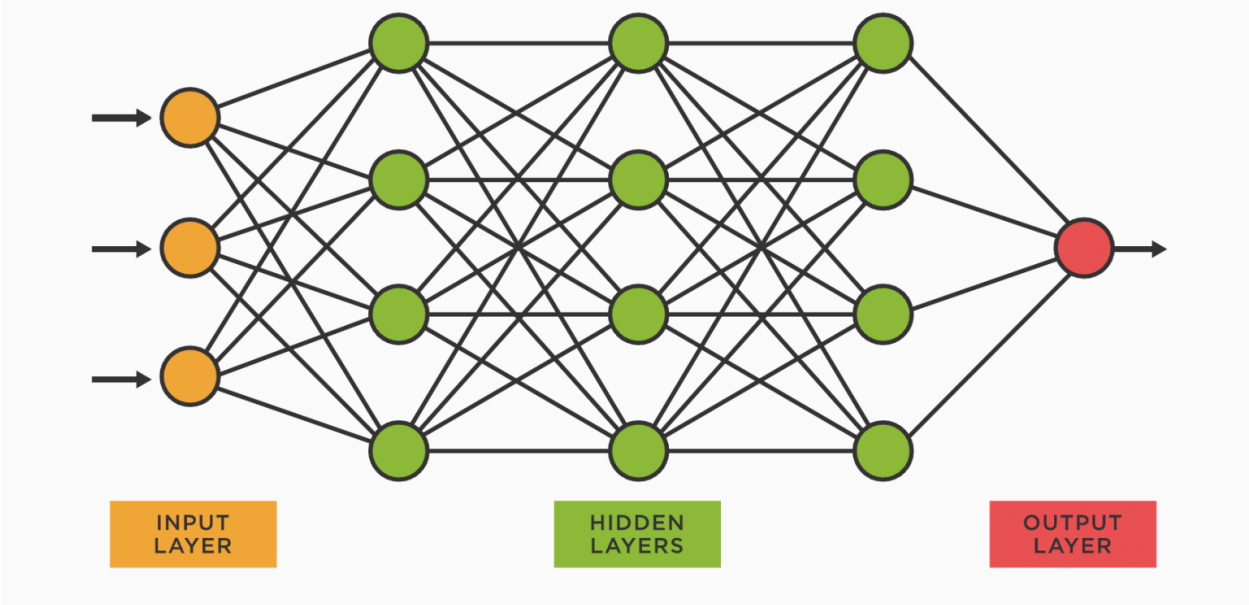


Fig. 2.21: Feed forward neuron network with 3 input neurons (orange), 3 hidden layers each with 4 neurons (green), and a single output neuron (red).

We would say that this function can be described by the number of neurons in each layer as a 3x4x4x4x1 network. This is known as network architecture. A neural network with multiple hidden layers is referred to as a deep neural network (Ivakhnenko & Lapa, 1967). Hereon, we will just use the term neural network for both shallow and deep neural networks.

As before, we can perform gradient descent to find the optimal weight tensor. If you recall, to update a weight, we subtract the learning rate multiplied by the gradient of the error with respect to the weight. This means we have to take the partial derivative of the output with respect to each weight. However, this is very computationally inefficient, especially for weights earlier in the function.

Due to the properties of the multivariable chain rule, function composition leads to a lot of overlap of partial derivatives. Weights leading directly to the same node are not distinguished between in subsequent layers, and thus will share most of their partial derivatives. Specifically, all partial derivatives with respect to network output of weights contributing to some subsequent layer node will contain the partial derivatives with respect to network output of the subsequent layer weights that are multiplied by this subsequent layer node. To update weights efficiently, we must avoid redundant gradient calculations.

For these reasons, we must first calculate the error for the final layer's weights and then propagate this error back to the weights originating from the input layer. By computing the errors for each layer's weights in reverse order, we can update a layer's weights simply by propagating the previously calculated error back one additional layer. This is known as **backpropagation**.

By performing simple matrix calculus assuming a ReLU activation function, we get the following result for the partial derivative of the error with respect to each weight. The final formula for the error of the weight connecting output i in layer k−1 to output j in layer k, propagated back from the weights connecting output j in layer k to each output h in layer k+1, where r^{k+1} is the number of outputs in layer k+1 is as follows:

$$\frac{\delta E}{\delta W_{ji}^{\ k}} = a_i^{\ k-1} * ReLU'(z_j^{\ k}) * \sum_{h=1}^{r^{k+1}} w_{jh}^{\ k+1} \frac{\delta E}{\delta W_{hj}^{\ k+1}} \tag{2.27}$$

The summation term is shared by all weights leading to output j in layer k, and is referred to as the ‘local error’ of that output/node. Note that the weight’s error will be 0 if the output it leads to or the output it emanates from do not activate. Since we are using ReLU, the partial derivative of a with respect to z is given by ReLU’. If we used a different activation function for a different type of approximation, we would use the derivative of that activation function.

As a developer, you must tune the number of layers and neurons in each layer to enable modeling of complex functions. Each layer must have sufficient neurons to model complex nonlinear functions. Additionally, there must be a sufficient number of layers to accurately model and utilize function composition. These two hyperparameters make up the model **architecture**. As mentioned above, insufficient segments in a layer can lead to underfitting, while too many segments can lead to overfitting. Additionally, as mentioned above, insufficient number of layers leads to underfitting unless there are a lot of function segments, which in turn is likely to lead to overfitting. However, if there are too many layers, the model has the potential to learn complex nonlinear boundaries and functions that learn noise, randomness, or fluctuations in the data that do not represent the true function.

2.4 - Classification

The world is continuous and infinitely detailed, but humans (and machines) have limited capacity to perceive, process, and remember such complexity. We also frequently encounter new situations. However, because certain fundamental aspects of the universe, such as the laws of physics, and human nature, such as the desire for romantic relationships, remain relatively stable, we can leverage past knowledge to make predictions and informed decisions in unfamiliar scenarios. This ability to generalize from learned information to unseen instances also facilitates effective communication and interpretation of concepts and meanings.

Generalization can be applied to generate continuous functions. Since we cannot effectively process, remember, or communicate continuous information and concepts, we need to concentrate on the most important aspects. For instance, we cannot convey the continuous spectrum of light reflected by letters on a page, which are also arranged continuously in space. Also, humans and robots interact with and interpret the world in ways that impact their states and potential actions, certain actions and ideas take on value that must be measured and conveyed. For example, things like interactions, objects, steps, tools, and completed tasks can influence decisions and outcomes. We only need to focus on information that has value to us. These require us to use discrete assignments for different pieces of information, such as individual letters or distinct colors. By organizing information into these discrete units, we can effectively utilize our prior knowledge about them when needed.

Consider one example of this in the physical space: Objects are composed of interconnected and interdependent parts, allowing knowledge about some parts to inform understanding of the whole. For instance, if one part of an object malfunctions, it often impacts the functionality of the entire object. This principle also applies to kinematic properties such as position, velocity, and acceleration, due to the internal forces that maintain the cohesion of the object. For example, if the front part of a ball moves at a certain velocity, the back part is likely moving at the same velocity. These interconnections enable us to generalize information from

specific parts to the entire object and vice versa, treating the object as a cohesive unit when analyzing its properties.

This means that we must have clearly delineated ideas as to what prior knowledge can or cannot be applied to different scenarios. So, people and machines must draw information-rich, yet somewhat arbitrary, conceptual boundaries around different aspects of the world, such as specific objects and concepts, grouping together these aspects based on certain similarities while leaving other aspects out of these groups due to differences. These groups generally have some underlying conceptual properties, known as the 'essence' of the group, which determine whether some aspect of the world falls into a group or not. Each group along with its essence is known as a **category**. Examples of categories are tall vs. short, water bottle vs. not water bottle, the brand of an item of clothing, the type of a disease, the breed of a dog, etc.In the universe, categories merely represent differences in atomic configuration with no connection. However, they have meaning because of how they are used and interpreted by humans and machines. Since humans already have established categories, we typically reuse them for robots.

Inside of groups, there are often special instances of the groups which are distinguished from the rest of the members of the group. For example, different dog breeds inside of the category of dog. This creates a parent-child hierarchy of groups with some groups being mutually exclusive, and others being overlapping. We often know general information about a parent category of different aspects of the world we are analyzing. For example, the fact that something we are seeing is a physical object. However, we must figure out specific subcategories, such as whether it is animate or inanimate, if it is animate, what type of being it is. The relevant parent category, such as the type of object or the type of animal depends on the specific applications.

This means that robots must be able to determine the categories of different aspects of the world, including which sub-category of a specific parent category that an aspect of the world falls into. This need is further reinforced by the fact that robots must act in a human world, meaning they must be able to understand human signals, such as what certain pieces of information mean to humans. For example, pick up oranges off of a conveyor belt but leave apples. This situation defines a series of subcategories, known as **classes**, that the robot must classify some aspect of the world as. This task of using data to predict which subclass of a relevant parent class something falls into is known as **classification**.

Humans and robots observe data via our senses, synthesize this data to get an understanding, and assign categories accordingly. We generally either collectively agree on assigning values, or act in predictable manners in specific situations. Essentially, the essence behind class attribution for examples/instances of the class is generally contained within specific information about the example, and this generally does not change significantly over time. For example, various attributes, such as physical attributes, utility, etc. influence whether something

is a house, car, etc. Since people assign values based on this data in similar manners, the information for classification contained within this data. This means that there is a relatively accurate function mapping inputs to classes in these situations.

Since sensors gauge physical quantities and not entity-defined ideas, these values are inherently unobservable. Since they are entity-defined, there are inherently no physical relationships from which we can derive them. This means we must figure out the true function mapping inputs to outputs via supervised machine learning. This means that we must use neural networks to figure out the function mapping observable data to classes. This means that a neural network learns the optimal function that maps inputs to classes. Essentially, the network learns to divide the input space into regions that correspond to different classes. Through iterative adjustments, the network refines the boundaries separating the different classes in the input space, known as **decision boundaries**. Two pictorial/graphical representations of a function drawn by a classifier, both with 3 classes, are shown below:

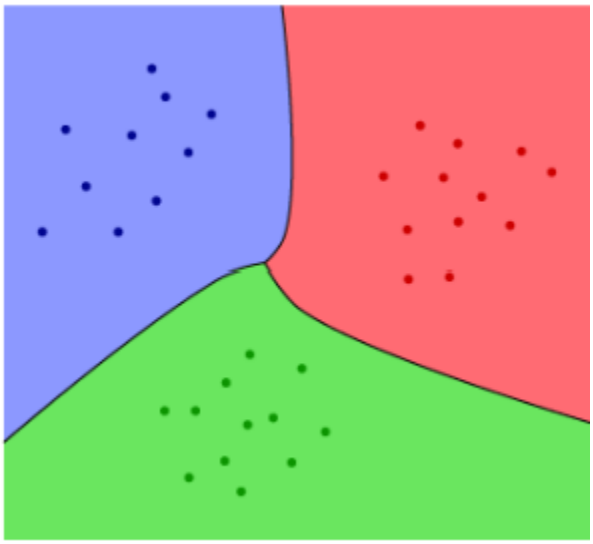


Fig. 2.22: A learned decision boundary dividing the input space into three regions, each corresponding to a distinct class (blue, red, and green), as determined by a classifier trained on labeled data.

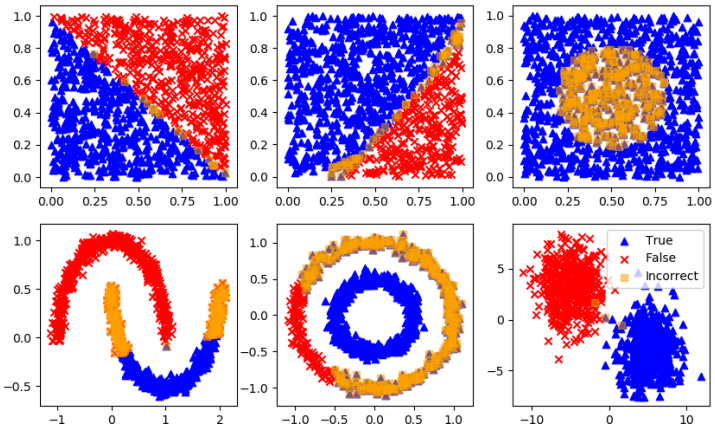


Fig. 2.23: A second example of decision boundaries learned by a classifier. Each subplot shows two-class classification problems (blue triangles and red Xs) with orange squares highlighting misclassified points. These examples illustrate increasingly complex class boundaries across various datasets

Generally, mathematics and computers cannot properly handle non-numerical data. So, we must convert our data to numeric data. For classification with N classes, there are N possible options for a prediction and ground truth. If all of the classes are mutually exclusive, all classes will be false except for the correct class which is true. To represent this, we assign each class an index from 0 to N-1 in an array of length N. The value at index corresponding to the true class is set to 1, while the values at other indices are set to 0. For example, if we want to identify whether a dog is a pitbull, golden retriever, or poodle, we would use an N=3 length array. If we set pitbull to index 0, golden retriever to index 1, and poodle to index 2, then a data sample with a label of pitbull would be represented as [1, 0, 0].

In regular regression, we predict some actual quantity. However, in classification, we are working with classes not quantities. Since a true value corresponds to '1' and a false value corresponds to '0', we can predict a value between 0 and 1 for each class and the highest one represents the prediction. In the case of mutual exclusivity, a high prediction in one category means that it is less likely to be in each other category. Since the sum of values in the true array equals 1, so should the values in the predicted array. Also, this allows us to interpret class prediction values as a discrete probability distribution

This means that each output must be between 0 and 1, with them summing to 1. One function capable of this is known as the **softmax function (**Boltzmann, 1862). It is given as below:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}} \tag{2.28}$$

The partial derivative of the i-th output with respect to the j-th input is as follows for i=j and i≠j respectively:

$$\frac{\partial \sigma(z)_i}{\partial z_i} = \sigma(z)_i(1 - \sigma(z)_i) \tag{2.29}$$

$$\frac{\partial \sigma(z)_i}{\partial z_i} = -\sigma(z)_i \sigma(z)_j \tag{2.30}$$

So, we pass our output for each class through this function to generate a corresponding array of length N-1 where the value at each index represents the predicted probability of the true class being the one corresponding to that index. In the case of classification, the fact that the sum of predicted class probabilities equals 1 means that higher predicted probabilities in incorrect classes necessarily denotes lower predicted probabilities in the correct class. This means that we only need to look at the correct class as it captures all of the information needed to quantify error.

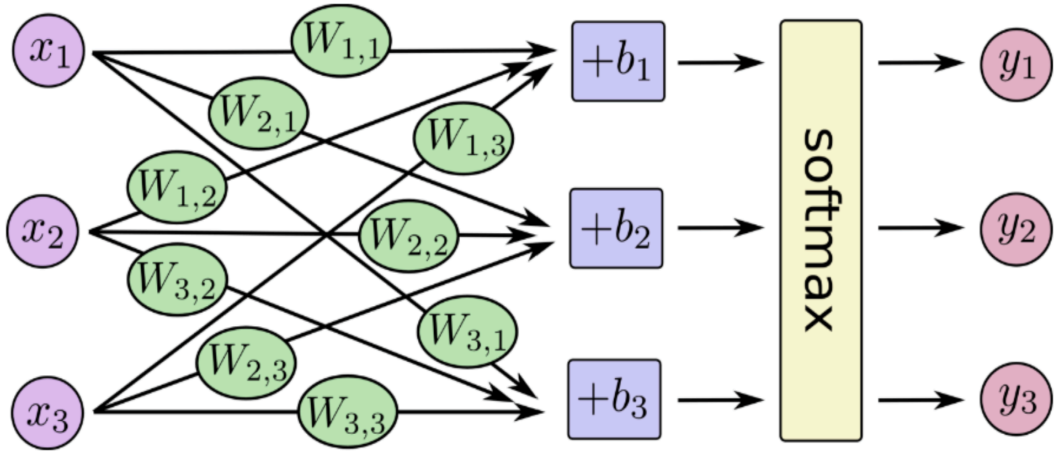


Fig. 2.24: A neural network for multi-class classification using the softmax function. Each output y_i represents the predicted probability for a class, computed by applying softmax to the linearly combined inputs x_j weighted by $W_{i,j}$ and shifted by biases b_i.

As demonstrated, the higher the predicted probability at the correct index, the better. So, our goal is to maximize this value. To evaluate output we only have the predicted class

probabilities array (x), where x_i is the predicted probability of the i-th output, and corresponding true output array (y), where y_i is 1 if the i-th class is the true class and 0 otherwise. This means to evaluate output we must calculate the probability of a correct prediction via a function of the two arrays. The function should equal the predicted probability corresponding to the 1 index, unaffected by predicted probabilities corresponding to a 0 index. Therefore, our correctness likelihood function for predicted x and true y arrays for C classes is as follows:

$$P = \prod_{i=1}^{C} x_i^{y_i} \tag{2.31}$$

However, this is difficult to use in a loss function because the product operator is difficult to manage mathematically. Sums on the other hand are much less computationally intense than products and simplify the gradient process. To convert a product into a sum, we can use the logarithm operator. The logarithm is a monotonic function, meaning that maximizing the likelihood and maximizing the log-likelihood yield the same parameter estimates. This property allows us to focus on the log-likelihood without losing the goal of maximizing the likelihood. This brings us to the following equation:

$$L = log(\prod_{i=1}^{C} x_i^{y_i}) = \sum_{i=1}^{C} y_i log(x_i) \tag{2.32}$$

However, since x_i will be less than 1, the output will be negative. Also, to stay consistent with regression loss, the goal higher numbers should be worse to enable gradient descent for error minimization. So, we multiply this loss function by -1. This brings our equation to:

$$L = -\sum_{i=1}^{C} y_i log(x_i) \tag{2.33}$$

This is known as **Categorical Cross-Entropy** (Shannon, 1948).

However, there is a special case of classification. If the classification is between two mutually exclusive classes, then knowledge of the true value not being one class entails that it must be the other. This special property differentiates this type of classification. So, if the classification is between multiple classes, or two classes that are not mutually exclusive, it belongs to a sub-category of classification known as **multi-class classification**. If the classification is between two mutually exclusive classes, it belongs to a sub-category of

classification known as **binary classification**. A pictorial/graphical representation of a function drawn by a binary classifier is shown below:

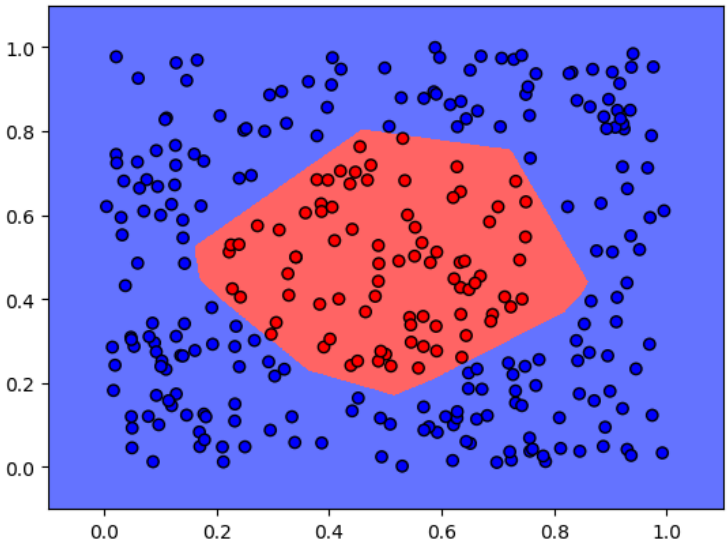


Fig. 2.25: A binary classification function dividing the input space into two mutually exclusive regions—one colored red and the other blue—each corresponding to a distinct class. The classifier predicts one class if the input falls in the red region, and the other if it falls in the blue region.

In binary classification, if the true class is one of them, it must not be the other. This means that if there are two classes, class 0 and class 1, we are essentially predicting whether the true class is class 1 or 0/not 1. This can be represented by a single binary number instead of an array. For example, has heart disease → 1, does not have heart disease → 0; died on the titanic → 1, lived on the titanic → 0, happy →1, sad → 0, etc. This simplifies the problem of binary classification to taking in input data and predicting whether the output is a 0 or a 1, or equivalently, whether it is 1 or not.

In this case, since the probability of class 0 = 1 - the probability of being class 1, and vice versa, the softmax function for an array reduces to a special case forcing a single input between 0 & 1 known as the **sigmoid function**.

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.34}$$

$$\frac{d}{dz}\sigma(z) = \sigma(z)(1 - \sigma(z)) \tag{2.35}$$

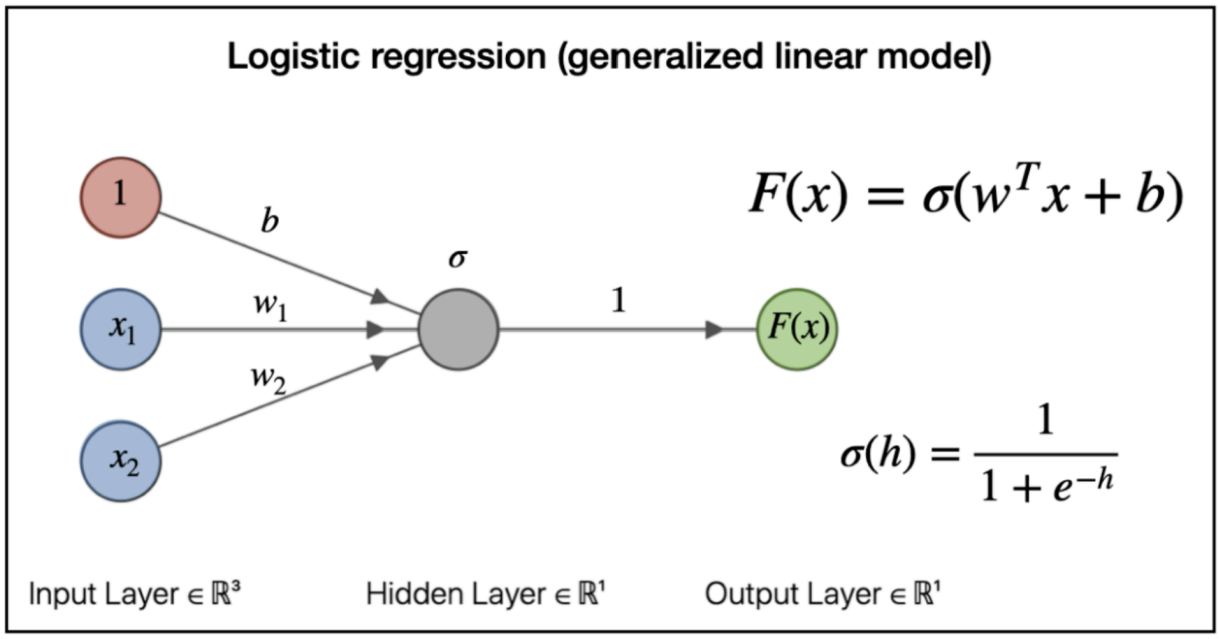


Fig. 2.26: A diagram of logistic regression. The model computes a weighted sum of the inputs plus a bias term and passes the result through the sigmoid activation function to produce an output between 0 and 1, representing the predicted probability of the positive class.

This also causes the loss to reduce to:

$$L = -\ (ylog(x)\ +\ (1\ -\ y)log(1\ -\ x)) \tag{2.36}$$

This equation for evaluating binary classification loss is known as **Binary Cross-Entropy**.

Due to the existence of categories and discrete concepts, they impact and hold value for humans and robots in various ways and magnitudes. For example, interactions, objects, steps, tools, and completed tasks all possess magnitudes that can influence decisions and outcomes. To properly interpret and compare these magnitudes, especially when they need to be communicated or used in calculations, they must be assigned an absolute quantity or number. This is why we create 'mental quantities,' such as amounts, counts, and trackers. For instance, the number of votes a candidate receives or the dollar price of a house.

While these numbers don't inherently exist, they gain meaning through use. While these values are significant for robots operating independently, the necessity for robots to function in a human world emphasizes the importance of understanding these values, as they are shaped by human interactions. We generally either collectively agree on assigning values or act in predictable manners in specific situations. For example, various attributes influence housing prices, such as location, size, and the number of bedrooms. Since people tend to assign housing

values based on this data in similar ways, the information for determining a house's price is embedded within this data. Note that this idea of the multiplicity of categories gives rise to the idea of integers. However, as discussed earlier regarding categories, these values are inherently unobservable. These values must be extracted using supervised machine learning, often employing neural networks to uncover the underlying patterns and relationships in the data.

Section II: Planning

Chapter 3: Path Planning and Mapping

3.1 - Introduction to Path Planning

For any autonomous system, the whole purpose it exists is to perform some task to alter some environment, whether applying physical forces, performing calculations, etc. This means it must be able to influence the likelihood of desired outcomes. This ability is typically embedded in its design. Each possible choice, operation, or decision which is realized into some environmental alteration is known as an **action**. Examples of actions are do nothing, move 1 meter forward, actuate some joint 2 rad/s, and buy stock X.

In robotics, the purpose of acting is to apply forces at specific contact points—a process known as **actuation**. At every moment, the robot must precisely control its actuation mechanisms to produce the desired effect. These mechanisms are typically motors that take electrical signals as input and generate force or torque as output. Each distinct configuration of these controls defines an action. To carry out its tasks effectively, the robot must be designed to support all movements necessary to interact with its environment. The complete set of possible actions defines the robot's **action space**, and each action can be viewed as a sample drawn from this space.

Since robots operate in a continuous world and motor adjustments are typically continuous—or at least very small—the action space is generally **continuous** as well. This means that the robot must choose from an infinite set of actions, typically represented by real-valued vectors. For example, a robot arm can rotate its joints to any angle within a range. Additionally, a drone can apply any thrust and rotate at any rate within physical limits. In contrast, a **discrete** action space consists of a finite set of possible actions the robot can choose from. For example, a robot can choose to move left, right, or stay. Additionally, a game-playing AI can select from actions like jump, run, or shoot. Note that since we live in a continuous world, robot actions can usually be continuous, but are constrained for simplicity.

To operate continuously in dynamic environments without requiring excessive human labor for long periods or being constrained by human reaction times, precision, etc., this acting must occur **autonomously**. For example, robotic arms in manufacturing use motorized joints to weld, assemble, or paint car parts based on pre-planned instructions or autonomous algorithms.

As mentioned in the introduction, the primary purpose of a robot is to move an external or internal object in some way. In many applications, this involves traveling from one location to another to perform a task. In both cases, the robot must actuate in specific ways to move from point A to point B. Whether it’s a mobile robot or drone moving its entire body, or a robotic arm

repositioning its fingers or end-effector, the robot is controlling its joints or motors to transition from one pose to another. This overall process—and the algorithms required to perform it reliably—is referred to as **navigation**.

Machine planning must ultimately result in motor motion. Motors are typically actuated by specifying either velocity or force/torque, depending on the control scheme. In many motors, specifying a velocity indirectly determines the required torque/force, and vice versa, due to the relationship governed by the system's dynamics, motor characteristics, and external forces such as friction and load. This set of final actions is the set of all possible combinations of joint values that define the configuration of a robot. Each point in joint space corresponds to a specific pose (or shape) of the robot's body, even if it doesn't directly specify the position of the end-effector in physical space. This space is known as the **joint space**. For example, a 2-link planar arm with two revolute joints has a joint space defined by the two joint angles: Joint space = (θ_1, θ_2). Each pair (θ_1, θ_2) corresponds to one unique configuration of the arm.

In mobile robots and drones, end goals are rarely specified in terms of motor configurations. Instead, they focus on the robot's overall motion through the physical environment. Robots, the obstacles they must avoid, and the targets they interact with all exist and move within this shared external space. As a result, most robotic tasks are naturally defined in terms of desired positions, orientations, or trajectories in the environment—for example, "pick up the object at this location" or "follow this path."

Since goals and obstacles are both expressed in physical coordinates, planning in this space greatly simplifies collision checking and path planning. Whether a motor configuration is considered effective depends on the resulting motion it produces in this space, including velocities and accelerations.

This space is known as the **task space** and refers to the space in which a robot interacts with the external world. It is typically defined by the position and orientation of the end-effector (e.g., the robot's hand, gripper, or sensor) in a fixed coordinate frame. Each point in this space corresponds to a unique pose of the part of the robot that performs the task. For example, the task space of a mobile robot on a 2D plane may be defined as (x,y,θ), where (x,y) is the position and θ is the heading angle.

In manipulator robots, the mapping from joint space to task space is highly non-linear and often redundant—meaning multiple joint configurations can produce the same end-effector pose. As a result, the robot must choose among several joint-level solutions for a given task-space goal. Additionally, manipulators must account for potential self-collisions, since certain joint motions or configurations can cause different arm links to interfere. Joint limits—such as maximum allowable angles or extension distances—further constrain the feasibility and quality of joint actions. This means that a single task-space action may correspond to multiple possible joint-space actions, whose suitability depends on factors like collision risk

and joint constraints. Since each of these considerations are only visible in joint space, we must plan in joint space to ensure that the selected trajectory is physically valid, safe, and executable by the robot.

Unlike manipulators, drones and mobile robots typically lack redundancy and the ability for self-collisions. For these reasons, we typically plan in the task space.

To describe all the relevant positions and orientations a robot can assume—recognizing that this depends on the type of robot, such as a mobile robot, drone, or manipulator—we introduce the concept of configuration space. **Configuration space**, often abbreviated as C-space, is a mathematical space in which each point represents a unique, valid configuration of the robot, typically defined by its degrees of freedom (DOFs). A single configuration captures all the internal parameters necessary to fully specify the robot's pose or state for planning purposes.

This abstraction allows us to plan in a unified way by mapping each robot's possible states into a configuration space, even though the nature of this space varies depending on the type of robot. For the reasons described above, the C-space of a mobile robot or drone is often equivalent to its task space—the space in which it moves and acts. In contrast, the C-space of a manipulator is typically its joint space, since the relevant planning variables are the joint angles rather than the end-effector's position directly.

Since the mathematical mapping between the kinematics and dynamics of the robots are derivable, we can easily obtain the forward and inverse kinematics and dynamics of the robot. This means that we convert some desired position, velocity, acceleration, etc. to the corresponding motor motion. To ensure that the motor configurations align with the desired velocities or acceleration, they must be continuously altered based on the observed kinematic and dynamic properties of the robot via the control feedback loop.

This means that machine planning entails that the robot follows some set of intermediate poses from a start configuration to an end configuration from which it derives wheel and/or joint motor motion. This set of poses is known as a **path**. Each viable path must respect a set of required specifications, such as avoiding obstacles. This refines our goal of navigation to mapping the state to the optimal path, meaning the one that meets the required specifications and will probabilistically maximize the reward function. This is known as **Path Planning**.

Paths must be continuous in both pose space and joint space because discontinuities would violate the physical and mechanical constraints of real-world robotic systems. Even when planning with discrete steps, the resulting motion must remain physically continuous—robots and their components cannot instantaneously jump from one position to another. In other words, robots cannot teleport; all movement must occur through smooth, connected transitions.

It is important to note that the kinematic aspects of planning are always relevant as they determine where the robot is going and where it will be. However, the robot also has orientational components to its pose that denote its heading. In many situations, the robot's

orientation affects its kinematic abilities. An example of this is robots that are nonholonomic, meaning that they cannot independently and simultaneously move in any relevant pose direction. Nonholonomic robots such as differential drive and car-like robots have turning constraints (e.g., they cannot move sideways). Planning without considering orientation would result in infeasible paths. Additionally, these types of robots cannot turn instantly, they require a smooth trajectory. Smooth trajectories considering momentum, turning radius, and inertia are especially important when moving at high speeds. Also, robots with elongated bodies or trailers need orientation-aware motion planning to avoid collisions. Additionally, considering orientation is important for precise navigation where the robot must align itself with an exact pose to perform a task, such as a drone landing on a charging pad. In such situations, the path planner must explicitly consider orientation.

However, this is not always the case. Many robots are **holonomic**, meaning they can move in any direction without needing to reorient themselves. Additionally, if the robot is moving slowly and minimizing time is not a critical objective, it can stop, rotate in place, and then resume moving. Also, in situations where orientation matters but is not a major factor in the overall reward, the appropriate orientation can often be handled by a lower-level motion controller based on the desired path.

Another important note is that in many scenarios, considering the robot's current velocity and acceleration are relevant for optimal control. This is relevant for dynamic feasibility as the planner must ensure that the commands it produces are achievable given the robot's momentum and inertial properties. Additionally, incorporating current dynamics helps in generating smooth trajectories that avoid abrupt changes, which is important for both safety and performance. Also, the robot's current kinematic state will alter the robot's trajectory as it transitions to the new kinematic state.

However, in many scenarios where the robot does not need absolute precision of navigation or is not moving very quickly, there will not be significant harm if the path is not very smooth and the robot is unlikely to act significantly outside of its dynamic constraints. As with orientation, if the current kinematic state is relevant but not a large determiner of reward, then the smooth transition from the current to desired kinematic state can be handled by a lower level motion controller.

3.2 - Introduction to Machine Planning

While a machine may have many possible actions, some are more advantageous than others and more likely to align with the stakeholders' goals. For instance, prioritizing certain tasks can lead to faster completion, which is often more desirable. Likewise, different navigation paths can impact outcomes such as travel time, crash risk, or energy consumption. To guide the machine's design, guide the machine's behavior, and compare the effectiveness of different

actions, it is essential to define an overarching reward metric. This metric assigns numerical values to outcomes, enabling the system to evaluate and select better actions. In navigation tasks, three main reward metrics are commonly used:

- **Damage Sustained:** Damage reduces the system's ability to function properly and increases costs over time. In many applications, damage avoidance is a matter of safety and sustainability. A damaged system may become non-operational, causing mission failure or requiring costly repairs. A drone navigating through a forest must avoid collisions with trees, since damage to its rotors could result in loss of control and mission failure.
- **Time to Completion:** Time is essentially a scarce resource. In almost every engineered system, completing tasks faster improves the overall system throughout. Completing a task quickly generally leads to higher overall productivity and efficiency. Additionally, many applications require real-time or near-real-time responses (e.g., autonomous driving, surgical robots).
- **Energy Consumption:** Energy is a fundamental physical and economic constraint. Efficient energy use allows for longer, more cost-effective operation. Mobile robots, drones, and autonomous vehicles are typically battery-powered, making energy a limited resource. Reducing energy consumption allows the system to operate for longer periods without needing to recharge or refuel. Energy use translates directly to operational costs (e.g., fuel, electricity). Reducing energy consumption lowers these costs.

Since higher values of the three metrics above for physical systems are typically seen as negative, we sometimes use the term **cost** (instead of reward) to refer to the metric that combines damage, energy, and/or time. These terms are often used interchangeably since minimizing cost is the same as maximizing reward. The overarching goal of the robot then becomes minimizing this overall cost.

As mentioned in the introduction chapter, at each point in time the robot gauges the external world via observations from sensors. We use a combination of information processing and our current and relevant historical observations to gauge all relevant observable factors for decision making. This set of variables is known as the robot's **state**. Ideally, the state would contain all the information needed to determine the optimal action and predict the resulting state for each possible action, but this is not always feasible. Many minor variables are not analyzed for computational purposes. Additionally, many relevant factors are unobservable, such as objects behind walls.

By design, the state contains all relevant information needed to make decisions and predict future states, incorporating all current and past measurements. As a result, past states are not needed directly, since any useful information they held is already encoded in the current state. For example, a self-driving car relies solely on its current position, velocity, and map of

nearby obstacles to decide whether to break or change lanes—without referencing its entire driving history. This allows us to assume that future states depend only on the current state, meaning the optimal action is based solely on the current state. This assumption is known as the **Markov Property** (Markov, 1906). Note that the state at time step t is denoted as s_t, and the action taken at time step t is a_t. The markov property can be mathematically described as below for the probability of a given subsequent state and reward as a function of prior states and actions at various time steps:

$$P(s_{t+1}|s_t, a_t, s_{t-1}\, a_{t-1},..., s_0,\ a_0) = P(s_{t+1}|s_t, a_t) \tag{3.1}$$

$$P(r_t|s_t, a_t, s_{t-1}\, a_{t-1},..., s_0,\ a_0) = P(r_t|s_t, a_t) \tag{3.2}$$

Note that the combination of taking a specific action in a specific state is known as a state-action pair. The value of a state action pair is essentially the expected reward received from taking that action in that state. The Markov Property entails that each state-action pair has a single value independent of past states or actions. This additionally entails that each state has some optimal action independent of past states or actions. This refines the goal of machine planning to finding a function (π_θ) that deterministically maps states to expected optimal actions. This function is known as the robot's **policy**.

Note that an action in a state is not inherently good or bad—its value depends on the states it leads to and the rewards received in the future. Additionally, once you are in a given state, the past is fixed and cannot be changed. Any action you take can only influence what happens next. For these reasons, when evaluating a state-action pair, we focus solely on future states and rewards, not past ones. Since an action can potentially affect all future outcomes, we must consider the entire future trajectory of the robot's behavior when assessing its value.

In each state, the agent takes the optimal action for that state. This means that the value of a state is simply the expected reward of the optimal action taken in that state, assuming that the currently believed optimal policy is followed subsequently.

$$a^*_t = argmax_{a_t}(G(s_t, a_t)) \tag{3.3}$$

$$V(s_t) = G(s_t, a^*_t) \tag{3.4}$$

Meanwhile, the value of a state-action pair consists of the value of the subsequent state. However, each state transition usually contributes to the overall reward factors. In fact, the total reward is made up of the contributions of each state transition. In robotics, each state transition necessarily takes some time, has some risk potential, and uses some amount of energy. The total time spent, energy spent, and damage accumulated consists of the time spent, energy

spent, and damage accumulated from each state transition. This is definitionally true. The reward captures the quality of the robotic endeavor, which necessarily consists entirely of the state transitions that occurred during the robotic endeavor. Each state transition imparts some immediate reward, represented as r_t.

Considering r_t is a part of gauging the optimal action in a state. However, future damage sustained, energy consumption, and time to completion are not directly observable. For the collision aspect, distance of a pose to an obstacle generally provides an observable, sufficient heuristic. If the robot is close to an obstacle, you want to incentivize a greater distance between the obstacle and the robot. For gauging wear and tear to the robot, path smoothness measurements such as jerk, acceleration, or velocity may be used as heuristics. For time and energy, the distance between two poses is generally an observable, sufficient heuristic. If the velocity of traversal is known, the time-based cost can be directly calculated. Otherwise, a predictive or empirical model must be used to estimate traversal time, considering factors such as terrain, slope, and weather. The same is true for energy usage if it cannot be easily predicted based on distance or velocity. When these factors change dynamically, the cost of each edge must be re-gauged or calculated using an appropriate model before path planning.

The value of a state can be viewed as a recursive function of immediate rewards of state transitions. We define total return as a sum because it naturally captures the idea that each time step contributes an independent piece of feedback (reward) that adds to the overall performance. This means that throughout some endeavor, the total reward is the sum of immediate rewards experienced throughout the endeavor. This is expressed below as the total reward following an action at time step t, denoted as G_t, defined as a function of the reward received at each subsequent time step, r_t:

$$G_t = \sum_{k=t}^{T} r_t \tag{3.5}$$

Note that if the robotic task has a defined end time or termination condition, then T represents the finite final time step, and the task is considered **episodic**. If no such endpoint exists, T is treated as approximately infinite, and the task is considered **continuous**.

The reason we have to explicitly consider r in path planning is that different actions may have different immediate costs, such as energy consumption, time, risk, or control effort. However, if each action has approximately similar costs, then we would merely have to consider the subsequent state that the action led to. This simplifies the optimal action to the one that leads to the optimal state. Additionally, this would refine our path planning goal to minimizing the number of time steps/decisions made, as each decision/time step would have the same cost.

The recursive nature of G means that the reward of some state action pair can be written as a function of the immediate reward and reward following the subsequent state. This alleviates the need to always consider all future states. This is described as below:

$$G_t = r_t + G_{t+1} \tag{3.6}$$

This means that the value of a state-action pair, typically called the 'Q-value' of that pair, as well as the value of a state can be denoted as follows:

$$Q(s, a) = r_t + Q(s_{t+1}, a^*_t) \tag{3.7}$$

$$V(s) = r_t + Q(s_{t+1}, a^*_t) \tag{3.8}$$

These are modified versions of what are known as the **Bellman Equations**.

As mentioned above, the total future reward for a state-action at time step t pair is entirely determined by the sequence of states and actions from time step t to final time step T. This sequence known as a **trajectory** and denoted as τ in equations, is shown as follows:

$$\tau = \{s_t, a_t, s_{t+1}, a_{t+1}, \ldots, s_{T-1}, a_T, s_T\} \tag{3.9}$$

In reality, trajectories are continuous, with each state separated by an infinitesimal amount of time. However, when we evaluate trajectories or collect data, we are limited by the rate at which we can sample data or update actions. As a result, we often work with discrete trajectories, which consist of states and actions recorded at specific time intervals.

In many environments, the same action in the same state can lead to different outcomes due to randomness in system dynamics or if the robot does not observe all relevant variables for state-action value estimation. This randomness makes the environment **stochastic**. This means that there is a probability associated with transitioning to each possible subsequent state given the current state and action. This probability is denoted as $P(s' \mid s,a)$. Essentially, for each state-action pair, there is some true probability distribution we sample from when taking an action. In a finite setting, this is represented by a discrete probability distribution. This means that many trajectories, possibly even infinite, could follow a state-action pair. In a state, the agent's actions influence the subsequent trajectory probabilities, meaning that different actions have different subsequent trajectory probabilities. To maximize how much reward will be accumulated in the future, the agent must seek to maximize the expected return following a state, weighting the reward reaped by possible trajectories by their respective probabilities. In other

words, the optimal action in a state is the one that maximizes expected subsequent return following the same policy.

If this is the case, then the above modified Bellman equations are realized as followed:

$$Q(s, a) = E[r_t + Q(s_{t+1}, a^*_{t+1})|s_t = s, a_t = a] \tag{3.10}$$

$$V(s) = E[r_t + Q(s_{t+1}, a^*_{t+1})|s_t = s, a_t = a^*] \tag{3.11}$$

3.3 - Global Path Planning

To select the optimal action, we must approximate the cost a robot will incur and the reward it will receive after transitioning to a new state. However, future costs are not directly observable, and in stochastic (random) environments, even the immediate reward may be uncertain. While we can attempt to estimate these quantities, doing so can be challenging.

When analyzing two poses—such as a future pose at time t+1 or a final pose at the end of a trajectory—we often have information about their positions and, in some cases, additional context like terrain, weather, path congestion, or energy usage, depending on the cost metrics involved. However, the total cost of a trajectory depends on the characteristics of the path between these poses. For instance, terrain may vary continuously across the workspace, some areas may have speed restrictions, and obstacles may increase traversal cost. Therefore, to accurately evaluate the cost, we must consider the entire trajectory—not just the individual poses.

Even when the cost metric is purely based on distance, the distance between two poses alone might not fully capture all the necessary information. For example, if a direct path between two poses is blocked by obstacles (walls, cliffs, or restricted zones), simply calculating the Euclidean distance between them is insufficient. Heuristic approaches like straight-line distance assume direct movement, but real-world paths are often constrained by the environment.

Lastly, in greedy planning, an action is selected without regard as to whether the subsequent path connects to the goal in an efficient manner. Greedy approaches that prioritize immediate proximity might not account for whether the chosen path leads to the goal or whether it might force the robot into a dead-end, requiring backtracking and making the search inefficient. They also might miss narrow pathways that connect to the goal and opt for a less efficient path.

This illustrates that selecting the optimal action requires analyzing the entire trajectory from the current pose to the goal, assuming the robot has all relevant state information to do so, and said state information will not drastically change. This paradigm where the robot calculates the full trajectory before making a move is known as **global planning**.

When planning a global path, the trajectory needs to be continuous in both position and orientation, must reach the goal pose precisely, and must satisfy constraints like obstacle avoidance. Moreover, the optimal action for the robot depends on the pose it currently inhabits. Since we plan the path ahead of time, we must have some knowledge of the robot's pose at each kinematic alteration. If the robot were to plan only in terms of velocity, it would have to integrate velocities over time to reconstruct the pose, which introduces unnecessary complexity and potential drift. Therefore, for global planning, it is more direct and reliable to plan in the pose space, where both position and orientation are explicitly considered. Higher order kinematic properties such as velocity and acceleration should be considered as they relate to the overall reward metrics.

In global planning, our path consists of an action to take at each point, given a specific state and ending point. This means that our policy only has to take into account states that will occur during the path, potentially with some probabilistic deviations if our state transitions have elements of stochasticity. If there are some potential states that we are unlikely to see on the current path, our policy does not need to explicitly take those states into account or figure out the action to take in those states. Doing so may compromise performance in the robot's expected scenario and require unnecessary processing power, time, or data. This is especially relevant as global path planning generally assumes that the robot will not deviate heavily from its path and the environment is approximately static. This means that the goal, and obstacles do not move. This also assumes that we do not find out a lot of new information about the environment during navigation, as it may give us knowledge that would render our old solution sub-optimal. In this sense, we are essentially choosing a single kinematic trajectory to take out of all feasible trajectories, given our start pose, end pose, environmental state, and constraints, that minimizes expected total cost. Choosing a policy is essentially choosing a trajectory. When the environment is fully known and static, selecting a rule or mapping from states to actions results in a deterministic path the robot will follow from its starting state to the goal.

In order for the robot to plan a global path that avoids obstacles and successfully connects to the goal, it must be aware of all relevant obstacles and free space in the environment. Additionally, to generate an optimal path, the robot must be aware of all relevant information for determining the cost of navigation through each pose of the workspace, such as weather, terrain, distance to the nearest obstacle, external velocity/acceleration limits, etc. as applicable for that application. This means that the robot must have some stored encoding of all of this relevant spatial and other information about its workspace in a format that can be used to inform navigation. This is known as a **map**.

These maps are usually 2D (excluding vertical (z)) or 3D depending on the flatness of the environment, whether the robot can move in all 3 positional directions, as well as the robot's computational and storage requirements. Note that if the entire workspace and relevant

information can be sensed by the robot in real time, such as knowing obstacle locations, then it can act accordingly without prior maps.

If not, then the robot is unable to plan a path all the way to its goal. This is very limiting as information indicative of the best path, such as the locations of obstacles behind other obstacles or out of the range of the robot's sensors, will be unobserved by the robot. In this situation, the robot must have all or part of a premade map encoding this relevant information for the entire workspace.

However, having a premade map for these areas is not always feasible. This requires prior sensor scanning or manual data input for workspaces. This means that the workspace must be known ahead of time, making it infeasible for robots that need to be deployed to one of many locations on demand, such as search-and-rescue robots. It also must be previously explored, making it infeasible in previously unexplored environments. Additionally, spatial maps rely on static (still) obstacles as dynamic (moving) obstacles would invalidate any map because they would soon be at a different location as they were before. This means that a premade map is infeasible if terrain, obstacles, pathways, etc. map change frequently. Examples of this include warehouses with moving shelves, robots, or forklifts; urban environments with shifting traffic patterns, road closures, or construction zones; and natural settings where fallen trees, landslides, or flooding alter navigable paths. Lastly, environments that are too large to map completely make storing and processing a full map impractical. Examples of this include long-distance delivery drones covering entire cities or regions, underwater exploration where the terrain is vast and largely unknown, and expansive indoor facilities such as airports and large warehouses. Note that this is also largely a function of the robot's memory capabilities.

In the above situations, global path planning is not feasible. However, in some other situations where the robot has no pre-made map, it can map the entire workspace in real-time. However, since global path planning requires planning an entire path consisting of many dependent decisions, it can be computationally intensive and time consuming. This means that the robot may not be able to calculate paths in real-time. A robot may be able to calculate its path before path traversal. However, the guarantee of the path's optimality relies on the attributes of the environment which affect path cost, such as terrain, location of free space/obstacles, speeds, etc., staying constant. This is known as a static environment. For example, a path created when a person was to the left of the robot may be invalid or at least suboptimal if the person walks forward 5 feet. The global algorithms have no inherent mechanism of reaction or adjustment.

In situations with a low frequency of meaningful change, such as weather or a slow moving obstacle, a robot can still replan or adjust its path with advanced hardware and the proper algorithms. However, the necessary speed of path adjustment or replanning for global algorithms is currently not computationally feasible with multiple moving obstacles, or a quickly moving obstacle or target. In these highly dynamic environments, the optimal decision at point in time

changes, invalidating a prior decision for what to do at a certain position. Additionally, in large environments, smaller or less frequent changes may require more computational steps as there is likely to be more path that requires replanning.

Given that the future is inherently uncertain, direct knowledge of future outcomes remains elusive. To account for this, the robot can try to predict how the environment will change. However, we often lack an accurate function or model to describe how states transition over time. This challenge arises from the complexity of physical relationships that can be too intricate to derive analytically, as well as from the presence of conscious actors whose decisions are not predetermined. The state as well as the new knowledge we gain as we move will often change in unpredictable, random, or chaotic ways that lead to non-negligible error in predictions. In such cases, the robot cannot rely on a fixed plan and must instead make decisions in real-time based on its current state. The robot must gauge this state via its immediate, updated sensor inputs. In this paradigm, the robot must periodically use its current and relevant past sensor data to gauge the state, then map it to the subsequent action. The robot usually only plans one action ahead of time as it will have to re-evaluate before the subsequent action. The frequency of this update depends on the computational power of the robot as well as the environmental dynamics. This paradigm where the robot only plans one action at a time based on the current state is known as **local path planning**.

In local planning, maps are not required but may benefit the robot with knowledge beyond what it can currently observe, which may help compute subsequent costs and rewards. As mentioned above, this requires some form of spatial memory.

Note that both local and global planning are forms of machine planning, they are just different ways of finding the solution to the optimal action equations defined in 3.2.

3.4 - Mapping

3.4.1 - Map Creation

In many path planning scenarios, the robot's available information — consisting of any pre-made map and its current sensor data — often does not provide a comprehensive understanding of the environment. As the robot moves through its workspace, it will inevitably encounter new parts of the environment due to changes in its position and orientation, which shift the field of view of its sensors. Additionally, there may be relevant information that the robot cannot currently observe but has previously detected. For example, a robot may turn away from an object and need to remember its location even when it is no longer in view. Additionally, the robot may need to revisit areas it has already explored, making it essential to recall previously observed details. Even if a pre-built map is available, it might be incomplete or outdated due to environmental changes—such as a newly placed obstacle or a piece of furniture that has been moved. Such changes can impact navigation costs and alter which areas are

accessible or blocked. In such cases, the robot must incorporate previously observed spatial information into its working memory and actively update its internal map in real time. This means the robot must continuously perform real-time map creation and augmentation.

If we were using a LiDAR sensor, we would generate a **point cloud** (series of 3D environmental points) for surfaces in our environment to be used as our map or as input for further processing to obtain our map. For a 3D map, a point cloud is a set of points typically represented as a vector of coordinates in a 3D space. This is shown below for point cloud A:

$$A = \{(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots (x_n, y_n, z_n)\} \tag{3.12}$$

For a given point cloud, if a beam returns after hitting an obstacle, that point indicates the surface of an obstacle. In this situation, because the light was unobstructed in its path to the obstacle, this indicates that there are no obstacles at points along the path of the light. If the beam does not return within its maximum range, this indicates that there are no obstacles at points from the sensor to the maximum range along the direction of the light ray. If many points along the outside of some region have been sensed but none inside of that region, then light is never reflecting off of the inside of the region back to the LiDAR sensors. This means that the inside of the region is likely inaccessible to light because it is contained within an obstacle. It doesn't really matter that this region is inaccessible to our lidar sensors as there will be no mapped path to the region and it will not affect the smallest distance to an obstacle from any point on the map.

Since we must store and compare points collected at different times with LiDAR sensors at different positions, we must store the positions of all points in a shared absolute cartesian reference frame. We receive the LiDAR data with some position relative to the robot. Via odometry/IMU/encoder data, we estimate the position of the robot in the absolute reference frame as well as the angular rotations of the robot's axes relative to the absolute axes. By using those sensors along with a LiDAR sensor, we can create a full map of points in some absolute coordinate frame while the robot is moving.

As is well known, to transform a point from the robot's local reference frame to a global (absolute) reference frame, we first rotate the point using the robot's rotation matrix R, which defines its orientation relative to the global frame. We then translate the result by the robot's global position t. This transformation is expressed by the following equation:

$$p_i^{global} = Rp_i^{local} + t \tag{3.13}$$

In an ideal scenario, one could measure every point in the environment to create an extremely accurate representation—a process known as **dense mapping**. However, dense

mapping is often impractical due to limitations in computational power, storage capacity, and processing time. Additionally, dense maps frequently contain a large amount of redundant or unnecessary data. For example, a flat wall might be represented by thousands of nearly identical points. Additionally, areas like ceilings—where no interaction occurs—may still be densely captured despite offering little value for decision-making or navigation.

To solve these problems, for larger maps or maps where computational efficiency is a priority, we often selectively process and store a smaller number of points which are chosen for their significance in defining the structure and features of the environment. We filter out the other points from the point cloud. By only analyzing the most important points, we can extract relevant information for mapping while reducing the usage of vital resources. This approach is known as **Sparse Mapping**.

3.4.2 - Grid Mapping

The robot's mapping and navigational goals will determine the **map representation** used for formatting and storing the relevant spatial information. Points are infinitesimally small. Even if we assume each point represents an area according to the resolution of our sensors, each point is still very small. This is wildly inefficient as we would have to store an extremely large amount of information to even map small spaces. Storing the occupancy of each possible point is untenable. Additionally, adjacent points in a dense map often correspond to the same continuous surface or share nearly identical distance measurements. For example, when scanning a flat wall, neighboring points will typically lie on the same plane and report very similar depth values. In such cases, storing every individual point becomes redundant, as the information provided by one point is nearly identical to that of its neighbors.

Also, for navigation, we do not usually care about very small spaces. Our robots take up a significant amount of space and thus require that amount of space to navigate. In addition, we know that our sensors and actuators are not perfect. So, if we know the location of a closeby group of points, we do not want to navigate very close to these points and would want to mark close-by locations as non-navigable. Essentially, we wish for the robot to maintain a safety margin around known obstacles, called a buffer zone, that prevents navigation too close to potentially risky areas.

To reduce redundancy and simplify processing, we should group nearby points that share similar locations into a single representative unit. This process effectively discretizes the continuous environmental variables in the state space by dividing the environment into distinct regions. Each region summarizes the data of its contained points, allowing the robot to reason about the world at a coarser but more manageable resolution. To ensure a uniform map with the simplest shape for minimal computational intensity, these points are usually grouped into small squares for 2D maps, or cubes for 3D maps. Each square is known as a **cell** or **voxel**. Each cell

contains the relevant environmental state variables for the region it represents, such as position, terrain, distance to the nearest obstacle, and of course occupancy. This approach where we break up our environment into a grid is called **Grid Mapping** (GM). A grid map is a practical way to represent large environments without requiring excessive computational resources.

Since we generally have an area we want to map, this grid is initialized over the entire mapping area, typically based on the boundaries defined by the environment's dimensions or the known operating area of the robot. If the robot is mapping existing stored or observed data, then the map is initialized over the environment to which this sensor information pertains. Each voxel's location is determined by its grid indices (i, j, k) in the chosen coordinate system. For a 2D map used in flat environments or simplified representations, voxels are placed on a plane (x, y), with each voxel having a fixed height (z). In a 3D map used in more complex environments, voxels are placed at coordinates (x, y, z) in the 3D space, covering the volume of the environment. The orientation of each voxel is typically aligned with the axes of the coordinate system used. For example, in a 2D map, voxels are oriented along the x and y axes. In a 3D map, they are oriented along the x, y, and z axes. However, the environment often changes and information is not all available to the robot at once. So, as the robot moves and explores the environment, the grid is dynamically updated to include new voxels based on new sensor observations. To allow for quick retrieval of voxel information and efficient memory usage, 2D OGMs are typically stored using a **quadtree** and 3D OGMs using an **octree**.

For navigation, we primarily care which areas the robot can navigate through and which ones it cannot. So, we should store the navigability of each voxel. The primary cause for a voxel not being navigable is the presence of an obstacle in that voxel, or if the robot being in the center of the voxel would entail a collision with a nearby obstacle. So, we must store whether each voxel is free or occupied. This is called **Occupancy Grid Mapping (OGM)** (Moravec & Elves, 1985). Since the occupancy of a cell is generally binary, we can store each cell as a 1 for occupied, or 0 for free.

Each LiDAR ray sent out indicates 0 or more voxels as free, and 0 or 1 as occupied. For each indicated voxel, the proportion of occupied indications vs. unoccupied indications informs whether the voxel is occupied or not. Some threshold is usually used where if the occupancy proportion is above that threshold, the voxel is marked as occupied (1), otherwise it is marked as free (0).

The simplest way to create a grid map is to make each cell the same size. In this situation, we decide the area or volume of each voxel ahead of time. This determines the resolution of the map. This approach is called **Uniform Grid Mapping** (UGM). By using UGM, we can easily store and retrieve cell information. Also, we can easily predict how much storage we will need ahead of time.

To be precise enough for near-optimal navigation, the voxels have to be small enough such that we would need a lot of storage for large environments. So, UGM is usually used in small-medium sized environments where objects can be discretely represented.

However, it is very common for robots to need to navigate through, and thus store maps of, complex environments where the obstacles, free space, and terrain vary in size/scale and shape. In UGM, we could use large voxels in these situations, but then we would not properly map smaller or more complex obstacles/free space. This is because the smaller or more complex objects would entail partial occupancy of a voxel. This introduced inaccuracies into our map. If only part of a voxel is occupied and we mark it as occupied, we are ignoring free space that is potentially vital for optimal navigation. If we marked that same voxel as free, we could potentially run into obstacles or plan a path that only works if that entire voxel is free. This shows that a partially occupied voxel should instead be represented with smaller voxels to more accurately represent the spatial occupation. We could simply use smaller voxels, but that would require a lot more storage and would lead to a lot of redundant voxels for larger and simpler objects/free space.

In these situations, large regions that are fully free or fully occupied can be properly mapped with larger voxels. We want a sparser map of these regions for efficiency. However, regions where there are smaller but relevant occupancy/surface variations, which would entail partial occupancy if contained within a large voxel, require smaller voxels. We want a denser map of these regions for accuracy.

Additionally, when navigating near obstacles, the robot needs to make precise adjustments to avoid collisions. If the waypoint resolution is too low near obstacles, the planner might produce paths that cut too close to obstacles, leading to collisions, or fail to find narrow passages. In open space, smaller path adjustments will likely have a smaller effect on path feasibility or cost. This is mainly because obstacles aren't present, so the path is mostly direct, and coarser waypoints are sufficient to create a smooth, efficient path. This means that high resolution would add unnecessary computational cost without improving path quality.

This shows the need for voxels of different sizes in these complex environments. This type of mapping where we vary the size of voxels as needed is called **Variable Grid Mapping** (VGM).

An image of a 2D and 3D VGM are below:

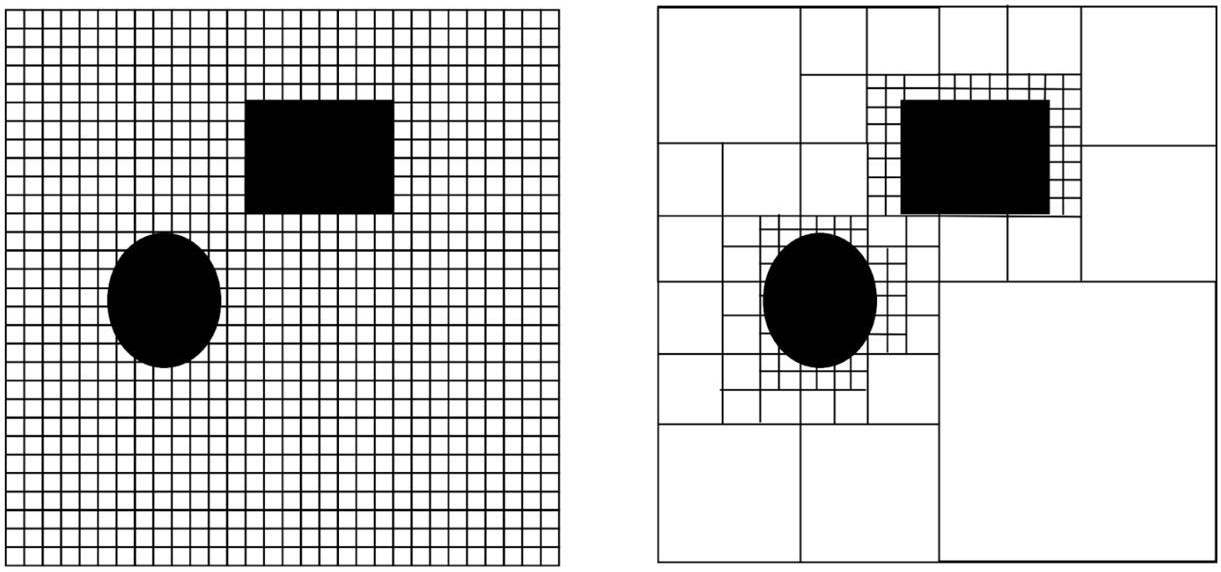


Fig. 3.1: Comparison of uniform grid decomposition (left) and variable grid map decomposition (right) in a 2D environment with circular and rectangular obstacles. The uniform grid uses fixed-resolution cells throughout the space, while the VGM adaptively subdivides space into finer cells near obstacle boundaries and uses coarser cells in free space, optimizing memory and computation for path planning. [MIT Center for Bits and Atoms (CBA)]

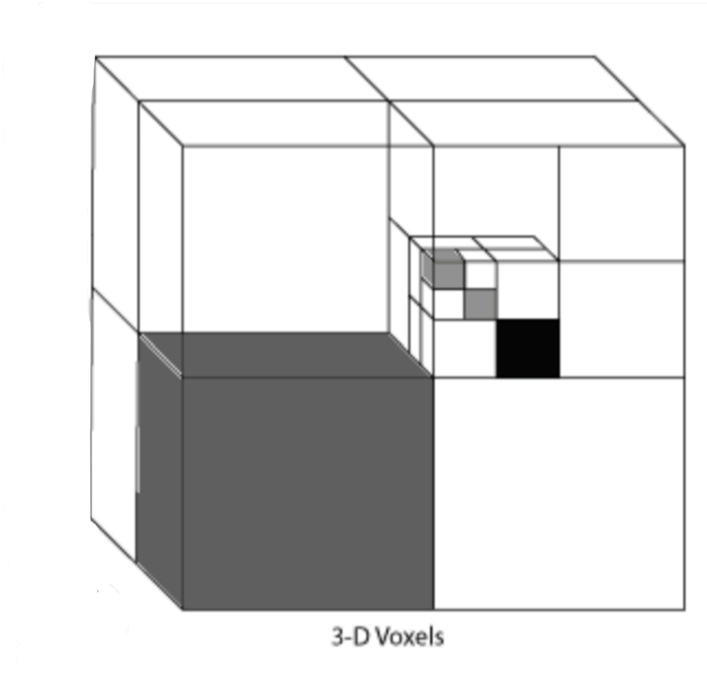


Fig: 3.2: Three-dimensional hierarchical voxel grid representation, commonly used in octree-based occupancy mapping. Larger voxels represent free or uniformly occupied regions, while finer voxels provide higher resolution near boundaries and complex areas. [Mathworks (n.d.)]

To reduce memory requirements, allow our map to be used in environments where we don't need high resolution in many areas, and cover large areas initially with the ability to increase the resolution for areas as needed, we start with large voxels and use smaller voxels as needed. Essentially, we initially map the relevant area with large voxels. We recursively check if the resolution for each specific voxel properly maps the portion of the environment represented by said voxel, and split up the voxel into smaller voxels if needed. This means that we check if each voxel is partially occupied. If it is, the voxel is subdivided into a fixed number of smaller voxels. Each of these smaller voxels is also checked as to whether it should be subdivided into a fixed number of smaller voxels and so on.

Note that to ensure equal sized squares/cubes when splitting a voxel, we usually divide voxels into four voxels for 2D maps or 8 voxels for 3D maps. Also note that the choice of initial voxel size depends on the computational capabilities of the robot's onboard hardware as well as the estimated level of detail required for the map based on the specific application.

Partial occupancy of a voxel means that some light rays passing into a voxel will be reflected back while others will pass through. This means that we will have conflicting sensor data. This means that we will have a high uncertainty as to whether the voxel is occupied or free. This means that the occupancy confidence will be closer to .5. This means that we can pick some range centered around .5 and subdivide voxels if their occupancy probability falls within this range. Note that before deciding to subdivide a voxel, the algorithm checks whether there is sufficient sensor data available for that voxel's region.

If we choose a narrow range for subdivision, such as .3 - .7, we will most likely not properly map small areas of occupancy in larger areas of freedom or small areas of freedom in large areas of occupancy as the small areas will not be enough to bring the voxel's confidence close enough to center to qualify for subdivision. However, a thin range reduces the number of subdivisions thus reducing the computational load and making mapping more efficient. The opposite of these two properties, greater sensitivity to sensor discrepancies and more computationally intensive, is true for wider ranges such as .1 - .9. Choosing the appropriate range involves balancing the need for a detailed representation of the environment against computational efficiency.

As we gain more information about voxels, we can split them accordingly. However, if we subdivide a voxel into sub-voxels, but, after gathering more information for the sub voxels, realize that either all sub-voxels are free or all sub-voxels are occupied, then the larger initial voxel was most likely either fully free or fully occupied. For this reason, if either all subdivisions of a voxel are above the subdivision range or all subdivisions of a voxel are below the subdivision range, we merge them into a larger voxel and take the mean occupancy confidence as the larger merged voxel occupancy confidence. Merging or subdividing after collecting more

data also ensures that the level of detail of a region changes as the environment changes, thus supporting dynamic environments.

3.5 - Graph-Based Planning Algorithms

3.5.1 - Introduction to Graph Based Path Planning

Let us begin by examining the case of global path planning, deriving optimal algorithms for a variety of scenarios.

The real world is inherently continuous, meaning there are infinitely many potential paths the robot could take. Evaluating all these paths to find an optimal solution is computationally impossible. We could lessen this search by solving complex mathematical optimization problems, but this would entail finding some function of position, velocity, acceleration, etc. of time to translate into direct motor actions. This also requires taking into account obstacles.

These are still computationally expensive and often require solving partial differential equations or performing exhaustive searches in high-dimensional spaces. This is fine if the robot knows about its environment, start pose, and end pose ahead of time and can plan the path well before it needs to fulfill the path. Technically, the robot could do this for all paths, then just figure out which one it should take based on the desired start and end pose. However, this is impossible with an infinite number of start and end poses. Even if there were a finite number of possible start and end poses, storing all of them then using them is often infeasible. Firstly, with any sizable number of start and end poses, the number of possible pairings will be very large. In fact, for n start and end poses, the number of possible combinations is n^2. Additionally, the feasible poses may change over time due to obstacle motion. Lastly, due to changes in terrain, regulations, etc. the cost of different path sections may change over time.

Additionally, robotic systems often operate in real-time and need rapid decision-making. This shows that directly finding the optimal continuous action at each point in time is often not feasible. In these situations, a robot must either solve a simpler optimization problem at each point in time, or come up with a pre-computed solution providing the action at each point in time given the pose and potentially other state variables.

Let’s consider the first situation of reducing the complexity of the optimization problem. In many robotics applications, absolute precision of navigation is not extremely important. This means we often do not exactly need the optimal solution, simply a near optimal solution. When a robot is not operating in a very cluttered environment with a small navigational space, or there are no relevant performance metrics which are highly sensitive to precise paths, we do not care about very small spaces or path deviations. In addition, we know that our sensors and actuators are not perfect, so absolute precision of navigation may not be very feasible. This prior reason also means that we want to create a buffer zone between the robot’s path and any obstacles.

In such cases, we do not need to treat the environment as continuous, but can instead assume that nearby actions are redundant. This means that at each state, we can sample a discrete set of actions from the continuous action space that sufficiently covers it. Doing so greatly reduces the action search space, and thus significantly decreases optimization time. For example, consider a robot that can turn at any angle between 0° and 360°—this defines an infinite action space. In contrast, a robot constrained to turn only at fixed intervals such as [0°, 45°, 90°, ..., 315°] has just 8 possible turning actions. Eight is a dramatically smaller search space than infinity, yet these coarse-grained actions may be sufficient for many planning tasks. This machine planning paradigm—where the robot selects from a finite set of actions—is referred to as a discrete action space. Technically, it is possible to discretize even high-precision tasks, but doing so would require an extremely fine resolution, leading to an explosion in the number of possible actions. This would demand excessive computational and memory resources, often without significant benefit over using a continuous action space. Furthermore, a discrete action space allows the robot to explicitly calculate and compare the cost of each possible action, which is infeasible in a continuous action space due to the infinite number of options.

If nearby states/positions are similar or the optimal action in these states are similar, we can group them together to form a **discrete state space**. This further reduces optimization time, provided the number of states is manageable, because we no longer need to account for the infinite range of possible robot positions or the infinite outcomes that could result from an action. This also eliminates the need to deal with computationally intensive continuous functions such as integration, and differential equations.

Another situation where we can group poses together is when a robot committing to some pose necessitates that it will go through some other pose. This is usually found in sequences of poses where being in one pose leaves only the option of going to the next pose in the path (e.g., roads, hallways, aisles). In these situations, we can reduce the search space by grouping these sequences of mutually necessitating poses into a single path. Instead of deciding to go through each pose and where to go next, we merely have to decide when and whether to take this path. This makes path planning more efficient by ignoring redundant information and decisions. In an office building, a robot moving down a long hallway can be abstracted as transitioning from "start of hallway" to "end of hallway," instead of tracking every possible position along the way. A robot navigating a warehouse with shelves doesn’t need to consider every centimeter of movement along an aisle—it just needs to decide which aisle to enter and when to turn.

When the state space is discrete, the action space is usually discrete as well. This is because similar actions taken in the same state tend to lead to similar results, which we often group into the same or nearby discrete states. In other words, the granularity of the state space influences how finely we need to distinguish between actions—if similar actions produce

indistinguishable outcomes at the resolution of the state space, then treating those actions as equivalent becomes reasonable.

As discussed in Section 3.2 on machine planning, a robot often needs to consider the probability distribution over possible next states in order to maximize expected reward. However, global path planning operates under a different set of assumptions: it typically assumes a known and static environment, where the outcome of each action is deterministic. In robotics, this simplification is often valid—especially when using discrete state spaces—because robot controllers generally execute kinematic commands with high precision. For example, if the robot is instructed to move 6 inches forward, the controller will usually execute the command accurately enough that the resulting state aligns with the intended discrete representation. This reliability allows us to treat deterministic actions—such as position or velocity commands—as effectively guaranteeing a transition to the expected next state. At the level of abstraction defined by the discrete state space, the system can thus be treated as deterministic. Since the reward function is typically defined based on the resulting state, this greatly simplifies action evaluation. Rather than computing the expected value over a range of possible outcomes, we can directly assess the reward associated with each state transition.

This means that each action corresponds to choosing a neighboring state to move to. As a result, the path planner searches for the optimal sequence of discrete states, from which a continuous trajectory can later be derived. In path planning, each state typically encodes both the robot's relevant pose—and possibly its kinematic variables—as well as an environmental representation of nearby obstacles. To make the problem computationally manageable, we group all poses within a small region into a single discrete state. For these states to be useful in planning, each one must be reachable, meaning there must exist a feasible path from the start to that state, and from that state to the goal. In this context, the task of path planning reduces to identifying a sequence of intermediary discrete poses, commonly referred to as **waypoints**, that connect the start and goal through a valid and efficient path. Each discrete pose in this sequence serves as a waypoint.

To determine the sequence of waypoints needed to reach a target, we must define each waypoint in terms of the adjacent waypoints that can be reached directly from it. In environments where movement is constrained—such as along roads, hallways, or corridors—waypoints are typically placed at junctions (e.g., intersections or corners), and the paths between them are predefined and obvious.

In such settings, connectivity is transitive through adjacent waypoints. Consider a case where A is adjacent to B, B is adjacent to C, and A is not adjacent to C. If the robot must travel from junction A to junction C, it will naturally pass through the intermediate junction B if A and C are not directly connected. Therefore, it is unnecessary—and even counterproductive—to store

a direct connection from A to C when a path through B already exists. Including such redundant edges increases computational overhead without improving the planner's capability.

The same logic applies in open environments—such as open fields, large rooms, or other spacious areas—where there are no predefined paths. Even in these cases, a robot cannot instantly move from one distant location to another; it must physically traverse space continuously. This means that moving between distant waypoints still requires passing through a series of intermediate, nearby waypoints. These intermediate steps reflect the robot's physical limitations: it cannot teleport and must follow a continuous path governed by its motion capabilities. As a result, each waypoint only needs to be connected to its adjacent neighbors—those that the robot can reach directly, given its motion model and spatial resolution. These local connections are sufficient for building longer paths, since all feasible long-range movements must still be composed of a sequence of short, physically possible steps.

In order to create a path through waypoints, the robot must have a set of stored waypoints, each with a numerical value for each dimension for each Cartesian degree of freedom. As discussed above, for a mobile ground robot, this will be the x and y coordinates and potentially the z orientation. The waypoints should only be generated in the subset of the configuration space that is relevant for the robot. This area is known as the **workspace** and usually consists of 2D or 3D positional ranges, with potential orientational ranges.

Our navigational structure will entail a series of waypoints and paths between adjacent waypoints. Note that whether these paths are unidirectional or bidirectional depends on the application. To represent this pictorially, conceptually, and programmatically, we usually treat these structures as the analogous computer science data structure known as a **graph**. In computer science, graphs consist of relevant points or locations called **nodes**, and connections between nodes called **edges**. In our situation, waypoints are nodes and paths are edges. A pictorial representation of a graph is shown below:

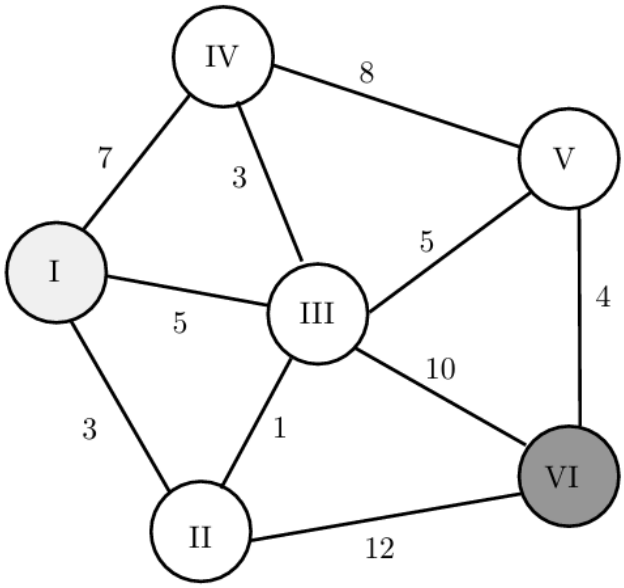


Fig. 3.3: Undirected weighted graph with six nodes labeled I through VI and edge weights representing traversal costs. Node I is shaded light gray (start), and node VI is shaded dark gray (goal). [MIT Center for Bits and Atoms (CBA)]

Planning the optimal sequence of nodes on a graph is known as **graph-based path planning**. A map of an environment via a graph is known as a **topological map**. The cost between waypoints is known as the edge cost, and represents the immediate cost in terms of distance, time, energy, and/or collision potential of an edge. This is r_t in the formalized path/machine planning equations. Remember that r_t can represent cost or reward depending on our feelings about the quantities it represents.

This refines graph based path planning to utilizing some graph (G = (V, E)) with a series of nodes (V) and edges (E), then, for some start node and end node, to figure out the sequence of edges which minimizes the sum of edge costs. Let's reconsider the graph displayed above, with 6 nodes, each having a series of displayed edge costs. Note that in this graph, the cost of traveling along an individual edge is independent of the direction of travel (starting and ending node). This graph is known as having **symmetric edge costs**. In this situation, we do not have to consider direction or separately store the cost of the same edge but with different traversal directions.

The graph depicted above represents a very simple graph based path planning problem. Say we want to travel from I to VI with the lowest total cost. The optimal path is to go from I-II-III-V-VI, which seems indirect, yet it has the lowest cost.

Overall, in a given environment, the robot first builds the graph during the construction (or learning) phase, and then computes a path between the start and end node during the search (or query) phase.

3.5.2 - Single vs. Multi-Query Path Planning

For any given path planning scenario, there exists a theoretically optimal set of waypoints that minimizes the cost of the path. However, this optimal set is not known in advance—it must be discovered through the planning process. To enable this, we begin by generating a set of candidate waypoints that sufficiently cover the relevant space. These waypoints must strike a balance between resolution (to capture necessary detail) and efficiency (to minimize memory usage and computational overhead). Resolution refers to the level of detail or granularity with which the environment or the action space is represented during planning. In path planning, this often means how closely spaced the waypoints, grid cells, or sampled points are. Fine resolution means small spacing between waypoints, more detail, and finer control over the path. Coarse resolution means larger gaps between waypoints, less detail, and more coarse movements.

In situations where the robot only needs to navigate the environment once—such as in search-and-rescue missions—the planning graph is typically constructed after the start and goal waypoints are already known and is used for just a single query. This means it's only used for one specific request to find a path between a given start configuration and goal configuration within a robot's configuration space. This approach is also appropriate when consecutive planning queries involve similar start and goal locations, or when the environment changes frequently enough that previously computed waypoints and their connections become invalid. In such cases, the roadmap must be rebuilt or updated before each new query. Since these situations focus on planning a path between one specific pair of points, the waypoints and edges are generated with that pair in mind. This planning paradigm is known as **single-query path planning**.

However, the robot often encounters situations where it is navigating in an environment that is largely static with respect to consecutive path queries. It also often deals with scenarios where it will have to plan multiple paths with significantly different start and end points in said environment that it does not know the start and end point of before generating the first path. In these situations, the set of all waypoints that are optimal for at least one future path is likely vast and varying, comprehensively covering the relevant area of the C-space.

In these situations, some waypoints that may be used in one path may be used in another, meaning that certain waypoints can be reused. It is inefficient to eliminate and regenerate the same waypoint and waypoints connection for multiple paths. However, it is also inefficient to consider a waypoint or waypoint connection that is not part of the optimal path given a start and end waypoint.

In many path planning scenarios, especially in structured or constrained environments, there tends to be significant overlap between the waypoints and connections used in different planned paths—even when the robot's start and goal configurations are not the same. In other

words, even though the robot may need to travel between different pairs of locations, the paths it takes often share many of the same intermediate steps.

This is common when certain areas or corridors of free space—such as hallways, intersections, or open regions—are commonly traversed regardless of the exact origin or destination. These regions act as "shared highways" or bottlenecks in the environment. For example, a service robot in a building might travel from Room A to Room B, or from Room C to Room D, but both paths may pass through the same central hallway or doorway.

Such overlap between waypoints and connections is especially common in situations where there are strict path optimality requirements. When optimality is a key concern, it is not enough to consider only the most direct or shortest-looking routes. Instead, the planner must also evaluate indirect or initially unintuitive paths, because they may ultimately prove to be lower-cost once all real-world constraints are taken into account. These constraints can include static obstacles (like walls or furniture), dynamic changes (such as moving people or robots), terrain types, speed limits, or areas that consume more energy to traverse. As a result, a path that appears inefficient at first—such as one that takes a longer geometric route—may actually be more optimal overall if it avoids high-cost areas, navigates around congested zones, or follows smoother terrain. For example, a delivery robot in a busy environment may take a longer detour around a central hallway to avoid heavy foot traffic. Even though the detour increases the distance, it may reduce the time, risk, or energy consumption, making it a better path under the defined cost function.

Planning for optimality is typically more valuable than quickly computing a feasible (but suboptimal) path when the difference in cost between the best and merely acceptable paths is large enough to justify the extra planning time. However, if this cost difference is relatively small—or if real-time responsiveness is more important than path quality—it may be preferable to use a fast, suboptimal path. For example, if a service robot in a restaurant needs to move from one side of the dining area to the other, it might make sense to immediately begin following a slightly longer but readily available path rather than waiting for a perfect plan to be computed. If the robot spends too much time planning, the total time to reach the goal may be longer than if it had started moving right away.

Overlap in useful waypoints and connections is also more likely when the robot's configuration space has relatively few dimensions—for example, in mobile robots whose configurations are often limited to (x,y,θ). In low-dimensional spaces, fewer waypoints are needed to adequately cover the space, which means that each waypoint has a higher probability of being useful in multiple different queries. Similarly, when the environment or task allows for lower-resolution planning—that is, when the spacing between waypoints can be coarse without sacrificing task performance—the total number of required waypoints decreases, and each waypoint is more likely to be reused across multiple plans.

In situations where a robot may be asked to navigate between many different pairs of start and goal configurations, it becomes useful to construct a comprehensive roadmap that covers a large portion of the relevant configuration space. In such a roadmap, nearly any waypoint might end up being used in at least one of the many paths the robot needs to plan. In other words, although not every waypoint will be relevant to every query, most of the waypoints are likely to be useful across the full set of navigation tasks the robot may encounter.

Even if only a small number of these waypoints turn out to be unnecessary for a given individual query, the cost of storing and processing those extra waypoints is often justified—especially when compared to the cost of rebuilding a new set of waypoints and checking their connections for validity every time a new path is needed, as is done in single-query roadmaps. This includes the cost of performing collision checks, re-establishing connections between nearby waypoints, and running the planning algorithm from scratch.

In these cases—assuming the robot has sufficient time, memory, and computational resources—it is generally more efficient to precompute a dense set of waypoints and their valid connections and then reuse this roadmap for multiple planning queries. This strategy is known as **multi-query path planning**. It is especially well-suited for environments that are static or change infrequently, and for tasks where the robot is expected to receive many different navigation goals over time.

In contrast to multi-query roadmaps, there are many situations where it is more efficient to use a single-query roadmap, in which a new graph is built from scratch for each individual planning task. This approach is often preferred when the planning scenario presents one or more challenges that make precomputing and storing a large, reusable roadmap impractical or inefficient.

One common reason for preferring the single-query approach is that the task requires a high-resolution set of waypoints. A high-resolution roadmap means that waypoints must be placed very densely in the robot's configuration space in order to accurately capture fine-grained movement and navigate around small or intricate obstacles. This is often necessary in environments with tight clearances, narrow passages, or precise motion constraints—for example, a surgical robot navigating around delicate tissues or a manipulator assembling small components. In such cases, generating a full-coverage roadmap with this level of detail would result in an enormous number of waypoints and connections, leading to excessive memory usage and long preprocessing times. This makes precomputing the entire graph inefficient or even infeasible.

Another reason is that path optimality is not always critical. In many real-world applications, the goal is simply to find a feasible path quickly, even if it's not the most efficient in terms of length, energy, or time. For example, in time-sensitive or dynamic situations—such as search-and-rescue, collision avoidance, or reactive navigation—the robot must begin moving

almost immediately. In these cases, responsiveness is more important than computing the perfect path. Therefore, planners prioritize speed over path quality and avoid the overhead of using or maintaining a precomputed roadmap.

Finally, the dimensionality of the configuration space can strongly influence the choice of planning strategy. When a robot operates in a high-dimensional configuration space—such as a drone with six degrees of freedom (position and orientation in 3D) or a robotic arm with many joints—the number of waypoints needed to adequately cover the C-space is excessively large so maintaining a dense set of waypoints becomes computationally prohibitive. For example, doubling the number of dimensions can require orders of magnitude more waypoints to maintain the same level of resolution. In such environments, it is far more efficient to dynamically generate only the subset of waypoints relevant to the specific start and goal configurations, rather than constructing and storing a full roadmap.

In these cases, the overhead of maintaining and evaluating a large roadmap—most of which may be irrelevant to the current task—can exceed the cost of building a minimal, task-specific roadmap on demand. Therefore, a single-query strategy is often the better choice in such scenarios.

3.5.3 - Cell Decomposition

We will begin by focusing on the multi-query roadmap scenario, where reusing a shared set of precomputed waypoints across multiple tasks offers significant efficiency advantages.

In multi-query path planning, the robot must be able to plan paths between many different start and goal configurations, rather than a single fixed pair. To support this flexibility, the underlying roadmap or graph must allow multiple paths both to and from each node. This means that nodes should not have just a single incoming or outgoing edge—as in a tree structure—but instead must be connected to all neighboring nodes in free configuration space (C_{free}). This ensures that the graph accurately captures the adjacency relationships within the space.

Without these multiple bidirectional connections, certain pairs of configurations may remain disconnected, making it impossible to find valid paths between them—even when a feasible path exists geometrically. A connected graph guarantees that a single shortest-path search algorithm can be executed from any start node to any goal node without needing to rebuild or re-root the structure for each new query. To achieve this, the graph must be designed to capture the connectivity of C_{free} comprehensively, which requires nodes to maintain full adjacency to all other reachable configurations within their local neighborhood.

In environments with large open space where path optimality is a high priority, it is important to comprehensively cover the robot's C-space with well-distributed and well-connected waypoints. This ensures that the planner can find not just feasible paths, but high-quality ones. Additionally, when planning needs to support many different start and goal

pairs—as in multi-query scenarios—the waypoints should be laid out in a way that enables efficient reuse across a wide range of queries.

To achieve this, we require that waypoints are neither too sparse, which could cause important regions of C_{free} to be missed, nor too dense, which would waste memory and computational resources by placing redundant samples. This motivates the use of a deterministic sampling strategy, where the configuration space is uniformly discretized along each axis, and waypoints are placed at regular intervals. Sparse or irregular coverage might cause the planner to miss narrow passages, avoid efficient detours, or fail to capture smooth or low-cost trajectories. Deterministic sampling guarantees even spacing, ensuring that no region is overlooked and that optimal or near-optimal paths are discoverable. A deterministic sampling grid allows any new query to be answered without worrying about missing regions or randomness-induced gaps.

In practice, this means dividing the C-space into a grid and placing waypoints at the center of each grid cell, keeping only those that correspond to collision-free configurations. When focusing only on positional dimensions (such as x, y, z), this is equivalent to placing one waypoint at the center of each free voxel in an occupancy grid map (OGM). For higher-dimensional spaces that include orientation or joint angles, we extend this approach by placing uniformly spaced samples across all relevant degrees of freedom, again keeping only those in C_{free}.

This overall method of waypoint generation and space coverage is known as cell decomposition. It involves partitioning the configuration space into discrete, manageable regions (typically cells in a grid), identifying which cells lie entirely or partially in C_{free}, and then representing those regions with one or more internal waypoints. Cell decomposition provides complete coverage of the free space at a chosen resolution, making it particularly effective for optimal and multi-query planning tasks in relatively static environments.

Each cell represents a manageable portion of the space and is evaluated to determine whether it would cause an internal or external collision or not. Cells that are in C_{free} are retained, and one or more representative waypoints are placed inside them to model the traversable space. These waypoints can then be connected to their neighbors to form a roadmap, which is used for path planning.

This method provides complete coverage of C_{free} at a user-defined resolution. The resolution determines the size of each cell and, by extension, how precisely the planner can capture fine details like narrow corridors or tight clearances.

This method is known as **cell decomposition**. Since it systematically represents the full space, cell decomposition is particularly well-suited for scenarios that demand optimal paths, or where many different planning queries will be issued in a relatively static environment (where obstacles and free space do not change frequently). Once the roadmap is constructed, it can be reused to efficiently plan paths between many different pairs of configurations.

A pictorial representation of this for 2D position is shown below:

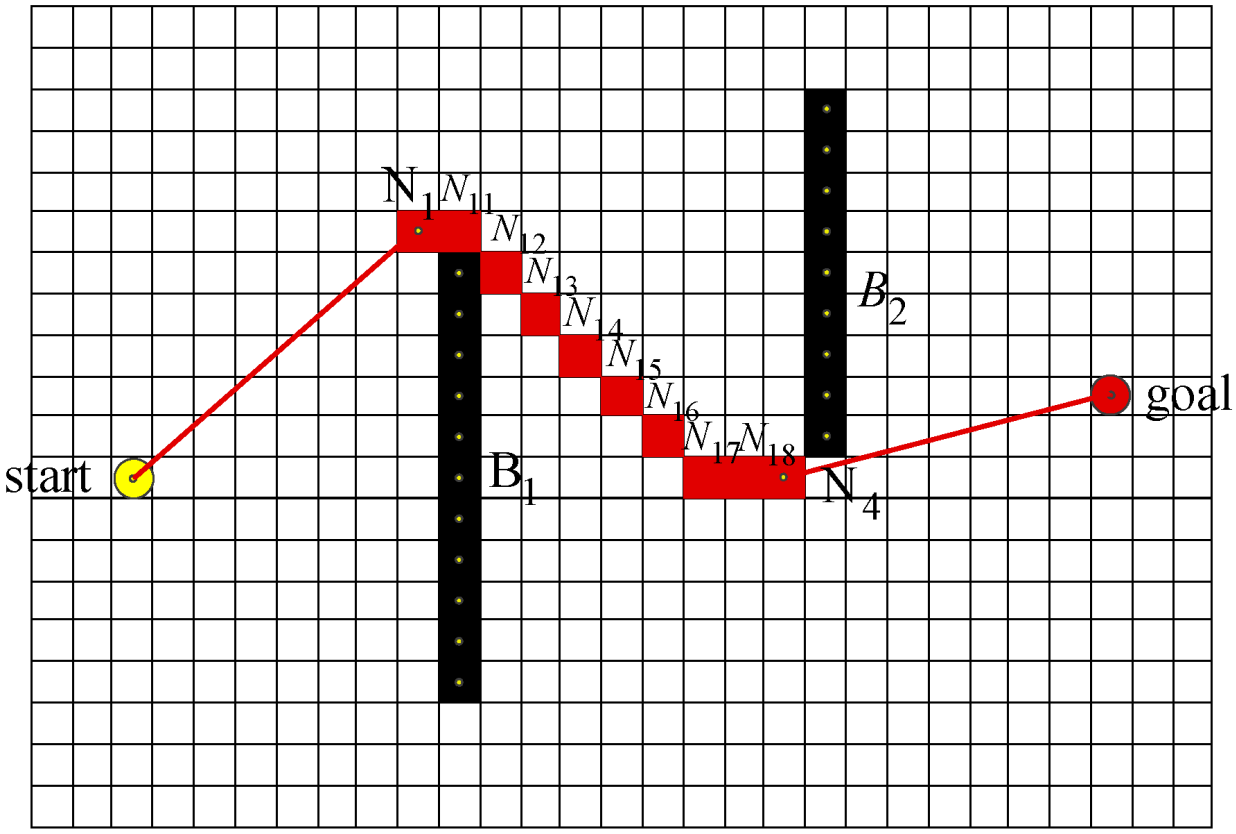


Fig. 3.4: Grid-based cell decomposition environment illustrating a path planning algorithm navigating from a start location (yellow) to a goal (red). The obstacles B_1 and B_2 are shown in black, while the valid path traverses a series of intermediate nodes N_1 through N_4, including expanded nodes N_{11} to N_{18} in red. The figure demonstrates how the planner avoids obstacles by generating a sequence of locally valid moves across the grid. [Wang & Sun (2023)]

3.5.4 - Probabilistic Roadmaps (PRM)

However, we often deal with higher dimensional spaces, such as a drone operating in 3 positional dimensions with potentially multiple orientational dimensions, or a robot arm with a middle or upper single digit DoF. If we want n cells per dimension with DoF = m, that requires worst-case n^m cells. If we want 20 cells per dimension with DoF = 6, we potentially have 20^6 = 64,000,000 voxels. Additionally, we often have to deal with larger environments or hardware with significant memory and processing limitations. In these situations, it may not be feasible to store all relevant waypoints. Note that the dimensions and resolution of an occupancy grid depends on the desired spatial dimensions and resolution of collision checking which may differ from that of planning. Also, we generally value being able to take a path relatively soon after knowing the start and end pose in real-time or time-critical applications such as manufacturing or search and rescue.

In time-sensitive situations, achieving an optimal or near-optimal path often requires high-resolution sampling, which comes with significant processing and memory costs. However, in many applications, path feasibility is more important than optimality. Moreover, the time spent searching for an optimal path can outweigh its execution benefits. For instance, if one planner takes 1 minute to find a 50-second path, while another takes only 30 seconds to find a 1-minute path, the latter solution results in reaching the goal 20 seconds earlier despite the longer execution time.

In the above scenarios, to alleviate these concerns with cell decomposition, we must significantly decrease the resolution of waypoints. This would entail grouping voxels together when planning the voxel traversal. This works great in open-environments where we can make long, straight connections that cover a lot of space. If we used higher resolution, then a path from one point in the open space to another may involve multiple nodes and edges. This is unnecessary when a single or few edges could provide coverage of the area, despite whether there are potentially better paths.

However, many real-world path planning scenarios involve narrow passages or thin obstacles, which present significant challenges for grid-based cell decomposition methods. If the grid cells are too large, a narrow passage may be entirely contained within a single cell that is marked as “occupied” or “uncertain”—even if free space actually passes through it. As a result, the planner may incorrectly treat the passage as blocked, simply because the coarse resolution fails to capture the fine detail.

A key limitation of traditional cell decomposition is that it uses a uniform resolution across the entire map. This means the grid cannot adaptively increase resolution in areas where greater detail is needed—such as tight corridors, doorways, or gaps between obstacles. In contrast, narrow passages often require a finer resolution than the open areas of the environment, but the uniform grid lacks the flexibility to represent this efficiently.

Additionally, thin obstacles—such as wires, poles, or thin walls—may only partially intersect a grid cell without fully occupying it. However, because of how occupancy grids discretize space, even a small intersection may cause the entire cell to be marked as “occupied.” This leads to inflation artifacts, where thin obstacles appear wider than they actually are, introducing false blockages that can prevent valid paths from being found.

To accurately handle these situations, we would need a high-resolution occupancy grid capable of detecting and representing small features. However, using such a high resolution uniformly across the entire map would be computationally expensive and memory-intensive, making it infeasible for most real-time planning applications.

For these reasons, we often take a variable size cell decomposition approach where we subdivide cells if they are partially occupied. Although variable‑resolution cell decomposition mitigates some costs, it still suffers from a few fundamental issues.

- **Implementation complexity.** Managing cell hierarchies requires data structures for splits and merges, logic to decide which mixed cells to refine next, and careful handling of boundary-case geometry in high dimensions.
- **Blind subdivision.** Resolution is focused around obstacles, but the algorithm cannot distinguish dead-end pockets from true passages; it may waste effort refining mixed cells that may never contribute to any viable path.
- **Axis-alignment artifacts.** Cell decomposition used axis-aligned cells, meaning that each cell has edges that run exactly parallel to the coordinate axes. Using axis-aligned cells forces slanted or curved corridors to zig-zag through many narrow cells, inflating cell counts and introducing "staircase" paths.
- **Rigid adjacency constraints.** Even when a direct, collision-free straight line exists between two points, movement must proceed via a chain of face-adjacent cells, incurring unnecessary intermediate steps and longer path extraction. One example of this is a long, but narrow hallway where cell decomposition would break it into many cells even though points on the end can be connected by a straight line.
- **You must commit to a single cell size before subdivision begins**—if it's too coarse, critical passages remain hidden; if it's too fine, subdivision becomes exponentially expensive—and you cannot adjust this resolution incrementally as planning time allows.
- **You cannot adapt resolution mid-run.** If you later discover that your chosen refinements still miss a narrow passage or over-refine an easy region, you must restart the entire decomposition at a new resolution or make global adjustments—there's no way to incrementally adjust only the problematic cells based on observed connectivity gaps.

  These issues point to a solution with the following attributes:
- **Flat data structures.** Flat data structures refer to representations in which the planning environment is modeled using only a set of discrete points (also called landmarks) and the pairwise connections (edges) between them. Unlike hierarchical structures such as quadtrees or octrees, which recursively divide space into nested cells, flat structures avoid the need for spatial trees, parent-child relationships, or split/merge bookkeeping. This simplifies memory management and improves computational efficiency, particularly in high-dimensional or dynamic environments. In a flat structure, the planner's state must be representable as a simple list of independent entities—nodes and edges—without relying on any embedded spatial relationships. Since flat representations do not support implicit spatial connectivity (e.g., adjacency through shared grid boundaries), the only way to capture the connectivity of free space is to explicitly define which pairs of points are connected. Each edge represents a feasible, collision-free path between two sampled

configurations. Any attempt to represent the environment using more complex geometric objects—such as spatial cells or volumetric regions—would require reintroducing hierarchical organization, in order to manage how these elements are nested, subdivided, or merged. Thus, to preserve flatness, planners must operate entirely on discrete samples and their direct connections.

- **Orientation‑agnostic connectivity.** Place points anywhere in the configuration space and connect them regardless of axis alignment—avoiding the "staircase" effect of axis-aligned cells. The only geometric primitive that is orientation-agnostic is the straight-line segment between two arbitrary points. Any grid or cell face is inherently axis-biased.
- **Direct, long-range edges.** Allow any two samples within a collision-free straight-line neighborhood to be connected in one edge, rather than forcing face-to-face adjacency. The planner must allow non-local graph edges—i.e. edges between any two nodes that satisfy some collision-free criterion—rather than only immediate neighbors. That criterion naturally takes the form "within some distance" or "k-nearest."
- **Connectivity-driven sampling.** Only retain points for path planning that lie in free space and can link into the existing roadmap—no effort spent on dead ends or obstacle interiors. You don't just need points in free space; you need points that help link disconnected portions of your roadmap. Each newly accepted sample must undergo a local-connection test against nearby nodes. If it doesn't connect to the other nodes in the graph at all, it is useless for your multi-query roadmap and can be discarded. This ensures every retained node has at least one contributing edge.
- **Anytime roadmap growth.** Stop adding points once the roadmap meets connectivity requirements; resume sampling only if further refinement is needed. The planner's architecture must be open-ended: adding more elements must never invalidate prior work, and graph queries must always run on the current structure. This rules out any method that requires a full rebuild to change its resolution. This points to an iterative point generation loop ("sample → connect → check").

This suggests a path planning strategy that iteratively samples configurations in the robot's configuration space (C-space), checks each sample for validity (e.g., whether it lies in collision-free space), and connects it to nearby existing samples using straight-line edges, as long as those connections are also collision-free.

To remain computationally tractable in high-dimensional spaces—such as those involving many joints, degrees of freedom, or complex constraints—the sampling mechanism must be extremely simple and fast. Ideally, generating a sample should involve no more than a single function call, such as drawSample(), which returns a random configuration. This simplicity is essential because any additional logic—for example, tracking where samples have

already been placed, or maintaining structured relationships between them—becomes increasingly expensive as the dimensionality of the space grows.

Deterministic or structured sampling approaches, such as Halton sequences, lattice sampling, or adaptive grids, attempt to systematically cover the space. However, these methods tend to either:

- Reintroduce bias, favoring some regions over others, thereby reducing the chance of discovering paths in less-sampled regions, or
- Require complex bookkeeping, such as tracking coverage, indexing multidimensional grids, or maintaining recursive subdivisions—all of which violate the requirement for a flat data structure, where the planner stores only points and edges, with no hierarchy or spatial indexing.

In contrast, pure random sampling is both trivially simple to implement and inherently dimension-agnostic. It does not rely on any underlying spatial structure or ordering, making it well-suited to high-dimensional C-spaces, including those with dozens of degrees of freedom. Random sampling allows the planner to focus on local connectivity without maintaining global structure.

Perhaps most importantly, random sampling provides a strong theoretical guarantee known as probabilistic completeness. This means that if a path exists between the start and goal configurations, the probability that the planner finds it approaches one as the number of random samples increases. In other words, given enough samples, a randomly constructed roadmap will almost certainly discover a solution if one exists. Deterministic sampling methods, on the other hand, do not offer this guarantee unless they fully decompose the space, which generally requires an exponential number of samples as the number of dimensions increases—making them infeasible in high-dimensional settings. This type of path planning method that involves random sampling of points is known as a **sampling based** method.

The overall algorithm derived above is described below:

- **Construction (Learning) Phase**
    1. **Sampling:** Randomly draw configurations q in the robot's configuration space C.
    2. **Collision Checking:** Discard samples that lie in collision; retain only those in free space C_{free}.
    3. **Connection:** For each retained sample, identify nearby roadmap nodes (within a fixed radius or the k-nearest in configuration space). Attempt to connect via a simple local planner (straight-line interpolation), adding only collision-free edges.
    4. **Graph Growth:** Repeat sampling and connection until the roadmap sufficiently covers the accessible portions of C_{free}, or a preset sample budget is reached.
- **Query (Search) Phase**

1. **Anchor Start/Goal:** If not already in the roadmap, connect the query's start and goal configurations to the nearest roadmap nodes via local planning checks.
2. **Graph Search:** Use a shortest-path algorithm (e.g. A* or Dijkstra) on the roadmap to find a sequence of nodes bridging start to goal.
3. **Path Extraction:** Concatenate the local paths along that sequence to form a continuous collision-free trajectory.

This specific method of graph construction is known as **Probabilistic Roadmap (PRM)** (Kavraki et al., 1996). An image of a PRM graph is below:

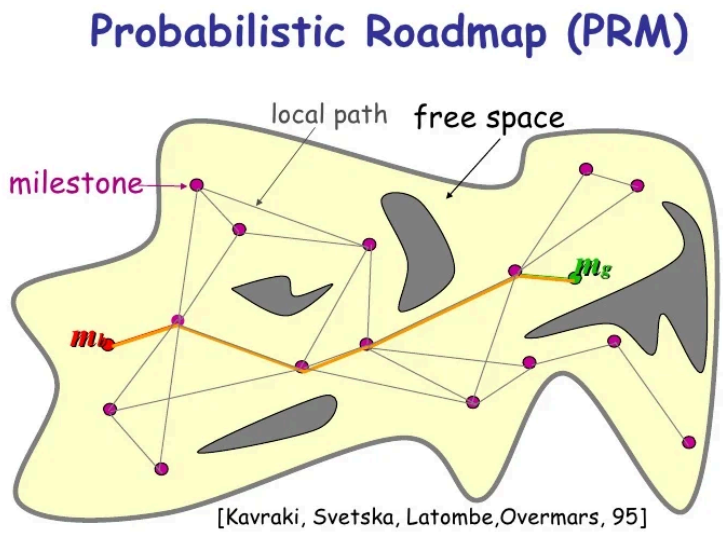


Fig. 3.5: Illustration of the Probabilistic Roadmap (PRM) algorithm. The free space is sampled to generate milestones (purple dots), which are connected via local paths to form a roadmap. A path from the start milestone m_s to the goal milestone m_g is constructed using connected components in the graph. Obstacles are shown in gray. [Adapted from Kavraki, Švestka, Latombe, and Overmars (1995)].

Note that many forms of PRM include sampling biases to favor areas near the goal, obstacles, and/or narrow passages.

3.5.5 - Dijkstra's Algorithm

In some path planning scenarios, the graph representing the environment has uniform or unweighted edge costs—that is, the cost of moving between any two connected nodes is the same. This can occur in settings such as occupancy grids where the robot is only allowed to move to adjacent, non-diagonal voxels, and the only metric of interest is the number of steps taken or distance traveled along the grid.

In these situations, the quality of a path—typically its length—is determined entirely by the number of intermediary nodes, since each edge contributes an equal cost. Under these

conditions, a simple algorithm like Breadth-First Search (BFS) is both correct and efficient, as it guarantees finding the shortest path in terms of edge count.

However, most real-world environments involve non-uniform edge costs, where transitions between nodes may vary in cost due to factors like distance, energy consumption, terrain difficulty, or risk. For example, diagonal motion might be allowed and cost more than straight motion, or some regions might be penalized for safety reasons. In these cases, BFS is no longer sufficient, because it assumes all edge costs are equal and does not account for variations in path cost.

To handle such scenarios, cost-aware graph search algorithms are required. These algorithms evaluate paths based on cumulative cost rather than step count, ensuring that the lowest-cost path is found even when edges differ in weight.

We know that the optimal path from the start node (Node S) to the end node (Node E) will consist of a series of intermediary nodes. However, we do not initially know which specific nodes will be along this optimal path. Theoretically, the path could traverse any of the edges connected to these intermediary nodes. Therefore, we must systematically explore paths starting from Node S to determine the optimal path to each other node. This process continues until we identify the optimal path to Node E. The set of nodes for which we have determined the optimal paths is called ‘Set O’. Initially, Set O only contains the starting node because we know the optimal path to it, with the associated cost being 0. The set of nodes to which we have not determined whether the best known path to that node is optimal is called ‘Set V’. This initially contains all nodes except the start node.

In graph-based path planning, the optimal path from Node S to a Node E is composed of a sequence of optimal sub-paths. This follows the principle of optimality, which states that if a subpath lies along the overall optimal path, then that subpath must itself be optimal. In other words, for any node along the optimal path from Node S to Node E, the path from Node S to that intermediate node must be the lowest-cost path available from Node S to that node.

For example, consider the optimal path from Node I to Node VI. This path includes the optimal path from Node I to Node V, which itself includes the optimal path from Node I to Node III, and so on. Ultimately, the entire path is built by stacking together optimal subpaths, starting with the optimal path from Node I to Node II. Each step builds upon the previous one, ensuring that the entire path from Node I to Node VI is the lowest-cost route through the graph.

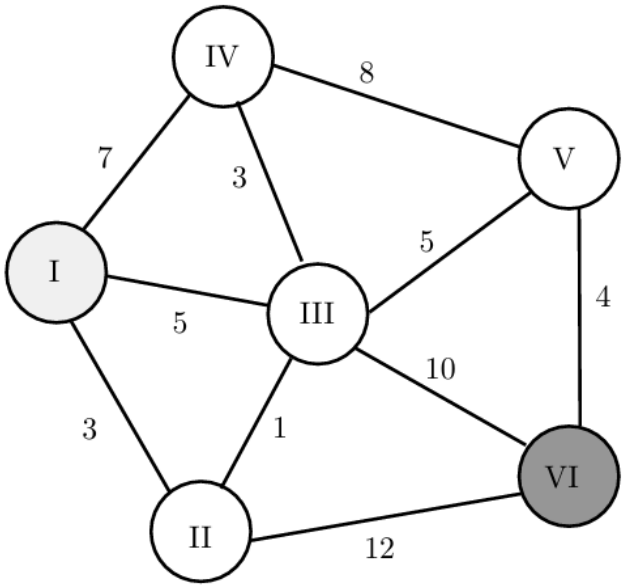


Fig. 3.3: Undirected weighted graph with six nodes labeled I through VI and edge weights representing traversal costs. Node I is shaded light gray (start), and node VI is shaded dark gray (goal). [MIT Center for Bits and Atoms (CBA)]

This means that to find the optimal path from Node S to Node E, we must first find the optimal path to the neighbor of Node E through which the optimal path travels. For example, to find the optimal path from node I to node VI, we must first determine the optimal path from node I to node II, then from node I to node III, followed by the path from node I to node V, and finally to node VI.

This means that any node in Set V that we eventually discover to have an optimal path must be a neighbor of some node already in Set O. In other words, the search for the next optimal node proceeds by exploring the neighbors of nodes in Set O.

For each such neighbor, we check whether the path to it through the current node in Set O improves upon the best known path to that neighbor. If the newly computed cost is lower than the previously recorded cost, we update the best known path and cost for that neighbor. This process ensures that we always expand the frontier in a way that maintains the optimality of Set O.

The optimal path to a node through one of its neighbors can never be better than the already optimal path to that neighbor plus the cost of the edge connecting them (assuming non-negative edge weights). Once the neighbor's path is finalized, this is the best possible route through it, so further exploration through that node offers no benefit.

This method entails that every node in Set V must be a neighbor of at least one node in Set O. Since Set O contains only nodes with finalized, optimal paths, the only candidate paths to any node in Set V are those that go through one of its neighbors in Set O. Therefore, the best

known path to a node in Set V is computed by evaluating all possible paths that reach it from its adjacent nodes in Set O and selecting the one with the lowest total cost—that is, the optimal cost to the neighboring node in Set O plus the cost of the edge connecting it to the node in Set V.

Any alternative path to a node in Set V must pass through some other node—either in Set O or in Set V. However, if the path goes through another node in Set V, it has not yet been confirmed to have an optimal path. Since all edge weights are positive, adding more edges to a path will always increase its total cost. Therefore, any path that reaches a node in Set V by first going through another node in Set V will necessarily have a higher total cost than a direct path from Set O—unless the initial path to that other node in Set V was already better, which contradicts the assumption that we're working with the current lowest-cost node in Set V.

This implies that the lowest-cost node in Set V must already have the globally optimal path from the start node. No alternative path can reach it at a lower cost, because any such path would either pass through nodes with higher known costs or through nodes whose paths are not yet finalized, both of which would result in a higher total cost due to positive edge weights. Therefore, the lowest-cost path to the node must be one of the existing paths through its neighbors in Set O, whose paths are already confirmed to be optimal. As a result, we can safely move this node from Set V into Set O, finalizing its optimal path. Once it is part of Set O, we can then explore its neighbors, potentially updating and improving the best known paths to other nodes still in Set V.

This means that we can start with Node S in Set O, iteratively explore the neighbors of the most recently added node to Set O, update the best known paths to nodes in Set V, then add the node in Set V with the lowest cost known path to Set O. When we update the best known path to a node, we store the preceding node along the path, thus allowing us to reconstruct the optimal path from each node back to Node S. Eventually, Node E will be added to Set O, and the optimal path from Node S to Node E can be reconstructed (assuming one exists). This algorithm is known as **Dijkstra's Algorithm** (Dijkstra, 1959).

The full algorithm is written below:

- Initialization:
    - Assign a cost (or distance) of 0 to the start node (because it costs nothing to reach itself).
    - Assign a cost of infinity (∞) to all other nodes, because we don't yet know how to reach them.

Create two sets:

- Set O: Nodes for which we have finalized the optimal cost from the start (initially contains only the start node).
- Set V: Nodes to which we have not yet confirmed an optimal path (initially contains all nodes except the start).

- Main Loop – Expand from Optimal Nodes:
    - While there are still nodes in Set V:
        - Select the lowest-cost node in Set V
            - This is the node with the smallest current known cost from the start.
            - Move it from Set V to Set O, finalizing its cost.
            - Note that we skip this step in the first iteration because we already know which node to explore
        - Explore its Neighbors
            - For each neighbor not already in Set O:
                - Calculate the cost of reaching the neighbor via the current node:
                    - cost to neighbor = cost to current node + edge weight
            - If this newly computed cost is less than the currently known cost to the neighbor:
                - Update the cost to the neighbor.
                - Update the path to that neighbor (i.e., remember that it came through the current node).
- Repeat Until All Nodes Are Finalized or Goal is Reached
    - Continue the process until either:
        - You have finalized the cost to all nodes, or
        - You are only interested in the shortest path to a specific goal node, and you've finalized it.
- Consider Dijkstra's algorithm on the following graph:

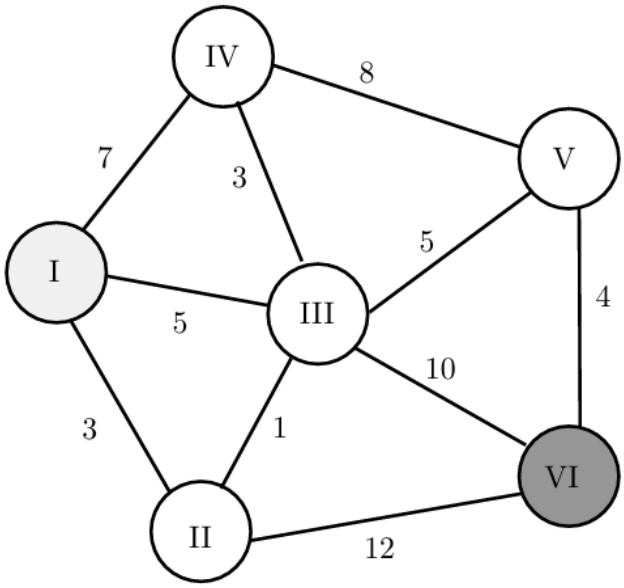


Fig. 3.3: Undirected weighted graph with six nodes labeled I through VI and edge weights representing traversal costs. Node I is shaded light gray (start), and node VI is shaded dark gray (goal). [MIT Center for Bits and Atoms (CBA)]

We begin at Node **I**, assigning it a cost of zero and assigning all other nodes an initial cost of infinity, since we haven't found paths to them yet. From Node I, we examine its immediate neighbors. We find that the path to Node II has a cost of 3, the path to Node III has a cost of 5, and the path to Node IV has a cost of 7. We update the cost of each of these nodes accordingly, marking that each path came from Node I. We then finalize Node I, meaning its shortest path has been found. Next, we move to Node II, which has the lowest temporary cost (3). From II, we can reach Node III for an additional cost of 1, giving a total path cost of 4—lower than the previously known cost of 5 to Node III—so we update III's cost to 4 and mark that the best path to it now goes through II. We also consider the path from II to VI, which has a cost of 12. This gives a total cost of 15 to reach VI through II, which we temporarily record. With II finalized, we move to Node III, which now has the lowest known cost (4). From III, we can reach Node V with a cost of 5, giving a total cost of 9; we update V's best known path and cost. We also find a new path to VI via III with a cost of 10, totaling 14—better than the previous 15—so we update VI's best known path to go through III. Finally, from III, we also check its neighbor IV, but the new path has the same cost as before (7), so no update is needed. We finalize III and move on to Node IV, which has a cost of 7. From IV, the only useful edge is to Node V, but the path through IV would have a total cost of 15, which is worse than the current 9, so no update is made. We then visit Node V, with a cost of 9. From V, we find a path to VI with a cost of 4, for a total of 13. This is better than the previous cost of 14, so we update VI again—now noting that the best path goes through V. Finally, we visit Node VI, which now has a finalized shortest path with a

cost of 13. Since VI is our goal, we stop here. The optimal path from I to VI is I → II → III → V → VI, with a total cost of 13.

3.5.6 - A* Algorithm

Dijkstra's algorithm always selects the node in Set V (the frontier) with the lowest known path cost from the start node, Node S. It then explores that node's neighbors and adds the node to Set O (the set of finalized nodes) only when it is guaranteed that the current path to that node is optimal. While this ensures correctness, it does not take into account how likely a node is to lie along the shortest path to the goal node, Node E. As a result, Dijkstra's algorithm explores nodes purely based on proximity to the start, without regard to whether those nodes are in the general direction of the goal. This can lead to the algorithm exploring many nodes far away from the goal—even on the opposite side of the graph—if they have low path costs. While it is theoretically possible that some of these distant nodes could be part of an optimal path to Node E, such cases are rare. In practice, this lack of goal-directedness increases the time and computational cost of the search, especially in large graphs or high-dimensional spaces.

To address the inefficiencies of Dijkstra's algorithm, we choose to explore the neighbors of nodes that are more likely to be part of this optimal path. We can choose to explore the neighbors of nodes that we are not certain we have found the optimal path to, but we have found a pretty good path to, and think may be part of a very good path from Node S to Node E. The lower the cost is of the optimal path from Node S to Node E through a node, which we will call 'Node N', the more likely that node is to be part of the optimal path. The total cost of this path is the cost of the optimal path from Node S to Node N **(g(N))** plus the cost of the optimal path from Node N to Node E **(h(N)).**

As in Dijkstra's, we use the cost of the best known path from Node S to Node N as g(N). However, we do not know the exact cost of the optimal path from Node N to Node E. To address this, we use a **heuristic function**, h(N), to estimate this value for each node. A reliable predictor of travel cost between two nodes is their proximity. Since each node stores its location, we can use the Euclidean distance, or Manhattan distance depending on the application, between two nodes to estimate the cost of the optimal path between them. Depending on what edge cost represents for the application, h(N) is usually some function of distance used to estimate travel distance, travel time, etc.

By prioritizing nodes with the lowest f(N), we explore nodes that are likely to be part of the optimal path, effectively balancing the actual cost to reach a node and the estimated cost to reach the goal, guiding the search towards the goal more efficiently. To meet our specific application requirements, balancing computational efficiency with path optimality, we multiply h(N) by a tuned weight before adding it to g(N) to get f(N). A low weight means we must have a higher degree of certainty that the path to a node is optimal before adding it to Set O, but

increases the computational power and time. Note that by setting the weight to 0, our search algorithm reduces to Dijkstra's.

$$f(N) = g(N) + w \cdot h(N) \tag{3.14}$$

Since we may explore the neighbors of nodes to which we have not found the optimal path, we explore nodes in Set O for path updates/improvements if they neighbor another node which was added to Set O. If a better path to a node in Set O is found, it is moved out of Set O and reconsidered for having its neighbors explored in subsequent lowest f(N) checks.

Eventually, we add Node E to Set O, signifying that we have found an optimal or near-optimal path from Node S to Node E. Nodes that were in Set V but not Set O represent nodes that we choose to ignore, even though there is sometimes a non-zero possibility that they would lead to an improved path. To determine the actual path, we trace back through the stored prior nodes from Node E to Node S. This algorithm is known as **A*** (Hart et al., 1968).

3.5.7 - D* Algorithm

The previous algorithms assume a static environment where edges and edge costs are constant, but this is not always the case. Certain factors can change edge cost or feasibility during runtime. In robotics, dynamic environments are commonplace, characterized by moving objects, changing weather conditions, fluctuating levels of traffic and human congestion, pathways undergoing construction, etc. Additionally, robots often navigate through partially mapped environments that are updated in real-time. These scenarios require the ability to handle the addition or removal of pathways and fluctuations in path costs during navigation. Such changes can involve adding or removing edges between nodes or altering the costs associated with existing edges. Consequently, these changes can affect the optimal path a robot should take from its starting point to its goal, necessitating continuous adaptation and replanning to maintain efficiency and accuracy.

The path replanning does not need to be employed in all situations. When an edge's cost in the map changes, whether it increases or decreases, we encounter different scenarios that necessitate specific responses. Here's a breakdown of how we should handle each situation:

- **Increase in Cost to an Edge Not on the Current Path:** If the cost of an edge that is not currently part of our planned path increases, it does not affect the optimality of our current path. Therefore, we continue with our current path without the need for replanning. This is because the increased cost does not offer a potentially better alternative route. Note that we often still keep track of the update and alter the graph accordingly at a later time when the robot is not performing heavy computation.

- **Decrease in Cost to an Edge on the Current Path:** If the cost of an edge that is currently on our planned path decreases, we can retain our current path. There's no need to replan the entire path because the decreased cost doesn't suggest a superior alternative path. However, we update the total path cost to reflect the reduced edge cost.
- **Decrease in Cost to an Edge Not on the Current Path:** When the cost of an edge not on our current path decreases, it opens up the possibility that a path through this edge could now be more cost-effective than our current path. In such cases, we evaluate whether replanning the path is necessary to potentially find a new, more optimal route.
- **Increase in Cost to an Edge on the Current Path:** If the cost of an edge that is part of our current path increases, it implies that another path, which previously seemed less optimal, might now be more favorable. This prompts us to reassess our current path and consider replanning to find a new, potentially more efficient route.

If a decreased-cost edge is not on the current path and connects two nodes that have not yet been explored (i.e., neither is in Set O), then the edge was never considered as part of a viable path. This is because any node added to Set O would have triggered exploration of its neighbors—including this edge. For such an edge to have been ignored, the cost of reaching either of its nodes must have been higher than the cost of reaching Node E, meaning the optimal path to Node E was found before these nodes were ever considered. As a result, lowering the cost of this edge cannot possibly yield a better path to Node E than the one already found. Therefore, we only trigger replanning when a decreased-cost edge is not on the current path but connects two nodes for which at least one has already been reached via a viable path. In that case, the new edge could form a lower-cost alternative route to the goal, and replanning may be necessary.

Updating the graph to reflect real-world changes requires some mechanism for obtaining and integrating information about nodes and edges. In some cases, this can be achieved through external sources such as overhead camera systems, environmental sensors, or user input—provided the robot has the ability to receive such updates in real time. However, this is not always feasible. Often, the robot must autonomously sense and evaluate the environment as it navigates, updating edge and node feasibility on the fly based on factors like proximity to obstacles, terrain conditions, or sensor confidence. These dynamic updates may render parts of the current path invalid or suboptimal, requiring the robot to adjust its plan accordingly. If the graph is intended to be reused in the future—such as in multi-query scenarios—data gathered during the robot’s traversal can be incorporated to improve future planning, allowing the roadmap to evolve over time.

To address dynamic changes in environments using A*, one could theoretically replan the entire path based on the robot's current position and the updated map. However, this approach can be time-consuming and computationally intensive, especially if frequent updates are required

along the path to the target. While A* excels in initial path planning scenarios, its efficiency diminishes when confronted with real-time adjustments to maps or edges. Therefore, there is a clear necessity for algorithms better suited to handle dynamic environments in real-time.

The main issue with A* is it doesn't take into account that large segments of the optimal path may stay the same. When replanning our path in dynamic environments, it's important to note that we don't always need to replan the entire route from the start node (Node S) to the goal node (Node E). The extent of replanning depends on how changes in the environment impact our current path. For instance, if the cost of a specific edge in the graph changes, this alteration may only affect the optimal path if the edge is directly part of that path or if it presents a new, potentially more efficient route.

To effectively replan our path, we must pinpoint which nodes are influenced by changes in the graph. A node is deemed affected if there is either a change in its optimal path cost or if it acquires a different predecessor. A different predecessor implies a divergence in path reconstruction, while a change in cost necessitates recalculating the optimal path, potentially altering the lowest known path cost. Consequently, during the replanning process, only nodes meeting these criteria should be reassessed.

As we know, the best known path to a node is determined by the lowest-cost path through any of its neighboring nodes. The cost of each such path is the sum of the cost from the start node (Node S) to the neighbor, plus the cost of the edge connecting the neighbor to the current node. Therefore, if the cost of the optimal path to any neighbor changes, or if the cost of any edge connected to a node is updated, that node's best known path may also be affected. The set of all nodes that are potentially affected by such changes is called **Set M**. Initially, Set M consists of the two nodes connected by the altered edge, since both may now have new, lower-cost paths or become part of a new optimal route.

This implies that if a node's optimal path is affected, its neighbors may also be affected. When examining a node, we update its g(n) value—the cost of the best known path to that node—by checking whether any neighboring node offers a lower-cost path. If the cost is improved, we propagate this change outward by adding the node's neighbors to Set M, continuing the process iteratively. However, if the g(n) value remains unchanged, then even if the node's predecessor is updated, its neighbors are not affected. In that case, we do not add them to Set M, as the cost or optimality of paths that pass through the node has not changed. This prevents unnecessary propagation, and we move on to the next node in Set M.

If a node that was previously assessed neighbors a node with a g(n) that changes, the previously assessed node is re-added to Set M because changes in the cost of n could affect their optimal path or predecessor.

As in A*, we prioritize nodes that are more likely to lie on the optimal or near-optimal path from the start node (Node S) to the goal node (Node E). Nodes that are unlikely to influence

the final path do not need to be reassessed, as their updates are unlikely to improve the solution. Therefore, we conduct a search toward the goal to determine whether its path needs to be updated. This involves selecting the node in Set M with the lowest estimated total cost, f(n). Although the actual cost-to-come g(n) for each node in Set M may have changed due to earlier updates, these changes are not known until the node is explicitly processed. So, we rely on the currently stored values of g(n) and h(n) to approximate f(n) during prioritization.

The process continues until either Node E is reached, indicating that an updated optimal/near-optimal path to Node E has been found, or no nodes remain in Set M, indicating that the current path from Node S to Node E remains optimal. If another change occurs, the sets are cleared and the replanning happens all over again. This algorithm is known as **Dynamic A***, or **D*** for short (Stentz, 1994).

3.5.8 - Rapidly Exploring Random Trees

As discussed in Section 3.5.2, single-query planning is preferable when the resolution of waypoints must be high, path optimality is not the primary concern, or the configuration space has many dimensions. For example, a drone operating in 6D space or a robotic arm with more than six degrees of freedom warrants single-query path planning. In such cases, the cost of considering a large number of irrelevant waypoints outweighs the cost of rebuilding a small, relevant set of waypoints for each new planning query. This makes it more efficient to generate waypoints and connections in real time for each path planning task. Within this single-query paradigm, where the focus is on finding any feasible path (rather than the best one), we aim to minimize the number of nodes and edges in the graph. Ideally, only those nodes and edges that can directly contribute to a path from the start to the goal should be included.

Given an arbitrary node in a graph, we cannot determine whether it contributes to a valid path from the start to the goal unless we have already found such a path that passes through it. In other words, to confirm that a specific node lies on a valid path from the start to the goal, we would first need to solve the path planning problem itself—which defeats the purpose of building the graph in the first place. This realization shifts our objective: instead of trying to selectively add only those nodes that will end up on the final path (which is unknowable in advance), we instead aim to add only those nodes that can be reached from the start node. Any node that cannot be reached from the start is irrelevant to the solution, since a path to the goal must begin at the start and proceed only through connected, reachable nodes.

To enforce this constraint during graph construction, we adopt an incremental, forward-growing strategy. We begin with the start node as the initial graph and grow outward by adding new nodes that can be connected—via collision-free local paths—to one or more nodes that are already part of the existing graph. This ensures that the subgraph remains fully reachable from the start at every stage of construction.

Moreover, to avoid ambiguity and complex dependencies that can arise when multiple new nodes are added simultaneously—especially if some of them could only reach each other but not the start—we add nodes one at a time. This sequential insertion guarantees that each new node's inclusion is meaningful: it extends the set of reachable configurations without introducing unreachable segments. It also allows each step to fully leverage the current structure of the graph, including existing edges and reachability information, making the process more efficient and principled.

There is also another reason for iterative node addition. To prevent any unnecessary nodes from being added, we should be able to stop the algorithm once the first feasible (collision free) path is achieved, with the ability to refine it afterwards. A batch process would only tell you "yes/no" after all nodes are in. This points to an iterative approach to point addition, where we add one node at a time.

In multi-query path planning, we expect to solve many different planning problems over the same environment. This means that the graph must support paths between many possible start and goal pairs. As a result, we often need multiple incoming and outgoing connections for each node to ensure flexible connectivity, even though this increases computational and memory demands.

In contrast, single-query planning focuses on just one specific pair of start and goal configurations. In cases where path feasibility—rather than optimality—is the main objective, we do not need to maintain multiple paths to the same node. Once a feasible path is found, we can terminate the search. This insight leads to a simpler and more efficient graph structure: instead of a general graph with many-to-many connections, we use a tree, where each node has only one predecessor.

Assuming nodes are added iteratively, this tree structure naturally emerges by growing outward from the start node. At each step, we attempt to connect a newly sampled node to a single existing node in the tree. If a valid, collision-free connection exists, we add the new node as a child—guaranteeing that it is reachable from the start. In this way, we incrementally grow a reachability tree rooted at the start node.

Once a newly added node is close enough to the goal and a direct edge between them is valid, we connect it to the goal node and terminate. This approach avoids unnecessary complexity while guaranteeing feasibility, making it well-suited for high-dimensional planning tasks where responsiveness is more important than path optimality.

There are additional reasons to prefer a single predecessor per node in path planning—especially when dealing with kinodynamic constraints. In many cases, we do not explicitly track the robot's full kinematic or dynamic state at each node. This is a major concern for nonholonomic robots or when kinodynamic planning is required—that is, planning that respects both the robot's kinematics and dynamics. If we allow a node to be reached from

multiple predecessors, the robot could arrive at that node in multiple possible kinodynamic states. Without knowing the exact arrival state, we cannot accurately determine which subsequent motions are feasible from that node. This would prevent us from knowing which possible nodes can be reached from it. To avoid this, we enforce that each node has a unique predecessor, meaning there is only one way the robot could have arrived at that node. As a result, the robot's kinodynamic state at that node is uniquely determined. This allows us to locally apply the robot's motion constraints to generate a kinodynamically feasible set of successor nodes without ambiguity or retroactive computation. Thus, maintaining a single predecessor is crucial for propagating consistent and physically valid trajectories through the graph.

As with kinodynamic planning, another reason to enforce a single predecessor per node is to simplify the process of path extraction. In graph-based planning, especially when nodes are added incrementally, we know that each new node is connected to the existing graph and thus reachable from the start. Once the end node is added, this guarantees that a feasible path exists. If we ensure that each node has only one predecessor, we can reconstruct the entire path from the goal back to the start simply by backtracking through the predecessor links—without needing to replan or search. This greatly reduces computational overhead. By contrast, if multiple incoming edges are allowed, we would need to search again to identify which combination of predecessors actually forms a valid path. Enforcing a strict parent-child relationship between nodes avoids this problem entirely and ensures that every node lies on exactly one path from the start, forming a well-structured tree.

When adding a single node to the graph, we must pick some existing node and extend it in some direction. To ensure that we do not generate redundant nodes and can explore the C-space effectively, our new node should not be too close to its predecessor. Additionally, to avoid making huge jumps in exploration that might skip over narrow passages or miss collision checks, we don't want the new node to be too far from its predecessor. To control the exploration, we introduce a step size that sets the length of the edge. This process of growing a node in some direction until reaching the maximum step size or an edge, potentially with kinodynamic considerations, is known as **steering**. Note that we disregard the potential subsequent node if it, or the edge between the nodes, collides with an obstacle or otherwise unnavigable region.

To promote exploration and reduce redundancy, the planner should favor expanding into less-explored regions of the space. In other words, areas that already contain many nodes should be sampled less frequently, while sparsely covered regions should have a higher chance of being explored next. This can be achieved by giving preference to nodes that lie near open, unexplored areas, and biasing extensions in the direction of greater free space. This strategy helps ensure broader coverage of the configuration space and avoids repeatedly sampling already well-covered regions.

An open region of the configuration space is an area that has not yet been densely explored by the path planning algorithm. In practical terms, this means there are few or no existing nodes from the planner's graph or tree structure in that part of the space. Because of this, any node located on the boundary of such an open region will be the nearest existing node to a large number of points within the region.

This observation is central to how exploration naturally emerges in certain planning algorithms. When you move outward from such a boundary node in the direction of the openness, you will not encounter any other tree nodes—simply because none have been placed there yet. There are no competing neighbors to challenge this node's proximity to points in the open space. As a result, the boundary node is the closest existing node to a large portion of that unexplored area.

This idea is formalized using the concept of a **Voronoi region**. For any given node, its Voronoi region is the set of all points in the configuration space that are closer to that node than to any other node in the tree. It effectively partitions the configuration space into areas of influence—one for each node. These regions are computed by bisecting the space between the node and each of its neighbors: each bisector defines the point where another node becomes equally or more nearby, and thus marks the edge of the Voronoi region in that direction. A picture of a serious of nodes (black dots) and their corresponding Voronoi regions is shown below:

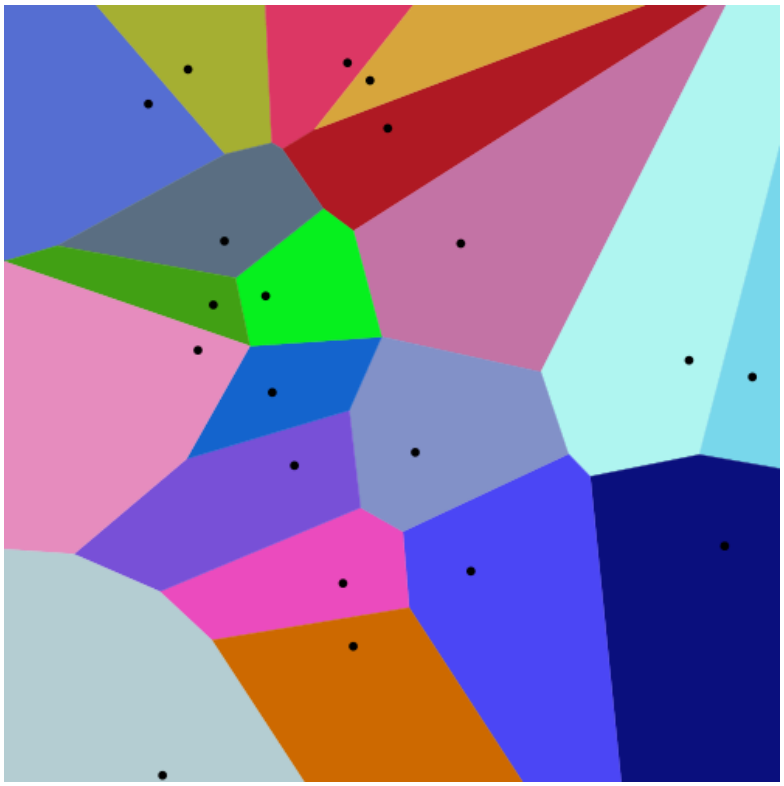


Fig. 3.6: 20 points and their Voronoi cells, calculated using Euclidean distance. [Wikimedia Commons].

In regions where the node has neighbors nearby—i.e., where the space has already been explored—the Voronoi region is cut off fairly quickly. The bisectors are close, and the node's influence does not extend far. However, in the direction of an open region, there are no nearby neighbors. As a result, there are no bisectors to limit the Voronoi region on that side. The cell fans out into the unexplored area, growing until it hits either the boundaries of the configuration space itself or intersects with the region of a distant node. This causes the node's Voronoi region to be much larger in the direction of free, open space.

Picture the cell as a fan or wedge: a narrow slice pointing back toward explored space, and a broad sector opening into the unknown. Geometrically, the solid angle and radial depth toward the open side make up the vast majority of the cell's volume. This is easily seen in the above image.

This geometric structure has important probabilistic implications for sampling-based planning algorithms. If the planner samples points uniformly at random throughout the configuration space, the likelihood that a sampled point falls within the Voronoi region of a particular node is proportional to the volume of that region. This means nodes with large Voronoi regions have a higher chance of being chosen for expansion. The probability that a point falls into the Voronoi region of node i, given by V_i, is the proportion of C-space volume that the volume of V_i takes up. This is mathematically described as below:

$$P(i) = \frac{Vol(V_i)}{Vol(C)} \tag{3.15}$$

Once you extend into that gap and plant a new node there, that new node "steals" the portion of the frontier cell closest to it—carving off a bisector—so the original node's cell shrinks by exactly the volume of the region you just explored. But until that happens, its cell remains large, and every sample in it continues to fall in the still-uncovered free space.

Overall, if we sample a point uniformly at random from the configuration space, it is most likely to fall within the Voronoi region of a node that has a large amount of nearby open space, especially in the direction of that openness. In other words, the sampled point is more likely to be closest to a node whose surroundings are less explored. Importantly, both uniform random sampling and finding the nearest node to a point (based on distance) are computationally efficient operations. Therefore, to encourage exploration in planning algorithms, we should sample a random point uniformly from the configuration space, find the node in the current tree that is nearest to this sample, and then grow the tree from that node in the direction of the sample, following the expansion rules described above.

To guide the tree toward the goal region—rather than allowing it to grow randomly forever—we occasionally bias the sampling process in favor of the goal. Specifically, instead of

sampling a point uniformly at random from the configuration space on every iteration, we replace the random sample with the goal configuration itself once every 5 to 10 iterations (the exact rate can be tuned based on the environment). This technique is known as goal biasing, and it works as follows: when the goal is used as the sample, the planner finds the node in the existing tree that is closest to the goal (based on Euclidean or C-space distance), and then attempts to grow the tree from that node directly toward the goal. If there are no obstacles between the two, this move brings the tree significantly closer to completion. By occasionally treating the goal as the sample, we increase the chances that the tree will connect to it efficiently, without sacrificing the planner's exploratory nature. The combination of uniform random sampling (for exploration) and goal biasing (for convergence) helps balance exploration and exploitation, which is crucial for finding a valid and efficient path, especially in complex environments.

This means that overall, we take the following steps for feasibility single-query path planning:

- **Initialize:** Create a tree T containing only the start configuration q_{start}
- **Loop Until Termination:**
    - **Sample:** With probability p_{goal}, set $q_{rand} = q_{goal}$; otherwise draw q_{rand} uniformly from the configuration space C.
    - **Nearest Neighbor:** Find the node $q_{near} \in T$ minimizing $\| q_{near} - q_{rand} \|$
    - **Steer:** Compute $q_{new} = \text{Steer}(q_{near}, q_{rand}, \delta)$, where δ is the maximum step size
    - **Collision Check**: If the path (straight-line or simulated trajectory) from q_{near} to q_{new} is in collision, discard q_{new} and go back to step 2.1 (sample)
    - **Add Node & Edge:** Insert q_{new} into T and set its parent pointer to q_{near}
    - **Check for Goal**: If $\|q_{new} - q_{goal}\| \leq \delta_{goal}$ (threshold for proximity to goal), terminate with success
- **Extract Path:** Once the goal is reached, backtrack from the goal node to q_{start} via parent pointers to assemble the collision-free path.

This algorithm is known as Rapidly-Exploring Random Trees (RRT) (LaValle, 1998). The sequence of images below illustrates how RRT builds a path through the configuration space. The images should be read in the following order: top left, then bottom left, followed by bottom right, and finally top right. In these images, the green dot represents the starting configuration, the red dot marks the goal configuration, black regions indicate obstacles, and the gray lines and points represent the growing tree—its edges and nodes. In the final image (top right), the highlighted segment shows the successfully constructed path from the start to the goal.

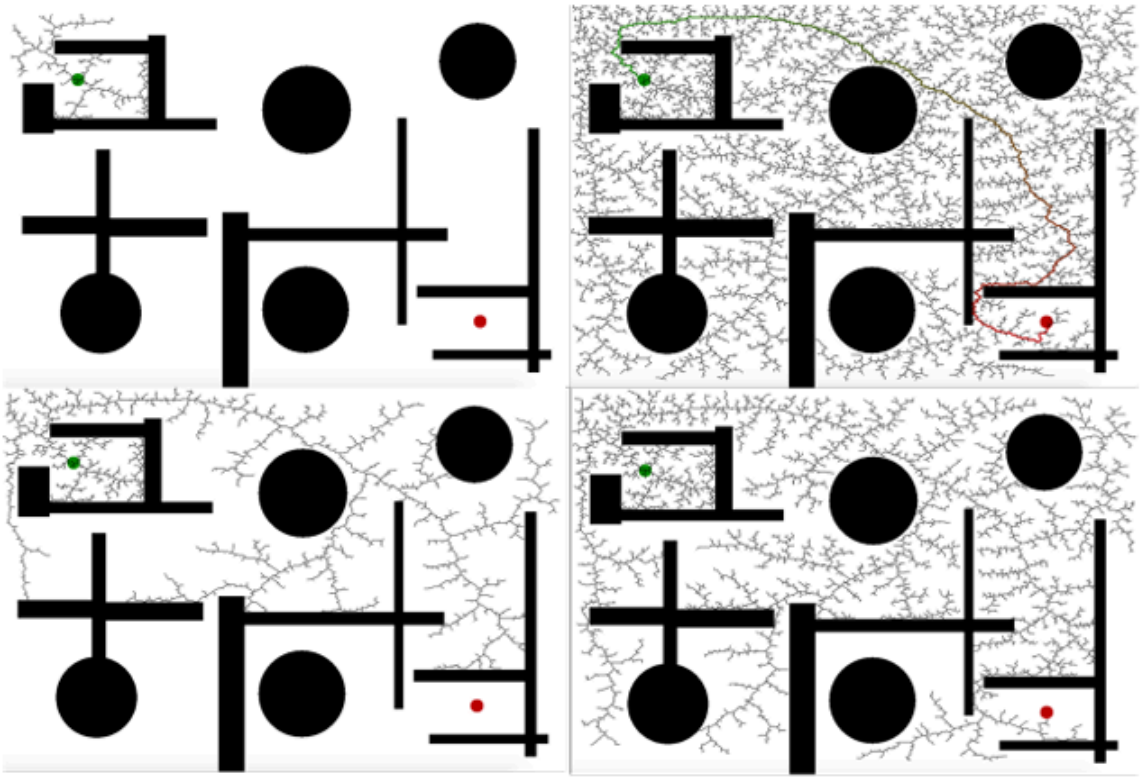


Fig. 3.7: Counterclockwise: Random exploration of a 2D search space using randomly sampled points and connecting them into a graph until a feasible path between the start and the goal is found. [Correll (2022), LibreTexts].

Note that there are versions of RRT that allow for some optimality considerations (mainly RRT*).

3.6 - Local Path Planning

3.6.1 - Introduction to Local Path Planning

As mentioned in 3.3, graph-based path planning has many potential issues. First off, it requires a map. This means that either the entire relevant C-space must be sensible at all times, or the C-space be explicitly explored, mapped, processed, and stored. This is oftentimes infeasible, such as when the map is too large to store, the environment changes over time rendering maps outdated, or the robot does not know where it will operate far ahead of time. Additionally, if there are fast moving targets or obstacles, or multiple moving obstacles, then the rate and magnitude of required replanning would be computationally infeasible in real-time for most processors.

In many real-world scenarios, both the robot's state and the environment can change in unpredictable, random, or chaotic ways, making long-term predictions unreliable. In such cases,

the robot cannot depend on a fixed, precomputed plan. Instead, it must make decisions in real time, continuously adjusting its behavior based on its current state. This state must be inferred from immediate and recent sensor readings, which reflect the robot's current understanding of its surroundings.

In this paradigm, the robot periodically processes its sensor data to estimate its present state and then selects the most appropriate next action. Typically, it only plans a single action at a time, since conditions may change before the next decision needs to be made. The frequency of these updates is determined by both the computational capacity of the robot and the rate at which the environment changes.

This strategy—where the robot plans and acts one step at a time, based only on current information—is called **local path planning**. While certain global planners like D* can accommodate some environmental changes, they generally require a pre-existing map and are not well-suited to highly dynamic environments with many rapidly moving elements. In contrast, local planning is designed specifically to handle such uncertainty and real-time adaptation.

In local planning, the robot does not generate a complete path all the way to the goal or account for future decisions beyond the immediate horizon. As a result, it does not need to operate in the full pose space. Instead, because the goal of local planning is to determine actuation commands—such as velocity or acceleration—it is often more efficient to plan directly in velocity space or acceleration space. Velocity planning is especially common because velocity (linear and angular) directly influences key aspects of the robot's state, such as how quickly it approaches goals or avoids obstacles. However, acceleration planning may also be incorporated when the robot must satisfy constraints on acceleration—such as smoothness, safety, or mechanical limits.

This refines the goal of local planning to the goal of using the current state to figure out the optimal robot velocity for the robot to strive for before the next decision point. Since the robot makes one move at a time, it must make the immediate move which it believes to maximize its expected cumulative reward. This means that the robot navigates according to some function that maps the current state (or observations which indicate the state) to the optimal velocity (usually). In this case the relevant set of linear and angular velocities make up the action space. This function is known as the robot's **policy**.

As with all forms of path planning, the goal is to choose the action that minimizes the total cost to reach the goal. This total cost includes both the immediate cost of moving to the next state and the estimated cost to reach the goal from that subsequent state. If the Markov property holds—that is, if the future depends only on the current state and action, not on the history—then we can associate each state with a specific cost-to-go value. This cost-to-go is often captured using a reward function (or cost function, depending on convention). Under the Markov assumption, the optimal action at any state is the one that minimizes the sum of the immediate

transition cost and the cost-to-go from the resulting state. This principle is formalized by the modified Bellman equations, as introduced in the machine planning section above:

$$Q(s, a) = E[r_t + Q(s_{t+1}, a^*_{t+1})|s_t = s, a_t = a] \quad (3.7)$$

$$V(s) = E[r_t + Q(s_{t+1}, a^*_{t+1})|s_t = s, a_t = a^*] \quad (3.8)$$

It is important to note that in simple scenarios, the optimal local navigation for robots can be human-derived. This means that people can derive a probably approximately correct/optimal action according to the overall trajectory reward function given the state. For example, a robot avoiding obstacles using a predefined repulsion force or simple if-then rules based on sensor proximity. This relies on the ability to predict the outcome of certain state-action pairs for some overall reward function. However, this is infeasible or at least very difficult when there are large state spaces, large action spaces, complex reward functions, or highly nonlinear relationships between the state and optimal action. For these reasons, manually defined policies are generally only viable in simpler scenarios.

3.6.2 - Artificial Potential Field

As discussed above, in many situations, such as slower moving robots or holonomic robots, the lower level motion controller can handle orientation given a path. Additionally, in lower speed scenarios or where precisely following a path is not particularly relevant, the transition from the current to desired kinematic state can be handled by a lower level motion controller and not explicitly considered by the path planner. For a local planner, the above two assumptions entail that the robot's action space consists of a velocity vector, consisting of direction and magnitude. Additionally, the only kinematic aspect of the robot's state is its position. As with all path planning, the state must also contain the position of the goal so the robot can act accordingly.

In many local path planning scenarios, the robot does not have access to precise empirical models or reliable predictive methods for estimating more complex costs—such as time, energy consumption, or risk—required to reach the goal from its current state. As a result, distance is commonly used as a proxy for these cost metrics. This simplifies the reward function and makes it easier to compute and reason about in real time. Specifically, the reward function is often designed to explicitly penalize large distances from the goal and encourage movement in the direction that reduces this distance, serving as a heuristic for goal-seeking behavior. Additionally, when it comes to obstacle avoidance, the distance between the robot and nearby obstacles is a highly observable and meaningful measure. A robot close to a wall, object, or hazard should be penalized more heavily than one in open space. Thus, the reward function must also factor in the

robot's proximity to surrounding objects to discourage unsafe or collision-prone paths. Therefore, a well-designed local reward function typically balances progress toward the goal (i.e., distance and direction relative to the target) with safety constraints (i.e., distance from obstacles), guiding the robot to make decisions that are both efficient and safe—even in the absence of full global knowledge or long-term planning capabilities.

This means that our state must contain information as to nearby obstacles. Assuming precise mapping is not relevant, we should represent this information in an Occupancy Grid Map (OGM). As mentioned in 3.4.2, mapping is not as relevant or feasible in largely dynamic environments, but if the environment contains some relevant static obstacles and it is feasible to store them, then static OGM voxels should be remembered.

Once the robot has constructed an Occupancy Grid Map (OGM) of its environment, it can use this representation to estimate the distance to the nearest obstacle in each direction around it. To do this efficiently, the robot only needs to consider voxels within a limited radius of its current position, since any given action will only move the robot a small amount. Obstacles that are far away are irrelevant for immediate collision avoidance, as they cannot be reached in the next time step.

Furthermore, if there are multiple obstacles aligned in the same general direction, only the closest obstacle matters for local planning. For example, if one obstacle lies directly behind another, the farther one can be safely ignored for now—a collision with it would require first colliding with the closer one or making a much larger movement than planned. This approach keeps obstacle checking focused and computationally efficient while still preserving safety.

This means that any cost function we design should account for both the direction and distance to nearby obstacles as well as the distance to the goal. Nearby obstacles should increase the cost (or decrease the value) of a state, discouraging the robot from moving too close to them, while proximity to the goal should decrease the cost, guiding the robot toward its objective. In this framework, the goal acts as an "attractive" force, pulling the robot toward it, while obstacles act as "repulsive" forces, pushing the robot away. This balance between attraction and repulsion defines the overall shape of the cost landscape used in local path planning. This is mathematically described below where x represents the robot's current state and G represents the cost of a state:

$$G(x) = G_{att}(x) + G_{rep}(x) \tag{3.16}$$

G_{att} is the continuous component of the cost function that pulls the robot toward the goal. It is lowest when the robot is at the goal and increases as the robot gets farther from the goal. Its purpose is to make progress toward the target location more desirable.

G_{rep} is the continuous component of the cost function that pushes the robot away from obstacles. Since each nearby object contributes to the difficulty or risk of navigating through a particular state, we assign a repulsive cost to every obstacle the robot perceives. These costs reflect how undesirable it is for the robot to be near each obstacle. This repulsive cost from an obstacle is highest at the obstacle's position and decreases with distance from the obstacle. The total repulsive cost at a given position is denoted as G_{rep}, which is calculated as the sum of the repulsive contributions from all observed obstacle voxels (grid cells) in the robot's vicinity.

This formulation defines G(x) as a continuous scalar-valued function over the robot's configuration space, where x typically denotes the robot's position and possibly other relevant state variables. Since G(x) quantifies the cost or undesirability of being in a particular state, the robot's objective is to move in a way that minimizes this function. In this simplified scenario, we assume that the robot's orientation and dynamic constraints (such as velocity or acceleration limits) either have negligible impact or are handled separately. As a result, the robot's decision-making can focus solely on minimizing G(x), without needing to account for its full kinematic or dynamic state. This makes G(x) sufficient for guiding action selection, where x is the robot's position.

A pictorial representation of this concept, where a robot is attracted to the goal and repelled by obstacles, is shown below:

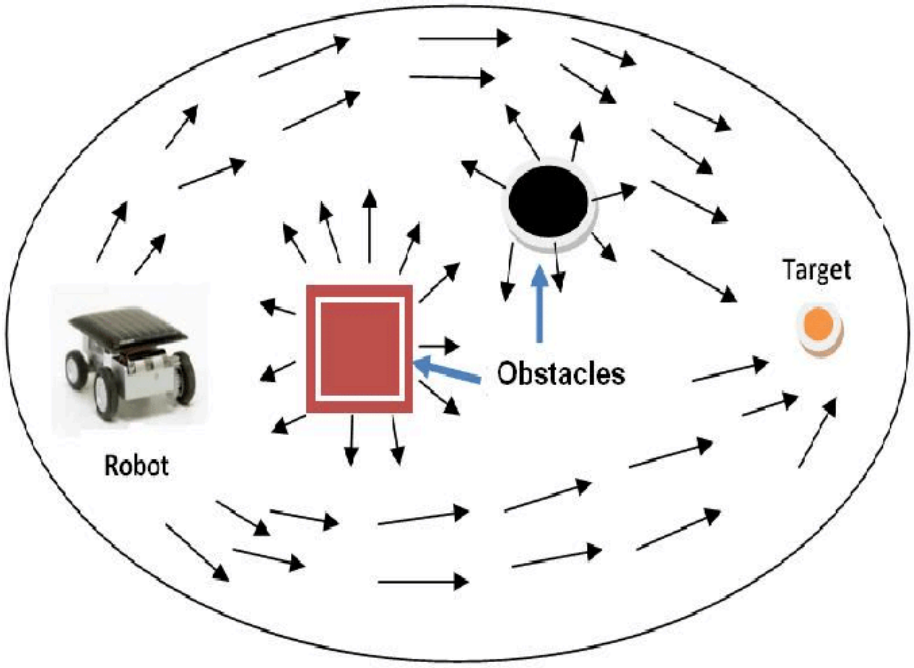


Fig. 3.8: Combination of attractive and repulsive potential fields due to a goal and obstacles. [Matoui et al, 2015].

As discussed in the machine planning section, we often need to explicitly consider immediate reward (or cost) in path planning because different actions may incur different

immediate penalties—such as energy consumption, traversal time, or safety risks. However, including these in the reward calculation introduces additional computational complexity. In many local path planning scenarios, this complexity can be reduced based on simplifying assumptions.

First, the robot updates its action at a fixed frequency, determined by its onboard computation and sensing capabilities. This means that the duration of each action is constant, so actions differ only in their motion direction or magnitude—not in how long they take. This allows us to ignore time as a variable in the cost function, assuming each action consumes the same fixed time interval.

Second, if the robot moves at constant speed, then each action also consumes approximately the same amount of energy, and we can ignore energy as a differentiating factor in reward. If, instead, the robot can vary its speed, then different actions might consume different amounts of energy. However, in many applications, this is either negligible or not worth modeling, especially if energy is not a dominant factor in the task's objectives.

Lastly, even if varying velocity does influence energy consumption, it may be desirable to move faster in some situations—such as when avoiding danger or reaching a goal quickly. In such cases, energy usage may not be treated strictly as a cost. While accounting for energy can improve the reward model, simplifying assumptions often strike a good balance between performance and efficiency, especially when energy is not a critical concern.

For these reasons, we often consider each action as having a roughly similar reward. This means that the value of an action in a state is determined entirely by the value of the subsequent state. This means that the best action is the one that leads to the lowest cost subsequent state. This heuristic can be mathematically described as below:

$$G(s_t, a) \approx G(s_{t+1}) \tag{3.17}$$

We want the robot to move in the direction that minimizes the cost function G(x), which represents the total cost associated with a given position x in the environment. Drawing on the principles of gradient descent, we know that the negative gradient of a function points in the direction of steepest decrease. Therefore, to reduce the robot's cost, it should move in the direction of the negative gradient of G with respect to x.

If the cost function G(x) is defined continuously and differentiably over the spatial domain, we can compute the gradient $\nabla G(x)$. The robot can then use the negative of this gradient as its desired velocity vector. The direction of this vector tells the robot which way to move, and the magnitude of the gradient indicates how rapidly the cost is changing in that direction. In practical terms, the steeper the gradient, the stronger the "pull" the robot experiences toward lower-cost regions—typically toward the goal and away from obstacles.

This approach ensures that the robot moves in the most locally beneficial direction at each time step. The resulting motion is smooth, reactive, and computationally efficient, as it only requires evaluating the gradient of G. This process is mathematically described below, with simplification based on standard gradient rules:

$$v = -\nabla(G(s_t)) = -\nabla(G_{att}(x_t) + G_{rep}(x_t)) = -\nabla G_{att}(x_t) - \nabla G_{rep}(x_t) \quad (3.18)$$

This insight refines our objective: we want to define spatially continuous attractive and repulsive cost functions, denoted G_{att} and G_{rep}, respectively. Let's begin with the attractive cost function G_{att}, which guides the robot toward the goal.

Fundamentally, the farther a point is from the goal, the less desirable it is. So, G_{att} should increase with distance from the goal—penalizing positions that are farther away. But this is not only about ranking locations; the shape of this function affects how the robot moves through space.

When the robot is far from the goal, we want it to move more decisively. This requires a steeper gradient in the attractive cost—the robot should receive a stronger "pull" to correct its course and make efficient progress. This is especially important in large or cluttered environments, where strong goal-directed guidance helps the robot stay focused despite distractions from obstacles or local irregularities in the terrain. Additionally, being far from the goal often means more potential obstacles lie in between, so the robot should weigh goal-seeking behavior more heavily than obstacle avoidance at that stage.

In contrast, when the robot is close to the goal, we want it to slow down and maneuver carefully. Fine-grained movements help it avoid overshooting or oscillating around the target. At this stage, the attractive cost gradient should be shallower, encouraging precise adjustments instead of aggressive movement.

To summarize:

- G_{att} should increase with distance from the goal.
- The magnitude of the gradient of G_{att}—i.e., how strongly it influences the robot—should also increase with distance. This ensures strong incentive to move efficiently when far away, and fine control when near.

Mathematically, this means that $G_{att}(x)$ should be a monotonically increasing function of distance to the goal, and its derivative should also increase with distance. This ensures the robot is more aggressively drawn to the goal when far away, and more precise and cautious when approaching the final target.

Depending on the situation, we may want to prioritize either navigation to the goal or avoidance of obstacles. So, we should be able to tune the strength of the attractive force via a

hyper/external parameter. So, we replace k with a scaling factor that must be chosen and tuned (α_{att}). This brings us to our final attractive cost equations, assuming g is the goal state position:

$$G_{att}(x) = \alpha_{att}||x - g||^2 \tag{3.19}$$

$$\nabla G_{att}(x) = F_{att}(x) = \alpha_{att}(g - x) = \alpha_{att}||g - x||(\widehat{g} - \widehat{x}) \tag{3.20}$$

This shows that the negative gradient, referred to as the attractive force, is in the g - x vector direction, as it points from x to g, and is proportional to the length of the vector g - x. Note that we remove constants, as constant scaling is handled by α_{att}.

These equations make the attractive force very strong when the robot is far away from the goal. This could lead to the robot getting dangerously close to obstacles because they are in the direction from the robot to the goal. To prevent these forces from being too dominant when x is very far away from g, we usually put a cap on F. This means that once the distance exceeds a certain value, we want the force to no longer increase as the distance increases. This means that we must make G_{att} piecewise, where once the distance eclipses a certain pre-chosen and tuned value, $U_{att} = \alpha||x - g||$, meaning that $F_{att} = \alpha(g - x)/||x - g||$.

As for obstacles, we want to avoid them. This means that the badness of a point x in the vicinity of an occupied voxel, should be inversely proportional to the distance between the robot and a nearby obstacle. We also want the force strength to increase as the robot is closer to the object as the risk of collision is greatly increased and we often want immediate avoidance of collisions to be prioritized over reaching the goal. Additionally, colliding with an obstacle is generally seen as worse than deviation from the path to the goal, so we often want the cost signal of not colliding with objects to be slightly more powerful than the attractive force, leading to a slightly greater force exponent.

As in attraction, the optimal strength of object fields varies in certain situations. We may want to weigh object avoidance more or less in certain situations. For example, when we know the environment will have more dynamic obstacles, since these obstacles can move, we want to make sure we are farther away from them. So, we would weigh obstacle avoidance more heavily. Therefore, we want a hyperparameter to control the strength of the effect of obstacles on the robot.

For these reasons, a common formulation of repulsive potential for an individual obstacle, **o**, is given as below:

$$G_{rep}(x,\ o)\ = \alpha_{rep}\frac{1}{||x-o_x||^2} \tag{3.21}$$

$$F_{rep}(x,o)\ = \alpha_{rep}\frac{(x-o_x)}{||x-o_x||^2} = \alpha_{rep}\frac{(\hat{x}-\hat{o_x})}{||x-o_x||^3} \tag{3.22}$$

When defining the repulsive cost function G_{rep}, which increases the cost of states near obstacles, we want to ensure that it reflects how the risk of collision decreases with distance. Specifically, the closer the robot is to an obstacle, the higher the cost should be—discouraging it from moving into dangerous regions. However, this influence does not need to extend infinitely.

At a certain predefined distance threshold (denoted d_{thresh}), the obstacle is sufficiently far away that its effect on immediate collision risk is negligible. For example, if the robot is more than 2 meters away from a wall and cannot reach it within a single action, it makes little sense to assign repulsive cost based on that wall. Continuing to assign cost beyond this distance would waste computation and artificially inflate gradients, potentially distorting the path unnecessarily. Therefore, we truncate the repulsive cost beyond this threshold—setting it to zero for any point farther than d_{thresh} from an obstacle.

To avoid a sharp discontinuity at the threshold (which could cause numerical instability or abrupt changes in behavior), we want the repulsive cost function to transition smoothly to zero at exactly d_{thresh}. A common way to achieve this is to define the repulsive cost at distance d (where $d \leq d_{thresh}$) as:

$$G_{rep}(x)\ =\ (\frac{1}{d(x)} - \frac{1}{d_{thresh}})^2 \tag{3.23}$$

Here, d(x) is the distance from point x to the nearest obstacle. This function has several useful properties:

- It is zero when $d(x) = d_{thresh}$
- It increases rapidly as d(x) approaches 0 (i.e., near the obstacle)
- It is smooth and continuous across the domain $0 < d(x) \leq d_{thresh}$
- It avoids singularities at d = 0 and eliminates non-differentiability at the threshold

This mathematical form ensures the robot is repelled only when it is close enough for an obstacle to matter, and the repulsion tapers off gracefully as the obstacle becomes less relevant. This balances computational efficiency, physical relevance, and smooth motion planning.

We can then merely use the position of each obstacle and the goal to calculate the repulsive forces, add the repulsive forces away from obstacles and the attractive force to the

goal, and determine motion accordingly. Due to the similarities to potential energy and the motion of particles under attractive and repulsive forces in fields, this algorithm is known as **Artificial Potential Field (APF)** (Khatib, 1986).

One issue with APF is that gradient descent can get stuck in local extrema, where all surrounding points are of greater cost than the current position of the robot. In this case, the robot would simply not move until the environment changed. To account for this, we often introduce random perturbations or movements to help the robot escape local minima, especially when the robot has not moved for a significant period of time. One other common improvement is to predict the motion, and thus future pose, of dynamic obstacles to estimate collision potential.

Chapter 4: Reinforcement Learning

4.1 - Introduction to Reinforcement Learning

It is important to note that in simple local planning scenarios, the optimal local navigation for robots can be human-derived. This means that people can derive a probably approximately correct/optimal action according to the overall trajectory reward function given the state. For example, a robot avoiding obstacles using a predefined repulsion force or simple if-then rules based on sensor proximity. This relies on the ability to predict the outcome of certain state-action pairs for some overall reward function. However, this is infeasible or at least very difficult in the following scenarios:

- **Complex Reward Function:** When the reward function involves multiple objectives, competing trade-offs, or delayed rewards, it becomes difficult to derive explicit rules. Balancing multiple objectives and determining the correct weight for each one is challenging. Small changes in the weights can result in drastically different behaviors, making manual optimization impractical. For example, a self-driving car needs to balance speed, safety, comfort, and fuel efficiency. A simple rule like "Minimize braking force" or "Always stay in the right lane" might optimize one goal but fail to address others. Similarly, in chess, the immediate reward of capturing a piece might not align with the long-term goal of winning the game. While humans can define individual heuristics (e.g., "Control the center of the board"), these do not guarantee optimal play.
- **Large State Space:** When there are many state variables, manually accounting for their combined effects becomes impractical. Even if the number of state variables is manageable, a large number of possible values per variable—especially in continuous spaces—makes manual control infeasible. Additionally, designing a precise mathematical function that captures the full probabilistic range of each variable across the entire state space is extremely difficult. For example, a self-driving car must process camera feeds, LiDAR scans, radar data, GPS, speed, and nearby vehicles' trajectories, making the state space massive.
- **Large Action Space:** A large number of action variables similarly increases complexity, making it hard to predict how small changes in one action affect outcomes. If actions are continuous, defining an explicit mathematical function that captures the nuanced effects of small adjustments becomes especially challenging. For example, a bipedal robot (e.g., Boston Dynamics' Atlas) must control dozens of motors to maintain balance while walking on uneven terrain.
- **Nonlinearity:** If the mapping from state inputs to actions is highly nonlinear, manual methods struggle because they must account for complex, multi-layered dependencies between variables. The relationship between a state change and the corresponding

optimal action may itself change across the state space, making it difficult to define simple rules. For example, in stock trading, a small shift in market sentiment, trading volume, or price trends can lead to vastly different optimal trading actions. The impact of these factors is context-dependent and nonlinear—a rise in volume might signal a rally in one situation but a sell-off in another.

In these situations, a different approach is needed to derive a policy. As previously discussed, the goal is to approximate the optimal policy function based on our underlying objectives, knowledge of the system's dynamics, and available data. However, in many real-world scenarios, humans lack the computational power or memory capacity to derive this function manually—especially in high-dimensional, dynamic, or uncertain environments. Therefore, we must rely on mathematical models and computational algorithms to learn the policy automatically.

For these algorithms to optimize a policy effectively, they need access to information about the rewards associated with different state–action pairs. Crucially, these reward signals must reflect the specific system being studied—including its typical environments, dynamics, and task structure—so that the learned policy is relevant and applicable to real-world behavior.

In some situations, supervised learning can be used to directly learn a policy. This approach relies on having labeled data: that is, knowing the correct or desired action for a given state. If such a dataset is available—consisting of many state–action pairs—we can train a model to predict the appropriate action for a new state based on what the expert (human or algorithm) would have done. This form of supervised policy learning is known as *imitation learning*. However, in real-world environments, this method is often impractical. The state space is typically vast, continuous, and dynamic, making it difficult to collect a sufficiently comprehensive and diverse set of expert demonstrations. Moreover, human decision-making is often limited by partial observability, uncertainty, and inconsistency, making it difficult to define "correct" behavior across all states. In fact, one of the key motivations for using machine planning and robotics is precisely to exceed human limitations—either by performing tasks more efficiently, more reliably, or in environments where human operation is undesirable or infeasible.

This points to a class of algorithms that use experience data from a system choosing a variety of actions in a variety of states, then gauging the reward to determine the expected optimal policy from the data. This would essentially be a type of machine learning where an agent learns to make decisions by interacting with an environment in order to maximize a cumulative reward over time. The agent would explore the environment, taking actions and receiving feedback (rewards or penalties) from the environment, which it uses to learn the best strategies or policies for achieving its objectives. Once this optimal policy is established, it can be used for effective navigation. This is known as **reinforcement learning (RL)** (Wiener, 1948).

4.2 - Reward Functions

4.2.1 - Bellman Equations

As mentioned in machine planning (3.2), the value in an action is embedded in its effects. These effects are both short term and long term. For these reasons, as mentioned above, to evaluate an action, RL algorithms must analyze the entire trajectory from that action onward. After each action the robot takes, a new state is reached, and the reward function evaluates how desirable this new state is in relation to the robot's objectives. Therefore, rewards should be assigned based on the subsequent states caused by the robot's actions. In tasks where the robot continuously takes actions, rewards can either be evaluated after each action or periodically based on changes in relevant state variables. Thus, the reward for a given action or sequence of actions is determined by the resulting state, s_{t+1}, and provides feedback that guides the robot's learning toward an optimal policy.

As discussed in machine planning, our guiding equations for path planning are as follows:

$$J(\theta) = E_{\tau\sim\pi_\theta}[G(\tau)] = \sum_{\tau} p(\tau|\pi_\theta)G(\tau) \quad (4.1)$$

J(θ): The objective function, i.e. the expected return. We want to maximize this by tuning the policy parameters θ.

θ: The parameters of the policy network $\pi_\theta(a|s)$. It determines the probability distribution over actions given a state.

τ: A trajectory = sequence of states and actions: $\tau = (s_0, a_0, s_1, ..., s_t)$.

p(τ|θ): The probability of sampling trajectory τ under the current policy π_θ and the environment's transition probabilities.

G(τ): The return (total accumulated reward) from the trajectory τ

E[...]: Expectation over trajectories sampled according to the policy π_θ

$$Q(s, a) = E[r_t + Q(s_{t+1}, a^*_{t+1})|s_t = s, a_t = a] \quad (4.2)$$

Q(s, a): The expected return starting at state s, taking action a, and thereafter following the current policy.

E[...]: The expected value over possible future states, actions, and rewards.

r_t: Immediate reward received after taking action a in state s.

$Q(s_{t+1}, a^*_{t+1})$: The estimated return from the next state s_{t+1}, assuming the next action a^*_{t+1} is chosen optimally (usually by the current policy).

a^*_{t+1}: The action selected by the policy in state s_{t+1}, i.e., $a^*_{t+1} = \pi(s_{t+1})$.

$s_t = s$, $a_t = a$: The current state-action pair from which this expectation is conditioned.

$V(s) = E[r_t + Q(s_{t+1}, a^*_{t+1})|s_t = s, a_t = a^*]$ (4.3)

V(s): The expected return when starting in state s and acting according to the policy thereafter.

However, rewards received further in the future are less reliable indicators of the quality of a specific action, because they depend not only on that action but also on many future decisions. To address this, we reduce the influence of distant rewards when evaluating a given state–action pair. In other words, we assign more weight to rewards that occur closer in time to the action being evaluated. This helps mitigate part of the *Credit Assignment Problem*, which refers to the difficulty of determining which past actions were responsible for a given outcome—especially when outcomes are delayed.

That said, we don't want to completely ignore future rewards. Instead, we want their influence to diminish gradually. To achieve this, we introduce a **discount factor**, denoted by γ, which is a number between 0 and 1. This factor is raised to the power of **i**, where **i** is the number of time steps into the future. Each reward $\mathbf{r_{t+1}}$ is multiplied by $\gamma^{\mathbf{i}}$, reducing its contribution to the total return the farther it is from the present time step.

This discounting ensures that each reward contributes less than the previous one: for example, a reward one step in the future is weighted by γ, two steps by γ^2, and so on. This creates a decaying influence that approaches zero asymptotically but never fully disappears.

$$G_t = \sum_{i=0}^{n} \gamma^i r_{t+i} \quad (4.4)$$

$$\gamma^{i+1} = \gamma * \gamma^i \quad (4.5)$$

This formulation allows us to evaluate the long-term impact of actions while prioritizing rewards that are more immediately attributable to those actions.

This refines our reward equations to what are known as the **Bellman Equations** (Bellman, 1957). These are mathematically described as below:

$Q(s, a) = E[r_t + \gamma Q(s_{t+1}, a^*_{t+1})|s_t = s, a_t = a]$ (4.6)

$V(s) = E[r_t + \gamma Q(s_{t+1}, a^*_{t+1})|s_t = s, a_t = a^*] = E[r_t + \gamma V(s_{t+1})]$

(4.7)

4.2.2 - Reward Design

In traditional path planning, immediate rewards are typically straightforward and are assigned when a predefined objective is met or a task is successfully completed. The primary purpose of these rewards is to reinforce desirable behavior—specifically, the achievement of targeted outcomes. These types of rewards are known as **direct rewards** because they correspond directly to a measurable, externally defined success condition. For example, if a robot successfully navigates through a maze and reaches the designated exit, it receives a reward for

completing the task. Similarly, in a video game environment, a player might receive a reward for completing a level, capturing a flag, or defeating an enemy. In warehouse robotics, a robot might earn a reward after delivering an item to the correct location. In each of these cases, the reward is tied to a clear, final success condition.

However, there are some issues with only using direct goals. Since the agent only receives feedback at the task's completion, any intermediate actions or partial progress go unrewarded. This results in infrequent feedback during the learning process, where the agent may perform many actions without receiving any indication of whether it's on the right track. When rewards are **sparse**, the agent may have to explore blindly without any clear indication of progress. For example, in many complex scenarios, the action has to take specific sequences of actions to accomplish the goal. If the robot explores blindly without rewards, it may take a very long time to get a reward. The robot will likely have to encounter this rare reward multiple times before its policy is sufficiently optimal to frequently receive a reward. This means it will take a lot longer for the robot to learn the optimal policy.

Also, direct rewards are usually designed to signal success or failure at the end of a task. These tasks often involve long sequences of actions, but the agent only knows whether its strategy worked after it has performed all of them. This is known as a **delayed reward**. When rewards are delayed, it becomes difficult for the agent to determine which specific actions were responsible for achieving the final result. This is a form of the credit assignment problem. For example, if a robot receives a reward only after successfully navigating a maze, it might struggle to understand whether the decisions it made at the beginning, middle, or end of the journey were most important for reaching the goal.

Delayed and sparse rewards both pose challenges because they treat all actions that do not directly achieve the robot's objectives equally, provide infrequent feedback, and often occur long after the catalytic actions or sequences of actions. However, some actions bring the robot closer to its goal, while others move it further away. For example, if a robot needs to pass through a small door in a wall, moving toward the door is beneficial, while moving away is detrimental. Similarly, in a maze, reaching checkpoints signals progress toward the target. Sparse rewards often treat goal completion as a binary outcome, but in reality, some attempts are better than others. For instance, a basketball shot that bounces off the rim is more desirable than one that misses the hoop or backboard entirely. In these cases, it makes sense to reward the robot for progress or for good attempts that improve its chances of success. These types of rewards are known as **shaped rewards**. Shaped rewards fall under the broader category of frequent rewards, commonly referred to as **dense rewards**. They fall under the category of rewards which are provided soon after the catalytic action was taken, known as **immediate rewards**.

However, care must be taken to ensure the robot does not learn a policy that exploits dense shaping rewards in ways that undermine or neglect the overall task objective—a

phenomenon known as reward hacking. For example, consider a robot navigating a cluttered environment. Early in training, it may frequently collide with obstacles while exploring. To discourage such behavior, one might introduce a shaping reward that encourages maintaining distance from obstacles. While this can be beneficial initially, it can later lead to overly cautious behavior, where the robot avoids efficient but safe paths in order to maximize the distance-based reward. In the final policy, the primary objective should be to reach the target efficiently. Therefore, shaping rewards should play a more prominent role during early learning and gradually diminish in influence as the policy improves. This is commonly achieved by applying a decaying weight—based on training progress or performance—to the shaping terms in the reward function.

In the simplest case, single-objective rewards target one specific aspect of the task, simplifying the learning process but potentially missing other important factors. For instance, focusing solely on reaching the maze's end might overlook the importance of efficient navigation. In these cases, the reward will be a function of a singular piece of data.

Multi-objective reward functions incorporate multiple goals or performance criteria, allowing the agent to balance different aspects of its behavior—such as efficiency, accuracy, and safety. While this approach introduces additional complexity, it enables more nuanced learning, as the agent learns to optimize across several objectives simultaneously. In such cases, the reward function depends on multiple inputs, each corresponding to a specific objective or desired behavior. To ensure proper learning, the agent typically receives a separate reward signal for each objective. However, these individual rewards must ultimately be combined into a single scalar value, since most reinforcement learning algorithms require scalar feedback. A common method is to sum the individual rewards, but because some objectives are more important than others, each component is usually scaled by a weight. These weights determine how much influence each objective has on the overall reward and must be carefully chosen to reflect their relative importance. In practice, these weights are often treated as tunable hyperparameters during training.

4.3 - Exploration vs. Exploitation

Each time the agent acts in accordance with some policy, they sample from the true distribution of subsequent states following some state-action pair. By the Law of Large Numbers (LLN), as the number of times we perform some state-action pair increases, the distribution of subsequent states and returns approaches the true distributions. Additionally, the expected subsequent return of experienced trajectories approaches the true expected return of a state-action pair.

At the start of training, the agent has no prior knowledge about the expected return—that is, the long-term cumulative reward—associated with any state-action pair. To gain this

knowledge, the agent must actively explore different states and actions within the environment. This **exploration** is a fundamental component of the RL process because it allows the agent to gather information about how the environment responds to its actions.

As the agent interacts with the environment, it observes the consequences of its actions—specifically, how taking a particular action in a given state leads to a new state and a reward. Through repeated interaction, the agent begins to build a dataset of state-action transitions and associated rewards. Using this data, the agent estimates which actions are more desirable—i.e., which ones tend to lead to higher expected returns in each state.

However, to ensure that the agent does not settle for a suboptimal policy (i.e., a strategy that only appears good based on limited experience), it must explore the state-action space thoroughly. This includes trying actions that may not initially seem promising, in order to discover potentially better long-term strategies. Only with sufficient exploration can the agent identify the global optimum policy—the policy that yields the highest expected return from any given state.

Once the environment's dynamics are well understood—meaning the agent has a good estimate of the probabilities of transitioning from one state to another under various actions, or the expected value of state-action pairs—it can make more informed decisions. At that point, the agent can rely more heavily on exploitation, selecting the best-known action in each state to maximize its cumulative reward.

The formal framework that describes this interaction between states, actions, rewards, and transitions is called a **Markov Decision Process (MDP)**. An MDP provides a probabilistic model of the environment, specifying the likelihood of transitioning from one state to another given a particular action, along with the expected reward. This structure underpins most reinforcement learning algorithms and provides the mathematical foundation for optimal decision-making over time.

However, in many real-world environments, the state and action spaces can be so vast that it becomes impractical for the robot to explore and learn about every possible state-action pair. For example, in environments with hundreds or thousands of potential states and actions, fully exploring each one would be computationally infeasible and time-consuming. Even if the robot had unlimited time, the sheer number of possible actions in each state would make it impossible to explore every possible combination exhaustively. This means the agent must selectively explore its environment.

In practice, not all policies and state-action pairs are equally valuable. Policies that consistently yield poor or marginal results are less critical to explore further. We do not need to increase our accuracy for rewards and probabilities in selecting certain state-action pairs that are almost certainly suboptimal. Instead, the focus should be on exploring policies that have the potential to be close to the global optimum, as they offer more promise for achieving high

rewards. In many scenarios, such as games, certain states or objectives are only reachable after completing specific tasks. This necessitates following a sufficient policy to enable further exploration, a process known as **exploitation**.

By concentrating on policies that are more likely to be optimal, we can efficiently validate their effectiveness and robustness. Even if a policy appears promising based on initial data, continued exploration ensures it performs well across different scenarios and conditions. This validation is essential for confirming that the policy consistently achieves high rewards across the state-action space. Exploration of the current best policy can also uncover nuances or edge cases not addressed during the initial learning phase, allowing for small adjustments or improvements. Additionally, if a policy is optimized based on limited or biased data, it might not generalize well to unseen states or actions. As new information becomes available, it's possible to discover better policies. This strategy of exploring the currently best-known policy is known as the **Greedy Strategy**.

However, if the agent always follows the currently believed optimal policy, it risks becoming trapped in a local optimum—a suboptimal policy that appears to perform well simply because the agent has not explored better alternatives. This problem arises from insufficient exploration. To overcome it, reinforcement learning requires a careful balance between exploration and exploitation. This trade-off implies that while we should favor actions believed to yield high rewards, we must not entirely exclude actions that currently appear less favorable, as they might turn out to be better with more experience. As a result, the policy used during training must be stochastic—it should assign probabilities to different actions rather than always selecting the single best one. Moreover, it must be reward-aware, meaning it uses estimates of expected returns to guide action selection while still maintaining some level of randomness to facilitate exploration.

The way that exploration is performed is inherently dependent on the structure of the state and policy. If each state inherently has a finite number of actions, one can simply evaluate the return for each action and select the one with the highest value. This approach that involves estimating the expected return of one or more actions is known as **Value-Based RL**. Even if the set of mechanically possible actions is continuous, the action space is often discretized. With a discrete action space, the agent only needs to explore a finite set of actions rather than an infinite set, making exploration more structured and efficient. Discrete actions allow straightforward computation of the best action. This can be beneficial if small changes in the chosen action do not significantly affect the expected return of the action.

In value-based reinforcement learning, we can manually iterate over each action to estimate its expected value and thereby identify the optimal action. In a discrete action space, balancing exploration and exploitation comes down to sometimes choosing the action we believe is best and sometimes choosing a random one. To control this balance, we introduce a

hyperparameter called ϵ (epsilon). At each decision point, the agent selects a random action with probability ϵ (exploration), and selects the currently estimated best action—the greedy action—with probability $1 - \epsilon$ (exploitation). A high value of ϵ encourages exploration, helping the agent avoid getting stuck in local optima, while a low ϵ promotes exploitation of known good strategies. This simple mechanism ensures that all actions are occasionally explored, while still favoring the most promising ones most of the time. This technique is known as the **epsilon-greedy strategy** (Sutton & Barto, 1998).

However, the epsilon-greedy strategy is typically implemented with a constant ϵ. While this works in principle, it doesn't account for the fact that exploration is most valuable early in training, when the agent has limited knowledge of the environment. As the agent's understanding of the underlying MDP improves, the need for exploration diminishes, and exploitation should be prioritized.

To reflect this shift, ϵ should start at a relatively high value and gradually decrease over time—either based on the number of steps, episodes, or some performance-based criterion. This technique is known as **epsilon decay**. Epsilon decay can be implemented in various ways, such as linear decay, exponential decay, or step-based schedules. No matter the decay type, we typically include a predefined lower bound for epsilon to ensure that some exploration persists throughout training.

However, this approach treats all suboptimal actions equally by assigning them the same probability of being chosen, which is problematic because it implies that all suboptimal actions have an equal chance of being optimal. In reality, actions with higher expected rewards are more likely to be optimal. Therefore, we should base the probability of selecting an action on its expected reward, ensuring that the likelihood of choosing each action is proportional to its expected reward. This approach better reflects the true value of each action.

Another limitation of the epsilon-greedy strategy is that it treats exploration uniformly across all states at a given time. That is, at each decision point, the probability of exploring versus exploiting is the same regardless of which state the agent is currently in. However, this approach ignores the fact that different states may have been visited with very different frequencies over the course of learning. For instance, some states may occur often—such as those that appear early in every episode—so they become well-explored. Others, particularly those that can only be reached through rare sequences of successful actions, may remain largely unexplored. These "deep" or "high-reward" states might be critical for learning the optimal policy, but if epsilon-greedy continues to explore equally in all states, it may waste exploration steps on already well-understood regions of the state space while neglecting those that are under-explored. This imbalance is problematic because the value of exploration is not the same in every state. Exploration is much more important in states where the agent has limited data or high uncertainty about outcomes. By failing to account for this, epsilon-greedy can lead to

inefficient learning—over-exploring familiar territory while failing to gather the information needed in rare or difficult-to-reach states.

In any given state, the necessity of exploration depends on how confident we are in the current estimates of action values. If one action has a clearly higher expected reward than all others—based on prior experience—then we can be reasonably confident that this action is the best, and further exploration of alternative actions is less necessary. Exploiting this high-value action is more efficient in such cases. However, if multiple actions have similar estimated rewards, and there is no strong reason to favor one over the others, then the agent lacks certainty about which action is truly optimal. In this situation, exploration is essential. By trying out these competing actions, the agent gathers more information, which helps improve the accuracy of its value estimates and better distinguish between the actions in the future. This insight suggests that the probability of selecting an action during exploration should depend not just on randomness (as in epsilon-greedy), but also on the relative quality or expected reward of each action. Actions with higher expected rewards should be chosen more often, but less-promising actions should still be tried occasionally—especially if their estimates are based on limited data.

This means we need to create a probability distribution over all actions for each state, where a higher expected reward corresponds to a higher probability of selection. The softmax function provides a way to achieve this. It handles negative values by assigning them lower probabilities compared to small positive values. Moreover, when there is a significant difference between the expected rewards of actions, the softmax function ensures that the probability of selecting actions with very low expected rewards becomes negligible. This is desirable because such actions are unlikely to be part of the optimal policy compared to actions with high expected rewards. The softmax function can be expressed using the following equation:

$$P(a_i|s) = \frac{e^{Q(s,a_i)}}{\sum_j e^{Q(s,a_j)}} \tag{4.8}$$

However, different scenarios require different balances between exploration and exploitation. To control this trade-off in a principled way, we introduce a hyperparameter known as the **temperature** (T). The temperature determines how sensitive the action selection probabilities are to differences in expected rewards. A high temperature should promote exploration by flattening the softmax probability distribution. This means the differences between high and low expected values are suppressed, so all actions receive more similar probabilities, even if some actions are clearly better than others. As a result, the agent is more likely to sample a wider variety of actions. Conversely, a low temperature should promote exploitation by sharpening the distribution. The probabilities become more skewed toward the actions with higher expected values, meaning the agent will strongly prefer those actions and be

less likely to explore alternatives. To achieve this effect, we scale each expected return (Q) by dividing it by the temperature before applying the softmax function. Specifically, the probability of selecting action a in state s is given by:

$$P(a_i|s) = \frac{e^{Q(s,a_i)/T}}{\sum_j e^{Q(s,a_j)/T}} \quad (4.9)$$

This formulation ensures that the temperature modulates the contrast between Q-values. When T→∞, all actions become equally likely (maximum exploration). When T→0, the highest value action dominates the distribution (pure exploitation). This provides a flexible and mathematically grounded way to control exploration behavior based on the agent's current level of certainty or the learning stage.

This approach is known as the **Softmax Strategy**. Similar to epsilon-greedy, temperature decay is often used to shift the balance from exploration to exploitation over time. This method is overall more computationally intensive, but provides the benefits explained above.

4.4 - Q-Learning

When evaluating the discounted expected reward of an action—referred to as its **Q-value**—the need for exploration becomes increasingly important due to the branching nature of future possibilities. As a reminder, the Bellman equations are shown below:

$$Q(s, a) = E[r_t + \gamma Q(s_{t+1}, a^*_{t+1})|s_t = s, a_t = a] \quad (4.6)$$

$$V(s) = E[r_t + \gamma Q(s_{t+1}, a^*_{t+1})|s_t = s, a_t = a^*] = E[r_t + \gamma V(s_{t+1})] \quad (4.7)$$

Each action can lead to a new state, and each of those states has its own set of possible actions and transitions. As a result, the number of possible future trajectories grows exponentially with time, making it essential to explore a wide range of paths to accurately estimate long-term returns.

Moreover, to compute the Q-value accurately, we must understand the transition probabilities associated with each state–action pair—i.e., the likelihood of reaching a particular next state given the current state and action. This means we need a model of the environment's dynamics — one that tells us how the system evolves in response to actions. However, accurately learning such a model becomes instrumentally unfeasible in high-dimensional or noisy environments. This is because planning requires that the model be reliable not only where data is abundant, but also in parts of the state space that the agent hasn't frequently visited. In high-dimensional spaces, the number of possible states explodes, making it practically impossible to gather enough data to ensure the model is accurate everywhere. Likewise, in environments where transitions are stochastic or sensor readings are noisy, a learned model must

capture that uncertainty faithfully — yet most models either oversimplify or overfit, introducing bias. This bias then leads to poor action selection when the agent plans using the model, because the selected actions exploit flaws in the model rather than reflecting true high-value behavior. Therefore, from the goal of stable and effective learning, it follows that relying on a learned dynamics model may introduce instability and bias.

In reinforcement learning, transition probabilities are often used as an intermediate tool to compute the expected return of taking a particular action in a given state. However, this intermediate step is not always necessary. From a decision-making perspective, what ultimately matters is the expected return itself, not the full distribution of possible next states. For instance, consider an action that leads to a reward of 6 with 50% probability and a reward of 8 with 50% probability. The expected return is 7. From the standpoint of selecting actions, this is indistinguishable from receiving a guaranteed reward of 7—regardless of the underlying outcome probabilities. This illustrates that if our goal is to learn which actions yield the highest expected returns, we can bypass explicit modeling of transition dynamics and instead learn the Q-values directly. This idea is central to **model-free** reinforcement learning, which focuses on estimating the value of actions from experience without learning the environment's transition model. In contrast, **model-based** approaches aim to learn the environment's transition and reward functions explicitly, using them to simulate outcomes and plan accordingly.

Importantly, it is suboptimal to recalculate the expected return for a state-action pair after each episode. Doing so may dilute the influence of new experiences, as recent insights could be minimized by the contributions from older, potentially outdated experiences. This is because recalculation treats all episodes equally, which can lead to scenarios where older experiences disproportionately sway the average, even if they are no longer relevant to the agent's current strategy.

In practice, it is often beneficial to give more weight to recent experiences than to older ones. This is because newer experiences are generated under a more refined policy—reflecting the agent's ongoing learning and improvement—and they may also capture recent changes in the environment's dynamics. To account for this, we update our value estimates using a weighted combination of the previous estimate and the new experience, effectively forming a running average. To control how much influence new information has, we introduce a parameter that defines this weighting. Different tasks and environments may require different levels of sensitivity to new data, so this parameter must be tuned accordingly. This approach allows the agent to remain responsive to new information while maintaining some stability from past knowledge.

$$Q(s,a) \leftarrow w_1 Q(s,a) + w_2 G \tag{4.9}$$

Q(s, a) is the prior Q-value belief, while G is the observed discounted cumulative return in the episode following the state-action pair.

Generally the weight on the current Q-estimate is significantly higher since we want all prior experiences to weigh more than a single current experience. By performing some simple algebra, noting that the weights must sum to 1, this can be rewritten as:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(G - Q(s,a)) \quad (4.10)$$

G - Q(s,a) represents the difference between the experienced return and the expected return. This is known as the **Temporal Difference (TD) Error**.

α is a tunable hyperparameter known as the **learning rate** which determines the weighting on the new experience relative to the old experience. Initially, a higher learning rate can help the agent make significant adjustments to its estimates based on new experiences. However, as the agent's knowledge increases and it approaches a more stable understanding of the environment, the learning rate can be reduced. This decay helps ensure that updates become more refined and stable, preventing oscillations in the Q-value estimates as the agent converges.

However, directly calculating G requires waiting for the entire episode to learn and update their Q-value estimates. This design is inefficient and unsuitable for non-episodic tasks, where a clear endpoint may not exist. Additionally, this approach leads to inherently sparse rewards, as information about actions taken during the early stages of an episode remains unused. For instance, if an agent takes an action that results in a negative outcome, it may repeat the same mistake multiple times. Furthermore, the agent cannot receive feedback on a policy that incorporates experiences from earlier in the episode. As a result, the policy converges more slowly, especially in environments with longer episodes.

This means that we must create an algorithm which updates Q-values during episodes rather than waiting for entire episodes to conclude. Where n is the number of steps from the current action until the end of the episode. In order to not have to wait until the end of the episode, we could wait a finite number of steps after an action before updating the Q-value. However, when calculating the return after a finite number of steps, you might not capture the full outcome of the episode. Many rewards may occur after the chosen number of steps, which means that you're potentially missing out on significant future rewards that could influence the decision-making process. Also, you still have to wait for multiple transitions before updating your Q-values.

This simplifies our goal to estimating the expected return beginning from the current state, assuming we take a specific action and then follow the optimal policy thereafter. Because we observe the immediate reward and the resulting next state before making future decisions, we can directly use this experience in our Q-value update. The tuple (s_t, a_t, r_t, s_{t+1}), known as an **experience**. In other words, we aim to estimate the value of a state–action pair based on the reward it yields and the return we expect from the next state onward. Fortunately, the recursive nature of the Bellman Equations provides an efficient way to do this. Specifically, the expected return for the current action can be approximated using the observed reward plus the Q-value of

the optimal action in the next state—an estimate that is refined through ongoing experience. Thus, our objective becomes updating each Q-value to reflect the total expected return from that point forward. Thanks to this recursive formulation, we can achieve this using just a single transition at a time. This approach is known as **one-step lookahead**, where each Q-value update depends only on the immediate reward and the estimated return from the next state. The mathematical formulation is provided below.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r_t + \gamma * argmax_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s,a)) \quad (4.11)$$

Initially, all Q-values are unknown, so they are typically initialized to arbitrary values—often zero—or assigned randomly. When the robot begins interacting with the environment, it selects actions at random due to its lack of knowledge. In cases where the robot takes an action and transitions to a previously unseen state, it must update the Q-value based solely on the immediate reward received. However, as the robot continues to explore, it will eventually reach states for which some Q-values have already been updated. This enables the Q-value update to incorporate not only the immediate reward but also the maximum expected future reward from the next state. This approach—updating current estimates using previously learned estimates—is known as **bootstrapping**. It allows future experience to influence the value of earlier decisions, enabling Q-values to gradually reflect longer-term outcomes. Over time, this recursive updating causes Q-value estimates to account for more steps into the future, improving their accuracy as the robot gathers more experience.

Another key factor contributing to improved Q-value estimates is the recursive nature of their updates. Specifically, once the Q-value estimate for a particular state-action pair becomes more accurate, this improvement directly benefits the estimation of earlier state-action pairs that lead to that state. This is because Q-learning updates not just based on immediate rewards, but also on the estimated future return—i.e., the maximum Q-value of the next state. As these future Q-values become more reliable, they improve the quality of updates for the Q-values of prior actions that transition into those future states. As the robot interacts with the environment and discovers better strategies in future states, it retroactively adjusts the Q-values of earlier decisions to reflect these higher-quality outcomes. Over time, and with sufficient exploration, this process causes the Q-values for all state-action pairs to become increasingly accurate, ultimately converging toward the values defined by the optimal policy.

Estimating the Q-value of a specific state-action pair using a one-step lookahead, where the expected return of the optimal action in the subsequent state is used to update the current state-action pair's value, is known as **Q-learning** (Watkins, 1989).

In regular Q-learning, the robot must store each state action pair, usually in a table known as a **Q-Table**. When encountering a state in training, the robot iterates over the Q-values for all actions in that state and chooses the action according to its exploration vs. exploitation method.

When encountering a state in testing, the robot iterates over the Q-values for all actions in that state and chooses the action with the highest Q-value.

4.5 - Deep Q-Learning

Q-learning works well when the state and action space is finite because each state-action pair can be represented and stored separately. However, using a discrete function of the state and action approach becomes impractical when dealing with continuous state spaces, where there are an infinite number of values for a state.

Since the same action will likely have similar effects in similar states, we can discretize continuous variables, dividing them into intervals. However, this comes with significant drawbacks. First, discretization of the state-space can result in a loss of information, as continuous states are reduced to coarse-grained approximations. Small but meaningful differences between states may be lumped together, preventing the agent from making fine distinctions. For example, if the position and velocity of a robot are discretized, subtle changes in these variables that could lead to different outcomes may be lost, causing poor decision-making. For example, a robot navigating a grid might have its continuous position (e.g., (1.25, 3.67)) discretized to a coarser representation (e.g., (1, 3)). Another position close by, such as (1.2, 3.6), could also be discretized to (1, 3), and the agent would treat both as identical, even though they represent slightly different scenarios in the real world.

Secondly, discretization increases computational complexity in environments with large or high-dimensional state spaces. Storing and updating Q-values for every discrete combination of state variables quickly becomes infeasible.

Furthermore, discretization hinders generalization. A core concept in reinforcement learning is leveraging past experiences to learn about similar state-action pairs. However, in standard Q-learning, the agent updates its knowledge only for the exact state-action pair it encounters. When the environment is discretized, this limitation becomes more severe: the agent cannot use experiences from adjacent, yet nearly identical, states to inform its understanding of new ones. For instance, if a robot's position is discretized into 0.1-meter intervals, an experience at 4.001 meters teaches the agent nothing about being at 3.999 meters—even though these states are functionally the same. This restriction is problematic because similar states typically yield similar outcomes when the same action is taken. As a result, the agent must re-learn from scratch in every discrete state, slowing down learning and requiring significantly more data. To learn more efficiently, we need models that can generalize across states and share knowledge between similar situations.

These limitations show that we cannot have a discrete policy or Q-estimation function, and must instead have a **continuous function** that can map states as inputs to Q-values for each action as outputs, providing smooth estimates across the entire state space. Then the action with

the highest predicted value in the state is the predicted optimal action. Note that having each action's Q-value as an output assumes a discrete action space.

Since we often have no prior knowledge of the underlying function that maps states to Q-values, to learn this function from past experiences, we must use machine learning to find this continuous function. Due to the complexity and high dimensionality of the state and action spaces in many real-world problems, as well as the fact that we often have no prior knowledge of the format of such a function, neural networks are an ideal choice as function approximators. This network is known as the **Q-Network**. The predicted Q-value of an action in a given state is denoted as $Q_{Network}(s,a)$.

For the Q-Network to reflect the true values of state-action pairs and reflect the environment's true dynamics, it must be trained on experiences which reflect actual state transitions and immediate rewards received from state-action pairs. This entails using a network's predictions on past experiences, comparing them to the ground truth, and updating the network's parameters accordingly. The algorithm's structure leads to gradual improvements in accuracy as the true optimal actions are figured out over time as well as increases in Q-values as more steps into the future are able to affect the Q-network. This is known as **Deep Q-Learning (DQL)**.

As in Q-learning, the optimal prediction minimizes TD Error. Since we still experience the subsequent state before making subsequent actions, we can still use the one-step-lookahead to predict the Q-value for the subsequent state as well as the observed immediate reward to calculate TD Error. This means we add our observed immediate reward to the highest Q-value outputted by the Q-network for the subsequent state, which is scaled by the discount factor. This means that our ground truth comparator Q-value is given as follows:

$$Target = r + \gamma * max_{a'}(Q_{Network}(s', a')) \quad (4.12)$$

Since this is a regression task, we can use Mean Squared Error (MSE), where error is TD error, as our loss function. This means that TD Loss is given as below:

$$TD\ Loss = (Target - Q_{Network}(s, a))^2 \quad (4.13)$$

Through exploration, we take actions in states that give us immediate rewards (**r**) and transition the agent to a subsequent state. To be able to learn from this data, we must store the tuple (s, a, r, s'), known as an **experience**.

To train the neural network in deep reinforcement learning, one natural idea is to use stochastic gradient descent (SGD), where the network is updated after each individual experience. However, this approach poses several challenges specific to reinforcement learning. Unlike supervised learning, where training data can often be assumed to be independently and identically distributed (i.i.d.), reinforcement learning data is sequentially generated by an agent interacting with the environment. This means that consecutive experiences are highly correlated.

For instance, if a robot is moving in a straight line, then over several time steps, the observed states will be nearly identical. As a result, the experiences the agent gathers over that window are very similar and lack diversity. From the perspective of statistical learning theory, this violates the i.i.d. assumption and causes the training process to suffer from high variance and low generalization. When a neural network is trained repeatedly on highly similar samples, it may overfit to local structure—learning to predict Q-values accurately for recently visited states but generalizing poorly to different parts of the state space.

Moreover, updating the network immediately after each new experience can lead to instability and catastrophic forgetting—a situation where previously learned knowledge is rapidly lost. This is particularly problematic in non-stationary environments, where the agent's experience distribution changes over time and long-term planning requires retaining older knowledge. When the neural network is updated based only on recent experiences, the weights may shift disproportionately toward modeling the latest data, overwriting valuable insights from earlier, dissimilar experiences.

This behavior introduces a form of distributional drift: if the model continually adapts to short-term trends, it effectively begins optimizing for the current joint input-output distribution, which may differ significantly from the true distribution of experiences the agent should be learning from. As a result, the learned policy may become biased toward recent patterns, reducing sample efficiency, harming generalization, and ultimately leading to suboptimal long-term performance.

Finally, from an efficiency standpoint, learning from each sample only once is wasteful. A single noisy experience may not provide enough signal to justify an immediate network update. Instead, reusing past experiences—especially diverse ones sampled from various states—would allow the agent to better estimate expected returns, reduce variance, and learn more robust representations.

These issues show that instead of updating the network after every action, we must train the network on a subsection of recent and distant past experiences. This entails updating the network based on a random sample of past experiences. So, we store past experiences in a **replay buffer**, then randomly sample a **mini-batch** of experiences. This process of sampling a mini-batch of experiences from the replay buffer for learning is called **experience replay**. Generally, once a certain number of experiences have been added to the replay buffer, the mini-batch is sampled and used for training every m time steps or actions. This means that we compute the gradients by averaging them over the N samples. This is often done after a set number of steps (e.g., every 4 or 10 steps) to ensure that the network is updated frequently while still allowing for sufficient exploration.

Over time, older experiences in DQL may lose relevance or even become misleading as the agent's policy improves or the environment shifts. For instance, past experiences may include

suboptimal actions the agent would no longer take or states it's unlikely to encounter with its updated policy. Training on such outdated experiences could cause the agent to improve in less relevant situations while hindering performance in current, high-priority state-action pairs. Since these outdated experiences are also less informative, they add inefficiency to the training process. Additionally, storing all past experiences would eventually require vast amounts of memory, especially in complex environments where agents encounter millions of states and actions. For these reasons, replay buffers are typically capped to retain only a limited number of recent experiences. This allows the agent to continually interact with the environment, refresh its experiences, and sample relevant batches from the buffer for efficient training—a process known as **Standard (Vanilla) DQL** (Mnih, 2013).

However, this approach presents a significant challenge. The DQL update relies on the maximum estimated Q-value for the next state to compute the target. This maximization step introduces selection bias, as the agent will consistently select the action with the highest Q-value. If the Q-values are noisy or slightly inaccurate, there's a natural tendency to select an overestimated action because it appears optimal. Once this overestimated Q-value is chosen as the target, it is used in the loss function to train the network. Specifically, the DQL update rule adjusts the Q-value for the current state-action pair (s,a) towards this target. When the target is overestimated, it causes the Q-value for (s,a) to increase beyond its true value. As a result, the network learns to associate a higher-than-accurate Q-value with this state-action pair. This bias is known as **overestimation bias**.

The problem with overestimation bias is that it becomes entrenched in the Q-network's parameters. Once the network learns to overestimate the Q-value for a particular action in a given state, it is likely to continue overestimating the Q-value for that action in similar states. This becomes especially problematic because future targets may involve states similar to those where overestimation was first introduced. If the overestimated action remains the best choice in these similar states, the target for future updates will also be overestimated. Furthermore, the overestimated Q-value increases the likelihood that this action will be selected as the optimal action in future updates.

This process creates a feedback loop, where the overestimation is reinforced each time the Q-values are updated. Over time, this bias accumulates and compounds, making it increasingly difficult for the agent to correct its estimates and leading to suboptimal policy performance.

Since the same network is used for both target generation and prediction, overestimated targets continue to propagate through the network's updates, reinforcing the bias with each training step. To address this issue, it is important to prevent overestimations from becoming entrenched in the target generation. The target generation network should reflect the learned Q-values, but not be influenced by the overestimations that might be present in the prediction

network. This entails using a separate network for prediction and target generation. To prevent the compounding effect from beginning, we must prevent recent overestimations due to overestimated targets from affecting the evaluation network. To do this while still using the learned Q-Network, we must not update the target generation network every time we update the prediction network. This entails that the Q-network is impacted by each training example, but the evaluation network is unaffected.

This means that we maintain a separate target network, which serves as a stable reference for computing the target Q-values used in the loss calculation. Unlike the main Q-network, which is updated at every training step, the target network is held fixed for a number of steps to prevent instability caused by rapidly changing targets. It is only updated periodically—for example, by copying the weights from the Q-network every 4 training steps. This periodic update helps prevent newly overestimated Q-values from immediately influencing future updates, thereby stabilizing the learning process.

In some cases, instead of a hard update (complete weight copy), a soft update is used. In this approach, after every N steps, the parameters of the target network are adjusted slightly toward the parameters of the Q-network. This is typically done using a weighted average:

$$\theta_{target} \leftarrow \tau * \theta_Q + (1 - \tau) * \theta_{target} \quad (4.14)$$

where $\tau \in (0,1)$ is the soft update rate.

The concept of using two networks—one to select the action and one to evaluate it—is a key component of Double Deep Q-Learning (Double DQL), introduced by Hasselt (2015). In Double DQL, the Q-network is used to select the action with the highest estimated Q-value in the next state, but the target network is used to evaluate the value of that selected action. This decoupling reduces overestimation bias by avoiding the use of the same network to both choose and evaluate actions.

4.6 - Deep Deterministic Policy Gradient (DDPG)

Value-based RL relies on the idea that, in any given state, the agent can use a function to estimate the value of each possible action and then iterate over these actions to select the best one. This approach works well in environments with a finite set of actions, known as a discrete action space. However, in the physical world, many tasks involve actions with numerical parameters. Due to the continuous nature of the world and physical principles, these actions are often unbounded or continuous. For instance, in robotic arm or leg control, the agent must adjust joint angles, velocities, or forces, which can take on a continuous range of values (e.g., from $-\pi$ to π radians). Similarly, in self-driving cars, the agent controls the vehicle's speed, choosing any value within a given range. In these cases, at least one component of the action vector has an infinite number of possible values, creating what is known as a **continuous action space**.

In continuous action spaces, each state has a continuous function that maps actions to rewards. With an infinite number of possible actions, it's impossible to iterate over all actions to find the best one. One approach could be to discretize the action space, but as with discretized state spaces in value-based RL, this sacrifices important information and hinders generalization, as referred to in the last section. This shows that we must have a continuous output, where small changes in the action are reflected in the output. Given this reality, we could sample actions within a state and take the sampled actions with the highest expected return to approximate the optimal action, but this would either require an intensive search—especially in high-dimensional spaces or with wide action ranges—or lead to a likely suboptimal choice. Another option could involve optimization algorithms to systematically sample and refine the best actions, but this demands sophisticated, time-consuming computations every time the agent enters a state. These challenges show that we cannot simply predict state-action values, but rather find a function that maps the state directly to the optimal action.

Since we don't initially know the optimal policy, we need to learn a continuous function mapping inputs to outputs while allowing the flexibility in learning any functional relationship. We typically use a neural network to represent that policy. A RL algorithm which directly uses optimization to guide changes in parameters for a NN mapping states to actions is known as a **policy gradient method**.

To train a neural network to output optimal actions in reinforcement learning, we must optimize its parameters (weights) so that the agent maximizes its expected cumulative reward across all possible trajectories—that is, across all sequences of states and actions the agent might experience over time.

Unlike supervised learning, where each input is associated with a fixed target label, reinforcement learning involves sequential decision-making, where each action not only yields an immediate reward but also influences all future states, actions, and rewards. This means that the consequences of an action can extend far into the future, and so the quality of a given action can only be properly evaluated by looking at the entire trajectory that unfolds after it. Therefore, to judge whether the neural network is making good decisions, we must look beyond immediate outcomes and evaluate how well the entire sequence of actions leads to high cumulative reward.

Furthermore, to learn a robust and generalizable policy, we cannot only consider a few specific trajectories—we must consider all possible trajectories the agent might encounter in the environment. This is because different initial conditions, actions, or stochastic events (such as noise in sensors or dynamics) can lead to different trajectories. If the network is only trained on a limited subset of trajectories, it may perform poorly in new situations that deviate even slightly from the training scenarios. So, during training, we aim to maximize the expected reward: that is, the average total reward across the full distribution of all possible trajectories, weighted by how likely each one is under the current policy.

So, to train a neural network to output optimal actions in reinforcement learning, we must optimize its parameters (weights) so that the agent maximizes its expected cumulative reward across all possible trajectories (sequences of state-action pairs). This expected cumulative reward is typically denoted as the objective function $J(\theta)$, where θ represents the weights of the neural network.

$$J(\theta) = E_{\tau\sim\pi_\theta}[R(\tau)] = \sum_{\tau} p(\tau|\pi_\theta)R(\tau) \quad (4.1)$$

As a reminder:

- π_θ is the policy determined by the network
- $R(\tau)$ is the total reward accumulated along trajectory τ
- The expectation E is taken over all possible trajectories according to the probabilities induced by the current policy and environment dynamics.
- $p(\tau\,|\,\theta)$ is the probability of the trajectory τ occurring when the agent follows the policy π_θ

To perform this optimization using gradient-based methods such as gradient descent, we need to compute the gradient of the objective function with respect to the network parameters, written as $\nabla_\theta J(\theta)$. This requires that $J(\theta)$ is a differentiable function of θ. In other words, the expected reward must change in a continuous and smooth way as the neural network's output changes—because the network output directly determines the policy, which in turn affects the distribution of actions and resulting rewards.

If the output of the network affects the reward in a discontinuous or non-differentiable way, we cannot compute meaningful gradients, and standard backpropagation will fail. Therefore, to make gradient-based optimization feasible, the expected reward must be a continuous and differentiable function of the neural network's output (and thus its weights).

For weights vector θ, our gradient would be as follows:

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_{\tau} p(\tau|\pi_\theta)R(\tau) = \sum_{\tau} \nabla_\theta (p(\tau|\pi_\theta)R(\tau)) \quad (4.15)$$

$$\nabla_\theta J(\theta) = \sum_{\tau} \nabla_\theta p(\tau|\pi_\theta)R(\tau) + p(\tau|\pi_\theta)\nabla_\theta R(\tau) \quad (4.16)$$

To improve the agent's performance over time, we aim to increase the expected cumulative reward it receives across all possible trajectories. For learning to occur, the neural network's output must directly affect the expected return. In other words, the gradient guides the network to maximize $R(\tau)$.

In reinforcement learning, rewards are typically delayed and depend on the sequence of actions taken, making it challenging to directly associate individual actions with their outcomes.

Additionally, environmental transitions often involve randomness, making it difficult to predict subsequent states from state-action pairs. For these reasons, we cannot simply hand-derive a continuous function mapping individual actions to future rewards. This shows the necessity for using machine learning, specifically neural networks, to create this continuous function mapping state-action pairs to expected rewards.

This means that we simultaneously learn the reward function, as well as a deterministic policy which directly maximizes that reward function for estimating expected reward. The action generation network is the **actor** and the action evaluation network is the **critic**. This paradigm is known as an **actor-critic** RL algorithm.

This means that we have a continuous function to take the gradient of with respect to our actor parameters, and that the probability of an action taken is a constant equalling 1. This is because our policy is deterministic. Since the probability is a constant 1, its gradient is 0, and we can remove the first product. Since 1 times any number is the number itself, we can remove it from the second product. This refines our objective function as follows:

$$\nabla_\theta J(\theta) = \sum_\tau \nabla_\theta p(\tau|\pi_\theta) R(\tau) + p(\tau|\pi_\theta) \nabla_\theta R(\tau) = \sum_\tau \nabla_\theta R(\tau) \tag{4.17}$$

By the Monte Carlo approximation, we can estimate the expectation over trajectories using a sample average of N sampled trajectories τ_i. Furthermore, since the environment satisfies the Markov Property, then the current state contains all the necessary information to predict future outcomes—meaning the future is conditionally independent of the past given the present state. This allows us to reason about expected returns based on individual state-action pairs, rather than needing to explicitly model or analyze entire trajectories at once. As a result, we can use single-step experiences (state, action, reward, next state) to update our estimates of the policy or value function. This means we do not have to wait until the end of the episode to perform training.

$$\nabla_\theta J(\theta) = \sum_\tau \nabla_\theta R(\tau) \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta Q_\phi(a^i_t, s^i_t) \tag{4.18}$$

Since our actor function produces the action given the state, this equation refines to:

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta Q_\phi(\pi_\theta(s^i_t), s^i_t) \tag{4.19}$$

By the multivariable chain rule, this equals:

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\pi_\theta(s^i_t)} Q_\phi(s^i_t, \pi_\theta(s^i_t)) * \nabla_\theta \pi_\theta(s^i_t) \tag{4.20}$$

$$= \frac{1}{N} \sum_{i=1}^{N} \nabla_{a^i_t} Q_\phi(s^i_t, a^i_t) * \nabla_\theta \pi_\theta(s^i_t) \tag{4.21}$$

We want this method to support **off-policy learning**, meaning it should be able to learn from experiences collected using earlier versions of the policy (i.e., older actor networks), not just the current one. This is important for improving sample efficiency and avoiding recency bias—where only the most recent experiences are considered, potentially causing the agent to 'forget' valuable past experiences. To enable off-policy learning, we primarily rely on the state transitions observed in one step experience tuples discussed earlier in this section (state, action, reward, next state). As long as the dynamics of the environment remain relatively unchanged between when the experience was collected and when it is used for training, the observed transition from a state to subsequent state after taking an action remains valid. Moreover, the immediate reward associated with this transition is fixed and does not degrade over time. Finally, even if the action a is no longer likely under the current policy, that does not significantly hinder learning. The experience still provides useful information about the environment's dynamics and the potential long-term return, which we can use to improve our policy. In this way, we can leverage past experiences—regardless of the policy that generated them—to update the current policy effectively.

However, estimating the Q-value of the optimal subsequent state-action pair relies on knowing which action is taken. To prevent the policy from learning based on outdated state-action pairs in Q-estimation, we re-run the subsequent state through our current actor function to generate the subsequent action when training. To ensure that our Q-estimates are up to date, we also use our current Q-function for estimating the Q-value of the experienced and theoretical subsequent state-action pairs.

The Q-function, the critic in this case, learns using experience replay with a replay buffer in the same way as in DQL since this method is off-policy. As a reminder, the loss function for the critic is as follows where π_0 is the policy when the experience occurred, π_θ is the current policy, and ϕ are the parameters of the critic function.

$$L(\phi) = \frac{1}{N}\sum_{i=1}^{N}\left(Q_\phi(s^i_t, \pi_0(s^i_t)) - \left(r^i_t + \gamma Q_\phi(s^i_{t+1}, \pi_\theta(s^i_{t+1}))\right)\right)^2 \tag{4.22}$$

$$L(\phi) = \frac{1}{N}\sum_{i=1}^{N}\left(Q_\phi(s^i_t, a^i_t) - \left(r^i_t + \gamma Q_\phi(s^i_{t+1}, \pi_\theta(s^i_{t+1}))\right)\right)^2 \tag{4.23}$$

Note that the critic and actor often uses a double architecture for evaluating the subsequent state-action pair and choosing the subsequent state's action, respectively as in double DQL.

Since this algorithm uses a deterministic policy—meaning it always outputs the same action for a given state—the agent can become overly exploitative of its current knowledge. This often leads to premature convergence to a suboptimal policy, especially in continuous action spaces where exploration is more challenging. Deterministic policies inherently lack

randomness, so without additional mechanisms, the agent may fail to sufficiently explore the action space.

Although this is an off-policy algorithm and uses experience replay to learn from past interactions, this alone does not guarantee sufficient exploration. To address this, action noise/randomness is added to the deterministic policy's output during training. The injected noise introduces stochasticity into the agent's behavior, allowing it to explore a broader range of actions rather than always selecting the perceived optimal action. This helps the agent avoid getting stuck in local optima and improves its ability to discover higher-performing strategies.

The noisy action is executed in the environment, stored in the replay buffer, and used to update both the actor and critic networks. Over time, as the policy improves, the level of noise can be reduced to shift the focus from exploration to exploitation.

$$a_t = \pi_\theta(s_t) + \epsilon_t \tag{4.24}$$

To model the fact that we want better actions to have higher probabilities, and facilitate easy calculation of the probabilities of actions, our action noise PDF should be a Gaussian Distribution. We assume the action values are independent, meaning that the output distribution merely requires a mean and standard deviation for each value in the action vector, then samples from the joint Gaussian distribution.

As the agent learns and becomes more confident in its policy, the noise is typically decayed over time to reduce the level of exploration. This means that at the beginning of training, the agent will explore more, but as training progresses, the noise decreases, and the agent will rely more on its learned policy.

This gradual reduction in noise helps the agent balance exploration and exploitation over time:

- **Exploration** occurs when the noise is high and the agent is trying new actions.
- **Exploitation** happens when the noise is low and the agent relies more on its deterministic policy.

This overall RL method is known as **Deep Deterministic Policy Gradient (DDPG)** (Lillicrap, 2015). As of the writing of this textbook, DDPG and its variants remain the primary uni-agent RL algorithms for robots in continuous action spaces.

Chapter 5: Task Execution

5.1 - Reactive Control

As discussed, robots take in sensor data, process it, and act in a manner designed to achieve the goals set by their operators. In this framework, actuation is fundamentally a function of the input data we define, meaning the robot responds directly to current sensor readings to fulfill its objectives. This reactive approach involves the robot initiating its actions, following a defined function based on the sensor input, and ultimately reaching a stopping condition when a specific criterion is met. This process is known as **reactive control** (Brooks, 1986).

One example of reactive control is a robot following a light source. Robots equipped with light sensors can move toward a light source. The motor speeds are adjusted based on sensor readings, causing the robot to follow the brightest light. The response is immediate and does not involve any memory of previous actions. Another example is obstacle avoidance. Robots using proximity sensors (like ultrasonic or infrared) can detect nearby obstacles and adjust their path accordingly. For instance, if an obstacle is detected, the robot can steer away from it by modifying the speed of its wheels based on the sensor readings. An additional example is a robot meant to navigate to a set pose then stop upon attaining that pose.

5.2 - Finite State Machines (FSM)

Reactive control excels in straightforward tasks that do not necessitate complex planning. However, many advanced robotic tasks require actions dependent on past experiences. Reactive control is typically ineffective for tasks that involve switching between different operations based on the robot's state and environmental conditions or for those requiring a sequence of actions to achieve more complex goals. For example, assembling components in a specific order requires planning and execution beyond immediate sensor inputs, as the robot must remember and follow the correct steps throughout the task. Similarly, a robotic arm tasked with stacking blocks needs to recall the heights and positions of previously stacked blocks to ensure stability. If it experiences a collapse in a specific configuration, it can learn to avoid repeating that mistake by adjusting its stacking strategy. Search and rescue robots that locate, navigate to, pick up, and transport injured individuals must adapt their actions to each of these four scenarios. Similarly, robotic vacuums may remain idle until activated by a user command to clean or be prompted by a low battery to return to the docking station. When cleaning, they may continue until the area is finished, at which point they can either return to the docking station or clean another area, depending on whether all areas have been addressed.

The robot’s overall objectives can be broken down into discrete, actionable units called **tasks**. Each task represents a self-contained operation the robot performs to achieve a specific subgoal. A task is typically governed by a function or algorithm that activates when a starting

condition is met and continues until a termination condition is satisfied. For instance, a task might begin when an obstacle is detected and end once the robot has successfully navigated around it.

Most tasks map sensor inputs—such as camera data, LiDAR readings, or proprioceptive signals—to a sequence of actions. In some cases, historical data may also be considered, such as previous velocity commands or positions. For example, an object-following task might process real-time vision data to continuously update movement commands. However, not all tasks require actuation. Some tasks may be passive, such as data logging, mapping, or processing sensor information to generate a plan (e.g., path planning or environmental modeling).

The most basic form of a task, known as an atomic task, is a single function that maps sensor inputs (and possibly historical data) directly to an action, without any internal decomposition into smaller sensor-to-action mappings. Atomic tasks do not involve nested decision-making or sequential sub-tasks, making them easier to design and plan for, as they typically involve a narrow and well-defined set of states with clearly actionable outcomes. For instance, the task 'turn right when an obstacle is detected within 0.5 meters ahead' qualifies as atomic—it is triggered by a simple sensor condition and results in a direct motor command. While many complex tasks can theoretically be decomposed into smaller sub-tasks, it is often suboptimal to do so, since certain algorithms are designed to handle multi-step or multi-goal behavior more efficiently when treated as a single integrated task.

Since, in these situations, the way the robot reacts to the sensors depends on its current task, environmental context, and/or progress in a sequence of goals, it must keep track of such data. This is known as the **task-state** of the robot. Each task-state is defined by a unique specific function for reactive control based on the robot’s sensor data.

In these situations, the robot's reactions to sensor data depend on its current task, environmental context, and progress in a sequence of goals. To effectively navigate these complexities, the robot must keep track of this contextual information, which is referred to as the robot's state.

Since the task or goal may change depending on the environment, state of the robot, or completion of objectives, the robot may transition from one state to one or more other states. This means that each state will have specific criteria for it to transition to each state that it can transition to. These criteria may include relevant variables of the current state of the robot and its environment as well as data from the past such as which objectives have been completed.

For example, a vacuum cleaner may have the following plan for task execution:

- **Starting the Cleaning Cycle:**
  - The user activates the robotic vacuum through an app or button. This command transitions the vacuum from the Idle State to the Cleaning State.
- **Navigating the Environment:**

    - While cleaning, the vacuum continuously gathers sensor data, detecting the surrounding environment. It employs various sensors (e.g., infrared, ultrasonic) to identify obstacles.
    - If it encounters a chair, it enters the Obstacle Avoidance State, where it calculates a new path around the obstacle to continue cleaning.
- **Battery Management:**
    - As the battery level decreases, the vacuum's sensors monitor this status. If the battery falls below a predefined level, the vacuum transitions to the Docking State and heads back to its charging station.
- **Spot Cleaning Request:**
    - If the user identifies a specific area for extra cleaning (e.g., a spill), they can issue a command through the app. The vacuum transitions to the Spot Cleaning State, performs a focused cleaning task, and then resumes the Cleaning State or returns to Docking based on battery status.

This construct, defined by a finite number of states, the possible states each state can transition to, and the criteria for each state to transition to each other state, staying in its own state, is known as a **Finite State Machine (FSM) (**Mealy, G. H., 1955).

FSMs are often represented in diagrams as graphs with nodes being states and edges being transitions. This is shown below for a light following robot:

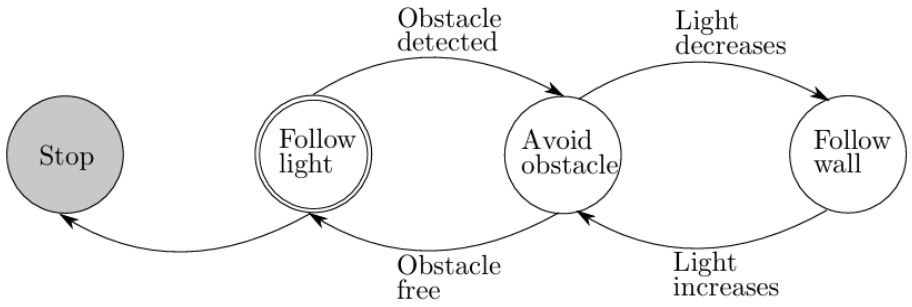


Fig. 5.1: A pictorial representation of a potential FSM for a light-following robot. “Follow Light” is the starting state and potential ending state.

In FSM diagrams, a double line circle represents a starting state, and a gray circle represents an ending state.

5.3 - Hierarchical Finite State Machines (HFSM)

Implementing a FSM can become inefficient and difficult to maintain when the system involves a large number of states. This is because in a regular FSM, each state is conceptualized and implemented as an entirely separate entity, even if many states exhibit similar behavior or

share transition conditions. In practice, different states often respond to the same events in the same or similar ways—for example, a robot might respond to a low battery signal the same way whether it is navigating, idling, or mapping. However, standard FSM implementations do not inherently support abstraction or inheritance between states. As a result, developers are forced to duplicate the same logic across multiple state definitions, which leads to bloated, repetitive code and increased chances of bugs or inconsistencies. This not only makes the system harder to scale or extend but also significantly increases the cost of development and maintenance. Recognizing and factoring out these behavioral similarities is essential for creating more efficient and modular task control systems.

For instance, if multiple states share the same sensor processing, transition conditions, or general behaviors, implementing these redundantly in a regular FSM is inefficient. When transitioning between states, we don't need to completely redefine how the robot behaves for each new state. Instead, we only want to modify the specific code and methods associated with the aspects that change between states.

This means that if two states share transitions to another state based on the same criteria, we should not implement this twice. For example, several states might transition to "Obstacle Avoidance" when the robot detects an obstacle. Rather than repeating this logic in each state, it’s more efficient to handle it once and apply it wherever necessary.

This means that aspects of information processing, decision-making, and state transitioning should be modularized and reused when necessary. States that share common behaviors or decision logic can be grouped together, while those with distinct logic remain separate. Within these groups, there can be even more specific subgroups that share particular modules.

For instance, consider the 'Movement' superstate, which encapsulates various navigation-related behaviors. Within this superstate, we might have the 'Pursuit' substate, which can be further divided into:

- **Obstacle Avoidance**: The robot detects and avoids obstacles while pursuing the target.
- **Target Tracking**: The robot continuously updates its pursuit based on the target's position.
- **Engagement**: The robot reaches the target and engages, either by stopping or performing a specific action.

When navigating a maze, the robot may switch between different movement strategies, such as turning left or right, based on sensor data, like the proximity to a wall. However, the overall transition between higher-level tasks—such as moving from ‘Exploring’ to ‘Obstacle Avoidance’—is typically governed by the logic established at the superstate level, such as the detection of a significant obstacle.

This hierarchical organization leads to a structure known as a **Hierarchical Finite State Machine (HFSM)** (Harel, 1987). In an HFSM, a higher-level state that encapsulates shared behaviors or logic is referred to as a 'superstate.' The individual states within that superstate, which share these common elements, are known as 'substates.' This design offers greater flexibility, efficiency, and maintainability in complex systems, as shared behaviors can be centralized and reused across multiple substates, reducing redundancy.

Each state with substates categorizes its substates based on shared processes, allowing for streamlined transitions and behavior management. For example, in a robot navigation system, a superstate labeled 'Movement' may include substates like 'Pursuing Target,' 'Avoiding Obstacles,' and 'Patrolling.' Each of these substates can utilize common sensor processing and decision-making algorithms, promoting modularity. Conversely, states without substates define the reactive control specific to that state, such as the behavior of the robot when it encounters an unexpected obstacle. This approach not only simplifies the overall design but also enhances the system's adaptability to varying contexts and requirements.

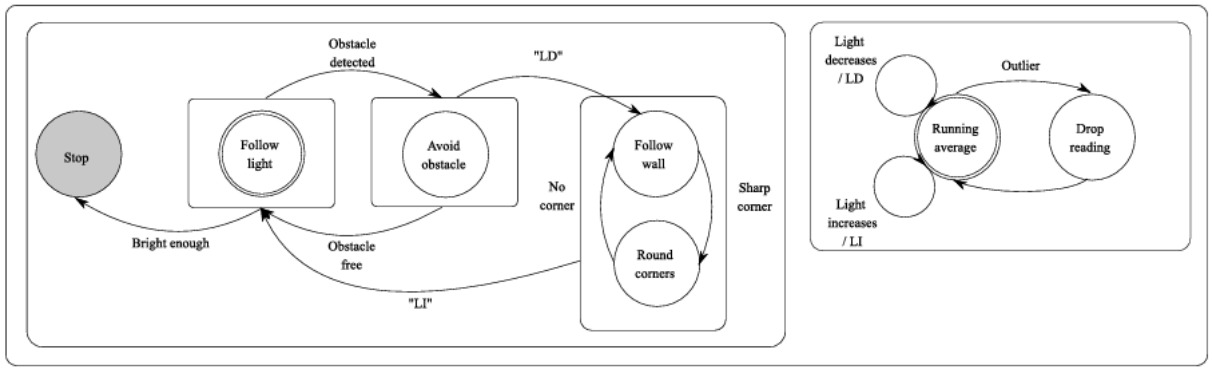


Fig. 5.2: A pictorial representation of a HFSM for a light following robot with different levels of state categories nested within each other

In practice, HFSMs are implemented as distinct processes that operate independently and asynchronously. These processes can communicate using an inter-process communication (IPC) framework, which facilitates the sharing of data structures between processes running on the same or different computers through a networking interface.

Numerous IPC frameworks specifically target robotics, offering abstractions for robot-specific data structures such as coordinate frames and video streams. These frameworks also provide associated tools for managing these data structures and language bindings that enable different components to publish and subscribe to information effectively. A prominent example of such a framework is the Robot Operating System (ROS), which supports the development of modular and distributed robotic systems.

Section III: Localization

Chapter 6: Sensor Fusion

6.1 - Sensor Values as a Probability Distribution

In different situations, the same action or sequence of actions may lead to varying results. This means that the actions which maximize the reward function differ depending on the context. For instance, factors like the machine's location and velocity, its proximity to nearby obstacles, and the type of terrain it's navigating on all influence the effects of specific actions. To determine the action with the highest expected reward, it is necessary to gather relevant data. While perfectly predicting the outcomes of different actions requires extremely comprehensive data, we can often make accurate estimations based on key variables. To collect and store this data, the machine relies on electromechanical devices that measure specific physical attributes of the environment—these are known as **sensors**. At any given moment, the data we read from these sensors is referred to as an **observation (z)**, with the full range of possible observations known as the **observation space**. For example, an observation from a machine moving wood pallets might include its position from wheel encoders, its angular velocity from an IMU, a point cloud from a LiDAR sensor, and the weight of the pallet from a strain gauge or load cell. Similarly, a self-vacuuming robot like a Roomba uses infrared and bump sensors to detect walls, furniture, and dirt. A self-driving car, on the other hand, processes camera images, LiDAR data, and GPS to detect lane markings, nearby vehicles, and pedestrians.

In an ideal scenario, sensors would provide the exact value of the quantities they are designed to measure. However, interactions with the physical environment and inherent noise in sensor readings introduce imperfections. As a result, sensors only provide an approximation of the true value. This true value is referred to as the **ground truth observation ($\hat{z}$)**.

Since observations involve physical quantities, they are numerical and continuous. This means that the observed value is simply the ground truth added to the difference between the observation and ground truth, known as the error (**e**). For example, if the observed temperature is 34 degrees Celsius, and the true temperature is 33 degrees Celsius, there is an error of 1 degrees Celsius. This is represented as below:

$$e = z - \hat{z} \tag{6.1}$$

For each reading, the error may be 0 (perfect measurement), positive (overestimation), or negative (underestimation). The magnitude of the error represents the severity of the deviation.

Since error varies between sensor readings and the exact error for an individual reading is generally unknown, it is typically viewed as a random variable. For this reason, it is generally impossible to eliminate all errors in a sensor. For each sensor, the error will tend to concentrate in certain areas over others in a way that remains relatively stable over time. This means that sensor

readings can be viewed as probability distributions, with each individual reading acting as a sample from that distribution.

Since regression tasks involve continuous probability distributions, the probability of observing any exact specific value of x or e is infinitesimally small—effectively zero—due to the infinite number of possible values. However, in robotics, we generally care about ranges of values, such as greater than some threshold or inside some margin of error. Additionally, for tasks such as figuring out predictions for a value or likely errors of a prediction, it is mainly important to consider probabilities of values relative to each other. For this reason, we can use a distribution of probabilities that show relative probabilities of individual values, with the exact probability of individual values being 0, and the probability of ranges being calculated by analyzing the relative probabilities in that range. This is known as a **Probability Density Function (PDF)**, with probability density denoted as **P** in equations, whereas exact probability of a range or discrete values is denoted as **p**. The PDF measures how likely it is for the values to fall within a small interval around a given point, representing the density of probability. For a continuous variable, this interval is given by dx. By definition, a PDF must be non-negative at all inputs, and integrate to 1 over the relevant input range (usually negative infinity to positive infinity). In this paradigm, the integral of the PDF from input a to input b yields the probability that a given value falls within that range. These definitional elements are mathematically described below:

$$p(a < x < b) = \int_{a}^{b} P(x)dx \tag{6.2}$$

$$\int_{-\infty}^{\infty} P(x)dx = 1 \tag{6.3}$$

$$P(x) \geq 0, \; for\; all\; x \tag{6.4}$$

Each sensor value is treated as a sample from this PDF. According to the Law of Large Numbers (LLN), the probability of sampling a specific error value approaches the proportion of all gathered values that have that error value as the number of samples increases. Essentially, as the number of samples grows, the sampled distribution of errors converges to the true distribution of errors. For example, imagine one is sampling values from a Gaussian distribution with a mean of 0 and standard deviation of 1. As the number of samples approaches infinity, the probability distribution of sampled values approaches a Gaussian distribution with a mean of 0 and standard deviation of 1.

The type of distribution that best represents the measurement error and sensor value depends on various factors, including the sensor's characteristics, the environment where measurements are taken, and the nature of the phenomenon being measured. Since most noise is the result of many small causes of noise added together, the most common distribution used is the **Gaussian distribution**, also known as the normal or bell curve. This is shown

mathematically to be probable with a sufficient number of individual additive causes of error by the Central Limit Theorem. However, if this is not the case, the distribution might differ, such as being heavy-tailed, uniform, multimodal, Poisson, etc. Researchers need to determine the appropriate distribution for a sensor in its specific environment to accurately interpret and extract useful information from the data.

6.2 - Sensor Calibration

We want our sensor readings to be as accurate as possible. This means that the observed value does not consistently overestimate or underestimate the true value. In other words, errors are entirely random and have no systematic bias (unbiased). This means that they do not tend to overestimate or underestimate the true value. This entails that e is centered on 0, meaning that z is centered on $\hat{z}$. This also means that the ground truth should be the value that minimizes error in sensor values. This entails two properties of our sensor readings for a specific ground truth:

- The first property is that the average of a series of samples of z converges to $\hat{z}$. As you obtain more and more samples, the Law of Large Numbers (LLN) entails that the proportion of all samples equal to a specific value converges to the probability of sampling that value. This means that the average value of a PDF, given by the sum of all values weighted by their respective probability density, should equal the ground truth. The value that does this for a probability distribution is known as the **expected value** or **mean** and denoted as E. This means that our sensor PDF should have the following properties:

$$E[e] = 0 \tag{6.5}$$

$$\hat{z} = E[z] = \int_{-\infty}^{+\infty} zP(z)dz \tag{6.6}$$

- The second property is that the ground truth should be the optimal predictor for the value of a single sample. The optimal value is the one that minimizes overall prediction errors. To assess this, we calculate the deviation of each sample from the ground truth. By summing these deviations and taking the average, we obtain a representative measure of the error. However, to avoid sign dependence, where the direction of the deviation affects the result, and to focus solely on the magnitude of the error, we square the differences before summing them. This means that the ground truth should be the value that minimizes the expected square of the error, also known as the **variance**. This value is described mathematically below:

$$f(E[z]) = E[(z - E[z])^2] = \int_{-\infty}^{+\infty} (z - E[z])^2 P(z)dz \tag{6.7}$$

Since the variance is a function of the predictor, our task simplifies to finding the minimum of this function. Since the variance of a predictor is going to increase the farther the predictor is from the optimal predictor, the error is **convex**. This means that there is a local minima in the error function, and it is the function's only minima. This means that the optimization can be achieved by setting the partial derivative of the variance with respect to the predictor equal to 0 and solving for the optimal predictor. This approach is described mathematically below:

$$\frac{\delta}{\delta \hat{z}}\left(\int_{-\infty}^{+\infty}(z-\hat{z})^2 P(z)dz\right) = 0 \tag{6.8}$$

$$\hat{z} = E[z] = \int_{-\infty}^{+\infty} zP(z)dz \tag{6.9}$$

This means that through both derivations we get the same value for the ground truth:

$$\hat{z} = E[z] = \int_{-\infty}^{+\infty} zP(z)dz \tag{6.10}$$

When a sensor is first created, it often lacks the accuracy needed for precise measurements. To address this, the sensor must be tuned and calibrated such that the expected sensor reading is the ground truth. Calibration typically involves either hardware adjustments or mathematical corrections.

To assess the measurement error for a given reading, we need to know the ground truth. During sensor calibration, this typically involves subjecting the sensor to various tests and comparing its readings to known ground truth values. For instance, we might use a ruler to measure the distance of an object from an ultrasonic sensor and compare these measurements to the sensor's reported values. This comparison helps us gauge the error and determine the type of distribution that describes it.

Note that depending on the sensor, the error value may be expressed as a raw quantity or a percentage. For some sensors, this value remains constant across different ground truths, while for others, it varies with the ground truth. For instance, the variance of an ultrasonic sensor often increases as the distance from the sensor grows. In this situation, the error will be a function of the ground truth.

When calibrating a sensor or using sensor readings to estimate the ground truth, we do not know the PDF of the state—we only obtain samples from it in the form of sensor readings. As a result, we must approximate the true PDF using the sampled distribution. Sensor calibration should therefore ensure that the expected value of the sampled distribution matches the ground truth. Likewise, when estimating values from sensor data, we rely solely on samples, so we must be able to estimate the expected value based on these samples.

Since we no longer are working with a continuous x, we cannot use integration to find the expected value. As we know from calculus, integration is merely an infinite sum as described below:

$$\int_a^b f(x)dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i)\Delta x_i \tag{6.11}$$

$$\int_{-\infty}^{\infty} zP(z)dz = lim_{n\to\infty} \sum_{i=1}^{n} zP(z)\Delta z \tag{6.12}$$

As mentioned above, from our PDF, $P(z)\Delta z$ for a small interval is simply the probability of obtaining z. This refines our formula as below:

$$E[z] = lim_{n\to\infty} \sum_{i=1}^{n} zp(z) \tag{6.13}$$

If we summed over an infinite number of z values, the result would converge to the true expected value. However, in practice, we can only take a finite number of samples. Since this sum does not include all possible z values, the total probability may not equal one. This violates a fundamental property of probability distributions and causes the result to be scaled downward. To correct for this, we must normalize by dividing by the total probability mass of the sampled observations. This leads to the refined formula below:

$$E[z] = \frac{\sum_{i=1}^{n} z_i p(z_i)}{\sum_{i=1}^{n} p(z_i)} \tag{6.14}$$

If we have some way of estimating the probability mass of our samples then we can directly calculate the above equation to find the expected value. However, in calibration we often do not have such prior knowledge. In this situation, we must assume that each sample has the same probability. This refines our expected value equation to the unweighted mean as follows, where N is the number of samples:

$$\hat{z} = E[z] = \frac{1}{N}\sum_{i=1}^{n} z_i \tag{6.15}$$

Alternatively, we could find the optimal predictor of a sensor reading. Since we have a finite number of readings, each sample has an equal probability, so our error formula to minimize is mathematically described as below:

$$MSE = \frac{1}{N}\sum_{i=1}^{N} (E[z] - z_i)^2 \tag{6.16}$$

This formula is known as **Mean Squared Error (MSE)**. Upon minimizing this with respect to E[z] as above, we obtain the following:

$$\hat{z} = E[z] = \frac{1}{N}\sum_{i=1}^{N} z_i \tag{6.17}$$

Note that if each sample had some known probability, then E[z] would be mathematically described as follows:

$$\hat{z} = E[z] = \frac{\sum_{i=1}^{N} z_i P(z_i)}{\sum_{i=1}^{N} P(z_i)} \tag{6.18}$$

For a properly calibrated sensor where our ground truth equals our expected value, MSE is the same as variance.

As with the continuous case, both discrete expected value formulas are the same. This shows that our ground truth for some distribution can be obtained by approximating the continuous PDF with a series of samples, and either minimizing MSE from discrete samples of the distribution, or by approximating integration with summation.

This also shows that to estimate the ground truth for a variable, we must find the expected value of that variable. If we have one sensor measurement corresponding directly to each variable in our observation, then our predicted values are simply our observed values. This is described as follows:

$E[z] = E[z|z] = z$

6.3 - Introduction to Sensor Fusion

To ensure the machine selects optimal actions, we need a function or algorithm that maps observations to actions. However, much of the relevant data for determining the best action may not be immediately observable due to technological or situational limitations, a concept known as **partial observability**. First, observations are instantaneous snapshots, whereas states often incorporate temporal context (e.g., velocity is derived from changes in position over time). For instance, a sensor cannot predict where a moving object will be in five seconds. To make use of past insights or forecast future outcomes, machines must maintain a model of how the environment evolves. Additionally, some data may involve human-made labels or categories that do not correspond directly to physical quantities in the real world but derive meaning from their functionality. Examples of this include determining whether an area is occupied by an obstacle or whether a fruit is an apple or an orange. Lastly, some information may be theoretically observable but cannot be directly observed due to limitations in our sensors. For example, an IMU provides data on linear acceleration, angular velocity, and orientation. To derive position and linear velocity, additional mathematical operations are required.

Additionally, some data has the potential to be represented in a simpler form that is easier for the system to use in decision-making. Observations can be high-dimensional (e.g., images with millions of pixels or points), particularly with camera and LiDAR data. However, these

large data sets can often be represented more efficiently with simplified, lower-dimensional states, which make action calculation computationally feasible. For example, the type of object detected or whether a region is occupied by an obstacle.

Due to fundamental physical laws, such as kinematics and dynamics, and the inherent relationships within the system being measured, there are often true equations that map the physical quantities making up an observation to the relevant state variables. For example, in a car, the relationship between the motor's rotation and the car's forward movement is defined by these physical laws. By understanding these principles, we can derive mathematical functions that relate measurable inputs to the desired relevant state variables. As another example, an ultrasonic sensor measures the distance to an obstacle by emitting ultrasonic waves and calculating the time it takes for the waves to return after bouncing off an object. However, the sensor typically does not directly provide the distance. Instead, to estimate the distance, we use our best estimates for relevant inputs and apply them to the appropriate function relating the quantities. This is described mathematically below:

$$x = f(z) \tag{6.19}$$

Overall, to process information in a format that is usable for planning algorithms, machines must employ techniques to extract meaningful insights from the data they observe. An individual vector that represents the relevant information for decision making at a given point in time is known as the **state (x)**. For example, the state of a machine that moves wood pallets from one location to another could include the machine's position, velocity, the location of the nearest obstacle, a boolean indicating whether it is currently carrying a pallet, the pallet's type, data about the floor surface, and whether the target has moved in the last 10 seconds. Similarly, consider a self-driving car navigating through a city. The state could include the car's position (GPS coordinates), speed, steering angle, the distance to the closest vehicle or pedestrian, the status of traffic lights (red, yellow, green), the car's current lane, road conditions (e.g., wet or dry), the car's battery level, and the time of day. The set of all possible states is known as the **state space**.

We assume that changes in each relevant state variable will have physical effects that can be observed. For example, different machine velocities will affect IMU readings. This means that the sensor array of a machine must ensure that there are sufficient physical observations that are influenced by the state such that one can deduce the state from the observations.

This means the machine must use algorithms to map observations to states, a process referred to as **information processing** or **machine perception**. The function mapping observations to states is known as the **state estimation function**. Since all the data we have is our sensor reads, we must find the expected value of our state given our sensor reads. This refines our state estimation to the following:

$E[x|z]$

If we had a single observation value per input to our state estimation function, then we could simply pass the observations through the state estimation function to get the state estimation. This is described mathematically below:

$$E[x] = E[f(z)] \tag{6.20}$$

However, trying to estimate the expected value from a distribution with only an individual sample from each input distribution leaves the result prone to significant errors. By the LLN, sampled distributions more closely approximate the distributions from which they are sampled as the number of samples increases. The expected value of a distribution given multiple samples has a lower variance than that of one sample. This means to get a more reliable estimate of state variables, one can get more samples from the continuous distribution of each observation variable.

For this reason, robots often have multiple of the same sensor. For example, a robot may have wheel encoders on each wheel, or multiple IMUs. Additionally, each sensor may take multiple readings. For example, an IMU might take multiple consecutive readings which all give insight into pose-related state variables. Lastly, a single state variable may influence multiple physical attributes. This means that different sensors with different observations can be used to deduce a state variable. For example, angular velocity can be gauged by an IMU as well as encoders. As another example, one may use an infrared sensor as well as an ultrasonic sensor. To minimize the effects of a weakness of one sensor, it is often beneficial to include different sensors from which the same state variable can be gauged.

This means that for a given state variable, there may be multiple observed values that are mutually exclusive as the ground truth. For example, an ultrasonic sensor reads that the nearest object is 4 cm in front of the robot, while an infrared sensor reads that the same quantity is 5cm. These two cannot both be true. Additionally, integrating the acceleration of an IMU may provide a velocity in the x direction of 1.3m/s, while analyzing the change in x position over time from a GPS may yield 1.1m/s. This means that we cannot directly pass the observed values through the state estimation function, and must instead directly find the expected value of each state variable given the observations influenced by that state variable. However, we may also have a different certainty in each of these values, meaning that some are more indicative of the ground truth state variable than others. This means that we must find some mathematical method of 'fusing' the observed sensor values influenced by a state variable to gauge the ground truth of that state variable. This is known as **Sensor Fusion**. Note that since we assume each sensor is calibrated, we can assume that each individual sensor will be unbiased and that we are aware of the probability distribution of each sensor for a given ground truth.

In order to properly fuse this sensor data, we must know how much to weigh or believe different observations or contributions to a function output. For example, imagine we get two observations, one telling us that our velocity is 4m/s and the other telling us that our velocity is

6m/s. If we have a stronger belief in the accuracy of the first sensor, our estimate should be closer to 4m/s, but not exactly 4m/s because the 6m/s estimate still provides relevant information. This means that we must be aware of the different probabilities, certainties, uncertainties, etc. of different observations or estimates. We can then find the expected value by minimizing estimate uncertainty or finding the center of probability density.

6.4 - Recursive Estimation

As shown before, we initially start out with some state estimation. Then, we must get samples to update our distribution for x. However, we very rarely get a full view of the true PDF for x from one sample. So, we must take multiple samples. This means that the robot must use all previous observations to gauge the correct state. Even though past data may not be as relevant as current data, it still provides some insights in most situations.

Robots generally sense data and actuate/alter their current configuration accordingly at intervals. For example, the control of a mobile robot or robot arm which processes data and issues updated control commands at fixed intervals (e.g. every 100 ms). These involve discrete inputs where the input is instantaneous but happens at discrete time intervals leading to discrete output. These systems can be modeled as a series of discrete steps, where state and output are functions of the time step k. Then the next state becomes the current state and we continue. This is called a **Discrete-Time System**.

In real-world systems, data arrives one measurement at a time rather than all at once. Unlike an offline batch problem—where you gather every sensor reading before processing—robots must act in real time and can't wait indefinitely for more data. For example, a GPS-guided robot can't pause for 100 location fixes between each wheel rotation; it has to move based on the information it already has. This shows that the robot must continuously maintain a belief about the true state throughout operation. At time k, you only have access to observations z_1, z_2,…,z_k, not any future z_{k+1}, z_{k+2},… Therefore, after each new measurement you must update your state estimate using all previous data.

This is mathematically described as below:

$E[x|z_{1:k}]$

As mentioned in prior chapters, proper state design entails that all relevant information is captured by the state. If there was relevant past information, it would simply be incorporated into the state. This means that past states do not provide additional information. This means we only have to analyze our most recent state to predict subsequent states and determine the optimal action. This is known as the **Markov Property**.

When estimating our value at time step k, x_k, we already have our current state prediction x_{k-1}. Due to the markov property, all prior states, $x_{1:k-1}$ provide no additional relevant information. Since x_{k-1} was generated by analyzing $z_{1:k-1}$, it fully encapsulates that data. The only new

information we get since our x_{k-1} prediction is z_k. This means that to determine our current state estimation we only need data regarding our previous state estimation and current observation.

This refines our sensor fusion goal to the following:

$E[x|z_k, x_{k-1}]$

This is known as **recursive estimation** or **recursive filtering**.

6.5 - State Space Modeling

In robot systems, we often deal with continuously evolving quantities as well as discreetly evolving quantities. This means that the ground truth changes in between sensor readings. For example, a moving robot's position will not be the same at consecutive IMU readings. In a discrete-time system, each subsequent step is simply the previous step with some change.

In these systems, our past state prediction cannot act as an initial starting point of fusion because it is outdated. However, past states often indicate future states because the state changes in predictable ways. For example, a robot moving at 5m/s will not likely be 20m away from where it was .1s ago. To optimize our current state prediction, we must be able to leverage our previous state prediction even when the state is changing/dynamic.

Usually the state contains information about how the state changes over time. For example, a state with velocity and position can leverage the velocity estimate to gauge how the position is likely to change. Additionally, changes state variables are generally dictated by commands given to the robot, known as **control inputs**. For example, the robot's planning programs may command the robot's acceleration, which affects its subsequent velocity. Since the robot determines these control inputs, sensor fusion programs have access to them. The control input of a system at time/step k is represented by a vector $\mathbf{u_k}$ of the input variables.

In most cases, the state, control inputs, and knowledge of environmental dynamics combined provide information to gauge the subsequent state, albeit with decreased certainty. This is especially true if there are high observation rates, as predictions decrease in accuracy as more time occurs between the last state estimation. This refines our sensor fusion goal in dynamic state systems to predicting our current state from our previous state, then fusing this prediction with our new observation. This model of how our state transitions over time inputting x and u is known as our **state-transition model** or **state-space model**, and is given by $f(x_{k-1}, u_k)$.

The way that control inputs and the current state map to the next state depends on the system's dynamics. If that underlying relationship itself changes over time, the model is **time-varying**; but in most applications it remains constant, so we treat it as **time-invariant**.

There will always be some noise in f, denoted as w_k, due to randomness or uncertainty in a system. For example, it might represent unmodeled effects, friction, or errors in control. w_k is known as the **process noise**. This can be described as below:

$$x_{k+1} = f(x_k, u_k) + w_k \tag{6.21}$$

The function is tuned and calibrated to ensure maximal accuracy as well as that the expected value of the entries w_k is 0.

$P(x_k|x_{k-1}, u_k)$ is a PDF centered on $f(x_{k-1}, u_k)$ with noise given by w. Through calibration, we can estimate the parameters of the noise distribution of w, thus finding the full distribution of $P(x_k|x_{k-1}, u_k)$.

This refines our sensor fusion goal to the equation below:

$$E[x_k|z_{1:k}, u_{1:k}] = E[x_k|x_{k-1}, z_k, u_k] \tag{6.22}$$

By using x_{k-1} and u_k and the uncertainties/probability changes introduced by the state space model, we can predict x_k and get the associated probabilities/certainties. This prediction is denoted as $x_{k|k-1}$. This refines our equation as below:

$$x_{k|k-1} = f(x_{k-1}, u_k) \tag{6.23}$$

$$E[x_k|x_{k-1}, z_k, u_k] = E[x_k|x_{k|k-1}, z_k] \tag{6.24}$$

This type of estimation is called **Dynamic Recursive Estimation**.

To solve this problem, we simply need a way to represent probabilities/uncertainties of each of these, and a mechanism to use this to find the expected value. We then must propagate our prior state estimate and uncertainties/probabilities to our prediction, then fuse the result with the observation.

As discussed above, we can find the expected value of such a filter from our observed samples by doing one of the following:

- Calculate the center of probability density weighted by value, using state and observation probabilities.
- Calculate the minimizer of mean squared error, using state and observation mean squared error.

6.6 - Particle Filter

Let's analyze the first situation above where we must use samples to directly approximate the true PDF, and calculate the center of probability density weighted by value, using state and observation probabilities. This method may be preferred as one not only knows the expected value, but can gauge the probability of different states. This allows us to compute additional statistics like the median, mode, or confidence bounds.

The expected value of the state given the previous state, control input, and observation is as follows:

$$E[x_k|z_k, x_{k|k-1}] = \int_{-\infty}^{\infty} xP(x|z_k, x_{k|k-1})dx \tag{6.25}$$

However, in many real-world systems, the relationships between variables are highly nonlinear and may not follow any simple, closed-form probability distribution. For example, the system dynamics or measurement models might involve complex, nonlinear functions that do not lend themselves to exact analytical solutions. In such cases, the resulting continuous probability distributions are difficult—or sometimes impossible—to describe analytically.

Additionally, many systems do not conform to the assumption of Gaussian noise. Non-Gaussian noise distributions can be skewed, heavy-tailed, or multi-modal, etc. Representing and manipulating these distributions continuously requires complex integrals and transformations that rarely have closed-form solutions.

Additionally, if the state is high-dimensional, these integrals become computationally expensive due to the curse of dimensionality. The number of function evaluations required for accurate numerical integration grows exponentially with the dimension of the state space.

Lastly, when the state transition or observation models are nonlinear, the integrals involved in the update do not simplify easily. This nonlinearity often means that the integrals cannot be solved exactly, forcing one to resort to numerical methods or approximations. In many cases, these numerical integrations are too slow or imprecise for real-time applications.

Overall, exact manipulation of continuous distributions would require the evaluation of complex integrals at each step of the filter. This means that we must approximate the result. Technically, we could perfectly approximate the PDF with infinite samples, but taking an infinite number of samples is impossible. So, we must approximate PDF with a discrete set of samples, each with an associated probability within the discrete system. Each value in the finite set of values we weight to approximate the true PDF is known as a particle.

As we know from calculus, integration is merely an infinite sum. When using a discrete set of samples, the integral form of expected value reduces to the discrete. Note that we use p (lowercase) to represent the probability mass that the state is actually the value of our sample state in our approximate discrete distribution of the continuous state PDF. If we have a discrete prior probability distribution for $x_{k|k-1}$, we can use to calculate our discrete distribution for x_k. This use of p (lowercase) is in contrast to our previous use of P (upper case) to represent probability density. In the discrete form of expected value, we merely take the weighted sum of possible states/particles (where the weight is the probability), then normalize by dividing this sum by the sum of weights as mathematically described below:

$$E[x_k|z_k, x_{k|k-1}] = \frac{\sum_{i=1}^{n} x_i p(x_i|z_k, x_{k|k-1})}{\sum_{i=1}^{n} p(x_i|z_k, x_{k|k-1})} \tag{6.26}$$

We do not have prior knowledge of probabilities of each of these particles. We cannot simply sample from the PDF or gauge probabilities from the PDF since the whole reason we must use a discrete set of particles is that we cannot obtain such a PDF at each step. This means

we must estimate the probabilities of each particle, then calculate the expected value accordingly. In other words, we must use our prior probability distribution of particles and z_k to estimate the probability of each particle in the distribution for x_k to approximate the true distribution for x_k and find the expected value. Such a filter is known as a **Particle Filter (PF)** or **Sequential Monte Carlo (SMC) Filter** (Gordon et al., 1993).

We often have multiple state variables to estimate, and multiple observed values per state variable. This presents a problem because it complicates gaining a presupposed model of $P(x_k|z_k,x_{k|k-1})$. This would entail looking at similar z values and seeing the likelihood of different x values are. This is difficult because we do not directly manipulate the sensor readings once the sensors are calibrated. All we can do is manipulate our environment. Additionally, sensor values do not influence the state, it is the other way around. Directly estimating $P(x_k|z_k,x_{k|k-1})$ often requires large amounts of data for each specific value of z to get an accurate conditional distribution. In practice, obtaining sufficient data for every condition of z can be challenging. The estimation of $P(x_k|z_k,x_{k|k-1})$ can be affected by sample variability, especially if z spans a wide range or if x has a complex relationship with z. So, we must refine $P(x_k|z_k,x_{k|k-1})$.

By the definition of conditional probability, the probability of observing dummy variable A given that we have observed dummy variable B is equivalent to the proportion of all times where B is correct/observed that A was also correct/observed. In our situation the probability of observing a specific x value given that we have observed a specific z, $x_{k|k-1}$ value is equivalent to the proportion of all times the z value was collected and we were previously as $x_{k|k-1}$ that the specific x value was also collected. This is described mathematically below:

$$P(x_k|z_k, x_{k|k-1}) = \frac{P(x_k, z_k|x_{k|k-1})}{P(z_k|x_{k|k-1})} \tag{6.27}$$

The numerator is the joint probability—given our prior state —that the true new state is x_k and that we observe z_k. The denominator is the marginal likelihood of seeing measurement z_k at all given our prior state.

Finding $P(x_k,z_k|x_{k|k-1})$ is oftentimes difficult to obtain because it requires covering the entire ground truth range and sensor range. Also, many of the same reasons above also apply for gauging $P(x_k,z_k|x_{k|k-1})$ as we still need to gauge different states for the same observation and different observations for the same state. So, we cannot directly use $P(x_k,z_k|x_{k|k-1})$. We need to decompose this such that when we gauge the probability of a state, the observation is given and vice versa.

To do this, we can use the conditional probability relationship for z_k given x_k, $x_{k|k-1}$ as follows:

$$P(z_k|x_k, x_{k|k-1}) = \frac{P(x_k, z_k|x_{k|k-1})}{P(x_k|x_{k|k-1})} \tag{6.28}$$

This is the same general formula as above with z_k and x_k switched.

By rearranging the above equation for $P(x_k,z_k|x_{k|k-1})$, then substituting that back in the 1st conditional probability equation, we get the following equation:

$$P(x_k, z_k|x_{k|k-1}) = P(z_k|x_k, x_{k|k-1})P(x_k|x_{k|k-1}) \quad (6.29)$$

$$P(x_k|z_k, x_{k|k-1}) = \frac{P(x_k, z_k|x_{k|k-1})}{P(z_k|x_{k|k-1})} \quad (6.27)$$

$$P(x_k|z_k, x_{k|k-1}) = \frac{P(z_k|x_k, x_{k|k-1})P(x_k|x_{k|k-1})}{P(z_k|x_{k|k-1})} \quad (6.30)$$

Note that since the Markov Property ensures that the observations are only dependent on the current state, not the past state, we can simplify the numerator as follows:

$$P(x_k|z_k, x_{k|k-1}) = \frac{P(z_k|x_k)P(x_k|x_{k|k-1})}{P(z_k|x_{k|k-1})} \quad (6.31)$$

This formula is known as **Bayes Theorem**.

The denominator is common to all possible current states. It is often referred to as the normalizing constant or marginal likelihood. It ensures that the posterior probability is properly normalized to ensure the posterior is a valid probability distribution, i.e., that the total probability sums to 1. Since it's the same for all states, it's often omitted when we are calculating the relative probabilities of the different states. If utilizing some method that relies on explicit probability distributions, we can compute the un-normalized posterior, and the normalization step (which ensures the total probability sums to 1) can be done afterward. This approach is computationally more efficient. This refines our equation to calculating the un-normalized probability distribution as follows, where $\propto$ is the proportionality symbol:

$$P(x_k|z_k, x_{k|k-1}) \propto P(z_k|x_k)P(x_k|x_{k|k-1}) \quad (6.32)$$

The term $P(z|x_k)$ is known as the Likelihood Function and represents the probability of observing a measurement given the state. Finding $P(z|x_k)$ is relatively simple as it does not tend to change heavily in different environments. Calibration directly involves manipulating the ground truth and observing the sensor's response, which makes it relatively straightforward to estimate the likelihood $P(z|x_k)$. By exposing the sensor to known states x and recording the resulting observations z, you can construct the likelihood function $P(z|x_k)$.

Since sensor values are typically influenced by the system state, it is often straightforward to establish a function that maps states to sensor values. This mapping is generally known as the **measurement model** or **observation model**. After calibration, this model can be used to predict the expected sensor readings for a given state in an unbiased manner, where each true state corresponds to an expected observation based on the observation model. This is given as below:

$$\hat{z} = h(\hat{x}) \quad (6.33)$$

The accuracy of this mapping depends on the quality of the sensor and the calibration process, as well as any potential noise or distortion in the measurements. If the noise follows a

specific distribution, calibration can be performed to figure out the parameters of these distributions. The observation centered on the observation corresponding to the state belief, with some noise that has a mean of 0 and follows some distribution. This is given as below for observation z, state belief x, and observation noise **v**:

$$v = z - h(\widehat{x}) \tag{6.34}$$

We do not have to account for any errors in our belief about x because it is our best accuracy and the likelihood term involves assuming x is correct as a given.

As mentioned above, through calibration we can estimate the parameters of the noise distribution of v, thus finding the full distribution of $P(z|x_k)$. Due to the Central Limit Theorem, we usually approximate v as Gaussian. Gaussian distributions are also computational simple, and can approximate to many other symmetric or slightly skewed distributions. This means that P(z|x) is a Gaussian distribution with mean h(x), and covariance matrix equaling the covariance matrix of v found via calibration. Note that if another distribution is assumed, then calibration can be used to find the relevant distribution variables. This covariance matrix, denoted as **R** in equations, is known as the **observation noise covariance matrix**. R is an m x m matrix, where m is the number of observation channels/variables. R is given as below:

$$R = E[vv^T] \tag{6.35}$$

$$R_{ij} = Cov(v_i, v_j) = E[v_i v_j] \tag{6.36}$$

Each diagonal element R_{ii} represents the variance of the noise for the i-th sensor or observation channel. Each off-diagonal element R_{ij} represents the covariance between the noise in the i-th and j-th channels. If the sensor noise is independent across channels, the off-diagonal terms are 0, making R a diagonal matrix.

This makes the distribution of P(z|x) for an m-dimensional observation vector, the multivariate Gaussian formula as follows:

$$P(z|x) = \frac{1}{\sqrt{(2\pi)^m |R|}} exp\left(-\frac{1}{2}(z - h(x))^T R^{-1}(z - h(x))\right) \tag{6.37}$$

In this situation, the difference between the observation and expected observation, z - h(x), is known as the **innovation**. h is known as the **observation function** or **measurement function**.

This means that we the expected value can be mathematically described as follows for n particles, and directly calculated:

$$E[x_k|z_k, x_{k|k-1}] = \frac{\sum_{i=1}^{n} x_i \frac{P(z_k|x_i)P(x_i|x_{k|k-1})}{P(z_k|x_{k|k-1})}}{\sum_{i=1}^{n} \frac{P(z_k|x_k)P(x_i|x_{k|k-1})}{P(z_k|x_{k|k-1})}} \tag{6.38}$$

Initially, we choose the number of particles that will represent the state distribution. If little or no prior information is available, particles are uniformly sampled across the state space within a predefined range. If prior knowledge about the initial state is available, particles are typically sampled from a Gaussian distribution centered on the expected initial state with a covariance matrix that captures uncertainty in the initial state. If the initial state is precisely known (e.g., from a highly reliable measurement), particles can all be initialized at the same location.

These initial particles represent the initial state probability distribution. As time progresses, the state evolves, meaning that after many iterations, many or all of the particles may no longer represent probable states. If a significant portion of the particles represents unlikely states, only a few particles will have significant weights, while most will have negligible weights. As a result, particles far from the likely state will have almost zero weight and contribute very little to the overall state estimate. These particles contribute to the computational intensity without contributing to the accuracy of the estimate. Additionally, if probable regions of the distribution are not well represented, the posterior estimate becomes inaccurate. This phenomenon of particles contributing little information to the posterior distribution due to state evolution is known as the **particle degeneracy problem**.

As the system evolves, the state of each particle should be updated according to the system's dynamics (described by a model of how the state changes over time). Since the probable states will essentially be the probable states that the probable past states transitioned to, we can simply propagate each part particle through the state transition function to get our new set of particles.

$P(z_k|x^i_k)$ represents the likelihood of the current observation given the current particle x^i_k. It measures how well the current particle explains the observation. $P(x^i_k|x_{k|k-1})$ represents the probability of a particle given the previous state. However, the previous state is represented by a series of particles, each with their own weight. This means that to calculate $P(x^i_k|x_{k|k-1})$, we must consider the probability that x^i_k is the state, given the set of prior particles. By the law of total probability, this is given as the sum that each prior particle's state would lead to a current particle's state, multiplied by the probability of the particle being the true previous state. This means that the expected value is formulated as below:

$$P(x^i_k|x_{k|k-1}) = \sum_{j=1}^{n} P(x^i_k|x^j_{k|k-1})P(x^j_{k|k-1}) \tag{6.39}$$

$$E[x_k|z_k, x_{k|k-1}] \propto \sum_{i=1}^{n} x_i P(z_k|x^i_k) \sum_{j=1}^{n} P(x^i_k|x^j_{k|k-1})P(x^j_{k|k-1}) \tag{6.40}$$

However, this calculation can be computationally inefficient. The inner summation involves evaluating the transition probability for each prior particle to each current particle. For n particles, this results in n^2 computations. This quadratic scaling becomes computationally infeasible for large particle sets, especially in real-time applications.

Each particle is generated exclusively from its corresponding predecessor based on the system's dynamics and control inputs. There is no interaction between different particles during the transition step, ensuring that particles evolve independently. Additionally, any potential that a particle may evolve from a different prior particle is almost always significantly less than the probability of that particle evolution from its corresponding prior particle. This means that it will likely not have a significant effect on particle weighing. Lastly, even if there was a high enough probability that a different particle would have evolved to the current particle, this high probability will likely be reflected by a higher density of particles in that region, effectively capturing the higher concentration of probability in that region. For these reasons, we assume that $P(x^i_k|x^j_{k|k-1})$ is 0 for i≠j. This refines our equation as below:

$$E[x_k|z_k, x_{k|k-1}] \propto \sum_{i=1}^{n} x^i_k P(z_k|x^i_k) P(x^i_k|x^i_{k|k-1}) P(x^i_{k|k-1}) \tag{6.41}$$

The probability of sampling a particle based on its corresponding predecessor, $P(x^i_k|x^i_{k|k-1})$, is inherently included in the sampling process. During sampling, particles are drawn from the transition model, ensuring that higher-probability transitions are more likely to occur. This process inherently aligns the particle distribution with the system's dynamics without requiring explicit computation of the transition probabilities during the weight update step. By sampling from the transition model at each time step, the transition probabilities are implicitly incorporated into the particle set. This eliminates the need for their explicit calculation, reducing computational complexity while maintaining accuracy. Importantly, this sampling is unbiased, meaning that over time, it accurately represents the underlying probability distribution without overestimating or underestimating specific states.

Since the transition probabilities are already embedded in the sampling process, we can safely omit them from the weight update equation, effectively assuming that $P(x^i_k|x^i_{k-1}, u_k) = 1$. This simplification further enhances computational efficiency without compromising the quality of the filter's performance.

This refines our equation as follows:

$$E[x_k|z_k, x_{k|k-1}] \propto \sum_{i=1}^{n} x^i_k P(z_k|x^i_k) P(x^i_{k|k-1}) \tag{6.42}$$

At each step, we can simply normalize the weight on each particle by the sum of particle weights. This means that we will not explicitly incorporate the prior probability of the i-th particle, but its weight. This is described as below:

$$E[x_k|z_{1:k}, u_{1:k}] \approx \sum_{i=1}^{n} x^i_k * \frac{P(z_k|x^i_k)w^i_{k|k-1}}{\sum_{j=1}^{n} P(z_k|x^j_k)w^j_{k|k-1}} \quad (6.43)$$

The initial weights can either be all equal, or normalized probabilities according to the initial state distribution. Then, for each observation, we can simply propagate the particles through the state estimation function, then calculate the equation above to get our state estimate.

Due to inaccuracies in the state transition function, many particles will deviate greatly from the true state, and thus the observations. This is another cause of the particle degeneracy problem, bringing along with it the same issues as discussed above.

In this situation, the high weight particles are informative, but the low weight particles are not. So, before state estimation, we get rid of the low weight particles. To keep a sufficient number of particles, we resample in informative areas. To do this, we duplicate higher weight particles. These duplicates will deviate from each other in subsequent state estimations due to process noise. This overall process is known as **resampling**.

When resampling, higher weight particles are more likely to be representative of the true state, however, lower weight samples may simply be influenced by incorrect observation. So, we want all particles after the deletion to potentially be duplicated, with higher weight particles having a greater probability. Since the normalized weights (w') sum to one, they accurately represent a probability distribution. So, before state estimation, we can sample n particles with each particle having a sampling probability equal to its normalized weight. The standard method for this from statistics is to create a discrete cumulative distribution function partitioning [0,1] into segments of lengths w'_i (normalized i-th weight). For each draw, we generate a uniform random number u $\in$ [0,1] and pick the particle whose segment contains u. For a sample u, we set the new particle equal to x_j if the following equation is true:

$$\sum_{i=1}^{j-1} w'_i \leq u < \sum_{i=1}^{j} w'_i \quad (6.44)$$

This is known as the **multinomial resampling method**. Since the u values are completely random, it is possible for u values to be very close to each other or for some u ranges to be entirely unrepresented. This can lead to excessive deviation in duplications relative to weight proportion. To prevent this, we want to ensure that each u segment has a particle. This calls for a **systematic resampling method**.

Instead of drawing random numbers for each particle, this method divides the cumulative weight distribution into equal-sized intervals and selects one particle per interval. We divide the cumulative distribution into N equal intervals. A random offset is chosen between 0 and 1/N, and

the first particle is selected by sampling at this offset. Each subsequent u value is chosen by adding 1/N to the previous particle.

Resampling could happen periodically, for example every 7 state estimations, but this could lead to resampling before necessary, leading to increased computational intensity, or failure to resample when necessary, leading to inaccurate state estimations. So, we should only resample when necessary. When particles degenerate, more and more particles become degenerate over time. This means that an easy way to diagnose the necessity for resampling is when only a few particles are relevant, meaning that weight is concentrated in a few particles.

The quantity $\sum w_i^2$ measures the concentration of the weights. When the weights are more uniform, $\sum w_i^2$ is close to 1/N. However, when one weight dominates, $\sum w_i^2$ increases to 1, reflecting a reduction in the effective number of particles. This gives rise to the metric of the Number of Effective Particles (N_{eff}), given by $1/\sum w_i^2$. As the weights get increasingly spread out, N_{eff} approaches N. As the weights get increasingly more concentrated, N_{eff} approaches 1.

$$N_{eff} = \frac{1}{\sum_{i=1}^{N} w_i^2} \tag{6.45}$$

This means that when N_{eff} drops below some threshold, the weights are too concentrated and resampling is necessary. A common threshold is when $N_{eff} < N/2$, reflecting that most particles are no longer contributing significantly to the estimate.

Resampling can lead to the particles converging to 1 or a small number of states. However, if filtering for a dynamic state, the state changes over time in a nondeterministic manner due to noise in the state space model. For a given state at step k-1, there are multiple potential states at time step k. If we have multiple nearby particles at time step k-1 and pass them through the deterministic state space model to get the particles at time step k, the particles will likely be very close together and not represent the range of possibilities for x_k. We want to have particles representing all areas with a non-negligible probability of being the true state. Without sufficient diversity, the particle filter can miss important aspects of the true state and might fail to track the state correctly. Additionally, modelling the propagation of these particles as deterministic does not align with the true system dynamics.

To avoid deterministic paths predicted by the model, which could lead to overconfidence in the model's accuracy and result in overconcentration on a particular state, we add noise to the output of propagating each particle through the state space model. To ensure that the particles explore the state space in a way that is consistent with the system's real-world dynamics, and that the particles accurately reflect different possible subsequent states given the prior states, the noise is consistent with the process noise. This process is described below, where w is the process noise:

$$x_k = f(x_{k-1}, u_k) + w \tag{6.46}$$

Due to the Central Limit Theorem, we usually approximate w as Gaussian. Gaussian distributions are also computational simple, and can approximate to many other symmetric or slightly skewed distributions. This means that $P(x_k|x_{k-1}, u_k)$ is a Gaussian distribution with mean $f(x_{k-1}, u_k)$, and covariance matrix equaling the covariance matrix of w found via calibration. This covariance matrix, denoted as **Q** in equations, is known as the **process noise covariance matrix**. Q is an n x n matrix, where n is the number of state variables. Q is given as below:

$$Q = E[ww^T] \tag{6.47}$$

$$Q_{ij} = Cov(w_i, w_j) = E[w_i w_j] \tag{6.48}$$

If the noise in different state dimensions is uncorrelated, all off-diagonal elements are zero.

This makes the distribution of $P(x_k|x_{k|k-1})$ for an n-dimensional state vector, the multivariate Gaussian formula as follows:

$$P(x_k|x_{k|k-1}) = \frac{1}{(2\pi)^{n/2}|Q|^{1/2}} exp\left(-\frac{1}{2}(x_k - f(x_{k-1}, u_k))^T Q^{-1}(x_k - f(x_{k-1}, u_k))\right) \tag{6.49}$$

This means that the probability of a particle propagating to each subsequent state should align with the Gaussian distribution centered on the output of propagating the particle through the state estimation function along with the control input, with a covariance equal to the process noise covariance matrix Q. To calculate this, we usually propagate each prior time step particle through the state transition function, then, for each particle, randomly sample noise from a Gaussian distribution with a mean of 0 and covariance described by the process noise covariance matrix Q. Once we generate these new particles, we can directly calculate the expected value equation.

This means that our Particle Filter is as follows:

- Initialize particles and weights
- For each observation:
    - Propagate particles with added noise
    - Weigh particles
    - If N_{eff} is below threshold:
        - Resample particle
        - Re-weigh particles
    - Estimate the state

6.7 - Kalman Filtering

6.7.1 - Kalman Filter (KF)

As we know, a recursive estimator must find $f(x_{k|k-1}, z_k)$ to optimally fuse $x_{k|k-1}$ with the new insights from z_k. In the particle filter, we approximated the distribution for $x_{k|k-1}$ with a discrete set of particles. However, this has some issues. First, it requires approximate knowledge of $P(z|x)$ which may be hard to gauge or represent if it does not fit into a nice continuous PDF. Additionally, the particle filter requires the propagation, analyses, and potential resampling of points throughout the entire distribution. The overall cost scales linearly with the number of particles, which is typically high to ensure a good approximation, making the particle filter much more computationally intensive. This can be very computationally intensive for large state vectors, a complex $P(z|x)$ distribution, a complex observation function, and/or a complex state transition model.

However, approximating the entire distribution to fuse $x_{k|k-1}$ with z_k is often unnecessary as we usually only care about the expected value. This leads to the goal of representing $x_{k|k-1}$ and relevant associated data in a less computationally intensive manner.

As detailed above, the second method of expected value estimation is calculating the minimizer of posterior (after fusion) variance/mean squared error. This is known as the MMSE solution. Essentially, we fuse z_k and $x_{k|k-1}$ in a way that minimizes posterior variance. This means that we only need information about z_k and $x_{k|k-1}$ which would indicate the posterior variance. This means we just need some simple way to represent this information, a formula for fusing z_k and $x_{k|k-1}$, and some tunable parameter that affects the fusion for which we can find the optimal value.

The uncertainty/variance in the output of an equation comes from the variance in the inputs. This means that to find the output variance, we merely need to propagate through the variances of the inputs gained through calibration. This means that we must use our estimate for the variance in $x_{k|k-1}$ and for z_k.

However, the variances in the values in these vectors are often dependent values. This means we must also consider some relevant covariances. If the sensors are placed very close to each other or have a common source of error (e.g., both are affected by a nearby heat source), their readings may be dependent, and errors will be correlated. As another example, if a camera gives both x and y coordinates that tend to shift together under certain lighting or vibration conditions. Depending on the situation, the error in sensors may be positively correlated, negatively correlated, or have no correlation. Additionally, the values in the state may be physically related, such as position and velocity. This means that we must consider the covariances of measurement values, and therefore state variables within a time step. This means that we must consider the covariance matrices of $x_{k|k-1}$ and z_k. Since we will have to make future fusions after x_k, we will also rely on the covariance matrix of x_k and thus must calculate this full

matrix. Solely propagating means and covariance matrices requires far fewer calculations than the full particle operations required of the particle filter.

The noise vector for the measurement is as follows, with $\hat{x}_k$ symbolizing the predicted ground truth state vector at time step k since x_k represents the actual ground truth:

$$v_k = z_k - h(x_k) \tag{6.50}$$

v_k is a zero-mean noise vector. The measurement noise covariance matrix, $\mathbf{R_k}$, encapsulates sensor uncertainty. Element i, j in R_k is the covariance between z_i and z_j, with element i, i in R_k equalling the variance of z_i. This means that the diagonal elements of the matrix will be variances, and the matrix will be symmetric about its main diagonal (top left to bottom right). All variances and covariances can be found in initial calibration. R_k is mathematically described as below:

$$R_k = E[v_k v_k^T] \tag{6.51}$$

Note that for most applications, this is constant, but in some applications certain elements of R_k are dependent on certain variables. In these situations, R_k must be recalculated at each step.

The noise vector for state estimates is as follows:

$$e_k = x_k - \hat{x}_k \tag{6.52}$$

This is also a zero-mean vector. Note that this formulation assumes that the state variables can be modeled with additive error. In other words, we assume that the state space is a vector space, allowing us to define error as a linear difference.

However, this assumption breaks down for states that reside on nonlinear manifolds, such as orientation and pose representations. For example, rotations belong to the Lie group SO(3), the Special Orthogonal Group in 3D. When translations are included, the full pose belongs to SE(3), the Special Euclidean Group in 3D. Neither of these are vector spaces. In such cases, error must instead be defined on the tangent space of the manifold using group operations, rather than simple subtraction. This leads to the development of manifold-aware filters, such as the Invariant Kalman Filter (IKF) — particularly the Invariant Extended Kalman Filter (IEKF). These filters redefine how error and uncertainty are represented and propagated, preserving invariance and improving consistency in nonlinear spaces. A detailed exploration of these advanced filters is beyond the scope of this textbook. For now, we assume error is defined additively as shown in the equation above.

The state estimate covariance matrix $\mathbf{P_k}$ encapsulates the uncertainty in the estimate of the state. Diagonal elements of P_k store the expected/propagated variance of the corresponding state element of that row and column. Off-diagonal elements store the expected/propagated covariance of the corresponding state elements of that row and column.

The only information we have as to potential incorrectness of $x_{k|k-1}$ given our new measurement is how much z_k deviates from the expected value of z_k from $x_{k|k-1}$, denoted as $\hat{z}_k$. Meaning how much z_k deviates from $E[z_k|x_{k|k-1}]$. Since we have some uncertainty about $x_{k|k-1}$, we must take into account the inherently distributive nature of $x_{k|k-1}$ given by our estimate and relevant uncertainty data, our estimate is not simply $h(x_{k|k-1})$. Note that if h is approximately linear, then these two equations are the same; the expected value of z_k is simply $h(x_{k|k-1})$. Note that h(x) = x if x and z contain the same variables. If under some x value, the observation we predict is the observed value, then we should not update the estimate. The only change we should make is to lower our uncertainty in our state. However, if we have a disparity, then we should update x. Generally, the larger the disparity between z_k and our expected z_k the larger the update we should make to the state. This disparity is known as the **innovation** and is given as below:

$$y_k = z_k - \hat{z}_k \tag{6.53}$$

This means that our update function should be as given below:

$$x_k = f(\hat{x}_{k|k-1}, z_k - \hat{z}_k) \tag{6.54}$$

Since $x_{k|k-1}$ is our initial expected value, our probability distributions of updates will be centered on $x_{k|k-1}$, our prior estimate. Since the expected innovation is 0, the expected value for x_k before obtaining z_k is $x_{k|k-1}$. This simplifies our equation as we can view our update function as a correction to $x_{k|k-1}$ based on the direction and magnitude of the innovation. This is given as below:

$$\hat{x}_k = \hat{x}_{k|k-1} + f(z_k - \hat{z}_k) \tag{6.55}$$

Note that f(0) = 0.

To estimate this update value, we would have to figure out which directional changes in x bring the estimate closer or farther away from z. For example, if our h function is h(x) = 4x + 3, our predicted z value was 1, and our observed z value was 1.5, we would want to increase x. It is important to note that this rule only works if h is monotonic, at least for any likely errors of x, because it is possible that a small change in x could move z in one way, and a larger increase in x could move z the other way. If this criteria is not met, then there is no valid reason why a larger disparity would indicate a larger change. However, this is not usually an issue as most functions meet this criteria, or the sensor errors are too small for this to affect sensor fusion. If this is not met, then the particle filter should be used. This idea of monotonicity between the predicted x and true x, is met for linear h, square roots, logarithms, etc. For higher order polynomials, one has to consult their exact h functions as well as input covariance matrix. In practice, this criteria is usually met for quadratic functions, but often can lead to significant errors in higher order polynomials.

Assuming this criteria, we also have to figure out how much to update x. This is obviously going to depend on our state prediction and observation certainties, but also how h(x) varies with x. If a small change in x yields a big change in z, we do not want to change x as much for some innovation as we would if a small change in x yielded a smaller change in z. This change depends on the function h, near our predicted x.

To do this, the update must take into account how much to change x to obtain the observed z prediction. Sensor fusion essentially finds the optimal point between the current x and the x that would result in that observation. This requires taking into account how changes in x change z at each point between x and the nearest x that would result in z. For example, if $h(x) = x^2$ and we predicted $z = 0$, a deviation of 16 would warrant twice as much of an alteration of our state as a deviation of 4.

Taking into account many higher order polynomial functions or other complex functions can be computationally intensive because they introduce extra components into the partial differentiation, require complex formulas for what changes in x lead to the desired balance between x and z, lead to more calculations if they have more than one operation, cause the relationship between changes in x and changes in z from x to z will change depending on our observation, and also lead to a more difficult propagation of the covariance matrices. Additionally, many functions are linear or at least approximately locally linear with respect to likely errors in predictions or observation noise. Lastly, since an innovation-based update already can't work if h(x) is highly non-monotonic, there are not many functions that are highly nonlinear that it can work on. For these reasons, we assume h is approximately locally linear and treat the innovation as linearly proportional to update. Note that if the observation function is highly nonlinear with respect to the likely errors, then the particle filter should be used. This means that each innovation term will be linearly proportional to each state adjustment term with some slope. This means that an n times greater deviation in the innovation will lead to an n times greater deviation in the state. Note that this slope will be 0 if the innovation term is unrelated to the corresponding state term. This multi-input multi-output linear function can be captured with matrix multiplication. One of the fundamental theorems of linear algebra tells us that every such linear map $f: R^n \rightarrow R^m$ can be represented by an m×n.

This refines our update equation as follows:

$$\hat{x}_k = \hat{x}_{k|k-1} + K_k(z_k - \hat{z}_k) \tag{6.56}$$

In this situation, K_k is called the **Kalman Gain** of the kth step. The values in this matrix make up the parameters in this update, meaning that they are what we choose to minimize posterior MSE. This means that we simply have to propagate forward the uncertainties, then use partial differentiation to figure out the optimal K_k. Note that since we unbiased estimators for our observations, state transition model, and observation model due to calibration, our posterior MSE is the same as our posterior variance.

It is important to note that each element of K_k will only affect one state variable of x_k. Each state estimate variable is affected by the corresponding row in K_k. This means that optimizing K_k for one state variable will not affect K_k for other state variables. We could separately optimize each row of K_k, however this is unnecessary as the optimal K_k matrix will contain each optimal row. This means that we can simply perform this optimization one time for the sum of variances, instead of separately for each row. The benefit of minimizing a single scalar value is that it reduces the complexity of the optimization problem.

The covariance propagation is shown as below:

$$P_k = Cov(e_k) = E[(e_k - E[e_k])((e_k - E[e_k])^T] = E[e_k e_k^T] \quad (6.57)$$

By expanding our prediction, we get the following:

$$e_k = x_k - \hat{x}_k = x_k - (\hat{x}_{k|k-1} + K_k(z_k - \hat{z}_k)) = (x_k - \hat{x}_{k|k-1}) - K_k(z_k - \hat{z}_k) \quad (6.58)$$

By using the relationships of the covariance of a difference, we get the following:

$$P_k = Cov(x_k - \hat{x}_{k|k-1}) - Cov(x_k - \hat{x}_{k|k-1}, K_k(z_k - \hat{z}_k)) - Cov(K_k(z_k - \hat{z}_k), x_k - \hat{x}_{k|k-1} + Cov(K_k(z_k - \hat{z}_k)) \quad (6.59)$$

The first term is already known to us from propagation as the covariance matrix of the prior prediction. This refines our equation as below:

$$P_k = P_{k|k-1} - Cov(x_k - \hat{x}_{k|k-1}, K_k(z_k - \hat{z}_k)) - Cov(K_k(z_k - \hat{z}_k), x_k - \hat{x}_{k|k-1}) + Cov(K_k(z_k - \hat{z}_k)) \quad (6.60)$$

By using matrix rules, given that K_k will contain known constants with 0 variance, we can factor it out as follows:

$$P_k = P_{k|k-1} - Cov(x_k - \hat{x}_{k|k-1}, z_k - \hat{z}_k)K_k^T - K_k Cov(z_k - \hat{z}_k, x_k - \hat{x}_{k|k-1}) + K_k Cov(z_k - \hat{z}_k)K_k^T \quad (6.61)$$

As we know, the covariance matrix of two values is the transpose of the covariance matrix of the values in reverse order, $Cov(a, b) = Cov(b, a)^T$. This refines our equation as below only needing to calculate the cross-covariance matrix once:

$$P_k = P_{k|k-1} - Cov(x_k - \hat{x}_{k|k-1}, z_k - \hat{z}_k)K_k^T - K_k Cov(x_k - \hat{x}_{k|k-1}, z_k - \hat{z}_k)^T + K_k Cov(z_k - \hat{z}_k)K_k^T \quad (6.62)$$

In this equation, $Cov(z_k - \hat{z}_k)$ is the covariance matrix of the innovation, and is denoted as S_k. $Cov(x_k - \hat{x}_{k|k-1}, z_k - \hat{z}_k)$ is the cross-covariance matrix between state prediction error and innovation at time step k, and is denoted as P_{xz}. This refines our equation as below:

$$P_k = P_{k|k-1} - P_{xz}K_k^T - K_k P_{xz}^T + K_k S_k K_k^T \quad (6.63)$$

By performing standard analytical optimization on this equation, we get the following result for K_k:

$$K_k = P_{xz}S_k^{-1} \quad (6.64)$$

When substituting this back into equation 6.63 for P_k, we obtain the following:

$$P_k = P_{k|k-1} - P_{xz}S_k^{-1}P_{xz}^T \quad (6.65)$$

By substituting in K_k to equation 6.65 , we get the following:

$$K_k = P_{xz}S_k^{-1} \quad (6.64)$$

$$P_k = P_{k|k-1} - K_k P_{xz}^T \quad (6.67)$$

This means we simply have to calculate the Kalman Gain, then can substitute it into the equation for the posterior covariance matrix.

Focusing on part of the last term:

$$S_k = Cov(z_k - \hat{z}_k) = Cov(h(x_k) + v_k - \hat{z}_k) = Cov((h(x_k) - \hat{z}_k) + v_k) \quad (6.68)$$

v_k is the measurement noise and $h(x_k) - \hat{z}_k$ is how much true underlying values we are attempting to observe differ from their predicted value given $x_{k|k-1}$ and h

By using properties of covariance propagation:

$$S_k = Cov(h(x_k) - \hat{z}_k) + Cov(v_k) + Cov(h(x_k) - \hat{z}_k, v_k) + Cov(v_k, h(x_k) - \hat{z}_k) \quad (6.69)$$

The cross-covariances encapsulates the correlation in errors between z_k and $h(x_k)$ and errors between $\hat{z}_k$ and $h(x_k)$. Measurement noise is generally caused by the current external interrupters of proper sensor functioning. On the other hand, errors in $\hat{z}_k$ are mainly caused by process noise in the state transition function and errors accumulated in past measurements predictions. In most cases process noise and measurement noise have different causes and are uncorrelated, so we generally assume as such. Also, measurement noise is often not affected by the true state. Additionally, we ensure in calibration that measurements are unbiased, meaning that the error in them is random. Additionally, in most situations for most sensors, the causes of error vary and have varying effects, justifying the assumption that measurement error is uncorrelated over time. In these situations, the cross-covariance is not significant and it is justified to assume that it is 0, thus eliminating the need to consider it.

This refines our covariance for this term as follows:

$$S_k = Cov(h(x_k) - \hat{z}_k) + Cov(v_k) \quad (6.70)$$

We already know cov(v_k) from calibration as the measurement noise covariance matrix, thus refining our equations as follows:

$$S_k = Cov(h(x_k) - \hat{z}_k) + R_k \quad (6.71)$$

The first term is essentially the correlation between uncertainty in predicted observation variables. Even though in an ideal world the true state x_k is fixed, in the filter we don't know it; we only have our probabilistic model of it given by $x_{k|k-1}$ and its covariance matrix. This covariance does not describe the variability of the true state itself (which is fixed) but rather the uncertainty in our estimate of that state. $\hat{z}_k$ is the expected value of $h(x_{k|k-1})$ given the probabilistic knowledge of $x_{k|k-1}$.

Since $\hat{z}_k$ is then a constant and not a distribution, it has no effect on uncertainty/covariance and can be removed. This refines our equations as follows:

$$S_k = Cov(h(x_{k|k-1})) + R_k \quad (6.72)$$

This refines the overall equations as follows:

$$K_k = P_{xz} S_k^{-1} = P_{xz}(Cov(h(x_{k|k-1})) + R_k)^{-1} \quad (6.73)$$

$$P_k = P_{k|k-1} - K_k P_{xz}^T \quad (6.74)$$

For P_{xz}, since the innovation contains $h(x_{k|k-1})$, $x_{k|k-1}$ affects both innovation and state prediction error, so they are dependent and have a non-zero cross covariance. We can refine P_{xz} as follows:

$$P_{xz} = E[(x_k - \hat{x}_{k|k-1})(z_k - \hat{z}_k)^T] = E[(x_k - \hat{x}_{k|k-1})(h(x_k) + v_k - \hat{z}_k)^T]$$
$$= E[(x_k - \hat{x}_{k|k-1})((h(x_k) - \hat{z}_k) + v_k)^T] \quad (6.75)$$

Since we assume that prior state prediction error is uncorrelated with measurement error, we can distribute prior state prediction error, then remove the cross-covariance between measurement noise and prior state prediction error. This refines our equations as below:

$$P_{xz} = E[(x_k - \hat{x}_{k|k-1})(h(x_k) - \hat{z}_k)^T] \quad (6.76)$$

Where x_k is a distribution, $\hat{x}_{k|k-1} = E[x_k]$ and $\hat{z}_k = E[h(x_k)] \approx E[h(x_{k|k-1})]$

This refines our overall equations as below:

$$K_k = P_{xz} Cov(h(x_{k|k-1}) + R_k)^{-1} \quad (6.77)$$

$$K_k = E[(x_k - \hat{x}_{k|k-1})(h(x_k) - \hat{z}_k)^T](Cov(h(x_{k|k-1})) + R_k)^{-1} \quad (6.78)$$

$$P_k = P_{k|k-1} - K_k E[(x_k - \hat{x}_{k|k-1})(h(x_k) - \hat{z}_k)^T]^T \quad (6.79)$$

We know h, and we know $x_{k|k-1}$, so we must use this to estimate the posterior covariance matrix. This presents an issue since h is often nonlinear. Note that we will necessarily have some errors in nonlinear systems due to the linearity of the update function. However, this alone only prevents the use of this technique for h functions that are highly nonlinear. However, due to some combination of the h being nonlinear, but not too nonlinear, and small expected errors, this limitation often does not lead to significant errors that would warrant switching to the particle filter. For these reasons, we must attempt to maximize the capabilities of this fusion technique for nonlinear h functions.

Additionally, as shown above, we often have to fuse sensor data for evolving systems. This means that our current state estimate is no longer a valid prediction for our true state and cannot be used to predict the inputs. So, we must generate a prediction of our future state. This can be done via state-space modeling. This entails that our filters will have a prediction step, then a fusion step, where the fusion step fuses the input with predicted state according to our state-space model.

We will no longer be fusing the prior state estimate with the current sensor readings, but the state space prediction with the current sensor readings. This means that the state vector we fuse with the sensor readings will have additional noise which must be taken into account for fusion.

Through calibration, we can gauge the variance of each state prediction by comparing it to the ground truth. We can also gauge the covariances between elements of the state prediction vector. So, we can get a covariance matrix, called the **process covariance**, and denoted as **Q** in equations

In many practical applications, control inputs are directly commanded by the system and can be measured or calculated with high precision. For example, in a robotic system, motor commands or actuator inputs are known accurately because they are determined by the system's control logic. This means that we can model u_{k-1} as having no covariance matrix. This reduces our problem to propagating P_{k-1} through f, where f has a covariance matrix **Q**.

The process noise can be written as w_k below where x_k and x_{k-1} are true states such that the only output error is caused by f:

$$x_k - f(x_{k-1}, u_{k-1}) = w_k \quad (6.80)$$

There is also noise in the prior prediction as described below:

$$x_{k-1} - \hat{x}_{k-1} = e_{k-1} \quad (6.81)$$

The true value for x_k is mathematically described below:

$$x_k = f(\hat{x}_{k-1} + e_{k-1}, u_{k-1}) + w_{k-1} \quad (6.82)$$

$$e_{k|k-1} = x_k - \hat{x}_{k|k-1} = f(\hat{x}_{k-1} + e_{k-1}, u_{k-1}) + w_{k-1} - f(\hat{x}_{k-1}, u_{k-1}) \quad (6.83)$$

$$e_{k|k-1} = (f(\hat{x}_{k-1} + e_{k-1}, u_{k-1}) - f(\hat{x}_{k-1}, u_{k-1})) + w_{k-1} \quad (6.84)$$

Note that if f is nonlinear then we cannot directly pass our prior distribution for x through f to get our estimate for our updated x, as E[f(x)] is not necessarily equal to f(E[x]) if f is nonlinear. This means we must instead take $E[x_k|x_{k-1},u_{k-1}]$.

Since the expected value of errors will be 0 due to calibration, $P_{k|k-1}$ can be written as follows:

$$P_{k|k-1} = E[\{(f(\hat{x}_{k-1} + e_{k-1}, u_{k-1}) - f(\hat{x}_{k-1}, u_{k-1})) + w_{k-1}\}\{(f(\hat{x}_{k-1} + e_{k-1}, u_{k-1}) - f(\hat{x}_{k-1}, u_{k-1})) + w_{k-1}\}^T] \quad (6.85)$$

By distributing the difference in fs and noting that the process noise over time is assumed to be random and uncorrelated, w_{k-1} and e_{k-1} are independent. This means that the terms have no cross covariance. This refines $P_{k|k-1}$ as follows:

$$P_{k|k-1} = E[(f(\hat{x}_{k-1} + e_{k-1}, u_{k-1}) - f(\hat{x}_{k-1}, u_{k-1}))(f(\hat{x}_{k-1} + e_{k-1}, u_{k-1}) - f(\hat{x}_{k-1}, u_{k-1}))^T] + E[w_{k-1}{w_{k-1}}^T] \quad (6.86)$$

By noting that the last equation equals Q, we refine $P_{k|k-1}$ as follows:

$$P_{k|k-1} = E[(f(\hat{x}_{k-1} + e_{k-1}, u_{k-1}) - f(\hat{x}_{k-1}, u_{k-1}))(f(\hat{x}_{k-1} + e_{k-1}, u_{k-1}) - f(\hat{x}_{k-1}, u_{k-1}))^T] + Q_k$$
$$= Cov(f(x_{k-1}, u_{k-1})) + Q_k \quad (6.87)$$

In this situation, we make no assumption as to the linearity of f, meaning that we must propagate the covariance matrix of x_{k-1} through a potentially nonlinear f.

This brings us to the following equations

$$P_{k|k-1} = Cov(f(x_{k-1}, u_{k-1})) + Q_k \quad (6.87)$$

$$K_k = E[(x_k - \hat{x}_{k|k-1})(h(x_k) - \hat{z}_k)^T](Cov(h(x_{k|k-1})) + R_k)^{-1} \quad (6.78)$$

$$P_k = P_{k|k-1} - K_k E[(x_k - \hat{x}_{k|k-1})(h(x_k) - \hat{z}_k)^T]^T \quad (6.79)$$

In the case where both f and h are linear, they are equivalent to their matrix forms. This matrix can be obtained by taking the Jacobian of h_k to get H_k, with $h_k(x) = H_k x$. We can also take Jacobian of the state-space model with respect to x_{k-1} to get A_{k-1}, and similarly with respect to u_k to get B_{k-1}, such that $x_{k|k-1} = A_{k-1}x_{k-1}+B_{k-1}u_{k-1}$. A is called the state transition matrix, while B is called the control input matrix. By noting that $B_k u_{k-1}$ imparts no additional error, it is irrelevant for covariance calculation. Additionally, since f is linear, we can say that the expected value of x

can be obtained by directly passing x_{k-1} and u_{k-1} through f. By applying the covariance operator accordingly, we get the following equations for propagating covariance through the state transition function:

$$\hat{x}_{k|k-1} = f(\hat{x}_{k-1}, u_{k-1}) \tag{6.88}$$

$$f(\hat{x}_{k-1} + e_{k-1}, u_{k-1}) = f(\hat{x}_{k-1}, u_{k-1}) + A_{k-1}e_{k-1} \tag{6.89}$$

$$Cov(f(x_{k-1}, u_{k-1})) =$$
$$= E[(f(\hat{x}_{k-1} + e_{k-1}, u_{k-1}) - f(\hat{x}_{k-1}, u_{k-1}))(f(\hat{x}_{k-1} + e_{k-1}, u_{k-1})$$
$$- f(\hat{x}_{k-1}, u_{k-1}))^T] \tag{6.90}$$

$$= E[(f(\hat{x}_{k-1}, u_{k-1}) + A_{k-1}e_{k-1} - f(\hat{x}_{k-1}, u_{k-1}))(f(\hat{x}_{k-1}, u_{k-1})$$
$$+ A_{k-1}e_{k-1} - f(\hat{x}_{k-1}, u_{k-1}))^T \tag{6.91}$$

$$= E[(A_{k-1}e_{k-1})(A_{k-1}e_{k-1})^T] = A_{k-1}E[e_{k-1}e_{k-1}^T]A_{k-1}^T = A_{k-1}P_{k-1}A_{k-1}^T \tag{6.92}$$

$$P_{k|k-1} = A_{k-1}P_{k-1}A_{k-1}^T + Q_k \tag{6.93}$$

We can do the same for h. Additionally, we can directly pass input beliefs through h or f to get the output belief since $E[f(x)] = f(E[x])$ if f is linear.

$$\hat{z}_k = h(\hat{x}_{k|k-1}) \tag{6.94}$$

$$h(x_k) = h(\hat{x}_{k|k-1}) + H_k(x_k - \hat{x}_{k|k-1}) \tag{6.95}$$

$$h(x_k) - \hat{z}_k = h(\hat{x}_{k|k-1}) + H_k(x_k - \hat{x}_{k|k-1}) - h(\hat{x}_{k|k-1}) = H_k(x_k - \hat{x}_{k|k-1}) \tag{6.96}$$

$$P_{xz} = E[(x_k - \hat{x}_{k|k-1})(h(x_k) - \hat{z}_k)^T] = E[(x_k - \hat{x}_{k|k-1})(H_k(x_k - \hat{x}_{k|k-1}))^T]$$
$$= E[(x_k - \hat{x}_{k|k-1})(x_k - \hat{x}_{k|k-1})^T]H_k^T = P_{k|k-1}H_k^T \tag{6.97}$$

$$P_k = P_{k|k-1} - K_k(P_{k|k-1}H_k^T)^T = P_{k|k-1} - K_kH_kP_{k|k-1}^T = P_{k|k-1} - K_kH_kP_{k|k-1}$$
$$= (I - K_kH_k)P_{k|k-1} \tag{6.98}$$

Keep in mind that $P_{k|k-1}$ is symmetric so $P_{k|k-1} = P_{k|k-1}^T$

$$Cov(h(x_{k|k-1})) = E[(h_k - \hat{z}_k)(h(x_k) - \hat{z}_k)^T] =$$

$$= E[(h(\hat{x}_{k|k-1}) + H_k e_{k|k-1} - h(\hat{x}_{k|k-1}))(h(\hat{x}_{k|k-1}) + H_k e_{k|k-1} - h(\hat{x}_{k|k-1}))^T]$$

$$= E[(H_k e_{k|k-1})(H_k e_{k|k-1})^T] = H_k E[e_{k|k-1} e_{k|k-1}^T] H_k^T = H_k P_{k|k-1} H_k^T \quad (6.99)$$

This brings our final equations as below:

$$\hat{x}_{k|k-1} = f(\hat{x}_{k-1}, u_{k-1}) \quad (6.88)$$

$$P_{k|k-1} = A_{k-1} P_{k-1} A_{k-1}^T + Q_k \quad (6.93)$$

$$K_k = P_{k|k-1} H_k^T (H_k P_{k|k-1} H_k^T + R_k)^{-1} \quad (6.100)$$

$$\hat{x}_k = \hat{x}_{k|k-1} + K_k(z_k - h(\hat{x}_{k|k-1})) \quad (6.56)$$

$$P_k = (I - K_k H_k) P_{k|k-1} \quad (6.98)$$

Note that if the system dynamics are time-invariant (i.e., the system follows the same linear rules regardless of the time step), then A_k can be constant for all time steps. In this case, $A_k = A$ for all k. Additionally, if the relationship between the state and the measurement is fixed (the sensor model doesn't change over time), then $H_k = H$ for all k. This means that these matrices do not have to be recalculated.

These equations collectively are known as the standard form of the **Kalman Filter (KF)** (Kalman, 1960).

Note that we often have to propagate a covariance matrix through a slightly nonlinear function, meaning one that is nonlinear but can be sufficiently approximated as locally linear between an estimate and the true value. In this case the statement that the true value is the predicted value added to some matrix of the error is an approximation. In this case, we would have to recalculate the Jacobians, A and/or H, at each propagation step. This form of the Kalman Filter is known as the **Extended Kalman Filter (EKF)**.

It is important to note that the output mean and covariance also equals the approximate mean and covariance matrix of the output distribution. These are known as the first and second moments of the output distribution respectively. This can be viewed as a Gaussian approximation of the output distribution. This means that if the inputs are approximately Gaussian and the state transition function and observation model are approximately linear then Gaussianity is preserved under these transformations and the output state distribution will be approximately Gaussian. This means that we can get a full probability distribution of the output, gauging information such as the probability that some value is in some range.

6.7.2 - Unscented Kalman Filter (UKF)

If we are dealing with highly nonlinear functions, meaning that the 2nd derivative has a high magnitude at least some of the time, then the first derivative will vary greatly for small changes in input at some relevant part of the function. In other words, the slope at x and the slope at a neighboring point x+δ can be very different. In terms of sensor fusion, this means that the derivative of function output with respect to function input changes significantly between the expected state and the expected perturbed state (i.e., the expected state plus its anticipated deviation). If the curvature is high, that Jacobian is a poor stand-in for the actual sensitivity at the perturbed state. This means that the assumption of a constant slope made by the Jacobian Approximation will lead to inaccurate covariance propagation.

To better approximate the function, we could simply take a higher order Taylor approximation, but this is significantly more computationally intensive. Higher-order Taylor approximations require the calculation of higher-order derivatives, such as second-order (Hessian matrices) or even higher. These calculations are computationally expensive, especially in systems with large state vectors or complex nonlinear functions. The added computational cost can be prohibitive for real-time applications.

Also, if the function does not neatly fit into a lower order polynomial, such as x^2, the higher order derivatives will likely be very volatile and change significantly between the prediction and ground truth. This will lead to highly inaccurate covariance estimation. So, higher order Taylor approximations are seldom used. Note that it is occasionally used if the function is known to be a 2nd degree polynomial, or at least very close to one. In such cases, the second-order Taylor expansion provides an exact representation of the nonlinear function because all higher-order terms beyond the second order are zero for a polynomial of degree two.

This means that we require a new method for propagating covariance through nonlinear functions that respects the full non-linearity, as well as a method for estimating the posterior mean.

As we know, we can refine E[x] and Cov(x, y) as follows:

$$E[x] = \int_{-\infty}^{\infty} xP(x)dx \tag{6.6}$$

$$Cov(x,\ y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} (x - E[x])(y - E[y])^T P(x,y)dxdy \tag{6.101}$$

This means that we can refine both expected value equations in the posterior covariance formula as below:

$$E[f(x)] = \int_{-\infty}^{\infty} f(x)P(f(x))dx \tag{6.102}$$

$$Cov(f(x),\ g(y)) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}(f(x) - E[f(x)])(g(y) - E[g(y)])^T P(x,y)dxdy \quad (6.103)$$

For reasons mentioned in the previous section, with nonlinear and complex functions in high dimensions, performing this integration with continuous distributions is infeasible. This requires approximation methods. Recall that the posterior distributions essentially encode the probability of obtaining a specific output. This probability is definitionally equivalent to the probability of sampling an input which, when passed through the function, results in the aforementioned output. Also recall that integration is simply an infinite sum and that by sampling an infinite number of inputs, you cover the entire input space according to the input distribution. Therefore, every possible output is produced according to its true probability. This means that as the number of inputs converges to infinity, the sampled inputs increasingly align with the true joint input distribution and the obtained outputs increasingly align with the true joint output distribution.

Since infinite sampling is impossible, we must utilize a sufficiently large, yet computationally feasible number of samples from the input distribution to compute these functions. We can then estimate the expected value of the output, then compute the posterior covariance matrix. By performing monte carlo sampling, the proportion of being sampled is equal to a sample's probability, so probability weighting in the calculation of the expected value and covariance is implicitly included via the sampling process. This means that we can assume equal probability of samples. This refines our equations as below:

$$E[f(x)] = \frac{1}{n}\sum_{i=1}^{n} f(x_i) \quad (6.104)$$

$$Cov(f(x),\ g(y)) = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - E[f(x)])(g(y_i) - E[g(y)])^T \quad (6.105)$$

This requires knowledge of the functions, which we have through the observation model and the state transition function. However, this also involves knowing the probability of the variables for proper sampling. However, we only know the approximate first two moments of the state distribution, mean and covariance. This presents an issue if these moments do not give a good approximation of how likely a point should be sampled. This is usually due to strong skewness, heavy tails, or multimodality. If the skewness, kurtosis, and/or multi-modality is known ahead of time via calibration, then those can be used to fill in the missing information. Additionally, if the output follows some known continuous PDF formula, then the relevant distribution parameters can be filled in for sampling. However, this is usually not the case. This means that this method only works if the output distribution is not highly non-Gaussian. If it is and there is moderate nonlinearity in the observation model, or high nonlinearity in the state transition function, then the particle filter should be used.

Assuming the state/input distribution is not highly non-Gaussian, this method works. This above process of covariance estimation is known as Monte Carlo Covariance Estimation. However, this method requires a large number of samples to achieve low variance/error in the estimation. This means that we could either take a lot of samples, which can be computationally expensive, or take a few samples, which leads to high variance/error. This means we must develop a method that provides more accurate and computationally efficient estimation of the mean and covariance for nonlinear transformations of random variables.

One main reason that random sampling has these issues is that it is fully stochastic, meaning that each sample is independent of previous samples. This means that multiple samples could be very close to each other or that there may be disproportionately more samples with some input variable below its mean than above its mean. Additionally, there may be certain regions without a sample. Additionally, when approximating a distribution with the first two moments, the distribution will be symmetric about the mean. This means that an even number of points would be above and below each input variable's mean, with points capturing different areas of the input space to avoid redundancy.

Rectifying these issues points to a form of systematic or deterministic sampling. To maximize efficiency, we should only utilize inputs that have some significant probability. This means that the input vector must have some realistic magnitude and direction deviating from the mean.

To ensure symmetry about the mean, we can use the first two moments to generate n spread out points of realistic direction and magnitude away from the mean, then reflect them over the mean to capture the opposite side of the distribution. Essentially, for every chosen deviation vector, we should generate two points: the mean + the deviation vector, the mean - the deviation vector. Note that since the mean is the most likely input (mode) in a Gaussian distribution, we also propagate the mean through the nonlinear function in our posterior covariance estimation. This means that we will have 2n +1 points that we propagate, where n is the number of deviation vectors. These 2n + 1 points are called **sigma points**.

To get likely directions of deviations, we should ensure that all dimensions and covariances are properly accounted for. We must ensure that relative covariances are taken into account and that a comprehensive set of deviation directions is generated. Luckily this information is already provided to us in the form of the input covariance matrix. Each column represents the covariances with respect to a different variable. Each column points along a principal direction of dispersion in the state space. Additionally, these columns span the state space, meaning they can comprehensively capture it. Overall, this means that these columns provide likely deviation directions in a way that accurately and comprehensively covers the shape of the distribution given its first two moments. This means that for directional purposes, we could generate each point by adding/subtracting a column of the input covariance matrix.

However, the magnitude that each input variable deviates from the mean is important in representing the corresponding errors in the output variables. The magnitude of deviations determines how far the input points are from the mean, which affects how well they capture the nonlinear effects of the transformation. If deviations are too small, they might not capture the full impact of the nonlinearity. If they are too large, they might introduce inaccuracies by overrepresentation of extreme variations. Properly scaled deviations ensure that the points cover the relevant range of the distribution, allowing for an accurate representation of how the distribution behaves after transformation. To solve this problem, we can choose points that deviate from the mean by likely values.

However, the columns of the covariance matrix do not reflect our expected magnitude of deviation. This is easy to see with the regular variance, the square of expected deviation. By taking the square root, our answer more closely resembles likely deviations. So, we take the square root of the covariance matrix and add/subtract each column. This square root can be obtained by using Cholesky decomposition.

$$\sqrt{\Sigma} = L \quad \text{where} \quad \Sigma = LL^T \tag{6.106}$$

The columns accurately and comprehensively represent the magnitude and direction of the deviation vectors, maintaining the desired properties of the covariance matrix. Additionally, the columns of L are linearly independent, and L is invertible. Also, in many cases, the columns of L are orthogonal. Overall, this means that we can take the columns of L, as our n deviation vectors, with n being the dimension of the state space.

This brings our sigma point generation equations to:

$$\chi_0 = \mu \tag{6.107}$$

$$\chi_i = \mu + L_i \quad \text{for } i = 1, 2, \ldots, n \tag{6.108}$$

$$\chi_{i+n} = \mu - L_i \quad \text{for } i = 1, 2, \ldots, n \tag{6.109}$$

From this, we can estimate the output mean, then the output covariance matrix. This process is known as the **Unscented Transform (UT)**.

It is important to note that the optimal spread of points may vary depending on the situation. For example, in high-dimensional spaces the points may be cluttered near the mean due to the presence of more points. Since we want points to not be redundant and fully capture the distribution, this leads to the desire for sigma points to be farther from the mean. If the points are too close to the mean in high dimensions, they might miss significant variations in the function,

leading to an inaccurate estimate of the mean and covariance after transformation. This can also be affected by how nonlinear the function is, how non-Gaussian the input distribution is, etc. For these reasons, we often scale the values in the covariance matrix by a constant λ that is obtained through empirical testing.

After generating these points, using them to estimate the output mean, and then using them along with the output mean to generate the output covariance. We can then directly use it to calculate the posterior covariance of the state transition function. We can also use it to estimate the posterior covariance of the observation model, then K_k.

By substituting this into the equation for posterior covariance of the state transition function we obtain the following:

$$\chi^{i}_{k|k-1} = f(\chi^{i}_{k-1}, u_{k-1}) \tag{6.110}$$

$$\hat{x}_{k|k-1} = \frac{1}{2n+1}\sum_{i=0}^{2n} \chi^{i}_{k|k-1} \tag{6.111}$$

$$P_{k|k-1} = \frac{1}{2n+1}\sum_{i=0}^{2n} [\chi^{i}_{k|k-1} - \hat{x}_{k|k-1}][\chi^{i}_{k|k-1} - \hat{x}_{k|k-1}]^{T} + Q_k \tag{6.112}$$

The posterior covariance of the observation model prediction is given as below, note that Z represents observation model outputs, not observations.

$$Z^{i}_{k} = f(\chi^{i}_{k|k-1}) \tag{6.113}$$

$$\hat{z}_k = \frac{1}{2n+1}\sum_{i=0}^{2n} Z^{i}_{k} \tag{6.114}$$

$$Cov(h(x_{k|k-1})) = \frac{1}{2n+1}\sum_{i=0}^{2n} [Z^{i}_{k} - \hat{z}_k][Z^{i}_{k} - \hat{z}_k]^{T} \tag{6.115}$$

S_k which represents the covariance of the innovation, is given as below:

$$S_k = \frac{1}{2n+1}\sum_{i=0}^{2n} [Z^{i}_{k} - \hat{z}_k][Z^{i}_{k} - \hat{z}_k]^{T} + R_k \tag{6.116}$$

The posterior covariance of the estimate is given as follows:

$$P_k = P_{k|k-1} - K_k S_k K^{T}_{k} \tag{6.65}$$

In calculating the Kalman Gain, we must calculate P_{xz}, $K_k = P_{xz}S_k^{-1}$. This is given as below

$$P_{xz} = \frac{1}{2n+1}\sum_{i=0}^{2n} [\chi^{i}_{k|k-1} - \hat{x}_{k|k-1}][Z^{i}_{k} - \hat{z}_k]^{T} \tag{6.117}$$

$$= \frac{1}{2n+1}\sum_{i=0}^{2n} [\chi^{i}_{k|k-1} - \hat{x}_{k|k-1}][h(\chi^{i}_{k|k-1}) - \hat{z}_k]^{T} \tag{6.118}$$

Note that if h is highly non-linear, then the UT will be needed for calculating P_{xz} and Cov(h(x)). In this case, the state prediction sigma points generated to propagate covariance through h can be used for calculating P_{xz}.

If f is highly non-linear, then the UT will be needed for calculating $P_{k|k-1}$. Note that it is possible for the UT to be needed for one function and not for the other. We use the regular KF, EKF, and UT as needed for either function. You simply pick the most reliable method to propagate through h or f: apply a KF‑style update if linear, an EKF if mildly nonlinear, and a UT if it's strongly curved. By mixing KF, EKF, and UT in the same filter—using UT only when and where it's truly needed—you keep computational cost down while still capturing the essential nonlinear effects whenever linearization would introduce unacceptable error. When using the UT for both functions, we call the algorithm the **Unscented Kalman Filter (UKF)** (Julier et al., 1995). Note that in this situation we can reuse post-propagation points from estimating $P_{k|k-1}$ and $x_{k|k-1}$ when estimating Cov(h(x)) and P_{xz} as we assume they represent comprehensive and likely points. However, we cannot reuse post-propagation points for calculating Cov(h(x)) and P_{xz} as the estimate and covariance matrix will be updated due to the sensor fusion/update. For these reasons, the UKF generates sigma points once per update cycle — at the beginning of the prediction step.

Note that there are some minor adjustments that can be performed with how sigma points are sampled and weighted, but these are mostly minor and outside of the scope of this textbook.

Chapter 7: Simultaneous Localization & Mapping (SLAM)

7.1 - Introduction to SLAM

As established in prior chapters, two primary goals in robotics are localization and mapping. Generally, a robot must use sensors such as encoders, IMU, GPS, etc. for localization and use detailed sensors that gauge information from a distance such as LiDAR or Cameras for mapping. The robot continually uses sensors to localize and to get environmental information relative to the robot, and combines them to create a world map.

However, there are some issues with this approach. We cannot do either of these perfectly due to sensor inaccuracies and limitations. In terms of localization, errors with sensors that gauge pose change, such as IMU and encoders, can accumulate small errors over time which lead to wildly inaccurate pose estimations later on. This type of localization is known as **dead reckoning**. This hurts the ability of the robot to know the pose it occupies amongst a pre-made map or path, know its location relative to a goal, communicate its pose to people or other robots, know the absolute pose of objects or landmarks it senses, etc. Additionally, absolute pose sensors, such as GPS, are not sufficiently accurate for many robotics tasks as they frequently give reading up to multiple feet away from the true position at the time of publishing this textbook. Even if GPS was sufficient, it is not reliable or often available in many GPS-Denied environments such as indoors, underground/underwater, dense foliage/forest canopy, etc.

Even if localization was perfect, issues in our mapping sensors can cause a misalignment between consecutive sensor reads or between different scans sharing many points that were taking far apart in time. In these situations, the same environmental point can be sensed as occupying a different pose due to mapping sensor errors, or localization sensor errors for that matter. This leads to issues in map alignment and rectification. Note that mapping sensor error is generally independent of time, as LiDAR/camera quality don't usually degrade throughout an episode. This is in contrast to relative localization error which compounds over time. This means that at the beginning of an endeavor, both odometry drift and mapping sensor error will be relevant, with odometry drift quickly becoming the dominant source of error.

This overall leads to two goals:

- Helping the robot better correct drift in situations where absolute pose sensors are unavailable or insufficient
- Aligning map scans to create a single rectified map

Rectifying these goals entails either minimizing the localization error accumulated at each step, or correcting drift with some accurate baseline or reference for absolute pose. In order to do either of these, we need some information about pose or pose changes. Previously we focused on interoceptive (about the robot) sensors. However, exteroceptive (about the environment) sensors also give pose-indicative information. When mapping, we store the locations of points sensed in an environment. Due to the ability of LiDAR sensors and stereo

vision setups to gauge the 3D pose of points relative to a robot, they are pose indicative. If we know pose estimations of 3D points gauged from prior LiDAR scans, and we can gauge our current pose relative to these points, then we can know our pose estimation relative to the previous LiDAR scan. Since this provides extra information, we should use it to aid our localization.

This means that when we encounter these scenarios and have sufficient memory and processing, we should remember scans and use overlapping scans for localization and map alignment. Since consecutive scans will likely share a lot of points, a robot can continuously use an algorithm for matching points and estimating pose from the points for consecutive scans.

Even with smaller localization drift, over time these will lead to large localization inaccuracies and mapping discrepancies. This is because there is no stable baseline to correct drift. Gauging a stable baseline requires access to a more accurate sensor reading that gives us our absolute pose. However, we cannot do this at a given point at time as shown above due to difficulties with absolute pose sensing. All the information we have is the totality of our sensor readings throughout the robot's endeavor. By the nature of drift, latter pose estimates will have higher variance than prior pose estimates. This means that prior estimations of the robot's pose and environment were more accurate.

This means we should analyze how our current sensor data relates to past sensor data to correct drift. We must be able to use this more accurate pose-indicative information and figure out what it indicates about our pose. When we are at a similar pose as one we have been in before, we know that we have been there before. However, we have no way of knowing the true pose based on this information alone. When mapping, we can store and access the locations of points previously sensed in an environment. If we know more accurate pose estimations of 3D points gauges from prior LiDAR scans, and we can gauge our current pose relative to these points, then we can rectify our current pose estimation and estimation of our poses in between the current time and the time when we last corrected for drift. In other words, when the robot revisits a previously mapped location, its current scan should align with stored map features. By detecting that "I've been here before" and computing a global registration between the current LiDAR (or stereo) scan and the earlier submap, we obtain a highly reliable relative transform to correct our current and recent poses and LiDAR scans. This is known as **loop closure**. Note that this is the same matching and localizing as in consecutive scan alignment, but with the additional step of correcting past estimates from this new information.

Localizing a robot within a map using loop closure and/or basic odometry and scan matching is known as **Simultaneous Localization and Mapping (SLAM) (**Smith & Cheeseman, 1986). Note SLAM does not mean you are simply localizing and also mapping at the same time, it requires the use of mapping to aid in localization in some way.

The formal definition of the SLAM problem is to simultaneously estimate a map of the environment and the robot's trajectory over time, given a sequence of sensor measurements $z_{1:t}$ (e.g., LiDAR, vision, sonar) and control inputs $u_{1:t}$ (e.g., odometry, IMU). At any time t, the robot must infer its pose x_t and the map m, which may be represented as a set of landmarks, an occupancy grid, or a point cloud. Since both the robot's motion and its observations are uncertain, SLAM is framed probabilistically: the robot seeks to estimate quantities such as the expected values or maximum a posteriori (MAP) estimates of the joint distribution $p(x_{1:t}, m \mid z_{1:t}, u_{1:t})$. (7.1)

7.2 - Iterative Closest Point (ICP)

In the last section we established the goal for an algorithm that can analyze overlapping 3D point generating scans to match points and estimate the pose of the robot accordingly. We will first focus on the case when this scan sensor is a LiDAR sensor.

An image of two point clouds before alignment and the resulting point cloud after alignment for an outline in a 2D map is shown below:

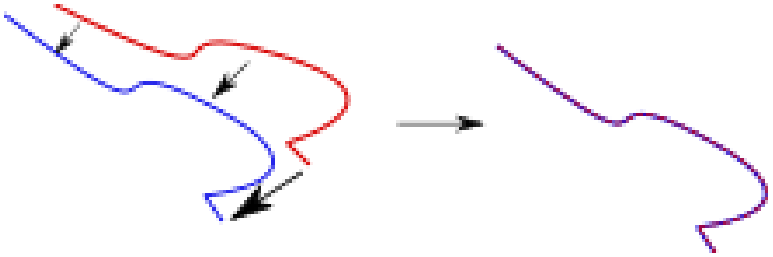


Fig. 7.1: The left image shows a new scan (red) being corrected to an existing map (blue) resulting in the fused map (purple) on the right

We assume that at some initial time we used some sensors to figure out the pose of a LiDAR sensor which collected the distance and angle of points on surfaces in its vicinity. The robot fused this data to generate an accurate point cloud of surface points and their corresponding locations called the **reference point cloud.**

Then the robot moved to another point, thus performing some translation and rotation of itself and its LiDAR sensor. Again, the LiDAR data and robot pose estimation sensor data was fused to generate a fairly accurate point cloud of surface points and their corresponding positions called the **source point cloud**.

We assume that there are some points that were captured solely by the reference or source point clouds individually, but not by both, and a significant number of points captured by both point clouds. Since the source point cloud was collected after the reference point cloud, and error in our LiDAR pose compounds over time, our LiDAR pose estimation is less accurate for the source point cloud. This means that our estimation of the position of each source point cloud point has more error than the reference point cloud estimations.

Inaccuracies in LiDAR pose estimation means our estimation for the position and orientation of the LiDAR sensor is not accurate. This means that to minimize inaccuracies in source point cloud estimations, we must perform some rotation and translation on the estimated pose of the LiDAR sensor to get the true LiDAR pose.

At its essence, aligning two point clouds means that their overlapping points are at the same locations. This means that we must find the rotation and translation to perform to the source LiDAR pose, equivalent to performing the transformation on source point cloud points, to minimize the location discrepancies between overlapping points.

The transformation equation from old source points to the transformed new source points follows the regular linear transformation equation:

$$p_i^{transformed} = Rp_i^{src} + t \tag{7.2}$$

As we know, the rotation matrix is made up of three rotation angles. These angles represent the rotation around the x, y, and z axes, respectively. These rotations are usually represented using Euler angles. The translation vector is made up of the translation along the x, y, and z axes, respectively. This means that we have 6 total parameters which define our transformation.

θ_x: Rotation around the x-axis

θ_y: Rotation around the y-axis

θ_z: Rotation around the z-axis

t_x: Translation along the x-axis

t_y: Translation along the y-axis

t_z: Translation along the z-axis

We assume that there is some unknown optimal R matrix and t vector that we can apply to our estimated points to maximize map cohesion and alignment. This means that there is some optimal set of those 6 parameters. This reduces our task to an optimization problem for 6 variables.

As we know from prior sections on optimization, such as gradient descent in chapter 11, since we have an equation that is a function of parameters (our 6 variables) mapping input (pre-transformed points) to output (transformed points), we can optimize the parameters. All we need is an error function that is a function of the input, output, and parameters. Then we can find the set of parameters which minimize this function.

To align the maps means that shared points are aligned in both point clouds. This means that the points which are captured by both point clouds are close to each other. This means that the error of a source point cloud transformation is how much the points in the estimated source point cloud which represent the same location in the environment as points in the reference point

cloud differ positionally from those corresponding reference point cloud points. This means that we must find all pairs of overlapping points, calculate the error in each pair, and sum the calculated errors of each pair for all corresponding pairs in the point clouds to get the total error of the source point cloud.

To get the error in a source point cloud, we sum the squares of the Euclidean distances between corresponding points in the reference and source point clouds. This involved subtracting corresponding points' position vectors and taking the square of the resulting vector length. This directly measures how well aligned the corresponding points are by summing up these squared differences. This method is called **Sum of Squared Differences (SSD)**. The following equation for the error of the source point cloud is as follows:

$$E = \sum_i ||p_i^{ref} - p_i^{src}||^2 \tag{7.3}$$

Using our known formula for source cloud points given our original transformation estimates, correction rotation matrix, and correction translation matrix, the source point cloud error is as follows:

$$E = \sum_i ||p_i^{ref} - (Rp_i^{src} + t)||^2 \tag{7.4}$$

This shows us that we can make some initial guess about the corrective rotation and translation, substitute the inputs, parameters, and outputs into the gradient formula for error with respect to t and with respect to R, update the parameters by the respective gradients multiplied by the learning rate, repeat until a certain number of iterations have been completed or there is minimal improvements between iterations. This means that we find matching points, use the discrepancies to calculate the partial derivatives with respect to error, adjust the corrective rotation matrix and translation matrix a little bit, update our source point cloud, find matching points, etc.

Initially, we do not know anything about the needed rotation or translation. Also, there may be no needed corrective translation or rotation, meaning our original estimates were correct. Thus, our initial rotation estimate is the identity matrix, which implies no rotation, and a translation vector equal to the **0** vector, which implies no translation. This estimate for rotation and translation gives us our initial estimated position of each point. From here we perform the small optimization adjustments via gradient descent. Our final source point cloud estimation is codified onto the map.

The partial derivative of the error with respect to the translation vector and rotation matrix are as follows:

$$\frac{\partial E}{\partial t} = -2\sum_i (p_i^{ref} - (Rp_i^{src} + t)) \tag{7.5}$$

$$\frac{\partial E}{\partial R} = -\ 2\sum_i (p_i^{ref} - Rp_i^{src} - t)\, p_i^{srcT} \tag{7.6}$$

$$t_{new} = t_{old} - \eta \cdot \frac{\partial E}{\partial t} \tag{7.7}$$

$$R_{new} = R_{old} - \eta \cdot \frac{\partial E}{\partial R} \tag{7.8}$$

Where η is the learning rate

Note that in real-time applications we often use other optimization methods such as **Levenberg-Marquardt** or **Gauss-Newton** which take advantage of second-order derivative information. This enables larger, more efficient updates compared to gradient descent. This leads to faster convergence, which is crucial in reducing the computational intensity and improving the speed of ICP, especially when dealing with large or complex datasets.

To perform this gradient descent, or other optimization, we must first find all overlapping points. This is called the **correspondence search**. In terms of determining how well two points align/correspond, we need some measurement of alignment. A feature, surface, edge, etc. that shows up in both point clouds will likely show up in a similar manner. This is because the points and their neighbors represent the same region/location. One approach is to operate on the principle that corresponding points between two clouds should be spatially close to each other in both point clouds. This means that a point in the reference cloud at a specific absolute reference frame location will be at a relatively similar absolute reference frame location in the source point cloud. This means that we would measure closeness as the Euclidean distance between positions.

To avoid asymmetry in point correspondences where multiple points in one cloud are matched to a single point in the other cloud, a point in the reference cloud and a point in the source cloud are only considered corresponding if they are both the closest points to each other. This is called **mutual closest point matching**. Avoiding many-to-one correspondences keeps your result robust, accurate, and faithful to the true shape of both point clouds.

To perform mutual closest point matching, for each reference point, find the most aligning source point. For each matched source point, find the closest reference point. If this reference point matches the original reference point, the pair is considered a valid correspondence. Apply this mutual validation process to ensure that correspondences are bidirectional and unique.

This process also enables us to easily discard outliers or points that are in 1 point cloud but not the other. Outliers—whether they come from sensor noise, dynamic objects (like moving people), or spurious reflections—can skew your estimate of the rigid transform or the surface you're trying to reconstruct. You don't waste time processing or smoothing over noise, and you end up with a more faithful, easier-to-interpret reconstruction. So, if a pair of points are part of a corresponding pair that is not very close/aligned, we discard the pair as the points do not likely

truly correspond to each other. This is done by setting a closeness threshold. If a pair does not meet this threshold, most likely at least 1 of the points is an outlier or only in one of the point clouds.

This overall algorithm is known as **Iterative Closest Point (ICP)** (Besl & McKay, 1992; Chen & Medioni, 1991). Overall, we can get an initial location estimate using IMU and encoders, then use ICP to align the point clouds and improve our localization and mapping.

Section IV: Computer Vision

Chapter 8: Image Processing

8.1 - Introduction to Computer Vision

A key goal in robotics is effective navigation through real-world environments. To navigate successfully, a robot must adapt its strategies based on its surroundings and the objects within them which can vary in location, material, and behavior. For instance, a robot may need to alter its path when encountering a movable object versus a stationary one. Similarly, a manipulator robot tasked with sorting different fruits, such as placing oranges in one container and apples in another, requires the ability to distinguish between them. However, these object categories are not inherently observable, as discussed in classification. To utilize prior knowledge about these entities, the robot must classify them using sensor data and machine learning techniques. For example, it would classify a fruit as either an apple or an orange based on the data collected from its sensors.

To classify physical objects, robots require distinct physical information, such as shape, color, and texture, which differentiates one object from another. These physical properties must result in measurable energy differences that can be detected and interpreted by sensors. Every sensor works by gauging some form of energy, whether mechanical, chemical, electrical, thermal, etc. and converting it to electrical energy that can be read by a computer. The computer then uses the electrical energy reading to discern the value of the initial energy difference we are concerned with. A strain gauge, for instance, is just a resistor whose resistance changes when it's stretched; you power it with a small excitation voltage and measure the resulting voltage drop. Mechanically deforming the gauge changes how much electrical energy is dissipated, and that change is what you read out.

Moreover, relying on only one or a few data points for an entire object may be insufficient for accurate classification. The wide range of object classes and the variability within each class make broad characteristics like height or weight often inadequate. Analyzing an entire object as a single unit can obscure the distinct characteristics of its individual parts. Many objects consist of smaller, functionally and visually different components that contribute significantly to their overall classification. For example, a chair includes legs, a seat, and a backrest, each with varying physical properties such as material and shape. A single, aggregate measurement of the object would fail to capture the differences between these parts. In this situation, we are unable to determine the effects of different aspects of the robot. Any measurement grouping together different points on an object blocks the robot's ability to analyze the points independently. Similarly, both a large bowl and a helmet may share comparable overall dimensions, yet their purposes are vastly different; it is the specific features of their components, such as the shape of the interior or the presence of straps in the helmet, that define their classification. Therefore,

accurately determining an object's class necessitates analyzing its individual components and understanding how these parts are arranged in relation to one another. This means that each point on an object must be sensed such that information about it can be discerned and separated from information about surrounding points.

Also, for efficient navigation, robots must be capable of quickly identifying multiple objects in their surroundings in situations where there are multiple relevant objects. To ensure that important information is not missed, robots must sense all relevant data across an entire object, avoiding gaps in their understanding due to incomplete data capture. Moreover, robots often face challenges where objects are located in difficult-to-reach positions, such as on top of a building or behind other obstacles. These constraints make it impractical for the robot to physically navigate to each object for closer inspection, as doing so restricts which objects it can analyze, the number of objects it can analyze at one time, and the portion of each object it can observe at a given time. Therefore, robots must be able to sense these physical properties for all points on an object from a distance.

To detect energy differences from an object at a distance, the necessary energy must propagate from the object to the observer. This can occur through energy emitted by the object or reflected from it. However, most objects do not naturally emit light, and attaching different emitters to all objects is impractical. Therefore, these objects must passively transmit energy by reflecting it. Consequently, we rely on the differences in how various objects reflect energy to distinguish between them.

The primary type of physical energy with reflective properties is waves. For effective object classification across various locations and conditions, these waves must consistently be present and possess stable properties. Electromagnetic waves are the primary type that meet these criteria, with nearly constant sources such as the sun and artificial light bulbs. The sun emits the most intense radiation in the visible light range, concentrating most of its energy between 400 nm (7.5×10^{14} Hz, violet) and 700 nm (4.3×10^{14} Hz, red). This range of light is referred to as the **visible light spectrum (VLS)**.

Due to variations in shape, size, and material, different objects reflect light differently. Each small point on an object exhibits varying levels of reflectivity across the light spectrum, which is referred to as that point's luminosity or **brightness**. This corresponds to the intensity or amplitude of the reflected light at that point. Additionally, each point reflects certain frequencies within the VLS more strongly based on its material properties. The predominant frequency of light reflected from a point is what defines its **color**.

Different objects consist of varying numbers of points, arranged in unique shapes, with distinct spatial patterns of brightness and color. The brightness and color of each point on an object are not independent of other points, but instead form a coherent pattern that is based on the object's shape, material, and structure. These patterns are defined by the position of each

point in relation to others across the object's surface. These comparative similarities and differences between neighboring points helps define the boundaries and details of objects. A point's brightness or color compared to adjacent points can reveal edges, textures, and features.

Since object classes are generally defined by utility and function, objects within the same class tend to share common patterns in shape, size, and surface material. This leads to similar reflective properties, resulting in consistent spatial patterns of brightness and color across different instances within the same class. These shapes cause light to reflect off their surfaces in predictable ways. For instance, the backrest of a chair will likely reflect light differently from the seat or legs. The spatially relative brightness and color patterns across these areas help distinguish a chair from another object, like a table, based on how the different parts of the object reflect light and how those parts are arranged. For example, at the boundary between two surfaces (like the edge of a table), there will be a sharp change in brightness or color between neighboring points, making the edge visible. Without this contrast, the edge would blend into the background.

Therefore, effective object classification can be achieved by providing algorithms with data on the spatial relationships of brightness and color in the light reflected off these objects. This general field of using visual data to gain a better understanding of the world is known as **computer vision**.

Since objects/physical materials generally absorb light, all obstructed light is unable to be perceived by sensors on the robot. This also means that no light will reach the insides of objects because it will be blocked or reflected by the surfaces. Since light travels in a straight line, emitted light without a straight path to the robot is not able to be perceived. This means that robots can only capture data from the surfaces of objects from which there is a direct line to the robot.

Due to the motion of objects as well as the robot, the light reflecting off of objects to the robot is constantly changing over time. This means that the relative spatial positions of objects and their points relative to each other as well as the robot change over time. To be able to adequately capture the position of points relative to each other as well as the robot, we must capture each point simultaneously and at a single point in time. If we group or average times together, we lose vital information as to the magnitudes and colors at different instances in that time interval. Every snapshot contains a unique set of information about the scene at that moment—object positions, lighting, shadows, colors, and textures. Therefore, we must collect each different light ray hitting the robot at a single point in time.

8.2 - Cameras

The goals established in 8.1 entail that we must collect and process the continuous spectrum of light available to the robot. We also established that we must perceive each point

independently. This means that we must have some form of transductive sensor which converts light energy into electrical energy which can be processed by a computer/processor. We must implement these sensor goals via physical hardware.

To capture light magnitude, we need a mechanism that measures brightness at a point. This is achieved through the photoelectric effect, where photons striking a semiconductor generate an electrical current proportional to light intensity. This relationship enables the creation of a sensor called a **photodiode**—a semiconductor device that harnesses this effect. By measuring the resulting current, photodiodes quantify light levels at specific points, effectively measuring brightness.

Photodiodes measure light magnitude but not color. Color is important for distinguishing materials and identifying object classes. Light consists of multiple superimposed wavelengths, and as an additive medium, different wavelengths combine to form new colors. A wave's color corresponds to its dominant frequency—the one with the highest intensity. By measuring brightness across frequencies, we can identify the dominant one as the perceived color.

To measure color, we must separate light into its component wavelengths using **color filters**—materials that transmit only a specific range of frequencies while absorbing others. For instance, a red filter blocks blue and green light, allowing only red to pass. By placing such filters over photodiodes, we can measure the intensity of individual color components at each point.

However, the spectrum of VLS frequencies is continuous. This means that to get the color despite the frequency, we cannot try every frequency. This means that we must use a finite number of filters such that each frequency has a unique position in the color space.

To capture a wide range of colors efficiently, we seek a minimal set of filters that provide good spectral coverage. A practical solution is a **trichromatic system**, using three filters—red, green, and blue—corresponding to long, medium, and short wavelengths. This setup enables color representation as a linear combination of the intensities measured through these filters, leveraging the additive nature of color mixing.

The relative proportions of the red, green, and blue values denote the perceived color, while the magnitude of the RGB vector reflects the overall intensity of the light. This means that we can efficiently store the brightness and color information of a light wave by capturing its red, green, and blue components, resulting in the **RGB Color Model**.

Since we established that each point must be perceived independently, the line reflected from each point, or a small area, on an object must be perceived independently. We cannot simply group light waves from vastly different points together because we lose granular information. This means that we cannot group together significantly differing points in space.

To have something physical only be contacted by a singular light ray, we cannot allow all possible angles of light relative to the robot to strike the sensor. Each diode should only be struck

by a singular angle or small range of angles. This means that our arrays of diodes should each be struck by a different angle of light relative to the robot. However, any significant space opening by the diode will allow multiple rays to strike each diode. So, we must have a very narrow opening for light. To allow different angles of light to spread out, our sensors must be behind this opening by some non-negligible distance. By doing this, we preserve the geometric structure of the scene and moment.

Since light travels in straight lines, light rays reflected from various points in a three-dimensional scene can pass through this small aperture and project onto a two-dimensional surface to form an inverted image on the opposite side. Because each point on the image corresponds to a unique direction in the scene, the mapping maintains the spatial relationships and proportions of the original scene. By placing an array of red, green, and blue filtered photodiodes on this wall, we can record the brightness and color information at each point, effectively capturing the image. This light capturing device is known as a **camera**, with the specific model described above known as the **pinhole camera model**. A visual representation of its principles is shown below:

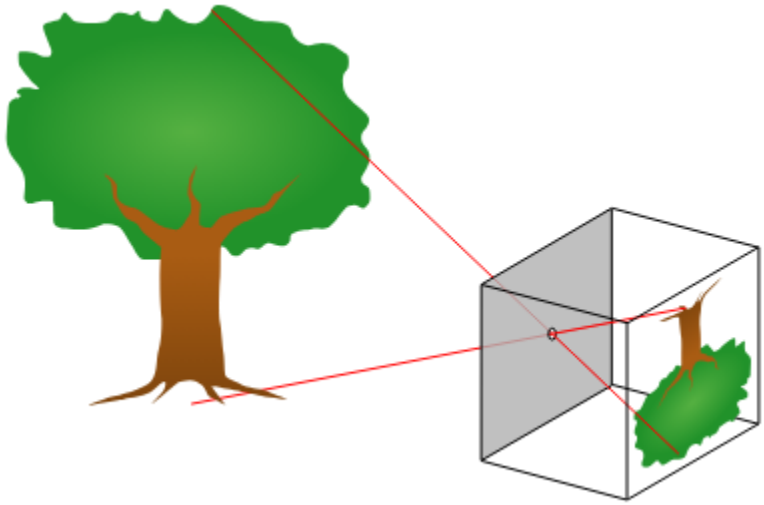


Fig. 8.1: Illustration of the pinhole camera model. Light rays from a 3D object (a tree) pass through a single focal point (the camera center) and project onto a 2D image plane, forming a perspective image. [Wikimedia Commons].

In essence, all cameras operate based on these fundamental properties of light. Lenses are introduced in practical cameras to gather more light and focus it onto the sensor, enhancing image brightness and clarity. They can also adjust focus and magnification, allowing us to capture detailed images of specific parts of a scene. However, the basic principles of projecting a three-dimensional scene onto a two-dimensional plane and capturing spatial relationships remain the same.

Now that we are able to get the color and brightness of an area, we must figure out a way to store this information while preserving the patterns and data that we want to abstract on. The world presents itself as a continuous flow of light and color, varying smoothly across space. However, to process this continuous information using digital systems, we need to convert it into a discrete form that computers can handle. To do this we must figure out a way to store this information while preserving the patterns and data that we want to analyze.

Since the world itself is a continuous flow of light and color, that means given points the surrounding points will be similar in color and brightness with little differences. Even the edge of an image will always have another point closer to the edge with a similar color and brightness of the original point given the attributes of a continuous function. We assume that brightness and color changes will not be erratic in space and time.

This property allows us to discretize space, capturing the average color and brightness of small areas. Patterns will still appear because of the spatial relationships of brightness and color. If the intervals we decide are two large, we lose some of the spatial patterns of the scene of light; however if they are too small, the scene of light will take a lot of room to store.

This gives us a 2D array corresponding to the back of the pinhole camera, with each individual square section of the 2D array being known as a pixel. An image, in the context of digital imaging, as discussed above is a collection of pixels arranged in a grid. Each pixel represents a sampled value of the light intensity or color at a specific point in the scene. By organizing these pixels in a structured manner, we preserve the spatial relationships and patterns inherent in the original scene, allowing for effective analysis and processing. A visual representation of an image is shown below:

Fig. 8.2: Pixelated image representation of a rocket ship in flight, composed of a 2D array of square pixels. Each pixel encodes color and intensity information at a specific spatial location, preserving the structure of the scene. This grid-based format mirrors the discrete sampling of light that occurs at the image plane of a digital camera, as described in the pinhole camera model. [Computer History Museum, 2022]

An image can be viewed as a discrete 2D function of n×m pixels, f(x,y), where n is the number of rows (height) and m is the number of columns (width) in the image. The function f(x,y) returns the RGB value of the pixel at the coordinates (x,y). Additionally, images can be represented as 3D arrays, where the third dimension corresponds to the color channels (Red, Green, and Blue), with each channel having a value at each pixel location.

Each x, y corresponds to a specific angle of light. In order to understand the spatial relationships of an image, including what areas they capture, the relative positions of objects/points, etc. we must know the ray corresponding to each pixel. In cameras, the x axis is the horizontal axis going left/right through the center of the pinhole, the y axis is the vertical axis going up/down through the center of the pinhole, and the z axis going out/in the center of the pinhole. This is shown as below:

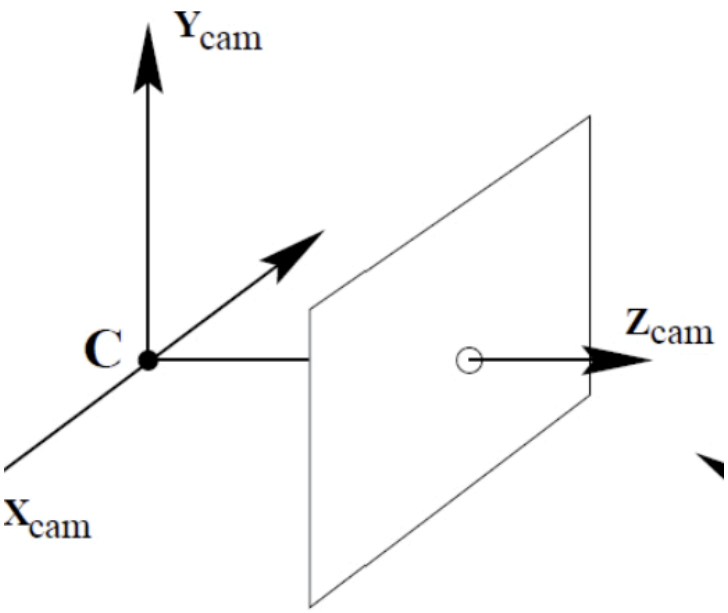


Fig. 8.3: Coordinate frame of a pinhole camera model. The origin C represents the optical center of the camera, with the Z_{cam}-axis pointing forward along the optical axis, X_{cam}-axis pointing to the right, and Y_{cam}-axis pointing downward or upward depending on the convention. The image plane is shown in front of the camera for visualization clarity, though it is typically located behind the optical center in a physical model. [Aldea, University of Pavia (n.d.)]

8.3 - Convolutional Neural Networks

8.3.1 - Introduction to Image Classification

Robots must perceive, understand, and interact with their environments autonomously. To make informed decisions and execute appropriate actions, they need to accurately identify and classify both objects and surroundings. For example, a warehouse robot might need to package items differently based on their type or material—such as distinguishing between a glass vase and a wooden chair to handle each one correctly. Similarly, a robot navigating an outdoor area may need to classify the terrain as either grass or concrete to adjust its navigation strategy. These examples underscore the importance of classification for robots, both for understanding their environment and for managing the objects within it.

To achieve this, robots rely on various sensors, with cameras being the primary tool for capturing detailed information needed to differentiate between classes of physical objects and environments. Just as humans use their vision to classify what they see, robots use images captured by cameras to perform similar tasks. The process of interpreting an image to determine its category is known as **Image Classification**. Robots also must be able to give numerical quantifiers to what it is seeing, such as determining the number of red balls in an image. For

example, determining that there are four people, three apples, and 0 oranges in an image. The general task of identifying, classifying, and quantifying objects in an image is known as **Object Classification**.

This means that a robot must be able to perform classification and regression on images. Since computer vision concerns itself with classes and categories that are primarily visually defined, it is valid to assume that there is a true function mapping images to a class or number. Since object classes are complex and inherently unobservable, we must use machine learning. Since these functions have many inputs, are very complex, and have many differing patterns and relationships, we use a neural network for this function approximation. Since an m x n image is simply a 3 x m x n tensor (in this case a 3D array) of data, we can train a NN to take in images as well as ground truths and learn the function.

Neural networks as we have talked about require a 1 dimensional input, where each node corresponds to a particular input variable. In order for each image to be taken as input to a NN, the image must be an n x 1 vector/array. This means that the tensor must be flattened to have 1 dimension.

We flatten an image into a 1D array such that each index consistently corresponds to the same color channel at the same (x, y) location across all images. For example, if index 5 stores the red intensity of pixel (1, 2) in one image, it must do so in all images. This consistency ensures that each input node always represents the same type of information, enabling reliable function approximation.

To flatten an image:

- Separate the 3 colors of each image into 3 matrices
- Concatenate the rows of each matrix to get a 1D array of pixel values for each color
- Concatenate the color arrays in the same order for each image to generate a full image array

Since the flattening process follows a consistent order—row by row within each color channel and a fixed color order across channels—each index in the final array corresponds to the same color and (x, y) coordinate across all images. Now the image is ready to be passed to the NN.

8.3.2 - Convolution Layers

In traditional neural networks, we assume each input corresponds to distinct, independent data points. However, image data is inherently spatial and is not classifiable by the absolute position of pixel values, but by the relationships between nearby pixels. For instance, consider two images of the digit “8” written in different parts of the image frame. Although the exact pixel values may appear at different absolute positions, the local pattern—such as two loops stacked vertically—remains the same.

In other words, an image of an object should be recognizable whether the object appears in the top left corner or the center of the image. For effective image classification, we desire properties like the ability to recognize an object even if it is shifted or translated in the image, known as **translational invariance**. For example, an image of a cat where the cat is translated 20 pixels to the left or is facing the other way should still be classified as a cat. These desired properties imply that the location of data in the input image should not affect how it is processed by the neural network. We want the network to generalize patterns learned from objects in one part of the image to classify objects in other parts of the image as well.

Each section of an image can contain important features that contribute to the overall understanding of the scene. For example, edges, textures, and corners can be crucial for identifying shapes and objects. Different regions of an image may contain different features relevant to the task (e.g., detecting an object). Therefore, multiple neurons in the hidden layers are needed to extract each feature from each input region.

In a **fully connected (dense) layer** that traditional NNs have, each neuron in the next layer connects to every current layer neuron. When the input size is large, this results in a massive number of weights. For example, an input layer with 10,000 pixels and a hidden layer of 100 neurons would require 1 million weights. In many structured data problems, this is unnecessary—predicting house prices might only require 10–20 features. But images are different: a 640×640 RGB image has 1,228,800 input values, making dense layers computationally expensive.

In traditional settings where each input corresponds to a fixed type of information, detecting a specific feature typically involves analyzing a specific subset of inputs. For example, a model predicting whether someone will become a multi-millionaire might identify high school academic performance as relevant and only need to examine inputs like GPA, SAT scores, and extracurricular involvement. In contrast, image data presents a different challenge: features such as eyes or noses can appear anywhere in the image. This requires the model to search for each feature across all locations, analyzing many different input subsets per feature. As a result, far more hidden neurons are needed to account for the spatial variability—for instance, detecting whether a face is present may require evaluating nose-like patterns at every position in the image.

The sheer number of parameters (weights) to be learned in these cases requires significant memory and computational resources. Training such a large model would take a considerable amount of time, and inference (the process of using the model to make predictions) would also be slow due to the high number of computations involved.

This shows that we must extract features without each input node connecting to each output node. Since objects in images are generally spatially continuous, the information needed to extract features from a given part of an image is usually contained within that specific region.

In other words, since image content is locally organized, the features of a particular area can often be extracted using only the nearby pixels. For example, to detect each individual lace on a football, the network only needs to analyze the pixels containing that lace. This means that we should instead connect x and y spatially adjacent patches of input pixels to each node in the subsequent layer. This means that we can have each output node in the subsequent layer connect to a spatially contiguous subsection of the previous layer. Connecting each node to a subset of nodes in the previous layer is known as a **sparse layer**. We can then have multiple subsequent layer nodes per previous layer region to capture the presence and magnitude of multiple different features in the previous layer region.

In order to allow this to detect the same features regardless of the location of the features in the input image, the same weights should be applied to each region of the input image regardless of location. When using multiple functions, this will be multiple weight sets, each one applied to a spatial region around each pixel in the input image. This technique is known as **weight sharing**. When updating these weights during training, we wish to ensure they learn effectively regardless of the location in the image. So, the gradient descent updates should take into account every instance of the pattern in the input data where the weights are applied. So, we take the sum of gradients for each weight across all locations before updating the weights.

This means that these layers will consist of multiple **filters**, called **convolution kernels**. that each slide over the input image to extract features. Each filter is a small matrix, to be able to detect small features, that learns to detect specific patterns, such as edges or textures. The matrix size should be small enough such that it is mostly taken up by the smallest objects. Depending on the task and image resolution, this can be anywhere from 3x3 to a width and height well into the double digits. Although inputs to fully connected layers are often flattened into a 1D vector, image data is naturally represented as a 3D tensor (multi-dimensional array) with dimensions corresponding to height, width, and the number of channels (e.g., RGB for color images). We preserve this 3D structure to take advantage of spatial relationships, rather than flattening the input. Each filter is a smaller 2D (or 3D if considering multiple channels) matrix that performs the convolution operation. The number of filters defines the depth of the output feature map.

The filter starts at the top-left corner of the input image and slides across it horizontally and vertically. Generally, the kernel will move over 1 pixel at a time, but occasionally to reduce computational complexity while still allowing all pixels to affect the feature map, it will move over two pixels at a time. The pixel number it moves is known as the **stride**.

When a convolutional filter is applied to an image, it slides over the image one region at a time. At each position:

- Element-wise multiplication is performed between the filter values and the corresponding input pixel values under the filter. Note that ‘corresponding’ simply means the pixel that

occupies the same relative position in the image region as the filter value does in the filter grid

- The results of these multiplications are summed together to produce a single number—this becomes one value in the output feature map.
- This value is then passed through a ReLU (or occasionally other) activation function, which sets all negative outputs to zero while leaving positive values unchanged

This process is repeated for every position the filter moves to, resulting in a **feature map** (or **activation map**) that highlights where the filter detected the specific feature it was trained to recognize. In general, high values in a CNN feature map indicate the presence of the pattern the filter is detecting, while low values (especially zeros after ReLU) indicate its absence. To detect different features in the input data, multiple filters are applied to the same input image in a convolutional layer, enabling the network to capture a diverse range of patterns and resulting in multiple feature maps. These layers are known as **Convolution Layers**. This is visually portrayed below:

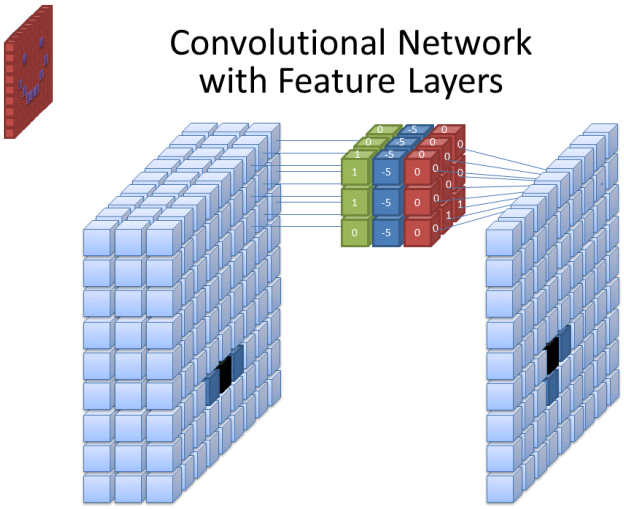


Fig. 8.4: A pictorial representation of a single convolution kernel in a convolution layer

In this image, the left block is the input volume. This is the input image, represented as a 3D tensor. The depth of this tensor is 3, corresponding to the RGB channels. Each small cube represents a single pixel value at a specific (x,y) location and color channel. The middle cube is a convolutional filter/kernel. It also has a depth of 3 to match the input's color channels. Each colored slice corresponds to one color channel's filter (green for G, red for R, blue for B). The numbers inside the cubes (e.g., 1, -5, 0) are the weights of the filter. These weights are multiplied elementwise with the corresponding pixel values under the filter's current position, and then summed together to produce one output. The right block is the output feature map resulting from the convolution. It's a 2D grid (since convolution collapses the color channels into one value per region). Each cube represents a single activation value, indicating how strongly the filter

detected its target pattern at that location. In this image, the dark blue cube represents a strong activation, meaning the filter found the pattern it was looking for in that region of the image.

However, these kernels only capture small features. To allow the network to capture larger images, we must also include kernels which take into account a larger number of pixels in the original image. The region of the original image that has the potential to affect the value of a node or pixel in a feature map in a convolutional neural network is called the **receptive field**. To perform this, we could have multiple different sizes of convolution kernels on the original input image. However, as discussed in neural networks, we must be able to efficiently model function composition. This is especially important because larger objects are generally made up of smaller objects. So, we must have the outputs of feature maps be taken as input to subsequent layers. This means that we generate a series of feature maps, then, to take into account the magnitude and presence of each feature, we pass a shared weight kernel across the 2D array of pixels that takes into account all features. This happens for multiple layers.

For example, in this network to detect faces in an image, the early layers might extract small and simple pixel value patterns such as edges or simple textures, middle layers might extract moderately sized and complex features such as, eyes, mouths, and noses, and later layers might extract large and complex features such as faces.

This means that if the previous convolution layer used M number of kernels to generate M feature maps (1 per kernel), where the kernels each have a K x K window size and C kernel depth (number of channels per kernel). This means that the kernels in the subsequent layer will have input size K x K x M. The kernel depth of the previous layer is condensed to 1 channel per feature map, thus not affecting the input size of the kernels in the subsequent layer. Since each kernel takes input from all feature maps generated by the previous layer, the kernels in this layer will have a depth equal to the number of kernels used by the previous layer.

By taking into account the feature map values for adjacent convolution kernels, we take into account a larger receptive field. By using 3x3 kernels in each convolution layer, subsequent layers are able to capture information from progressively larger regions of the original image. This happens because each pixel in a convolutional feature map is a result of a weighted sum of a 3x3 region in the previous layer's feature map.

To illustrate, consider a pixel in the first convolution layer's output. This pixel is influenced by a 3x3 region of the input image. In the second convolution layer, each pixel is influenced by a 3x3 region of the first layer’s output. Since each pixel in the first layer's output was already influenced by a 3x3 region of the original image containing some pixels that did not affect the outputs of kernels centered on adjacent pixels, each subsequent layer’s receptive field grows by 2 pixels in each direction. This means that by the second layer, each pixel in the feature map is influenced by a 5x5 region of the original image, by the third layer a 7x7 region, and so on.

This is visually portrayed below where the middle pixel in layer 3 is affected by a 3x3 square of the previous layer, which is in turn affected by a 5x5 square in the previous layer. Note that the red 3x3 in the leftmost grid corresponds to the region affecting the red square in the middle image.

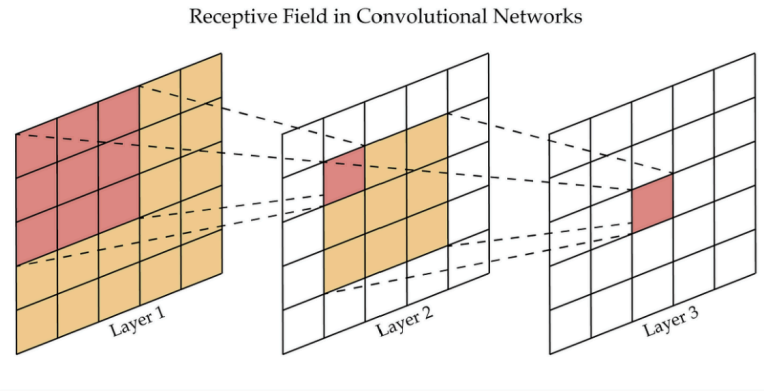


Fig. 8.5: A pictorial representation of changes in the receptive field of pixels as layers progress

Due to the fact that we cannot have kernels centered on the edges of images, our feature maps get smaller as the layers continue. This loss in size can also be caused by using a stride greater than 1. To prevent the depth maps from getting too small, we often add zeros around the border of the input. This allows kernels which would previously have been cutoff to still capture regions that may have features.

Convolutional layers excel at extracting spatial features from an image, such as edges, textures, and object parts, by preserving the local relationships in the input image. However, by the end of the convolutional stack, we need to combine all these localized features to make a global prediction or classification. So, we must feed the output of the convolution layers to a fully-connected (dense) network. Note that because the features are largely extracted by the convolution layers, we only need a small number of hidden layers, minimizing the effect of a large number of inputs. Additionally, the total number of values in the last convolutional layer is typically fewer than in the input RGB image, due to progressive spatial downsampling across the network. This means that we will have a series of convolution layers, flattening, then a series of feed forward layers. This overall structure is known as a **Convolutional Neural Network (CNN)** (Fukushima, 1980; LeCun et al., 1989).

We first randomize the weight values in the convolution and fully-connected layers. Then, a series of images are passed through the network. Each image is passed through the convolution layers generating a series of output feature maps. The feature maps are flattened and inputted to the feed-forward NN. The NN propagates the inputs forward to generate a prediction. The error is calculated and propagated backwards to adjust the weights of the fully-connected

NN. The backpropagation continues backward to adjust the adaptive kernel weights. This happens many times until the model is sufficient.

For a sparse layer where each neuron in layer k connects only to a subset of neurons in layer k+1, backpropagation is performed in much the same way as in fully connected layers. The primary difference is that, when calculating the gradient for a specific weight, only the outputs of neurons to which this weight contributes are considered. To calculate the gradient of the loss with respect to each kernel weight, we sum the gradients from all locations in the output where this kernel weight was used.

8.3.3 - Pooling

If you have a CNN that takes in a 1200 x 1000 pixel images with two consecutive convolution layers with the first having 10 kernels, the second layer inputs (1200 * 1000) * (10) = $1.2 * 10^7$ neurons. That is a lot of data.

The amount of data makes the above CNN very memory intensive and computationally intensive. We want our CNNs to be able to handle high resolution images, many convolution kernels, and many convolution layers. Also, the high dimensionality increases the chance that the model memorizes noise in the training data (overfits), thus reducing its ability to generalize for future inputs. So, we must reduce the amount of data being passed through the CNN.

Since adjacent pixels are very close spatially, they will likely all represent the same feature or absence of a feature. So, we can group together adjacent pixels for each individual feature map and represent them by their mean. This transforms the feature maps to feature maps of smaller sizes, a process known as **downsampling**. To not reduce too much information, this usually comes in the form of 2:1 downsampling where each 2x2 feature map square (patch) is reduced to a 1x1 value in the resultant feature map. This process of downsampling each individual feature map for dimensionality reduction, while preserving most of the relevant information is known as **pooling**.

Each patch is reduced to an individual value equal to the mean of the feature map values that make up the patch. This process, known as **average pooling**, preserves information from each pixel in the patch and treats them equally. This also means that average pooling produces a smoother output image because it is not very sensitive to extreme outliers. The effect of extreme outliers is partially mitigated by the influence of other values.

However, due to the fact that high convolution kernel outputs signify the prevalence of a feature, large outliers can sometimes mean that the desired feature has been identified in a patch. We often wish to preserve high prevalence values as they are more informationally rich than low prevalence values. Average pooling reduces the effect of the large outliers. This means that sometimes average pooling reduces a model's ability to detect prevalent features. Since the highest feature map values entail the highest feature prevalence for a specific area in the input

image, downsampling is sometimes performed by setting the reduced feature map pixel value to the maximum value found in the corresponding patch on the original feature map. This is known as **max pooling**.

To prevent the convolution layers from generating too much data at any point in the CNN and ensure each layer analyzes a slightly larger scale than the last, each convolution layer is generally immediately succeeded by a pooling layer.

Usually, image input is fed to a few interleaved convolution and pooling layers (convolution→pooling→convolution→pooling, etc.), then the output of the pooling layer of the last convolution-pooling combined process is flattened and inputted to the fully-connected NN.

For max pooling, the only way backpropagation is affected is that we only propagate errors to weights in sparse layers/convolution kernels that led to nodes which 'made it through' the max pooling layer because they were the most prominent of the 2x2 square. Weights leading to nodes that were not passed through have an error of 0.

To keep track of which nodes are spatially related in the convolutional layers, we store the pixel or feature map values in a 3D tensor throughout the forward pass of the convolutional and pooling layers. This tensor contains the feature map matrices produced at each layer. By organizing the data this way, we can use raw pixel indices to determine which values are passed to each node in the next layer. After the final pooling layer, we flatten the tensor—feature map by feature map, row by row—into a 1D array of values. This process is represented by the CNN diagram below:

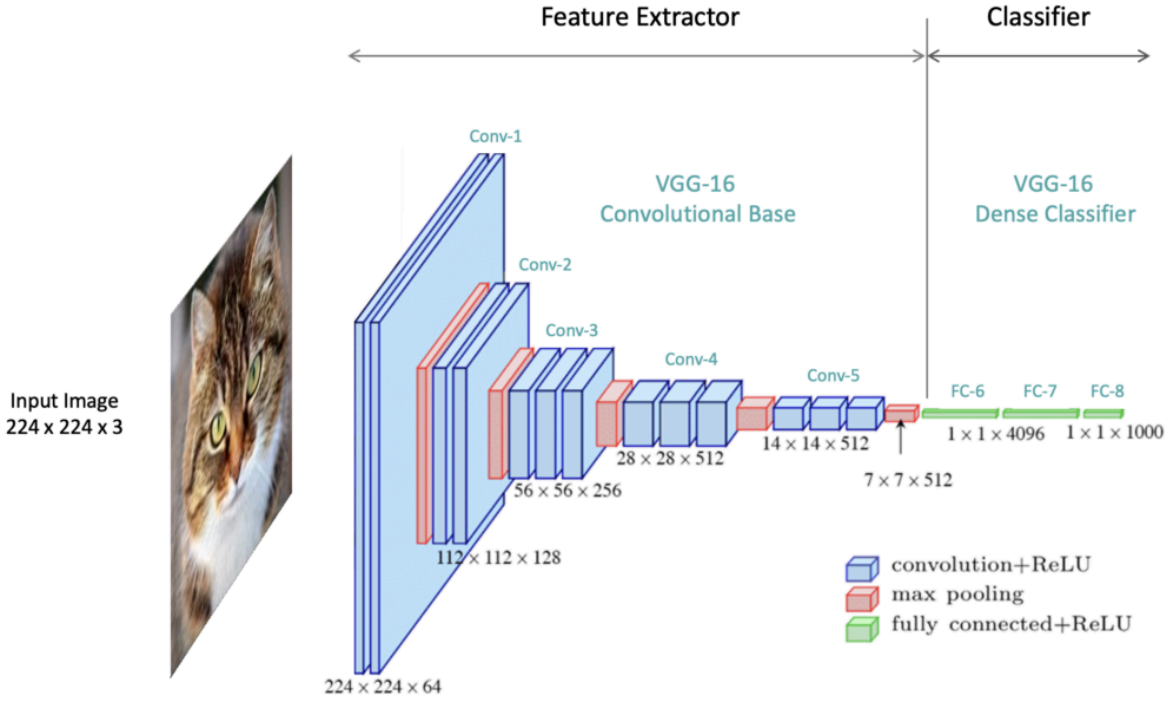


Fig. 8.6: A pictorial representation of an image being processed through a series of 5 convolution layers with decreasing height and width, but increasing depth, followed by flattening then 3 fully connected layers.

Chapter 9: Object Detection

9.1 - Introduction to Object Detection

Using image classification alone to determine what kinds of objects are present, known as **Object Classification**, has its limitations. In many situations, it's not enough to merely identify an object; the robot also needs to know where each object is located, either relative to the robot, relative to other objects in an image, or within a reference frame. This process is known as **Object Localization**. For example, autonomous cars must accurately identify and locate various street signs and obstacles, as each type conveys different information. Distinguishing whether an object is a person or another vehicle, or whether a sign is a yield or stop sign, leads to different decisions. Likewise, the car must respond very differently depending on whether a pedestrian is on the sidewalk or directly in front of the vehicle. Inaccurate detection can have serious consequences, potentially resulting in injury or even death. This underscores the need for robots to not only classify objects but also to accurately determine their location and size within an image. This is known as **Object Detection**.

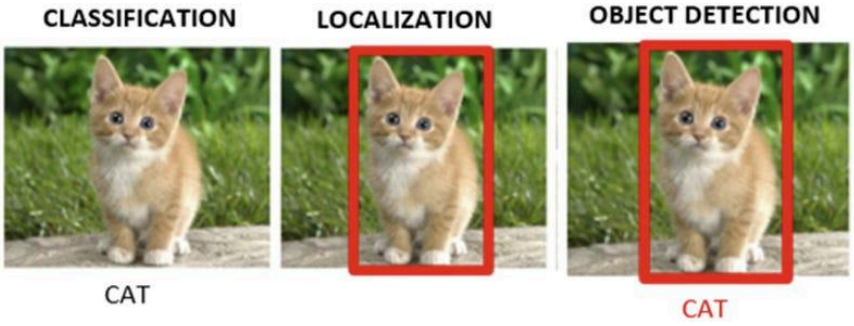


Fig. 9.1: A side-by-side comparison of classification, localization, and object detection for a single object in an image.

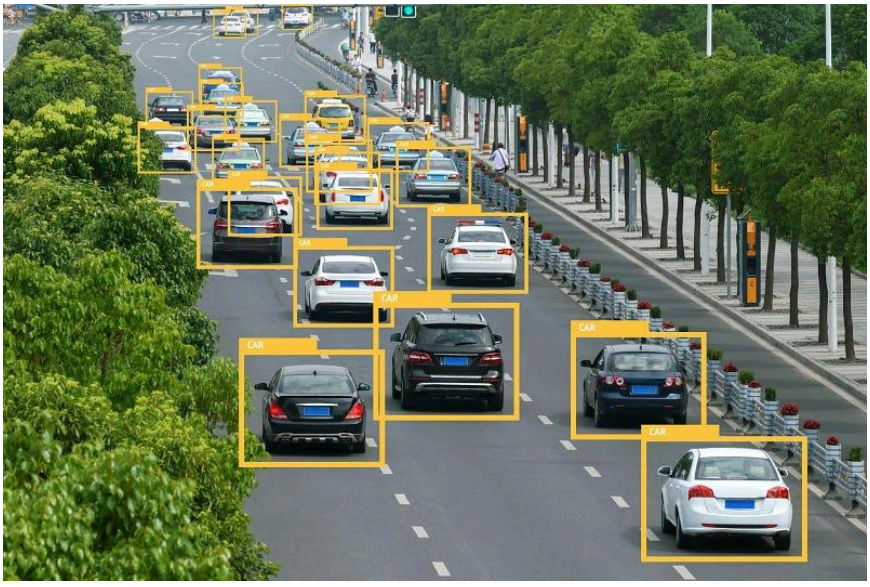


Fig. 9.2: Example of object detection in a real-world scenario, where multiple cars are localized and labeled within a traffic scene.

Moreover, image classification assigns a single label to an entire image, which can be insufficient for real-world applications involving multiple objects. For example, in a basic image classification task, a model might label an image as containing an endangered species. However, it would be far more useful to locate and classify each individual animal within the image to gain a complete understanding of the scene.

Lastly, classifying an image with one or more objects can be challenging due to the presence of irrelevant background noise. By focusing on specific regions that contain objects rather than analyzing the entire image, the task of classifying individual objects becomes much easier. Smaller regions minimize the influence of distracting elements and help the model concentrate on features relevant to each object. This need to isolate and analyze regions of interest, along with determining the size of each object, is a key motivation for incorporating object localization into the classification process.

9.2 - Sliding Window Approach

Since knowing an object's location helps with classification, we should first localize the object before attempting to classify it. Most objects don't take up the whole image, but rather a segment of the actual image. Therefore, to locate an object in an image, you simply need to figure out the segment of the image that contains the object. You could simply break down the image into a grid of nonoverlapping cells and analyze each cell, but this would not allow you to find objects that traverse multiple cells. In reality, objects in images can appear at any location, and their exact positions are initially unknown. So, to find the segments containing each object, you need to check all locations. This entails breaking down the image into all possible segments. Then you can analyze each for the presence and class of objects, usually with a CNN. After that, you can return segments containing an object of interest and the class of the object it contains.

Each object generally has some width and some height in an image, so a simple way to capture an object is with a fixed rectangular frame of pixels, called a **window**. So, we break the image down into all possible segments of the window's pixel width and height at different locations. This entails that each window, besides ones touching the outside of the image, will have a window 1 pixel to the left, 1 pixel to the right, 1 pixel above, and 1 pixel below. A picture of some image with an overlaid window is shown below:

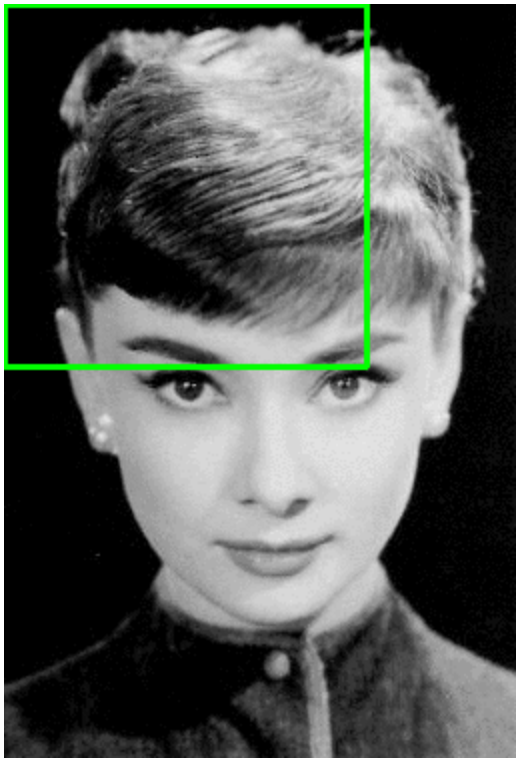


Fig. 9.3: A single green sliding window overlaid on an image as part of a classification process.

This approach, however, presents a small issue. Trying every single possible window is computationally intensive and likely redundant. It is very unlikely that shifting a window 1 pixel in any direction will have a negligible effect on a classifier's ability to detect an object in the window. Also, different objects captured by the image are almost always going to be multiple pixels apart. This shows that not all possible windows need to be processed. If a particular window is processed, it would only be beneficial to analyze windows a certain number of pixels above, below, to the right, or to the left. So, we leave a horizontal gap between a window and the closest windows at the same y position, and a vertical gap between a window and the closest windows at the same x position. This small, fixed gap is known as the **stride**. This is a hyperparameter that is much smaller than the window size and must be tuned. The horizontal and vertical strides are usually the same, but do not have to be, especially if the width and height of the image vary drastically. So, to get all segments to analyze, we use the horizontal stride, vertical stride, window width, and window height, to get a list containing the centers of each segment we wish to analyze. By using the window width and height, we can obtain the full frame of each segment and send the pixels contained in the frame through a classifier for analysis. If we have some center, we add and subtract window y value to the center height to get the y range of pixels. We add and subtract window width to the window center x value to get the x range of pixels.

For each of these windows, the classifier returns a **confidence level** for each object class that represents the probability the model believes that the region contains that type of object, which is signified by $P(c_i)$. However, there is a chance that the model contains no object. We could follow the approach of no object being one of the probabilities. Then the highest confidence class is believed to be contained in that segment.

However, determining whether a segment contains an object and which class of object at the same time is difficult. These are two fundamentally separate questions: object or no object, and which class of object. Attempting to combine them can lead to false positives where there is no relevant object, but because the segment may look much more like one object than any of the other objects, the model classifies the region as containing that specific object. Also, since class probabilities must add up to 1, a segment that looks similar to two competing classes could cause each of the two classes' confidence levels to be lower than the probability of there not being any object in the segment. Since the probability of there not being an object is the highest, the model will believe there is no object captured even when there is a high chance combined that the segment is just capturing one of the two competing classes. This shows that we really care about all the probability it's any class vs no class, instead of no class vs each individual class. These scenarios show that we want to separate the calculations of the probability of no object from the calculation of each individual class probability.

Also, in the case where there is a high probability there is no object captured in the segment, computing individual class probabilities doesn't matter. We only care about class probabilities if it has been determined that a relevant object is present. So, the model separately calculates the probability that the segment contains a relevant object, known as **Objectness** and signified by P(o), and the **Conditional Class Probabilities** that the segment contains each specific object class given there is an object in the segment, signified by $P(c_i \mid o)$ for class i. If the objectness is above a pre chosen threshold, the model believes a relevant object is present. The predicted class of this object is the class corresponding to the highest of the conditional class probabilities.

For a segment to contain an object of a specific class, two conditions must be met: it must contain an object, and that object must belong to the specified class. For a segment to contain an object of a specific class, it must first contain an object at all. Therefore, by the rules of conditional probability, the probability that a segment contains an object of a specific class is equal to the probability that the segment contains a relevant object, multiplied by the probability that the object belongs to that class given that it is relevant. This can be expressed as the product of objectness and the conditional class probability. This is described below:

$$P(c_i) = P(c_i \cap o) = P(o) * P(c_i \mid o) \quad (9.1)$$

When detecting objects, we only care about windows that contain objects. So, the model keeps track of the location of segments determined to have an object along with the respective conditional class probabilities for that window, thus cementing the window in the object detection output. This means that the model output will contain a series of descriptors, each containing, the box width (w) and height (h), a center pixel (x, y), objectness P(o), and a vector of conditional class probabilities P(**c** | o) where element i in the vector equals $P(c_i \mid o)$. Each of these output descriptors/frames are known as **Bounding Boxes**. An example of a model output is shown on the right. Yellow bounding boxes represent the locations of filled soda slots and pink ones represent the locations of unfilled soda slots. The percentages represent the class confidence level, the product of objectness and conditional class probabilities. The algorithm described above is known as **Sliding Window Approach** (SWA) (Rowley et al., 1998).

Fig. 9.4: Classification results on a vending machine image, with neon yellow windows indicating bottle presence and pink windows indicating empty slots. Confidence values are displayed on each window.

However, if a certain window detects an object, the nearby overlapping windows could also detect the object, as a slight shift might not significantly change the visibility of the object. This could result in both windows predicting bounding boxes that indicate the same object. This provides redundant information and means that all but one bounding box provides slightly inaccurate information about the object's location. We only want the bounding box with the best/correct position to represent our object's location. To ensure we only have this box, we want

to find instances of bounding boxes that indicate the same object's location and only keep the bounding box with the most accurate representation.

Bounding boxes that are capturing the same object's locations will capture a lot of the same area, resulting in some bounding box overlap. However, there could be instances of overlapping bounding boxes that are capturing two separate objects that are adjacent to each other rather than the same object. Since each object has one location, whereas separate objects have distinct locations, two boxes that represent the same object will generally have a greater proportion of overlap compared to boxes that represent different objects. So, we can distinguish between the two cases by calculating their intersectional area in terms of pixels. If two boxes have an intersection above a pre chosen threshold, then they are determined to represent the same bounding boxes.

After determining if two bounding boxes are capturing the same object's location using intersection scores, we want our model to determine which bounding box is the more accurate representation of the object's location. That information was already computed by the model in the form of the confidence level. This is because the model returns a high confidence level for a particular bounding box when it has a low amount of irrelevant background space/noise, captures all of the object's features, and has the majority of the area occupied by the actual object. Therefore, given a series of overlapping boxes, we can find the best bounding box by looking for the bounding box with the highest confidence level. This process of computing intersection scores and comparing confidence levels is known as **Non-Maximum Suppression (NMS)**.

Since the bounding box with the highest confidence level is considered by the model as the most accurate representation of the corresponding object out of all bounding boxes, we want to start checking intersection scores between that particular bounding box and every other bounding box determined to have some overlap. If the intersection score is above a certain threshold, the boxes represent the same object, so the bounding box with the lower confidence level is removed. Once all lower confidence overlapping boxes have been removed for the highest confidence box, we move the bounding box with the highest confidence level to a "checked" set and move on to the unchecked bounding box with the next highest confidence level. We would then compare intersection scores with the remaining unchecked bounding boxes. We iteratively perform this process until each bounding box is checked or removed. Typically, the list of bounding boxes is first sorted in descending order to quickly find the bounding box with the highest confidence level.

So, given the window width, window height, horizontal stride, vertical stride, confidence threshold, and intersection threshold, we can detect all objects in an image. However, the output of SWA is going to be heavily dependent on the chosen hyperparameters. For example, the size of the window is key for detecting objects of certain sizes. If the window is too big, some objects may only take up a small portion of the segment, resulting in imprecise localization and the

segment capturing a lot of irrelevant background along with the object. On the other hand, if the window is too small, the segment might only capture a portion of an object at any given time. This will cause the model to lack the complete set of features necessary for accurate detection and be unable to gauge all locations occupied by the object. Each application has its own optimal window size. Therefore, it is imperative to find the perfectly sized window that allows the model to find a segment that accurately captures all the locations occupied by an object with minimal irrelevant background.

An additional point we have to consider is that objects of interest in an image could have varying size. Different classes of objects oftentimes have varying sizes in an image, like a car and a penny. Objects of the same class could also have varying sizes, like a limousine and a microcar. Also, objects closer to an image take up more pixels than objects farther away. This means that each object will have a potentially distinct window size that can accurately capture the object's location with minimal background.

SWA has a limitation of a single fixed size for the window, which is used to create segments. However, objects of different scales require different window widths and heights. Finding each object's perfect window size is difficult, as the model doesn't know the size of the objects of interest prior to processing. So, we must try different window sizes on the entire image, and use the output for each size to choose the best one for each object. This means that each box descriptor will also contain the width (w) and height (h).

However, multiple window sizes will likely capture enough of the same object for detection, but not too much noise to the point where an object won't be detected. This means that we will have overlapping boxes. In this case, intersectional area no longer has the same meaning because the same intersectional area can be a lot for small boxes and not a lot for large boxes. Two boxes that capture the same object's location will share a lot of their pixels. If they don't, then they likely capture unique information. The whole purpose of NMS is to remove boxes that don't capture new information. This means that the magnitude of overlap is only relevant relative to the total number of pixels occupied by the two bounding boxes. If two boxes share most of their pixels, they are likely redundant, and vice-versa. So, we use the number of pixels captured by both boxes divided by the number of pixels captured by either box. This is known as **Intersection over Union (IoU)**. We can then use this in the NMS process. Also, the optimal size will likely entail the greatest prediction confidence, so we can use NMS to keep the best size. This means that we try a series of hyperparameter sets on an image, then use IoU and NMS to optimally fuse the output bounding boxes of each image and get one final output. This set of output bounding box labels can be compared to the ground truth to evaluate an output, and thus find the optimal set of hyperparameter sets to generate the series of images that are fused. This algorithm is called **Multiscale Sliding Window Approach**.

9.3 - Evaluating Object Detection Output

To achieve optimal algorithm performance, it is crucial to search for and determine the best set of hyperparameters, including both Sliding Window Approach (SWA) hyperparameters and Convolutional Neural Network (CNN) hyperparameters. This involves analyzing the performance of different models to select the best-performing one. To do so, we need a quantifier of how well the predictions match ground truth, known as a loss function. By minimizing this loss across training and validation sets, we identify the hyperparameter configurations that lead to the most accurate and generalizable model.

For performance evaluation, we need a series of ground-truth images with bounding boxes to compare against the model's final output. These bounding boxes, known as **Ground-Truth Bounding Boxes**, must be manually drawn to accurately fit the size and location of each relevant object in the image. In other words, they must tightly enclose the objects. Each ground-truth bounding box includes an object class label, position, and dimensions. Since these true boxes represent specific objects, the ground-truth vector for a bounding box will contain a value of 1 for objectness, 1 for the true class conditional probability, and 0 for all incorrect class conditional probabilities, along with the true x and y coordinates and the true width and height.

We want our predicted images to align closely with the ground truth images. This means our predicted bounding boxes should match the ground truth bounding boxes as closely as possible based on their description vectors. To evaluate the model’s performance and optimize the hyperparameters, we need a method to quantify the deviation between the predictions and the ground truth labels. To achieve this, we must define a **loss function**, as discussed in the chapter section on machine learning.

When predicting bounding boxes, the algorithm may successfully identify a relevant object, resulting in a correctly high predicted objectness score. This aligns the predicted bounding box with a ground truth bounding box. However, especially in the early stages of training, the algorithm might incorrectly believe an object is present, indicating that the predicted objectness was too high. This leads to a predicted bounding box that does not correspond to any ground truth bounding box. Additionally, the model may miss relevant objects entirely, suggesting that the predicted objectness was too low. In this case, ground truth bounding boxes will exist without corresponding predicted bounding boxes. Conversely, the model may accurately determine that no relevant objects are present in a region, resulting in an appropriately low objectness score. In this scenario, there will be neither predicted nor ground truth bounding boxes. Since this is the correct determination and there are no better predictions, true negatives do not contribute to the loss function. The four combinations of predicted and ground truth bounding boxes can be visualized in the confusion matrix below:

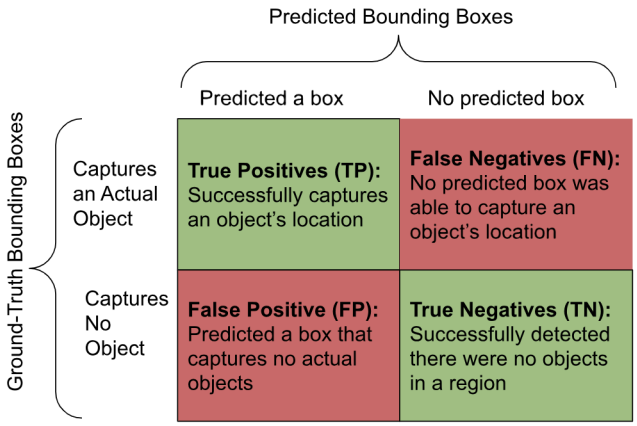


Fig. 9.5: Confusion matrix descriptions for combinations of ground truth bounding box presence and predicted bounding box presence.

Initially, the predicted and ground-truth bounding boxes lack explicit indications of their classification. We can't know whether a predicted bounding box is a TP or FP until analyzing the ground truth bounding boxes, and vice-versa. It is essential to distinguish between the three situations, true positives, false negatives, and false positives, as each requires different loss evaluations. True positives are identified by matching predicted bounding boxes with their corresponding ground truth boxes. Since a bounding box is intended to capture the location of a singular object, it should not be matched with multiple bounding boxes of the other type. This is known as a **One-to-One Matching Strategy**. So, once all ground truth bounding boxes with corresponding predicted boxes are identified, any remaining ground truth boxes are classified as false negatives. Conversely, after determining all predicted bounding boxes that match a ground truth box, any unpaired predicted boxes are classified as false positives.

To identify true positives and the pairs of predicted and ground-truth bounding boxes, we need to determine which pairs capture the same object's location. We already know how to do this from when we determined which predicted bounding boxes in the same image corresponded to each other in SWA to eliminate overlap. We can evaluate object locations across different outputs using IoU scores between the ground-truth and predicted bounding boxes. The model calculates the IoU score for each possible pair of predicted and ground-truth bounding boxes.

Since the highest overall IoU score indicates the strongest correlation between a pair of bounding boxes capturing the same object's location, we aim to match those two boxes together. Remaining pairs involving either bounding box in different combinations will have lower IoU scores and, consequently, weaker correlations. Therefore, we sort the IoU scores in descending order and match the highest IoU pairs of previously unmatched bounding boxes. When no further unmatched boxes have an IoU above some predetermined threshold, the algorithm stops. This approach to matching is known as the **Greedy Algorithm**.

For the true positives, we compare two bounding box description vectors: the predicted and ground-truth bounding boxes, focusing on localization, confidence level, and conditional class probabilities. This comparison forms part of the overall loss function, known as **Object Loss**. Since this loss can only be computed for true positives, we generally multiply the object loss by 1 for true positives and 0 for anything else. 1_i^{obj} is the piecewise function used to represent this, where i represents the i-th matched pair of predicted and ground truth bounding boxes. Based on the distinct aspects of predictions, the object loss must be further broken down into three, separately calculated sub-components: **Localization Loss**, **Objectness Loss**, and **Classification Loss**.

The first subcomponent of object loss is localization loss, which quantifies the positional error between the predicted bounding box and the corresponding ground-truth bounding box. Since bounding boxes are defined by four parameters (x, y, width, and height), the localization error is calculated by taking a linear combination of the Mean Squared Error (MSE) for each of these variables. However, variations in object and image sizes affect the scale of the width and height of bounding boxes. This means that the same percentage error for a large bounding box will result in a significantly larger absolute error than for a smaller bounding box, which can skew the evaluation in favor of larger objects. To mitigate this bias, the width and height are typically square-rooted before applying MSE. Assuming there are **n** bounding boxes, the localization loss function, with ground-truth bounding box values denoted by an arrow or hat above them, is represented as follows:

$$Localization\ Loss = \frac{1}{n}\sum_{i=1}^{n} 1_i^{obj}\left[(x_i - \widehat{x_i})^2 + (y_i - \widehat{y_i})^2 + (\sqrt{w_i} - \sqrt{\widehat{w_i}})^2 + (\sqrt{h_i} - \sqrt{\widehat{h_i}})^2\right] \quad (9.2)$$

The second subcomponent of object loss is the **objectness loss**, which measures the error between the predicted and ground-truth objectness values. This means that we should punish deviations of predicted objectness from ideal predicted objectness. An initial idea of ground-truth objectness is to train it to be binary: 1 if it contains an object, and 0 if it doesn't. Since this type of loss only deals with true positives and false negatives, all ground-truth objectness in this type of loss will be 1. However, many bounding boxes will have incorrect positions and sizes, leading to them only covering a portion of the object. A bounding box that captures 60% of an object should not have the same ideal predicted objectness as one that captures 99% of an object. A bounding box that captures more of the object should get a higher objectness. Additionally, a predicted bounding box that captures less noise should get a higher objectness prediction.

Instead, after we determine if the bounding box captures an object or not, we should see the proportion of the object captured to calculate objectness. We can quantify that proportion through IOU_{pred}^{truth} between the matched pair of bounding boxes. To combine both components,

objectness, in general, is trained to be the product of the two components: $P(O) \cdot IOU_{pred}^{truth}$. We're using the syntax of $\left[P(O) \cdot IOU_{pred}^{truth}\right]_i$ to show the expected objectness of the i-th predicted bounding box. For objectness loss, P(O) is 1, as it is guaranteed to capture an object, so objectness is trained to be $\left[P(O) \cdot IOU_{pred}^{truth}\right]_i$ for true positives and false negatives. Assuming there are **n** predicted bounding boxes, the objectness loss function is shown below:

$$Objectness\ Loss = \frac{1}{n}\sum_{i=1}^{n} 1_i^{obj}\left(\widehat{P(o)}_i - P(o)_i\right)^2 = \frac{1}{n}\sum_{i=1}^{n} 1_i^{obj}\left(\left[P(O) \cdot IOU_{pred}^{truth}\right]_i - P(o)_i\right)^2$$

$$= \frac{1}{n}\sum_{i=1}^{n} 1_i^{obj}\left(\left[IOU_{pred}^{truth}\right]_i - P(o)_i\right)^2 \quad (9.3)$$

The third and final subcomponent of the object loss is the **classification loss**, which measures the error between the predicted and ground-truth conditional class probabilities. The ground-truth conditional class probability is 1 for the correct object class, and 0 for all other classes. For each predicted bounding box, we retrieve the vector of ground-truth conditional class probabilities associated with the matched ground-truth bounding box. The predicted conditional class probability for the **j**-th class, given that an object is present, for the **i**-th predicted bounding box is represented as $\widehat{P(c_j \mid o)}_i$.

Given that there are **n** predicted bounding boxes and **C** possible object classes, the classification loss is computed by comparing the predicted conditional class probabilities for each bounding box with the corresponding ground-truth probabilities, penalizing deviations. The overall classification loss can then be expressed as follows:

$$Class\ Loss = \frac{1}{n}\sum_{i=1}^{n} 1_i^{obj} \sum_{j=1}^{C}\left(P(c_j \mid o)_i - \widehat{P(c_j \mid o)}_i\right)^2 \quad (9.4)$$

For false negatives, we have found a ground-truth bounding box that captures an object of interest and wasn't captured by our model. This means that this combination doesn't have a prediction box to compare our ground truth box with. However, it does deal with capturing an actual object, so it is fundamentally different from both true and false positives. In this situation, the model was unable to detect the object to begin with, which is measured by objectness. However, that isn't the sole source of error: the model could've failed to localize the object or predict the correct class either. There are several scenarios of the prediction, such as the model predicting the right object class and optimal localization but had a low objectness to make that prediction. In that scenario, the model should learn that that set of predictions for object class and localization was optimal and not make any major adjustments. In both true positives and false negatives, the goal is to have our model to learn to accurately predict all three variables of an

actual object: location, objectness, and class probabilities. However, for predictions that capture no objects (false positives), location and class probabilities are irrelevant. Therefore, we should consider this source of error for true positives and false negatives. All 3 types of error are part of **Object Loss**.

However, what do we compare the ground-truth bounding box with? There is no predicted bounding box that was matched with the ground-truth bounding box that are false negatives. Typically, the model will predict more bounding boxes than necessary in the first stages of learning to be able to learn from all of the ground-truth bounding boxes. It allows the model to learn which predictions were correct and which were incorrect quickly, as the model is able to try multiple guesses and see which ones were correct. This means that there usually is at least one unmatched predicted bounding box that we can use. If that's not the case, the model typically incurs a full objectness loss of 1 to make the model learn there is an object at that area. We can choose the most suitable predicted bounding box and compare it with the ground-truth bounding box, which will allow the model to learn to shift the predictions from a wrong location to the correct location. The most suitable predicted bounding box is the bounding box with the highest correlation with the ground-truth bounding box, which is measured by the IoU score. If there are no predicted bounding boxes that are overlapping the ground-truth bounding box, leaving it unmatched would be suboptimal because it would reinforce the model's behavior of ignoring that object class or region. Instead, we want to find the closest distanced unmatched predicted bounding box to match with, as it encourages the model to predict closer boxes. The chosen bounding box is then compared with the false negative ground-truth bounding box.

For false positives, we have found a predicted bounding box that doesn't capture any object of interest. This means that this combination doesn't have a ground-truth box to compare our predicted box with, so we won't be able to perform the same evaluation as true positives. In this case, the error is in the objectness, as the model determined the predicted bounding box contained an object when it didn't. Therefore, the error is placed in a separate loss function, known as **No-Object Loss**. 1_i^{noobj} is the piecewise function in the no-object loss that returns 0 if the i-th predicted bounding box does contain an object and 1 if it doesn't.

In No-Object Loss, since there is no object captured in the predicted bounding box and no ground-truth bounding box that was matched with the predicted bounding box to compare with, we can't compute localization or conditional class probabilities loss subcomponents. The main source of error for predicted bounding boxes that capture no objects was the objectness. Objectness is trained to be the product of the two components: $P(O) \cdot IOU_{pred}^{truth}$, but since it doesn't capture any part of an interesting object, P(O) is 0. The ground-truth objectness for this loss component becomes 0. We can compare the predicted with the ground-truth objectness to

punish high predicted objectness for boxes that captured no objects. Assuming there are n predicted bounding boxes, the no-object loss function is shown below:

$$No\_Object\ Loss\ = \frac{1}{n}\left[\sum_{i}^{n} 1_i^{noobj}(P(o)_i - \widehat{P(o)}_i)^2\right] = \frac{1}{n}\sum_{i} 1_i^{noobj}(P(o)_i)^2 \quad (9.5)$$

One question that could arise is that we already considered objectness loss in the object loss so why can't we combine them? When dealing with predicted bounding boxes that didn't capture any objects, we generally want those boxes to contribute higher to the overall loss function than the objectness loss function for predicted bounding boxes that successfully captured an object. We do so by assigning a higher weight on the no-object loss than objectness loss. Since each component has different meanings with varying importance to the error, the overall loss function is a weighted sum of the individual loss components. The weights for the 4 loss function, which are represented by λ_{coord}, λ_{obj}, λ_{class}, and λ_{noobj} respectively, are hyperparameters that must be fine tuned to accurately reflect the error of the model. Since each term has a lambda scalar coefficient, I am including the scalar of 1/n in it as well. Also, every loss component is scaled by a different weight, so the ratio of the lambda weights for each loss component is the real key in determining loss, not the magnitude of the scales. For example, 1,1,1,1 weights are no different in a loss function than 2,2,2,2 weights, as a change magnitude will only change the magnitude of the gradient descent, not the individual shifts. Therefore, we can assume one scalar = 1, and thus don't need to include that scalar. Since objectness and conditional class probabilities are probabilities between 0 and 1 while being part of object loss, λ_{obj} and λ_{class} are going to be very similar, so some object detection algorithm developers decided to remove those two weights in the loss function.

The overall loss function is shown below, where the green is the object loss and the red is the no-object loss.

Total Loss = λ_{coord} · Localization Loss + Objectness Loss + Class Probability Loss + λ_{noobj} · No_Object Loss

$$= \lambda_{coord}\left[\sum_{i=1}^{n} 1_i^{obj}\left[(x_i - \widehat{x}_i)^2 + (y_i - \widehat{y}_i)^2 + (\sqrt{w}_i - \sqrt{\widehat{w}_i})^2 + (\sqrt{h}_i - \sqrt{\widehat{h}_i})^2\right]\right] +$$

$$\left[\sum_{i=1}^{n} 1_i^{obj}\left(\left[P(O) \cdot IOU_{pred}^{truth}\right]_i - P(o)_i\right)^2\right] + \left[\sum_{i=1}^{n} 1_i^{obj}\sum_{j=1}^{C}(P(c_j \mid o)_i - \widehat{P(c_j \mid o)}_i)^2\right] +$$

$$\lambda_{noobj}\left[\sum_{i} 1_i^{noobj}(P(o)_i)^2\right] \quad (9.6)$$

9.4 - YOLO

Even though the Sliding Window Approach was a pivotal development in object detection, it has several problems. Sliding window approach involves a 2 step process: first localizing by finding/breaking down the image into all segments that could contain an object, then running each through a classifier to determine which, if any, object class is contained in the segment. Methods that follow this general 2 step process are known as **2-Stage Object Detection Algorithms**. The main disadvantage of this approach is that each step is separately computed, which can be computationally intensive and time-consuming. Also, the classifier is run multiple times in SWA: once on each segment, slowing down the process. This is not ideal for real life applications, such as robots who are attempting to process real time image data. The robot's cameras are taking an image every couple of milliseconds and the time to make the next decision is very slim. Additionally, in SWA, the same regions of the image are processed multiple times due to overlapping windows, leading to inefficient use of computational resources.

This method typically analyzes small patches of the image independently, resulting in a loss of spatial context and relationships between objects. This shows that for effective and efficient object detection, we must have an algorithm which takes in the entire image. Furthermore, SWA often struggles with detecting objects at varying scales because it relies on fixed-size sliding windows.

Instead of doing the two processes separately and running the classifier multiple times, we want to find a way to do both components simultaneously and run the classifier once to speed up the process. We want to be able to input an image to a model and have it map the entire image pixel data to each object's location (bounding boxes) and class label (along with confidence level). This is known as a **Single Forward Pass**. We can evaluate the outputs by alignment with the ground-truth bounding boxes using the loss function derived in the last section. This means that we can treat object detection as a giant regression problem via one convolutional neural network that finds the optimal mapping function.

In the sliding window approach, we tried to facilitate solving the object detection problem by breaking down the image into all possible segments and analyzing those segments separately instead of the full image. When looking at each segment, the problem is reduced down to detecting objects captured in the segment, rather than the whole image. In this sense, the segment has **"the responsibility"** of detecting/classifying an object captured in the segment. However, the object had to be fully, not partially, captured in the segment and the localization was performed separately via brute force, which is inefficient. We want to find a similar but more efficient way in simplifying the object detection task by localizing the responsibility of detecting objects into smaller, more manageable sections rather than the whole image.

Instead of assigning segments the responsibility to detect an object that is fully captured in that segment, it is more efficient to assign them the responsibility to detect objects that have a central coordinate that lies in that particular segment regardless if the object exceeds the bounds of the segment. This increases the likelihood the segment can be useful and detect an object. Also, since the central coordinate of an object only needs to be captured by one specific area, it eliminates the need to have overlapping segments, making the overall process more efficient. Therefore, we break down the image into a **SxS grid**, where S is a hyperparameter that must be tuned. Because inputted images have variation in width and height, it first resizes the image to a fixed size to provide a consistent scale for the model to train/test on. The resized image is then equally divided into a SxS grid, where the grid cells have the same width/height to maintain consistency. Each created grid cell is assigned the responsibility of detecting objects with a central coordinate that is within that particular grid cell.

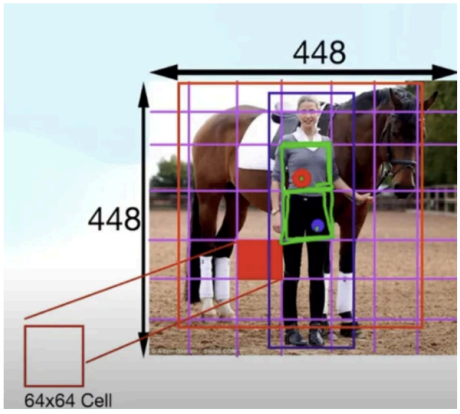


Fig. 9.7: A resized input image divided into an S×S grid (here 7×7 with 64×64 cells), where each grid cell is responsible for detecting objects whose center falls within it—regardless of the object's full extent—enabling efficient and non-overlapping object assignment.

So, after breaking down the image into grid cells, we need to determine how many bounding boxes the model should predict in each grid cell. In object detection, objects with central coordinates in a certain grid cell can vary greatly in size and spatial arrangement. The model has to account for those factors when predicting bounding boxes. Predicting a singular bounding box for each grid cell means that the model only has one opportunity to accurately localize any objects with a center point in that grid cell. However, multiple objects could have their center coordinate in the same cell, so predicting one per cell could lead to underfitting. Therefore, we want to predict multiple bounding boxes per grid cell. However, the model shouldn't predict too many bounding boxes, as that will increase the time complexity and lead to potential overfitting. Therefore, the model predicts a fine-tuned number of bounding boxes per grid cell, signified by hyperparameter **B**.

An example of the grid breakdown is shown on the right. The image contains two objects of interest: a human and a horse. Their object central coordinates are captured by the blue and red dots respectively. The image is broken down into a 7x7 grid, where each grid cell is 64x64 pixels. The grid cell with the red dot is assigned the responsibility of predicting the bounding box of the horse and the grid cell with the blue dot is assigned the responsibility of predicting the bounding box of the human.

Consider the following scenario: the coordinate representing the center pixel of a small image represents a pixel near the top left of a much bigger image, as the top left corner is the (0,0). A large object in a small image might have the same width/height as a small object in a larger scaled image. This inconsistency in scale complicates the learning process for the model and its ability to generalize across images of different sizes. We want to modify the localization variables (x, y, w, and h) so the model has a consistent scale to learn on.

Currently, the localization variables are scaled based on the size of the image. If we make the variables relative to the image scale, the results will be on a consistent scale of 0 to 1, making the model's learning process more stable. Therefore, we normalize width and height by dividing the two metrics by the image's width and height respectively. Since each grid cell is assigned responsibility to detect an object if its center coordinates lie in that cell, it is known the center coordinates (x and y) can only be within that grid cell and not elsewhere. Therefore, to make the scale more localized to that particular grid cell, we can convert x and y to relative coordinates in respect to the assigned grid cell rather than the whole image's, making the upper left corner of the grid cell the origin. This is done by subtracting the upper left coordinates with the object's center absolute coordinates (x and y). It is then normalized by dividing the two relative metrics by the grid cell's width and height respectively. Finally, objectness isn't normalized or modified, as it's always on a consistent scale of 0 and 1 and still applies to this method. During training, each ground-truth bounding box is assigned to the grid cell that contains the center coordinate of the bounding box for comparison.

For conditional class probabilities, it is traditional to have a conditional class probabilities vector for each bounding box. Methods like SWA use that. However, this doesn't account for partially hidden objects, known as **Occlusions**. In the example image of the human and horse, the horse is partially blocked by the human. The model will output high conditional class probability of a horse for the areas captured by the two split sides of the horse and low conditional class probability of a horse for the area blocked by the human. This often leads to an error in prediction where the model would predict two bounding boxes of the same horse at the two split sides of the horse rather than a larger one that fully captures the horse. The model would have a difficult time learning that, as each ground-truth bounding box is only matched to one predicted bounding box. There's also a chance that the model doesn't predict a bounding box on one of the split sides of the horse. Because conditional class probabilities aren't computed in

regions where no bounding box was predicted, the model will have hard time learning to predict a bounding box that includes that missed region as well. Because of that, we need to find a new way to calculate and store conditional class probabilities of regions.

We want the model to look at conditional class probabilities in a broader scope, where the whole image is considered. If the model sees cases where there are high probabilities of the i-th object class separated by a small area of low probabilities of the i-th object class, that highly indicates occlusion. Therefore, we calculate the conditional class probabilities vector for each grid cell, rather than each bounding box, and notice patterns in class probabilities in the scope of the entire image.

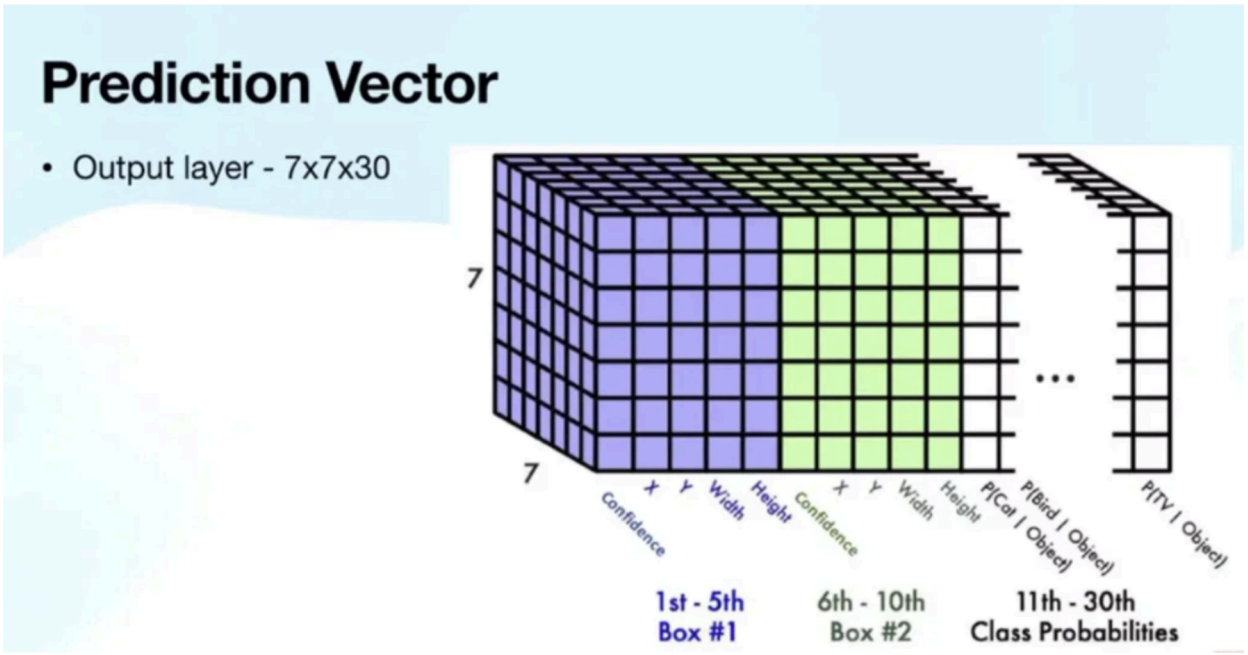


Fig. 9.8: The YOLO model's 7×7×30 output tensor, where each grid cell outputs two bounding boxes (each with coordinates and confidence) and a single 20-dimensional vector of class probabilities conditioned on object presence—enabling occlusion-aware reasoning by comparing class likelihoods across the full image.

Since confidence levels are no longer calculated for each bounding box with position, size and objectness, each bounding box is represented with 5 variables (x, y, w, h, and c). Note that the lowercase c represents confidence level, which is another name for objectness but separate from the class confidence levels, **$P(c_i$ & Object)**. Assuming each grid cell is assigned to predict B bounding boxes and there are C number of object classes. That means we would have $(B \cdot 5) + C$ variables for each grid cell. Assuming the image is initially broken down into a SxS grid, that means that the model's output is stored via a $S \times S \times ((B \cdot 5) + C)$ tensor. Let's assume S = 7, B = 2, and C = 20, the output tensor's dimension and structure is shown below.

The entries at every width (7) and height (7) of the matrix represent the predictions that corresponding grid cells made. The purple and green portions of the matrix is the first and second bounding box each grid cell predicts respectively. Those two portions of the tensor are each 5-layers long, as it's the 5 variables used to represent the two predicted bounding boxes. Finally, the last 20 layers, as shown in white, represent the object class probabilities that the predicted bounding box contains that particular object class.

This tensor shape would also be the same for training images' labels, where it would contain the ground-truth values for the bounding boxes, confidence levels, and probabilities. An example of this method being applied to training data with B = 1 is shown below. For better visualization, we flatten the SxS that represents grid cells of the matrix into one S^2 dimension in the ground-truth value tensor. The image of the horse and human is split into a 7x7 grid (S=7), where A1 to A49 represents each grid cell. The horse and human center is in grid cell A11 and A32 respectively, so the two row's confidence level and probability for the correct object class are 1, as highlighted by red. The bounding box information is filled in A11 and A32, as colored in green. The remaining rows are 0, as those grid cells don't capture the center of objects of interest.

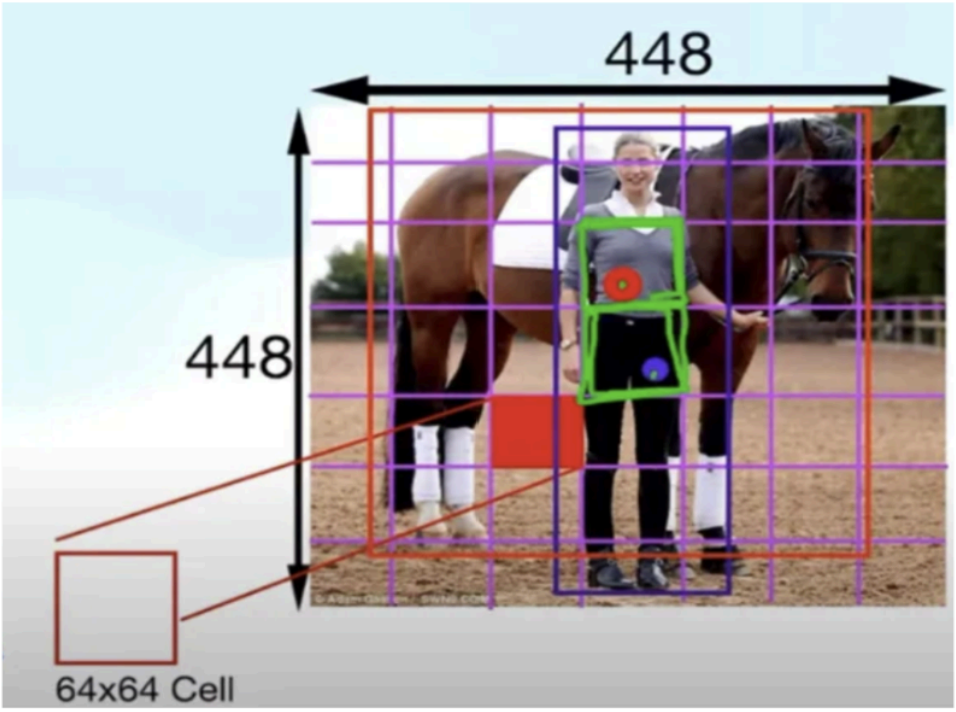


Fig. 9.7: A resized input image divided into an S×S grid (here 7×7 with 64×64 cells), where each grid cell is responsible for detecting objects whose center falls within it—regardless of the object's full extent—enabling efficient and non-overlapping object assignment.

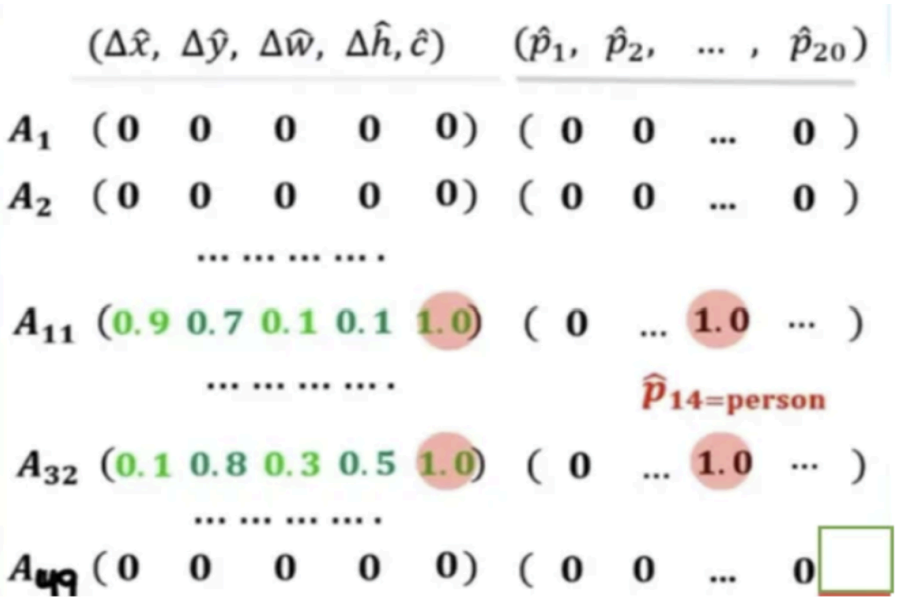


Fig. 9.9: A visualized training label tensor (flattened from 7×7 to 49 rows) where each row corresponds to a grid cell (A_1 to A_{49}); object centers for a horse and a person are located in cells A_{11} and A_{32}, respectively, with bounding box offsets (green) and one-hot class probabilities (red) indicating confident detections.

This model would then use the loss function described in 9.3 to find the most optimal set of weights for each pixel. There are slight modifications in this method's loss function.

- Since B bounding boxes are created in each grid cell, where there are S^2 grid cells, we have to iterate through each bounding box in each grid cell in a double summation in every loss component except for the conditional class probability.
- Because grid cells are assigned the responsibility of detecting objects with central coordinates that lie in that particular cell, the one-to-one matching process between predicted and ground-truth bounding boxes shouldn't match a predicted bounding box assigned to a grid cell with a ground-truth bounding box that is assigned to a separate grid cell. Therefore, the matching is only done within the grid cell.

The overall loss function is described below:

Total Loss = λ_{coord} · Localization Loss + Objectness Loss + Class Probability Loss + λ_{noobj} · No_Object Loss

$$= \lambda_{coord} \sum_{i=1}^{S^2} \sum_{j=1}^{B} 1_{ij}^{obj} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right] +$$

$$\left[\sum_{i=1}^{S^2}\sum_{j=1}^{B} 1_{ij}^{obj}\left(\left[P(O)\cdot IOU_{pred}^{truth}\right]_{ij} - \widehat{P(o)_{ij}}\right)^2\right] + \left[\sum_{i=1}^{S^2} 1_{i}^{obj}\sum_{j=1}^{C}\left(P(c_j|o)_i - \widehat{P(c_j \mid o)}_i\right)^2\right]$$

$$+ \lambda_{noobj}\sum_{i=1}^{S^2}\sum_{j=1}^{B} 1_{ij}^{noobj}\left(P(o)_i\right)^2 \qquad (9.7)$$

Just like other neural network models, it optimizes the overall loss function by altering the weights in accordance with gradient descent, finding the best set of weights to accurately predict bounding boxes, objectness, and conditional class probability.

During inference, the bounding boxes are scaled to a higher size for better resolution. Multiple bounding boxes may overlap significantly. A grid cell may believe an object center falls in that cell but an adjacent cell might also believe that. This would cause both grid cells producing bounding boxes with different central coordinates but attempting to capture the same object's location. Similar to how we dealt with this problem in SWA, we use Non-Maximum Suppression (NMS) based on confidence and IoU. After removing redundant bounding boxes, the neural network would then multiply the conditional class probabilities and objectness to calculate the confidence level for each class and return the bounding boxes along with their respective class labels and confidence scores.

This entire object detection algorithm is the 1st version of **YOLO (You Only Look Once)**, or YOLO V1 (Redmon, 2016). The acronym makes sense considering the inputted image is both localized and classified in a single forward pass. This approach of treating object detection as a regression problem allows YOLO to be faster and more efficient compared to traditional object detection methods, like SWA.

An example application of YOLO is shown below. We are initially given an image of a dog, bike, and car, all of which are objects of interest. We first break down the image into an SxS grid (Left pic), predict bounding box and compute class probabilities in each grid cell (Top and Bottom middle pic respectively), and the predictions are made by filtering out bounding boxes (Right Pic).

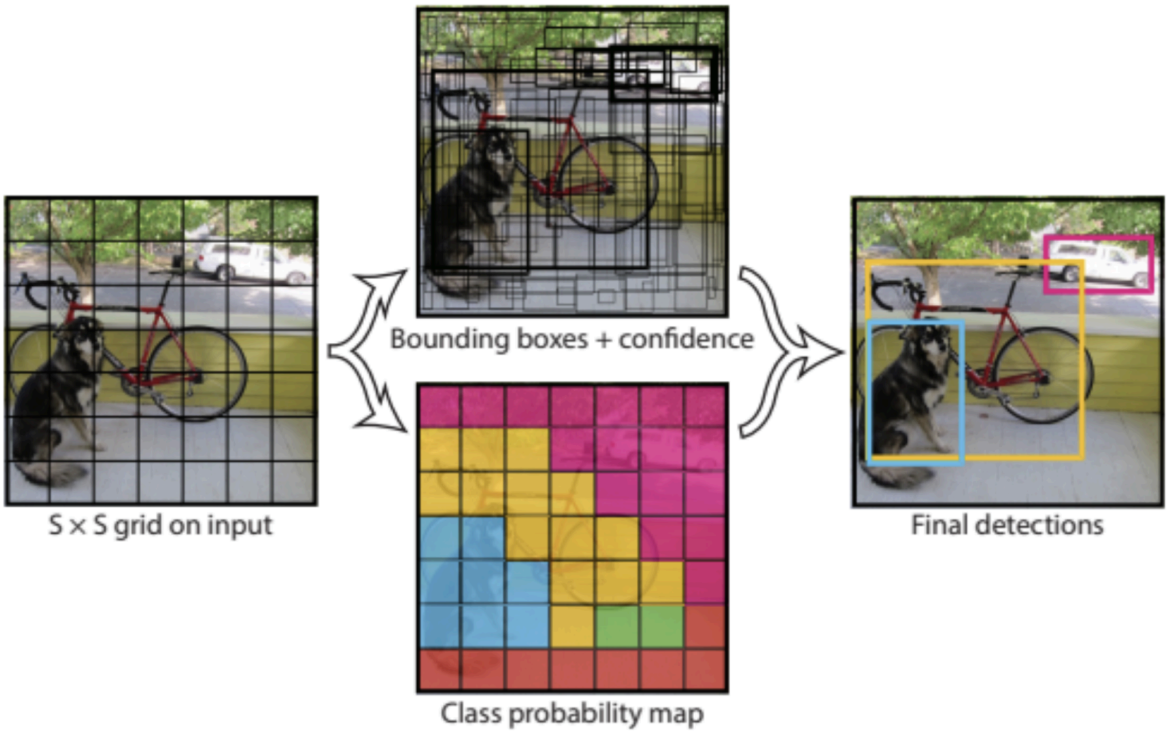


Fig. 9.10: YOLOv1 example pipeline—(Left) input image divided into an S×S grid; (Top middle) bounding box predictions; (Bottom middle) class probability map across grid cells; (Right) final detections after filtering, showing bounding boxes for the dog, bike, and car.

Chapter 10: 3D Vision

10.1 - Introduction to Depth Perception

As shown in prior chapters, some fundamental goals of robotics entail knowing the location of objects and goals, mapping its environment, estimating the precise space an object occupies for manipulation, etc. A single 2D camera is sufficient to identify these objects and gauge some elements of their relative positioning, behind/in front, to the left/to the right. However, it does not give depth information. Without depth, the robot might recognize an object but not know whether it's within its path or far away in the background. Depth data is used to generate costmaps, 3D occupancy grids, or point clouds, which are essential for map-based navigation. A gripper must know exactly how far to reach to grasp an object—2D image coordinates alone are insufficient. This shows the need for robots to know the 3D position, including depth of external objects relative to the robot.

To get semantic data - information about object classes, we must use at least an image from a single camera, known as monocular vision. The mathematical relationships underlying a pinhole camera model tell us the light ray that caused the values of each specific pixel. However, we want the specific point with depth information which this does not provide. Gauging the depth of a point captured by a camera requires additional information about that point.

To gauge the distance of a point, we must interact with it. As shown in the chapter introducing computer vision, interacting with many points at a distance on the same object and different objects points to using reflected light off of these objects. As we already know, for our purposes, light moves in a straight line with a fixed speed.

We can leverage this constant speed to use the time it takes a light ray to travel from a point to a light sensor to gauge the distance. To know when the light is reflected, we must use an active sensor that sends out light and receives reflected light, then divides the total time-of-flight of the light by two to get the time a light ray took to travel from the object to the light sensor. This is the basis of what is known as a **LiDAR** sensor. Since light only reflects off surfaces, each angle of light sent out and received indicates a different point we know about in the world. The aggregation of these points creates a raw 3D (or 2D (fixed height) depending on the LiDAR) representation of the environment known as a **point cloud**. For a given identified object, location, or angle, we can get the position of the part of the object that is closest to the sensor. Note that this assumes the object is unobstructed.

To determine the depth of a class or object in an image, we must incorporate insights from object detection. Using the mathematical relationships defined by the pinhole camera model, we can infer the angle of incoming light that produced the intensity at each pixel. Once an object is detected, we can identify the corresponding 3D rays for the pixels representing it and locate the LiDAR point closest to each ray. This works by calculating the direction of each pixel's ray in the camera frame and searching for the nearest LiDAR point that lies along or near

that direction in 3D space. Alternatively, we can reverse the process: starting with a 3D point measured by a LiDAR sensor, we can compute its projection onto the camera's image plane. This involves applying the camera's intrinsic and extrinsic parameters to transform the 3D point into a 2D pixel coordinate in the image.

However, low-reflectivity surfaces (e.g., black cars, asphalt, matte plastic) reflect much less light back to the sensor. This leads to dark or absorbent objects that may not be detected at all. Additionally, LiDAR sensors are active, meaning they emit their own energy for proper sensing. This requires more energy and sophisticated hardware (e.g. laser emitters, high-speed photodetectors, precise timing electronics, rotation and scanning mechanisms) that make 3D LiDAR sensors very expensive. 3D LiDAR sensors also must capture a lot of individual points which make them very computationally intensive and require a lot of storage. This presents an issue as we often must get the depth of objects that are at varying heights or angles relative to the robot. For example, an autonomous delivery robot may need to detect a sidewalk curb (low obstacle) and a street sign (high obstacle). Additionally, a home service robot has to perceive a coffee table leg (low and thin) and a chandelier (high and suspended), both of which can pose collision risks but are hard to detect with basic LiDAR setups. A 2D LiDAR only sees one horizontal slice—so it cannot see both. Additionally, the laser pulse weakens as it travels, and only a small fraction of it is reflected back to the sensor. The farther the object, the weaker the return signal becomes, often below the sensor's detection threshold. This causes objects beyond the maximum range (e.g., 100m for some high-end LiDARs, 10–30m for consumer-grade) will not be detected reliably. Additionally, at long ranges, angular resolution limits become significant. Two points close together in space may fall within the same angular "beam" and be indistinguishable. This results in sparse, less detailed point clouds at distance. As a result, there is poor feature detection and obstacle classification at range.

These issues show the desire for a light-sensor that can leverage dark or absorbent materials and information. They also show the desire for this sensor to be passive, naturally detecting light from a wide 3D range of angles, avoiding issues related to the parts that go into emission and detection as well as the light attenuation/weakening problem associated with emitting your own light. This points to using cameras, from which we can deduce the light rays that show up at specific pixels using specifics of the design of the pinhole camera.

A single camera can use this matrix to figure out the line coming out of the camera that the point its sensing resides on. However, we want the specific point with depth information which this does not provide. Each point in the environment reflects light in a line in all unobstructed directions. Each of these lines intersect at the point, and only at that point. This means that knowledge of two rays that stem from the same point can be used to deduce the location of the point. Since a single camera only gets one angle of light from a point, this requires a second camera. We wish to maximize the amount of overlap between cameras and

minimize the amount of light that is obstructed for one camera and not for the other. For example, there may be a direct line from a point to one camera, but the line to the other camera is obstructed, removing the ability for the 3D pose to be deduced. Since closer cameras would lead to similar angles, there would be more overlap in sensed points. For this reason, we want to have two nearby cameras which simultaneously capture images, so we can figure out where relevant points are in each image, use the cameras parameters to figure out the line connecting each camera to the point, and then find the intersection of the lines to find the 3D position of the point. This paradigm is known as **stereo vision**. As a fun fact, this is how humans perceive depth with two eyes. Images of stereo vision are shown below:

Fig. 10.1: A physical stereo vision system consisting of two synchronized cameras mounted side-by-side on a fixed baseline. [e-con Systems, n.d.]

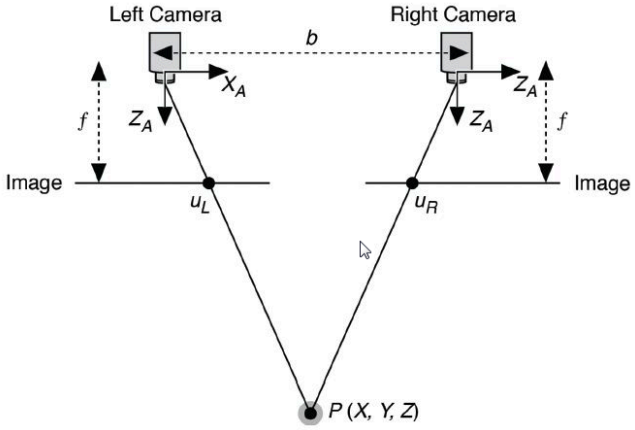


Fig. 10.2: Geometric model of stereo vision. Two calibrated cameras with baseline b observe a 3D point P(X,Y,Z), which projects to pixel coordinates u_L and u_R in the left and right images, respectively. [Tech Briefs, 2012]

It is important to note that fusing a single camera and single LiDAR is generally less computationally intensive and LiDAR is not as affected by adverse lighting conditions and

weather. Also, stereo vision relies on matching points in different images which is difficult in low texture environments such as blank walls.

10.2 - Camera Calibration

An image can be viewed as a discrete 2D function of n×m pixels, f(x,y), where n is the number of rows (height) and m is the number of columns (width) in the image. The function f(x,y) returns the RGB value of the pixel at the coordinates (x,y). Additionally, images can be represented as 3D arrays, where the third dimension corresponds to the color channels (Red, Green, and Blue), with each channel having a value at each pixel location.

Each x, y corresponds to a specific angle of light. In order to understand the spatial relationships of an image, including what areas they capture, the relative positions of objects/points, etc. we must know the ray corresponding to each pixel. We also must be able to gauge the line for a given pixel to match it with a LiDAR point or pixel in another camera.

The pixel that a light ray strikes is determined by the position of the point relative to the camera's relative reference frame. However, we must be able to interpret light rays from different cameras or a LiDAR and a camera with respect to the same reference frame so we can figure out where they intersect/project. Note that this simply entails transforming points from the world frame into the camera frame by multiplying the world point with the 4x4 homogeneous transformation matrix ($T_{w \to c}$). This matrix is known as the **camera extrinsic matrix** and is the inverse of the camera pose in the world frame. This matrix, when applied to the camera's pose, will output the identity rotation and zero translation, meaning the camera's origin and axes align with the world frame. Applying any other poses will result in the rotation and translation components that describe the orientation and position of the world as seen from the camera. Note that we usually store the camera's pose relative to the world frame so we can easily construct T_{cw}. Note that since $R_{w \to c}$ and $R_{c \to w}$ are orthogonal matrices, their inverses (each other) are also their transposes. This brings us to the following equation where X_w is the point pose relative to the world and X_c is relative to the camera:

$$T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \tag{10.1}$$

$$T^{-1} = \begin{bmatrix} R^T & -R^T t \\ 0 & 1 \end{bmatrix} \tag{10.2}$$

$$X_c = (T_{c-->w})^{-1} \cdot X_w \tag{10.3}$$

To transform a point from the camera frame to the world frame, apply the homogeneous transformation matrix representing the camera's pose in the world frame—that is, its rotation and translation relative to the world coordinate system. This is the inverse of the camera extrinsic matrix. In the pinhole camera model, a ray is assumed to originate at the camera's pinhole and extend outward through a pixel. Therefore, the ray's origin in the world frame is simply the camera's position in the world. If a ray's direction is expressed in the camera frame, it must be rotated into the world frame for comparison with other rays. This is mathematically described as below:

$$d_{world} = R_{c->w} \cdot d_{cam} \tag{10.4}$$

$$origin_{world} = - R^T \cdot t \tag{10.5}$$

In cameras, the x axis is the horizontal axis going left/right through the center of the image plane, the y axis is the vertical axis going up/down through the center of the image plane, and the z axis going out/in the center of the image plane and pinhole. This is shown as below:

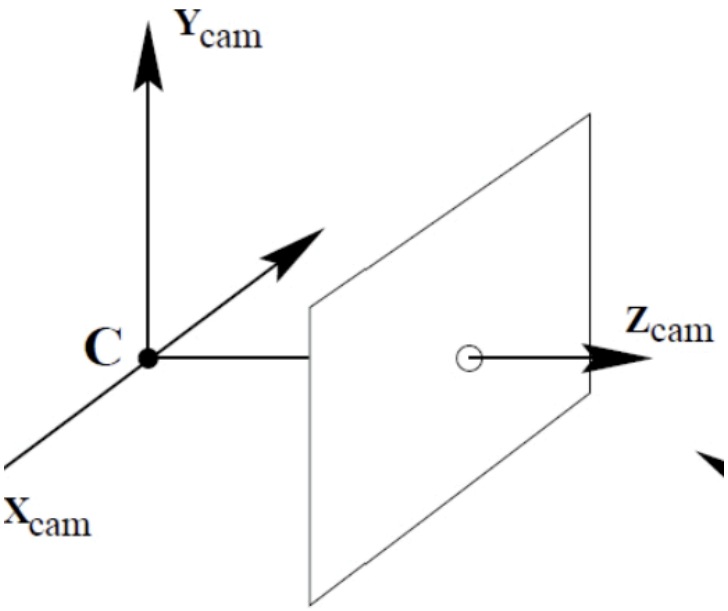


Fig. 8.3: Coordinate frame of a pinhole camera model. The origin C represents the optical center of the camera, with the Z_{cam}-axis pointing forward along the optical axis, X_{cam}-axis pointing to the right, and Y_{cam}-axis pointing downward or upward depending on the convention. The image plane is shown in front of the camera for visualization clarity, though it is typically located behind the optical center in a physical model. [Aldea, University of Pavia (n.d.)]

The origin (0,0) is located at the top-left corner. The bottom-right corner is at pixel (w−1, h−1), where w is the width in pixels of the image along the x axis, and h is the height of the image in pixels along the y axis. The pinhole projected straight back to the image plane is where a dead-center light beam would hit and is denoted as (c_x, c_y) and called the **principal point** or optical center. In an ideal case, this is in the direct center of the image, meaning half of the image is to the left/right, and half is above/below. This is mathematically described as below:

$$c_x = \frac{w}{2} \tag{10.6}$$

$$c_y = \frac{h}{2} \tag{10.7}$$

For a given light ray, the deviation along the x and y axes are constant. Each pixel has a specific x and y deviation from the pinhole. Since we know how far back the image plane is from the pinhole, we know how far the light traveled between the pinhole and the point in the image plane corresponding to (x, y). This distance is known as the **focal length** of the image and is denoted as f. An image of this is shown below:

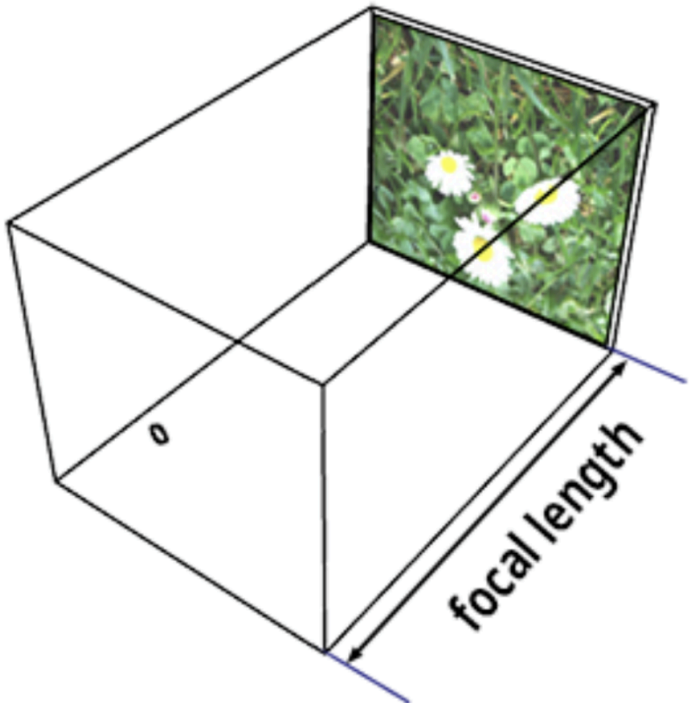


Fig. 10.3: Visualization of focal length in the pinhole camera model. [Scratchapixel, n.d.]

This brings us to the following equations, where X_c, Y_c, and Z_c are the coordinates of some point in the cameras coordinate system (relative to its axes and principal point), and x_d and y_d are the distance deviations from the principal point in the image plane:

$$\frac{x_d}{f} = \frac{X_c}{Z_c} \tag{10.8}$$

$$\frac{y_d}{f} * \frac{Y_c}{Z_c} \tag{10.9}$$

However, we do not get x and y in distance from the focal point, we get them in pixels. This means we must transform our equations to take in x and y pixel coordinates as inputs. This means we must consider the size of each pixel in the image plane known as p, where p_x is pixel width in mm, and p_y is the pixel height in mm. If W is the width of the image plane in mm, H is the heights of the image plane mm, w is the number of pixels in a single row of the image, and h is the number of pixels in a single column of the image:

$$p_x = \frac{W}{w} \tag{10.10}$$

$$p_y = \frac{H}{h} \tag{10.11}$$

Assuming x, y are pixel coordinates, subtracting the principal point gets the number of pixels from the center. Multiplying by the size per pixel gets the total distance. This is mathematically described as below:

$$x_d = p_x(x - c_x) \tag{10.12}$$

$$y_d = p_y(y - c_y) \tag{10.13}$$

This means our overall equations are as follows:

$$p_x(x - c_x) = f * \frac{X_c}{Z_c} \tag{10.14}$$

$$x = (\frac{f}{p_x} * \frac{X_c}{Z_c}) + c_x \tag{10.15}$$

f/p_x is essentially the number of pixel widths that fit the focal length and is denoted as f_x in equations. This refines our overall equations as follows:

$$x = f_x * \frac{X_c}{Z_c} + c_x \tag{10.16}$$

$$y = f_y * \frac{Y_c}{Z_c} + c_y \tag{10.17}$$

This can be equivalently written in matrix form as:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = K \cdot \begin{bmatrix} X_c/Z_c \\ Y_c/Z_c \\ 1 \end{bmatrix} \tag{10.18}$$

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{10.19}$$

K, the matrix that projects camera coordinate onto the image plane (pixels), is known as the **camera intrinsic matrix**. In cases where we must account for any non-perpendicularity between the x- and y-axes of the camera, there is a value s between f_x and c_x called the **skew**. Usually, for most modern cameras, s ≈ 0 (i.e., the axes are approximately orthogonal). Note that to get the 3x1 vector that K multiplies, we would simply multiply X_w by T_{wc}, then divide each entry by Z_c.

K maps between a ray and a pixel. The 3x1 vector that K multiplies gives us the direction vector for the line. The 1 at the bottom can be viewed as Z_c/Z_c. The proportion between the first and second indices of the vector give the slope of the line, meaning Y_c/X_c.

We can take the inverse of K to get a matrix mapping a pixel to the corresponding ray direction. This is mathematically described below for pixel (u, v):

$$\mathbf{d}_{cam} = \left[\frac{u - c_x}{f_x}, \frac{v - c_y}{f_y}, 1 \right] \tag{10.20}$$

The ratio of the first and second indices give the ratio between the pixels offset from the principal point in the x and y directions which give us the ratio between the change in x and y per unit of z. The line extends into the scene as a 3D ray - an infinite number of 3D points P, where C is the camera center, such that:

$$P = \lambda d_{cam}, \lambda > 0 \tag{10.21}$$

The ray's direction, initially expressed in the camera frame, must be rotated into the world frame for comparison with other rays using the camera extrinsic matrix. The final equation for the ray given the origin and direction in the world frame is mathematically described as below:

$$Ray\,(\lambda) = origin_{world} + \lambda \cdot d_{world} \tag{10.22}$$

Where λ is a dummy variable scaling d_{world} that represents the distance traveled along d_{world} from $origin_{world}$. The specific value of λ is unknown, figuring it out gives us the position of the point in the environment that a pixel captures relative to the world frame.

10.3 - Stereo Vision

10.3.1 - Horizontal Stereo

For a given pixel in one image, we must find the corresponding pixel, meaning the one capturing the light from the same point, in another image in order to find the position of the external point. This process is called **stereo matching** or correspondence search. After this, we must compute the 3D point from the rays cast by corresponding image points and camera parameters. This process is called **triangulation**. In stereo matching, each pixel that gives depth information must be in both images. We arbitrarily select the pixels of the left image and search for the corresponding pixel in the right image.

When seeing if rays connect, we have to algebraically solve for both the x, y, and z position of the connection. Solving for this in 3 dimensions can be computationally intensive. Also, due to improper matching and errors introduced by discretizing the image plane into pixels, we cannot guarantee that the rays emanating from the left and right will connect, meaning they are not coplanar. As a result, determining the best estimate of the intersection point becomes an optimization problem: finding the point that minimizes its distance to both rays. This usually requires singular value decomposition or some least squares calculation.

To simplify depth estimation, we want to ensure that for any given depth Z, the position of the 3D points corresponding to each camera's ray differ in only one position axis - either x or y. Technically it could be along a 1D line that doesn't align with the x or y axis, but this introduces extra mathematical and computational steps. This ensures that one positional component of the rays is identical and that the other component contains all the information needed for depth via disparity. This setup reduces the reconstruction problem to a 1D algebraic calculation along the varying axis.

This leads to the goal that, at each absolute depth in the environment, the two rays from the two cameras should share either the same x-coordinate or the same y-coordinate. Here, P_z refers to the actual depth of a specific 3D point (P_x, P_y, P_z) in the environment, while Z represents an arbitrary depth at which we want to evaluate the (x, y) coordinates the ray passes through. For a given depth Z in the environment, the x and y coordinates of the point at that depth along the ray connecting an environmental point (P_x, P_y, P_z) to a camera located at (c_x, c_y, c_z) are given by:

$$X(Z) = c_x + \frac{P_x - c_x}{P_z - c_z}(Z - c_z) \qquad (10.23)$$

$$Y(Z) = c_y + \frac{P_y - c_y}{P_z - c_z}(Z - c_z) \tag{10.24}$$

Essentially, it is the origin plus the slope of the ray with respect to changes in z, times the change in z from the origin. Since we want one of these to be equal for the two cameras, we get the following:

$$Y_1(Z) = Y_2(Z),\ for\ all\ P,\ Z \tag{10.25}$$

$$c_{y1} + \frac{P_y - c_{y1}}{P_z - c_{z1}}(Z - c_{z1}) = c_{y2} + \frac{P_y - c_{y2}}{P_z - c_{z2}}(Z - c_{z2}),\ for\ all\ P,\ z \tag{10.26}$$

In other words, the rays must be the same/coincide/collinear in the Y-Z plane. For the rays to be the same/coincide/collinear in the Y-Z plane, the lines have the same slope for all P, z. They also must share all points, with no points that one ray goes through that the other one doesn't.

Note that for this to work, the camera's must have the same Y-Z directions or else changes in Y and Z have different effects for different cameras. This allows us to not effectively use this framework to come to an agreed upon y-value or z-value of the environmental point. This means the optical and y axes of the cameras must be the same. Note that this leaves the possibility for opposite x-axes, but for simplicity we choose a common x axis direction. This means the cameras must have the same reference frame axis directions.

Additionally, for the two rays to have the same direction, slope, and overlap at all points for every possible P and z, both cameras must have the same c_y and c_z. Technically, for a specific point P, two cameras with different c_y and c_z could still share a common line to that point if the line happens to pass through both camera centers. However, to ensure this alignment for all possible P and z, the only solution is for both cameras to have identical c_y and c_z. Any vertical or depth offset would cause some rays to land at different y-coordinates. Therefore, the two cameras must have parallel optical axes, identical c_z values, and either matching c_x or c_y, depending on the orientation.

It is important to note that there is another motivation for this alignment. When searching for the corresponding pixel in the right image, we must conduct a 2D image search. To make this more efficient, we wish to reduce this to a 1D search. For mathematical ease, we would prefer these to be in the same row or column, removing the need for 2D math and additional pixel calculations. If all matches lie along the same row or column, we simply have to do a 1D search where we increment 1 pixel dimension by 1 for each pixel search.

Given two cameras with the same focal length, the pixel row and column of a 3D point in each image is given by (u, v) = (f * Y/Z, f * X/Z). To get the points in either the same row or column, either the Y & Z distances of the point must be the same for both cameras, or the X and Z distances must be the same for both cameras. To ensure Z is consistently the same for all points, the Z-axes of the cameras must be the same, and the cameras must have the same Z-position. This is true for all axes for similar reasons to the last sentence and as described in above paragraphs. This means their baseline is a pure translation along the x and y axes, and that the image planes are coplanar. To ensure the Y distance is the same the cameras must occupy the same Y position, meaning they are horizontally aligned, and vice-versa for the same X distance.

Since it is usually important to capture more width of a scene than height and also horizontally aligning cameras is less likely to increase the space the robot takes up, we usually opt for horizontal alignment. This means that the left camera is considered as the origin of the stereo system (0,0,0), and the right camera is translated along the x-axis by the baseline of distance B to be at point (B, 0, 0). For a given point, $Y_{Right} = Y_{Left}$, $Z_{Right} = Z_{Left}$, and $X_{Right} = X_{Left} - B$. Note that even though a point will approach the camera it is closer to with a steeper vertical angle, the vertical displacement per change in Z is the same meaning that the row it hits in the image plane is the same. An example of this setup is shown below:

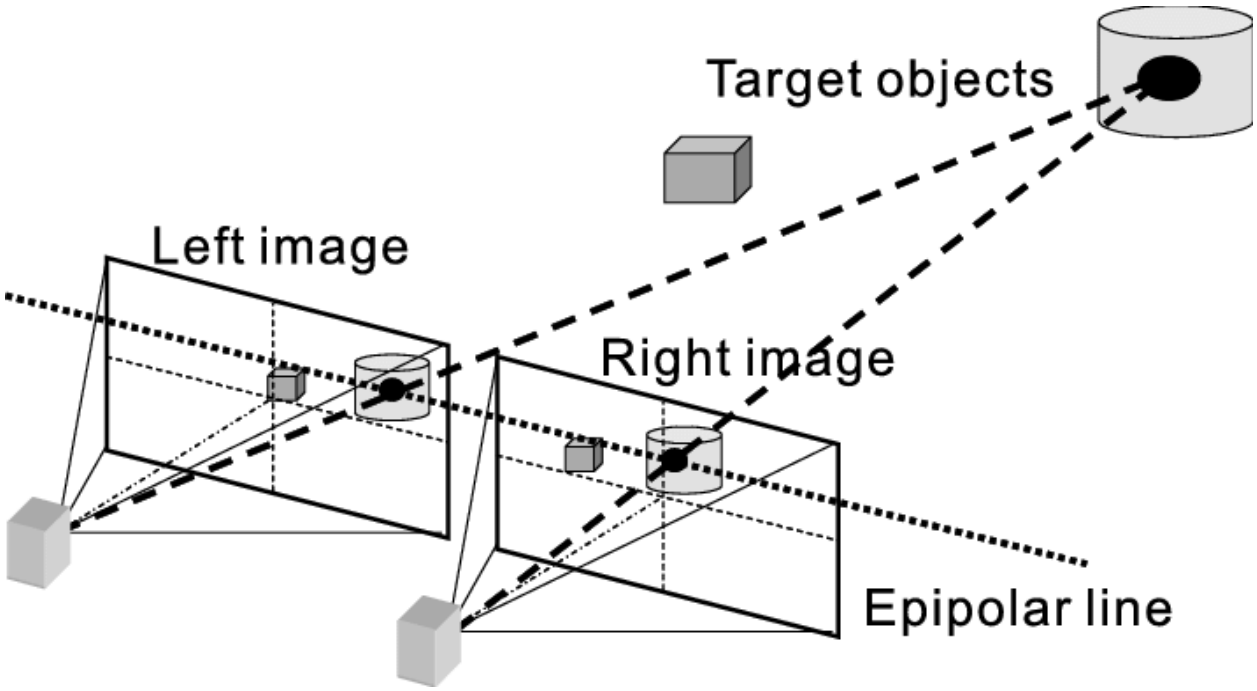


Fig. 10.4: Stereo vision setup with horizontally aligned cameras. The left camera serves as the origin of the stereo coordinate frame, and the right camera is translated along the x-axis by a fixed baseline distance B. As shown, a 3D point projects to both image planes with corresponding points lying along the same horizontal epipolar line. This geometry ensures that the vertical coordinates (Y) of the projections are equal, simplifying correspondence matching in stereo depth estimation. [Hariyama et al, 2008]

Treating X_{Left} as X, meaning X_{right} is X - B, the horizontal pixel in the left and right image respectively is given by:

$$u_l = f \cdot \frac{X}{Z} \tag{10.27}$$

$$u_r = f \cdot \frac{X-B}{Z} = f \cdot \frac{X}{Z} - f \cdot \frac{B}{Z} = u_l - f \cdot \frac{B}{Z} \tag{10.28}$$

Rearranging for Z gets us:

$$Z = f \cdot \frac{B}{u_l - u_r} \tag{10.29}$$

u_l - u_r is the horizontal distance between the pixels and is known as **disparity** (signified by d). This refines our final triangulation equation as follows:

$$Z = f \cdot \frac{B}{d} \tag{10.30}$$

For a given system, f and B are fixed beforehand. We simply calculate d for each relevant pixel on the left and use this equation to generate a depth component to each pixel.

For a given 3D point, the point appears more rightward in space relative to the left camera's pinhole than it does relative to the right camera's. As a result, the ray from the point to the left camera makes a steeper leftward angle than the ray to the right camera. In terms of projection, this means the point moves more leftward per unit Z-depth in the right image and less leftward (i.e., more rightward) in the left image. Thus, the point projects to a more rightward pixel in the left image and a more leftward pixel in the right image, producing a positive disparity $d > 0$. From a correspondence search perspective, if you identify the pixel location of a point in the left image, its corresponding match in the right image must lie somewhere to the left (or at the same location) in the same row.

This overall setup is known as **horizontal stereo**. This is a hardware solution to the broader problem of aligning two images to create horizontal search lines and vertical consistency of rays in the environment. When this hardware solution is not the case, we must solve this process by mathematical/software preprocessing, an endeavor known as **horizontal rectification**.

10.3.2 - Block Matching

As discussed above, when working with horizontally rectified images and a given pixel in the left image, we only need to search leftward along the same row in the right image to find the corresponding pixel. Once identified, we can apply the equation from the previous section to compute the 3D position associated with that pixel. This requires figuring out the corresponding pixel in the right image amongst a series of candidate pixels, a process known as stereo matching or correspondence matching.

This is fundamentally an optimization problem, where each candidate pixel is assigned a score that reflects how suitable or unsuitable it is as the corresponding match. To select the best match, we define a mathematical cost function that quantifies this 'badness' and then choose the pixel that minimizes this cost. We then choose the disparity for each pixel to minimize the badness. We can often calculate this manually using prior knowledge of indicators of pixel match.

A 3D point projects to corresponding pixels in both images, so true matches should exhibit similar brightness and color. This gives rise to a cost metric capturing the difference in intensity and color between the known pixel in the left image and the candidate pixel in the right image. We usually incorporate some block region around the pixel (e.g. 3x3, 9x9) to reduce the effects of sensor noise and incorporate any relevant surrounding information. To calculate the disparity, we can use the simple, yet effective, disparity equation of Sum of Absolute Differences (SAD). We could also use something like Sum of Squared Differences, or another cost function but the results are largely the same. This brings us to the equation for a window W:

$$\mathrm{SAD}_{\text{color}} = \sum_{(i,j)\in W} |R_\ell - R_r| + |G_\ell - G_r| + |B_\ell - B_r| \tag{10.29}$$

It is important to note that brightness is more information rich as it captures edges, textures, and gradients. Additionally, lighting changes (shadows, highlights) affect color channels in complex ways, but the overall brightness pattern remains more stable under small viewpoint or exposure shifts. Matching on brightness is thus more robust to those variations. Also, keeping track of color is 3x more computationally intensive than brightness. So in situations where computational intensity is very important and color will not likely matter very much, we should use intensities instead as given below:

$$\mathrm{SAD}(u, v; d) = \sum_{(i,j)\in W} |I_\ell(u+i,\, v+j) - I_r(u+i,\, v+j-d)| \tag{10.30}$$

The metric of cost based on local color or brightness information is known as **photometric cost**. Disparity calculation based on minimal photometric cost based on a local block of pixels is known as **Block Matching (BM)** (Lucas & Kanade, 1981).

10.3.3 - Semi-Global Matching

However, block matching presents several challenges. Even with perfect photometric consistency, many pixels—particularly in flat or repetitive regions—may have multiple disparity values that yield similar costs. In textureless areas (such as blank walls), the photometric cost function often yields a nearly flat curve across a wide range of disparities, making it difficult to identify a clear minimum. Additionally, random sensor noise or isolated mismatches can introduce further errors.

Fortunately, there is an additional cue for estimating pixel disparity. Most surfaces in the environment are locally planar or smoothly curved, which means that neighboring pixels typically have similar depths—and therefore similar disparities. This warrants an additional cost for the smoothness of disparities of neighboring pixels, on top of photometric cost. In flat or repetitive regions, this smoothness cost helps bias the solution toward coherent, continuous surfaces. It also helps suppress the effects of sensor noise and photometric outliers by encouraging disparities to align with those of neighboring pixels, effectively averaging out anomalies.

This means that we should enforce that neighboring pixels ought to have smooth disparities. Since disparities represent pixel shifts, disparity differences are always whole numbers. To enforce smooth disparities, we should have no cost or punishment if two neighbors, p & q, choose the same disparity. To punish small disparity differences while allowing for gentle slopes and slanted surfaces, we give a small penalty for a disparity difference of 1. We want to heavily punish large jumps, without over-penalizing depth discontinuities (e.g. object edges). For this reason, we use a larger constant P_2 to penalize all disparities > 1. However, using a pure linear penalty $|d_p - d_q|$ would heavily punish true edges. Instead, truncation ensures that the large disparity differences caused by edges are not over-punished. In practice, these are tuned on validation data to match typical scene geometry and noise levels. Note that some sophisticated methods explicitly employ edge detection to allow for edges to be treated differently than large disparity differences caused by poor matching. This is outside of the scope of this textbook. This overall smoothness cost of neighboring pixels discussed above is known as **Truncated-Linear Penalty (TLP)** and is mathematically described by V as below:

$$V(d_p, d_q) = \begin{cases} 0, & d_p = d_q, \\ P_1, & |d_p - d_q| = 1, \\ P_2, & |d_p - d_q| > 1, \end{cases} \tag{10.31}$$

It is important to note that the smoothness depends on the disparity values of multiple pixels. Since the smoothness cost of some pixel-disparity pair (p, d) depends on the disparities of

each other pixel in the image, we must consider these pixels when choosing the disparity for p. The best disparity for p is the one that minimizes the overall cost function. This means that when evaluating (p, d), we only care about the optimal set of pixel disparities given (p, d). We don't care about sub-optimal sets as these would never be chosen and should not be compared to other options. This means that the cost of (p, d) is the cost of the optimal disparity set given (p, d). Incorporating smoothness into a cost function means that our function is global since it is considering disparities of all pixels in the image simultaneously.

The overall smoothness cost for a set of pixels and their corresponding disparities depends on the consistency of disparities across neighboring pixels. Evaluating the smoothness of any individual pixel requires considering its neighbors, each of which in turn depends on their own neighbors, and so on. As a result, the total smoothness cost must account for how well disparities agree across the entire neighborhood structure. Note that this can be the 4 cardinal neighbors (←, →, ↑, ↓), or the 8 neighbors including the diagonals if one wants additional computational intensity for a better result. Since every neighbor pair contributes equally and incrementally to the total cost, the total cost should be the sum of smoothness costs of each pixel and its neighbor. This is described as below where E is the set of pairs of neighboring pixels:

$$\sum_{(p,q)\in\mathcal{E}} V(D_p, D_q) \tag{10.32}$$

Since this and the pixel intensity/color disparity both contribute to the total cost, we add them. Note that if we wanted to scale the smoothness cost up or down, that would be reflected in the choice of penalty constants. This brings us to the full global matching cost for a set of disparities:

$$E(\{D_p\}) = \sum_{p} C_p(D_p) + \sum_{(p,q)\in\mathcal{E}} V(D_p, D_q) \tag{10.33}$$

Since the photometric cost of disparities are independent of other pixels, we can calculate these ahead of time before the smoothness costs.

In theory, one could compute the smoothness for every possible disparity assignment, but that incurs a combinatorial explosion—far too costly for any practical image size. Even "exact" optimization methods that avoid exhaustive search must still model all pixel‑to‑pixel interactions (e.g., via large graph cuts or belief‑propagation messages), which becomes prohibitively expensive. Iterative solvers (variational or message‑passing) further compound the cost by requiring many full‑image passes to converge. These realities motivate the use of faster, heuristic approximations that strike a balance between computational efficiency and depth accuracy.

The cost for a pixel–disparity pair (p,d) is defined as the total smoothness plus photometric cost of the best overall disparity assignment in which p is fixed to d. Since we only need that one best assignment, we can compute this cost by assuming each neighbor q picks the disparity d′ that minimizes its own combined photometric and smoothness cost given p's choice of d.

However, forcing pixel q to adopt disparity d' instead of its own optimal value introduces additional smoothness and photometric penalties across the image. By not selecting its best-matching disparity, q incurs a higher data cost C(q, d'), as the local image patch around q no longer aligns as well in the stereo pair, increasing the raw intensity difference. Furthermore, since q's disparity has changed, the disparity differences between q and each of its neighboring pixels may also change. This in turn modifies the additional smoothness cost incurred due to the disparity differences between q and each of its neighbors. These effects are not confined to q alone. Since q's neighbors now experience altered disparity difference costs with q by changing the disparity of q, their own optimal disparities may shift. This leads to further increases in both photometric and smoothness costs in their respective neighborhoods—and so on. In this way, constraining q to a non-optimal disparity d' can trigger a cascade of additional costs throughout the image, beyond the original smoothness penalty V(d, d') applied between p and q.

When computing the smoothness cost for a pixel–disparity pair (p,d), we must consider more than just the disparity difference between p and each of its neighboring pixels q. Specifically, we must also account for the total cost associated with assigning each neighbor, q, its own optimal disparity d′, as determined in the context of minimizing the overall smoothness cost for (p,d). The total cost of assigning disparity d′ to neighbor q already includes both the photometric cost C(q,d′) and the smoothness cost from q's relationships with its own neighbors. That smoothness cost itself incorporates any penalties from disparity differences with neighboring pixels, along with further downstream costs incurred by propagating changes to minimize the total cost at (q,d′). This creates a recursive structure: the downstream effects of assigning p to disparity d are encapsulated within the optimal disparity selections and associated costs of its neighbors. If the optimal disparity for a neighbor q, given p is set to d, is d′, then all the costs incurred by setting q to d′ are already included in q's total cost. Therefore, to calculate the total cost at (p,d), we sum: The photometric cost C(p,d), the total cost of each neighbor q assigned to its optimal disparity d′, and the direct disparity difference penalty V(d,d′) between p and each neighbor q.

The optimal cost for a pixel-disparity pair, considering smoothness cost along all directions and photometric cost, is given by S(p, d). We assume that the disparity d′ at a neighboring pixel q is the one that minimizes S(p,d), meaning it is selected to minimize the total cost at (p,d). Since the disparity at q only affects the smoothness cost of (p, d) and not its photometric cost, we refine the definition of q's optimal disparity to be the value that minimizes

the smoothness cost contribution to (p, d). The smoothness cost contributed to (q, d') by (p, d) is the sum of the disparity transition penalty V(d, d′) and the additional cost at (q, d′) beyond the cost of q's optimal disparity—i.e., the disparity d′that minimizes S(q, d′).

- If d = d′, then V(d, d′) = 0, and the smoothness cost contribution from q is simply S(q,d′) - $min_{d'}S(q, d')$
- If $|d - d'| = 1$, then a small penalty P_1 is applied, so the total contribution is S(q, d′) - $min_{d'}S(q, d') + P_1$
- If $|d - d'| > 1$, then a larger penalty P_2 is applied, giving S(q, d′) - $min_{d'}S(q, d')+ P_2$

We want the d' in the disparity range that minimizes this smoothness cost of (p, d). For a given pixel p, disparity d, neighbor q, the cost of d considering the disparity with q and penalty accrued by q is mathematically described below:

$$min(S(q,\ d),\ S(q,\ d\ +\ 1)\ +\ P_1,\ S(q,\ d\ -\ 1)\ +\ P_1, min_{d'}S(q,\ d')\ +\ P_2)\ -\ min_{d'}S(q, d') \quad (10.34)$$

The total smoothness cost incurred by assigning disparity d to pixel p is computed by summing the smoothness costs across all directions, each denoted by r. The smoothness cost in a particular direction r—that is, due to the interaction with a single neighboring pixel—is denoted $L_r(p,d)$. We refer to the neighboring pixel in direction r. With this notation, we can express $L_r(p,d)$, the total cost smoothness cost at (p, d) (we will call this L(p,d)), and S(p, d) as follows:

$$L_r(p, d)\ =\ min(S(q_r, d),\ S(q_r, d\ +\ 1)\ +\ P_1,\ S(q, d\ -\ 1)\ +\ P_1, min_{d'}S(q, d')\ +\ P_2)\ -\ min_{d'}S(q, d') \quad (10.35)$$

$$L(p, d)\ =\ \sum_r L_r(p, d) \quad (10.36)$$

$$S(p, d)\ =\ C(p, d)\ +\ L(p, d)\ =\ C(p, d)\ +\ \sum_r L_r(p, d) \quad (10.37)$$

The optimal disparity must minimize the total cost from each direction as described below where we chose the disparity for p to be the one that minimizes S:

$$D(p)\ =\ argmin_d S(p, d) \quad (10.38)$$

Overall, we would simply calculate L for each pixel, neighbor, and disparity. Then we would calculate S for each pixel and disparity. Lastly, we would calculate D for each pixel. After this, we have a full disparity image which we can use to find depth. Note that we must tune P_1 and P_2 to determine the relative weighing of smoothness cost vs. photometric cost, as well as the relative weighing of disparity differences of 1 vs. disparity differences more than 1.

However, this algorithm assumes that the true cost of the neighboring pixel is already known for each disparity. This assumption does not hold when optimizing disparities across the entire image simultaneously. Moreover, the optimal disparity of each neighboring pixel depends

on the L values of its neighbors, including the current pixel, meaning none can be computed independently. To approximate this, we would have to assume a fixed L value for each pixel. Yet, because every pixel has multiple neighbors—even corner pixels have three—this approximation cannot be applied consistently.

For a reliable heuristic, computing L for each pixel must account not only for its immediate neighbors but also for more distant pixels. Additionally, the process must begin with at least one pixel whose L value is fixed. Calculating the smoothness penalty V(d,d′) only requires examining the disparity along the straight line between pixels p and q. Furthermore, computing the photometric cost C(q,d′) does not require considering q's neighbors. However, the effects of assigning disparity d′ to q propagate to its neighbors, which in turn affect the calculation of their own L values. This dependency implies that when calculating L_r for a given pixel and its neighbor, the neighbor's value must be determined in advance—which itself depends on the values of its own neighbors, and so on.

Overall, this means we need a heuristic to estimate the ripple effects of assigning disparity d′ to pixel q on the rest of the image. This includes accounting for both nearby and distant neighbors, while ensuring that the S values are already determined for all disparity values of any pixels involved in evaluating these ripple effects. This process must then be repeated for each neighbor of pixel p.

One such heuristic assumes that the ripple effects of assigning disparity d′ to pixel q, for the purpose of calculating $L_r(p,d)$, only influence pixels further away from q along direction r—that is, along the straight line connecting q to p. For example, if p=(55,40) and q=(55,39), which lies directly above p, we consider the ripple effects to extend only to pixels higher in the same column, meaning we must only consider those pixels when considering $L_r(p,d)$ for that direction.

Along each direction, this process continues until reaching a pixel on the edge of the image, which has no neighbor in that direction and therefore contributes no smoothness cost. This allows us to treat the edge pixel as a fixed anchor, enabling direct computation of the cost at the next pixel along the scanline. From there, the costs can be propagated recursively. This linear chain of pixels is referred to as a **scanline**.

Consequently, we consider not only each immediate neighbor of a pixel, but also all downstream pixels along the direction of that neighbor. Each pixel is part of 4 or 8 such scanlines, depending on the chosen connectivity. Each scanline has an associated cost term L_r that represents the cost propagated along the path from a relevant neighbor to the current pixel. For the four cardinal directions, scanlines follow: top→bottom, bottom→top, left→right, and right→left. If using 8 directions, additional scanlines include: top-left→bottom-right, bottom-right→top-left, top-right→bottom-left, and bottom-left→top-right.

Since the smoothness cost of changing a disparity affects both disparity differences and photometric consistency, L_r should account for both. The smoothness cost at a point in a given direction represents the total additional cost—beyond a baseline—incurred by enforcing smoothness for the selected (p,d), including both disparity transitions and photometric differences. This is consistent with our earlier calculation of L_r, where we considered the photometric costs of neighboring pixels and their associated smoothness penalties. The difference now is that we compute this cost along a scanline rather than considering all neighboring pixels and their neighbors.

After computing L_r values along all relevant scanlines, we sum them to obtain the total cost S(p,d), which is then used to select the disparity that minimizes this aggregated cost. This means that overall, we compute each pixel's photometric cost, C(p, d) for each d within the range. For every direction $r \in R$ where q = p - r is the previous pixel along the direction, create an empty cost volume $L_r(p,d)$. For the first pixel on each scanline, set $L_r(p,d) = C(p,d)$ for all d. For each direction r, traverse every scanline in the order of r. For each pixel p (after the first) and each d, use the following equation to get L_r:

$$L_r(p,d) = C(p,\ d) + \min(L_r(q_r, d),\ L_r(q_r, d+1) + P_1,\ L_r(q_r, d-1) + P_1, \min_{d'} L_r(q_r, d') + P_2) - \min_{d'} L_r(q_r, d') \quad (10.39)$$

At each (p,d), sum over directions, noting that we do not have to consider photometric cost since it was considered in calculating L_r:

$$S(p,d) = \sum_r L_r(p,d) \quad (10.40)$$

For each pixel p, pick the disparity with minimal aggregated cost:

$$D(p) = argmin_d S(p,d) \quad (10.38)$$

The result is a dense, piecewise-smooth disparity map that closely approximates a full global solution, computed in $O(|R| \cdot N \cdot D)$ time, where R is the set of directions, D is the max disparity, and N is the number of pixels in the image. It is especially fast because it is a dynamic programming approach where prior calculations can be reused in the recursion.

This overall matching algorithm is known as **Semi-Global Matching (SGM)** (Hirschmuller, 2005).

10.3.4 - ML Approaches

However, hand-crafted stereo methods struggle in low-texture or repetitive regions—think blank walls, whiteboards, or vast skies—and can be easily fooled by patterns like fences or tiled floors. By contrast, a deep network learns higher-level features (edges, corners, even semantic cues) that remain distinctive when raw pixel intensities are ambiguous. Likewise,

simple color matching assumes constant brightness and diffuse materials; specular highlights, cast shadows, or auto-exposure shifts break this assumption. Learned features, however, can become invariant to such lighting and small geometric distortions, preserving reliable correspondences. Classical smoothness penalties often either blur across object boundaries or leave holes where matches are uncertain. In contrast, networks can implicitly learn to respect depth discontinuities—preserving thin poles, wire fences, and tree branches—and even predict occlusion masks where no valid match exists. Finally, real-world scenes—with slanted planes, curved surfaces, or rolling-shutter distortions—often violate simple smoothness priors. A model trained on representative data naturally encodes richer, more flexible "priors" for how disparity varies across diverse environments.

Overall, when you need centimeter-level accuracy, robustness to lighting and material changes, reliable performance on untextured, fine-grained, or highly repetitive regions, or semantic/feature-driven matching, a machine-learned stereo method will outperform purely hand-crafted approaches. Note that this requires a sufficient dataset for training your model or fine-tuning an existing one and a decent bit of real-time computational power.

The primary algorithm for this is a more complex version of a CNN, known as StereoNet (Khamis et al., 2018). The inner workings of this mechanism are outside of the scope of this textbook. As a brief overview, StereoNet takes in two horizontally rectified images and a maximum disparity, processes the images separately to extract a feature map for the left image and one for the right, evaluate combinations of the left feature map and the right-shifted right image for various disparities to develop a cost for each cell/voxel for each disparity, perform upsampling and interpolation to get back to the full image size, then chose the minimal cost disparity for each pixel.

Section V: Multiple Robots

Chapter 11: Multi-Robot Systems

11.1 - Introduction to Multi-Robot Systems

In robotics, many real-world applications involve goals that cannot be accomplished by a single robot alone. This necessity arises either because the tasks are inherently spatially distributed—requiring action or sensing in multiple places at once—or because the task load exceeds the capabilities of one robot in terms of time or efficiency. For instance, in warehouse automation, goods must often be retrieved, sorted, and transported concurrently. If a single robot were responsible for all of these operations, bottlenecks would form, leading to delays. Therefore, to meet the demand for simultaneity in task execution, multiple robots become not merely advantageous but functionally necessary. Similarly, in tasks like environmental monitoring or search and rescue, the fundamental requirement is to gather information or perform actions across a large and spatially diverse environment. A single robot is inherently limited by its sensor range and travel speed; thus, deploying multiple robots becomes the only viable way to satisfy the spatial coverage constraint within a reasonable timeframe. This entails that in scenarios where sensing or intervention must occur in parallel across distinct locations, a **multi-robot system (MRS)** is the only efficient and scalable solution.

Once the need for multiple robots is established, it becomes essential to ensure that the system remains flexible, resilient, and easy to manage. If each robot were unique in design and function, the system would face increased complexity in coordination, fault tolerance, and maintenance. Moreover, if every robot in a MRS were different, they could not easily substitute for one another in the event of a failure, potentially compromising the entire system. To reduce computational burden and ensure interchangeability when needed, it is beneficial to make the robots mechanically and functionally identical. This design choice supports modularity, allowing control strategies, task allocation, and software to be generalized across all units without accounting for individual differences. Such systems are known as **homogeneous** multi-robot systems.

However, many MRS require specialization, where different tasks need to be accomplished. For example, a MRS may require different types of complex sensors that are costly and computationally intensive. Additionally, a MRS may need the ability to navigate on different types of terrain. This need for multiple sensing and actuation capabilities entails that some MRS should have robots with different sensors or mechanical builds. Such an MRS is known as **heterogeneous**.

11.2 - Multi-Robot Coordination

11.2.1 - Introduction to Multi-Robot Coordination

If robots act independently without coordination, they risk duplicating efforts or interfering with one another's operations. For example, in a warehouse, multiple robots might navigate the same aisle simultaneously, leading to congestion or collisions. Independent actions can also result in inefficiencies, such as two robots retrieving the same item or overlooking critical tasks. In search-and-rescue missions, robots must effectively divide the search area to avoid redundant coverage while ensuring no region is left unexplored. More broadly, the system must ensure that overall goals are met—even if individual robots must sometimes sacrifice their own efficiency for the collective good. Coordination enables robots to intelligently divide tasks and align their behaviors, improving collective efficiency, reducing delays, and preventing operational conflicts. In the absence of coordination, robots may remain unaware of each other's planned paths, increasing the risk of collisions—especially in dynamic environments. For instance, in an automated factory, two robots heading toward the same intersection without mutual awareness could collide, causing damage and downtime. Similarly, autonomous delivery robots operating in crowded warehouses must avoid both static obstacles like shelves and dynamic ones like other robots or humans. Shared environments become increasingly difficult to manage when robots act in isolation. By accounting for the positions and trajectories of nearby agents, coordinated systems enable smoother, safer navigation—minimizing accidents and ensuring uninterrupted operation.

In order to maximize system efficiency, ensure the collective system goals are prioritized, and collisions are avoided, the paths that the robots plan must take into account those of the rest of the system. These paths can be local, such as Artificial Potential Field (APF), global such as A* or D*, and/or reinforcement learning algorithms such Deep Deterministic Policy Gradient (DDPG).

11.2.2 - Decentralized Control

For optimal productivity, robots in a multi-robot system must work together effectively as a team. These teamwork challenges can often be decomposed into individual tasks assigned to different robots, with some tasks depending on the status or completion of others. In a single-robot system, the robot manages its tasks using onboard processing, enabling it to interpret information and make decisions autonomously. Similarly, in a multi-robot system, each robot can manage its own tasks and make independent decisions. For example, on a production line, several robots might simultaneously drill holes in different parts of a product. This reflects a paradigm in which each robot operates autonomously while contributing to the collective operation, illustrating a basic form of **decentralized decision-making** in a MRS (Vidyasagar, 1985).

In decentralized decision-making using graph-based path planners such as D*, each robot is assigned its own goal and maintains a local map of the environment based on its sensor data. The robot uses D* to plan a path over this local map and continuously update the path as new information becomes available. When a robot detects changes in the environment, it replans its path independently, enabling it to avoid obstacles and potential collisions. However, because each robot plans without fully accounting for the future movements of other robots, this decentralized replanning can result in suboptimal coordination across the team.

In local path planning with a decentralized control framework, each robot has a potentially unique goal to navigate to. Each robot computes its path based on its perception of the environment. Robots sense nearby obstacles and other robots and adjust their trajectories accordingly. Repulsive forces prevent robots from colliding with each other or obstacles. This is inherently decentralized because robots independently calculate forces based on local data. However, APF can lead to local minima, where robots become "stuck" in suboptimal configurations. Additionally, without global awareness, robots might fail to coordinate optimally over long-term objectives.

In reinforcement learning in decentralized systems, robots each learn their own policy and react to information they sense. To ensure that their goals contribute to the overall system goals, policy training generally involves a reward metric that measures the collective goal fulfillment of the system. This is known as **Centralized Training with Decentralized Execution (CTDE)**. The broader field of RL for multiple agents is known as **Multi-Agent Reinforcement Learning (MARL)**.

11.2.3 - Centralized Control

However, decentralized control introduces several challenges of its own. In such systems, each robot must be equipped with sufficient onboard capabilities to independently handle sensing, computation, storage, decision-making, and communication. While this autonomy allows for flexible and scalable operation, equipping every robot in a large system with powerful processors and memory is both cost-prohibitive and energy-intensive. As the number of robots increases, the cumulative cost and power consumption can become impractical, especially in resource-constrained environments.

Moreover, decentralized systems can lack a global perspective—a unified understanding of the entire environment and the goals of all robots. This perspective is often critical in optimizing the overall system’s performance, particularly when the robots must coordinate closely to achieve a shared objective. While individual robots can maintain local maps and share limited information, giving every robot access to a complete global view would require each one to store and process all available data. This results in redundant computation and data storage, which wastes resources and undermines the scalability benefits of decentralization.

In many cases, robots must plan interdependent actions simultaneously. When each robot plans independently, their actions may become misaligned, leading to inefficient or even conflicting outcomes. Effective coordination requires that each robot consider not only its own objectives but also the goals, current states, and intended actions of all other robots in the system. Achieving optimal collaboration, therefore, demands synchronization of planning across all robots.

As the number of robots increases, this coordination becomes significantly more complex. Each robot must ensure that its plan is not only individually feasible but also compatible with the plans of all other robots. This involves resolving potential conflicts and optimizing interactions between agents. Because the outcome of any one robot's action can affect—and be affected by—the actions of others, no robot can finalize its plan in isolation. Instead, the entire system must develop a single, unified plan that jointly accounts for all actions and constraints.

This requirement implies the need for a single decision-making entity that can generate a coherent global plan. While it is theoretically possible for each robot to compute the full plan for the entire system and then carry out just its own portion, this approach would be highly redundant, expensive, and inefficient. Moreover, the processing power and energy required to make such global decisions are difficult to support onboard mobile robots, which often have limited computational and electrical resources.

As a result, it is more practical to delegate the responsibility of planning to a separate unit that does not move and is specifically designed for centralized computation. This unit, known as the **controller**, makes decisions on behalf of the entire system. This approach is referred to as **centralized decision-making**.

In global path planning for a centralized system, a multi-agent extension of the D* algorithm—known as **M***—is used to plan coordinated paths for all robots. In this approach, a central controller has access to global information, including the current positions and states of all robots, their goal locations, and the locations of all known obstacles in the environment. Using this comprehensive view, the controller computes paths that minimize an overall system cost—such as total time, energy consumption, or distance traveled—while also preventing conflicts between robots.

All robots share a common representation of the environment, meaning they operate based on a unified global map that includes obstacle locations, free space, and possible paths. M* leverages this integrated map to plan joint paths that ensure robots do not interfere with one another—for example, by avoiding situations where multiple robots attempt to pass through the same narrow corridor or occupy the same region simultaneously.

Since the system is centralized, all updates are handled by the central controller. When any robot's state or the environment changes (e.g., due to the appearance of a new obstacle), the

controller recalculates the paths for all robots accordingly. This guarantees that each robot's new path reflects the most up-to-date information about the environment and the real-time positions of all other robots, maintaining global coordination throughout the operation.

In centralized decision making with APF, the central controller computes a combined potential field for all robots, factoring in both attractive forces (goals) and repulsive forces (obstacles and other robots). In RL, a centralized RL agent is trained to control all robots simultaneously, learning policies that maximize a global reward function. The reward function incorporates system-level objectives, such as minimizing task completion time, balancing workloads, and maximizing energy efficiency.

Depending on the system requirements, robotics engineers must decide between decentralized and centralized decision-making and communication. In some cases, a hybrid approach can be beneficial. This involves using a central controller for certain decision-making tasks while allowing individual robots to handle others. One common approach is where a central controller handles overarching system-wide tasks or global tasks, while robots retain the ability to react autonomously to local environmental changes, making it well-suited for complex, dynamic environments. This approach aims to leverage the advantages of both systems, balancing the need for centralized coordination with the flexibility of decentralized operations.

11.3 - Task Allocation

The overall goal of a robot system is to achieve a desirable result. To do so, the robots should execute a series of actions that improve their current state towards the goal state. Those actions would consist of specific motor configurations or movements that transition the robot's current state toward the goal state. However, most overall goals don't have a predetermined implementation that the robots can execute, so it must be derived.

For a goal to be implementable, the robot must execute a predetermined or self-derived function or algorithm. Before executing the function, the robot will look for a specific starting state that satisfies the preconditions of the function. Once that's satisfied, the robot will run the function to derive an implementation until it reaches the goal state from the starting state. This means the function could run once or more until a defined endpoint is reached. This function typically maps sensor data (and sometimes historical data) to a sequence of actions. This discrete actionable unit that achieves a specific goal is called a **Task**.

For example, a robot could be able to derive an implementation that navigates to point X. In this case, the robot will run the function several times to find the best movement action at each step until it reaches the desired position. Another example is picking up object Y using the Z robot arm. In this case, the precondition is that the robot can't pick up object Y unless the robot is in close range to it. The function that determines the process of pressing that button will only

trigger once the starting condition, which is the robot being near the button, is satisfied. In these examples, X, Y, and Z are parameters to the function that specifies the goal state.

Some tasks are complex and require the robot to perform multiple different functions, depending on whether certain conditions have been met or not. This means that the overall goals of the robot consist of subgoals with unique functions, preconditions, and potential ending conditions. The resulting subgoals would be the desired goals of the subtasks of the overall task. The function of the overall task would then combine the resulting outputs of the functions of each subtask to form the implementation of the overall goal. This process of breaking down goals or tasks (task division) is applied until the task cannot be broken down into smaller discrete actionable units, resulting in a single function that does not contain any inner functions mapping sensor data to actions.

Every time a goal is broken down into smaller subgoals, each subgoal will have a set of prerequisites of other subgoals that need to be completed before, affecting the order chosen to complete the subgoals. The first subgoals that need to be completed would have an empty set of prerequisites. For example, if the overall goal was to make a peanut butter and jelly sandwich, that goal could be refined into four subgoals: collecting the ingredients, making the top half of the sandwich with peanut butter, making the bottom half of the sandwich with jelly, and combining the halves of the sandwich. Collecting the ingredients has to be completed first, as the remaining 3 subgoals have that as a prerequisite. On the other hand, combining the halves of the sandwich has to be completed last, as it has a prerequisite of the other subgoals.

Certain subgoals may have a prerequisite that requires having the same robot to execute all the subtasks with those subgoals. For example, the robot that executed the task of picking up the apple is the only robot that can be assigned to execute the task of putting the apple on the table. Other robots simply can't put the apple on the table if they didn't pick up the apple. This means that the subtasks can't be allocated separately and must be allocated using the overall task. This task is known as an **Indivisible Task** and is allocated to one robot.

On the other hand, subgoals that don't require completion of the other and vice versa mean that they are independent and the state of completion of that subgoal won't affect the ability to complete the other. This means that the subtasks can be allocated to distinct robots and executed simultaneously. Using the PB&J example, making the top half and the bottom half of the PB&J sandwich is independent of each other. Therefore, the series of tasks for the two subgoals can be allocated to two robots to be done simultaneously. This increase in efficiency from using multiple robots to perform tasks simultaneously incentivizes the use of Multi-Robot Systems. Unlike indivisible tasks, the subtasks can be allocated to distinct robots and, thus, doesn't need to be allocated as a group using the overall task. This type of task is called a **Divisible Task**. Instead of allocating the overall task, the system allocates its subtasks, where it would consider all possible allocations. This recursively applies to subtasks that are also

divisible, meaning the resulting set of tasks that are allocated to robots will only consist of indivisible tasks.

For each task, there are various possibilities for which robot to allocate it to and when to execute it. The choice of allocation and timing impacts the system's performance, with each overall configuration yielding a different outcome. For example, allocating tasks to robots that are closer to the task's specified location would result in less completion time and a better performance than a robot that is farther away. Tasks also have to be allocated and done in an order that doesn't violate the prerequisites of the subgoals or the constraints of each robot, as that contributes to the success or failure of the system. The challenge lies in finding the allocation of tasks that yield the most optimal performance for the system as a whole. This process is called **Task Allocation**.

In Multi-Robot Task Allocation (MRTA) of n robots, the goal is to find the set of task allocations that result in the most optimal performance. When a task is allocated, the task and the robot assigned is known as the **Robot-Task Pair**. The robot would be from the set of the mobile robots, which is signified by **R**, where r_i ;{i= 1,2, ...n} is the ith robot. The task would be from the set of tasks, which is signified by **T**, where t_j ;{j= 1,2, ...n_t} and n_t is the total number of tasks.

As mentioned before, performance is quantified by a metric called cumulative reward, which is computed using a reward function of the various performance factors. An optimal task allocation configuration is one that maximizes the expected improvement in cumulative reward achieved by the overall robot system, compared to leaving those tasks unallocated. This metric is known as **Overall Utility**. In a search-and-rescue mission, overall utility could measure how efficiently & effectively all robots combined are expected to cover the search area. The goal is to find the task allocation configuration that maximizes the overall utility.

When optimizing using overall utility, the search space of task allocation configurations is n^{n_t}, as there are n possible robots to allocate each task (n*...*n). Even though this will guarantee the global optimal solution, it's inefficient to search through this high dimensional space, especially when the total number of tasks and robots are scaled higher. Through this lens, any change in the task allocation configuration entails an entirely different solution, so it must be re-evaluated by the overall utility metric.

In many scenarios involving task allocation among robots, some tasks are independent of one another, meaning the way one task is assigned does not influence how another task performs when assigned. Consider a simple example where two tasks need to be completed: "slice an apple" and "retrieve a plate." Suppose we have multiple robots capable of performing either task. If we assign "slice an apple" to Robot A and "retrieve a plate" to Robot B—or vice versa—the performance of the apple-slicing task will not affect the plate-retrieving task, and vice versa. The two processes occur independently.

Due to this independence, the overall utility of the system (that is, the combined benefit or performance resulting from assigning tasks to robots) can be decomposed into the utility contributed by each individual robot-task pair. In the above example, the total utility of assigning two robots to two tasks can be computed as the sum of the utilities for each separate assignment, rather than needing to consider the assignments as a coupled or interdependent unit.

If this kind of independence holds for all tasks in the system, we can generalize this principle. Specifically, instead of considering the utility of an entire multi-robot task allocation as a single complex function that depends on all assignments simultaneously, we can instead think of it as the sum of individual utilities—each one resulting from assigning a specific robot to a specific task. This simplification transforms the problem of global task allocation into a series of local assignment problems, one for each task.

As a result, optimizing the overall utility of the system becomes much more computationally efficient. Originally, to determine the best assignment of n_t tasks to n robots, we would have had to search over all possible combinations of full task assignments—an exponentially large space of size n^{n_t}. This means we would consider every possible way to assign each of the n_t tasks to any of the n robots all at once, which becomes infeasible as the number of tasks or robots grows.

However, under the task independence assumption, we no longer need to evaluate complete configurations. Instead, we can optimize the assignment of each task separately. For each of the n_t tasks, we perform a search over the n possible robots that could be assigned to it. This results in n_t independent searches, each of size n, for a total of $n \times n_t$ evaluations. Since $n \times n_t$ grows linearly with the number of tasks and robots, this drastically reduces the computational complexity of the problem from exponential to linear in the number of tasks, making the task allocation problem much more tractable in practice.

Each component of the sum would represent the expected improvement in cumulative reward of assigning each task to a specific robot compared to leaving the task unallocated, known as **Individual Utility**. This can be done by assigning every task to the highest individual utility contributed out of all the robots. In a sense, each robot would enter a bid for each task that captures the utility gained if it completed the task, and the central agent would allocate that task to the highest bidder, similar to an auction. For example, a robot closer to the button than another would have a higher bid for the task of hitting that button. This process would occur for every task, where each allocation would be determined one at a time.

To compute individual utilities, a mathematical model called the **Utility Function** is used. One possible utility function is a human-engineered model, such as a function that evaluates how far the state resulting from a specific robot-task pair allocation is from the goal state. While straightforward, this approach is prone to inaccuracies and will not account for all of the complexity in real-world systems. To address these limitations, machine learning or

reinforcement learning can be applied. For example, in ML, a neural network could be trained on historical reward data to predict the utility of a robot-task pair. In RL, the system could use Q-learning to learn a utility function through trial and error, where robots receive rewards for completing tasks efficiently. These data-driven approaches allow the utility functions to adapt to dynamic environments and complex interactions.

To store and organize these individual utilities in a structured way, the individual utilities are stored in a utility matrix, signified by $\mathbf{U}$, where $\mathbf{u_{ij}}$ is the utility of the i-th robot to execute task j. This allows easy comparison and evaluation of different possible task allocation configurations.

However, the assumption of independence sometimes can't be applied to task-robot pairs with the same robot. Assigning too many tasks to a robot could cause others to remain idle, as there is no consideration of the current workloads of each robot when performing each task allocation. The imbalance causes inefficiency in performance and potential constraint violation, as the idle robot could've executed other tasks other robots are going to do in the future to save resources. Using the previous example, assigning the tasks "slice an apple" and "retrieve a plate" to the same robot will be less efficient if there is another robot that could've executed one of the two tasks simultaneously, showing their allocations aren't fully independent. By allocating all the tasks using constant bids, it doesn't allow the system to consider each robot's workload from tasks that were previously allocated. Instead, the system should consider if it's more efficient to give a robot with a small workload another task than a robot with a high workload, as the high workload robot may take longer to start executing the task that could be completed by another robot in the meantime.

Another complication is that previous task allocations—and other dynamic factors—alter the state from which a robot will begin future tasks. Its location, available energy, and other internal resources may change, directly impacting its utility and bid values for remaining unallocated tasks. For example, if a robot is expected to be low on fuel due to its current workload, it may no longer be capable of taking on additional tasks. As a result, it would lower its bids to reflect this limitation. This interdependence violates the assumption that all robot-task pairs are independent.

As a result, the strategy of optimizing individual utilities independently based on constant bids must be revised to incorporate new information as tasks are allocated. In a global context, the utility of each robot-task assignment depends on all other task allocations, meaning that a single change in the configuration can lead to a completely different solution. This interdependence reflects the fact that utilities are non-modular. However, since tasks are assigned sequentially for computational purposes, the system has access to past allocations and each robot's current state when making the next decision—but it lacks information about future allocations. To maintain computational efficiency, we approximate the utility of each assignment

using only the information available from previous allocations, allowing the system to make informed decisions without evaluating the full space of possible configurations. For instance, if Robot A has just completed a high-energy task and now has limited battery, the system will account for this when estimating its utility for upcoming tasks—perhaps assigning it a nearby, low-effort task rather than one that requires long-distance travel. This decision is made without knowing what tasks will appear later, but still adapts dynamically to the robot's evolving state.

Using this information, each robot would recalculate their individual utility for unallocated tasks. They would adjust their bid and communicate their bids to the auctioneer, who would update the utility matrix and make the next task allocation. Changing every robot's bids would take into account all potential interdependencies between tasks, but this is computationally intense. If previous task allocations don't affect other robots, the bids could be updated only for the task previously allocated to a robot.

Also, task allocations must be done in an order so they don't violate dependencies and prerequisites and follow inherent orders of certain tasks. To satisfy prerequisites, the tasks that are considered to be allocated to be executed at a given time must either have no prerequisites or prerequisites that are already all satisfied. Determining the first task allocation and the process of determining the next task allocation based on the bids varies. One greedy approach is to first allocate a task and robot pair with the highest overall utility bid, as this locally maximizes utility. Using this logic, the next task allocation would be the unallocated task and robot pair with the highest utility bid. However, each task might have different importance in the completion of the overall goal, where a major task could have a significant impact on the success or failure of the MRS. Therefore, another approach is to weigh each utility alongside task priorities and use the greedy algorithm with the adjusted utilities. These auction-based approaches are known as **Market-Based Methods** (Hussein & Khamis, 2013).

To capture the task allocations that result in the best global performance, a central agent must be able to access full information about the system, tasks, and constraints from each robot to fully gauge all the advantages of each robot executing a certain task. After finding the optimal allocation, this agent would then send commands of the tasks each robot is allocated in return. This approach is known as **Centralized Allocation**.

However, this approach has a high reliance on a single central agent. If the agent fails, the whole system will fail. It also limits the scalability of the system, as the central agents have limits on how many robots and tasks it can work with. To reduce the reliance on a central agent, the administrative tasks of task allocation could be dispersed between the agents of the multi-robot system. Each robot would communicate its information with the others and allocate tasks themselves. It wouldn't be able to produce a good global solution due to incomplete information but it can produce a good local solution with more robustness and scalability. This approach is known as **Decentralized Allocation**.

In a centralized approach, the central agent can act as the auctioneer, receiving bids from each robot and allocating tasks in return. In a decentralized approach, there is no central agent to conduct the auction, but robots can implement market-based methods autonomously. With no outside agent making the decisions, the robots are required to exchange information and collaborate to reach consensus on task allocation decisions themselves. These methods are called **Consensus Algorithms**. In some cases, each robot takes turns acting as the auctioneer. However, achieving consensus in decentralized systems can introduce challenges, such as communication delays or conflicts, which require robust coordination protocols. By leveraging either centralized or decentralized approaches, MRTA systems aim to optimize task assignments, ensuring efficient use of resources while addressing fairness and scalability in complex, real-world scenarios.

The decisions the agent makes on the set of task allocations rely on using information about the current state of each robot and its environment and all the tasks needed to be completed. To determine the set of task allocations that results in the most optimal performance, it is suitable to use the information of all the tasks and allocate them all to the robots before execution starts. This is known as **Static Task Allocation**.

However, not all tasks allocated will be done immediately by a robot. Therefore, any changes to the state and environment as the robots are performing current tasks could cause suboptimality in performance when doing future tasks. Static task allocation can account for predictable changes in the utility function ahead of execution. However, if it's not predictable and more stochastic, the utility function wouldn't be able to fully capture the accurate utility gained for future tasks, only tasks that are immediately done. Therefore, instead of allocating all the tasks, allocating tasks that are immediately going to be performed based on new information can account for new changes and make the locally optimal allocations. After a robot is done with their current task, the agent can allocate an uncompleted task based on the current state until all tasks are complete. This is known as **Dynamic Task Allocation**.

There are other factors to keep in mind that impact the chosen task allocation approach. In systems of robots with the same capabilities, also known as **Homogeneous Systems**, the robots have no physical differences to consider when allocating tasks. On the other hand, if the robots have diverse capabilities, also known as **Heterogeneous Systems**, it would require matching tasks to robot abilities. The robots could also be limited to work on one task at a time or multiple tasks at a time. The assignment and order of task execution for each robot could also heavily impact the overall performance.

11.4 - Multi-Robot Communication

11.4.1 - Introduction to Multi-Robot Communication

As we know from uni-robot systems, for a robot to function properly, it requires information about its environment, status, and task progression/completion. Through their

sensors, robots can gain information as to their state and the state of their immediate environment. Robots can remember this information by creating a map. However, there are some limitations with this approach. Each robot can usually only sense a subsection of their environment. Also, robots cannot gain knowledge of the current state of imperceptible locations and aspects of their environment. These issues are even more prevalent in multi-robot systems, where there are additional key environmental aspects such as the locations of other robots and the tasks they are accomplishing. If robots do not know each other's tasks and locations, they may get in each other's way, try to perform the same task, not optimally cover a region, etc. This means that each robot needs to know the state and tasks of each other robot as well as key environmental data that they have sensed. Since each robot knows its tasks and information it has gained by its sensors, a lot of the information a robot needs or would benefit from knowing is known by another robot. This shows that robots in a multi-robot system cannot sense and act independently, but must instead share knowledge with each other. This is known as **Multi-robot communication**.

In a decentralized control system, each robot independently senses data and makes decisions, so it needs to share its observations with others and be aware of information gathered by its peers. Direct communication allows robots to transmit data directly to other robots with no intermediary steps, facilitating this information exchange. This method is known as **decentralized communication**.

In centralized control, since the central controller handles all processing and decision-making, individual robots do not need to process environmental or task information themselves. Instead, they only need to send information to and receive instructions from the central controller. This setup is known as **centralized communication**.

However, centralized communication comes with significant challenges. Since all information must pass through the central controller, it creates a single point of failure; if the controller fails, the entire system is at risk. Moreover, as more robots are added, the load on the controller and the communication network increases, potentially leading to delays and congestion.

11.4.2 - Direct vs. Indirect Communication

In multi-robot systems, specific robots often need to receive specific information from others quickly, at a precise moment, or when certain conditions are met. For example, in collision avoidance, robots must communicate their positions and trajectories to prevent accidents. If two robots are moving toward the same point, one must immediately signal the other to adjust its path or stop to avoid a collision. This exchange of information needs to happen in real time, as even a slight delay could result in a crash. Another example is robot coordination

during object handling. In a warehouse, one robot might lift an object while another waits to transport it. The second robot must be informed as soon as the object is ready to be picked up.

These scenarios highlight the need for robots to send direct messages to one another, a process known as **direct communication**. Direct communication can be compared to human interaction, such as speaking or calling each other. In multi-robot systems, this type of communication can occur through various channels, including Wi-Fi, cellular networks, wired connections between robots, etc.

However, there are many situations where direct communication may not be viable because information either doesn't need to be communicated immediately or cannot be directly communicated to specific robots. In some cases, communication without electronics is necessary. Direct communication requires each robot to establish and maintain a connection, which can be resource-intensive and impractical in large environments where robots are dispersed. Additionally, it is often unclear which robots will need specific information or when, as needs may arise in particular scenarios. Lastly, as the number of robots increases, communication overhead grows, potentially leading to network congestion and decreased performance. When many robots communicate simultaneously, the network can become saturated, causing delays, dropped messages, and reduced efficiency. These challenges underscore the need for a more scalable and flexible communication method, where information can be shared without constant direct exchanges.

In situations where a robot needs information but direct communication is not required, some information becomes relevant only when the robot encounters specific conditions. A common scenario is when a robot requires information upon entering a particular region. For instance, a robot might need to know the speed limit of a street or adhere to regional protocols in a warehouse. In such cases, robots must gather this information independently, without relying on direct communication from other robots. This means that the robots must depend on their sensors to detect relevant environmental cues. To facilitate this, the communicating robot must alter the environment in a way that allows other robots to perceive and interpret the changes. An example of this is flipping a "closed" sign at a restaurant. This method of communication, where robots share information by leaving signals or markers in the environment that can later be recognized by others, is known as **Indirect Communication** (Stigmergy). Depending on the communication requirements, indirect communication can either replace or complement direct communication.

Since environmental markers can sometimes be difficult to identify or interpret, many robotic systems use dedicated physical markers, such as QR codes, RFID tags, or painted lines, to convey specific instructions or data to passing robots. For example, in automated warehouses, tags are strategically placed throughout the facility. Robots can scan these tags to upload or retrieve information about inventory status, including the location, quantity, and order status of

specific items. Additionally, to provide robots with specific information upon entering a region, systems can be pre-programmed to deliver relevant data based on location criteria. For instance, robots can deposit virtual information (e.g., geotags) into shared databases, which other robots can access when they enter the designated area.

Conclusion

Robotics is ultimately the study of how machines perceive, reason, and act in the world. In this textbook, we have explored the foundations of perception, decision-making, and planning—from low-level sensor interpretation to high-level task coordination. The goal has not simply been to describe algorithms, but to uncover the structure and insight that guide them—the reasons behind the rules.

While the field continues to evolve, the core challenge remains the same: enabling intelligent action in uncertain, dynamic environments. As you move beyond these pages, whether into research, industry, or your own creative projects, we hope you carry forward the mindset of principled design—one that prioritizes clarity, structure, and purposeful reasoning in every robotic system you build.

The future of robotics is open-ended. This book is only the beginning. Always remember: if you don't understand why an algorithm must be optimal, there's a chance it isn't.

Glossary

Section I: Introduction to Cognitive Robotics

Chapter 1: Introduction to Perception & Planning

Actuation - The process by which a robot converts planned motion or control signals into physical movement, typically using motors, servos, or other actuators. It is the final stage of the sense–plan–act loop.

Action - A discrete or continuous control command that a robot can take at a given time, such as changing velocity, steering angle, or executing a grasp. In planning and reinforcement learning, actions transition the robot between states.

Action Space - The set of all possible actions a robot can perform. This space may be discrete (e.g., turn left or right) or continuous (e.g., choose any steering angle between –30° and 30°), and is a key component of planning and control algorithms.

Autonomous - A robot is autonomous if it can perform tasks without human intervention by sensing the environment, making decisions, and acting accordingly. Full autonomy implies self-navigation, perception, and planning in dynamic or unknown environments.

Reward Function - In machine planning, this function assigns a numerical value (reward) to each action or sequence of actions, guiding the robot toward desirable behavior. It encodes the task objectives, such as minimizing energy or reaching a goal quickly.

Sensor - A hardware device that collects information from the environment, such as cameras, LiDAR, IMUs, or GPS. Sensors provide the raw data needed for perception and localization.

Observation - A sensor reading or set of readings received by the robot at a given time. Observations may be noisy or incomplete and are used to estimate the state of the environment.

Observation Space - The set of all possible observations a robot can receive from its sensors. In partially observable systems, this space is typically larger than the state space and may be high-dimensional (e.g., images or point clouds).

Partial Observability - A condition in which the robot cannot directly access the full state of the environment, often due to sensor limitations or occlusion. This requires estimation or belief modeling for effective planning.

State - A complete description of the environment or the robot's situation at a given time, sufficient for predicting future outcomes. Examples include position, orientation, and velocity of the robot and surrounding objects.

State Space - The set of all possible states a robot can be in. In planning and decision-making, this space defines the domain over which policies, paths, and control strategies operate.

Information Processing - The transformation of raw sensor data into meaningful representations that can be used for decision-making. In robotics, this includes filtering, feature extraction, perception, and data fusion.

Machine Perception - The use of algorithms to interpret sensory data (e.g., images, depth maps, point clouds) and extract useful information, such as object identity, location, or motion. It enables a robot to perceive its environment.
Markov Property - A property of decision processes where the future state depends only on the current state and action—not on past states. This assumption underlies many planning and reinforcement learning algorithms in robotics.
Machine Planning - The process of selecting a sequence of actions that achieve a goal. This includes path planning, motion planning, and task planning. Planning may operate in continuous or discrete spaces, and under uncertainty.
Robot - An autonomous or semi-autonomous system that can sense its environment, make decisions, and act on the world.
Algorithm - A step-by-step computational procedure for solving a problem. In robotics, algorithms are used for tasks such as localization, mapping, planning, and learning from experience.
Cognitive Robotics - A subfield of robotics that emphasizes high-level reasoning, learning, and decision-making. Cognitive robots use perception and planning to adapt their behavior, often incorporating elements of memory, inference, and goal modeling.

Chapter 2: Machine Learning

2.1 - Introduction to Machine Learning

Approximately Correct - An outcome or hypothesis is said to be approximately correct if its error rate is within an acceptable bound (ε) of the optimal error. This concept forms the foundation of learning guarantees like PAC learning.
Probably Approximately Correct (PAC) - A formal learning framework where a learning algorithm is said to succeed if, with high probability (at least $1 – \delta$), it returns a hypothesis whose error is at most ε. PAC learning defines what it means for an algorithm to "learn" from data in a probabilistic sense.
Stationarity - The assumption that the data distribution does not change over time. In supervised ML, this means that the joint distribution of inputs and outputs remains constant between training and testing, which is crucial for generalization.
Training - The process of fitting a machine learning model to a dataset by adjusting its parameters to minimize error on the training data. Training involves exposure to labeled data in supervised learning.
Underfitting - A situation where a model is too simple to capture the underlying patterns in the data, resulting in high error on both the training and test sets. Underfitting often occurs when the model lacks sufficient complexity or capacity.

Generalization - The ability of a model to perform well on new, unseen data. A model that generalizes well captures the underlying structure of the data rather than memorizing the training set.
Overfitting - When a model fits the training data too closely, including noise or random fluctuations, at the expense of performance on new data. Overfitting leads to poor generalization.
Testing - The process of evaluating a trained model on a separate dataset not seen during training. Testing estimates how well the model generalizes to new data.
Regression - A supervised learning task where the goal is to predict a continuous output variable given input features. Examples include predicting temperature, price, or steering angle.
Mean Squared Error (MSE) - A common loss function for regression tasks. It measures the average of the squares of the differences between predicted and actual values. Lower MSE indicates better predictive performance.
Loss Function - A mathematical function that measures the discrepancy between a model's prediction and the true output for a single data point. It quantifies how "bad" a prediction is, guiding the learning process. Common examples include Mean Squared Error for regression and Cross-Entropy for classification.
Machine Learning (ML) - A field of study concerned with designing algorithms that learn patterns from data and improve their performance over time without being explicitly programmed for every task.
Supervised Machine Learning (SML) - A branch of ML where the algorithm is trained on labeled data—that is, pairs of inputs and corresponding correct outputs. The goal is to learn a function that maps inputs to outputs accurately on unseen data.
Inductive Bias - The set of assumptions a learning algorithm uses to generalize beyond the training data. Every learning system must have some inductive bias to make predictions about unseen inputs, such as preferring simpler models (Occam's Razor) or assuming smoothness.
Parameter - A tunable value within a model that is learned from data during training. For example, the weights in a neural network are parameters. Parameters define the model's behavior and are adjusted to minimize loss.

2.2 - Optimization

2.2.1 - Iterative Optimization

Objective Function - A broader term that refers to the function being optimized during learning. In supervised ML, the objective function is usually the total (or average) loss over the training dataset. The goal of training is to find model parameters that minimize this function.
Analytical Optimization - An approach where the optimal parameters are found by solving for the minimum (or maximum) of the objective function using closed-form mathematical expressions. This typically involves setting derivatives to zero and solving equations directly.

Iterative (Numerical) Optimization - A method for minimizing an objective function when a closed-form solution is impractical or impossible. The model parameters are updated step-by-step using algorithms such as Gradient Descent. These techniques approximate the minimum over time through repeated computation rather than direct solution.

2.2.2 - Gradient Descent

Gradient Descent - An iterative optimization algorithm used to minimize a differentiable objective function by moving in the direction of the negative gradient. In ML, it adjusts model parameters to reduce the loss function, step by step.
Learning Rate - A hyperparameter that controls the size of the steps taken during gradient descent. A learning rate that is too high may cause divergence; too low may lead to slow convergence or getting stuck in local minima.
Hyperparameter - A configuration variable set before training that governs the learning process but is not learned from data. Examples include learning rate, batch size, number of epochs, and model architecture choices (like number of layers). Hyperparameters are tuned to optimize model performance.
Epoch - One full pass through the entire training dataset. During an epoch, the model sees every training example once, though possibly in smaller groups (batches). Multiple epochs are typically needed for the model to converge.
Batch Size - The number of training examples used to compute one update of the model parameters during training. It determines how much data is processed at once. Batch size affects memory usage, convergence stability, and training speed.
Batch Gradient Descent - A variant of gradient descent where the gradient is computed using the entire training dataset at once. It produces stable updates but can be slow and memory-intensive for large datasets.
Stochastic Gradient Descent - A variant where the gradient is computed and parameters are updated using a single randomly selected data point per step. It introduces noise into updates, which can help escape local minima, but may lead to highly fluctuating convergence.
Mini-Batch Gradient Descent - A compromise between batch and stochastic methods. The model is updated using small subsets (mini-batches) of the training data. It combines the stability of batch gradient descent with the efficiency and generalization benefits of SGD, and is the most common form used in deep learning.

2.3 - Neural Networks

2.3.1 - Piecewise Universal Function Approximation

Universal Function Approximator (UFA) - A class of models capable of approximating any continuous function on a compact domain to arbitrary accuracy, given sufficient parameters or

complexity. Neural networks are UFAs under certain conditions.
Piecewise Function - A function defined by different expressions over different intervals of the input space. These sub-functions are stitched together to form the overall function.
Piecewise Universal Function Approximator - A model that approximates any continuous function by composing simple functions over small input regions. For example, decision trees and ReLU-based neural networks act as piecewise approximators by modeling different behaviors in different regions of input space.

2.3.2 - Piecewise Linear Universal Function Approximation

Piecewise Constant - A function that takes a constant value within each region of its domain, but may change abruptly at boundaries. Example: step functions or decision trees without regression leaves.
Piecewise Linear - A function composed of linear segments, each defined over a specific region of the input. ReLU-based neural networks form piecewise linear functions, where the network's output is linear within each activation region.

2.3.3 - Shallow Neural Networks

Activation Function - A non-linear function applied to a neuron's output. It introduces non-linearity into the network, allowing it to learn complex functions. Common examples: sigmoid, tanh, ReLU.
Rectified Linear Unit (ReLU) - A popular activation function defined as: $\text{ReLU}(x) = \max(0, x)$. ReLU introduces sparsity and forms the basis for piecewise linear approximations in deep learning.
Neural Network - A model composed of layers of connected nodes (neurons), where each neuron applies a weighted sum of inputs followed by a non-linear activation. Neural networks learn complex functions through layered composition.
Shallow Neural Network - A neural network with only one hidden layer between the input and output. Despite being "shallow," such networks can still be universal function approximators if sufficiently wide.
Layer - A set of neurons at the same depth in the network. Each layer processes input from the previous layer and passes output to the next.
Input Layer - The first layer of a neural network that receives raw data (e.g., pixels, sensor values) and passes it to the hidden layers.
Output Layer - The final layer that produces the model's prediction (e.g., class label, continuous value). The activation function here depends on the task (e.g., softmax for classification, linear for regression).
Hidden Layer - Any layer between the input and output layers. These layers extract increasingly

abstract features and enable the network's representational power.

2.3.4 - Deep Neural Networks

Tensor - A multi-dimensional array (generalization of scalars, vectors, and matrices). In deep learning, inputs, parameters, and activations are often represented as tensors to allow efficient computation on GPUs.

Forward Pass - The process of computing the output of a neural network by passing input data through all layers using current weights. It evaluates the function defined by the network.

Backpropagation - An algorithm for training neural networks by computing gradients of the loss function with respect to each weight using the chain rule. It propagates errors backward from the output to earlier layers during training.

Deep Neural Network - A neural network with multiple hidden layers. Deep networks enable hierarchical feature learning and are capable of representing highly complex, nonlinear functions.

Model Architecture - The structural design of a neural network, including the number and type of layers, activation functions, connections, and other components. It determines the model's capacity and inductive bias.

2.4 - Classification

Category - A named grouping or label used to describe a type of item or outcome. In ML classification, categories are the high-level groupings that the model assigns inputs to, such as "car", "cat", or "pedestrian."

Class - A specific target label in a classification task. Each data point in supervised classification is associated with a class, and the model's goal is to predict the correct class based on input features.

Classification - A supervised learning task in which the goal is to assign an input to one of a set of predefined classes.

Decision Boundary - A surface (in input space) that separates different predicted classes. It defines where the model changes its classification decision and is shaped by the model's parameters and training data.

Softmax - An activation function that converts a vector of real-valued scores into a probability distribution over multiple classes. Used in the output layer of multi-class classification models to interpret scores as class probabilities.

Categorical Cross-Entropy - A loss function used in multi-class classification tasks where the true labels are one-hot encoded. It compares the predicted probability distribution (via softmax) with the true class.

Multi-Class Classification - A classification task where each input is assigned to exactly one of

three or more possible classes. The model typically uses softmax activation and categorical cross-entropy loss.

Binary Classification - A special case of classification where there are only two possible classes (e.g., spam vs. not spam). The output is usually a single probability interpreted via a threshold (e.g., 0.5).

Sigmoid - An activation function used in binary classification that maps any real number to the interval (0, 1). It is used to model the probability of the positive class.

Binary Cross-Entropy - A loss function for binary classification tasks. It measures the difference between the predicted probability and the true label. This loss is minimized when the predicted probabilities match the true labels.

Section II: Planning

Chapter 3: Path Planning and Mapping

3.1 - Introduction to Path Planning

Continuous - A domain or space in which variables (e.g., positions, velocities, or actions) can take any value within a range. In path planning, continuous planning involves computing trajectories through a continuous configuration space, often requiring numerical optimization.

Discrete - A domain or space consisting of a finite or countable set of states or actions. In discrete path planning, the robot moves through a graph or grid where each node and edge represents a predefined state and action (e.g., turning left or moving forward one cell).

Navigation - The overall process by which a robot moves from a starting point to a goal while avoiding obstacles. Navigation includes perception, localization, path planning, and control, and is often used as an umbrella term in mobile robotics.

Path Planning - The process of computing a collision-free path from a start to a goal location in the robot's environment. It typically ignores time and dynamics and focuses on the geometric feasibility of the route. The path may later be time-parameterized for execution.

Motion Planning - A more general form of planning that computes physically feasible trajectories, taking into account kinematics, dynamics, and constraints such as velocity, acceleration, or curvature. Motion planning includes path planning as a subproblem.

Task Space - A space defined by the robot's end-effector or body position in the environment (e.g., Cartesian coordinates of a robot arm's gripper). In mobile robots, task space typically refers to workspace positions (e.g., x,y) rather than internal joint values.

Joint Space - The space defined by the values of the robot's joint variables (e.g., angles, wheel rotations). Path planning in joint space involves planning trajectories for actuators, often used in manipulator arms.

Configuration Space (C-space) - An abstract space in which each point represents a complete

configuration of the robot, including position and orientation, and sometimes joint states. Obstacles in the physical environment are transformed into regions in C-space, making path planning a geometric search problem.

Holonomic - A system is holonomic if every degree of freedom (DOF) is directly controllable, meaning the robot can move in all directions allowed by its configuration space. Example: a drone or omnidirectional robot that can move laterally and rotate independently.

Nonholonomic - A system with kinematic constraints that limit motion, such that not all degrees of freedom are directly accessible. Nonholonomic robots, like cars or differential-drive robots, cannot move sideways and require curved paths to reach certain configurations.

3.2 - Introduction to Machine Planning

Cost - A numerical penalty assigned to taking an action or entering a state. In planning, the goal is often to minimize the total accumulated cost along a trajectory. Cost is the inverse of reward and encodes task difficulty, resource use, risk, or undesirability.

Policy - A mapping from states to actions that defines the agent's behavior. In planning, a policy tells the robot what action to take in each situation. Optimal policies minimize expected cost or maximize expected reward over time.

Episodic - A task structure where the planning process is broken into episodes—each with a clear starting state and terminal condition (e.g., reaching a goal). The agent resets at the start of each episode. Most reinforcement learning tasks are modeled this way.

Continuous - A task structure where the planning process does not have a set end, requiring planning for an infinitely receding horizon.

Trajectory - A sequence of states and actions representing the path taken through state space under a policy. Trajectories may be fixed (open-loop) or generated by reacting to observations (closed-loop), and are evaluated based on accumulated cost or reward.

Bellman Equations - Recursive equations that define the value of a state (or state-action pair) in terms of immediate cost and the value of successor states. They are central to dynamic programming and reinforcement learning. These equations characterize the optimal policy.

Stochastic - A system is stochastic if transitions or outcomes are governed by probability distributions. In planning, this means the same action in the same state may lead to different next states, requiring policies that reason about uncertainty.

Deterministic - A system is deterministic if the outcome of every action is predictable—that is, taking the same action in the same state always leads to the same next state. Deterministic planning is simpler but often unrealistic in real-world robotics.

Static - A static environment is one where the world does not change unless the agent acts. All dynamics are under the agent's control, and external events (e.g., moving obstacles) do not occur. Many planning algorithms assume static settings.

Dynamic - A dynamic environment contains time-dependent or autonomous changes beyond the agent's control, such as moving obstacles or changing goals. Planning in dynamic environments requires real-time updates and prediction of future world states.

3.3 - Global Path Planning

Global Planning - The process of computing a high-level path from the robot's current position to the goal using a map of the environment. Global planning operates in configuration or task space, usually over long distances, and typically assumes a static map. Examples include A*, Dijkstra's algorithm, and PRM.

Map - A spatial representation of the environment used for planning and navigation. Maps may encode obstacle locations, free space, semantic objects, or terrain types, and can be 2D (e.g., occupancy grids) or 3D (e.g., voxel maps, OctoMaps). Planning algorithms rely on maps to reason about paths and collisions.

Local Planning - The process of computing short-term, reactive trajectories based on local sensor data and immediate obstacles. Local planning refines the global plan to ensure safe, feasible movement under real-world constraints like dynamics, uncertainty, or unexpected changes. Examples include Dynamic Window Approach (DWA), Model Predictive Control (MPC), and local trajectory rollouts.

3.4 - Mapping

3.4.1 - Map Creation

Point Cloud - A set of 3D points representing surface measurements in the environment, typically generated by LiDAR or depth cameras. Each point has spatial coordinates (x,y,z) and may include additional data like intensity or color. Point clouds serve as raw inputs for 3D mapping.

Dense Mapping - A mapping technique where a large number of measurements are used to reconstruct fine-grained details of the environment. Dense maps cover space continuously and are often built from high-resolution depth or stereo data. They are computationally expensive but highly informative.

Sparse Mapping - A mapping technique that captures only key features or selected points from the environment, rather than covering space densely. Sparse maps are used in feature-based SLAM and visual odometry to save memory and computation.

3.4.2 - Grid Mapping

Map Representation - The data structure or format used to store information about the environment. Map representations vary based on resolution, dimensionality (2D vs. 3D), and encoding of occupancy, elevation, or semantic data. Choosing a map representation impacts

planning and memory usage.
Voxel - A volumetric pixel—i.e., a cube-shaped element in a 3D grid. Voxels discretize 3D space for mapping, and each voxel may store occupancy, distance, or other information. Voxel maps are common in LiDAR-based 3D mapping.
Grid Mapping - A technique that divides the environment into a uniform or hierarchical grid and estimates whether each cell or voxel is occupied or free. Grid maps are widely used in 2D navigation and can be extended to 3D.
QuadTree - A hierarchical 2D grid representation in which each square cell is recursively divided into four smaller squares as needed. It allows variable resolution depending on local complexity, making it efficient for large-scale mapping.
OctTree - The 3D counterpart to a QuadTree. Each cubic cell (voxel) can be recursively divided into eight smaller cubes. OctTrees efficiently represent sparse 3D environments and are used in data structures like OctoMap.
Occupancy Grid Map - A map that divides space into discrete cells, where each cell holds a probability (or binary value) indicating whether it is occupied or free. Built from sensor data using probabilistic updates, occupancy grids are a foundational map representation in mobile robotics.
Uniform Grid Map - A grid map with fixed-size cells throughout the environment. It offers simple memory access and is often used in occupancy grids, but can become inefficient when high resolution is needed in only certain regions.
Variable Grid Map - A grid map that uses adaptive cell sizes, increasing resolution where more detail is needed. Representations like QuadTrees and OctTrees are examples. Variable grids balance memory efficiency with mapping accuracy.

3.5 - Graph-Based Planning Algorithms

3.5.1 - Introduction to Graph Based Path Planning

Discrete Action Space - A set of predefined actions that a robot can take at any point, such as "move forward," "turn left," or "rotate 90°." These actions correspond to edges in a planning graph and simplify planning by constraining possibilities.
Discrete State Space - A finite set of distinct states the robot can be in, such as grid cell positions or predefined poses. Graph-based planners treat each state as a node in a graph, facilitating algorithmic search.
Waypoint - A specific location or configuration used to construct a path. Waypoints can be used to represent intermediate goals or samples in the environment through which a robot must pass.
Graph - A mathematical structure consisting of nodes (vertices) and edges (connections). In robotics, graphs represent possible robot configurations (nodes) and feasible transitions between them (edges).

Node - A vertex in a graph representing a robot state or location, such as a grid cell, sampled configuration, or waypoint.
Edge - A connection between two nodes in a graph that represents a valid motion or action from one state to another. Edges may have associated costs (e.g., distance, time, energy).
Topological Map - A graph-based representation of an environment that emphasizes connectivity over precise geometry. Useful for high-level planning, it shows how locations are connected without encoding exact positions.
Graph-Based Path Planning - A family of planning algorithms that operate over a graph by searching for the optimal path from a start node to a goal node using graph search techniques such as A*, Dijkstra, or D*.

3.5.2 - Single vs. Multi-Query Path Planning

Single Query Path Planning - A planning strategy that constructs a plan for one specific start and goal pair. The graph or search space is built or explored specifically for that instance (e.g., RRT).
Multi Query Path Planning - A strategy that precomputes a roadmap or search structure (e.g., PRM, Cell Decomposition) that can be reused for multiple different start-goal pairs. Efficient in environments with many planning queries.

3.5.3 - Cell Decomposition

Cell Decomposition - A path planning method that divides the free space into non-overlapping cells, then searches for a path by connecting adjacent free cells. Cells can be uniform (grid) or adaptive (QuadTree).

3.5.4 - Probabilistic Roadmaps (PRM)

Sampling Based Path Planning - A class of planners that sample random configurations in the free space and connect them to form a roadmap or tree. These planners are useful in high-dimensional or complex spaces.
Probabilistic Roadmap (PRM) - A multi-query sampling-based planner that builds a graph (roadmap) by randomly sampling collision-free configurations and connecting nearby samples with valid edges. It can be reused for multiple goals.

3.5.5 - Dijkstra's Algorithm

Dijkstra's Algorithm - A graph search algorithm that finds the shortest path from a start node to all other nodes using uniform edge costs. It guarantees optimality but is slower than A* when goal-directed planning is needed.

3.5.6 - A* Algorithm

Heuristic Function - A function that estimates the cost from a node to the goal. In A*, it guides the search toward the goal more efficiently. A good heuristic speeds up planning while preserving optimality.

A* - A best-first search algorithm that finds the optimal path by combining the cost so far (g-value) with a heuristic estimate of the remaining cost (h-value). A* balances efficiency and optimality.

3.5.7 - D* Algorithm

D* - A dynamic replanning algorithm designed for environments where the map may change during execution. D* efficiently updates previous plans as new information becomes available, making it ideal for real-time robotics.

3.5.8 - Rapidly Exploring Random Trees

Steering - A function used in RRT that connects two points in configuration space by generating a feasible path segment, often subject to constraints like maximum curvature or collision avoidance.

Voronoi Region - The region of space closest to a given point (node) relative to others. RRT exploits the Voronoi bias, naturally favoring exploration of large unexplored areas by sampling in these regions.

Rapidly-Exploring Random Trees (RRT) - A single-query sampling-based planner that incrementally builds a tree rooted at the start configuration by randomly sampling points in the space and expanding the tree toward them. It is fast and suited to high-dimensional, complex spaces but does not guarantee optimality.

3.6 - Local Path Planning

3.6.1 - Introduction to Local Path Planning

Local Path Planning - The process of computing short-term motion commands or trajectories in real-time, based on the robot's current state and nearby environment. Local planners operate at a finer time and spatial scale than global planners and are responsible for avoiding dynamic obstacles, satisfying kinematic constraints, and adjusting to map inaccuracies. Local planning typically refines a global path while reacting to sensor data.

3.6.2 - Artificial Potential Field

Artificial Potential Field (APF) - A local path planning method that models the robot as a particle moving under the influence of artificial forces. Attractive forces pull the robot toward the goal. Repulsive forces push the robot away from obstacles. The combined “potential field”

guides the robot's motion. APF is computationally simple and responsive but can suffer from issues like local minima and oscillations near obstacles.

Chapter 4: Reinforcement Learning

4.1 - Introduction to Reinforcement Learning

Reinforcement Learning - A learning framework in which an agent learns to make decisions by interacting with an environment, receiving rewards or penalties, and adjusting its behavior to maximize long-term cumulative reward.

4.2 - Reward Functions

4.2.1 - Bellman Equations

Credit Assignment Problem - The challenge of determining which actions were responsible for received rewards, especially when rewards are delayed. Solving this is crucial for learning effective policies.

Discount Factor - A value $\gamma \in [0,1]$ that determines how much future rewards are weighted relative to immediate rewards. Lower values favor short-term gains; higher values encourage long-term planning.

Discounted Cumulative Reward (Return) - The total reward an agent expects to receive, discounted over time. This is used as the optimization objective in RL.

4.2.2 - Reward Design

Direct Reward - A reward signal that directly reflects task performance, such as giving +1 for reaching the goal and 0 otherwise.

Sparse Rewards - Reward signals that are given only rarely or at the end of an episode. They make learning difficult because useful feedback is infrequent.

Dense Rewards - Frequent reward signals that provide continuous feedback, such as penalizing distance to the goal at every timestep.

Delayed Reward - A reward received some time after the action that caused it. This makes credit assignment more difficult.

Shaped Reward - A modified reward function that includes additional feedback to guide learning (e.g., heuristic bonuses), designed to accelerate convergence but must be crafted carefully to avoid biasing the policy.

Immediate Reward - The reward received directly after taking an action, used to compute return and update the value function.

4.3 - Exploration vs. Exploitation

Exploration - The act of trying new or less certain actions to gather more information about the environment or improve future decision-making.
Markov Decision Process (MDP) - A mathematical model for RL environments defined by states, actions, transition probabilities, rewards, and a discount factor, assuming the Markov property (future depends only on present state and action).
Exploitation - Choosing the action believed to yield the highest reward based on current knowledge. It focuses on using known information rather than gathering new data.
Greedy Strategy - Always choosing the action with the highest known value. It maximizes short-term gains but lacks exploration.
Value-Based RL - A class of RL algorithms that learn a value function (e.g., Q-function) to determine the best action. Examples: Q-learning, Deep Q-Learning
Epsilon-Greedy Strategy - A policy that chooses the best-known action with probability $1-\epsilon$ and a random action with probability ϵ\epsilon, balancing exploration and exploitation.
Epsilon Decay - A technique that gradually decreases ϵ over time in ϵ-greedy policies, allowing the agent to explore early and exploit more as it learns.
Temperature - A parameter controlling randomness in softmax-based action selection. Higher temperatures increase exploration; lower ones lead to greedier behavior.
Softmax Strategy - An exploration strategy that selects actions probabilistically based on their estimated values using the softmax function. It encourages stochastic but value-guided exploration.

4.4 - Q-Learning

Temporal Difference Error - The difference between the predicted value and the updated estimate after observing a reward and next state.
Learning Rate - A parameter $\alpha \in [0,1]$ that controls how much new information affects existing value estimates. Low values lead to slow learning; high values can cause instability.
One-Step Lookahead - Evaluating the expected return based on the immediate reward and the value of the next state. Used in TD learning and Q-learning.
Bootstrapping - Updating estimates based on other learned estimates rather than waiting for final outcomes. Central to temporal difference methods like Q-learning.
Q-Learning - A value-based, off-policy RL algorithm that learns the optimal action-value function $Q^*(s,a)$ by iteratively updating Q-values using experience and the Bellman optimality equation.
Q-Table - A tabular representation of state-action values in discrete environments. Each entry stores the estimated value of taking an action in a given state.

4.5 - Deep Q-Learning

Q-Network - A neural network used to approximate the Q-function in environments with large or continuous state spaces, replacing the Q-table in deep RL.
Deep-Q Learning - An extension of Q-learning that uses a Q-network to estimate Q-values. Enables RL in high-dimensional spaces (e.g., images, continuous control), popularized by DeepMind's Atari agent.
Experience - A tuple (s,a,r,s′) representing an interaction between the agent and environment, used to update value estimates during training.
Replay Buffer - A memory that stores past experiences. The agent samples from this buffer to break correlations between sequential data and improve learning stability.
Mini-Batch - A small random subset of experiences sampled from the replay buffer during training. Used to compute gradient updates efficiently.
Experience Replay - The technique of reusing past experiences by sampling mini-batches from a replay buffer to improve sample efficiency and reduce variance.
Overestimation Bias - A common issue in Q-learning where the max operator over noisy value estimates leads to systematically overestimated Q-values.
Target Network - A separate copy of the Q-network used to compute stable targets for learning. Updated slowly to reduce feedback loops and stabilize training.
Double Deep Q-Learning - An improvement to DQN that mitigates overestimation bias by decoupling the selection and evaluation of the action in the Q-target.

4.6 - Deep Deterministic Policy Gradient (DDPG)

Continuous Action Space - An action space where actions take values from a continuous range (e.g., real-valued steering or acceleration). DDPG is designed to handle such spaces.
Policy Gradient Method - A class of RL algorithms that directly optimize the policy by computing gradients of expected reward with respect to policy parameters.
Off-Policy - A learning approach where the agent learns from data generated by a different policy than the one it is currently optimizing. DDPG and Q-learning are off-policy.
On-Policy - A learning approach where the agent learns using data generated by the same policy it is updating. Example: REINFORCE, PPO.
Actor-Critic - A framework combining two components: the actor learns the policy (actions), and the critic estimates the value function to guide the actor's updates.
Gaussian Noise - Random noise added to the policy's actions to encourage exploration in continuous spaces. DDPG often uses Gaussian noise during training.
Deep Deterministic Policy Gradient (DDPG) - An off-policy, actor-critic RL algorithm for continuous action spaces. It combines deterministic policy gradients with Q-function learning and experience replay, and uses target networks for stability.

Chapter 5: Task Execution

5.1 - Reactive Control

Reactive Control - A control strategy where the robot responds directly to sensor inputs with predefined behaviors. It is fast and simple, ideal for dynamic environments, but lacks long-term reasoning. Examples include obstacle avoidance via potential fields or behavior-based control.

5.2 - Finite State Machines (FSM)

Task - A high-level objective composed of a sequence of subtasks or actions that a robot must complete to achieve a goal. In execution systems, tasks are often decomposed into manageable states or behaviors.

Task-State - A specific mode of behavior within a task, typically associated with a particular goal or action (e.g., "searching," "grasping," "navigating"). Each task-state has transitions to other states based on sensor inputs or internal conditions.

Finite State Machine - A computational model used to describe task execution as a set of states, with transitions between them triggered by conditions or events. FSMs are widely used in robotics to manage discrete behavior switching in tasks like object manipulation or navigation.

5.3 - Hierarchical Finite State Machines (HFSM)

Hierarchical Finite State Machine - An extension of FSMs where states can themselves contain other FSMs. This allows for modular, scalable task execution by enabling nesting of behaviors. HFSMs manage complex tasks like multi-step manipulation or mission-level planning by organizing control across multiple levels of abstraction.

Section III: Localization

Chapter 6: Sensor Fusion

6.1 - Sensor Values as a Probability Distribution

Sensor - A hardware component that measures a specific property of the environment or robot state (e.g., IMU, LiDAR, camera). Sensors provide raw data used for perception and state estimation.

Observation - A measurement or reading from a sensor at a given time. Observations are typically noisy and used to infer the underlying state.

Observation Space - The set of all possible values a sensor can output. In high-dimensional sensors (e.g., cameras), the observation space may be large and continuous.

Ground Truth Observation - The ideal, noise-free measurement that accurately reflects the real-world property the sensor is attempting to measure. Ground truth is often used for benchmarking and calibration.

Probability Density Function - A function that describes the relative likelihood of a continuous random variable taking on a specific value. PDFs are used to model sensor noise and uncertainty in observations.
Gaussian Distribution - A symmetric, bell-shaped PDF defined by a mean and variance. It is widely used to model sensor noise due to the Central Limit Theorem and its tractable mathematical properties.

6.2 - Sensor Calibration

Expected Value - The average (mean) value of a random variable over many trials. It represents the center of a probability distribution and is often used as a point estimate.
Variance - A measure of how much a random variable deviates from its expected value. In localization, variance represents sensor uncertainty.
Convex - A function or set is convex if a line segment between any two points lies entirely within it. Convexity ensures optimization problems have a single global minimum.
Mean Squared Error - A common loss metric that computes the average of squared differences between estimated values and ground truth.

6.3 - Introduction to Sensor Fusion

Partial Observability - A condition where the robot's sensors do not provide full access to the true state, necessitating inference and estimation.
State - A complete internal representation of the robot and/or environment needed for decision-making, such as position, orientation, and velocity.
State Space - The set of all possible states the robot could be in. State estimation involves narrowing this space using sensor data.
State Estimation Function - A function that uses past observations (and optionally controls) to infer the most likely current state.
Sensor Fusion - The process of combining data from multiple sensors to produce a more accurate and reliable estimate of the state than could be obtained from any single sensor alone.

6.4 - Recursive Estimation

Discrete-Time System - A system where state transitions and observations occur at discrete time steps. Most filtering algorithms are formulated this way.
Markov Property - The assumption that the current state contains all relevant information about the future, given the present. In filtering, this enables recursive updates.
Recursive Estimation/Filtering - An online process of updating the state estimate at each time step using new sensor data and control inputs, without storing the entire observation history.

6.5 - State Space Modeling

Control Inputs - Commands given to the robot (e.g., velocity, acceleration) that influence its next state. These are included in the process model.
State Space Model - A mathematical model that describes how the state evolves over time based on control inputs and noise..
Time-Invariant - A system or model whose parameters do not change with time. Most filters assume time-invariance unless modeling explicitly dynamic systems.
Process Noise - Random variation in the state transition process, due to model inaccuracies or unmodeled dynamics. Represented as a noise term in the process model.
Dynamic Recursive Estimation/Filtering - Recursive estimation that explicitly models state transitions, accounting for motion and process noise over time (e.g., using a Kalman filter).

6.6 - Particle Filter

Particle Filter - A sampling-based recursive estimator that represents the belief over the state using a set of weighted particles. It is suited for nonlinear systems.
Bayes Theorem - A fundamental rule for updating beliefs based on evidence that is used in recursive filtering to update the state estimate using new observations.
Observation Model - Used in recursive filtering to update the state estimate using new observations.
Observation Noise - Random variability in sensor measurements. It is modeled as a distribution and used in the observation model to explain discrepancies.
Observation Noise Covariance Matrix - A matrix describing the variance and correlations of errors in multivariate sensor observations. It quantifies uncertainty in the measurement model.
Innovation - The difference between the actual observation and the predicted observation from the current state estimate. Used to correct predictions in filters.
Particle Degeneracy Problem - A situation where, after several updates, most particle weights become negligible. It reduces the effectiveness of the filter and motivates resampling.
Resampling - A step in particle filtering where new particles are drawn based on their weights to combat degeneracy, concentrating samples on high-probability regions.
Multinomial Resampling - A standard resampling method that samples particles with replacement based on a categorical distribution over their normalized weights.
Systematic Sampling - A more efficient resampling method that ensures low variance by spreading sample selection evenly across the weight distribution.
Process Noise Covariance Matrix - A matrix that quantifies uncertainty in the robot's motion or process model. It is used to perturb particles or states during prediction.

6.7 - Kalman Filtering

6.7.1 - Kalman Filter (KF)

Kalman Gain - A weight that determines how much the estimate is corrected based on the measurement. It balances trust between the prior and the new observation.
Kalman Filter - An optimal recursive estimator for linear systems. It uses a two-step process: prediction based on motion and correction using observations.
Extended Kalman Filter (EKF) - A nonlinear extension of the Kalman Filter that linearizes the process and observation models around the current estimate using Jacobians.

6.7.2 - Unscented Kalman Filter (UKF)

Sigma Point - A set of carefully chosen points that capture the mean and covariance of a distribution. These points are propagated through nonlinear functions in UKF.
Unscented Transform - A deterministic sampling method used to approximate the distribution of a nonlinear transformation of a random variable, without linearization.
Unscented Kalman Filter - A nonlinear filtering algorithm that uses the unscented transform instead of linearization to estimate the state. It provides better accuracy than EKF for highly nonlinear models.

Chapter 7: Simultaneous Localization & Mapping (SLAM)

7.1 - Introduction to SLAM

Dead Reckoning - A method of estimating a robot's current position by integrating past velocity and heading over time. It is simple and fast but accumulates drift due to sensor noise, making it unreliable for long-term localization without correction.
Loop Closure - The process of recognizing that the robot has returned to a previously visited location. Loop closure is critical in SLAM for correcting accumulated drift and realigning the map with the true environment structure.
SLAM - The computational problem of building a map of an unknown environment while simultaneously estimating the robot's location within it. SLAM algorithms fuse sensor data over time to reduce drift and maintain a globally consistent trajectory and map.

7.2 - Iterative Closest Point (ICP)

Reference Point Cloud - The fixed or previously observed 3D point cloud to which another point cloud (the source) is aligned during registration. In SLAM, it typically represents the known map or prior scan.
Source Point Cloud - The new or moving 3D point cloud that is transformed (rotated, translated) to align with the reference point cloud in ICP. It typically comes from the robot's most recent scan or depth measurement.

Levenberg-Marquardt Optimization - A nonlinear least-squares optimization method used in ICP and SLAM to refine transformation estimates. It combines gradient descent and Gauss–Newton steps, making it stable and robust for pose estimation.
Correspondence Search - The process of finding the best matching point in the reference cloud for each point in the source cloud. This step is central to ICP and determines the accuracy of the alignment.
Mutual Closest Point Matching - A stricter form of correspondence search where a match is only valid if each point is the closest to the other in both clouds. This improves robustness by reducing false matches.
Iterative Closest Point - An algorithm for aligning two point clouds by iteratively alternating between finding correspondences and minimizing the distance between matched points through rigid transformation updates. Widely used in LiDAR-based SLAM for scan matching.

Section IV: Computer Vision

Chapter 8: Image Processing

8.1 - Introduction to Computer Vision

Visible Light Spectrum (VLS) - The portion of the electromagnetic spectrum detectable by the human eye, typically spanning wavelengths from ~400 to 700 nm. Most vision sensors, including cameras used in robotics, operate in this range.
Brightness - A measure of the intensity of light in an image. In grayscale images, brightness corresponds to pixel intensity; in color images, it reflects overall luminance.
Color - The perceptual result of light's wavelength composition, typically represented numerically in digital images using models such as RGB or HSV. Color conveys important semantic and structural information in visual processing.
Computer Vision - A field of study focused on enabling machines to interpret, analyze, and extract information from visual inputs such as images or videos. In robotics, computer vision enables perception tasks like localization, object detection, and scene understanding.

8.2 - Cameras

Photodiode - A light-sensitive semiconductor device that converts photons into electrical current. Arrays of photodiodes are used in camera sensors to detect light intensity at each pixel.
Color Filters - Filters placed over photodiodes to restrict the light reaching each sensor to a specific wavelength range (e.g., red, green, or blue). Used to construct full-color images by combining filtered signals.
Trichromatic System - A system (biological or artificial) that perceives color using three types of sensors, each sensitive to a different range of wavelengths. Most cameras and human vision

systems are trichromatic.
RGB Color Model - A color representation model that encodes color as a combination of Red, Green, and Blue components. It is the most common format for digital images and camera output.
Camera - A sensor that captures light and converts it into a digital image. It consists of lenses, a sensor array (e.g., CMOS or CCD), and often image processing hardware.
Pinhole Camera Model - A geometric model that describes how 3D points in the world are projected onto a 2D image plane through a single focal point (the "pinhole"). It is the basis for camera calibration, projection, and image formation equations.

8.3 - Convolutional Neural Networks

8.3.1 - Introduction to Image Classification

Image Classification - The task of assigning a single label to an entire image, identifying what object or scene it contains. It is often the first step in visual recognition pipelines.
Object Classification - A subset of image classification focused specifically on identifying object categories (e.g., dog, car, robot) in an image. Unlike detection, it does not localize objects spatially.

8.3.2 - Convolution Layers

Translational Invariance - A property of CNNs where learned features respond similarly even when the object's position shifts slightly. Achieved through convolution and pooling operations.
Fully Connected (Dense) Layer - A layer in which every neuron is connected to every neuron in the previous layer. Used in traditional neural networks and the final stages of CNNs for classification.
Sparse Layer - A layer where each neuron connects only to a subset of inputs, reducing computation and improving local feature extraction. Convolution layers are sparse by design.
Weight Sharing - A technique in CNNs where the same filter (kernel) is applied across the entire image, drastically reducing the number of learnable parameters and enabling spatial invariance.
Convolution Kernel - A small matrix of learnable weights used to extract features from local patches of the image via element-wise multiplication and summation.
Stride - The number of pixels by which the convolution kernel moves across the input image. Larger strides reduce the output size and increase downsampling.
Feature Map - The output of applying a convolution kernel to an image. It highlights regions of the image where the kernel detects specific features like edges or textures.
Convolution Layer - A neural network layer that applies multiple learnable convolution kernels across the input to extract spatial features. It preserves locality and enables hierarchical feature

learning.
Receptive Field - The region of the input image that affects the output of a given neuron. In CNNs, deeper neurons have larger receptive fields, allowing them to capture more global information.
Convolutional Neural Network (CNN) - A deep learning architecture designed to process grid-like data (e.g., images) by stacking convolution, pooling, and dense layers. CNNs are the foundation of modern visual recognition systems.

8.3.3 - Pooling

Downsampling - The process of reducing the spatial resolution of an image or feature map. In CNNs, downsampling helps reduce computational cost and introduces translation invariance.
Pooling - A layer in CNNs that aggregates features over local regions, typically to reduce dimensionality and focus on salient patterns. Pooling operates on small patches and replaces them with summary statistics.
Average Pooling - A pooling method that replaces a region with the average of its values. It smooths feature maps and retains broader contextual information.
Max Pooling - A pooling method that replaces a region with its maximum value, emphasizing the most prominent features and aiding in translation invariance.

Chapter 9: Object Detection

9.1 - Introduction to Object Detection

Object Classification - The task of assigning a class label (e.g., "car," "dog") to an entire image or region. It answers the question: "What is it?"
Object Localization - The task of determining the spatial location of an object in an image, typically represented with a bounding box. It answers: "Where is it?"
Object Detection - A computer vision task that combines classification and localization, identifying what objects are present in an image and where they are, typically using bounding boxes.

9.2 - Sliding Window Approach

Window - A rectangular region of the image selected for analysis. In the sliding window approach, this window is moved across the image to detect objects in different locations.
Stride - The number of pixels the window shifts at each step as it slides across the image. Smaller strides improve accuracy but increase computation.
Confidence Level - A score indicating how certain the model is that a detected region contains an object. Higher confidence suggests a stronger belief in the presence of an object.
Objectness - The model's confidence that any object (of any class) exists in a region, regardless

of class. Used to differentiate between object and background.
Conditional Class Probabilities - The probability distribution over object classes given that an object is present in the region. These are typically predicted after objectness is determined.
Bounding Boxes - Rectangular boxes that surround detected objects, typically defined by their center coordinates, width, and height. They are the main output of detection systems.
Sliding Window Approach - A traditional object detection method where a window is moved across the image at multiple locations and scales, and each region is classified independently. It is accurate but computationally expensive.
Non-Maximum Suppression - A post-processing step that eliminates redundant bounding boxes by keeping only the one with the highest confidence and removing overlapping ones based on an IoU threshold.
Intersection Over Union (IoU) - A metric that measures the overlap between two bounding boxes. It is used for evaluating detection quality and in NMS.
Multiscale Sliding Window Approach - An extension of the sliding window method where the image is resized or the window is scaled to detect objects of varying sizes.

9.3 - Evaluating Object Detection Output

Ground Truth Bounding Boxes - The manually annotated boxes indicating the true locations of objects in the image. Used as reference for evaluating predictions.
Loss Function - A mathematical function that quantifies the error between the predicted output and ground truth. It guides learning by penalizing incorrect predictions.
One-to-One Matching Strategy - A matching strategy where each predicted bounding box is matched to at most one ground truth box (and vice versa), used during training and evaluation.
Greedy Algorithm - An algorithm that makes locally optimal choices at each step. In object detection, greedy NMS selects boxes with the highest confidence and removes overlapping ones.
Objectness Loss - Part of the overall detection loss function that penalizes incorrect predictions about whether an object exists in a region.
Localization Loss - Penalizes differences between predicted and ground truth bounding box coordinates. Often implemented using MSE or smooth L1 loss.
Classification Loss - Penalizes incorrect class predictions for detected objects. Typically implemented using cross-entropy loss.
No-Object Loss - Penalizes false positives—when the model predicts objects in regions where no object exists.
Object Loss - The loss component applied to regions where an object is present, often combining localization and classification loss.

9.4 - YOLO

2-Stage Detection Algorithm - An object detection framework that first generates region proposals (Stage 1), then classifies and refines them (Stage 2). Examples: Faster R-CNN.
Single Forward Pass - A processing approach where the model performs detection in one pass through the network, rather than in stages. YOLO and SSD use this method for real-time performance.
The Responsibility - In YOLO, each grid cell is responsible for predicting objects whose center falls within that cell. This helps assign training targets and structure the output.
Occlusion - A condition where part of an object is blocked by another object or structure. Detection under occlusion requires robust features and spatial reasoning.
YOLO - A single-shot object detection algorithm that divides the image into a grid and simultaneously predicts bounding boxes and class probabilities for each cell. YOLO is fast, fully convolutional, and trained end-to-end.

Chapter 10: 3D Vision

10.1 - Introduction to Depth Perception

LiDAR - A sensor that emits laser pulses and measures the time they take to return after reflecting off surfaces. LiDAR produces accurate 3D information about the environment, often in the form of a point cloud.
Point Cloud - A collection of 3D points, typically in (x,y,z) format, representing the surfaces of objects in space. Point clouds are generated by LiDAR, depth cameras, or stereo vision systems.
Stereo Vision - A technique that infers depth by comparing two images taken from slightly different viewpoints. It mimics human binocular vision to estimate disparity, which can be converted into depth.

10.2 - Camera Calibration

Camera Extrinsic Matrix - A transformation matrix that encodes the position and orientation of the camera in the world coordinate system. It maps 3D points from world space to the camera's coordinate frame.
Principal Point - The point on the image plane where the camera's optical axis intersects. It is usually close to the center of the image and is part of the intrinsic matrix.
Focal Length - The distance from the camera's center of projection (lens) to the image plane. It controls magnification and field of view and is a key component of the intrinsic parameters.
Camera Intrinsic Matrix - A matrix that encapsulates internal camera parameters like focal length, principal point, and skew, used to map 3D camera coordinates to 2D pixel coordinates.
Skew - A parameter representing the angle between the image axes. In most modern cameras, the skew is zero, meaning the axes are perpendicular.

10.3 - Stereo Vision

10.3.1 - Horizontal Stereo

Stereo Matching - The process of finding corresponding pixels between two rectified stereo images. The pixel displacement (disparity) is used to compute depth via triangulation.
Triangulation - A geometric method for calculating 3D coordinates of a point by intersecting lines of sight from two known camera positions. Used in stereo vision to derive depth from disparity.
Disparity - The difference in the horizontal position of corresponding points in the left and right stereo images. Larger disparities correspond to closer objects.
Horizontal Stereo - A stereo vision configuration where two cameras are aligned horizontally and face forward. This setup enables efficient disparity computation due to dominant horizontal shifts between views.
Horizontal Rectification - The process of transforming stereo image pairs so that corresponding points lie on the same horizontal line, simplifying disparity computation and matching.

10.3.2 - Block Matching

Photometric (Data) Cost - A measure of similarity between pixel patches in the left and right stereo images. Common metrics include absolute difference or squared difference.
Block Matching - A stereo matching method that compares small blocks (windows) of pixels between images to find correspondences. It is simple but can struggle with textureless regions and depth discontinuities.

10.3.3 - Semi-Global Matching

Truncated Linear Penalty - A penalty function used in SGM to discourage disparity changes between neighboring pixels, while allowing discontinuities. It truncates the penalty beyond a certain disparity difference to preserve edges.
Scanline - A row (or column) of pixels across the image. In SGM, the cost aggregation is performed along multiple scanline directions to approximate global consistency.
Semi-Global Matching (SGM) - A stereo matching algorithm that computes disparity by minimizing a global energy function, approximated by aggregating costs along multiple scanlines. It balances accuracy and efficiency and is widely used in robotics.

10.3.4 - ML Approaches

StereoNet - A deep learning-based stereo vision model that estimates dense disparity maps directly from rectified stereo pairs. It replaces hand-engineered cost functions and aggregation steps with learned representations and end-to-end training.

Section V: Multiple Robots

Chapter 11: Multi-Robot Systems

11.1 - Introduction to Multi-Robot Systems

Multi-Robot System (MRS) - A group of robots that operate within the same environment and may collaborate to achieve shared or individual goals. MRS can involve coordination, communication, and distributed decision-making.

Homogeneous - A multi-robot system in which all robots are identical in hardware, sensing, and capabilities. Homogeneous MRSs simplify coordination and allow task redundancy.

Heterogeneous - A system composed of robots with different capabilities, sensors, or roles. Heterogeneous MRSs are more flexible and capable but require more sophisticated task allocation and coordination strategies.

11.2 - Multi-Robot Coordination

11.2.1 - Introduction to Multi-Robot Coordination

Multi-Robot Coordination - The process of synchronizing or organizing the behavior of multiple robots to achieve shared goals or avoid conflicts. This includes formation control, path planning, and resource distribution.

11.2.2 - Decentralized Control

Decentralized Decision-Making - A control paradigm where each robot makes decisions independently, often based on local information and peer communication. It enhances scalability and robustness in dynamic environments.

Centralized Training with Decentralized Execution (CTDE) - A learning framework in multi-agent reinforcement learning (MARL) where agents are trained together with access to global information but operate independently during deployment.

Multi-Agent Reinforcement Learning (MARL) - A class of RL algorithms designed for environments with multiple agents (robots). Each agent learns to optimize its behavior while accounting for interactions with others.

11.2.3 - Centralized Control

Controller - A system or algorithm that determines the actions of robots based on goals, sensor inputs, or learned policies. In centralized control, the controller governs all robots collectively.

Centralized Decision Making - A model in which a single controller or node plans and assigns actions to all robots. It can achieve optimal coordination but requires reliable communication and high computation.

11.3 - Task Allocation

Task - A unit of work or objective assigned to a robot. Tasks can vary in complexity and may be spatial (e.g., go to location) or functional (e.g., transport object).
Indivisible Task - A task that must be completed by a single robot; it cannot be split among multiple agents (e.g., flipping a heavy switch that requires one robot's full capacity).
Divisible Task - A task that can be partitioned and completed collaboratively by multiple robots (e.g., mapping, surveillance, or parallel data collection).
Task Allocation - The process of assigning tasks to robots based on capability, availability, and cost. Good task allocation maximizes overall efficiency and utility.
Robot-Task Pair - A candidate assignment that links a specific robot with a specific task. Used in evaluating and ranking task allocation strategies.
Overall Utility - The total benefit or efficiency gained by the entire multi-robot system from a particular task allocation. Often the optimization goal in centralized planning
Individual Utility - The benefit or cost incurred by a single robot from being assigned a specific task. Used in distributed planning and negotiation
Utility Function - A mathematical function that evaluates how beneficial a robot-task assignment is, based on metrics like energy, time, or skill fit.
Market-Based Method - A decentralized task allocation strategy where robots "bid" for tasks based on utility. The task is assigned to the highest bidder, mimicking economic markets
Centralized Allocation - A method in which a central node assigns tasks to robots based on global knowledge and optimization criteria.
Decentralized Allocation - A distributed strategy where robots negotiate or select tasks autonomously, often using local information and peer-to-peer protocols
Consensus Algorithms - Protocols that allow multiple robots to agree on shared decisions (e.g., task assignment or motion plans) through iterative communication. Common in decentralized coordination.
Static Task Allocation - Protocols that allow multiple robots to agree on shared decisions (e.g., task assignment or motion plans) through iterative communication. Common in decentralized coordination.
Dynamic Task Allocation - Tasks are continuously reassigned based on real-time information, allowing the system to adapt to failures, delays, or environmental changes

11.4 - Multi-Robot Communication

11.4.1 - Introduction to Multi-Robot Communication

Multi-Robot Communication - The exchange of information between robots to support coordination, planning, or learning. Communication can be direct (explicit messages) or indirect (environmental cues).

Decentralized Communication - Communication that occurs peer-to-peer, without a central server or hub. It enhances robustness and scalability but can suffer from inconsistency or delays.
Centralized Communication - Communication through a central coordinator or server. This ensures data consistency and global awareness but creates a single point of failure.

11.4.2 - Direct vs. Indirect Communication

Direct Communication - Explicit data exchange between robots via wireless protocols such as Wi-Fi or Bluetooth. It allows precise coordination but depends on connectivity.
Indirect Communication - Also known as stigmergy, where robots communicate by modifying or sensing the environment (e.g., leaving markers or shared maps). It enables coordination with minimal messaging.

References

Bellman, R. (1957). Dynamic programming. Princeton, NJ: Princeton University Press. Retrieved from https://gwern.net/doc/statistics/decision/1957-bellman-dynamicprogramming.pdf

Boltzmann, L. (1868). Studien über das Gleichgewicht der lebendigen Kraft zwischen bewegten materiellen Punkten [Studies on the equilibrium of kinetic energy among moving material points]. Wiener Berichte, 58, 257–274.

Brooks, R. (1986). A robust layered control system for a mobile robot. IEEE Journal on Robotics and Automation, 2(1), 14–23. https://doi.org/10.1109/JRA.1986.1087032

Cauchy, A. L. (1847). Méthode générale pour la résolution des systèmes d'équations simultanées. Comptes Rendus de l'Académie des Sciences.

Chen, Y., & Medioni, G. (1991, April 1). Object modeling by registration of multiple range images. IEEE Xplore. https://doi.org/10.1109/ROBOT.1991.132043

Cybenko, G. (1989). Mathematics of Control, Signals, and Systems Approximation by Superpositions of a Sigmoidal Function*. Math. Control Signals Systems, 2, 303–314. https://web.njit.edu/~usman/courses/cs677/10.1.1.441.7873.pdf

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1(1), 269–271. https://doi.org/10.1007/BF01386390

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics, 36(4), 193–202.

Gauss, C. F. (1809). Theoria motus corporum coelestium in sectionibus conicis solem ambientium

Gordon, N. J., Salmond, D. J., & Smith, A. F. M. (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation. IEE Proceedings F Radar and Signal Processing, 140(2), 107. https://doi.org/10.1049/ip-f-2.1993.0015

Harel, D. (1987). Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8(3), 231–274. https://doi.org/10.1016/0167-6423(87)90035-9

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100–107. IEEE Transactions on Systems Science and Cybernetics. https://doi.org/10.1109/TSSC.1968.300136

Hasselt, H.V., Guez, A., & Silver, D. (2015). Deep Reinforcement Learning with Double Q-Learning. AAAI Conference on Artificial Intelligence. Retrieved from https://www.semanticscholar.org/paper/Deep-Reinforcement-Learning-with-Double-Q-Learning-Hasselt-Guez/3b9732bb07dc99bde5e1f9f75251c6ea5039373e

Hirschmuller, H. (2005). Accurate and efficient stereo processing by semi-global matching and

mutual information. IEEE Xplore. https://doi.org/10.1109/CVPR.2005.56

Hussein, A., & Khamis, A. (2013). Market-based approach to multi-robot task allocation. In Proceedings of the 2013 International Conference on Individual and Collective Behaviors in Robotics (ICBR) (pp. 69–74). IEEE. https://doi.org/10.1109/ICBR.2013.6729278

Ivakhnenko, A. G., & Lapa, V. G. (1967). Cybernetics and forecasting techniques. American Elsevier.

Julier, S. J., Uhlmann, J. K., & Durrant-Whyte, H. F. (1995, June 1). A new approach for filtering nonlinear systems. IEEE Xplore. https://doi.org/10.1109/ACC.1995.529783

Kalman, R. E. (1960). A New Approach to Linear Filtering and Prediction Problems. Journal of Basic Engineering, 82(1), 35–45. https://doi.org/10.1115/1.3662552

Kavraki, L. E., Svestka, P., Latombe, J.-C., & Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Transactions on Robotics and Automation, 12(4), 566–580. IEEE Transactions on Robotics and Automation. https://doi.org/10.1109/70.508439

Khamis, S., Fanello, S., Rhemann, C., Kowdle, A., Valentin, J., & Izadi, S. (2018). StereoNet: Guided Hierarchical Refinement for Real-Time Edge-Aware Depth Prediction. Proceedings of the European Conference on Computer Vision (ECCV), 8–14.

Khatib, O. (1986). Real-Time Obstacle Avoidance for Manipulators and Mobile Robots. The International Journal of Robotics Research, 5(1), 90–98. https://doi.org/10.1177/027836498600500106

LaValle, S. (1998). Rapidly-exploring random trees: A new tool for path planning. The Annual Research Report. https://www.semanticscholar.org/paper/Rapidly-exploring-random-trees-%3A-a-new-tool-for-LaValle/d967d9550f831a8b3f5cb00f8835a4c866da60ad

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation Applied to Handwritten Zip Code Recognition. Neural Computation, 1(4), 541–551. https://doi.org/10.1162/neco.1989.1.4.541

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Silver, D. (2015). Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971.

Lucas, B., & Kanade, T. (1981). An Iterative Image Registration Technique with an Application to Stereo Vision. Retrieved April 6, 2025, from https://www.ijcai.org/Proceedings/81-2/Papers/017.pdf

Markov, A. A. (1906). Extension of the limit theorems of probability theory to a sum of variables connected in a chain. Mathematics of the Imperial Academy of Sciences of St. Petersburg, 32, 1–20. Retrieved from https://www.scirp.org/reference/referencespapers?referenceid=1744550

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous

activity. The Bulletin of Mathematical Biophysics, 5(4), 115–133. https://doi.org/10.1007/BF02478259

Mealy, G. H. (1955). A method for synthesizing sequential circuits. The Bell System Technical Journal, 34(5), 1045–1079. https://doi.org/10.1002/j.1538-7305.1955.tb03788.x

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., et al. (2013). Playing Atari with deep reinforcement learning. arXiv[Preprint].arXiv:1312.5602

Moravec, H., & Elfes, A. (1985). High resolution maps from wide angle sonar. Proceedings. 1985 IEEE International Conference on Robotics and Automation, 2, 116–121. 1985 IEEE International Conference on Robotics and Automation. https://doi.org/10.1109/ROBOT.1985.1087316

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. arXiv preprint arXiv:1506.02640. Retrieved from https://arxiv.org/pdf/1506.02640

Rowley, H. A., Baluja, S., & Kanade, T. (1998). Neural network-based face detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, 20(1), 23–38. https://doi.org/10.1109/34.655647

Shannon, C. E. (1948). A mathematical theory of communication. Bell System Technical Journal, 27(3), 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

Smith, R. C., & Cheeseman, P. (1986). On the Representation and Estimation of Spatial Uncertainty. The International Journal of Robotics Research, 5(4), 56–68. https://doi.org/10.1177/027836498600500404

Stentz, A. (1994). The D* Algorithm for Real-Time Planning of Optimal Traverses. https://www.semanticscholar.org/paper/The-D*-Algorithm-for-Real-Time-Planning-of-Optimal-Stentz/8e60573c99d2d290dd9163272ea21727c382d00b

Stone, M. H. (1948). The generalized Weierstrass approximation theorem. Mathematics Magazine, 21(5), 237–254. https://doi.org/10.2307/2687506

Sutton, R. S. (1995). TD models: Temporal difference learning with function approximation. In Advances in Neural Information Processing Systems (Vol. 7, pp. 1058–1064). MIT Press. Retrieved from https://papers.nips.cc/paper_files/paper/1995/hash/8f1d43620bc6bb580df6e80b0dc05c48-Abstract.html

Valiant, L. G. (1984). A theory of the learnable. Communications of the ACM, 27(11), 1134–1142. https://doi.org/10.1145/1968.1972

Vidyasagar, M. (1985). Control system synthesis: A factorization approach. MIT Press.

Watkins, C. J. C. H. (1989). Learning from delayed rewards. PhD thesis, University of Cambridge. Retrieved from https://www.researchgate.net/publication/33784417_Learning_From_Delayed_Rewards

Wiener, N. (1948). Cybernetics: Control and communication in the animal and the machine. MIT Press. Retrieved from https://mitpress.mit.edu/9780262537841/cybernetics-or-control-and-communication-in-the-animal-and-the-machine/