CS2109S
Introduction to Artificial Intelligence and Machine Learning

AY2021/22 Semester 2

Notes by Jonathan Tay

Last updated on February 23, 2022

Contents

1 Intelligent Agents

1.1 Agents

An **agent** is anything that can perceive its **environment** through **sensors** and acting upon that environment with **actuators**.

For example, humans have eyes and other organs for sensors, as well as hands, legs, and various other body parts for actuators.

Percepts are the content that an agent's sensors perceive, and the complete history of percepts is known as the **percept sequence**.

In general, an agent's choice of action at any given instant depends on its built-in knowledge and on the entire percept sequence, but not anything it has yet to perceive.

1.2 Rationality

In order for an agent to know to do the right thing (through its actions), we measure the the **desirability** for the outcome sequence of states in the environment.

This measure is known as the **performance measure**, and it encompasses several metrics:

1. Whom is it best for?
2. What are we optimizing for?
3. What information is available?
4. What are the unintended effects?
5. What are the costs?

As a general rule, it is better to design performance measures according to what one actually wants to be achieved in the environment rather than according to how one thinks the agent should behave.

For each possible percept sequence, a **rational agent** will choose the action that is expected to **maximize the performance measure**, given the evidence provided by the percept sequence and its built-in knowledge.

1.3 Environments

Task environments are the "problems" to which rational agents are "solutions". We use the **P.E.A.S.** framework to model them as follows (with examples for autonomous driving):

- **Performance measure** (safety, speed, comfort)
- **Environment** (road, weather, traffic)
- **Actuators** (steering, accelerator, brakes, signals)
- **Sensors** (cameras, LIDAR, GPS, speedometer)

1.3.1 Fully observable vs. partially observable

A **fully observable** environment is one in which an agent's sensors give it access to its **complete state** at each point in time.

Environments may be **partially observable** because of noisy or inaccurate sensors - or essentially if any sensor data is missing.

If the agent lacks sensors, then the environment is **unobservable**.

1.3.2 Single-agent vs. multi-agent

An agent in a **single-agent** environment operates by itself and does not consider other objects as agents.

Multi-agent environments may be **competitive** or **co-operative**.

1.3.3 Deterministic vs. stochastic

A **deterministic** environment allows its next state to be completely determined by the current state and the action executed by the agent. Otherwise, it is **stochastic**.

In multi-agent environments, if the environment is deterministic except for the actions of other agents, then it is **strategic**.

1.3.4 Episodes vs. sequential

In an **episodic** environment, the agent's experience is divided into atomic episodes, during which the agent receives a percept and performs a single action.

The choice of action in each episode depends only on the episode itself.

Otherwise, the environment is **sequential**.

1.3.5 Dynamic vs. static

If the environment can change while an agent is deliberating, it is **dynamic**. Otherwise, it is **static**.

In a dynamic environment, when an agent has not decided on its action when asked, it counts as deciding to do nothing.

If the environment itself does not change with time but the agent's performance score does, the environment is **semi-dynamic** — an example of which is chess played with a clock.

1.3.6 Discrete vs. continuous

A **discrete** environment has a finite number of distinct and clearly-defined percepts and actions, while **continuous environments** do not.

1.4 Agent Programs

An agent's behavior is described by the **agent function** $f : P* \to A$ that maps from percept histories to actions.

This agent function is implemented internally in an artificial agent by an **agent program** which runs on physical **agent architecture** to produce f.

Summarily, agent = architecture + program, and the goal is to implement the rational agent function concisely.

1.4.1 Table-Lookup Agents

A **table-lookup agent** keeps track of the percept sequence, using it to index into a table of actions to decide what to do.

However, drawbacks include exponential space complexity, lack of autonomy, and the duration it takes to compute table entries.

Nonetheless, table-lookup agents does do what we want, assuming the table is filled in correctly, since it implements the rational agent function.

1.4.2 Simple Reflex Agents

The simplest kind of agents, **simple reflex agents** rely on their sensors to perceive their environment in its current state, ignoring their percept history. Then, **condition-action rules** govern the action which they perform.

However, simple reflex agents perform best in fully observable environments. They are vulnerable to infinite loops when their environment is only partially observable.

1.4.3 Model-based Reflex Agents

The most effective way for an agent to handle partial observability is to keep track of the part of the environment it cannot currently see.

The information in this internal state has to be updated over time.

First, a **transition model** is needed for the agent to understand the effects of its actions, as well as how the world evolves independently of the agent.

Second, a **sensor model** represents the knowledge of how the state of the world is reflected in the agent's percepts.

Together, the use of such models enables **model-based reflex agents**.

1.4.4 Goal-based Agents

Knowing the current state of the environment is not always sufficient for a decision.

Agents may require **goal information** that describes desirable outcomes, such that the agent can choose the actions to achieve the goal.

1.4.5 Utility-based Agents

Goals alone are usually insufficient to generate high-quality behavior in most environments, since goals dichotomize states into favorable and disfavorable states.

Rather, the use of **utility functions** by agents allows the **internalization of the performance measure**.

The utility function measures the agent's preferences among states of the world, allowing the agent to choose the action that leads to the best **expected utility**.

1.4.6 Learning Agents

Learning allows agents to operate in initially unknown environments and to become more competent that allowed by its initial knowledge alone.

A learning agent has four conceptual components:

1. The **learning element**, which makes improvements.
2. The **performance element**, which selects external actions.
3. The **critic**, which provides feedback on how the agent is doing with respect to a fixed performance standard. It also determines how the performance element should be modified to do better in future.
4. The **problem generator**, which suggests actions for new and informative experiences, especially through **exploration**.

1.5 Exploitation vs Exploration

Agents in the real world must often choose between maximizing expected utility with its current knowledge of the world (exploitation) and learning more about the world (exploration).

The performance element of a learning agent would prefer to keep doing actions that are best, given what it knows.

However, if the agent is willing to explore a little and perhaps do some sub-optimal actions in the short run, it might discover much better actions for the long run.

The problem generator's role is also to suggest these exploratory actions.

2 Uninformed Search

2.1 Problem-Solving Agents

Agents may need to consider a **sequence of actions** that form a path to a goal state when the correct action is not immediately obvious.

In order to do so, agents follow a four-phase problem-solving process:

1. **Goal formulation**: Goals limit objectives and hence the actions that need to be considered.
2. **Problem formulation**: Describe the states and actions necessary to reach the goal.
3. **Search**: Derive a **solution**, which is a sequence of actions from the initial state that reaches a goal state. There may be multiple or no solutions.

4. **Execute**: Perform the actions in the solution.

There are many different search problem types depending on the environment.

2.1.1 Single-state Problem

A deterministic, fully observable environment creates a **single-state problem** where the agent knows exactly where it will be.

2.1.2 Sensorless Problem

A non-observable environment creates a **sensorless / conformant problem** where the agent may have no idea where it is.

2.1.3 Contigency Problem

A non-deterministic, and/or partially observable environment creates a **contingency problem** where percepts provide new information about the current state.

Search and execution are often interleaved.

2.1.4 Exploration Problem

In unknown environment, the agent may need to explore the environment to find the goal.

2.2 Single-state Problem Formulation

The **state space** is the set of all possible states the environment can exist in.

Formally, a search problem is defined as follows:

1. The **initial state** is the state the agent starts in.
2. The **actions** available to the agent. Given a state x, the successor function $S(x)$ returns a set of **action-state pairs**.
3. The **goal test**, which is defined **explicitly** as a condition, or **implicitly** as a function which takes a state. *For example,* checkmate(x) *is defined implicitly.*
4. The **path cost function** $c(x, a, x')$ which gives the additive cost of performing action a in state x to reach state x'.

Often, we will need to abstract state spaces since the real world environment is absurdly complex, with a state space that is too huge.

2.3 Search Algorithms

A **search algorithm** takes a search problem as input and returns a solution or an indication of failure.

In essence, a tree search simulates the exploration of the state space by generating successors of already-explored states, starting from the initial state.

A **state** is a representation of a physical configuation of the environment.

A **node** in the search tree corresponds to a state in the state space. The **edges** of the search tree correspond to actions.

As a data structure, nodes include state, parent nodes, actions, path cost, and depth.

The *expand* function creates new nodes using the successor function to create the corresponding states.

2.3.1 Measuring Performance

The order of node expansion defines the **search strategy** used by a search algorithm. However, we first need criteria to evaluate them, and we can do so along four dimensions:

1. **Completeness**: Is it guaranteed to find a solution if one exists? A complete algorithm must be able to explore every state reachable from the initial state.
2. **Time complexity**: How many nodes does it generate?
3. **Space complexity**: How much memory does it need to store the nodes at maximum?
4. **Cost optimality**: Does it always find a least-cost solution among all solutions?

2.3.2 Time and Space Complexity

In a state-space graph, theoretical algorithm analysis suggests complexity as measured based on $|V|+|E|$, where $|V|$ is the number of vertices (state nodes) and $|E|$ is the number of edges (state-action pairs).

However, in many AI problems, the graph is represented implicitly by the initial state and actions.

Therefore, complexity here is measured in terms of:

- b, the **branching factor** of the search tree,
- d, the **depth** of the **least-cost solution**, and
- m, the **maximum depth** of the search tree (number of actions along any path)

2.4 Breadth-first Search (BFS)

In **breadth-first search**, the root node is expand first, followed by all its successors, and so on.

Early goal tests offer a slight optimization, by performing goal tests on nodes when they are generated rather than popped (**late goal test**).

- **Completeness**: Yes, if b is finite.
- **Time complexity**: $O(b^{d+1}) = 1 + b + b^2 + b^3 + ... + b^d$
- **Space complexity**: $O(b^{d+1})$ since all nodes are stored in memory.
- **Cost optimality**: Yes, if all actions have the same cost.

The space complexity of breadth-first search is a bigger problem than its time complexity.

In general, exponential-complexity search problems cannot be solved by uninformed search for any but the smallest instances.

2.5 Uniform-cost Search (UCS)

Also known as **Dijkstra's algorithm**, **uniform-cost search** always expands the least-cost unexpanded node.

It is equivalent to breadth-first search if all action costs are equal. Most implementation use a **priority queue** ordered by path cost.

Let $C*$ be the cost of the optimal solution, and ϵ be the lower bound on each action cost where $\epsilon > 0$.

- **Completeness**: Yes, if all actions cost $\geq \epsilon > 0$.
- **Time complexity**: $O(b^{\lceil C*/\epsilon \rceil})$
- **Space complexity**: $O(b^{\lceil C*/\epsilon \rceil})$
- **Cost optimality**: Yes, since nodes are expanded in order of increasing cost.

$O(b^{\lceil C*/\epsilon \rceil})$ can be worse than $O(b^d)$ since uniform-cost search can explore large trees of actions with low costs before exploring paths with a high-cost but with useful action.

2.6 Depth-first Search (DFS)

In **depth-first search**, the deepest node is always expanded first.

- **Completeness**: Depends. Incomplete if it fails in loops (for some implementations) or infinite-depth spaces. Complete in finite spaces.
- **Time complexity**: $O(b^m)$
- **Space complexity**: $O(m)$ with backtracking, $O(bm)$ without.
- **Cost optimality**: No, since it always returns the first solution it finds.

The time complexity of $O(bm)$ can be terrible if m is much larger than d. However, DFS can be faster than BFS if the goals are dense.

2.7 Depth-limited Search (DLS)

Depth-limited search is a variant of **depth-first search** which limits the depth of the search tree to l, such that all nodes at depth l are treated as if they had no successors.

This prevents DFS from wandering down an infinite path.

- **Completeness**: No, if a poor choice of l is made it will fail to arrive at a solution.
- **Time complexity**: $O(b^l)$
- **Space complexity**: $O(bl)$
- **Cost optimality**: No, since it always returns the first solution it finds.

2.8 Iterative Deepening Search (IDS)

Rather than having to choose a single value for l like in DLS, **iterative deepening search** tries every possible value for l incrementally from zero.

There is an overhead cost to doing so, but most of the nodes are in the bottom level, so repetition of the top nodes is not entirely significant.

For example, for $b = 10$ and $d = 5$, DLS explores $1 + 10 + 100 + 1000 + 10000 = 11111$ nodes, while IDS explores $6+50+400+3,000+20,000+100,000 = 123,456$ nodes.

The overhead is a mere 11%.

- **Completeness**: Yes.
- **Time complexity**: $O(b^d) = (d+1)b^0 + db^1 + (d-1)b^2 + ... + b^d$
- **Space complexity**: $O(bd)$
- **Cost optimality**: Yes, if the action costs are equal.

2.9 Bidirectional Search

This alternative approach searches **simultaneously** from the initial state and backwards from the goal states, checking if a node appears in the other search tree.

Intuitively, $2 * O(b^{\frac{d}{2}})$ is less than $O(b^d)$.

Different search strategies can be employed for either half. However, pred(succ(n)) and succ(pred(n)) must be equal.

As an aside, memoization should be used to handle repeated states in search algorithms.

3 Informed Search

Informed search involves the use an **evaluation function** $f(n)$ which estimates desirability for every node in the search tree.

These search algorithms can therefore **find solutions more efficiently** by expanding the most desirable unexpanded nodes first.

A **heuristic function** $h(n)$ estimates the cost from state n to a goal state. Heuristics form a component of the evaluation function.

3.1 Greedy Best-first Search

Greedy best first search is a special case of best-first search which always **expands the node that appears closest to the goal** — the node with the lowest $h(n)$ value.

This algorithm is usually implemented with a **priority queue**, with nodes in the frontier **ordered in decreasing order of desirability**. The evaluation function $f(n) = h(n)$.

Greedy best-first search is **not complete**, as it can get stuck in loops.

It is **not optimal** either, since it tries to get as close to a goal as possible at each iteration, but this greediness can

lead to a solution with a higher cost. *In general, greedy algorithms are not optimal.*

It has a **time complexity of $O(b^m)$**, but a good heuristic function can reduce this significantly to $O(bm)$.

Likewise, it has a **space complexity of $O(b^m)$** since it needs to store every node in memory.

3.2 A* Search

A* search builds on best-first search by **avoiding the expansion of expensive nodes**, using an evaluation function $f(n) = g(n) + h(n)$.

$f(n)$ estimates the total cost of the path from n to a goal. $g(n)$ is the total cost to reach n from the initial state, while $h(n)$ **estimates** the cost from n to a goal.

A* search is **complete** as long as the state space is finite.

It is **optimal** as long as its heuristic is **admissible**. *An admissible heuristic never overestimates the cost to reach a goal, and is therefore optimistic.*

Formally, a heuristic is admissible if $\forall n(h(n) \leq h^*(n))$, where $h^*(n)$ is the true cost to reach a goal state from node n.

If $h(n)$ is admissible, `TREE-SEARCH` is optimal.

Furthermore, a heuristic is consistent if, for every node n and every successor n' generated by an action a, $h(n) \leq \text{cost}(n, a, n') + h(n')$.

If $h(n)$ is consistent, `GRAPH-SEARCH` is optimal.

Every consistent heuristic is admissible, but not vice versa. Therefore, with a consistent heuristic, A* search is optimal as well.

A* search has a **time complexity of $O(b^m)$**. However, A* search can **prune** away nodes not required for finding an optimal solution, *such that it is much more efficient than uninformed searches despite its exponential complexity.*

It also has a **space complexity of $O(b^m)$**, since every node needs to be stored in memory.

3.3 Memory-bounded Heuristic Search

With memory being the main issue with A* search, other algorithms exist with tricks to reduce the space complexity.

3.3.1 Iterative-deepening A* Search

IDA* search is to A* what iterative-deepening search is to depth-first search.

Instead of setting the cutoff based on depth, IDA* uses the f-cost of $g(n) + h(n)$, incrementing the cutoff to the best f-cost which exceeded the cutoff on the previous iteration.

3.3.2 Recursive Best-first Search

RBFS resembles the typical best-first search, but keeps track of the best alternative f-value before exploring the current node.

If the current node exceeds this limit, it recursively backtracks to explore the alternative path. While backtracking, it replaces the f-value of nodes along the way with the best f-value of that node's children.

3.3.3 Simplified Memory-bounded A* Search

A* search uses too much memory, while both IDA* and RBFS use too little memory — IDA* keeps only the f-cost limit, while RBFS uses linear space, which under-utilizes memory even if more is available.

SMA* improves on this by expanding the best node until all memory is used up.

Then, it drops the node with the highest f-value, backtracking and replacing that node's parent's f-value with the dropped node's f-value before continuing.

This algorithm is not perfect, however. On very hard problems, SMA* can switch back and forth between alternative paths, each time running out of memory and repeatedly regenerating the same nodes.

3.4 Formulating Heuristics

A search algorithm using a more efficient heuristic will never expand more nodes than one with a less efficient heuristic.

Given two admissible heuristics, $h_1(n)$ and $h_2(n)$, h_2 **dominates** h_1, if for all nodes n, $h_2(n) \geq h_1(n)$.

Therefore, it is better to use a heuristic with higher $h(n)$ values, since it will expand fewer nodes as long as it is consistent.

A **relaxed problem** is a modified problem with fewer restrictions placed on its actions. Any solution for the orgiginal problem also solves the relaxed problem.

The cost of an optimal solution to a relaxed problem is an admissible and consistent heuristic for the original problem.

3.5 Using Inadmissible Heuristics

If we are willing to to accept suboptimal solutions in exchange for fewer nodes explored, **inadmissible heuristics** can be used.

Inadmissible heuristics may overestimate costs, risk missing the optimal solution, but if they are more accurate, they can lead to fewer nodes explored.

This is especially useful when the state-space is large, and optimality is not a requirement.

4 Local Search

Local search algorithms search unsystematically from an initial state to its neighbors *without keeping track of paths and previous states.*

They are useful in **optimization problems** where finding the best possible state is important, and the path to get there is irrevelant.

4.1 Hill-climbing Search

On each iteration, **hill-climbing search** keeps track of one current state, and moves to the neighboring state with the **steepest ascent**.

It does not look beyond neighboring states, terminating at a peak, which means it can get stuck on **local maxima** and **plateaus**.

A solution to plateaus is allowing **sideways movement**, while also restricting the number of these moves so that the algorithm does not wander the plateau forever.

When there are many successors to any given state, choosing the **first-better-choice** is preferable. Another variant is **stochastic hill climbing**, which chooses a better successor randomly.

4.2 Simulated Annealing

Hill-climbing algorithms get stuck on **local maxima** since they never make downhill moves, while **random walk** algorithms will eventually reach the global maxima, albeit inefficiently.

By combining the two, doing something random once in a while allows for breaking out of local maxima, *while yielding both efficiency and completeness.*

Simulated annealing algorithms pick a random move each iteration, accepting it if it improves the situation. Otherwise, it is accepted with a probability inversely and exponentially proportional to its "badness".

Also, simulated annealing algorithms become less likely to accept bad moves as time goes on. *With enough time, it will find a global maxima with probability approaching 1.*

4.3 Beam Search

Rather than just keeping one node in memory, **beam search** performs k hill-climbing searches in parallel.

In **local beam search**, information can be shared between parallel search threads, allowing the algorithm to abandon unfruitful searches.

However, this means that all k states may end up clustered together in the state space, effectively making it a k-times slower hill-climbing search. To combat this, **stochastic beam search** randomly chooses successors independently instead.

4.4 Genetic Algorithms

In **genetic algorithms**, successor states are generated from a **recombination** of two parent states at random **crossover points**, and **random mutations** are allowed.

Not all successors are kept — **selection processes** evaluate individuals with a **fitness function**, **culling** those that do not meet a fitness threshold.

4.5 Online Search

So far, the algorithms have all been for **offline search**, in which the algorithms compute a complete solution and then execute it.

Online search, however, interleaves the computation and execution. It is useful in dynamic environments where the agent is penalized for extended computation.

It is also necessary in **exploration problems**, where the agent is clueless about the states and actions of its environment, and new observations are available only after acting.

5 Adversarial Search

Adversarial search problems arise from **competitive environments** wherein two or more agents have conflicting goals, such as in games.

Even if we could specify every possible reply to an opponent's move, it is often computationally intractable to do so. For this topic, we will only consider two-player games.

5.1 Two-player Games

In two-player games, **"move"** is a synonym for "action", and **"position"** is a synonym for "state".

A **ply** is used to mean a move by one player, increasing the depth of the **search tree** by one. This disambiguation is useful in games where a move means both players have acted.

Formally, games are defined with the following elements:

- S_0, the **initial state** of the game at the start.
- `TO-MOVE(`s`)`, the player whose turn it is to move in state s.
- `ACTIONS(`s`)`, the set of legal moves in state s.
- `RESULT(`s`,` a`)`, the **transition model** from s to s' after taking action a.
- `IS-TERMINAL(`s`)`, a **terminal test** whether the game is over (at **terminal states**).
- `UTILITY(`s`,` p`)`, a **utility function** returning a numeric value for player p when the game ends in terminal state

s.

5.2 Minimax Search

Minimax search is used in two-player games where one player is assigned `MAX` and the other `MIN`.

An optimal strategy can be determined by working out the **minimax value** of each state in the tree, where the players play optimally.

In every non-terminal state, `MAX` always seeks to maximize the minimax value, while `MIN` minimizes.

Minimax search is **complete** if the search tree is finite.

It is **optimal** only against an optimal player.

The **time complexity is $O(b^m)$** since it has to explore the entire game tree depth-first. Since the game is deterministic, with depth-first exploration, it has a **space complexity of $O(bm)$**.

However, the exponential time complexity makes it computationally intractable for complex games such as chess, with an average game having approximately 10^{123} states.

5.3 Alpha-Beta Pruning

We can optimize the computation a minimax value by keeping track of the minimum and maximum value seen thus far, ignoring paths which have no effect on the outcome of the game.

The parameter α is the best (highest) value we have found along the path for `MAX`, and conversely, β is the best (lowest) value for `MIN`.

We can think of α and β as a range of values $[\alpha, \beta]$

By **pruning** large parts of the game tree, we will not have to examine every single state, reducing the exponent in the time complexity.

5.3.1 Example of α–β Pruning

Consider a state s in which it is `MAX`'s turn to choose between two actions leading to s_A and s_B. We begin with an α–β range of $[-\infty, +\infty]$.

At s_A, when it is `MIN`'s turn, `MIN` will always choose the action with the minimum minimax value. Let this value be k.

Now, we know that the minimax value at the root node s is *at least* k, so we can update the α–β range to $[k, +\infty]$.

Then, we evaluate the minimax value for s_B. If we encounter any leaf node of S_B with a minimax value *less than* k, going down the path of s_B is a poor choice, since s_A guarantees a minimax score of *at least* k.

Therefore, the optimal path for `MAX` at node s is to go down s_A.

We did not have to consider the other leaf nodes of s_B once we had found a leaf node of s_B with value *less than* k, allowing us to **prune** the search tree!

5.3.2 Move Ordering

The order in which states are explored affects the effectiveness of α–β pruning.

With perfect ordering, only $O(b^{\frac{m}{2}})$ nodes need to be examined as compared to $O(b^m)$ by minimax.

Therefore, we can **double the search depth** at best, and at worst α–β pruning has no effect. Ultimately, **pruning does not affect the final result**.

5.4 Resource Limits

Computation time is usually limited. Eventually, we have to cut off the search when we are unable to reach all terminal states.

We therefore replace `IS-TERMINAL` with `IS-CUTOFF` with a certain depth limit, for example.

Consequently, we need a heuristic **evaluation function** `EVAL` which estimates the utility of a state, to replace the utility function `UTILITY`. Usually, it is a **weighted linear sum** of the various **features** of a state.

We can also deal with repeated states in `GRAPH-SEARCH` with **memoization**.

6 Linear Regression

Regression problems are one of the major classes of **supervised machine learning**. In regression problems, we are given data with m training examples, an input with x variables (**features**), and are tasked with predicting the output variables y.

Univariate linear regression is essentially "*fitting a straight line*".

It is defined by the **hypothesis function** $h_\theta(x) = \theta_1 x + \theta_0$. The θ_0 and θ_1 coefficients are known as **weights**.

The objective is to find the best h_θ that best fits the data well. *The line that best fits is the one with weights that minimizes the cost (mean squared error)* (L_2 loss), **without overfitting**.

By Occam's razor, we prefer the simplest hypothesis that is consistent with the data.

6.1 Cost Function

$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2$

The **cost function** is the mean of the squared errors between the data y_i and the hypothesis, as predicted by $h_\theta(x_i)$, over the m training examples in the data.

For simplicity, we often use the following simplified version of the squared errors function:

$J'(\theta_1) = \sum_{i=1}^{m}(\theta_1 x_i - y_i)^2$

Squared error, unlike absolute error (L_1 loss) is differentiable everywhere, and is more amenable to mathematical optimization. It also penalizes larger errors more heavily.

6.2 Gradient Descent

On a contour plot, the cost function is **convex**. We can minimize error by iteratively updating the weights θ_0 and θ_1 in the direction of the gradient until convergence — a method called **gradient descent**.

During each iteration, the weights θ_j are updated **simultaneously** as follows:

$\theta_j := \theta_j - \alpha\frac{\partial J(\theta_0,\theta_1,\dots)}{\partial\theta_j}$

$\implies \theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}\big(h_\theta(x_i) - y_i\big)$

$\implies \theta_1 := \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}\big(h_\theta(x_i) - y_i\big)\cdot x_i$

The parameter α is called the **learning rate**, and is always positive. Since the error is quadratic, the partial derivative will be linear, and at every step, *the cost decreases at a decreasing rate toward the minimum.*

Care must be taken when choosing the learning rate. *It is important to choose a value that is small enough that the algorithm converges quickly, but large enough that the algorithm does not overshoot the minimum.*

6.3 Variants of Gradient Descent

The gradient descent covered earlier is known as **batch** (or deterministic) **gradient descent**.

Summing all m training examples in every step (also known as an **epoch**), may be very slow when m is large.

Stochastic gradient descent is faster, randomly sampling just one of the m training examples at each step. Alternatively, **minibatch gradient descent** samples a subset of the m training examples.

6.4 Multivariable Linear Regression

Regression problems are often multivariate, having more than one feature. We can extend x to n features with an n-dimensional vector.

The **hypothesis function** then becomes:

$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n = \theta^T x$

$\theta^T x$ is the dot product of the weights with the features. Since the w_0 term is the intercept, we can fix x_0 to 1.

The **cost function** becomes:

$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2$

Gradient descent for multivariable linear regression is similar to the univariate case, except that the x_i are now n-dimensional vectors:

$\theta_j := \theta_j - \alpha\frac{\partial J(\theta)}{\partial\theta_j}$

Therefore, for n features,

$\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}\big(h_\theta(x_i) - y_i\big)\cdot x_{0,i}$

$\theta_1 := \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}\big(h_\theta(x_i) - y_i\big)\cdot x_{1,i}$

$\vdots$

$\theta_n := \theta_n - \alpha\frac{1}{m}\sum_{i=1}^{m}\big(h_\theta(x_i) - y_i\big)\cdot x_{n,i}$

6.5 Feature Scaling

Feature scaling is a method to scale the data such that all the weights are in the same range and easily comparable. Gradient descent converges faster when this is applied.

6.5.1 Min-max Normalization

By **rescaling** the range of features to $[0, 1]$ or $[1, -1]$, we can make the weights more comparable. The general formula for a scaled value x' is:

$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$

6.5.2 Mean Normalization

Alternatively, by computing the mean μ_i and standard deviation σ_i for each feature, we can **standardize** the features to have a mean of 0 and a standard deviation of 1. The general formula for a scaled value x' is:

$x' = \frac{x - \mu_i}{\sigma_i}$

6.6 Polynomial Regression

The relationship between features is not always linear.

$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$ may be valid, and so may $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2\sqrt{x}$. Feature scaling will need to be done.

6.7 Normal Equation

A matrix equation to calculate the weights, also known as the **normal equation** is:

$\theta = (X^T X)^{-1} X^T Y$

This method can be used feasibly instead of gradient descent when there are few training examples and features.

However, due to the $O(n^3)$ complexity of the normal equation, it is not recommended for large datasets, unlike gradient descent.