CS2109S
Introduction to Artificial Intelligence and Machine Learning

AY2021/22 Semester 2

Notes by Jonathan Tay

Last updated on August 22, 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,* $\text{checkmate}(x)$ *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, $\text{pred}(\text{succ}(n))$ and $\text{succ}(\text{pred}(n))$ 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:

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

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

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

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,

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

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

$\vdots$

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

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.

7 Classification

The problem of identifying which set of categories a data point belongs to.

7.1 Decision Trees (DT)

A representation of a function that maps a vector of attribute values to a single output value.

We construct a decision tree for **boolean classification** from inputs of *discrete values* for outputs which are either `true` (**positive example**) or `false` (**negative example**).

We want to find the *most compact* decision tree. However, it is computationally intractable to construct all 2^{2^n} distinct decision trees with n Boolean attributes.

There are a total of $3n$ distinct **conjunctive hypotheses** where each attribute in n is either *in (positive)*, *in (negative)*, or *out*.

A more expressive **hypothesis space** increases the chance that the target function can be expressed and also increases the number of hypotheses consistent with the training set.

7.1.1 Entropy

The measure of uncertainty inherent to a variable's possible outcomes.

$I(P(v_1), \ldots, P(v_n)) = -\sum_{i=1}^{n} P(v_i) \log_2 P(v_i)$

For a training set of p positive examples and n negative examples:

$I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = \left(-\frac{p}{p+n} \log_2 \frac{p}{p+n}\right) - \left(\frac{n}{p+n} \log_2 \frac{n}{p+n}\right)$

7.1.2 Information Gain (IG)

The expected reduction in entropy.

A chosen attribute A divides the training set E into subsets $E_1, \ldots, E_v$ according to their values for A, where A has v distinct values:

$\text{remainder}(A) = \sum_{i=1}^{v} \frac{p_i+n_i}{p+n} \cdot I\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right)$

We then measure the reduction in entropy:

$IG(A) = I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - \text{remainder}(A)$

We prefer to split on attributes with the highest IG.

7.1.3 Decision Tree Pruning

Prevent overfitting by preventing splits that do not separate training examples cleanly.

One method is **early stopping**, which stops generating nodes when there are no good attributes to split on.

Alternatively, we can use statistical techniques to determine whether the attribute is relevant.

This results in smaller trees with better accuracy.

7.1.4 Broadening the Applicability of DTs

If some examples are *missing values* for A, we can still use the training examples for the DT. If a node tests A, we assign the most common value of A with the same output value.

If some attributes have continuous values, it may suffice to use a **split point** inequality test using some threshold or interval.

However, if these continuous values do not have meaningful ordering, we can use **information gain ratio** instead:

$\text{SplitInformation}(C, A) = -\sum_{i=1}^{d} \frac{|E_i|}{|E|} \log_2 \frac{|E_i|}{|E|}$

$\text{GainRatio}(C, A) = \frac{IG(C,A)}{\text{SplitInformation}(C,A)}$, where

$E_1, \ldots, E_d$ are the subsets of the training set E divided by the attribute A.

Alternatively, **equality tests** can be used to separate specific values from the rest.

If some attributes have *differing costs*, we can replace IG with $\frac{IG^2(C,A)}{\text{Cost}(A)}$ or $\frac{2^{IG(C,A)}-1}{(\text{Cost}(A)+1)^w}$ where $w \in [0,1]$ determines the importance of the cost.

7.2 Logistic Regression

A **decision boundary** is a line (or hyperplane in higher dimensions) which separates two classes.

A linear decision boundary is called a **linear separator** and data that admit such a separator are called **linearly separable**.

In linear regression, we used a **threshold function** to separate the classes:

$$h_\theta(x_1, x_2) = \begin{cases} 1 \text{ (positive class)} & \text{if } \theta_0 + \theta_1 x_1 + \theta_2 x_2 > 0 \\ 0 \text{ (negative class)} & \text{otherwise} \end{cases}$$

However, $h_\theta(x)$ is *discontinuous* and *not differentiable*.

7.2.1 Sigmoid Function

Maps a real number to a real number in the range [0, 1] in an S-shaped curve.

The sigmoid function $g(z) = \frac{1}{1+e^{-z}}$.

In logistic regression, our hypothesis is $h_\theta(x) = g(\theta^T x)$, and we want to perform gradient descent.

7.2.2 Cost Function

We cannot use sum of mean squared errors for our cost function because it is *non-convex*. Gradient descent will converge at a local minima instead of a global minima, which is undesirable.

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log h_\theta(x_i) & \text{if } y = 1 \\ -\log 1 - h_\theta(x_i) & \text{if } y = 0 \end{cases}$$

This simplifies to the following:

$$\text{Cost}(h_\theta(x), y) = -y \log h_\theta(x) - (1-y) \log(1 - h_\theta(x))$$

Therefore, our overall cost function is:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}\left(h_\theta(x^{(i)}),\ y^{(i)}\right)$$

7.2.3 Gradient Descent

Our gradient descent update rule is the exact same as in linear regression!

$$\theta_n := \theta_n - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_n^{(i)},$$

where for the n-th feature, α is the learning rate, m is the number of training examples, and i is the index of the training example.

$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$, the hypothesis function, is the *estimated probability* that $y = 1$ for the given input x.

7.2.4 Multi-class Classification

Binary classification for $y \in \{0, 1\}$ works, but what if we have more than two classes?

In this case, we train a logistic classifier $h_\theta^{(i)}(x)$ for each class i to predict that $y = i$.

For each input x, we then predict the class i with the highest probability: $y = \text{argmax}_i\, h_\theta^{(i)}(x)$. This is known as **one-vs-all** or **one-vs-rest** classification.

8 Bias-Variance Tradeoff

A choice between more complex, low-bias hypotheses that fit the training data well, and simpler, low-variance hypotheses that may generalize better.

Machine learning models can only approximate the true function for the relationship between features.

Bias is the tendency of a predictive hypothesis to deviate from the expected value when averaged over different training sets.

When a model fails to capture the true relationship between features, it has high bias, and tends to **underfit**.

Variance is the amount of change in the hypothesis due to differences in datasets.

When a model fits a training set really well but performs poorly on a testing set, it is said to **overfit**.

9 Regularization

A technique to reduce the complexity of a model to reduce overfitting.

We want to keep all features in the model, and instead reduce the magnitude of the weight parameters θ_j.

9.1 Regularized Linear Regression

In linear regression, the cost function $J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$, i.e., the sum of squared errors.

Regularization introduces adds a regularization parameter λ to the cost function:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m} \sum_{i=1}^{n} \theta_i^2$$

This makes $\theta_1, \ldots, \theta_n$ smaller, effectively cancelling out higher order terms in the polynomial $y = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_n x^n$.

Gradient descent just has an extra term in the update rule: $\theta_n := \left(1 - \frac{\alpha\lambda}{m}\right) \theta_n - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_n^{(i)}$

The normal equation also has the λ term added to it: $\theta = (X^T X + \lambda D)^{-1} X^T Y$, where $D = I$ but with $D_{11} = 0$.

Unlike the regular normal equation where $X^T X$ must be invertible, the regularized normal equation can be solved even if $X^T X$ is not invertible as long as $\lambda > 0$!

9.2 Regularized Logistic Regression

The fully expanded cost function has the following form:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log\left(1 - h_\theta(x^{(i)})\right)\right)$$

Regularization appends the following to the back:

$\cdots + \frac{\lambda}{2m}\sum_{i=1}^{n}\theta_i^2$

Gradient descent also has an extra term in the update rule:

$\theta_n := \theta_n - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_n^{(i)} - \frac{\lambda}{m}\theta_n$

10 Support-vector Machine (SVM)

A technique to effeciently perform non-linear classification, in addition to linear classification.

The cost function for an SVM, $J(\theta) =$

$$\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\,\text{cost}_1(\theta^T x) + (1-y^{(i)})\,\text{cost}_0(\theta^T x)\right) + \frac{\lambda}{2m}\sum_{i=1}^{n}\theta_i^2$$

where $\text{cost}_1(\theta^T x)$ is some linear approximation of the non-linear $\log h_\theta$ term for values ≤ 1, and 0 otherwise.

$\text{cost}_0(\theta^T x)$ is some linear approximation of the non-linear $\log h_\theta$ term for values ≥ -1, and 0 otherwise.

The hypothesis for an SVM is more clean-cut than logistic regression:

$$h_\theta(x) = \begin{cases} 1 & \text{if } \theta^T x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Minimization of the cost function (where $C \approx \frac{1}{\lambda}$):

$$\min_\theta C\left[\sum_{i=1}^{m} y^{(i)}\,\text{cost}_1(\theta^T x) + (1-y^{(i)})\,\text{cost}_0(\theta^T x)\right] + \frac{1}{2}\sum_{i=1}^{n}\theta_i^2$$

Therefore, we simplify SVMs as $CA + B$, while logistic regression is $A + \lambda B$. They are relatively similar!

This minimization works because $\theta^T x^{(i)} \geq 1$ if $y = 1$, and $\theta^T x^{(i)} \leq -1$ if $y = 0$. As a result, the costs from cost_1 and cost_0 are minimized.

SVMs construct a **maximum margin separator**, which is a decision boundary with the largest possible distance to each training example.

A large λ value ignores outliers, and since $C \approx \frac{1}{\lambda}$, a small C value ignores outliers in SVMs. We need to ignore some outliers and allow misclassifications, otherwise, the **margins** of the decision boundary will be too small.

Ultimately, only the support vectors—the points closest to the separator—matter. All the weights for non support vectors are zero.

10.1 Kernel Tricks

A technique for enabling classification on non-linearly separable data.

The **Gaussian kernel** defines the hypothesis differently:

$$h_\theta(x) = \begin{cases} 1 & \text{if } \theta_0 + \theta_1 f_1 + \cdots + \theta_n f_n \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where $f_i = \text{similarity}(x, l_i) = e^{-\frac{|x-l_i|^2}{2\sigma^2}}$, and training example x and **landmark** l_i are vectors. This finds the "distance" between the two vectors.

This process does regression on a *transformed space*.

When x is very near l_i, the distance $|x-l_i|^2 \to 0$, causing $f(x, l_i) \to 1$ due to the exponentiation.

Likewise, when x is far from l_i, the distance $|x-l_i|^2 \to \infty$, causing $f(x, l_i) \to e^{-\infty} = 0$.

Every point $h_\theta(x)$ will then be a linear combination of the landmarks such that a hyperplane in a higher dimensional space separates included points from the excluded points.

The SVM does this "slicing", and the kernel does the "distortion".

10.1.1 Bias-Variance Tradeoff

For classification problems, instead of computing a loss value, we can use misclassification error:

$$\text{error}(h_\theta(x), y) = \begin{cases} 1 & \text{if } h_\theta(x) \leq 0.5 \wedge y = 1 \\ 1 & \text{if } h_\theta(x) < 0.5 \wedge y = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$J_{\text{test}}(\theta) = \frac{1}{m_{\text{test}}}\sum_{i=1}^{m_{\text{test}}}\text{error}(h_\theta(x_{\text{test}}^{(i)}), y^{(i)})$$

We can train multiple models and use cross-validation to choose the one with the lowest $J_{\text{cv}}(\theta)$.

The formula for $J_{\text{cv}}(\theta)$ and $J_{\text{train}}(\theta)$ are the same as $J_{\text{test}}(\theta)$, but instead using different training examples x.

We only use test set error $J_{\text{test}}(\theta)$ to assess the quality of the *chosen* model, and use cross-validation error $J_{\text{cv}}(\theta)$ to *choose* the model.

This is because cross-validation may introduce a bias in the model, and we want the training examples in the test set to be unseen.

10.2 SVM with Kernels

A larger $C \approx \frac{1}{\lambda}$ value will yield lower bias an higher variance, while a smaller C value will yield higher bias and lower variance.

The σ term determines the steepness of the "distortions" by the kernel.

If σ^2 is large, they are gradual (higher bias, lower variance). If σ^2 is small, they are sharp (lower bias, higher variance).

We will use all m training examples as landmarks l. The f vector comprises of similarity values between all training examples and each landmark: $f_m = \text{similarity}(x, l^{(m)})$.

The θ vector is has $m+1$ values, where θ_0 is the bias, and we want to minimize θ over the same function as in regularized logistic regression but with f instead of x, and n is replaced with m.

Remember to apply feature scaling *before* transformation with kernels!

SVMs with Gaussian kernels should be used when n is small and $m < 10$k, because each training example is used as a landmark.

Otherwise, use logistic regression or SVMs without kernels (linear kernel).

11 Neural Networks

A perceptron **calculates** the weighted sum of its inputs and passes it through a non-linear function:

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{n} w_i \cdot x_i\right)$$

where g is some non-linear activation function, $x_0 = 1$, and w_0 is the bias term.

Examples of non-linear activation functions are the sigmoid function, the tanh function, and the ReLU ($\max(0, x)$) function.

11.1 Perceptron Learning Algorithm

1. Initialize weights w_i to zero or random small values.
2. Classify $\hat{y}^{(i)} = \text{sgn}(w^T x^{(i)})$ for each instance i with features $x^{(i)}$.
3. Select one misclassified instance and update its weights: $w := w + \eta \cdot (y - \hat{y}) \cdot x$ where η is the learning rate.
4. Iterate steps 2 to 3 until convergence where the classification error is less than some threshold, or we reach a maximum number of iterations.

The PLA algorithm is not robust as it can select and linear separator and is non-determistic. It also does not converge on non-linearly separable data.

On the other hand, the Linear SVM is a perceptron of optimal stability which maximizes margin, and also converges on non-linearly separable data.

11.2 Gradient Descent

For $\hat{y} = g(f(x))$, $f(x) = w^T x$, the update rule for gradient descent in a neural network is:

$$w_i := w_i - \eta \frac{d\epsilon}{dw_i} = w_i - \eta(\hat{y} - y) g'(f) x_i$$

The derivative of the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ is:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

Generalizing this to the gradient descent update rule:

$$w \leftarrow w - \eta(\hat{y} - y)\hat{y}(1 - \hat{y})x$$

11.3 Multi-layer Perceptron

A neural network.

Suppose an l-th layer of a neural network has m neurons. The layer before it is the $l-1$-th layer, the layer after it is the $l+1$-th layer, and so on.

Among these m neurons, let's consider the j-th neuron in the layer. It is indexed as $a_j^{[l]}$.

Suppose the network is fully connected, such that this neuron is connected to all neurons in the previous layer.

Then, it is connected to the neurons:

$a_1^{[l-1]}, \ldots, a_j^{[l-1]}, \ldots, a_m^{[l-1]}$.

It is also connected to the bias term $a_0^{[l-1]} = 1$

Each of these neurons are weighted by:

$w_{j0}^{[l]}, w_{j1}^{[l]}, \ldots, w_{jj}^{[l]}, \ldots, w_{jm}^{[l]}$

Note that the superscript for the weights is $[l]$, not $[l-1]$.

11.4 Backpropagation

Let's first consider a single neuron (perceptron):

$$\hat{y} = F(w, x, b) = wx + b$$
$$\varepsilon = L(\hat{y}, y)$$

Now, we want to find $\frac{\partial \varepsilon}{\partial w}$ and $\frac{\partial \varepsilon}{\partial b}$.

We get the following gradients by applying the chain rule:

$$\frac{\partial \varepsilon}{\partial w} = \frac{\partial \varepsilon}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w}$$
$$\frac{\partial \varepsilon}{\partial b} = \frac{\partial \varepsilon}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b}$$

We don't have any downstream or upstream gradients, since we're working with a singular neuron.

When we use a typical non-linear activation function σ in $\hat{y} = \sigma(wx + b)$, we can split into two parts:

$$z = wx + b$$
$$\hat{y} = \sigma(z)$$

and then get the gradients by following the computation graph:

$$\frac{\partial \varepsilon}{\partial w} = \frac{\partial \varepsilon}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}$$
$$\frac{\partial \varepsilon}{\partial b} = \frac{\partial \varepsilon}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial b}$$

Now let's generalize it to a single layer. Suppose we have an input x with n features $x_1, x_2, \ldots, x_n$, and each feature has its associated weight w_i.

Therefore, x is an $n \times 1$ vector and w is a $d \times n$ matrix, where d is the **embedding dimension**.

Let's add another layer Z_1 and consider the original layer as $\hat{Y}$:

$$z^{[1]} = (W^{[1]})^T x + b^{[1]}$$
$$\hat{Y} = (W^{[2]})^T z^{[1]} + b^{[2]}$$
$$\varepsilon = L(\hat{Y}, y) = \hat{Y} - Y$$

Now, we need to compute two gradients: $\frac{\partial \varepsilon}{\partial W^{[1]}}$ and $\frac{\partial \varepsilon}{\partial W^{[2]}}$.
ε is a vector, and W_i are matrices. Suppose we have some arbitary matrices x, $W^{[1]}$, and $W^{[2]}$ as follows:

$$\begin{aligned} z_1 &= (W_1)^T x \\ &= \begin{bmatrix} w_{11}^{[1]} & w_{21}^{[1]} \\ w_{12}^{[1]} & w_{22}^{[1]} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= \begin{bmatrix} w_{11}^{[1]} x_1 + w_{12}^{[1]} x_2 \\ w_{21}^{[1]} x_1 + w_{22}^{[1]} x_2 \end{bmatrix} \\ \hat{Y} &= (W^{[2]})^T z_1 \\ &= \begin{bmatrix} w_{11}^{[2]} & w_{21}^{[2]} \\ w_{12}^{[2]} & w_{22}^{[2]} \end{bmatrix} \begin{bmatrix} w_{11}^{[1]} x_1 + w_{12}^{[1]} x_2 \\ w_{21}^{[1]} x_1 + w_{22}^{[1]} x_2 \end{bmatrix} \\ &= \begin{bmatrix} w_{11}^{[2]}(w_{11}^{[1]} x_1 + w_{21}^{[1]} x_2) + w_{21}^{[2]}(w_{12}^{[1]} x_1 + w_{22}^{[1]} x_2) \\ w_{12}^{[2]}(w_{11}^{[1]} x_1 + w_{21}^{[1]} x_2) + w_{22}^{[2]}(w_{12}^{[1]} x_1 + w_{22}^{[1]} x_2) \end{bmatrix} \end{aligned}$$

This results in a 2×1 vector $\hat{Y}$, which we pass to L:

$$\begin{aligned} e_1 &= y_1 \\ &\quad - w_{11}^{[2]} \left(w_{11}^{[1]} x^{[1]} + w_{21}^{[1]} x^{[2]} \right) \\ &\quad - w_{12}^{[2]} \left(w_{12}^{[1]} x^{[1]} + w_{22}^{[1]} x^{[2]} \right) \\ e_2 &= y_2 \\ &\quad - w_{12}^{[2]} \left(w_{11}^{[1]} x^{[1]} + w_{21}^{[1]} x^{[2]} \right) \\ &\quad - w_{22}^{[2]} \left(w_{12}^{[1]} x^{[1]} + w_{22}^{[1]} x^{[2]} \right) \\ \implies \varepsilon &= \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} \end{aligned}$$

This entire sequence is **forward propagation**. Now for backpropagation.

Since our weight matrices are 2×2, we can calculate 4 gradients each for both $\frac{\partial e_1}{\partial w^{[2]}}$ and $\frac{\partial e_2}{\partial w^{[2]}}$ from $\frac{\partial \varepsilon}{\partial w^{[2]}}$ using the chain rule.

11.5 Performance Measures

We consider a classification correct when a prediction $\hat{y}$ is equal to the actual label y: i.e., $\text{Correct} = [\hat{y}] = y$.

We can define **accuracy** as the average correctness across a dataset with m examples:

$$A = \frac{1}{m} \sum_{i=1}^{m} [\hat{y}_i = y_i]$$

where $\hat{y}_j = M(x_j)$ is the predicted label from model M for the j-th x_j, and y_j is the **ground truth** value of the j^{th} example.

11.5.1 Confusion Matrix

Visualizing the performance of a machine learning model.

For binary confusion, i.e., positive and negative labels, we can use the confusion matrix:

$$\begin{bmatrix} \text{true positive, TP} & \text{false positive, FP} \\ \text{false negative, FN} & \text{true negative, TN} \end{bmatrix}$$

The columns represent the *actual label*, while the rows represent the *predicted label*.

Confusion matrices are $N * N$, where N is the number of labels.

We can calculate accuracy as follows:

$$A = \frac{TP + TN}{TP + TN + FP + FN}$$

False positives are also known as **Type I error**, and false negatives as **Type II error**.

Precision is the fraction of selected items which are relevant:

$$P = \frac{TP}{TP + FP}$$

We want to maximize precision when false positives are costly, e.g., spam.

Recall is the fraction of relevant items which are selected:

$$R = \frac{TP}{TP + FN}$$

We want to maximize recall when false negatives are dangerous, e.g., cancer prediction.

A more robust measure of accuracy is the $\mathbf{F_1}$ **score**:

$$F_1 = \left(\frac{P^{-1} + R^{-1}}{2} \right)^{-1} = \frac{2\,TP}{2\,TP + FP + FN}$$

It is less sensitive to extreme values, and considers the numerators of P and R as the same, comparing their denominators instead.

11.5.2 Receiver Operator Characteristic (ROC) Curve

Visualizing the performance of a binary classifier as its discrimination threshold is varied.

An ROC curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold values π.

Random chance is a diagonal line increasing linearly from the origin. A model is more accurate than random chance if its ROC curve is above this random line.

We can use the area under the curve (AOC) of the ROC as a more concise metric for clearer comparisons.

An AOC value > 0.5 indicates that the model is more accurate than random chance, and AOC ≈ 1 indicates that the model is very accurate.

12 Deep Learning

A neural network with many layers is said to be a **deep network**, while one with few layers is a **shallow network**.

In deep learning, we want to automate feature extraction and classification.

12.1 Recurrent Neural Networks

Exploiting temporal structure.

12.2 Convolutional Neural Networks

Exploiting spatial structure in images.

Images are represented as 2D matrices.

We could naively use pixel value vectors, butthat would lose information on spatial structure, where pixels are more related the closer they are to each other.

12.2.1 Convolution

Consider groups or pixels at a time rather than individual pixels.

We employ a **kernel** to convolve with the image. This is done via a sliding window over the image, multiplying the window and kernel elementwise and summing to get a **feature map**.

Padding mitigates pixel loss around the edge by surrounding the image with a border of zeros.

Stride controls the distance between adjacent windows to speed up the computation of feature maps.

For an image of dimensions $W \times H$, the dimensions of the feature map are:

$$\text{width} = \lfloor \frac{W - K + 2 * P}{S} \rfloor + 1$$
$$\text{height} = \lfloor \frac{H - K + 2 * P}{S} \rfloor + 1$$

where K is the size of the kernel, P is the padding size, and S is the stride.

A linear layer had a 2D weight matrix $\mathbf{W}^{[l]}$ and a non-linear activation function $g^{[l]}$:

$$\mathbf{a}^{[l]} = g^{[l]}\left((\mathbf{W}^{[l]})^T \mathbf{a}^{[l-1]}\right)$$

However, in a convolution layer, both the weight and activation matrices are 3D, since we're concatenating the feature maps for the activation:

$$\mathbf{a}^{[l]} = g^{[l]}\left(\mathbf{W}^{[l]} * \mathbf{A}^{[l-1]}\right)$$

Each layer has multiple kernels, and each kernel output is a 2D matrix of a feature map.

The kernel input is a 3D matrix of feature maps, where each depth is a different kernel, akin to "stacking filters". Thus, each layer is a 3D matrix.

12.2.2 Pooling

Downsamples feature maps to help train later kernels in detecting higher-level features.

This reduces dimensionality via aggregation methods like max pooling, where the largest value in each window is selected.

Softmax is used to normalize the output of the last layer. It exponentiates every term and divides it by the sum of all exponentiated terms:

$$\text{softmax}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} \frac{e^{x_1}}{e^{x_1}+e^{x_2}+e^{x_3}} \\ \frac{e^{x_2}}{e^{x_1}+e^{x_2}+e^{x_3}} \\ \frac{e^{x_3}}{e^{x_1}+e^{x_2}+e^{x_3}} \end{bmatrix}$$

12.2.3 Regularization

We can employ **dropout** to prevent overfitting by randomly setting some activations to 0.

Alternatively, **early stopping** prevents overfitting by stopping training when the loss has stopped decreasing.

12.2.4 Other Problems

Vanishing gradients is a problem in backpropagation where the gradients are too small or even zero such that multiplying gradients doesn't change weights.

This is why we avoid the use of the sigmoid function which saturates at 0 and 1 for very large or small values.

Exploding gradients causes gradients to keep getting larger and larger, which causes gradient descent to diverge.

We mitigate these by:

1. Proper weight initialization.
2. Using non-saturating activation functions such as ReLU.
3. Batch normalization, c.f., feature scaling.
4. Gradient clipping.

13 Unsupervised Learning

Learning from data without labels.

We want our model to derive structure from data without prior knowledge of the effect of variables.

The model also does not receive feedback based on prediction results, since we have little or no idea what results should look like.

13.1 k-means Clustering

Unsupervised classification.

We have m training data points $x^{(i)}, x^{(2)}, \ldots, x^{(m)}$. We want to classify these data points into k clusters.

To do so, we initialize k centroids $\mu_1, \mu_2, \ldots, \mu_k$ to random points in the data.

We use $c^{(i)}$ to denote the index of the cluster centroid closest to $x^{(i)}$ such that $1 \leq c^{(i)} \leq k$.

We use the mean squared error as our cost function:

$$J(c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_k) = \frac{1}{m} \sum_{i=1}^{m} \left(x^{(i)} - \mu^{(i)} \right)^2$$

Then, we repeat the following until convergence (which is guaranteed):

- For each data point, find the closest centroid $\mu^{(i)}$ and assign it to the cluster $c^{(i)}$.
- For each cluster, find the new centroid $\mu^{(i)}$ by taking the mean of all the data points in the cluster.

There are many possible local optima that k-means will converge on, so we try multiple random initializations and choose the one that minimizes the cost function.

k-medoids is similar to k-means, but for each cluster, we instead pick the data point with minimum average distance to every other point as the centroid. This allows the final centroids to be more interpretable.

There are two ways to pick k.

The **elbow method** plots the total loss J against a varying number of clusters k and picking the k value at the inflection point.

More clusters will naturally improve the fit, where loss decreases sharply up to k and then increasing only slowly beyond k due to over-fitting.

The **business need** method simply picks k by the downstream need for a number of clusters.

13.2 Heirarchical Clustering

Initially, we treat every data point as a one-point cluster.

Then, we merge pairs of points that are nearest to each other and merge the two clusters.

We repeat until all data points are in one cluster.

We get a tree of clusters called a **dendrogram**, and we can cut off at some threshold, or at a certain number of clusters.

However, heirarchical clustering is impractical for extremely large datasets due to its high space and time complexity.

13.3 Challenges

In general, unsupervised learning has higher computational complexity, risk inaccurate results, take longer to train, need human validation of results, and lack transparency in clustering.

13.4 Principal Component Analysis

Dimensionality reduction.

Data with many features have high dimension, which is more computationally expensive to work with.

PCA allows us to extract only the relevant features in a training samples in a data set by reducing the dimension from n to k such that $x^{(i)} \in \mathbb{R}^n \mapsto z^{(i)} \in \mathbb{R}^k$.

We then recover the original data by projecting the reduced data back to the original space $z^{(i)} \in \mathbb{R}^k \mapsto x^{(i)}_{\text{approx}} \in \mathbb{R}^n$.

Therefore, we need to pick a value for k. We want to find a minimum value of k such that we retain 99% of the variance in the data.

Mathematically, this is represented by the following inequality:

$$\frac{\sum_{i=1}^{m} ||x^{(i)} - x^{(i)}_{\text{approx}}||^2}{\sum_{i=1}^{m} ||x^{(i)}||^2} = 1 - \frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{n} \sigma_i} \leq 0.01$$

$$\implies \frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{n} \sigma_i} \geq 0.99$$

In words, this means pick the first k values of σ such that they account for 99% of the variance, i.e., sum of all the σ values.

We get the σ values from `s` after calling `svd` on the data matrix X which performs **singular value decomposition**:

```
U, s, VT = svd(X)
```

`U` and `VT` are both $n \times n$ matrices.

`s` is typically an $n \times n$ diagonal matrix of σ values, but in code it is represented as a $1 \times n$ array such that $s[i] = \sigma_i$.

These σ values are sorted in descending order, and can be thought of as corresponding to "the importance of each feature" in the data.

We select the first k values of σ to get the k principal components:

$\mathbf{U}_{\text{reduce}} \equiv$ `Ur = U[:, :k]`

With this, we can apply PCA *only to the training set* and not the test and cross-validation sets.

Only once we get $\mathbf{U}_{\text{reduce}}^T$ can we apply it to the test and cross-validation sets:

$$\mathbf{Z} = \mathbf{U}_{\text{reduce}}\ \mathbf{X}$$
$$\mathbf{X}_{\text{approx}} = \mathbf{U}_{\text{reduce}}^T\ \mathbf{Z}$$

PCA is often used for compression and visualization of high dimensional data. Do not use PCA to prevent overfitting; use regularization instead.

14 Miscellany

Things of note and examples from tutorials.

14.1 Forward Propagation

Given a fully connected neural network with one hidden layer, a 2D input x with two neurons, and two output neurons $\hat{y}$, calculate the values after forward propagation including biases and using the ReLU function.

We are given the weights $\mathbf{W}^{[1]}$ and $\mathbf{W}^{[2]}$ and the input $\mathbf{X}$.

$$a^{[1]} = \text{ReLU}\left(\left(\mathbf{W}^{[1]}\right)^T \mathbf{X}\right)$$
$$y^{[2]} = \text{ReLU}\left(\left(\mathbf{W}^{[2]}\right)^T a^{[1]}\right)$$

14.2 Calculating Dimensions

In a regular neural network without convolution, given an $w \times h$ greyscale image, fill in the dimensions of the input and output tensors as well as the weight matrices.

After flattening our input image, we get input matrix of size $w \cdot h \times 1$. Given that we want our output dimension to be $k \times 1$, the weight matrix should have dimensions $k \times w \cdot h$.

Repeat for each subsequent layer.

14.3 Backpropagation

Given the following network,

$$f^{[1]} = \left(\mathbf{W}^{[1]}\right)^T \mathbf{X}$$
$$\hat{Y} = g^{[1]}\left(f^{[1]}\right)$$
$$\varepsilon = -\alpha\left(Y \cdot \log \hat{Y}\right) - \beta\left((1-Y)\log(1-\hat{Y})\right)$$

where

$$\mathbf{W}^{[1]} = [W_{00}^{[1]}, W_{01}^{[1]}, W_{02}^{[1]}, W_{03}^{[1]}]$$
$$\mathbf{X}_{0i} = [1, X_{1i}, X_{2i}, X_{3i}]$$
$$g^{[1]} = \sigma(s) = \frac{1}{1+e^{-s}}$$

find $\frac{\partial \varepsilon}{\partial \hat{Y}}, \frac{\partial \varepsilon}{\partial f^{[1]}}, \frac{\partial \varepsilon}{\partial W_{02}^{[1]}}$ using the chain rule.

$$\frac{\partial \varepsilon}{\partial \hat{Y}} = -\frac{\alpha Y}{\hat{Y}} + \frac{\beta(1-Y)}{1-\hat{Y}}$$

14.4 Decision Trees

First calculate the entropy for the entire data set:

$$I\left(\frac{p}{m}, \frac{n}{m}\right) = -\frac{p}{m}\log_2\frac{p}{m} - \frac{n}{m}\log_2\frac{n}{m}$$

where m is the number of examples in the data set, p is the number of positive examples, and n is the number of negative examples.

Then, for each attribute A, calculate $\text{remainder}(A)$. If the attribute is binary, i.e., yes/no, true/false, y/n:

$$\begin{aligned}
&\text{remainder}(A) \\
&= \frac{y_A}{m} \cdot I\left(\frac{y_A \cap p}{y_A}, \frac{y_A \cap n}{y_A}\right) + \frac{n_A}{m} \cdot I\left(\frac{n_A \cap p}{n_A}, \frac{n_A \cap n}{n_A}\right) \\
&= \frac{y_A}{m}\left(-\frac{y_A \cap p}{y_A}\log_2\frac{y_A \cap p}{y_A}, -\frac{y_A \cap n}{y_A}\log_2\frac{y_A \cap n}{y_A}\right) \\
&+ \frac{n_A}{m}\left(-\frac{n_A \cap p}{n_A}\log_2\frac{n_A \cap p}{n_A}, -\frac{n_A \cap n}{n_A}\log_2\frac{n_A \cap n}{n_A}\right)
\end{aligned}$$

where y_A is the number of examples with a "yes" for attribute A, and n_A is the number of examples with a "no" for attribute A.

With the remainder for the attribute, we can calculate the information gain by splitting on that attribute:

$$IG(A) = I\left(\frac{p}{m}, \frac{n}{m}\right) - \text{remainder}(A)$$

When we have all the information gain values for every attribute, we choose to split on the attribute with the highest information gain.

If this is the first split, we consider that attribute as the **root node**.

Now, we divide the data set into two, one for the "yes" for the attribute, and the other for the "no" of the same attribute we split on.

We repeat the the previous three calculation steps to determine the next attribute to split on, and so on, until we exhaust all attributes.