Outline of This Lecture

I. In-depth: Learning Decisions Trees

II. Reinforcement Learning
Decision Tree Learning

- DTL is one of the most widely used and practical methods for classification in supervised learning.
- It is particularly useful when one wants a compact and easily understood representation.
- In this lecture we will focus on discrete input variables only, in this context called “attributes”.

Example: A Decision Tree for PlayTennis

An instance is tested by starting at the root node and moving down the tree to a leaf node where a decision is made.

An instance is tested by starting at the root node and moving down the tree to a leaf node where a decision is made.
A Connection to Propositional Logic...

In general, a decision tree represents a disjunction of conjunctions of constraints on the attribute values of instances.

![Decision Tree Diagram]

\[(\text{Outlook} = \text{Sunny} \land \text{Humidity} = \text{Normal}) \lor (\text{Outlook} = \text{Overcast}) \lor (\text{Outlook} = \text{Rain} \land \text{Wind} = \text{Weak})\]

\[<\text{Outlook} = \text{Sunny}, \text{Temperature} = \text{Hot}, \text{Humidity} = \text{High}, \text{Wind} = \text{Strong}> \rightarrow \text{NO}\]

How Do We Train a Decision Tree Classifier?

The hypothesis space is the set of all possible decision trees
How large is this hypothesis space?
Assume the decision tree has binary inputs
Consider the a truth table for the input attributes, it has size $O(2^n)$
On top of this we have the possible output classifications for each row in the input truth table, in total $O(2^n)$ possibilities
All combinations of these is $O(2^{2n})$
Further, the attributes can be put in different order in a tree...
This is a huge hypothesis space to search over, what to do?
Decision Tree classifiers are usually trained by some clever greedy heuristic search over the hypothesis space
The Decision Tree Learning Algorithm

Decision trees are learned by constructing them top-down
1. Which attribute should be tested at the root of the tree?
   o Each attribute is evaluated using a statistical test to
determine how well it alone classifies the training samples.
   o The best attribute is selected and used as the test at the
   root node of the tree.
2. Descendants of the root node are created for each possible
   value of this attribute, and the training examples matching
   this value for the attribute are propagated down to that node
3. The entire process is then repeated using the training
   examples associated with each descendant node to select the
   best attribute to test at that point in the tree (step 1).
   This forms a greedy search for an acceptable decision tree, in
   which the algorithm never backtracks to reconsider earlier
   choices.

Which Attribute is the best classifier?

We want to select the attribute that is most useful for classifying
examples.

A statistical property called information gain will be used
It measures how well a given attribute separates the training
examples according to their target classification.

We will use this information gain measure to select among the
candidate attributes at each step while growing the tree.

We first require the notion of entropy for a collection S of examples.
Entropy measures the (im)purity in a collection of training examples
(mixture of positive and negative examples)
Entropy

Given a set $S$, containing positive and negative examples of some target concept, the entropy of $S$ relative to this boolean classification is

$$\text{Entropy}(S) = -p_+ \log_2 p_+ - p_- \log_2 p_-$$

where $p_+$ and $p_-$ are the proportions of positive and negative examples in $S$.

Suppose $S$ is a collection of 14 examples of some boolean concept with 9 positive and 5 negative examples. Then the entropy of $S$ would be

$$\text{Entropy}([9+,5-]) = - (9/14) \log_2 (9/14) - (5/14) \log_2 (5/14) \text{ bits}$$

An interpretation of entropy from information theory is that it specifies the minimum number of bits of information needed to encode the target concept.

- if $p_+ = 1$ the receiver knows the drawn example will be positive and thus the entropy is 0.
- if $p_+ = 0.5$, one bit is required to indicate whether the drawn example is positive or negative so the entropy is 1.

Information Gain

Information gain is simply the expected reduction in entropy caused by partitioning the examples according to an attribute.

The information gain, $\text{Gain}(S,A)$ of an attribute $A$ relative to a collection of examples $S$, is defined as

$$\text{Gain}(S,A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \times \text{Entropy}(S_v)$$

where $S_v$ is the subset of $S$ for which attribute $A$ has value $v$ and $\text{Values}(A)$ is the set of all possible values of $A$.

The second term is the expected value of entropy after $S$ is partitioned using attribute $A$. 
### Some Training Samples

<table>
<thead>
<tr>
<th>Day</th>
<th>Outlook</th>
<th>Temperature</th>
<th>Humidity</th>
<th>Wind</th>
<th>PlayTennis</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>Weak</td>
<td>No</td>
</tr>
<tr>
<td>D2</td>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>Strong</td>
<td>No</td>
</tr>
<tr>
<td>D3</td>
<td>Overcast</td>
<td>Hot</td>
<td>High</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D4</td>
<td>Rain</td>
<td>Mild</td>
<td>High</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D5</td>
<td>Rain</td>
<td>Cool</td>
<td>Normal</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D6</td>
<td>Rain</td>
<td>Cool</td>
<td>Normal</td>
<td>Strong</td>
<td>No</td>
</tr>
<tr>
<td>D7</td>
<td>Overcast</td>
<td>Cool</td>
<td>Normal</td>
<td>Strong</td>
<td>Yes</td>
</tr>
<tr>
<td>D8</td>
<td>Sunny</td>
<td>Mild</td>
<td>High</td>
<td>Weak</td>
<td>No</td>
</tr>
<tr>
<td>D9</td>
<td>Sunny</td>
<td>Cool</td>
<td>Normal</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D10</td>
<td>Rain</td>
<td>Mild</td>
<td>Normal</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D11</td>
<td>Sunny</td>
<td>Mild</td>
<td>Normal</td>
<td>Strong</td>
<td>Yes</td>
</tr>
<tr>
<td>D12</td>
<td>Overcast</td>
<td>Mild</td>
<td>High</td>
<td>Strong</td>
<td>Yes</td>
</tr>
<tr>
<td>D13</td>
<td>Overcast</td>
<td>Hot</td>
<td>Normal</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D14</td>
<td>Rain</td>
<td>Mild</td>
<td>High</td>
<td>Strong</td>
<td>No</td>
</tr>
</tbody>
</table>

### Computing Information Gain

Computing the information gain for attribute Wind

Values(Wind) = Weak, Strong

\[ S = [9+, 5-] \]

\[ S_{\text{Weak}} \leftarrow [6+, 2-] \]

\[ S_{\text{Strong}} \leftarrow [3+, 3-] \]

\[
\text{Gain}(S, \text{Wind}) = \text{Entropy}(S) - \sum_{v = \{\text{Weak,Strong}\}} \frac{|S_v|}{|S|} \times \text{Entropy}(S_v)
\]

\[ = \text{Entropy}(S) - \frac{8}{14} \times \text{Entropy}(S_{\text{Weak}}) - \frac{6}{14} \times \text{Entropy}(S_{\text{Strong}}) \]

\[ = 0.940 - \frac{8}{14}0.811 - \frac{6}{14}1.00 \]

\[ = 0.048 \]
A Decision Tree for PlayTennis Table

Choosing the first attribute:

- Gain(S, Outlook) = 0.246
- Gain(S, Humidity) = 0.151
- Gain(S, Wind) = 0.048
- Gain(S, Temperature) = 0.029

Outlook

- Sunny
- Overcast
- Rain

[D1, D2, D8, D9, D11] [D3, D7, D12, D13] [D4, D5, D6, D10, D14]

[2+, 3-] [4+, 0-] [3+, 2-]

- Repeat attribute selection for non-terminal leaves

Decision Tree Learning – Summary

- Decision Trees have a naturally interpretable representation.
  - Although for complex problems they can grow very large.
- They can be very fast to evaluate as variables are only included when they need to.
- The greedy heuristic used for training them do not necessarily yield optimal representations.
- In these examples they were discrete, but there are variants for continuous inputs.
- They are mostly used for classification, but there are regression trees also.
- They actually can overfit, pruning or early stopping is often employed to avoid this.
- There are robust algorithms and software (C4.5, ID3, etc) that are widely used.
Outline

1. Introduction
2. Markov Decision Processes
3. Reinforcement Learning
4. Summary
Introduction to Reinforcement Learning

- Remember:
  - In Supervised Learning agents learn to act given examples of correct choices.
- What if an agent is given rewards instead?
- Examples:
  - In a game of chess, the agent may be rewarded if it won.
  - A soccer playing agent may be rewarded if it scores a goal.
  - A helicopter acrobatics agent may be rewarded if it performs a loop.
- These are all examples of Reinforcement Learning.

Defining Reinforcement Learning

- How do we formally define this problem?
- An agent is given a sensory input consisting of:
  
  State \( s \in S \)
  Reward \( R \in \mathbb{R} \)
- It should pick an output
  
  Action \( a \in A \)
- What about utilizing supervised learning to learn \( f(s, a) = R \)?
- For any input state we could just pick the best action
  
  \[ a = \arg \max_a f(s, a) \]
Naive RL Agent

- First learn $R = f(s,a)$ from experience (regression problem)
- Always pick action $a = \arg \max_a f(s,a)$

An example...
- $S = \{\text{squares}\}$
- $A = \{N,W,S,E\}$
- Will this work?

- Reward will be zero for all actions in all states not adjacent to the two terminal states!

Markov Decision Processes

- The agent needs to think ahead!
- It should look for a good sequence of actions and rewards
- How do we formalize this?
- A Markov Decision Process is a framework for sequential decision making
- Cast the RL problem as learning in an MDP!
Markov Decision Processes - Actions

- But first we need some more definitions...
- We have:
  - State \( s \in S \)
  - Action \( a \in A \)
  - Reward \( R \in \mathbb{R} \)
- To capture the effect of actions, we need a state transition function
  \[
  P(s'|s, a)
  \] (1)
- MDPs are probabilistic models of the environment
- While there may be some uncertainty in the result of actions, in many applications it is fairly close to deterministic

Markov Decision Processes - Rewards

- What about the rewards?
- We define the reward for reaching a state \( s_i \) as \( R(s_i) \)
- To enable the agent to think ahead it must look at a sum of rewards over a sequence of states \( R(s_{i+1}), R(s_{i+2}), R(s_{i+2}), \ldots \)
- This can be formalized as the utility \( U \) for the sequence
  \[
  U = \sum_{t=0}^{\infty} \gamma^t R(s_t), \text{ where } 0 < \gamma < 1
  \] (2)
- Where \( \gamma \) is the discount factor making rewards far off into the future less valuable.
- A low \( \gamma \) makes the agent very short-sighted and greedy, while a gamma close to 1 makes it very patient.
Markov Decision Processes - The Policy Function

- We now have a utility function for a sequence of states
- ...but the sequence of states depends on the actions taken!
- We need one last concept, a **policy** function \( \pi(s) \) decides which action to take in each state

\[
a = \pi(s)
\]

(3)

- Clearly, a good policy function is what we set out to find

Markov Decision Processes - The Utility Function

- For simplicity, now assume that the state transitions are **deterministic**.
- Under a **particular** policy \( \pi \) we can now define a unique utility for each state as the sequence of states that follow under \( \pi \)

\[
U_\pi(s_i) = R(s_i) + \gamma R(s_{i+1}) + \gamma^2 R(s_{i+2})...
\]

(4)

- This can be **recursively** defined as

\[
U_\pi(s) = R(s) + \gamma U_\pi(s')
\]

(5)

- ...but how do we compute this recursive function?
By iterating that computation over states it will converge to the true $U_\pi(s)!$

Computing the utility function is called **policy evaluation**

With a **probabilistic** transition function it becomes

$$ U_\pi(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) U_\pi(s') $$  \hspace{1cm} (6)

But we wanted to find the **optimal policy** $\pi^*(s)$, which is the one that maximizes the utility

What about instead updating $U_\pi(s)$ with a **locally optimal** choice for each state?

$$ U_{\pi'}(s) = R(s) + \gamma \max_{a \in A} \sum_{s'} P(s'|s, a) U_\pi(s') $$  \hspace{1cm} (8)
Markov Decision Processes - Value Iteration

\[ U_{\pi'}(s) = R(s) + \gamma \max_{\alpha \in A} \sum_{s'} P(s'|s,a)U_{\pi}(s') \] (9)

- Like when computing the utility function of a particular policy, computing the utility of locally optimal choices will **converge** to that of the optimal policy \( U_{\pi^*}(s) \)
- Once we have \( U_{\pi^*}(s) \), we can easily extract \( \pi^*(s) \) by simply taking the locally best action in each state

\[ \pi^*(s) = \arg\max_{\alpha \in A} \sum_{s'} P(s'|s,a)U_{\pi^*}(s') \] (10)

- The algorithm is called **value iteration** and this approach to solving MDPs is often called *Dynamic Programming*.

Markov Decision Processes - An Example of VI

![Diagram showing a state transition graph with utility estimates and iterations required for different discount factors.](image)
Markov Decision Processes - Limitations

- Assumes we are given the reward function $R(s)$
- Assumes we are given the state transition function $P(s'|s,a)$
- Requires computation polynomial in the number of states
- The number of states grows exponentially with input dimension!
  - **The curse of dimensionality**
- Generally assumes discrete state and action spaces
  - Discretization
  - ..but useful grid sizes may result in a prohibitive number of states
Reinforcement Learning

- We have seen how rational (reward maximizing) agents interacting with an environment can be represented by an MDP.
- We have seen that such MDPs capture sequential decision making problems.
- We have seen that they can be solved by iterative algorithms (VI).
- But they assumed known state transitions and reward function.
- Reinforcement Learning is about learning some aspects of such a sequential decision problem.
- Usually the state transition function $P(s'|s,a)$ is unknown...
- Sometimes the state reward function $R(s)$ is unknown...
- Sometimes they both are!

Learning Models from Experience...

- In the simplest case where states and actions are discrete we can learn a model of these with a table for $R(s)$ and $P(s'|s,a)$.
- Given histories of state transitions $\ldots, s, R(s), a, s', \ldots$:
  1. We can update entries in the reward function table for $R(s)$ to be the average over experienced reward in each state $s$.
  2. We can update entries in the state-transition table for $P(s'|s,a)$ to be the frequency with which we end up in $s'$ when taking action $a$ in state $s$.
- Sometimes this experience is collected in an experiment first and given to the agent.
- If this information is sufficient to represent the unknown functions, we can just apply VI to find a policy for the agent.
Active Learning

- However, what happens when we let the agent loose on the environment with no prior knowledge?
- Even if it only has to learn the state transitions, having found one good path, why should it look for any other?
- Remember the VI update equation

$$U_{\pi'}(s) = R(s) + \gamma \max_{\alpha \in A} \sum_{s'} P(s'|s, a) U_{\pi}(s')$$  (11)

- A strictly utility-maximizing agent in a partially unknown environment is greedy!

Example of Greedy Active Learning

- Just doing VI while learning a function will get the agent stuck in a sub-optimal policy
If the agent had tried something new instead of only taking the action that seemed best right now, it may have found a better choice!

Clearly, there is some value in exploring unknown parts of the environment.

Such a better choice could then be used to increase the utility in the future.

But at the same time we may just find a poor choice, losing out on certain utility we could exploit now.

This is known as the exploration vs. exploitation trade-off.

---

Can we somehow calculate the value of exploration to compare it to exploitation?

Some questions that could be useful are:
- How uncertain are we of these actions?
- How much could we gain if we found a better alternative?
- The duration we could exploit a better alternative for...

It is possible to characterize this as a stochastic optimization problem, where we update probability distributions over all actions and rewards...

Warning: This is nearly always intractable to compute.

Exploration vs. Exploitation is a very general concept and widely regarded as very difficult to solve perfectly.
However, we can come up with heuristics that work rather well.

One particularly simple heuristic is to try an action randomly \( \epsilon \) of the time.

- Such approaches are usually called \( \epsilon \)-greedy.
- A better approach is to promote exploration in the utility function.

We rewrite the value iteration update equation

\[
U^+(s) = R(s) + \gamma \max_{a \in A} f\left( \sum_{s'} P(s'|s, a) U^+(s'), N(s, a) \right)
\] (12)

The exploration function \( f(u, n) \) should be increasing with \( u \) and decreasing with \( n \).

- \( N(s, a) \) is the number of times a state-action combination has been tried.

### Example of Exploration Function

- We rewrite the value iteration update equation

\[
U^+(s) = R(s) + \gamma \max_{a \in A} f\left( \sum_{s'} P(s'|s, a) U^+(s'), N(s, a) \right)
\] (13)

- Example exploration function

\[
f(u, n) = \begin{cases} 
R^+ & \text{if } n < N_e \\
u & \text{otherwise}
\end{cases}
\] (14)

- Where \( R^+ \) is some optimistic estimate of a possible reward and \( N_e \) is the number of times we want it to try for it.

- Note that the optimistic estimate will be propagated through \( U^+ \) so the agent can find an unknown state that is far away.
Comparison of Greedy and Non-Greedy RL

- We can see that the exploratory agent gets closer to the optimal policy and value functions while the greedy agent is stuck with the first decent choice it finds...

Model-Based Reinforcement Learning

- We have now seen how we can incorporate exploration to learn unknown parts of the environment.
- Sometimes the model of the transition function has a simple pattern and doesn’t depend explicit on every state.
- In our example when an agent goes north, the next state is most likely the state north of the current one, regardless of where we are.
- In dynamical systems like a car or rocket we often control accelerations which add to velocities, and are then added to position, regardless of where that position is.
- Taking a good model into account allows an agent to find a good policy with **a small number of interactions** with the environment.
Model-Free Reinforcement Learning

- Of course, finding a good representation for the model can be difficult
- The most common types of reinforcement learning algorithms are actually **model free**
- One such model-free algorithm is Q-learning
- Q-learning is arguably the most **well known** RL algorithm

Q-Learning

- Q-learning rests upon the observation that instead of calculating a utility function $U(s)$ over each state, we can calculate a utility function over each state-action combination $Q(s, a)$
- The recursive definition of value iteration then becomes
  \[ Q(s, a) = R(s) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a') \]  
  (15)

- To get rid of the transition probabilities, each time the agent passes a state we can nudge the Q-value towards the one observed
  \[ Q(s, a) \leftarrow Q(s, a) + \alpha (R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a)) \]  
  (16)

- This means that the Q-function can only be updated when the agent actually interacts with the environment
Q-Learning Summary

- It is very simple, having no model over the environment
  - It only needs a table of $Q(s,a)$ values!
- Once the $Q(s,a)$ function has converged, the optimal policy $\pi^*(s)$ is simply the action with highest utility in the table for each $s$
- Q-learning can also be combined with an exploration function
- Q-learning requires very little computational overhead per step
- Like all model-free methods it may require more interactions with the world than a model-based approach though
- If the agent is running in a computer, interaction may be cheap and the total amount of computation needed can be similar to model-based methods.

Generalization

- Remember some limitations of solving MDPs
  - The Curse of Dimensionality
  - Discretization of continuous state or action spaces
- In this chapter we have only talked about learning using simple tables
- One popular approach to attempt to tackle these problems is to use a supervised learning algorithm on the:
  - Q-function for a model-free approach
  - Transition, reward and/or utility-functions of a model-based approach
- If a learned function generalizes well one can get large gains in scalability
- Note that it may also lead to problems with convergence
Reinforcement Learning allows an agent to adapt to maximize Rewards in a potentially unknown environment.

To do that it needs to consider sequences of actions, formalized as an MDP.

Model-Free methods allow one to fore-go having to learn a model of the environment at the cost of requiring more experience.

By using a supervised learning algorithm instead of a table for Q(s,a), U(s) or the state-transition model one can speed up learning, especially in continuous spaces.

However, the curse of dimensionality can still be a problem:

"...Q-learning with inexpertly chosen input feature representation and/or inexperienced choice of function approximation can often lead to poor scaling with size of state space." - Satinder Singh