Review: Artificial Neural Network

- Artificial neural network (ANN) is composed of perceptrons modeled after neurons

- General network structure
  - One or multiple layers of perceptrons
  - Each perceptron takes raw input or output from perceptrons before
Review: Artificial Neural Network

⇒ More layers ⇒ more modeling power
  ⇒ E.g., single layer cannot express simple XOR
  ⇒ Two layers can address this

⇒ Can make perceptron differentiable
  ⇒ E.g., \( E(w) = \sum_j (y_j - g(w^T x_j))^2 \), \( w \leftarrow w - \alpha \frac{\partial E}{\partial w} \)
  ⇒ Can apply chain rule to do training (backpropagation)

⇒ The good
  ⇒ Approximates general functions
  ⇒ Easy to put together, more layers ⇒ more power

⇒ Problems
  ⇒ Can stuck in local optima
  ⇒ Training deep networks can cause instability
  ⇒ Difficult to analyze
  ⇒ Lots of choices (graph structure) – not clear how to choose!
Review: VC-Dimension

- **Definition:** the *cardinality* (size) of the *largest set of samples* that can be *shattered* by a given classifier.

- A *fixed* set of points is *shattered by a classifier* if *every subset* of the points can be *separated by the classifier* (with different parameters).

- **Important:** for a classifier to have VC dimension *at least* $n$, it only needs to shatter *ALL subsets* of a *SINGLE* set of $n$ points.

- **Example:** how many 2D points can be shattered using a perception (i.e., a half plane)
Review: Support Vector Machine

¬ The margin of linear classifier $w$ to a sample point $x$ is simply the distance between $x$ and the hyperplane defined by $w$.

¬ Can be written as $\frac{|x^T w + b|}{|w|}$ or $\frac{|x_i^T w|}{|w|}$ (one dimension higher).

¬ Maximize the margin: $w = \arg\max_w \min_i \gamma_i = \arg\max_w \min_i \frac{|x_i^T w|}{|w|}$.

¬ Equivalently: $w = \arg\min_w \left\{ \frac{1}{2} ||w||^2 : \forall (x_i, y_i), y_i x_i^T w \geq 1 \right\}$.

¬ This is a quadratic programming problem and can be solved in $O(n^3)$ time.

¬ Soft SVM penalizes for putting a data point on the “wrong side”.

$w = \arg\min_w \left\{ \frac{1}{2} ||w||^2 + C \sum \xi_i : \forall (x_i, y_i), y_i x_i^T w \geq 1 - \xi_i \right\}$.

¬ Handling non-linearity.
This Class

⇒ A little more on SVM
    ⇒ Lagrange multipliers and the dual perspective
⇒ Markov decision process (MDP)
⇒ Reinforcement learning
Lagrange Multipliers

Learned/used it before?

A general method for non-linear constrained optimization

Finds necessary conditions for optimizing $f(x)$ subject to $g_i(x) = 0$

To do so, build a Lagrangian (function)

$$\Lambda(x, \lambda) = f(x) - \lambda g(x)$$

Necessary conditions for optima must satisfy $\frac{\partial \Lambda}{\partial x_i} = 0$

An example: $\min (x_1^2 + x_2^2)$ subj. to $x_1 + x_2 = 1$

Now you try: $\min (x_1 + x_2)$ subj. to $x_1^2 + x_2^2 = 1$

When not all constraints are equalities, we get Lagrange multipliers with Karush-Kuhn-Tucker (KKT) condition

optimizing $f(x)$ subject to $g_i(x) = 0, h_j(x) \geq 0$

Similar Lagrangian, some constraints may not be “active”
Support Vector Machine: The Dual

\[ w = \arg\min_w \{ \frac{1}{2} ||w||^2 : \forall (x_i, y_i), y_i x_i^T w \geq 1 \} \]

we may apply the method of Lagrange multipliers with KKT condition

\[ \text{Lagrangian} \ (\lambda = (a_1, \ldots, a_n)) : \]

\[ \Lambda(w, \lambda) = \frac{1}{2} ||w||^2 - \sum_i a_i (y_i x_i^T w - 1) \]

\[ \text{Necessary conditions for optimal solution require} \ \frac{\partial \Lambda}{\partial w} = 0 \]

\[ w = \sum_j a_j y_j x_j, \text{ plugging back in } \Lambda(w, \lambda) \]

\[ \tilde{\Lambda}(\lambda) = \sum_i a_i - \frac{1}{2} \sum_i \sum_j a_i a_j y_i y_j (x_i \cdot x_j) \]

\[ \text{This is also a quadratic programming problem, can extend to kernel form} \]

\[ \tilde{\Lambda}(\lambda) = \sum_i a_i - \frac{1}{2} \sum_i \sum_j a_i a_j y_i y_j K(x_i, x_j) \]

\[ \text{This is the dual problem, meaningful if feature dimension is higher than number of data points} \]
Sequential Decision Making

Most of our decision making so far are episodic, i.e., making a single decision and not taking any action

- Search finds a path
- CSP solver finds a satisfactory assignment
- Classification (e.g., perceptron) finds a label

We have gotten close

- Sequential games requires making decisions depending on later outcomes
- In Bayesian networks, MCMC method does iterative sampling based on previous outcomes

Sequential decision making emphasizes taking action that affects later decisions

Markov decision process is one of the classic models
In a Markov decision process (MDP), an agent interacts with its environment through **probabilistic actions** and **deterministic observations**.

The agent’s goal is to decide the sequence of actions that yields the best payout (reward).

The process may be finite (ends in finite number of steps) or infinite (continuing forever).
Blackjack (twenty-one)

⇒ In blackjack, the goal is to get to as close to 21 as possible with going bust

⇒ You make decision as of whether to draw the next card
  ⇒ You get a card with certain probability (probabilistic action)
  ⇒ You can see what the next card is (fully observable)

⇒ This can be modeled as a MDP

⇒ If you are interested in reading further, check https://www.cs.rutgers.edu/~mlittman/courses(cps271/lect-14.ps}
Markov Decision Process: Definition

- A Markov decision process is defined by four main elements
  - \((S, A, P, R, s_0)\)
  - **States** \(S\) with a start state \(s_0\)
  - **Actions** \(A\). A state \(s\) may only have a limited subset of actions \(A(s) \subset A\)
  - **Transition** model \(T(s, a) = \{s'\}\) with probability \(P(s'|s, a)\)
  - **Reward function** \(R(s)\)

- What does Markov mean here?
  - Same as in Markov networks
  - 1-step dependency
  - Future outcome only depending on the current state/action
  - No residual memory of earlier state/actions given current state/action
  - A simplifying assumption that works well

- Solution is a **policy** \(\pi: S \rightarrow A\) that provides an action \(a\) for a given state \(s\)

Image source: P. Abbeel and D. Klein
A Grid World Example

- States – grid cells without obstacles
- Actions – \{left, right, up, down\}
- Transitions – for any action at a state, there is probability the agent ends up in any adjacent state
- Reward – pit (-1), diamond (1), or some small cost/reward per step

Image source: P. Abbeel and D. Klein
Transition Model and Goal

⇒ The transition is **probabilistic** and cannot be fully controlled by the agent – the agent can only pick the action.

⇒ The goal is to find an optimal policy that tells what action to take at a given state.
Optimal Policy for Different Step Rewards

- If the cost of taking an action is -0.04, then the optimal policy is
- For other $R(s)$ values
- Note the changes to policy as $R(s)$ changes
Solving MDP

The overall utility (or value) of a sequence of action can be written as

\[ V[s_0, s_1, s_2, \ldots] = R(s_0) + R(s_1) + R(s_2) + \ldots \]

Problem: if policy is infinite, so is \( U \)

To address this use *discounts*

Future rewards are discounted by a factor \( 0 < \gamma < 1 \)

\[ V[s_0, s_1, s_2, \ldots] = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots \]

Now the maximum possible total discounted reward is \( \frac{R_{\text{max}}}{1 - \gamma} \)
Bellman’s Equation

Given a policy \( \pi: S \to A \), we have:

\[
V^\pi(s) = \sum_{t=0}^{\infty} \gamma^t E[R(s_t)|s_0 = s]
\]

We may take out the first term

\[
V^\pi(s) = R(s) + \sum_{t=1}^{\infty} \gamma^t E[R(s_t)|s_0 = s]
\]

\[
= R(s) + \gamma \sum_{t=0}^{\infty} \gamma^t E[R(s_t)|s_0 \in T(s, \pi(s))]
\]

\[
= R(s) + \gamma \sum_{s' \in T(s, \pi(s))} P(s'|s, \pi(s)) \sum_{t=0}^{\infty} \gamma^t E[R(s_t)|s_0 = s']
\]

\[
= R(s) + \gamma \sum_{s' \in T(s, \pi(s))} P(s'|s, \pi(s)) V^\pi(s')
\]
Optimal Policy

Bellman showed that the **necessary and sufficient condition** for a policy to be optimal is

\[
V^\pi(s) = \max_{\pi} \left[ R(s) + \gamma \sum_{s' \in T(s, \pi(s))} P(s' | s, \pi(s)) V^\pi(s') \right]
\]

When a policy is optimal, it is usually written as \( \pi^* \)

From here we can solve MDP with
- **Value iteration**
- **Policy iteration**
Value Iteration

⇒ Value iteration is a iterative (greedy) update of $V$ using Bellman’s equation

$$V_{k+1}(s) = R(s) + \gamma \max_{a \in A(s)} \left[ \sum_{s' \in T(s,a)} P(s'|s,a)V_k(s') \right]$$

⇒ The update is greedy: always use the best action possible

⇒ The algorithm

⇒ Start with $V_0 \equiv 0$ (except terminal states), $k = 0$

⇒ Compute $V_{k+1}$ using (*) for all $s \in S$

⇒ If $|V_{k+1}(s) - V_k(s)| > \delta$ then $|V_{k+1}(s) - V_k(s)| = \delta$

⇒ Repeat until $|V_{k+1} - V_k| < \varepsilon$ for all $s \in S$ and for some fixed $\varepsilon > 0$

⇒ The algorithm converges reasonably well

⇒ Iterations till convergence $N = \left\lceil \frac{\log(\frac{2R_{\text{max}}}{\varepsilon(1-\gamma)})}{\log(1-\gamma)} \right\rceil$

⇒ Large $\gamma \rightarrow$ slow convergence; small $\gamma \rightarrow$ short horizon
Value Iteration Example ($\gamma = 1$)

$\Rightarrow V_0([4,3]) = 1, V_0([4,2]) = -1, 0$ for the rest

$\Rightarrow$ At $[3,3]$, compute $\sum_{s' \in T(s,a)} P(s'|s, a)V_0(s')$ for all actions

$\Rightarrow up: 0.1 \times 0 + 0.8 \times 0 + 0.1 \times 1 = 0.1$

$\Rightarrow right: 0.1 \times 0 + 0.8 \times 1 + 0.1 \times -1 = 0.7$

$\Rightarrow down: 0.1 \times -1 + 0.8 \times 0 + 0.1 \times 0 = -0.1$

$\Rightarrow left: 0.1 \times 0 + 0.8 \times 0 + 0.1 \times 0 = 0$

$\Rightarrow$ Best action is going right, $V_1([3,3]) = -0.04 + 0.7 = 0.66$

$\Rightarrow$ For all other $s \in S$, $V_1(s) = -0.04$

$\Rightarrow$ For $V_2$, at $[3,3]$, compute $\sum_{s' \in T(s,a)} P(s'|s, a)V_1(s')$

$\Rightarrow up: 0.1 \times -0.04 + 0.8 \times 0.66 + 0.1 \times 1 = 0.624$

$\Rightarrow right: 0.1 \times 0.66 + 0.8 \times 1 + 0.1 \times -1 = 0.766$

$\Rightarrow down: 0.1 \times -1 + 0.8 \times -0.04 + 0.1 \times -0.04 = -0.136$

$\Rightarrow left: 0.1 \times -0.04 + 0.8 \times -0.04 + 0.1 \times 0.66 = 0.03$

$\Rightarrow$ Best action is going right, $V_2([3,3]) = -0.04 + 0.766 = 0.726$

$\Rightarrow V_2([2,3]) = -0.04 + (0.2 \times -0.04 + 0.8 \times 0.66) = 0.48$

$\Rightarrow V_2([3,2]) = -0.04 + (0.1 \times -0.04 + 0.8 \times 0.66 + 0.1 \times -1) = 0.384$

$\Rightarrow$ For all other $s \in S, V_2(s) = -0.04 - 0.04 = -0.08$
Example Computation from Book

\(\Rightarrow \gamma = 1\)

\(\Rightarrow \epsilon = cR_{\text{max}}\)
Policy Iteration

⇒ An alternative method to value iteration is policy iteration
⇒ Starting with an arbitrary initial policy $\pi_0$
  ⇒ Repeat the following two steps
  ⇒ Policy evaluation: $V_{k+1}(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_k(s))V_k(s')$
  ⇒ Policy improvement: $\pi_{k+1}(s) = \arg\max_a \sum_{s'} P(s'|s, a)V_{k+1}(s')$
⇒ Stops when policy/value converges
Policy Iteration Example ($\gamma = 1$)

\(\Rightarrow\) To start, pick a random policy \(\pi_0\), e.g., all \textit{up} \uparrow

\(\Rightarrow\) Same \(V_0\) as before

\(\Rightarrow\) Compute the values, e.g.,

\[V_1([3,3]) = -0.04 + 0.1 \times 0 + 0.8 \times 0 + 0.1 \times 1 = 0.06\]

\[V_0([3,2]) = -0.04 + 0.1 \times 0 + 0.8 \times 0 + 0.1 \times -1 = -0.14\]

\[V_0([2,3]) = -0.04 + 0.1 \times 0 + 0.8 \times 0 + 0.1 \times 0 = -0.04\]

\(\Rightarrow\) ... 

\(\Rightarrow\) Compute updated policy \(\pi_1\)

\(\Rightarrow\) For \([3,3]\)

\[\sum_{s'} P(s'|s, \text{right})V^{\pi_k}(s') = 0.1 \times 0.06 + 0.8 \times 1 + 0.1 \times -1 = 0.706\]

\[\sum_{s'} P(s'|s, \text{up})V^{\pi_k}(s') = 0.1 \times -0.04 + 0.8 \times 0.06 + 0.1 \times 1 = 0.144\]

\[\sum_{s'} P(s'|s, \text{left})V^{\pi_k}(s') = 0.1 \times -0.14 + 0.8 \times -0.04 + 0.1 \times 0.06 = 0.014\]

\[\sum_{s'} P(s'|s, \text{down})V^{\pi_k}(s') = 0.1 \times 1 + 0.8 \times -0.14 + 0.1 \times -0.04 = 0.1\]

\(\Rightarrow\) So \(\pi_1([3,3]) = \text{right}\)
Reinforcement Learning

⇒ Markov decision process
  ⇒ \((S, A, P, R, s_0)\) are given
  ⇒ To solve, find policy \(\pi\) using
    ⇒ Value iteration
    ⇒ Policy iteration

⇒ Reinforcement learning is similar but
  ⇒ \(P\) and \(R\) are generally unknown
  ⇒ Must learn \(P, R\) (implicitly or explicitly) via exploration
  ⇒ Then must find policy \(\pi\) via exploitation
  ⇒ Generally a harder problem

⇒ Inspired by behavior psychology
  ⇒ It appears that humans also learn through trial and error
  ⇒ Through merely rewarding good actions and penalizing bad actions
An Example

Moving to a new apartment and go to work

Ideally, getting to office at about 9am

Day 1: getting up at 7 am, got to office at 8am – too early

Day 2: getting up at 8 am, too much traffic, got to office at 9:40 – too late
  Hopefully job is not lost because of this

Day 3: getting up at 7:15am, got to office at 8:30 – better, still too early

Day 4: getting up at 7:30am, got to office at 8:55 – pretty good

Day $n$: getting up at 7:35, get to office at 9:01, just before the boss does so
Reinforcement Learning: General Scheme

- In each step, take some action from the current state
- Observe the outcome
  - State is still fully observable
  - Some reward is associated with reaching the state
- Update internal representation regarding states/actions/rewards
- Restart if hitting a terminal state (e.g., getting stuck)
- Using what is learned so far to derive a policy
- Approaches
  - Model-based: try to learn $P$ and $R$, and then derive solution policy
  - Model-free: directly learn a policy (e.g., $Q$-learning)