Learning

What is Learning?

- "Learning denotes changes in a system that ... enable a system to do the same task more efficiently the next time." --Herbert Simon
- "Learning is constructing or modifying representations of what is being experienced." --Ryszard Michalski
- "Learning is making useful changes in our minds." --Marvin Minsky

What is Machine Learning?

There are two ways that a system can improve:

1. By acquiring new knowledge

   For example: acquiring new facts or skills

2. By adapting its behavior

   For example: solving problems more accurately or efficiently

Machine Learning

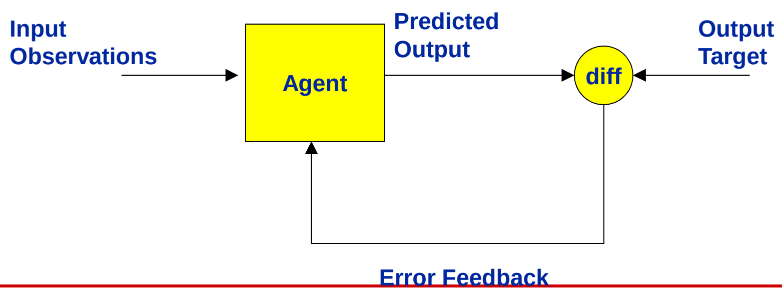
- Why is learning necessary?
  - learning is the hallmark of intelligence; a system that cannot learn is arguably not intelligent.
  - without learning, everything is new; a system that cannot learn is not efficient because it rederives each solution and repeatedly makes the same mistakes.
  - Can be used to understand and improve efficiency of human learning. For example, use to improve methods for teaching and tutoring people, as done in CAI -- Computer-aided instruction
  - Discover new things or structure that is unknown to humans. Example: Data mining
  - Fill in skeletal or incomplete specifications about a domain. Large, complex AI systems cannot be completely derived by hand and require dynamic updating to incorporate new information. Learning new characteristics expands the domain or expertise and lessens the "brittleness" of the system
- Why is learning possible?
  - because there are regularities in the world.

Major Paradigms of Machine Learning

- Rote Learning:
  - One-to-one mapping from inputs to stored representation.
  - "Learning by memorization."
  - Association-based storage and retrieval.
- Learning by being told.
- Induction:
  - Use specific examples to reach general conclusions
- Reinforcement:
  - Only feedback (positive or negative reward) given at end of a sequence of steps.
  - Requires assigning reward to steps by solving the credit assignment problem--which steps should receive credit or blame for a final result?
- Clustering:
  - Analogy.
  - Determine correspondence between two different representations
- Genetic Algorithms
- Discovery:
  - Unsupervised, specific goal not given

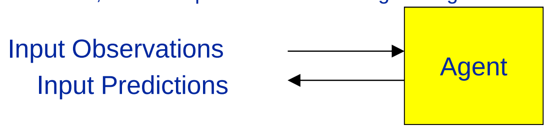
Supervised Learning

- For every input there is a target available
- The agent can generate an output based on input observations
- It also receives feedback from the environment to tell it how it ***should*** respond
- A signed error vector can thus be computed and used to update the agent's parameters
- Agent can thus learn a mapping between inputs and outputs



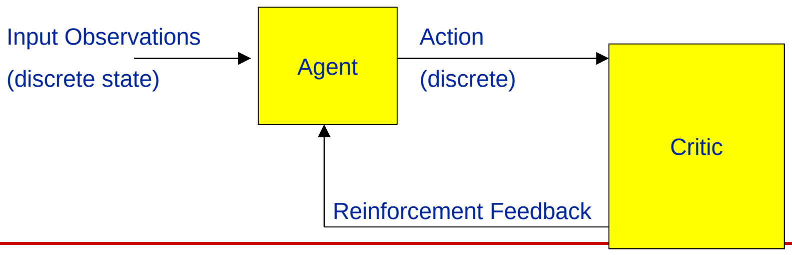
Unsupervised Learning

- There are only input observations
- The environment provides no feedback at all
- Opposite extreme to supervised learning
- All the agent can do is to model the input data
- It may be able to develop efficient internal representations of the input data
- These representations may provide useful input to other learning scheme that do interact with the environment
- Use it as a front end on supervised and reinforcement learning schemes
  - Vision, audition sparse efficient coding strategies



Reinforcement Learning

- For every input state the agent must choose an action
- It also receives feedback from the environment regarding how well it responded, but not what it should have done
- RL is learning with a **critic,** as opposed to learning with a **teacher**
- This scalar reinforcement can be used to indirectly update the agent's parameters
- Agent can thus learn a mapping between input state and actions
- Often we only get reward after a long sequence of actions.
- How do we solve the credit assignment problem?



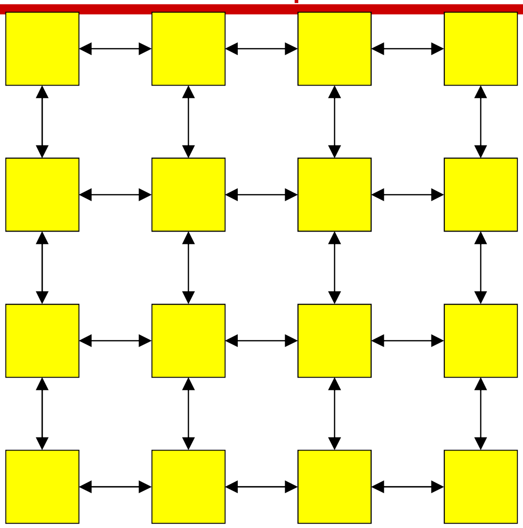
How to learn optimal behavior?

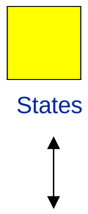
- Thorndike's law of effect (1911):

  If an action leads to a satisfactory state of affairs, then the tendency of the system to produce that particular action is strengthened or *reinforced*. Otherwise this tendency is weakened.

- Rewards
  - Rewards play an important role in reinforcement learning.
  - In fact, it is the only feedback the learner gets and it should be used to optimize action selection.

Grid World Example

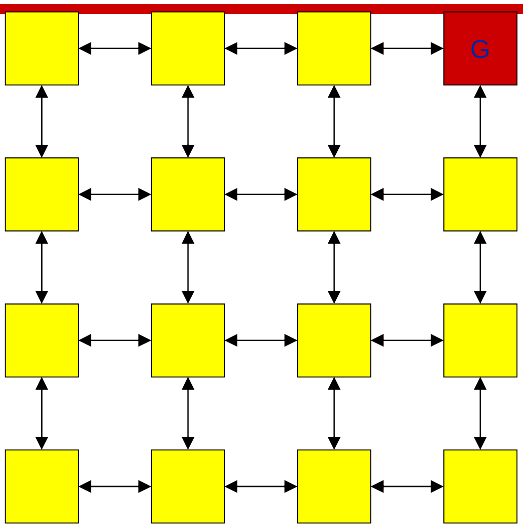




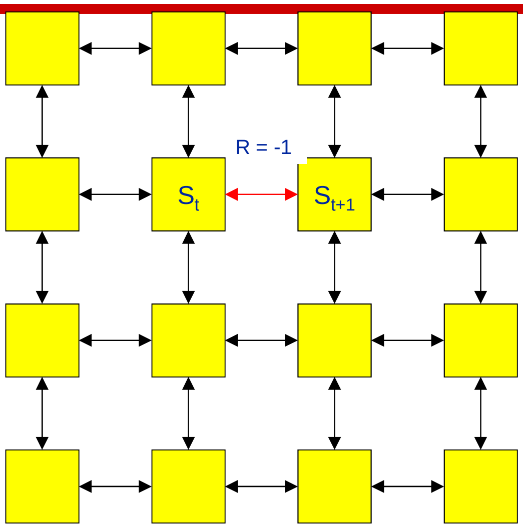
Consider the states as rooms

Actions correspond to passing through connecting doors

Goal State



Reward Function

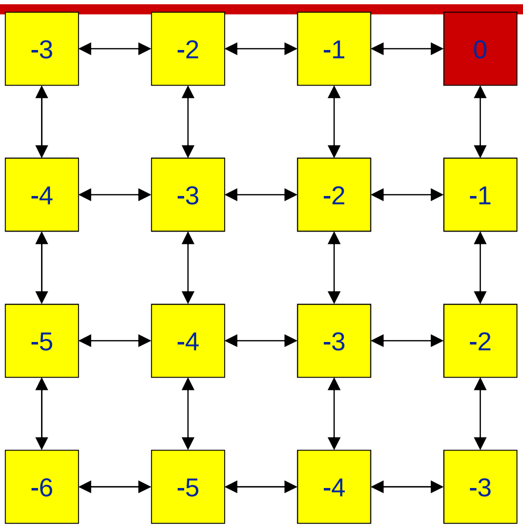


When we take an action, there is an associated cost called the reward function for the action.

Here we define a reward function R = -1 per transition which penalizes the length of the path taken.

This favors finding the shortest path from a given state to the goal.

Value Functions

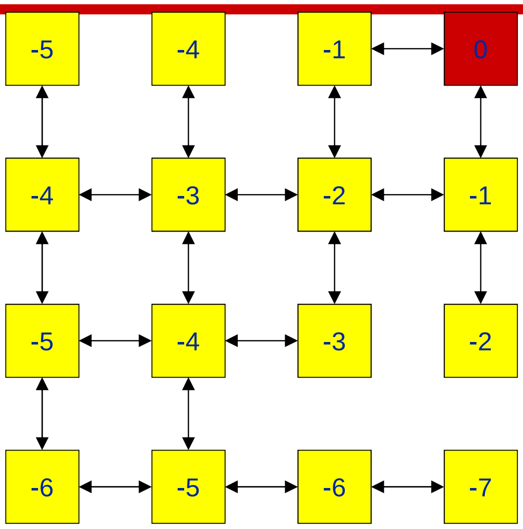


Value function represents the overall reward that can be accumulated moving from the given state to the goal.

Optimal values to reach the goal from each state are shown here .

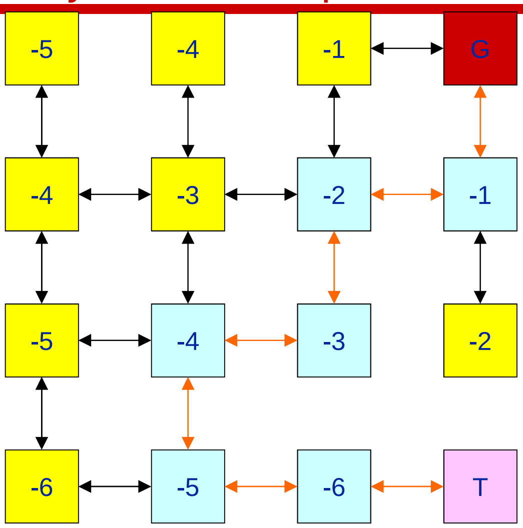
Here, the value function for each state is the shortest number of steps to the goal multiplied by -1

Value Functions are Problem Dependent



Changing the permitted transitions between the states will also change the optimal value functions

Policy to choose Optimal Path



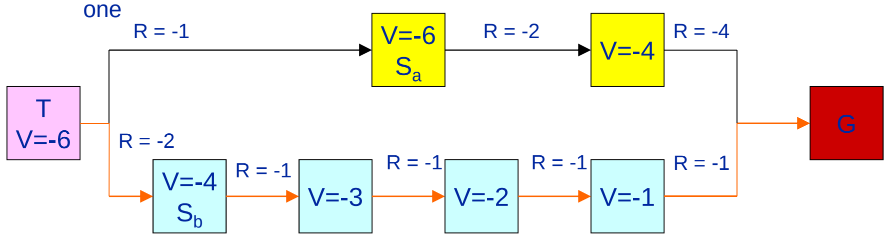
If we know the value functions, we can choose the optimal path from a target state T to the goal G.

We must use a valid action to move to the next state with the lowest value function (i.e. we must maximize the reward)

If we chose the action at random, the reward would be much less (i.e. it would take, on average, many more steps to get from T to G).

Reward and Value Functions

- **Reward** defines what is to be optimized
- It is the **local value** of the action from a given state
- The **value function** specifies the **global value** of a given state
- It takes into account future actions that will be carried out after the current one



- Observations:
  - The value function for state S_a is lower than for state S_b
  - However the local reward from T to state S_a is higher than for T to state S_b
  - In this case using the local reward would result in a sub-optimal path from T to G

Markov Decision Process (MDP)

MDP **M = < S, A, T, R >**
where

S: Finite set of states
$\{s_1, s_2, \ldots, s_n\}$

A: Finite set of actions $\{a_1, a_2, \ldots a_k\}$

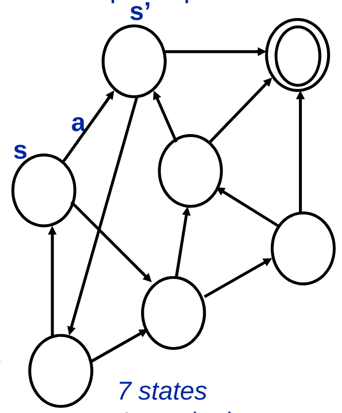
T(s, a, s'): probability to make a transition to state s' if action a is used in state s

R(s, a, s'): reward for making a transition from state *s* to state *s'* by doing action *a* (also possible: *R(s,a)* and *R(s)*)

step = transition from one state to another

$$s_t \xrightarrow{}_{a_t, r_t} s_{t+1}$$

Graphical representation



7 states
1 terminal state
2 actions

Some Properties

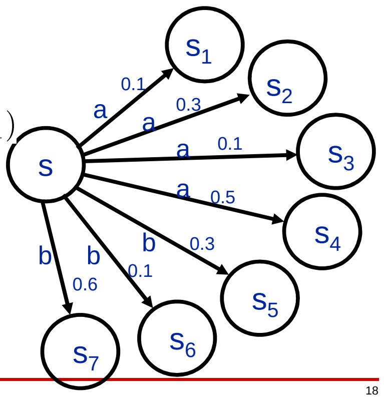
The Markov Property: The current state and action give all possible information for predicting to which next state the agent will step. It does not matter how the agent arrived at its current state. (otherwise partially observable, compare chess to poker):

$$T(s_{t+1}|s_t, a_t) = T(s_{t+1}|s_t, a_t, \ldots, s_1, a_1)$$

$$\forall a: \; 0 \leq T(s, a, s') \leq 1$$

$$\forall a: \; \sum_{s'} T(s, a, s') = 1$$

- States can be terminal (absorbing). Chains of steps will not be continued. Modelled as T(s,a,s)=1 (for all a) and R(s,a,s)=0.
- γ is the discount factor.



Policies, Rewards and Goals

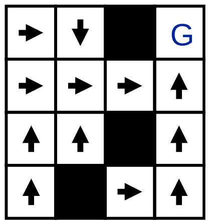
- A deterministic policy π selects an action as a function of the current state $a_t = \pi(s_t)$
- The goal of an agent is to maximize some performance criterion. Among several, the discounted infinite horizon optimality criterion is often used:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

Discount factor γ weighs future rewards.

- $$\pi^* = \arg\max_\pi E\left(\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t), s_{t+1}) | s_0 = s\right)$$

- There are $|A|^{|S|}$ policies. How to find π^*?



Example policy

Value Functions

- A value function (or: state utility function) represents an estimation of the expected sum of future rewards:

$$V^\pi(s) = E_\pi\left(r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s\right)$$
$$= E_\pi\left(r_t + \gamma V^\pi(s_{t+1}) | s_t = s\right)$$
$$= \sum_{s'} T(s, \pi(s), s')\left(R(s, a, s') + \gamma V^\pi(s')\right)$$

- A Q-function (or: state-action utility function) represents the value for selecting an action in a particular state:

$$Q^\pi(s_t, a_t) = \sum_{s_{t+1} \in S} T(s_t, a_t, s_{t+1})\left(R(s_t, a_t, s_{t+1}) + \gamma V^\pi(s_{t+1})\right)$$

5	6	■	10
6	7	8	9
5	6	■	8
4	■	6	7

Example value function
(in a deterministic environment):

Bellman Equations

- The optimal value functions satisfy the following recursive Bellman optimality equations [Bellman 1957]

$$V^*(s) = \max_a E\Big(r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\Big)$$
$$= \max_a \sum_{s'} T(s,a,s')\Big(R(s,a,s') + \gamma V^*(s')\Big)$$

- For Q-functions, the following equations hold:

$$Q^*(s,a) = E\Big(r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1},a') | s_t = s, a_t = a\Big)$$
$$= \sum_{s'} T(s,a,s')\Big(R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\Big)$$

- The relationship between Q* and V* is:

$$V^*(s) = \max_{a \in A} Q^*(s,a)$$

- Remarks:
  - V* and Q* are uniquely determined
  - π is not necessarily unique; many problems have multiple optimal policies.

Problem Setup and Possible Solutions

- When faced with a problem domain that is modeled as an MDP M = < S, A, T, R >, we have two cases:

1) Decision-theoretic planning (DTP)

If one has full knowledge about T and R, the setting is called model-based, and one can use (offline) dynamic programming techniques to compute optimal value functions and policies.

2) Reinforcement Learning (RL)

If one does not know T and R, the setting is called model-free, and one has to interact (online) with the environment to get learning samples. This exploration enables one to learn similar value functions and policies as in the model-based setting.

- Models (T and R) are often not available to the learner, but sometimes they can be learned (mixing both settings) (to which setting belongs chess? or robot navigation?)

Policy Evaluation

- If we fix a policy π, then we can compute the exact value of a particular state. This amounts to passive learning, or equivalently, to simulating a Markov system.
- The exact value of each state is determined by a set of equations (n linear equations, n unknowns → exactly one solution)
- Take, for terminal states: $V^\pi(s) = 0$

  And for all others:

$$V^\pi(s) = \sum_{s'} T(s,\pi(s),s')\Big(R(s,\pi(s),s') + \gamma V^\pi(s')\Big)$$

- First method: solve the equations (e.g. Gaussian elimination):

$$V^\pi(s_1) = \sum_{s'} T(s_1,\pi(s_1),s')\Big(R(s_1,\pi(s_1),s') + \gamma V^\pi(s')\Big)$$
$$V^\pi(s_2) = \sum_{s'} T(s_2,\pi(s_2),s')\Big(R(s_2,\pi(s_2),s') + \gamma V^\pi(s')\Big)$$
$$\ldots = \ldots$$
$$V^\pi(s_n) = \sum_{s'} T(s_n,\pi(s_n),s')\Big(R(s_n,\pi(s_n),s') + \gamma V^\pi(s')\Big)$$

- Second method: iterative procedure. Start with V(s)=0 for all states and repeat:

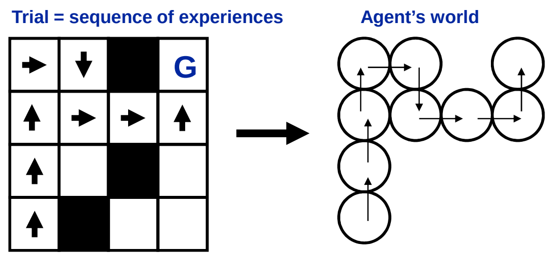
$$V^\pi(s) = \sum_{s'} T(s,\pi(s),s')\Big(R(s,\pi(s),s') + \gamma V(s')\Big)$$

a large number of times for all non-terminal states

Reinforcement Learning

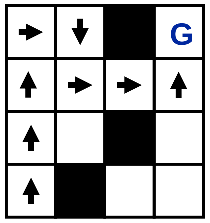
- RL: (T, R) not given, must be determined
- An RL agent learns a subjective (local) view of the world by interaction with the environment

- We need a policy, which is tested to find a new policy
- Exploration of the state space needed



Sampling: Exploration vs Exploitation

- Instead of backups through a model (as in DP), RL uses actual experience to perform sample backups.
- Exploitation: the agent should choose the best current action to maximize reward intake
- Exploration: the agent should try out new (unexplored) actions to find possibly even better options, and to visit unexplored parts of the state space
- Many types of (un)informed exploration. Often ε-greedy or Boltzmann



$$\frac{e^{-Q(s,a)/T}}{\sum_{a'} e^{-Q(s,a')/T}}$$

A generic model-free algorithm

```
For each episode do
    t := 0
    s_t ∈ S is initialized as the starting state
    repeat
        choose an action a_t
        perform a_t in the environment
        observe the new state s_{t+1} and receive reward r_t
        update T, R, Q and/or V using the
                experience < s_t, a_t, r_t, s_{t+1} >
        s_t := s_{t+1}
    until s_t is a goal state or maximum #steps reached
```

Explore/exploit

Actual experience (sample)

learning

Algorithms: Monte Carlo, TD-learning, Q-learning

Monte Carlo Sampling

- Determine for each state *s* in an epoch *k* the reward-to-go, which is the sum of all rewards in that epoch from the first moment that state is being visited in that epoch, until the epoch ends (denoted z_k).
- Then an estimate of a state's value is the average of all rewards-to-go of all the times it was visited in an epoch.
- Can be implemented incrementally as

$$V(s) = \frac{\sum_{i=1}^{k+1} z_i}{k+1} = \left(\frac{k}{k+1}\right) \cdot \left(\frac{\sum_{i=1}^{k} z_i}{k}\right) + \frac{z_{k+1}}{k+1}$$

- Advantage: unbiased expected value of total return
- Disadvantage: converges very slowly

Temporal-Difference Learning

- Instead of using the complete future state-chain for updating the value of a state, we can just use the next state
- This idea forms the core of temporal-difference learning
- We do it often in real life. Using estimates to compute other estimates. This is called bootstrapping. The idea is to shift an estimate (e.g. V(s)) towards the desired direction.
- For each step from state s to state s' in an epoch do
  - If s' is terminal: $V(s) := V(s) + \alpha\Big(R(s,-,s') - V(s)\Big)$
  - If s' is non-terminal: $V(s) := V(s) + \alpha\Big(R(s,-,s') + \gamma V(s') - V(s)\Big)$
  - Here α is the learning rate
- For a fixed α learning does not converge fully, but if α becomes smaller with the number of times a state s is visited, (and all states are visited infinitely often), this method will converge to the real value of states.

Q-learning

- Q-learning [Watkins, 1989] changes a single Q-value based on the experience $\langle s_t, s_t, r_t, s_{t+1}\rangle$

$$Q(s_t,a_t) := Q(s_t,a_t) + \alpha\Big(r_t + \gamma V(s_{t+1}) - Q(s_t,a_t)\Big)$$

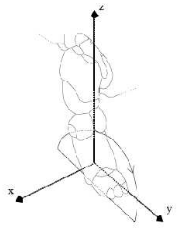
where $V(s) = \max_a Q(s,a)$

- Most often used method in model-free RL
- Simple to implement
- Off-policy (needed for various extensions such as HRL)
- Converges to the optimal Q-function if all state-action pairs are tried infinitely often (and the learning rate α decreases)
- Disadvantage: can take some time for rewards to propagate
- SARSA differs from Q-learning by taking $V(s) = Q(s, \pi(s))$ (on-policy)

RL for Quadruped Locomotion [Kohl04]

- Simple Policy-Gradient Example
- Optimize Gait for Sony-Aibo Robot

- Use Parameterized Policy
  - 12 Parameters
    - Front + rear locus (height, x-pos, y-pos)
    - Height of the front and the rear of the body



Quadruped Locomotion

- Policy: No notion of state – open loop control!
- Start with initial Policy $\{\theta_1, \ldots, \theta_N\}$
- Generate t = 15 random policies R_i

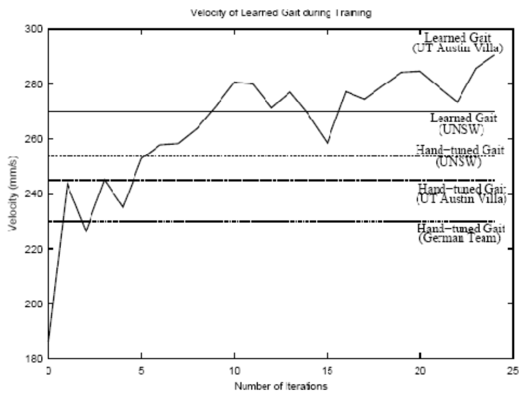
$$R_i = \{\theta_1 + \Delta_1, \ldots, \theta_N + \Delta_N\}$$

  - Δ_j is $+\epsilon_j$, 0, or $-\epsilon_j$
  - Evaluate Value of each policy on the real robot
  - Estimate gradient for each parameter
  - Update policy into the direction of the gradient

Quadruped Locomotion

- Estimation of the Walking Speed of a policy
  - Automated process of the Aibos
  - Each Policy is evaluated 3 times
  - One Iteration (3 x 15 evaluations) takes 7.5 minutes

Quadruped Gait: Results



Pegasus [Ng00]

- Policy Gradient Algorithms:
  - ↗ Use finite time horizon, evaluate Value
  - ↗ Value of a policy in a stochastic environment is hard to estimate
    - => Stochastic Optimization Process
- PEGASUS:
  - ↗ For all policy evaluation trials use fixed set of start states (scenarios)
  - ↗ Use „fixed randomization“ for policy evaluation
    - Only works for simulations!
  - ↗ The same conditions for each evaluation trial
  - ↗ => Deterministic Optimization Process!
    - Can be solved by any optimization method
    - Commonly Used: Gradient Ascent, Random Hill Climbing

Autonomous Helicopter Flight [Ng04a, Ng04b]

- Autonomously learn to fly an unmanned helicopter
  - ↗ 70000 $ => Catastrophic Exploration!
- Learn Dynamics from the observation of a Human pilot

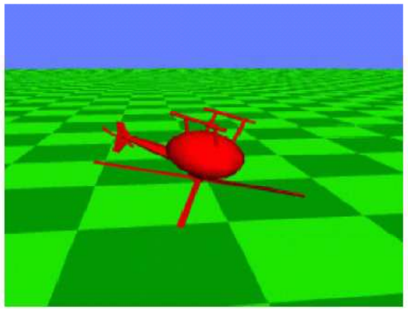
- Use PEGASUS to:
  - ↗ Learn to Hover
  - ↗ Learn to fly complex maneuvers
  - ↗ Inverted Helicopter flight

Helicopter Flight: Model Identification

- 12 dimensional state space
  - ↗ World Coordinates (Position + Rotation) + Velocities
- 4-dimensional actions
  - ↗ 2 rotor-plane pitch
  - ↗ Rotor blade tilt
  - ↗ Tail rotor tilt
- Actions are selected every 20 ms

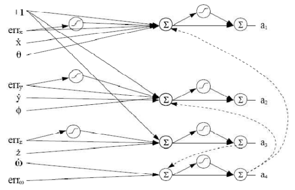
Helicopter Flight: Model Identification

- Human pilot flies helicopter, data is recorded
  - 391s training data
  - reduced to 8 dimensions (position can be estimated from velocities)
- Learn transition probabilities $P(s_{t+1}|s_t, a_t)$
  - supervised learning with locally weighted linear regression
  - Model Gaussian noise for stochastic model
- Implemented a simulator for model validation



Helicopter Flight: Hover Control

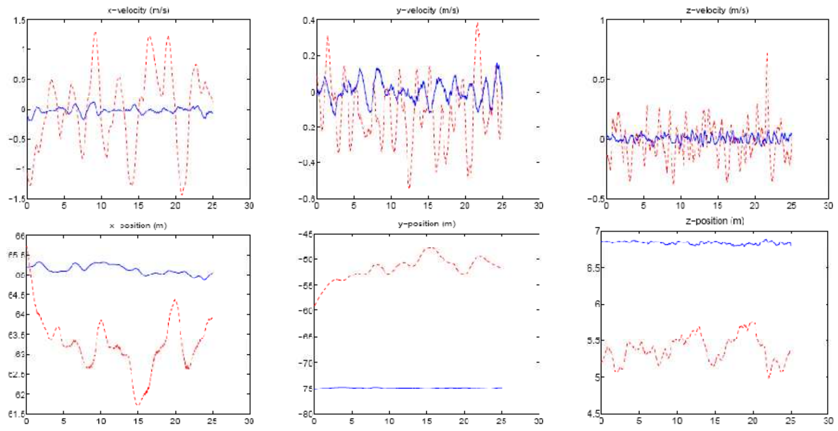
- Desired hovering position : $(x^*, y^*, z^*, \omega^*)$
- Very Simple Policy Class



  - Edges are optained by human prior knowledge
  - Learns more or less linear gains of the controller
- Quadratic Reward Function:
  - punishment for deviation of desired position and orientation

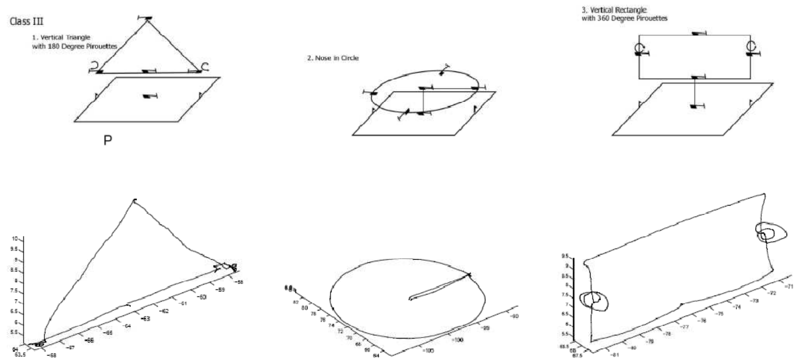
Helicopter Flight: Hover Control

- Results:
  - Better performance than Human Expert (red)



Helicopter Flight: Flying maneuvers

- Fly 3 manouvers from the most difficult RC helicopter competition class
- Trajectory Following:
  - punish distance from projected point on trajectory
  - Additional reward for making progress along the trajectory



Class III
1. Vertical Triangle with 180 Degree Pirouettes
2. Nose in Circle
3. Vertical Rectangle with 360 Degree Pirouettes
P

Helicopter Flight: Inverse Flight

- Very difficult for humans
  - Unstable!
- Recollect data for inverse flight
  - Use same methods as before
- Learned in 4 days!
  - from data collection to flight experiment
- Stable inverted flight controller
  - sustained position

High Speed Obstacle Avoidance [Michels05]

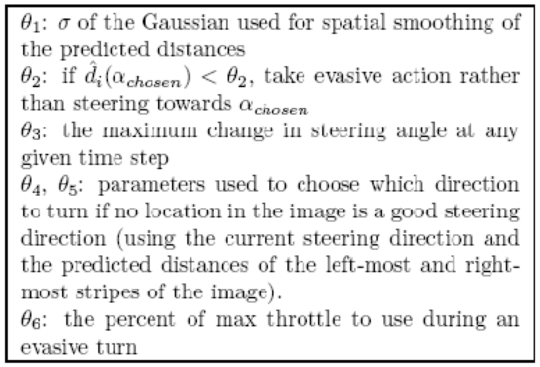
- Obstacle Avoidance with RC car in unstructured Environments
- Estimate depth information from monocular cues
- Learn controller with PEGASUS for obstacle avoidance
  - Graphical Simulation : Does it work in the real environment?

Estimate Depths Information:

- Supervised Learning
  - Divide image into 16 horizontal stripes
    - Use features of the strip and the neighbored strips as input vectors.
  - Target Values (shortest distance within a strip) either from simulation or laser range finders
  - Linear Regression
- Output of the vision system
  - α_{chosen} angle of the strip with the largest distance
  - Distance of the strip

Obstacle Avoidance: Control

- Policy: 6 Parameters
  - Again, a very simple policy is used



θ_1: σ of the Gaussian used for spatial smoothing of the predicted distances
θ_2: if $\hat{d}_i(\alpha_{chosen}) < \theta_2$, take evasive action rather than steering towards α_{chosen}
θ_3: the maximum change in steering angle at any given time step
θ_4, θ_5: parameters used to choose which direction to turn if no location in the image is a good steering direction (using the current steering direction and the predicted distances of the left-most and right-most stripes of the image).
θ_6: the percent of max throttle to use during an evasive turn

- Reward:
  - Deviation of the desired speed, Number of crashes

Obstacle Avoidance: Results

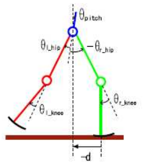
- Using a graphical simulation to train the vision system also works for outdoor environments

Table 1. Real driving tests under different terrains

Terrain	Obstacle Density	Obstacle Type	Ground	Mean Time (sec)	Max Time (sec)	Average Speed (m/s)
1	High	Sculptures, Trees, Bushes, Rocks	Uneven Ground Rocks, Leaves	19	24	4
2	Low	Trees, man-made benches	Even, with Leaves	40	80	6
3	Medium	Trees	Rough with Leaves	80	>200	2
4	Medium	Uniform Trees, man-made benches	tiled concrete	40	70	5

RL for Biped Robots

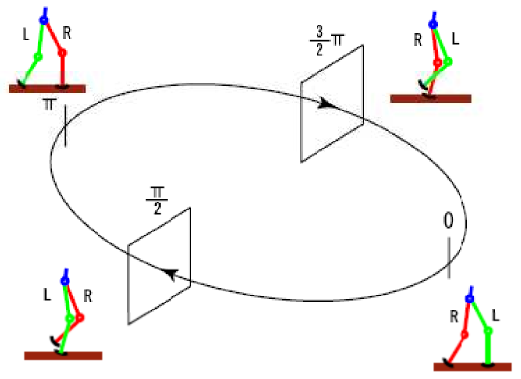
- Often used only for simplified planar models



  - Poincare-Map based RL [Morimoto04]
  - Dynamic Planning [Stilman05]
- Other Examples for RL in real robots:
  - Strongly Simplify the Problem: [Zhou03]

Poincare Map-Based RL

- Improve walking controllers with RL
- Poincare map: Intersection-points of an n-dimensional trajectory with an (n-1) dimensional Hyperplane
  - Predict the state of the biped a half cycle ahead at the phases :



Poincare Map

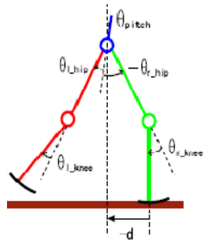
- Learn Mapping:

$$\hat{x}_{\frac{3\pi}{2}} = \hat{f}_1(x_{\frac{\pi}{2}}, u_{\frac{\pi}{2}}; w_1^m),$$
$$\hat{x}_{\frac{\pi}{2}} = \hat{f}_2(x_{\frac{3\pi}{2}}, u_{\frac{3\pi}{2}}; w_2^m),$$

  - Input Space : x = (d, d‘)
    – Distance between stance foot and body
  - Action Space :

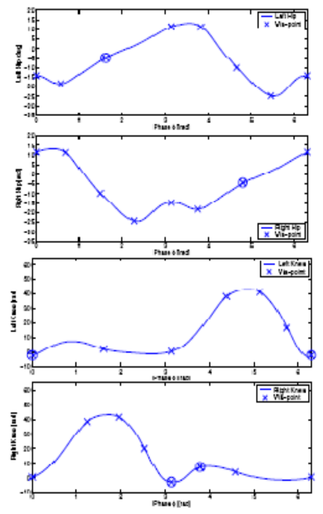
$$u = \theta^{act}(x) = (\theta^{act}_{hip}, \theta^{act}_{knee})$$

    – Modulate Via-Points of the joint trajectories



- Function Approximator: Receptive Field Locally Weighted Regression (RFWR) with a fixed grid

Via Points



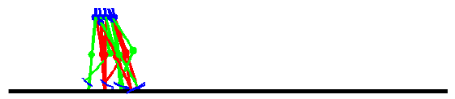
- Nominal Trajectories from human walking patterns
- Control output $\mathbf{u} = \theta^{act}(\mathbf{x}) = (\theta^{act}_{hip}, \theta^{act}_{knee}$ is used to modulate via points with a circle
  - Hand selected via-points
  - Increment via-points of one joint by the same amount

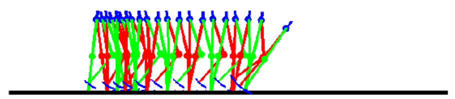
Learning the Value function

- Reward Function:
  - 0.1 if height of the robot > 0.35m
  - -0.1 else
- Standard SemiMDP update rules
  - Only need to learn the value function for $\phi = \frac{\pi}{2}$ and $\phi = \frac{3\pi}{2}$
- Model-Based Actor-Critic Approach
  - A … Actor $\mathbf{u}(t) = \mathbf{A}(\mathbf{x}(t); \mathbf{w}^a) + \sigma \mathbf{n}(t)$
  - Update Rule: $\mathbf{A}(\mathbf{x}; \mathbf{w}^a) \leftarrow \mathbf{A}(\mathbf{x}; \mathbf{w}^a) + \alpha \frac{\partial V(\mathbf{x})}{\partial \mathbf{x}} \frac{\partial \mathbf{f}(\mathbf{x}, \mathbf{u})}{\partial \mathbf{u}}$

Results:

- Stable walking performance after 80 trials
  - Beginning of Learning





  - End of Learning