AI final exam

Fall 2021

Haoming(Hammond) Liu

Introduction

- **Intelligent agent**: Perceive its environment through sensors, Act upon that environment through actuators
- **Environment**: external setting (e.g. maze)
- **Action selection**: percept => [?] => actions

- **Reflex Agent:** condition-action rule, respond to percepts
  - does not need to be rational, can be if complete env knowledge
- **Model based:** internal state of world => how world evolves, what my actions do
  - State: current conception; model: how the next state depend on curr
- **Goal based:** choose actions that maximize the goals (what'll it be if I ...)
  - involves consideration of the future (diff bw reflex)
- **Utility based:** maps a state onto associated degree of happiness
  - whether one world state is preferred to another
  - good for: conflicting goals or multiple goals

Informed & Uninformed

- **Uninformed Search** (always complete)
  - no additional information about states other than prob def
  - simply generate successors and discriminate a goal / non-goal state
  - BFS: queue; DFS: stack (visited: closed; not yet: open)
- **Informed Search** (Heuristic Search) (may not be complete)
  - know whether one non-goal state is “more promising”
  - function for estimation => how close to the goal state, more efficient
  - Greedy Best First: solely based on the heuristic function f(n) = h(n), incomplete
  - **A∗ search** (most widely known form of best first search)
    - g: the cost to reach the node (start => curr)
    - h: the cost to get from the node to the goal (curr => goal)
    - f = g + h (estimated cost of the cheapest solution through)
    - both complete and optimal (admissible hur => never overestimate)
    - Uniform-cost-search: g only
    - Open list: generated nodes; Closed list: expanded nodes
    - Tree: optimal; graph: consistent

A* Search, Hill Climbing, Completeness

Function *A** SEARCH(*problem*) **returns** a solution or failure
Put first node *s* on OPEN
if OPEN *is empty* **then**
  exit with failure
end
Remove from OPEN and place on CLOSED a node *n* for which *f* is minimum
if *n is a goal node* **then**
  exit successfully with the solution obtained by
  tracing back the pointers from *n* to *s*
end
(see next slide)

Function *A** SEARCH(*problem*) **returns** a solution or failure
(see previous slide)
else
  Expand *n*, generating all its successors, and attach to them pointers back to *n*.
  for every *successor* n' *of* *n* **do**
    if n' *is not already on* OPEN *or* CLOSED **then**
      estimate $h(n')$ (estimate of the cost of the best path from n' to some goal node) and calculate $f(n') = g(n') + h(n')$ where $g(n') = g(n) + c(n, n')$ and $g(s) = 0$
    end
    if n' *is already on* OPEN *or* CLOSED **then**
      direct its pointers along the path yielding the lowest $g(n')$
    end
    if n' *required pointer adjustment and was found on* CLOSED **then**
      reopen it
    end
  end
end
Go to step 2.

- **Hill climbing**
  - always go uphill, terminate at a “peak” (greedy local search)
  - Stochastic Hill Climbing: random, with p => vary steepness
  - First-choice Hill Climbing: until a better one
  - Random restart
- **Completeness**: if the algo terminates with a solution when one exists

Propositional Logic

- Ontological: facts only; epistemological: true/false/unknown
- **Syntax**
  - Logical constants: True / False
  - Proposition symbols: P, Q, etc.
  - Logical connective: [$\wedge$: conj], [$\vee$: disj], [$\Leftrightarrow$: equivalent], [$\Rightarrow$: imply], [$\neg$: not]
- **Semantics**: the interpretation of symbols, constants, logical connectives

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
False	False	True	False	False	True	True
False	True	True	False	True	True	False
True	False	False	False	True	False	False
True	True	False	True	True	True	True

Conjunctive Normal Form (CNF)

- To convert a proposition to CNF, we rely on the following 4 steps
  - Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
  - Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$
  - CNF requires the negation to only appears within each literal, so we move $\neg$ inwards by repeating the equivalences

$$\neg(\neg\alpha) \equiv \alpha \quad \text{double negation elimination}$$

$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta), \quad \text{De Morgan}$$

$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta), \quad \text{De Morgan}$$

  - Once we have a sentence containing only conjunctions and disjunctions, we apply the distributivity law, distributing $\vee$ over $\wedge$ wherever possible.

$$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$$

Horn & definite clauses & Chaining

- **Definite clause**: a disjunction of literals of which exactly one is positive
- **Horn**: a disjunction of literals of which at most one is positive
  - Closed under resolution (resolve two Horn => get back a Horn)
- **KBs with only Definite clauses are interesting**
  - definite clause => implication e.g. $(\neg L1 \vee \neg Breeze \vee B1)$ to $(L1 \wedge Breeze) \Rightarrow B1$
    - Premise is called body; conclusion is called head
  - Inference with Horn clauses: forward & backward chaining algorithm
  - entailment with Horn clauses: linear time of KB size
- **Forward Chaining:** Data-driven
  - if a single proposition symbol q is entailed by a KB with definite clauses
  - begins from the known facts (positive literals)
  - all the premises are known => conclusion can be added
- **Backward Chaining**: Goal-directed
  - If Query is true => done
  - Else => All implications whose conclusion is q => recursive

```
Function PL_ForwardChaining(KB, query q):
    input  : KB (initial state of the env.)
             q the query (propositional symbol)
    count (count[c] = num. of symbols in c's premises)
    inferred (inferred[s] is set to false for all symbols)
    agenda (queue of symbols known to be true in KB)
    while agenda is not empty do
        p ← Pop(agenda)
        if p = q then
            return True
        end
        if inferred[p] is False then
            inferred[p] ← True
            for each clause c in KB where p is in c.Premises do
                decrement count[c]
                if count[c] = 0 then
                    add c.Conclusion to agenda
                end
            end
        end
    end
    return false
```

Algorithm 2: Propositional Forward Chaining

PL Resolution Rule & Notions

$$\frac{a \vee \beta, \neg\beta \vee \gamma}{a \vee \gamma}$$

- Disjunction with p and ¬p => merge (a simple inference rule) exponential complexity
  - yields a complete inference algorithm with a complete search method
  - generate complete inference procedures
- Resolution algo (above): to show KB |= a, we show (KB^¬a) is unsatisfiable
  - (KB^¬a) => CNF => add new clauses for all pairs
  - (a) new clauses a subset of the original clauses: not entail
  - (b) unsatisfiable => empty clause: entail (Ground Resolution Theorem)

- **Entailment**: KB entails a (KB |= a)
- **Inference**: generate new sentences entailed by existing ones
  - Given KB, generate new sentences entailed by KB
  - Tell whether another sentence is entailed by a KB
  - inference procedure ı : a is derived from KB by ı (KB |-ı a)
- **Soundness / Truth-preserving** (inference rule):
  - An inference procedure that generates **only entailed sentences**
- **Completeness** (inference rule):
  - If it can derive (i.e. find a proof for) **any** sentence that is entailed

Resolution algorithm

input : knowledge base KB, propositional sentence α, the query $KN \models \alpha$

clauses $\leftarrow$ the set of clauses in the CNF representation of $KB \wedge \neg\alpha$

new $\leftarrow$ {};

while *(1)* **do**

  for *each pair of clauses* C_i, C_j *in* `clauses` **do**

    resolvents $\leftarrow$ `PL-Resolve`(C_i, C_j)

    if `resovents` *contains the empty clause* **then**

      `return` true

    end

    new $\leftarrow$ new $\cup$ resolvents

  end

  if `new` $\subseteq$ `clauses` **then**

    `return` false

  end

end

Algorithm 1: Propositional resolution algorithm

First Order Logic (FOL)

- first order predicate calculus with equality
- Ontological: facts, objects, relations; Epistemological: True / False / Unknown
- **Syntax**
  - Logical connectives, constant symbols, variable symbols, predicate symbols, function symbols, quantifiers
  - **Object**: constant symbols [John] (same object "=")
  - **Relations**: predicate symbols [Brother], takes arg(s), return a Boolean value
  - Functions: function symbols [LeftLeg], takes a from D, return a' in D
  - takes >=0 args from domain, returns another arg in domain
  - Ω is the domain of the variables (set of all constants)
  - King John's left leg => LeftLeg(John)
- **Semantics**: allow us to explicitly represent objects and relationships among object
- **Quantifiers**: express properties of entire collections of objects
  - For all: extended interpretations, makes statements about every objs
  - Exists: some object in the universe without naming it
  - Multiple quantifiers: the order is important

UI & EI & Herband & Turing & Church

- Also known as quantifier elimination
- Reduce first order inference to Propositional inference (propositionalization)

- **UI**: we can infer by substituting a ground term for the variable
  - Subst(θ, α), substitution θ to the sentence α [∀ v,α => Subst({v/g},α)]
  - Can be applied multiple times
- **EI**: the variable is replaced by a single new constant symbol
  - Subst({v/k},α), k that does not appear elsewhere in KB
  - Can only apply once
- **Herbrand**: if a sentence is entailed by a first order knowledge base, then there is a proof involving just a finite subset of the propositionalized knowledge base
  - finite depth of nesting: generate term of depth 1, 2, … (complete)
- **Turing & Church**:
  - The question of entailment for First Order Logic is semidecidable
  - Exists algo say yes to every entailed sentence, NE algo say no to not entailed

Generalized Modus Ponens

- Use subst. to make premises identical to sentence already in KB
  - => assert the result of implication after applying theta
  - Reason: premises do not belong to the KB, implications are meaningless
  - Lifted version of Modus Ponens. It raises Modus Ponens from PL to FOL.

- The resulting idea is known as Generalized Modus Ponens. For atomic sentences p_i, p_i' and q, where there is a substitution θ such that $\text{Subst}(\theta, p_i') = \text{Subst}(\theta, p_i)$, for all i, we can write

$$\frac{p_1', p_2', \ldots p_n', \ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\text{Subst}(\theta, q)}$$

In the example above $p_1' = \text{King(John)}$, p_2' is $\text{Greedy}(y)$, p_1 is $\text{King}(x)$ and p_2 is $\text{Greedy}(x)$. θ is $\{x/\text{John}, y/\text{John}\}$

Unification

- (merge two sentences => one unifier if there's one)
  - Find substitutions that make different logical expressions look identical
  - Unify(p,q) = θ where Subst(θ,p) = Subst(θ,q)
  - For every unifiable pair of expressions there is a single Most General Unifier (MGU) that is unique up to renaming and substitution of the variables
  - occur: avoid x=f(x) => avoid inf loop (also inc time); fetch return all unifiers

```
Function Unify-Var(var, x, θ):
    if {var/val} ∈ θ then
        return Unify(val, x, θ)
    end
    else if {x/val} ∈ θ then
        return Unify(var, val, θ)
    end
    else if Occur-check(var, x) then
        return failure
    end
    else
        return add {var/x} to θ
    end
```

```
Function Unify(x, y, θ):
    input : x variable, constant, list, or compound expression
            y, a variable, constant, list, or compound expression
            θ, the substitution built up so far (default empty)
    if θ = failure then
        return failure
    end
    else if is_Variable(x) then
        return Unify-Var(x, y, θ)
    end
    else if is_Variable(y) then
        return Unify-Var(y, x, θ)
    end
    else if is_Compound(x) and is_Compound(y) then
        return Unify(x.Args, y.Args, Unify(x.Op, y.Op, θ))
    end
    else if is_List(x) and is_List(y) then
        return Unifty(x.Rest, y.Rest, Unify(x.First, y.First, θ))
    end
    else
        return failure
    end
```

FOL CNF

- Step 1: eliminate implications
- Step 2: move ¬ inwards (e.g. ¬ ∀ x p becomes ∃ x ¬p)
- Step 3: Standardize variables (change to diff names)
- Step 4: Skolemize, Skolemization steps (removing existential quantifiers)
  - simplest case: ∃ x P(x) to P (A) [Instantiation] => May give wrong meaning
  - ∀ x [Animal(F(x)) ∧ ¬Loves(x ,F(x))] ∨ Loves(G(x),x)
    - F and G are Skolem funcs
  - Arguments of the Skolem functions are the universally quantified variables
- Step 5: Drop universal quantifiers: [Animal(F(x)) ∧ ¬Loves(x ,F(x))] ∨ Loves(G(x),x)
- Step 6: Distribute ∨ over ∧

FOL Forward Chaining

Function `FOL_Forward-Chaining(`*KB*, α`):`
- **input** : KB, the knowledge base, a set of first order definite clauses
  α, the query, an atomic sentence
- **local variables** new, the new sentences inferred on each iteration
- **while** *new is not empty* **do**
  - new $\leftarrow \{\}$
  - **for** *each rule in KB* **do**
    - $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ `Standardize-Variables(rule)`
    - **for** *each* θ *such that* $\texttt{Subst}(\theta, p_1 \wedge \ldots \wedge p_n) = \texttt{Subt}(\theta, p'_1 \wedge \ldots \wedge p'_n)$ *for some* $p'_1, \ldots p'_n$ *in KB* **do**
      - $q' \leftarrow \texttt{Subst}(\theta, q)$
      - **if** q' *does not unify with some sentence already in KB or* `new` **then**
        - add q' to `new`
        - $\phi \leftarrow \texttt{Unify}(q', \alpha)$
        - **if** ϕ *is not* $fail$ **then**
          - return ϕ
        - **end**
      - **end**
    - **end**
    - add `new` to KB
  - **end**
- **end**

FOL Resolution Rule

- We thus have

$$\frac{\ell_1 \vee \ldots \vee \ell_k, \quad m_1 \vee \ldots \vee m_n}{\text{Subst}(\theta, \ell_1 \vee \ldots \vee \ell_{i-1} \vee \ell_{i+1} \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee m_n)}$$

  where $\text{Unify}(\ell_i, \neg m_j) = \theta$

- For example, we can resolve the two clauses

$$[\text{Animals}(F(x)) \vee \text{Loves}(G(x), x)],$$
$$\text{and} \quad [\neg\text{Loves}(u, v) \vee \neg\text{Kills}(u, v)]$$

  by eliminating the complementary literals $\text{Loves}(G(x), x)$ and $\neg\text{Loves}(u, v)$, with unifier $\theta = \{u/G(x), v/x\}$, to produce the resolvent clause

$$[\text{Animal}(F(x)) \vee \neg\text{Killls}(G(x), x)]$$

Frame & Qualification

- **Frame Problem**
  - Fluent world: changes through time
  - The problem of getting a system to infer that the world remains largely the same before and after an action: it is the problem of getting the system to infer a large number of obvious non-changes
  - One solution: add all the frame axioms, explicitly asserting all the proposition that remains the same. m different actions and n fluents => O(mn)
  - Solving the representational frame problem involves defining the transition model with a set of axioms of size O (mk) rather than O (mn) (m actions being affected by the k fluents)
- **Qualification Problem** (about unusual exceptions => fail in some cases)
  - we need to confirm that all the necessary preconditions of an action holds for it to have its intended effect
  - E.g. Forward action stricken with a heart attack
  - The need to specify all those exceptions is known as the qualification problem
  - There is no complete solution to this problem within logic => how detailed to be

Reasoning Agent & Learning Agent

- **Reasoning:** generation or evaluation of claims in relation to their supporting arguments and evidence
- Reasoning skills: being able to generate and maintain viewpoints or beliefs that are coherent with, and justified by, relevant knowledge
- **Diff**: **Learning Agent can become smarter over time**; Not just from logical entailment, but from new possibly unexpected pairs of observations and feedbacks, or simply observations
- **Learning**: improves its performance on future tasks after making new observations about the world
    - Designer can't anticipate all possible situations
    - Designer can't anticipate all changes over time
    - Designer often has no idea how to program a solution
- Inductive Learning: learning a rule from specific input/output pairs
- Deductive learning: going from a known general rule to a new rule that is logically entailed but useful because it allows more efficient processing

Learning Types

- **Unsupervised learning:** the agent learns patterns in the input even though no explicit feedback is provided (e.g. clustering)
- **Reinforcement learning**: the agent learns from a series of reinforcements, rewards or punishments (e.g. Punish the agent if it doesn't give tips)
- **Supervised learning**: the agent observes some example input-output pairs and learns a function that maps from input to output (e.g. Classification)
- **Semi-supervised learning**: we are given a few labeled samples and we must make what we can out of a large collection of unlabeled examples (e.g. Tell people's ages, detect liars (compare the picture and the associated age information))

Decision Tree

- A sequence of tests(decisions), each internal node in the tree corresponds to a test of the value of one of the input attributes (e.g. wait for a table at a restaurant)
  - Greedy, divide and conquer
  - Classify correctly with a small number of tests
- Goal ⇔ (Path1 or Path2 or …) [disjunctive normal form]
- Always test the most important attribute first, the one that makes the most difference to the classification of an example (entropy)

- If all pos or neg => done
- If both pos and neg => pick best attr
- If no sample left return most common attr among parents
- If no attrs left & both pos and neg => return most common decision among the examples as the final decision for the node

- Prune to against overfit
  - Statistical significance test

```
if Examples is empty then
    return most_common(parent_examples)
end
else if all examples have the same classification then
    return the classification
end
else if attributes is empty then
    return most_common(examples)
end
else
    A ← argmax_{a∈attributes} Importance(a, examples)
    tree ← a ne decision tree with root test A
    for each value v_k of A do
        exs ← {e : e ∈ examples and e.A = v_k}
        subtree ←
         decision_tree_learning(exs, attributes - A, examples)
        add a branch to tree with label (A = v_k) and subtree subtree
    end
end
```

Algorithm 1: Learning a Decision Tree

Entropy

- a fundamental quantity in information theory
- a measure of the uncertainty of a random variable
- acquisition of information leads to a reduction of entropy

- Entropy of a Boolean random variable: B(q) = -(qlog2q + (1-q)log2(1-q))
- Entropy of goal attribute: H(Goal) = B(p / (p+n))
- For attr E_k, we have the average entropy after split
  - B(pk/pk+nk) => bits of info needed
- Information gain:
- Gain is what we will use in our importance function

$$\text{remainder}(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p + n} B\left(\frac{p_k}{p_k + n_k}\right)$$

$$\text{Gain}(A) = B\left(\frac{p}{n+p} - \text{Remainder}(A)\right)$$

- The Entropy of a variable V with values v_k each with probability $P(v_k)$ is defined as

$$\text{Entropy} : H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k)$$

- We can check that the entropy of a flipped fair coin is indeed 1 bit

$$H(\text{Fair}) = -(0.5 \log_2 0.5 + 0.5 \log_2 0.5) = 1$$

- If the coin is loaded to give 99% heads, we get

$$H(\text{Loaded}) = -(.99 \log_2 0.99 + 0.01 \log_2 0.01) \approx 0.08 \text{ bits}$$

Training/Test split; Overfitting

- A fair way to estimate the performance
- Trainset: (x1, y1), …, (xN, yN); y = f(x), and we derive a hypothesis h(x) to approx.
- We say a hypothesis generalizes well if it correctly predicts on test set
- A particular hypothesis is called consistent when it agrees with all the data.
- Multiple consistent hypotheses: Ockham's razor
- We say that a learning problem is realizable if the hypothesis space contains the true function.
    - split => holdout cross validation; Or more: K -fold cross validation

- **Overfitting**: memorizing too much (even unrelated) details
- Poly features
    - polynomials of higher degree will always interpolate the data better than polynomials of low degree.
    - Model selection: Complexity vs Goodness of fit
        - want it to learn a hypothesis that we will fit future data well
    - optimal degree in the regression problem

Regularization

- Training accuracy vs. model complexity

Ridge Regression : In ridge regression, the cost function is altered by adding a penalty equivalent to square of the magnitude of the coefficients.

$$\sum_{i=1}^{M}(y_i-\hat{y}_i)^2=\sum_{i=1}^{M}\left(y_i-\sum_{j=0}^{p}w_j\times x_{ij}\right)^2+\lambda\sum_{j=0}^{p}w_j^2$$

Cost function for ridge regression

$$\text{For some } c>0,\ \sum_{j=0}^{p}w_j^2<c$$

Supplement 1: Constrain on Ridge regression coefficients

Lasso Regression : The cost function for Lasso (least absolute shrinkage and selection operator) regression can be written as

$$\sum_{i=1}^{M}(y_i-\hat{y}_i)^2=\sum_{i=1}^{M}\left(y_i-\sum_{j=0}^{p}w_j\times x_{ij}\right)^2+\lambda\sum_{j=0}^{p}|w_j|$$

Cost function for Lasso regression

$$\text{For some } t>0,\ \sum_{j=0}^{p}|w_j|<t$$

Supplement 2: Lasso regression coefficients; subject to similar constrain as Ridge, shown before.

Linear Regression & Loss

- Training accuracy vs. model complexity
- Loss function: encodes the utility
  - Empirical loss
- Diff samples may give diff hypotheses

$\boldsymbol{\beta} \leftarrow$ any point in the weight space
while `num_iter` $<$ `max_iter` **do**
  for *each* β_i *in* $\boldsymbol{\beta}$ **do**
    $\beta_i \leftarrow \beta_i - \eta \frac{\partial}{\partial \beta_i} \text{Loss}(\boldsymbol{\beta})$
    num_iter $+= 1$
  end
end

Algorithm 2: gradient descent

Absolute value loss : $L_1(y, \hat{y}) = |y - \hat{y}|$

Squared error loss : $L_2(y, \hat{y}) = (y - \hat{y})^2$

0/1 loss : $L_{0/1}(y, \hat{y}) = 0$ if $y = \hat{y}$, else 1

$$\text{EmpLoss}_{L,E}(h) = \frac{1}{N} \sum_{(x,y) \in E} L(y, h(x))$$

$$\hat{h}^* = \underset{h \in \mathcal{H}}{\text{argmin}} \text{EmpLoss}_{L,E}(h)$$

$$\begin{aligned} \text{Loss}(h_\beta) &= \sum_{j=1}^{N} L_2(y_j, h_\beta(x_j)) \\ &= \sum_{j=1}^{N} (y_j - h_\beta(x_j))^2 \\ &= \sum_{j=1}^{N} (y_j - (\beta_1 x_j + \beta_0))^2 \end{aligned}$$

e to find the $\beta^* = \underset{\beta}{\text{argmin}} \text{Loss}(h_\beta)$

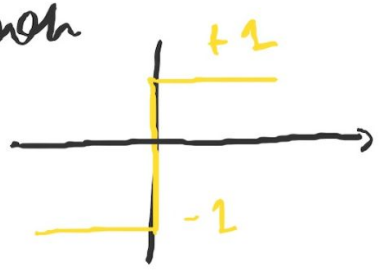
Logistic Regression & Perceptron (function)

$$h_\beta = \underset{h}{\operatorname{argmin}} \sum_{(x^{(i)}, t^{(i)}) \in E} L(h(x^{(i)}), t^{(i)}) \cdot \frac{1}{N} + \lambda \Omega$$

$$\sigma(\beta_0 + \beta_1 x_1 + \beta_2 x_2)$$

Perceptron

$\sigma(x) =$



$$\sigma(x) = \begin{cases} +1 & x \geq 0 \\ -1 & x < 0 \end{cases}$$

Output = binary decision

Logistic regression

$\sigma(x) =$

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$ as

Output interpreted proba

Logistic Regression & Perceptron (update)

→ We rely on two perceptron learning rules:

Misclassified points only

either $t^{(i)} \in \{0, 2\}$: $\beta \leftarrow \beta + \eta \left(t^{(i)} - \sigma(\beta_0 + \beta_1 x_1 + \beta_2 x_2) \right) x^{(i)}$

or $t^{(i)} \in \{-2, 2\}$: $\beta \leftarrow \beta + \eta\, t^{(i)} x^{(i)}$

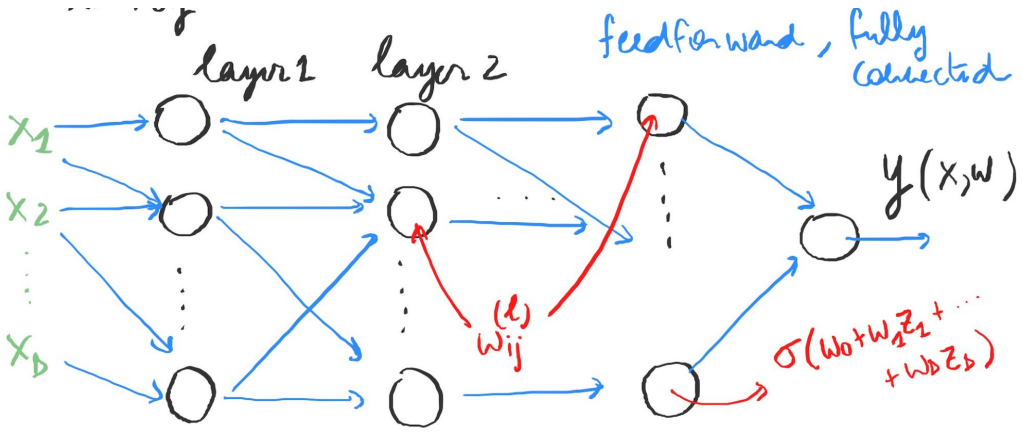
While iter < max iter

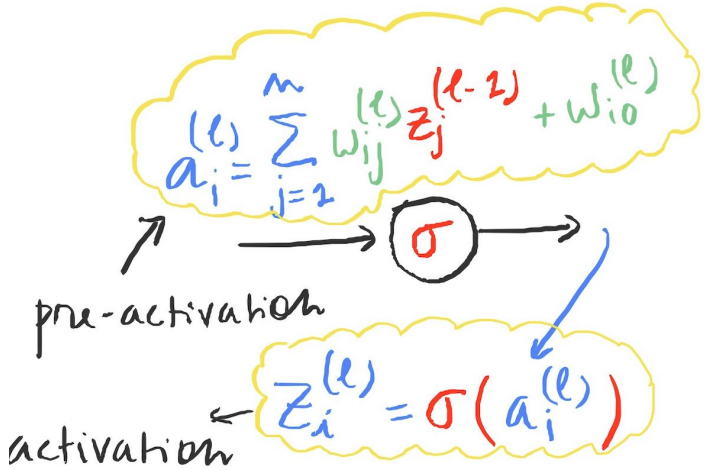
$$\beta \leftarrow \beta + \eta \frac{2}{N} \sum_{i=1}^{N} \left(t^{(i)} - \sigma(\beta_0 + \beta_1 x_1^{(i)} + \beta_2 x^{(i)}) \right) \times \sigma(\beta_0 + \beta_1 x_1^{(i)} + \beta_2 x_2^{(i)}) \left(1 - \sigma(\beta_0 + \beta_1 x_1^{(i)} + \beta_2 x_2^{(i)})\right) \times \tilde{x}^{(i)}$$

Neural Network (NN)

$$w^* = \underset{w}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \left(t^{(i)} - y(x^{(i)}; w)\right)^2$$

- Input Vector: [x_1, …, x_D], sigmoid: 1/(1+e^{-x})





$$l(w) = \frac{1}{N} \sum_{i=1}^{N} \left(t^{(i)} - y(x_i; w)\right)^2$$

Back Propagation

- Step 1: forward prop, compute pre-act a_i and post-act z_i
- Step 2: delta_out = dL/da_{out}
- Step 3: back prop & derive all delta_i at each layer l
- Step 4: compute gradients
- Step 5: gradient update

$$\delta_i^{(l)} = \sum_{j=1}^{m} \delta_j^{(l+1)} \cdot w_{ij}^{(l+2)} \cdot \sigma'(a_j^{(l)})$$

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} \, z_j^{(l-1)}$$

$$\frac{\partial L}{\partial a_{out}} = -\frac{2}{N}\left(t^{(i)} - \sigma(a_{out})\right) \times \sigma(a_{out})\left(1 - \sigma(a_{out})\right)$$

$\sigma(a_{out}) = y(x^{(i)}; w)$

let $\delta^{out} = \frac{\partial L}{\partial a_{out}}$

$$a_i^{(l)} = \sum_{j=2}^{m} z_j^{(l-1)} w_{ij}^{(l)} + w_{i0}^{(l)}$$

What we want are $\frac{\partial L}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} \cdot z_j^{(l-1)}$

Main elements of RL

- RL is learning what to do so as to maximize a numerical reward signal

- **Policy**: learning agent's way of behaving at any given time
- **Reward**: good and bad events signal send to the agent
- **Value function:** total amount of reward the agent can expect to accumulate over the future, starting from that state
- **Model for the environment:** enables inferences to be made on how the environment will react w.r.t a particular action
    - predict the next state and next reward given the current state and action

- Nonassociative: no need to associate different actions with different situations
    - find a single best action (when the task is stationary)
    - track the best action as it changes over time (nonstationary)

Markov Decision Process (MDP)

- A **sequential decision problem** for a **fully observable stochastic environment** with a **Markovian transition model** and **additive rewards**.

- Define a transition model P (s ′|s ,a)
  - prob of reaching state s ′ if action a is performed in state s
  - Can be defined as a 3D table with p; More refined: dynamic Bayesian network

- Assume model is Markovian
  - Prob depends only on prev state

- Utility Function (also: environment history)
  - Reward R(s) received at each step
  - For now, utility is just simply sum

- Solution: **policy**, traditionally denoted as **π(s)**
  - The action recommended by the policy pi for state s
  - Complete policy: no matter action outcome, agent always know what to do next

Expected utility, optimal policy; **Stationarity**

- Quality of a policy: **expected utility** of that policy
- Optimal policy: yields highest expected utility (π^*)
  - Optimal action: $\pi^*(s)$
- **Non-stationary**: optimal action may change over time
- **Stationary**: optimal action only depends on current state
- Compute Utility => Simplifying Assumption
  - Agent's preferences between state sequences are stationary
  - $[s_0, s_1, s_2, \ldots]$ same as $[s'_0, s'_1, s'_2, \ldots]$, if $s_0 = s'_0$

  - Additive rewards according to which the utility of a state sequence is defined as

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

  - Discounted rewards according to which the utility of a sequence is defined as

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

Maximum Expected Utility (MEU) Policy

- Compare policies by comparing their **expected utilities**
- Define S_t (a random variable), at time t, exec policy π
  - Prob distribution of [S1, S2, ...] depends on policy and initial state

- The expected utility obtained by executing π starting at s is then given by

$$U^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

Here the expectation is taken over state sequences determined by s and π.

- Now out of all the policies that the agent could choose to execute starting in s, one (or more) will have higher expected utlity than the others. We will use π^* to denote one of those policies

$$\pi_s^* = \underset{\pi}{\text{argmax}}\ U^{\pi}(s)$$

$$\pi^*(s) = \underset{\pi}{\text{argmax}} U^{\pi}(s)$$
$$= \underset{a \in A(s)}{\text{argmax}} \sum_{s'} P(s'|s,a)U(s')$$

Action Choice & Bellman equation

- Note that the expected utility of an action given some evidence e (e.g. the fact that we are in a state s), is simply the average value of the possible outcomes, weighted by the probability that the particular outcome will occur:

Best action: $$\pi^*(s) = \operatorname*{argmax}_a \sum_{s'} P(s'|s,a)U(s')$$

- From this it follows that one can decompose the utility of a state as the immediate reward for that state (that the agent fot when it reached that state) plus the expected discounted utility of the next states (given that the agent always choose its action so as to maximize this utility):

BE: $$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s')$$

The equation above is known as the Bellman equation named after Richard Bellman (1957)

- BE is basis of value iter algo
- N states => N BEs
- Each eq has N unknowns
  - Util of each state
- Not linear: max operator
- Can't be solved, but can use iterative approach

Value Iteration Algorithm

- In Value iteration, we start with **arbitrary initial values for the utilities** of each state, **calculate the right handside** of the Bellman equation and then **update the left handside** with this right handside

- Unlike policy iteration, Value iteration **does not require any policy evaluation**

Let $U_i(s)$ denote the utility value for the state s at the i^{th} iteration. The Value Iteration step, also known as Bellman Update then reads as

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$$

Function `Value-Iteration(`*mdp*, *varepsilon*`)`**:**

input : *mdp*: an mdp with states S, actions $A(s)$, transition model $P(s'|s,a)$, reward $R(s)$, discount γ
ε, the maximum error allowed in the utility of any state

local variables: U, U': vectors of utilities for states in S, initially zero
δ: the maximum change in the utility of any state at any iteration

while $\delta < \varepsilon(1-\gamma)/\gamma$ **do**
  $U \leftarrow U'; \delta \leftarrow 0$ **for** *each state s in S* **do**
    $U'[s] \leftarrow R[s] + \gamma \max_{a \in A[s]} \sum_{s'} P(s'|s,a) U[s']$
    if $|U'[s] - U[s]| > \delta$ **then**
      $\delta \leftarrow |U'[s] - U[s]|$
    end
  end
end
return U

Value Iteration Convergence: contraction

- Contraction f(x) => when applied to two diff inputs produces outputs that are closer together than the original inputs => || f(x) - f(y) || <= gamma * || x - y ||
  - fixed point: unchanged points by contraction

$$
\begin{aligned}
&\left\|\mathcal{B}U_i - \mathcal{B}U_i'\right\| \\
&= \max_s \left| \gamma \max_{a\in A(s)} \sum_{s'} P(s'|s,a)U_i(s') - \gamma \max_{a\in A(s)} \sum_{s'} P(s'|s,a)U_i'(s') \right| \\
&\le \max_s \max_{a\in A(s)} \left| \gamma \sum_{s'} P(s'|s,a)U_i(s') - \gamma \sum_{s'} P(s'|s,a)U_i'(s') \right| \\
&\le \gamma \max_s \max_{a\in A(s)} \sum_{s'} P(s'|s,a) \left| U_i(s') - U_i'(s') \right| \\
&\le \gamma \max_s \max_{a\in A(s)} \max_{s'} \underbrace{\left| U_i(s') - U_i'(s') \right|}_{\|U_i - U_i'\|} \sum_{s'} P(s'|s,a) \\
&\le \gamma \|U_i - U_i'\|
\end{aligned}
$$

To prove the existence of the fixed point

$$\|\mathcal{B}U_{n-1} - U_{n-1}\| \le \gamma^{n-1}\|U_1 - U_0\|$$

which for a sufficiently small γ and a sufficient number of Bellman updates gives $\lim_{n\to\infty} \mathcal{B}U_{n-1} = \lim_{n\to\infty} U_{n-1}$

$$\|U_i - U\| < \varepsilon \quad \Rightarrow \quad \|U^{\pi_i} - U\| < 2\varepsilon\gamma/(1-\gamma)$$

To prove the convergence to a fixed point, we first show that for any two functions f and g, we have

$$\left|\max_a f(a) - \max_a g(a)\right| \le \max_a |f(a) - g(a)|$$

Policy Iteration Algorithm

- Value iteration: possible to get an optimal policy through iteration
- **Better** action (utility) => **focus on** this action (even if not optimal yet)
- **Terminate** when no improvement in utility => reached a fixed point of the Bellman
  - U_i is solution to BE; π_i is optimal policy

Policy evaluation: given a policy π_i, calculate $U_i = U^{\pi_i}$, the utility of each state if π_i were to be executed

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) U_i(s')$$

- Max removed, now linear
- n linear eq with n unknowns
- LA solver $O(n^3)$

Policy Improvement: Calculate a new Maximum Expected Utility policy using

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s'|s, a) U(s')$$

- Small state spaces => policy evaluation using exact solution methods is favored
- Large => a number of simplified value iteration (k times) => estimate next utility update
  - modified policy iteration

Function `Policy-Iteration(mdp)`:

input : *mdp*: an mdp with states S, actions $A(s)$, transition model $P(s'|s,a)$

local variables: U a vector ot utilities for states in S, initially zero

π, a policy vector indexed by state, intially random

while *unchanged* **do**

$U \leftarrow \text{POLICY-EVALUATION}(\pi, U, mdp)$

unchanged? $\leftarrow$ true

Iterate until no change in utility

for *each state s in S* **do**

if $\max_{a \in A(s)} \sum_{s'} P(s'|s,a)U[s'] > \sum_{s'} P(s'|s,\pi[s])U[s']$ **then**

$\pi[s] \leftarrow \text{argmax}_{a \in A(s)} \sum_{s'} P(s'|s,a)U[s']$

unchanged? $\leftarrow$ false

end

end

end

return π

Reinforcement Learning

- MDPs: assume agent has complete model for env (i.e. know rewards in advance)

- General RL: no prior knowledge
  - Complex domains: RL is only way to train a program to perform at high levels
  - RL agents: assume that the environment is fully observable
    - current state is supplied by each percept

- Example: playing game
  - program can be told when it has won or lost
  - learn an evaluation function => prob of winning at any state

Passive & Active Learning

$$U^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

- Passive learning: agent's policy is fixed, the task is to learn the utilities of the states
  - In state s, agent always do π(s)
  - Only action of the agent is to learn how good the policy is (i.e. $U^{\pi}(s)$)
  - Similar to policy evaluation part of policy iteration
  - Only difference: don't know: transition model P (s ′|s ,a) & reward function

- Active learning: also learn what to do
  - Explore as much as possible => learn how to behave
  - Diff: learn a complete model with outcome probabilities for all actions, rather than just the model for the fixed policy
  - the agent has a choice of actions
  - Need to learn the utilities defined by the optimal policy
    - Obey BE, solved through value/policy iter
  - Greedy agent: always follow optimal policy
    - Often suboptimal for true env
  - Improve => do more than collecting rewards => exploration & exploitation

Direct Utility Estimation

- Each completed simulation provides a sample for the utility
- Idea: [expected utility of a state] = [expected total reward from that state onward]
  - Each trial thus provides a sample of this quantity for each state visited
- At then end of each sequence => calculate the expected reward to go for each state
  - update the utility for that state accordingly
  - infinitely many trials => converge to true expectation
- Reduce RL problem to inductive learning problem
  - Miss one important aspect: utilities of states are not independent
  - [Utility of each state] = [own reward] + [expected utility of its successor states]
  - Miss the opportunities for learning
    - BE: connectivity of states encodes the importance
    - DUE: need another experiment / sample
  - => usually converges slow in practice

Adaptive dynamic programming (ADP)

- Takes advantage of the **constraints among the utilities** of states
  - learn the transition model by keeping track of how each action outcome
  - learning the transition model that connects them
  - Solve the resulting MDP using **dynamic programming**

- For a passive agent (with known policy $\pi(s)$)
  - Plug transition model [$P(s' | s, \pi(s))$] and regard [$R(s)$] into BE => util of states
  - Linear equations, use LA solver; or use modified policy iteration

- Learn a transition model is easy => env fully observable
  - Essentially supervised learning task: map [state-action pair] to [resulting state]
  - Simplest case: represent transition model as a table of probabilities
    - How often [action outcome] occurs => estimate probability

- ADP agent uses maximum likelihood to estimate the transition model
- Solely based on the estimated model may not be a good idea (e.g. taxi, red light)
  - Better: choose not necessarily optimal policy, but work well for a range of model that have a reasonable chance of being a true model

Function `Passive-ADP-Agent(`*percepts*`)`**:**

input : percept, a percept sequence indicating
the current state s' and the reward signal r'

persistent : π, a fixed policy, $mdp(P, R, \gamma)$
U, a table of utilities (initially empty)
N_{sa}, a table of frequencies for sate-action pairs (init. zero)
$N_{s'|s,a}$, a table of outcome frequencies for state action pairs
s, a, the previous state and action

if s' *is new* **then**
  $U[s'] \leftarrow r'$, $R[s'] \leftarrow r'$
end
if s *is not null* **then**
  increment $N_{sa}[s, a]$ and $N_{s|s,a}$
  for *each* t *such that* $N_{s'|s,a}[t, s, a]$ *is nonzero* **do**
    $P(t|s, a) \leftarrow N_{s'|s,a}[t, s, a]/N_{s,a}[s, a]$
  end
end
$U \leftarrow Policy - Evaluation(\pi, U, mdp)$
if $s'.\text{TERMINAL?}$ **then**
  $s, a \leftarrow$ null
end
else
  $s, a \leftarrow s', \pi[s']$
end
return a

Temporal Difference Learning (TD)

- Another way for updating the transition probabilities and solving the MDP
- Use [observed transitions] to adjust [utilities of the observed states]
  - => agree with the constraint equations
- Subtlety
  - update only involves observed successors of state s; ave of $U^{\pi}(s)$ still converge
    - actual equilibrium equation => involves all the successors
  - Decreasing lr (as iter goes) => help converge
    - $\alpha(n) = C / (C + n)$, C is large constant like 60

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \eta \left(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s)\right)$$

which reduces the difference between the LHS and the RHS in the Bellman equation. η is the learning rate.

Function `Passive-TD-Agent(`*percepts*`)`**:**

input : percept, a percept sequence indicating the current state s' and the reward signal r'

persistent : π, a fixed policy, $mdp(P, R, \gamma)$
U, a table of utilities (initially empty)
N_s, a table of frequencies for states (init. zero)
s, a, r, the previous state and action and reward

if s' *is new* **then**
  $U[s'] \leftarrow r'$
end
if s *is not null* **then**
  increment $N_s[s]$
  $U[s] \leftarrow U[s] + \eta(N_s[s])(r + \gamma U[s'] - U[s])$
end
if $s'.\text{TERMINAL?}$ **then**
  $s, a, r \leftarrow$ null
end
else
  $s, a, r \leftarrow s', \pi[s'], r'$
end
return a

ADP & TD

- Closely related
  - make local adjustments to utility estimates => let each s agree with successors

- Diff 1
  - TD: adjusts a state to agree with its observed successors
  - ADP: adjusts a state to agree with all of the successors
  - Disappear when TD adjustments is averaged over a large iter_num

- Diff 2
  - TD: a single adjustment per observed transition
  - ADP: many as it needs to restore consistency
    - Between: util estimate [U] & env model [P]

Multi-armed (stationary) Bandit Problem

Expected Reward: $v(a) = \mathbb{E}\{R_t | A_t = a\}$

- Non-associative feedback problem
- faced repeatedly with a choice among k different possible options or actions
  - K-armed case: k levers of a slot machine
  - Reward with stationary probability (depends on action)
- Explore or exploit: average over past rewards
- Non-stationary: add weights, more to recent actions
- Policy gradient / Gradient Bandit Algorithm

In Gradient bandit algorithms, we define policies (and the corresponding preferences H_t) by means of a softmax distribution (equiv. Gibbs or Boltzmann distribution),

$$\pi_t(a) = \Pr\{A_t = a\} = \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}}$$

here $\Pr\{A_t = a\}$ really means "the probability that the optimal action at time t is a".

$$Q_n = \frac{R_1 + R_2 + \dots R_{n-1}}{n-1}$$

we compute Q_{n+1} as

$$Q_{n+1} = \frac{1}{n}\sum_{i=1}^{n} R_i = \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right)$$
$$= Q_n + \frac{1}{n}[R_n - Q_n]$$

1. Initialize, for every action $a = 1$, to k
   - 1.1 $v(a) \leftarrow 0$
   - 1.2 $n(A) \leftarrow 0$ (number of times A has been chosen)
2. Repeat
   - 2.1 $A \leftarrow \begin{cases} \underset{a}{\text{argmax}}\, v(a) & \text{with probability } 1-\varepsilon \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$
   - 2.2 $R \leftarrow \text{bandit}(a)$
   - 2.3 $n(A) \leftarrow n(A) + 1$
   - 2.4 $q(A) \leftarrow v(A) + \frac{1}{N(A)}[R - v(A)]$

Exploration Function

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

- Determines how greed (i.e. preference for higher values of U) is traded off against curiosity (preference for actions that have not been tried often and have low n)

- f (u,n) should be increasing in u and decreasing in n
- Making the agent try each action-state pair at least N_e times

- Modified Bellman update: U+ on RHS, important
  - Still use U: more pessimistic utility estimate, decline to explore further
  - Unexplored weights higher

$$U^+(s) \leftarrow R(s) + \gamma \max_a f\left(\sum_{s'} P(s'|s,a)U^+(s'), N(s,a)\right)$$

Q-Learning (TD)

$$Q[s,a] = R[s] + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q[s',a']$$

- Learns an action-utility representation instead of learning utilities
- Q [s ,a]: value of doing action a in state s, U (s) = $\max_a$ Q [s ,a]
  - Property: a TD agent that learn Q-func don't need model of form P (s ′|s ,a)
  - model-free method
  - constraint equation that must hold at equilibrium when with correct Q values
- Similar to ADP, use update eq on Q [s ′,a′] directly to compute Q-values
  - ADP: require that a model to be learned (since P (s ′|s ,a) in eq)
  - TD: require no model for state transitions => Q-values is all we need

The update equation for TD Q-learning is

$$Q[s,a] \leftarrow Q[s,a] + \alpha \left(R[s] + \gamma \max_{a'} Q[s',a'] - Q[s,a] \right)$$

Function `Q-learning-Agent(`*percept*`)`**:**

input : *percept*, a percept indicating the current state s' and reward signal r'

persistent : Q, a table of action values indexed by state and action, initially zero
N_{sa}, a table of frequencies for state action pairs, initially zero
s, a, r, the previous state, action, and reward, initially null

if *s is not null* **then**

increment $N_{sa}[s, a]$

$Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])\,(R + \gamma \max_{a'} Q[s', a'] - Q[s, a])$

end

$s, a, r \leftarrow s', \underset{a'}{\text{argmax}}\, f\left(Q[S', A'], N_{sa}[s', a']\right), r'$

return a

State-Action-Reward-State-Action (SARSA)

- The rule is applied at the end of each s, a, r, s'
- Diff
  - Q-learning: backs up the best Q-value from the state reached in the observed transition
    - Pay no attention to actual policies => off policy learning (diff act & eval strategy)
    - Assume take best regards => update table => take action by epsilon-greedy
    - More flexible => can learn to behave well even with a random explore policy
  - SARSA: waits until an action is actually taken and backs up the Q-value for that action
    - On policy learning
    - Take action by epsilon-greedy => update table
    - More realistic => better to learning what is happening in some cases
  - For a greedy agent: two are identical
  - When exploration is happening: differ significantly

Generalization in RL

$$\theta_0 \leftarrow \theta_0 + \alpha(u_j(s) - \hat{U}_\theta(s))$$
$$\theta_1 \leftarrow \theta_1 + \alpha(u_j(s) - \hat{U}_\theta(s))x$$
$$\theta_2 \leftarrow \theta_2 + \alpha(u_j(s) - \hat{U}_\theta(s))y$$

- Small state spaces => works reasonably well
- Larger state space => convergence takes much longer
  - Solution: function approximation

- The representation is viewed as an approximation of the true utility function
  - May not be true utility function & Q function
  - E.g. a linear function of a set of features
  - Benefit 1: fit in large state spaces
  - Benefit 2: generalize to unvisited states

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s)$$

$$\hat{U}(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

- Hypothesis space may fail to approximate true utility function
- [In all inductive learning]: tradeoff => size of the hypothesis space & time to learn a func

- Function approximation to represent the utility => supervised learning task
  - use an online learning algorithm to update the parameters after each trial
  - => gradient optimization based on error function
  - $u_j(s)$ -> observed total reward from state s onward
    - in the j-th trial

$$E_j(s) = \frac{1}{2}\left(\hat{U}_\theta(s) - u_j(s)\right)^2$$

Extend TF & Q to parametric representations

- Apply the ideas behind function estimation
- All we need to do is adjust the parameters to try to reduce the temporal difference between successive state

- Passive TD learning: update rule converge (when approx func is linear in the parameters)
- With active learning and nonlinear functions (e.g. NN)
  - Params may go off to inf (even true func in space)

- learning a model for an observable environment is a supervised learning problem
  - Since the next percept gives the outcome state

- Policy search: twiddle the policy as long as its performance improve, then stop

The new versions of the TD and Q-learning equations (for utilities and Q-values) are then given by

$$\theta_i \leftarrow \theta_i + \alpha \left[R(s) + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s) \right] \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

$$\theta_i \leftarrow \theta_i + \alpha \left[R(s) + \gamma \max_{a'} \hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a) \right] \frac{\partial \hat{Q}_\theta(s, a)}{\partial \theta_i}$$

Simple Genetic Algorithm

$$x = \sum_{i=0}^{l-1} a_i 2^{i-l} (\bar{x} - \underline{x}) + \underline{x}$$

- Evolutionary Algorithms
  - Population based, fitness oriented, variation driven
- Step 1: pick a population size (each individual has two property: location, value)
- Maintain a high quality mating pool: parents => offspring
- Binary representations (mimic genetic encoding)
- a = [a0, a1, ...] => genotype or chromosome
- Init => usually random
- Roulette wheel select- in proportion to fitness
  - normalize
- Ranking select: rank number → prob
- Cross-over & mutation

$$p_i = \frac{f_i}{\sum_{j=1}^{M} f_j}$$

p_i is then viewed as the probability of selecting individual i for the mating pool.

Repeat until iter < maxIter

Select sub population from previous population and generate mating pool

Repeat:
- Select 2 individuals from the mating pool randomly without replacement
- perform cross over with probability p_c
- perform mutation with probability p_m
- evaluate fitness

$$P(i) = \frac{2-S}{N} + \frac{2i(S-1)}{N(N-1)}$$

$1 < S \leq 2$ is a fixed param.

Evolution Strategy

- Step 1: parent population init
- Mul / Rho => num of parents involved in producing offspring
- Parent family of size rho, select parents of size mul
- Mutation
- Discrete recomb (uniform), intermediate recomb (ave), weighted multi-recomb

- In (μ, λ) selection, only the λ newly generated offspring individuals define the selection pool
- In $(\mu + \lambda)$ selection, μ parents are combined with λ offsprings to create the mating pool (the case $\mu+1$ is known as steady state ES)