Session 14

I. Announcements [5 minutes]

- **Homework 6A is due 12/5… next Monday.**
- **Assignment 6 has the dual due dates. Get it in by the 1st to leave plenty of time for cramming ☺.**
- Schedule:

11/28	**Assignment 5B is due**
11/29	NLP Lecture
12/1	NLP Lecture
12/6	Robotics
12/8	Philosophy
12/8 (?)	**Early turn-in for Assignment 6**
12/12 (?)	**Final turn-in for Assignment 6**
12/19	**FINAL EXAM**

II. Review of Reinforcement Learning

Reinforcement Learning (RL) – the task of using observed *rewards* to learn a (approximately) optimal policy for an environment by choosing an action that will maximize the *expected reward* given the current observed state of the agent.

- **Reward (Reinforcement) –** feedback that differentiates between *good* and *bad* outcomes; thus allowing the agent to make choices.
- Unlike MDPs, RL agents assume no prior knowledge of either the environment or the reward function.
- Three types of agent designs:
    - ***utility-based agent –*** learn a utility function for states, which the agent will use to select actions in order to maximize expected utility.
        - requires an environment model to map actions to successor states.
    - ***Q-learning agent –*** learns a utility function on the state-action pairs; a so-called Q-function.
        - able to compare actions without knowing their outcomes.
        - without knowing action outcome, look ahead is not possible.
    - ***reflex agent –*** learns a policy that maps states to actions.

Passive Reinforcement Learning – the agent's has a fixed policy π: perform action $\pi(s)$ in state *s*. This is similar to *policy iteration*, but we lack the *transition model* $T(s,a,s')$ and the *reward function* $R(s)$. Thus, the agent performs a set of **trials** and uses the observed rewards to estimate the expected utility of each state $U^{\pi}(s)$. Starting in state *s* we want to estimate the (discounted) expected reward from future states:

$$U^{\pi}(s) = \mathrm{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid \pi, s_0 = s\right]$$

- **Direct Utility Estimation –** the utility of a state is the expected reward starting from that state, so each **trial** is a sample for each state visited.
    - In this setting, the problem becomes a supervised learning problem of mapping state to value ➔ an inductive learning problem.
    - This *Monte-Carlo* approach assumes independence of the utility function between states. This ignores the fact that utilities are coupled in the Bellman equations! Thus, this approach does not **bootstrap**!
        - Without bootstrapping, invaluable information for learning is lost and thus the technique converges very slowly.
- **Adaptive Dynamic Programming (ADP) –** as the agent moves through the environment, the transition model is estimated and the MDP for the corresponding model estimate is solved incrementally using dynamic programming.
    - Learning the environment:
        - The transition model $T(s,a,s')$ is estimated from the frequency from state *s* to state *s'* via action *a*.
    - The MDP is solved using <u>policy iteration</u> or <u>modified policy iteration</u>.
    - ADP is intractable for large state spaces.
    - approximate ADP – bounds the number of adjustments per transition.
        - *prioritizing sweep heuristic* – prefers to adjust states whose successors have recently had a large utility adjustment.
- **Temporal Difference (TD) Learning –** a mixture of sampling and constraint bootstrapping in which the values of the observed states are modified to reflect the constraints between states given by the MDP.
    - TD equation: given a learning rate α we update the expected utilities:

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

      The TD equations converges to the MDP equilibrium even though only visited states are considered – the frequency of visits to a state are a substitute for the explicit transition model.
    - TD is an efficient approximation of ADP:
        - the utility function is updated by local adjustments.
        - TD only adjusts w.r.t. the observed transition and only makes a single update per transition.

Active Reinforcement Learning – policy is no longer fixed; active agents must decide on actions to take.

- **Exploration**
  - ***greedy agent*** – follows the current "optimal policy" according to the current estimates of the utility of each state.
    - unlikely to converge to the "optimal policy" since neglected states have poor estimates of their utility functions.
  - Trade-off between exploration and exploitation
    - ***exploitation*** – utilizing current knowledge to perform actions that maximize rewards.
    - ***exploration*** – trying suboptimal actions with the hope of improving our current estimates for the utility function.
  - **Greedy in the limit of infinite exploration (GLIE**) – exploration schemes that are eventually optimal.
    - simple GLIE scheme – try a random action with probability $1/t$; otherwise, perform the optimal action.
    - optimistic utility estimates that favor unexplored states:

      $$U^{+}(s) = R(s) + \gamma \max_{a} f\left(\sum_{s'} T(s,a,s')U^{+}(s'), N(a,s)\right)$$

      - U^{+} is the optimistic utility function
      - $N(a,s)$ is the # of times action *a* is done in state *s*.
      - *exploration function* $f(u,n)$ - trade-off between greed and curiosity that must increase in *u* and decrease in *n*. e.g.

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

      - policy converges quickly while utility estimates converge slowly, but all we need is correct policy!
- **Action-Value Function**
  - *TD-learning* can be adapted to the active setting simply by choosing an action based on the current *U* estimate via 1-step look-ahead. However, we still have to learn the environment model to select actions.
  - **Q-learning** – learns an action-value representation instead of utilities.
    - Q-values: $U(s) = \max_{a} Q(a,s)$
    - ***model-free*** – does not require an environment model for learning or action selection.
    - *Bellman equations for Q-values*:

      $$Q(a,s) = R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(a',s')$$

    - *TD Q-learning*: (model-free)

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

      - TD doesn't enforce consistency between values by using the model so it learns slower!

Generalization in Reinforcement Learning – we now consider methods for scaling RL to worlds with enormous state spaces. Standard tabular RL is impractical since the table has one entry per state and since most states would be visited rarely.

- **function approximation** - representing the value function in (approximate) non-tabular forms, e.g., a linear combination of ***features*** of the state:

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

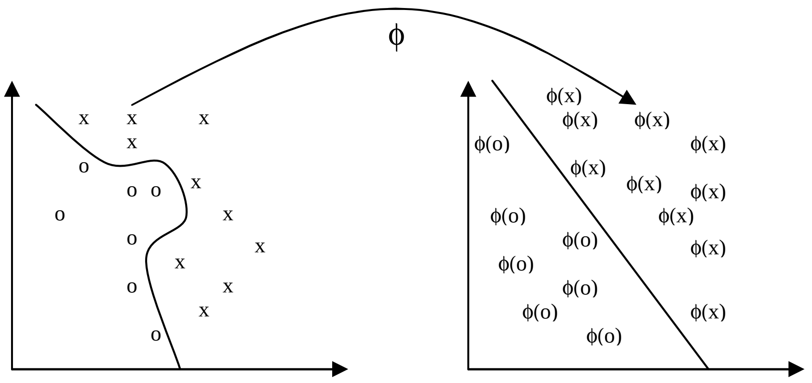
  - Thus we want to learn the parameters $\theta_1, \ldots, \theta_n$ to best approximate the value function. *Note*: features can be non-linear in the state variables.
  - *Function approximation allows the agent to broadly generalize between many states via states' common attributes.*
  - Unfortunately, the best utility function may be a poor estimate!
  - Online learning updates (**Widrow-Hoff** or **Delta Rule**): uses derivatives of squared error to update parameters.

$$\theta_i \leftarrow \theta_i + \alpha\left(u_j(s) - \hat{U}_\theta(s)\right)\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

    - TD update: $$\theta_i \leftarrow \theta_i + \alpha\left(R(s) + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s)\right)\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$
    - Q: $$\theta_i \leftarrow \theta_i + \alpha\left(R(s) + \gamma \max_{a'} \hat{Q}_\theta(a', s') - \hat{Q}_\theta(a, s)\right)\frac{\partial \hat{Q}_\theta(a, s)}{\partial \theta_i}$$
  - These updates converge to the optimal estimate for linear functions, but can wildly diverge for non-linear ones.
- Function approximation can also be used to estimate the environment model:
  - in *observable models*, this is a supervised task.
- in *partially-observable* models, DBNs with latent variables can be used

III. Supervised Learning[1]

Linear pattern – A pattern drawn from a set of patterns based on a linear function class.



Support Vector Machines

The Maximal Margin Classifier – based on the intuition that members of both class should be as far from the classification boundary as possible in order to decrease the probability that unseen examples will be misclassified. Thus, for a desired margin γ we want a margin $m(S,g) = \min_{1 \le i \le \ell} y_i g(\mathbf{x}_i) \ge \gamma$.

(linearly) separable – a characteristic of a set of points indicating that the classes of data in the set can be separated by a hyperplane of margin $\gamma > 0$. Thus, there exists a **w** and *b* such that, for the function $g(\mathbf{x}) = \langle \mathbf{w}, \phi(\mathbf{x}) \rangle + b$, the error $\xi_i = (\gamma - y_i g(\mathbf{x}_i))_+ = 0$ for all *i*.

consistent – a classifier that classifies all of its training set correctly.

Hard Margin SVM – this formulation of the SVM assumes that $\|\mathbf{w}\|^2 = 1$. Thus, $y_i g(\mathbf{x}_i)$ determines how far the image of point $\mathbf{x}_i$ is from the hyperplane. In this form, we want to maximize this margin with the constraint that all points must have a margin at least γ. However, by forcing this constraint, our solution is sensitive to a single outlier.

Primal:

$$\begin{aligned} &\min_{\mathbf{w},b,\gamma} && -\gamma \\ &s.t. && \forall i \in \{1,\ldots,l\} \quad y_i\left(\langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle + b\right) \ge \gamma \\ &&& \|\mathbf{w}\|^2 = 1 \end{aligned}$$

[1] Notes based from books outside this class – "Kernel Methods for Pattern Analysis" by John Shawe-Taylor and Nello Christianini and "An Introduction to Probabilistic Graphical Models" by Michael Jordan.

Dual:

$$\max_{\boldsymbol{\alpha}} \qquad W(\boldsymbol{\alpha}) = -\sum_{i,j=1}^{l} \alpha_i \alpha_j y_i y_j \kappa(\mathbf{x}_i, \mathbf{x}_j)$$

$$s.t. \quad \sum_{i=1}^{l} \alpha_i = 1 \quad \sum_{i=1}^{l} y_i \alpha_i = 0 \quad and \quad \forall i \in \{1, \ldots, l\} \quad \alpha_i \geq 0$$

Optimal Parameters:

$$\gamma^* = \sqrt{-W(\boldsymbol{\alpha}^*)}$$

$$\mathbf{w}^* = \sum_{i=1}^{l} y_i \alpha_i^* \phi(\mathbf{x}_i)$$

$$b^* = y_i (\gamma^*)^2 - \sum_{j=1}^{l} y_j \alpha_j^* \kappa(\mathbf{x}_i, \mathbf{x}_j) \qquad \text{where } i \in SV$$

KKT conditions: $\forall i \in \{1, \ldots, l\} \quad \alpha_i^* \left[y_i \left(\langle \mathbf{w}^*, \phi(\mathbf{x}_i) \rangle + b^* \right) - \gamma^* \right] = 0$

- This implies that either $\alpha_i^* = 0$ or $y_i \left(\langle \mathbf{w}^*, \phi(\mathbf{x}_i) \rangle + b^* \right) = \gamma^*$.
- Hence only vectors with a geometric margin of γ^* (lie closest to the hyperplane) have $\alpha_i^* > 0$ → *support vectors*.

Decision Function: $f(\cdot) = \operatorname{sgn}\left[\sum_{j=1}^{l} \alpha_j^* y_j \kappa(\mathbf{x}_j, \cdot) + b^* \right]$

Convexity

A common theme in kernel methods revolves around the fact that our kernel is positive semidefinite ➔ creating PSD kernel matrices for all training sets.

- PSD matrices lead to convexity in a number of (quadratic) optimizations of this chapter – these problems have desirable properties.
    - Solving a quadratic programming problem with PSD objective matrix leads to ***global optima*** – a unique solution.
    - Solutions to the quadratic programming can be found efficiently ($O(\ell^3)$ with the naïve methods but near linear with specialized approaches).

Duality Gap – when the primal is a minimization, the primal objective is always greater than or equal to the dual objective.

- Our problem satisfies ***Strong Duality,*** meaning that at the solution, there is no gap between the primal and dual objective.
- Thus, we can use the duality gap as a measure of convergence:

$$gap = \hat{\gamma} - \sqrt{-W(\boldsymbol{\alpha})} \qquad \hat{\gamma} = \frac{\min_{y_i=1} \langle \hat{\mathbf{w}}, \phi(\mathbf{x}_i) \rangle - \max_{y_i=-1} \langle \hat{\mathbf{w}}, \phi(\mathbf{x}_i) \rangle}{2}$$

Linear Regression and the LMS algorithm

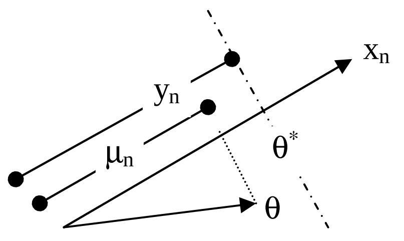
- This chapter focuses on the linearity assumption. In particular it is assumed that each data point imposes a linear constraint on the parameters.
- Convergence of the constraint satisfaction algorithm appeals to geometric notions.
- How data points are presented to the learner:
  - **batch setting** – data are available as a block.
    - ideal for situations where we are only interested in the estimate.
    - batch analysis often is easier than on-line.
  - **on-line setting** – data arrive sequentially.
    - ideal for situations where learner must respond in real-time.
    - Sweeps of iterative batch algorithms can be analyzed as on-line algorithms.

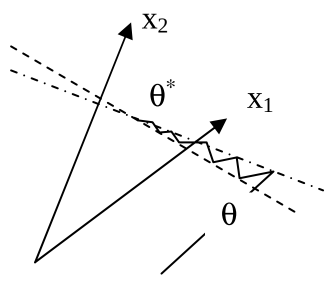
Least Mean Squares (LMS) algorithm – assumes linear dependence of one variable on another; namely $y_n = \langle \theta, x_n \rangle + \varepsilon_n$ where ε_n is an error term.

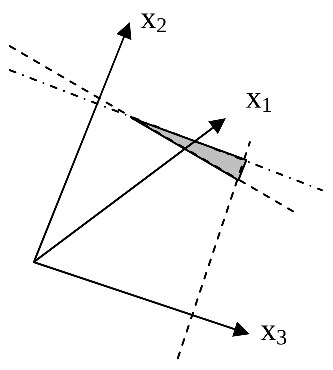
- The projection of θ onto the data point is written as:

$$\mu_n = \frac{\langle \theta, x_n \rangle}{\|x_n\|} x_n$$

- The line of possible parameters θ_n^* such that their projection onto x_n is the desired value y_n is orthogonal to x_n:







- LMS update:

$$\theta^{(t+1)} = \theta^{(t)} + \rho\left(y_n - \left\langle \theta^{(t)}, x_n \right\rangle\right) x_n$$

  - converges asymptotically for $0 < \rho < 2/\|x_n\|^2$ to θ_n^*.
  - if $\rho = 1/\|x_n\|^2$, the update steps directly to the solution θ_n^* in a single step.
- Training for multiple data points
  - $\chi = \{(x_n, y_n)\}_{n=1}^{N}$ where *N* is the size of the training set of *k* dimensions.
  - If *N* = *k* and the training data is linearly independent
    - we have *k* linearly independent constraints on *k* unknowns.
    - Hence there is a unique solution θ^* where $\forall n \quad \varepsilon_n = 0$.
    - LMS will converge to this solution by alternating between data points given that the step size is small enough.
  - If *N* > *k*, there is no unique solution.
    - LMS will converge to an approximate θ^* for small steps.

Minimizing Error

- Let y be a vector containing values y_n and let $\hat{y}$ be a vector with components $\langle \theta, x_n \rangle$: $\hat{y} = X\theta$
    - X can be viewed as spanning a vector subspace and $\hat{y}$ lies in that subspace whereas, generally, y does not (hence errors ε_n cannot all be 0).
- *Choosing orthogonal projection of y onto subspace spanned by X.*
    - difference vector $\varepsilon = y - \hat{y}$ must be orthogonal to the subspace. That is, $y - X\theta^*$ must be orthogonal to the columns of X:
      $$X^T\left(y - X\theta^*\right) = 0$$
    - **normal equations**:
      $$X^T X\theta^* = X^T y$$
- *Minimizing the least squares cost:*
    - least squares cost function:
      $$J(\theta) = \frac{1}{2}\sum_{n=1}^{N}\varepsilon_n^2 = \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \langle \theta, x_n \rangle\right)^2 = \frac{\left(y - X\theta\right)^T\left(y - X\theta\right)}{2}$$
    - gradient of $J(\theta)$:
      $$\nabla_\theta J = -X^T\left(y - X\theta\right)$$
    - setting gradient to 0 and solving yields *normal equations*:
      $$X^T X\theta^* = X^T y$$
- *Solving Normal Equations:*
    - if X is not full rank, this requires regularization.
    - if X is full rank, then $\left(X^T X\right)^{-1}$ is invertible since $X^T X$ is positive definite. Thus an exact solution can be obtained:
      $$\theta^* = \left(X^T X\right)^{-1} X^T y$$
        - Direct methods: Gaussian elimination, QR decomposition.
        - Iterative methods – converge in a finite number of steps to the optimal solution. e.g. LMS updates.