CS2109S

AY22/23 Sem 2

github.com/jasonqiu212

01. Introduction

- **Agent** - Anything that can perceive its environment through sensors and acting upon that env. through actuators
- **Agent Function** - Maps from percept histories to actions
- **Rational Agent** - Chooses an action that is expected to maximize its performance measure, given by percept sequence and built-in knowledge
- **Autonomous Agent** - If behavior is determined by its own expereince

Performance Measure of Function

- Motivation: For an agent to do the right thing, need a measure of goodness

Defining the Problem: PEAS

1. Performance measure
2. Environment
3. Actuators
4. Sensors

Characterizing the Environment

1. **Fully observable** - (vs. Partially) Agent's sensors can access complete state of env. all the time
2. **Deterministic** - (vs. Stochastic) Next state of env. is determined by **current state** and **action executed by agent**
   - **Strategic** - If env. is deterministic except for actions of other agents
3. **Episodic** - (vs. Sequential) Agent's experience is divided into atomic **episodes**, where each episode includes perceiving and an action, and **action depends on episode** itself
4. **Static** - (vs. Dynamic) Env. is unchanged while agent is deciding
   - **Semi** - Time does not affect env., but affects performance score
5. **Discrete** - (vs. Continuous) Discrete num. of percepts and actions
6. **Single Agent** - (vs. Multi-agent) Agent operating by itself in an env.

Implementing Agents (in ascending complexity)

1. **Simple Reflex Agents** - Fixed conditional rules
2. **Model-based Reflex Agents** - Stores percept history to make decisions about internal model of world with conditional rules. Eg. Roomba
3. **Goal-based Agents** - Keep in mind a goal and action aims to achieve it
4. **Utility-based Agents** - Find best way to achieve goal
5. **Learning Agents** - Learn from previous experiences

Exploitation vs. Exploration

- **Exploitation** - Maximize expected utility using current knowledge of world
- **Exploration** - Learn more about the world to improve future gains. May not always maximize performance measure.

02. Uninformed Search

- Deterministic, fully observable
- **Tree Search** - Can revisit nodes
- **Graph Search** - Tracks visited (Tree Search + Memoization)
- **Uninformed Search** - Uses only information available in problem definition

Formulating the Problem

1. How to represent state in problem?
2. Initial state
3. Actions: Successor function
4. Goal test
5. Path cost

- **Abstraction Function** - Maps abstracted representation to real world state
- **Representation Invariant** - $I(c) = \textbf{True} \rightarrow \exists a \textbf{ s.t. } AF(c) = a$

Breadth-first Search

- Idea: Expand shallowest unexpanded node using **queue**
- Given: b: Branching factor and d: Depth of optimal solution
- Complete: Yes (if tree is finite)
- Time: $O(b^{d+1})$
- Space: $O(b^d)$
- Optimal: Yes (if cost = 1)
- BFS is Uniform-cost Search with same cost

Uniform-cost Search

- Idea: Expand least-cost unexpanded node using **priority queue** (Dijkstra's)
- Given: C^*: Cost of optimal solution
- Complete: Yes (if step cost $\geq \epsilon$ where $\epsilon \geq 0$)
- Time: $O(b^{(C^*/\epsilon)})$ (C^*/ϵ is approx. number of layers)
- Space: $O(b^{(C^*/\epsilon)})$
- Optimal: Yes

Depth-first Search

- Idea: Expand deepest unexpanded node using **stack**
- Given: m: Maximum depth of tree
- Complete: No (fails with infinite depth or loops)
- Time: $O(b^m)$
- Space: $O(bm)$ (better than BFS)
- Optimal: No

Depth-limited Search

- Motivation: How to handle infinite depth for DFS?
- Idea: DFS with depth limit l where nodes at depth l have no children
- Time: $b^0 + b^1 + ... + b^{(d-1)} + b^d = O(b^d)$

Iterative Deepening Search

- Motivation: How to determine depth limit? We don't.
- Idea: Try different depths for depth-limited search
  - BFS pretending to be DFS to save space
- Complete: Yes
- Time: $(d+1)b^0 + db^1 + ... + b^d = O(b^d)$ (More overhead than DLS)
- Space: $O(bd)$

Summary

	BFS	Uniform Cost	DFS	DLS	IDS
Complete	Yes	Yes	No	No	Yes
Time	$O(b^d)$	$O(b^{C^*/\epsilon})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{C^*/\epsilon})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal	Yes	Yes	No	No	No

Bidirectional Search

- Idea: Search both forwards from initial state and backwards from goal state. Stop when searches meet.
- Time: $O(2b^{d/2})$
- Operators must be reversible
- Can have many goal states
- How to check if node intersects with other half?

03. Informed Search

Heuristic

- **Heuristic** - Estimated cost from n to goal
- **Admissible** - $h(n)$ is admissible if, for every node n, $h(n) \leq h^*(n)$ where h^* is the true cost
  - if h is admissible, then A* using tree search is optimal
- **Consistent** - $h(n)$ is consistent if, for every node n and every successor n' of n generated by action a, $h(n) \leq c(n, a, n') + h(n')$
  - Triangle inequality
  - If h is consistent, $f(n)$ is non-decreasing along any path ($f(n') \geq f(n)$)
  - If h is consistent, then h is admissible
  - if h is admissible, then A* using graph search is optimal

Dominance

- If $h_2(n) \geq h_1(n)$ for all n, then h_2 **dominates** h_1
- If h_2 **dominates** h_1 and both are admissible, then h_2 is better for search

How to invent admissible heuristic?

- Set fewer restrictions on actions
- E.g. Number of misplaced tiles, Total manhattan distance

Best-first Search

- Idea: Expand most desirable node using priority queue
- Evaluation Function: $f(n) = h(n)$
- Complete: No. Possible to be stuck in loop
- Time and space: $O(b^m)$
- Optimal: No

A* Search

- Idea: Take note of cost so far and heuristic
- Evaluation Function: $f(n) = g(n) + h(n)$ where $g(n)$ is cost to reach n
- Complete: Yes, unless non-increasing, since cost is factored in
- Time and space: Same as BFS
- Optimal: Yes, depending on the heuristic

Iterative Deepening A* Search (IDA*)

- Motivation: How can we save space?
- Idea: Have a cutoff for f and remember the best f that exceeds cutoff
  - Similar to IDS. Linear space complexity.
- Optimal and complete

Simplified Memory A* Search (SMA*)

- Motivation: How can we save space?
- Idea: Do normal A*. If memory is full, drop node with worst f.
- Lose completeness

Local Search

- Motivation: What if the goal state is the solution? The path is irrelevant.
- Idea: Keep single current state and try improving it
- Formulating the problem:
  1. Initial state
  2. Actions: Successor function
  3. Good heuristic
  4. Goal test

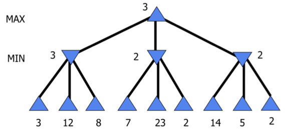
Hill-climbing

- Idea: Generate successors from current and pick the best using heuristic
- What if stuck into local minima?
  - Introduce randomness
  - **Simulated Annealing Search** - Allow some bad moves and gradually decrease frequency
- Why only keep 1 best state?
  - **Beam Search** - Perform k hill-climbing searches in parallel
- How to generate successors?
  - **Genetic Algorithms** - Successsor is generated by combining 2 parent states

04. Adversarial Search

- Assumption: Opponents reacts rationally
- Formulating the problem:
  - Initial state
  - Successor function
  - Terminal test
  - Utility function: Measures how good the move is for a player

Minimax



- Idea: Choose move that yields highest minimax value using DFS
- Complete: Yes (if tree is finite)
- Time: $O(b^m)$
- Space: $O(bm)$
- Optimal: Yes (against an optimal opponent)

Alpha-Beta Pruning

- Motivation: How to save time for Minimax?
- Idea: By tracking max. and min. values so far, can prune some paths that we would never choose
  1. α contains max. and β contains min.
  2. Initially, $\alpha = -\infty$ and $\beta = \infty$
  3. When going down, copy α and β
  4. Prune if $\alpha \geq \beta$
  5. When going up, depending on MIN/MAX level, update α or β
- With perfect ordering, time complexity: $O(b^{m/2})$. Doubles search depth.

Resource Limits

- In reality, search space for games can be very large. α-β pruning also not fast enough.
- Solution: Limit depth (Only see a finite moves ahead) and determine best move using evaluation function to estimate desirability of position (Heuristic)
- Other hacks:
  - **Transpoisitions** - Memoize equivalent states
  - Pre-computation of opening/closing moves

05. Introduction to Machine Learning

- A machine learns if it improves performance P on task T based on experience E. Where T must be fixed, P must be measurable, E must exist

Types of Feedback

- **Supervised** - Correct answer given for each example
  - **Regression** - Predict results within continuous output
  - **Classification** - Predict results in discrete output
- **Unsupervised** - No answers given
- **Weakly supervised** - Answer given, but not precise
- **Reinforcement** - Occasional rewards given

Decision Trees

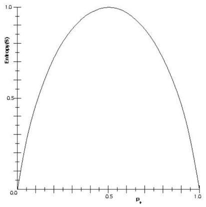
- DT can express any function of input attributes, if data is consistent
- Goal: Make DT **compact**. How?

Information Theory

- Idea: Choose attribute that splits examples into subsets that are ideally 'all positive' or 'all negative'
- **Entropy** - Measure of randomness in set of data

$$I(P(v_1),...,P(v_n)) = -\sum_{i=1}^{n} P(v_i)\log_2 P(v_i)$$

- For data with p positive examples and n negative examples:



$$I(\frac{p}{p+n},\frac{n}{p+n}) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}$$

- **Information Gain** - (IG) Reduction in entropy from attribute test
- Goal: Choose attribute with largest information gain
- Intuition: IG = Entropy of this node - Entropy of children nodes
- Given chosen attribute A with v distinct values:

$$\text{remainder}(A) = \sum_{i=1}^{v}\frac{p_i+n_i}{p+n}I(\frac{p_i}{p_i+n_i},\frac{n_i}{p_i+n_i})$$

$$IG(A) = I(\frac{p}{p+n},\frac{n}{p+n}) - \text{remainder}(A)$$

- **Decision Tree Learning** - Recursively choose attributes with highest IG
- IG is not the only way. Can use whatever objective function that achieves the criteria we want.

Performance Measurement

- **Correctness** - Correct if $\hat{y} = y$
- **Accuracy** - $\frac{1}{m}\sum_{j=1}^{m}(\hat{y_j} = y_j)$
- Confusion Matrix:

		Actual Label	
		+ve	-ve
Predicted Label	+ve	TP True Positive	FP False Positive
	-ve	FN False Negative	TN True Negative

- Accuracy = $\frac{TP+TN}{TP+FN+FP+TN}$
- **Precision** - $\frac{TP}{TP+FP}$ How precise are positive predictions?
- **Recall** - $\frac{TP}{TP+FN}$ How many actual positives are predicted?
- **F1 Score** - $\frac{2}{1/P+1/R}$ Harmonic mean of precision and recall

- Type I Error: FP, Type II Error: FN
- FP Rate = $\frac{FP}{FP+TN}$ TP Rate = $\frac{TP}{TP+FN}$

Pruning

- Motivation: DT overfits to training set, but performs poorly on test set
- Occam's Razor: Simple hypothesis preferred
- **Pruning** - Ignores outliers, which reduces overfitting
  - Idea: Go with the majority of T/Fs
  - E.g. Min-sample, Max-depth

06. Linear Regression

Notation

- m = Number of training examples
- n = Number of features
- $x_j^{(i)}$ = Input feature j of ith training example
- y = Output variables

Hypothesis

$$h_w(x) : w_0 + w_1 x$$

Cost Function (Square Error Function)

$$J(w_0, w_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)})^2$$

- Goal: Minimize cost function. Thus, hypothesis is close to training samples
- Why squared error? Convenience, since we need to differentiate later

Gradient Descent

1. Start at some (w_0, w_1). Pick nearby point that reduces $J(w_0, w_1)$.
2. Repeat until convergence:

$$w_j := w_j - \alpha \frac{dJ(w_0, w_1, \ldots)}{dw_j}$$

- All updates done at end
- How to do $\frac{dJ(w_0,w_1)}{dw_j}$? Partial derivative: Hold everything else constant
  - $\frac{dJ(w_0,w_1)}{dw_j} = \frac{d}{dw_j}(\frac{1}{2m}\sum_{i=1}^{m}(w_0 + w_1 x^{(i)} - y^{(i)})^2)$
  - $\frac{dJ(w_0,w_1)}{dw_0} = \frac{1}{m}\sum_{i=1}^{m}(w_0 + w_1 x^{(i)} - y^{(i)})$ (Note: Chain rule)
  - $\frac{dJ(w_0,w_1)}{dw_1} = \frac{1}{m}\sum_{i=1}^{m}(w_0 + w_1 x^{(i)} - y^{(i)})x^{(i)}$
- Time complexity: $O(kmn)$ where k is number of iterations

Learning Rate

- If α too small, then descent is too slow. If α too big, then might overshoot.
- Given constant α, descent will grow smaller as we approach minimum

Variants of Gradient Descent

- Batch gradient descent: Consider all training examples when updating
- Stochastic gradient descent: Consider 1 random data point at a time (Cheaper and more randomness)
- Mini-batch gradient descent

Using Matrices

- Given: $w = \begin{pmatrix} w_0 \\ \vdots \\ w_n \end{pmatrix}$ and $x = \begin{pmatrix} x_0 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} 1 \\ \vdots \\ x_n \end{pmatrix}$
- $h_w(x) : w^T x$

Feature Scaling

- Motivation: Gradient descent does not work well if features have different scales
- **Mean Normalization** - $x_i \leftarrow \frac{x_i - \mu_i}{\sigma_i}$

Normal Equation

$$w = (X^T X)^{-1} X^T Y$$

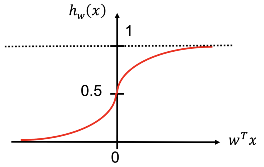
- No need to choose α and feature scaling
- $X^T X$ needs to be invertible
- Time complexity: $O(n^3)$. Slow if n is big

07. Logistic Regression

- Motivation: Classification with continuous input values.
- Idea: Come up with a decision boundary to separate data points. $h_w(x_1, x_2) = 1$, if $w_0 + w_1 x_1 + w_2 x_2 > 0$, or 0, otherwise

Sigmoid Function

- Motivation: Step function is discontinuous and not differentiable



$$h_w(x) = g(w^T x)$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

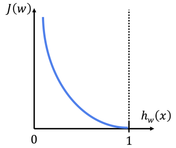
Hypothesis

$$h_w(x) : \frac{1}{1 + e^{-w^T x}}$$

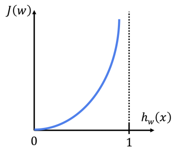
- Interpretation: Estimated probability that $y = 1$ given input x

Cost Function

- Problem: Least square error gives non-convex cost function, which is bad for G.D.
- Solution: Use log



- If $y = 1$, let cost be $-log(h_w(x))$
  - $h_w(x) \rightarrow 0, J(w) \rightarrow \infty$
  - $h_w(x) \rightarrow 1, J(w) \rightarrow 0$



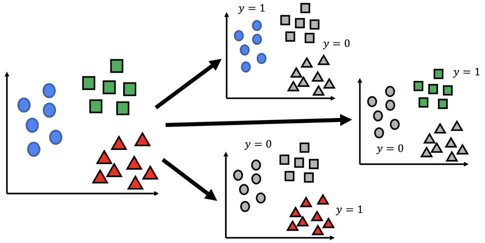
- If $y = 0$, let cost be $-log(1 - h_w(x))$
  - $h_w(x) \rightarrow 0, J(w) \rightarrow 0$
  - $h_w(x) \rightarrow 1, J(w) \rightarrow \infty$

$$J(w) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} log h_w(x^{(i)}) + (1 - y^{(i)}) log(1 - h_w(x^{(i)}))$$

Gradient Descent

- Same algorithm as linear regression
- $\frac{dJ(w)}{dw_j} = \frac{1}{m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})x_n^{(i)}$ (Same as linear regression)

Multi-class Classification



1. Train binary classifier $h_w^{(i)}(x)$ for each class i to predict $y = i$
2. For each input x, pick class i is greatest (i.e. $\max_i h_w^{(i)}(x)$)

08. Model Evaluation and Selection

Linear Regression

$$J_{\text{test}}(w) = \frac{1}{2m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} (h_w(x_{\text{test}}^{(i)}) - y_{\text{test}}^{(i)})^2$$

Logistic Regression

$$\text{error}(h_w(x), y) = \begin{cases} 1 & \text{if misclassification} \\ 0 & \text{otherwise} \end{cases}$$

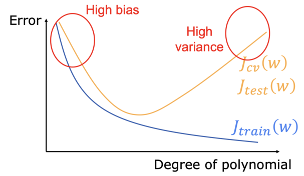
$$\text{Test Error} = \frac{1}{m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} \text{error}(h_w(x_{\text{test}}^{(i)}), y_{\text{test}}^{(i)})$$

Model Selection

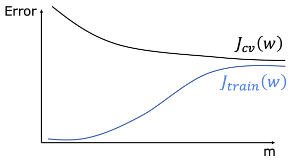
- How to choose model? (i.e. x^2, x^3, etc. for hypothesis)
  1. Split data into 3 sets: Training set, validation set, and test set
  2. Train each model using training set
  3. Compute $J_{cv}(w)$ for each model and pick model with lowest $J_{cv}(w)$
  4. Use $J_{\text{test}}(w)$ to estimate performance on unseen samples

Bias and Variance

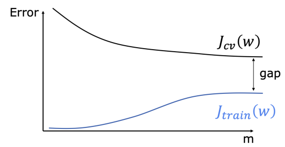
- **High bias** - Underfit, **High variance** - Overfit
- As degree of polynomial increases:



Impact of m: High Bias



Impact of m: High Variance



- Observation: If overfit, more training samples does not help
- How do we address overfitting?
  - Reduce number of features
  - **Regularization** - Keep all features, but reduce corresponding weights

09. Regularization

- Idea: Include w inside $J(w)$ to minimize weights and get simpler hypothesis
- What if λ is too large? Weights tend to 0, resulting in straight line
- How to choose λ? Use validation set
  - Small λ: $J_{\text{train}}(w)$ low and $J_{cv}(w)$ high (Overfit)
  - Large λ: $J_{\text{train}}(w)$ high and $J_{cv}(w)$ high (Underfit)

Linear Regression with Regularization

$$J(w) = \frac{1}{2m}(\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{i=1}^{n} w_i^2)$$

$$w_n := w_n - \frac{\alpha}{m}\sum_{i=1}^{m}((h_w(x^{(i)}) - y^{(i)})x_n^{(i)}) - \frac{\alpha\lambda}{m} w_n$$

Logistic Regression with Regularization

$$J(w) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} log(h_w(x^{(i)})) + (1-y^{(i)}) log(1-h_w(x^{(i)})) + \frac{\lambda}{2m}\sum_{i=1}^{n} w_i^2$$

$$w_n := w_n - \frac{\alpha}{m}\sum_{i=1}^{m}((h_w(x^{(i)}) - y^{(i)})x_n^{(i)}) - \frac{\alpha\lambda}{m}w_n$$

$$w = (X^TX + \lambda I)^{-1}X^TY \text{ where } I\text{'s first column is 0}$$

- X^TX can be non-invertible

10. Support Vector Machine

- Idea: Maximize margin between positive and negative samples

Decision Rule

$$w \cdot x \geq c \rightarrow \text{Positive}; w \cdot x < c \rightarrow \text{Negative}$$

- Dot product: $u \cdot v = u^Tv = p||v||$ (By law of cosine)
- Intuition: Decision boundary $\perp$ Weight vector. $w \cdot x = p||w||$. Sample a is on decision boundary if $w \cdot a = c$
- Constraints: Let $b = -c$
  - $w \cdot x + b \geq 1$ if $y = 1$
  - $w \cdot x + b < -1$ if $y = 0$
  - Combined: Let $\hat{y}^{(i)} = 1$ or -1 for pos. and neg. samples respectively. $\hat{y}^{(i)}(w \cdot x^{(i)} + b) \geq 1$
  - If x is inside margin, then $w \cdot x + b = 1$ or -1 respectively
  - Why add these constraints? Mathematical convenience for hinge loss

Margin

$$\text{Margin width} = \frac{2}{||w||}$$

- Let x^+ and x^- be closest positive and negative samples
- Margin width $= (x^+ - x^-) \cdot \frac{w}{||w||}$ (i.e. Length of projection of $x^+ - x^-$ on weight vector) $= \frac{1-b+1+b}{||w||}$ (since x^+ and x^- are inside margin) $= \frac{2}{||w||}$

Objective Function for Hard-Margin

$$\min \frac{1}{2}||w||^2 \text{ s.t. } \hat{y}^{(i)}(w \cdot x + b \geq 1)$$

1. Maximize margin: $\max \frac{2}{||w||} = \min ||w|| = \min \frac{1}{2}||w||^2$
2. Classify correctly

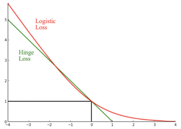
- **Hard-Margin** - No samples inside margin
- Before, we assume hard margin. What if there exists outliers that causes SVM to overfit?

Objective Function for Soft-Margin

$$h_w(x) = \begin{cases} 1 & \text{if } w^Tx \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

- **Slack Variable** - (ξ) Loss of misclassified point
- Goal: Maximize margin and allow misclassification by tweaking C
  - Large C overfits (Basically hard-margin). Small C underfits.
  - i.e. $\min(\frac{1}{2}||w||^2 + C\sum_i \xi^{(i)})$ s.t. $\forall i, \xi^{(i)} \geq 0$ and $\hat{y}^{(i)}(w \cdot x^+ b) \geq 1 - \xi^{(i)} \rightarrow J(w) = C\sum_i \max(0, 1 - \hat{y}^{(i)}(w^Tx^{(i)})) + \frac{1}{2}\sum_{i=1}^{n} w_i^2$
- Hard-margin: Must follow constraint. Soft-margin: Constraint is flexible with slack variable, so can define cost function to minimize.

$$J(w) = C\sum_{i}^{m} y^{(i)}\text{cost}_1(w^Tx^{(i)}) + (1-y^{(i)})\text{cost}_0(w^Tx^{(i)}) + \frac{1}{2}\sum_{i=1}^{n} w_i^2$$



- **Hinge Loss**
  - $\text{cost}_1(z) = \max(0, 1-z)$
  - $\text{cost}_0(z) = \max(0, 1+z)$

Kernel

- Motivation: What if data is not linearly separable?
- Idea: Map features to higher dimensions (i.e. $\phi(x)$). High dim. is slow.
- **Kernel Function** - Dot product between two points in transformed dim.

$$K(u,v) = \phi(u) \cdot \phi(v)$$

- Property: Getting $K(u,v)$ does not need $\phi(u)$ and $\phi(v)$
- **Linear Kernel** - $K(u,v) = u \cdot v$
- **Polynomial Kernel** - $K(u,v) = (u \cdot v)^d$
- **Gaussian Kernel** - $K(u,v) = e^{-||u-v||^2/2\sigma^2}$ (Note: $\phi(u)$ can map to infinite dimensions)

Kernel Trick

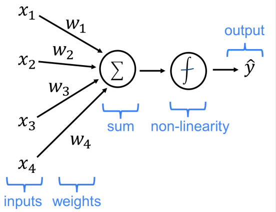
- By manipulating SVM objective func. and decision rule, $x^{(i)} \cdot x^{(j)}$ emerges
- If we map features (i.e. $\phi(x)$), can replace above term with kernel ($K(u,v)$)
- Trick: No need to compute transformed features explicitly

Similarity Features

- Idea: Compute new features based on proximity to landmarks
- Given example x. $f_i = sim(x, l^{(i)}) = K(x, l^{(i)})$ where K is gaussian kernel
- Transformed x: $f = [f_0, f_1, \ldots, f_m]^T$. Plug into obj. function.
- Do feature scaling, else similarity value may be dominated by some features
- Large σ underfits (Smoother features). Small σ overfits.
- Not all similarity functions get valid kernels. Need to satisfy Mercer's Thm.

11. Neural Networks

- Idea: Models after human brain (Perceptron)



$$\hat{y} = g(\sum_{i=0}^{n} w_ix_i) \text{ where } x_0 = 1$$

- **Activation Function** - (g) Non-linear function (e.g. Step func. $sgn(x)$, $\sigma(x)$, $tanh(x)$, $max(0,x)$)

- Supports both classification and regression. Depends on activation func.
  - If activation function is linear, then regression

Perceptron Learning Algorithm (PLA)

1. Initialize weights w
2. Select misclassified sample and update w. Repeat until convergence.

$$w \leftarrow w + \eta(y - \hat{y})x$$

- Why this works? Each update gets closer to correct classification
- Different from GD!
- Ends when no more misclassification. Thus, can select any linear model
- Cannot converge on non-linear data

Single-Layer Neural Network

- Basically logistic regression, if activation function is $\sigma(x)$
- How to determine weights? Gradient descent
  - Many ways to calculate error. e.g. MSE ($J(w) = \frac{1}{2}(\hat{y} - y)^2$).
  - Given $\hat{y} = g(f(x)); f(x) = w^Tx$, $\frac{dJ(w)}{dw_i} = \frac{dJ}{dy}\frac{d\hat{y}}{df}\frac{df}{dw_i} = (\hat{y} - y)g'(f)x_i$

$$w_i := w_i - \eta(\hat{y} - y)g'(f)x_i$$

- If activation function $g = \sigma(x)$, then $g'(x) = \sigma'(x) = \sigma(x)(1 - \sigma(x))$

$$w_i := w_i - \eta(\hat{y} - y)\hat{y}(1 - \hat{y})x_i$$

Forward Propagation

- How to fit non-linear data? Have many layers. 1 layer computes more complex features as next layer's input
- Given layer l:
  - Vector: Bolded, lower case. Matrix: Bolded, upper case

$$\mathbf{a}^{[l]} = \mathbf{g}^{[l]}(\mathbf{f}^{[l]})$$

$$\mathbf{f}^{[l]} = (\mathbf{W}^{[l]})^T\mathbf{a}^{[l-1]}$$

Backpropagation

- Big idea: Gradient descent with lots of chain rules
- Goal: How to get $\frac{d\epsilon}{d\mathbf{w}}$?
- E.g. $\frac{d\epsilon}{dw_2} = \frac{d\epsilon}{d\hat{y}}\frac{d\hat{y}}{dw_2}$ and $\frac{d\epsilon}{dw_1} = \frac{d\epsilon}{d\hat{y}}\frac{d\hat{y}}{da_1}\frac{da_1}{dw_1}$
- But, very costly to do this for many layers. Can we vectorize it?

$$\hat{y}'(\mathbf{W}^{[l]}) = \frac{d\mathbf{f}^{[l]}}{d\mathbf{W}^{[l]}}(\delta^{[l]})^T = \mathbf{a}^{[l-1]}(\delta^{[l]})^T$$

$$\delta^{[l]} = \frac{dg^{[l]}}{d\mathbf{f}^{[l]}}\frac{d\mathbf{f}^{[l+1]}}{dg^{[l]}}\delta^{[l+1]} = g'^{[l]}(\mathbf{f}^{[l]})\mathbf{W}^{[l+1]}\delta^{[l+1]}$$

12. Deep Learning