Introduction to AI

Dedicated to Bayes

November 18, 2020

1 Machine Learning

1.1 Linear Prediction L2,L3

- **Supervised Learning** - Information is labelled, i.e. consists of input-output pairs. **Output** is sometimes referred to as variates/labels/target and **Input** is sometimes referred to as features/predictors.
  - **The Supervised learning setting**: There is a training data set D_{train} of input-output pairs $\{\mathbf{x}_{1:n}, y_{1:n}\}$. Each input $x_i \in \mathcal{R}^{1\times d}$ has d attributes. *See how a table is formed with this notation*, $(\{x_{1:n,1:d}, y_{1:n}\})$. The output/target is y_i, and depending on what y_i is, there are different prediction tasks:
  - Notation is key! $\mathbf{X}$ means $\mathbf{X} = \begin{bmatrix} \mathbf{x_1} \\ \vdots \\ \mathbf{x_n} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & ... \\ \vdots & \vdots & \vdots \\ x_{n1} & x_{n2} & ... \end{bmatrix}$ (1)
    - **Regression**: y_i is a real number.
      - **Loss function**: $J(\theta) = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$ (residual of least squares), where y_i is the true output and $\hat{y}_i$ is the estimated output, i.e. $\hat{y}_i = \sum_{j=1}^{d} x_{ij}\theta_j$, assuming $x_{i1} = 1$. In matrix form its $\boldsymbol{\hat{y} = X\theta}$, with $\boldsymbol{y \in R^{n\times 1}}, \boldsymbol{X \in R^{n\times d}}$ and $\boldsymbol{\theta \in R^{d\times 1}}$. $\boldsymbol{J(\theta) = (\hat{y} - X\theta)^T(\hat{y} - X\theta)}$
      - **Optimisating parameters**: Want to minimise loss function $\hat{\theta} = argmin_\theta J(\theta)$. $\boldsymbol{\hat{\theta}_{LS} = (X^T X)^{-1} X^T y}$
    - **Classification** y_i is yes/no (binary), one of K labels (multiclass), a subset of K labels (multilabel).
      - $y = sign(\boldsymbol{x^T\theta}), y \in \{+1, -1\}$, note $\boldsymbol{\theta}$ are the weights of the line that split the data, i.e. classify the data. I think $\boldsymbol{\theta}$ is normal to the decision boundary.
      - **Score**: Confidence in prediction: $score = f_\theta(X) = \boldsymbol{x^T\theta}$. The bigger that $|\boldsymbol{x^T\theta}|$ is the more confident we are.
      - **Margin**: How correct the prediction is: $margin = y.\boldsymbol{x^T\theta} \longrightarrow$ Negative means incorrect and positive means correct.
      - **Loss function** $L(y, f_\theta(\mathbf{x}))$ is measure of how unhappy you would be if you used θ to make a prediction on $\mathbf{x}$ when the correct output is y.
      - **Loss Minimisation**: $argmin_\theta TrainLoss(\theta) = argmin_\theta \sum_{x,y\in D_{train}} Lossfunction$
      - **Grandient Descent**: Uses $\nabla TrainLoss(\theta)$ (the direction that increases the loss the most) - computationally very slow!
      - **Stochastic Gradient Descent** - uses $\nabla Lossfunction$. Much more efficient and easy to implement.
    - **Ranking**: y_i is a permutation.

Perceptron Algorithm

- Initialise $t = 1$, $\theta = [0, 0, ...]^T$ and normalise $||\mathbf{x}|| = 1$.
- Given example $\mathbf{x}_i$ iff $\mathbf{x_i}^T\boldsymbol{\theta} > 0$ predict positive class, else predict negative.

- On mistake, i.e. $y_i \neq sign(\mathbf{x_i}^T \boldsymbol{\theta})$ then if mistake
  - on positive (i.e. $sign(\mathbf{x_i}^T \boldsymbol{\theta}) \leq 0$, but $y_i > 0$) then: $\boldsymbol{\theta^{t+1}} = \boldsymbol{\theta^t} + \mathbf{x}_i$
  - on negative (i.e. $sign(\mathbf{x_i}^T \boldsymbol{\theta}) > 0$, but $y_i < 0$) then: $\boldsymbol{\theta^{t+1}} = \boldsymbol{\theta^t} - \mathbf{x}_i$
- If classification correct then $\boldsymbol{\theta^{t+1}} = \boldsymbol{\theta^t}$
- Then repeat ($t = t + 1$) until convergence.

Algorithm will converge if the data is **linearly separable** (even just if its just one element in many of the class). $x^T \theta^{t+1} = x^T(\theta^t \pm x) = x^T\theta \pm 1$ so will converge in steps of one.

1.1.1 Generalisation

- Loss minimisation optimises the parameters on the training data but does not necessarily perform well on unseen data (i.e. prediction)
- Best thing is to split your training data into training and test data.
- Test error = Bias + Variance $\longrightarrow$ **Bias** is how wrong the predictions are from the true value (caused by taking too little information into consideration - underfitting) and **Variance** is how spread out they are (caused by too complex models that take too much noise into consideration - overfitting). The aim is to have a classifier with low bias and low variance.
  - Bias quantifies precision, while Variance quantifies sensitivity.
- Best model is where the test error is the lowest!
- **Hypothesis Class** - set of possible classification functions (linear predictors) you're considering. (Small hypothesis class implies small polynomial degree fitting models, which implies high bias, but low variance (underfitting), while large hypothesis class implies large polynomial degree fitting models - high variance, low bias (overfitting))
- Controlling size of hypothesis class $\longrightarrow$ 1) reduce dimensionality of parameter space $\boldsymbol{\theta}$ (add/remove features if they reduce test error) 2) Keep norm of $\boldsymbol{\theta}$ small (Regularisation)
  - For Regularisation:
    - Could add term to objective function that penalised length of $\boldsymbol{\theta}$. Which leads to Ridge regression solution $\longrightarrow \boldsymbol{\hat{\theta}_{LS} = (X^T X + \lambda I_d)^{-1} X^T y}$ (Levenberg-Marquard no?)
    - Early stopping: Stop when test error of training cycles gets bigger.
- Extra parameters (hyperparameters) should not be chosen to minimise either the test or training error - as this will lead to an unreliable estimate of error.
- **Validation** - randomly take out 10 - 50% of training data and make it test data (validation data).
- **Cross validation** (K-fold cross validation) - split the data and iterate through it using one segment as validation data and the rest as training data.
- If Algorithm shows High Bias (underfitting) $\longrightarrow$ add more features/ use more sophisticated model/ use fewer examples/ decrease regularisation.
- If algorithm shows High Variance (overfitting) $\longrightarrow$ use fewer features/ use more training features/ increase regularisation.

1.2 Unsupervised Learning L5

The unsupervised learning setting: The training data set D_{train} only contains input $\mathbf{x}$:

- **Clustering**: Partitioning the data such that *similar* data is in the same cluster and *dissimilar* data are in different clusters. The partitioning is represented by an assignment vector $\mathbf{z} = [z_1, ..., z_n]$, such that $z_i \in \{1, ..., K\}$. E.g. suppose there are two clusters $\{C_1, C_2\}$ and 5 data points, then $\mathbf{z}$ could be for example $\mathbf{z} = [C_1, C_2, C_1, C_1, C_1]$. The index of $\mathbf{z}$ corresponds to data point.
- **Performance Assessment**: Good clustering algorithm means 1) **intra**-cluster/class similarity is high and 2) **inter**-cluster/class similarity is low.

———— K-means Clustering ————

- Objective: $min_{\mathbf{z}} min_{\boldsymbol{\mu}} Loss_{Kmeans}(\mathbf{z}, \boldsymbol{\mu}) = min_{\mathbf{z}} min_{\mu} \sum_{i=1}^{n} ||\mathbf{x} - \boldsymbol{\mu}_{\mathbf{z}_i}||^2$
- Algorithm: 1) Choose number of partitions/clusters (K) 2) (Randomly) Assign values to centroids $\boldsymbol{\mu}_{\mathbf{z_i}}$ 3) Find assignments given these centroids 4) Find centroids given these assignments 5) Repeat 4,5 until nothing changes anymore (converges).
- K-means is guaranteed to find the local optimum but not necessarily the global optimum.

- **Dimensionality reduction**: The idea is too reduce the number of dimensions while retaining as much information as possible.

——— Principle Component Analysis ————

- The idea is that the most interesting properties in the data are found where there is the most variance (highest Eigenvalues of principle components).
- Find the principle components by finding the eigenvalues (λ_k) and eigenvectors ($\mathbf{w}_k$) from the matrix $\mathbf{X}^T\mathbf{X}$, where $\mathbf{X}$ is like (1).
- Normalise $\mathbf{w}$ and finally project the data onto the principle eigenvalues that you want, in order to reduce the dimensionality: $\mathbf{V} = \mathbf{XW}$, where $\mathbf{W} = [\mathbf{w}_1, ...\mathbf{w}_k] = \left[\begin{bmatrix} w_{11} \\ w_{12} \\ \vdots \end{bmatrix}, \begin{bmatrix} w_{21} \\ w_{22} \\ \vdots \end{bmatrix}, ... \right]$
- It is important to **standardise** the data first, PCA is sensitive to relative scaling: Find mean $x_j = \frac{1}{n}\sum_{i=1}^{n} x_{ij}$, then variance $\sigma_j^2 = \frac{1}{n}\sum_{i=1}^{n}(x_{ij} - x_j)^2$ and finally $x_{ij} = \frac{x_{ij} - x_j}{\sigma_j}$
- Reconstruction error is the distance from the data projected onto an eigenvector line to the orginal data (e.g. if the data are all on a line then the reconstruction error is zero because the principle component will pass through all the points).

1.3 Non-parametric models

———— Nearest Neighbours classifier ————

- **Algorithm** Training: Store training data. Prediction: Given $\mathbf{x}_{new}$, find $(\mathbf{x}, y)$ such that $||\mathbf{x} - \mathbf{x}_{new}||$ is smallest, then return y.
- Non-parametric because it adjusts to data not parameters.
- Visualise using Voronoi diagram - space is partitioned such that each region has the same closest point.
- (NN can also be used for regression - called piecewise constant interpolation)

- K-NN you go by radial regions until you find K neighbours more than another kind and assign it to that class.
- KNN can overfit when the distance function is set to work well for the training data but then fails for the test data (because every point in the data set defines it's own decision boundary). To overcome overfitting, cross validation can be used to adjust K and distance metric. Increasing K generally should reduce overfitting as well.

1.4 Probability theory L6

- Assume you know how to work with marginal, joint and conditional probabilities by now.
- **Support**, let $\mathbf{S_X}$ be the support of $\mathbf{X}$, where $\mathbf{S_X} = \{x \in \mathbf{X} : p(x) > 0\}$. Essentially its the alphabet for which probabilities exist.
- Expectation: $E[X] = \sum_x xp(x)$, $E[g(x,y)] = \sum_{x,y} g(x,y)p(x,y)$ (can also take expectation w.r.t one variable only too).
- Variance: $E[(X - E[X])^2] = E[X^2] - (E[X])^2$
- Independence: X, Y are independent if $p(x,y) = p(x)p(y), \forall x, y$ (or $p(x|y) = p(x)$)
- **Conditional Independence**: X is conditionally independent of Z given Y if: $p(x, z|y) = p(x|y)p(z|y)$ or $p(x|z, y) = p(x|y)$.
- **Expected Prediction Error**: $EPE(f) = \int \int Lossfunction(f(x), y)p(x, y)dxdy$
- For Least Squares: $f^*(x) = E[Y|X = x]$. For K-Nearest neighbours: $f^*(x) = Ave(y_i|x_i \in N_k(x))$, where $N_k(x)$ are the k nearest neighbours of x. So both least squares and nearest neighbours compute conditional expectations by averages. However, least squares assumes $f(x)$ is well approximated by a globally linear function, while K-NN assumes f(x) is well approximated by a locally constant function.

1.5 Information Theory L7

Idea: Maximising the amount of information that can be transmitted via imperfect communication channels.

- Entropy: The uncertainty associated with a distribution: $H(X) = -\sum_x p(x) \log_2(p(x))$. (It is lower bound on the number of bits needed to transmit a message fully).
  - Entropy is the measure of average uncertainty in the random variable.
  - Entropy is the average number of bits needed to describe the random variable.
  - Entropy is the lower bound of the average description of the shortest description of the random variable.
  - Entropy is always non-negative ($H(X) \geq 0$). There is an upper-bound ($H(X) \leq log_2(|\text{cardinality of sample space}|)$ iff X is a uniform distribution.
  - Use $H_b(X) = log_b(a)H_a(X)$ to switch bases. Therefore, don't need to specify the base (base 2 entropy called 'bits', base e entropy called 'nats').
- Joint Entropy: $H(X, Y) = -\sum_x \sum_y p(x, y) \log(p(x, y))$
- Conditional Entropy $\boxed{H(X|Y) = H(X, Y) - H(Y)} = \boxed{-\sum p(Y) \sum p(X|Y) log_2(X|Y)}$. Note, $H(X|Y) \neq H(Y|X)$!
- Mutual information: Measures how much information is in common between distributions X and Y: $I(X;Y) = H(X) + H(Y) - H(X, Y)$. Note $I(X;Y) = I(Y;X)$ and $I(X;Y) = 0$ iff X, Y are independent. Also $I(X;X) = H(X) - H(X|X) = H(X)$ so entropy is also considered as self-information.

- Kullback-Liebler Divergence: A distance measure between two probability distributions: $KL(p||q) = \sum p(x) \log \frac{p(x)}{q(x)}$. Properties: anti-symmetric ($KL(p||q) \neq KL(q||p)$) and can be related to mutual information as $I(X;Y) = KL(p(x,y)||p(x)p(y))$

————————-t-SNE Algorithm ————————-

- Find distance between data, then find joint probability that measures pairwise similarity, finally minimise KL divergence.
- (Think this might be what we were doing in Coursework 2 in Machine Learning).
- Not guaranteed to find global optimum.

————————————————————————————-

1.6 Decision Trees L8

- Structure: Hierarchical, root node to internal nodes (split nodes) to leaf nodes.
- Idea: It's essentially a function that takes as its input attributes/features and returns as an output a decision. You know the algorithm.
- The aim is to find a tree (through training data) that has the optimal amount of nodes, with the most significant attribute nodes that split the training data best being closer to the root.*A good attribute splits the data into subsets that are (ideally) all positive or all negative.*
- Thus we choose attributes to split on that have the largest information gain. $Gain(A) = H(Goal) - Remainder(A)$, where $H(Goal)$ is the entropy of the goal attribute for the hole set. For binary case this is $H(\frac{p}{p+n}) = H(0.5) = 1$ Bit always. $Remainder(A) = \sum_{i=1}^{k} \frac{p_k+n_k}{p+n} H(\frac{p_k}{p_k+n_k})$ which is essentially the expected entropy (uncertainty) remaining after splitting on attribute A).
- Problems: Tree size (over fitting the data can be an issue so the tree needs to be 'pruned'), Instability (usually have high variance due to hierarchical nature), can be computationally expensive.
- Practically how to do it: Consider data $XXXOOOO$ and it gets split into e.g. yes:XOO| no:$XXOO$ then the information gain is $H(goal) - Remainder(A) = (3/7log(7/3) + 4/7log(7/4)) - [3/7(1/3log(3/1) + 2/3log(3/2)) + 4/7(2/4log(4/2) + 2/4log(4/2)]$. This should clarify all the math.
- Decision trees can also be used for regression.

1.7 Random Forests L8

- Idea: Create random decision trees from a subset of training data (usually around 66%) to form a combined (using weighted averages or majority votes) stronger more robust and generalised tree.
- Strengths: Computationally fast. Weaknesses: Overfit easily when used for regression tasks.

2 Search L9

- The idea is to find a set of actions that optimise a problem. It works by decomposing a complex problem into smaller simpler ones.
- Lingo: s_{start} - starting state, $Actions(s)$ - possible actions from state s, $Cost(s,a)$ - cost of performing action a from state s, $Succ(s,a)$ - successors (states that follow on) from state s given action a, $Goal(s)$ - true is s is a goal state.
- Performance measures: Completeness (does the search algorithm always find a solution), Optimalilty, Time Complexity, Space complexity. The parameters used to define these are: b - branching factor (max number of successors of any node, **think of it is the maximum number of possible moves**), d - depth from root node to nearest goal node (solution depth), D - length of the longest path (max depth)

The following (incl graph search) are uninformed (blind) search methods as they use no additional information (heuristics).

2.1 Tree Search

- **Backtracking Search** - Check every path. Go from left to right always going from a start node to an end node. Time complexity: $O(b^D)$ (huge), Space Complexity (Memory needed) $O(D)$ (small).
- **Depth-first Search (DFS)** - Backtracking search + stop when first goal state is reached. It assumes $Cost(s, a) = 0$ throughout the tree. It 1) goes to all the way to a leaf node (depth-first) and then 2) goes along a different branch and then repeats those steps. Time complexity: $O(b^D)$ (huge), Space Complexity (Memory needed) $O(D)$ (small). Not complete, not optimal.✓
- **Bredth-first Search (BFS)** - explore all nodes at each level of depth (i.e. go across branches from left to right at the same depth). It assumes $Cost(s, a) = const. = c, c \geq 0$ throughout the tree. Time complexity: $O(b^d)$ (potentially better - if there are longer paths than from the starting node to a goal node), Space Complexity (Memory needed) $O(b^d)$ (huge). Space is more of an issue however as computers have limited storage. ✓
- **DFS with iterative deepening (DFS-ID)**: Essentially just DFS up to a certain user decided depth. It assumes $Cost(s, a) = const. = c, c \geq 0$ throughout the tree. Time complexity: $O(b^d)$ (same as BFS), Space Complexity (Memory needed) $O(d)$ (much better unless $d = D$ in which case its the same as DFS).

The following subsections are considered 'Graph search' instead of tree search.

2.2 Dynamic Programming

- The aim is to avoid the exponential run time of the algorithms above by only having to compute repeated routes once.
- It assumes that the state graph is acyclic.
- Here a state is a summary of all past actions sufficient to choose optimal future actions.
- It is essentially backtracking search with 'memoization' - i.e. it stores already visited paths.

2.2.1 Uniform Cost Search L10 ✓

- Expands its search with the node n that has lowest cumulative cost $g(n)$. This sometimes means stopping down one path and adding a new node to the list from an older branch because it's $g(n)$ is lower than the new explored nodes.
- (It is essentially the same as Djikstra's algorithm). Note nodes can not be added more than once to the visited list. But the order in which nodes are explored can have repetitions.
- Algorithm: Add unexplored nodes to the frontier (nodes to be considered) and then add from the frontier to the explored (visited) nodes with the lowest cumulative cost $g(n)$ and repeat. Essentially keep track of every expansion and expand on those nodes with the lowest $g(n)$. Write it in a table $Node_{wherefrom}|Cost(c_{ij})|G(n)$ or use a graph.
- UCS is optimal and complete if $c_{ij} \geq \epsilon \forall x, y$ for some constant ϵ (otherwise it could get stuck in an infinite loop). When all costs are the same (const.) then UCS is similar to BFS, except UCS explores all nodes, while BFS stops when a goal node is reached.

Now we come to informed (heuristic (estimated cost)) search algorithms. They use an evaluation function $f(n)$ which biases the search and/or a heuristic function $h(n)$ which is an estimate of the cost from a node n to a goal node. Note, previously (above), $f(n) = g(n)$. The following are also referred to as best-first algorithms. The aim is to make UCS faster.

2.3 Greedy best-first search L12

- Uses: $f(n) = h(n)$. So it just chooses the best path with the lowest heuristic $h(n)$ values. Note at each split it chooses the path with the lowest heuristic not the one with the lowest cumulative heuristic! Similar to DFS, except it only explores one path I think. So its super quick, but much can go wrong.
- Not complete and not necessarily optimal.

2.4 A^* L12

- Uses: $f(n) = g(n) + h(n)$ - its UCS + Greedy Search. It distorts costs to favour goal states.
- It's identical to UCS except now $g = g + h$. Note that $h(n)$ is not cumulative like $g(n)$, so once you add more nodes to your path, add up the total cost of getting there and then finally add the heuristic of the latest added node.
- Can also be formulated as $Cost'(s, a) = Cost(s, a) + h(Succ(s, a)) - h(s)$. Note, s is a summary of previous states, not the same as a node n. Note, this means calculating the total cost to a node and then adding the heuristic at that node - don't add costs and heuristics on the way!
- For A^* to be complete and optimal, $h(n)$ must be:
  - Admissible if $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost to reach the goal state from n. $h^*(n) = g(Succ(n))$. If $h(n)$ is not admissible then A^* is called A. They are 'optimistic' because the assume the cost is less than it actually is.
  - Consistent if all arcs in the network are admissible, i.e. that that $Cost(AtoB) \leq h(A) - h(B)$. In Raul's notation: $h(n) \leq Cost(n, a) + h(n'), \forall an' = Succ(n)$. Corollary: If a heuristic is consistent it is automatically admissible - not the other way round however. If $h(n)$ is consistent we can prove the following properties
    - The values of $f(n)$ along any path are non-decreasing: Suppose $n' = Succ(n)$ and thus $g(n') = g(n) + Cost(n, a)$ then $f(n') = g(n') + h(n') = g(n) + (Cost(n, a) + h(n')) \geq g(n) + h(n) = f(n)$.
    - Whenever A^* selects a node for expansion the optimal path to that node has been found: Since $f(n)$ is increasing or staying the same (non-decreasing) it will always choose new frontier nodes n' that have the lowest cost, so optimality is guaranteed.
  - Aside, A^* can be visualised as using concentric (oval shaped) contours - whereas UCS uses circular contours.
- Deciding on heuristics:
  - Let h_1 and h_2 be two **admissible heuristics**, then if $h_2(n) \geq h_1(n)$ for all n then the heuristic h_2 *dominates* the other heuristic $h_1(n)$.
  - **To guarantee that a function of admissible heuristics is admissible, it must be less than or equal to the max admissible heuristic**. Implies sums and products don't work, but weighted averages do.
  - Use *Relaxation*: A 'relaxed' problem will find a better (less costly) route because of less restrictions and thus the solution (actual cost) to a relaxed problem can be used as an admissible heuristic in the actual problem. A relaxation P' of a search problem P has costs that satisfy $Cost'(n, a) \geq Cost(n, a)$ and then use the relaxed heuristic $h(n) = h'^*(n)$ as the minimum cost from n to a goal state using $Cost'(s, a)$
  - Or use subproblems - relax problem into independent subproblems or learn from experience

3 Bayesian Networks

3.1 Bayesian Graphical Networks L13, L14

- Bayesian Inference: $p(h|d) = \frac{p(d|h)p(h)}{\sum_{h' \in H} p(d|h')p(h')}$

- A Bayesian network describes the dependencies between random variables and allows you to fully describe the joint distribution.
- Bayesian DAG - Directed Acyclic Graph - (no cycles in the network, nodes = r. var, edges = influence). The Joint disribution is given by: $p(x_1, ..., x_n) = \prod_i^n p(x_i|parents(x_i))$
- **Conditional Independence** $(a) \rightarrow (B) \rightarrow (c)$ (head to tail) and $(a) \leftarrow (B) \rightarrow (c)$ (tail to tail) are conditionally independent given B. $(a) \rightarrow (B) \leftarrow (c)$ (head to head) are marginally independent (i.e. without observing B, but conditionally *dependent* after observing B. Use probability theory to derive these. You can apply **d-separation** (https://www.youtube.com/watch?v=aA-gTNxy1rw) to find conditional indepencies in a network.
- **A node is conditionally independent of all other nodes in a network given its Markov Blanket, i.e. its parents, children, or children's parents**.
- **A node is conditionally independent of its non-descendants given its parents.**
- **Markov Blanket**: $M(X) = Parents(X) + Children(X) + Parents(Children(X))$
- **Inference: Brute force method** → just for loop over every combination of joint probability (marginalising JPD). **Variable Elimination**, like Brute force, but just take out conditional distributions that sum to 1. However this only works for JPDs where the sums can be separated, e.g. $\sum_A \sum_D P(W|C, A)P(D|B, W) = \sum_A P(W|C, A) \sum_D P(D|B, W) = \sum_A P(W|C, A)$. For more complex graphs we require methods like variational Bayes and Gibbs sampling.

3.2 Dynamic Bayesian Networks L14

- DBNs are directed graphical models of stochastic processes. They 'exploit' temporal redundancy: Markov Assumption → future is independent of the past, Stationarity: transition model is stable over time. Each state is described by a random variable.
- **Markov Chains** are sequences of random variables that are each only conditionally dependent on the previous.
  - Order: The number of nodes a node depends on (i.e. has preceding it).
  - 0-order has 'no memory' and is equivalent to a multinomial probability distribution. Order 1 has a memory of 1 and is defined by a table of probabilities given by $p(x_n = t_i|x_{n-1} = tj)$.
  - **Note that the transitions are conditional probabilities!**
- **Hidden Markov Chains** (simplest form of DBN):
  - Assume an initial distribution $p(s_1)$, a transition model $p(s_t|s_{t-1})$ $S \rightarrow S'$ and an observation model (emmision probabilities) $p(o_t|s_t)$ $S \rightarrow S'$. The original state sequence (s) is not usually known.
  - Used for speech recognition, and car tracking (and image segmentation).

3.3 Learning in Bayesian Networks L15

- Goes into how to work out the conditional probabilities from data. If $P(D) = 3/5$ and there are 2 $(D, x), (D, x)$ then $P(x|D) = 2/3$. This 'count-and-normalise' is just a basic form of maximum likelihood.

3.3.1 Expectation maximisation

- Useful when some data is not observed. The aim is to find the marginal likelihood, i.e. to find the parameters that best generate the data given the missing information.
- Idea: Start with random initial parameters. Find likelihood that subsection of data came from parameters, then given these parameters generate data (**E Step:** Calculate posterior given current parameters and in the **M Step:** Do a maximum-likelihood estimation). Do this with all data and compute new parameter values and repeat until convergence.

4 Markov Decision Processes L17

- **Transition model**: $P(s'|s,a)$ - the probability of reaching state s' given action a is done in state s.
- **Markov Assumption**: The probability of reaching s' from s depends only on s not on earlier states, which implies the transition model can be represented as a dynamic bayesian network.
- **Reward** $R(s)$ - positive or negative reward for being in state s - bounded by $\pm R_{max}$.
- **Utility or Value function** $U_h([s_1, s_2, ...])$: Captures the long term advantages of being in state s. Utility of s given π is: $U^\pi(s) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t)|\pi, s_0 = s]$.
- **Policy** $\pi(s)$ Plan/strategy that the robot does in each state. π^* is the optimal policy that gives the highest expected utility.
- Other formulations: $Action(s)$ - possible actions, $Goal(s)$ - true if at end of process.
- **Finite horizon** - There is a fixed time N after which nothing changes $U_h([s_1, s_2, ..., s_{n+k}]) = U_h([s_1, s_2, ...s_n])$ $\rightarrow$ leads to non-stationary optimal policies $\rightarrow$ N matters.
- **Infinite horizon** - no fixed deadline, N doesn't matter, so you get stationary optimal policies.
- Additive reward - $U_h([s_1, ...]) = R(s_1) + R(s_2) + ...$, while Discounted reward $U_h([s_1, ...]) = R(s_1) + \gamma R(s_1) + \gamma^2 R(s_2)...$, where $\gamma \in [0, 1]$ is the **discount factor**.
- Infinite horizon awards - solution 1) utility of infinite sequence is finite $U_h([s_0, s_1, ...]) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \frac{R_{max}}{1-\gamma}$, solution 2) use proper policies, i.e. robot will eventually terminate, sol.3) compare average reward.
- **Expected utility** $U^\pi(s) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t)]$- obtained by robot executing policy π starting in s. **Optimal policy** is thus $\pi_s^* = argmax_\pi U^\pi(s)$.
- Note, discounted rewards ($\gamma < 1$) and infinite horizons implies the policy is independent of the starting state, but not the action sequence! Think of the arrows in the grid.

4.1 Value Iteration L18

- Idea: Compute the utility of each state and then find optimal policy (set of actions) based on these utilities.
- Criterion for optimal policy $\pi^* = argmax_\pi E[\sum_{t=0}^{\infty} \gamma^t R(s_t)|\pi]$ which is basically saying 'the optimal policy π maximises the expected utilities given all policies'.
- **Bellmann Equation**:
$$U(s) = R(s) + \gamma \max_{a \in Actions(s)} \sum_{s'} P(s'|a,s)U(s').$$
For n states there are n Bellmann equations with n unknowns. **Value iteration** is an iterative approach to solving these equations:
$$U_{i+1}(s) = R(s) + \gamma \max_{a \in Actions(s)} \sum_{s'} P(s'|a,s)U_i(s').$$
- Optimal Policy (do this once you have all the utilities from the Bellman equation):
$$\pi^*(s) = arg \max_{a \in Actions(s)} \sum_{s'} P(s'|s,a)U_(s')$$
- Can use Value Iteration for large state spaces,(but not super large spaces). But according to 2015 exam it is also quicker than policy iteration on smaller state spaces.
- **Essentially value iteration solves a set of linear equations using an iterative technique, whereas policy iteration does so by inverting a matrix**. Hence Value iteration is better for larger spaces than policy iteration.

4.2 Policy Iteration L18

- Idea: Given a policy π_i calculate the value function (utilities) of each state given this policy and then find a new (improved) policy given the previous value function and then repeat until the policy no longer changes. It uses *Policy evaluation* (simpler than solving Bellman equations because actions are fixed by policy) and *Policy improvement*.
- Policy evaluation: $U_i(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s))U_i(s')$
- Use for small state spaces.
- For each iteration n simultaneous equations need to be solved, which takes around n^3 operations.

4.3 Reinforcement Learning L18

- Actions have non-deterministic effects (transition function unknown), Rewards/punishments are infrequent (reward function unknown).
- Naive Approach - act randomly to explore actions, learn transition and reward function and then apply policy or value iteration. If action leads to reward reinforce that action else avoid it.
- Model based learning - learn MDP, solve MDP to find optimal policy, treat difference between actual and expected reward as an error signal.
- Model-free learning - learn utility function directly $\rightarrow$ TD-learning, Q-learning.
- Reinforcement learning is an 'online' process.

5 Game Theory L19

- Game theory includes cooperation, chance, imperfect knowledge, simultaneous moves and aims to replicate real-world decision making.
- **Normal form**: Represent information in a **payoff matrix** which contains the payoffs (total gain) or utilities (future gains) for each player.

	P2 C	P2 D
P1 C	Reward for cooperation (P2,P1)	Sucker's payoff
P2 D	Temptation to defect	Punishment for mutual defection

- Zero-sum game has equal and opposite payoffs and utilities for both players $U_{p1} + U_{p2} = 0$, or more generally constant-sum games $U_{p1} + U_{p2} = const$.
- Markov game - means fully a observed game - nothing unknown.
- Imperfect information game - partially observed game.
- Deterministic and chance games and 2 player and multiplayer games can also be considered when differentiating between games.
- Prisoner's dilemma is the case where $T > R > P > S$ [mistake in notes i think]
- **Dominant strategy** - the strategy that is best for a player regardless of the other players strategy
- **Nash equilibrium** - a pair of strategies such that no player is worse off if they were to change, provided the other player sticks to the strategy. This occurs when both players have the same dominant strategy.
- Stag hunt - $R > S = P > T$, Coordination game - $R > P > T = S$, anti-coordination chicken game - $S > P > T > R$.

5.1 Game Theory in A.I. L19/20

- Notation: s_0 - initial state, $Actions(s)$ - available actions, $Result(s, a)$ - transitional model, result of an action, $Terminal - test(s)$ - true if s is a goal state, $Utility(s, p)$ - scoring function, assigns values to states, $GameTree$ - encodes all possible games.

———————— Minimax algorithm ————-
Minimax(s) =

- $Utility(s)$ if $Terminal - test(s)$ true
- Take max if it's $MAX\triangle$'s turn
- Take min if it's $MIN\nabla$'s turn

Time complexity $O(b^m)$, space complexity $O(bm)$, where b is the number of legal moves and m is the max depth of the tree. ————————————

- Alpha-Beta pruning - don't explore certain branches because they don't affect anything - do this by checking whether an unknown node if higher or lower than a known node has any affect on the root nodes above.
- **Evaluation function** is an *estimate* of the expected utility from a certain game position. It measures goodness of a game position ($f(n) > 0$ - position n is good for me but bad for you, $f(n) = +\infty$ - win for me, e.g. Alan Turing's function for chess $f(n) = \frac{w(n)}{b(n)}$), **Heuristic** is a non-negative estimate of the cost to a goal. **Zero-sum assumption** a single evaluation describes goodness of a state w.r.t both players.
- Evaluation functions need to be time efficient to make them worth while and need to have the same terminal state ordering as the true utility function, i.e. States that win must be evaluated as higher than draws etc... They need to be strongly correlated with the true evaluation function.
- **Stochastic games** - unpredictable external events. Represented by a circle in a tree and probabilities on the edges. Simply take expectation (i.e. sum the multiplications between nodes and edges to get root node) $\rightarrow ExpectiMax$ equation. So its just the same as the $MiniMax$ algorithm except it takes randomness into consideration - good for imperfect games. $O(b^m n^m)$