Linear Algebra:
- Vector Multiplication: $x^T y = \sum_{i=1}^{n} x_i y_i$
- Matrix Multiplication: $C = AB \rightarrow C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$
- Gradients: $\nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right]^T$
- Gradients of Common Functions: $\nabla(a^T x) = a$, $\nabla(x^T A x) = (A + A^T)x$

Machine Learning
- Supervised Learning: Given labelled data, attempt to learn function that assigns labels to unlabelled data
- Unsupervised Learning: Given unlabelled data, attempt to find structure
- K-nearest Neighbours: Classify based on majority label of k nearest points
- Confusion matrix:

	Actual: yes	Actual: no
Predicted: yes	TP	FN
Predicted: no	FP	TN

- Precision = $\frac{TP}{TP+FP}$ Recall = $\frac{TP}{TP+FN}$ Specificity = $\frac{TN}{TN+FP}$
- Cross Validation: Split data into k subsets, train on k-1, test on 1, repeat k times
- Overfitting: Model fits training data too closely, fails to generalise
- Underfitting: Model too simple to capture underlying structure
- Decision Trees: Create best-split feature node, for all split data repeat
- Binary Classification: $f(x) \rightarrow y \in \{1, -1\}$
- Regression: $f(x) \rightarrow y \in \mathbb{R}$
- Hypothesis class: Function our algo will produce
- Loss function: How accurate is our predictor?
- Optimization Algo: How do we minimize loss?
- Linear Regression:
  - Hyp: $f(x) = w_1 + w_2 x$ or $f_w(x) = w \cdot \phi(x) : w = [w_1, w_2], \phi(x) = [1, x]$
  - Loss: $Loss(x, y, w) = (f_w(x) - y)^2$
  - $TrainLoss(w) = \frac{1}{|D_{train}|} \sum_{(x,y) \in D_{train}} Loss(x, y, w)$
  - Optimization: $\nabla_w TrainLoss(w) = \frac{1}{|D_{train}|} \sum_{(x,y) \in D_{train}} 2(w \cdot \phi(x) - y)\phi(x)$
  - Gradient Update: $w \leftarrow w - \eta \nabla_w TrainLoss(w) : \eta = 0.1$
- Binary Classification: Create a linear function representing a boundary
  - Hyp: $f(x) = sign(w \cdot \phi(x)) : sign(z) = 1 : z > 0, -1 : z < 0, 0 : z = 0$
  - Zero-One Loss: $Loss_{0-1}(x, y, w) = 1[f_w(x) \neq y]$
  - Hinge Loss: $Loss_{hinge}(x, y, w) = max\{1 - (w \cdot \phi(x))y, 0\}$
  - Logistic Regression: $Loss_{logistic}(x, y, w) = \log(1 + e^{-(w \cdot \phi(x))y})$
- Stochastic Gradient Descent: Gradient descent is slow!
- Neural Networks
  - Hyp: $f(x) = V\sigma(W\phi(x))$ where σ is activation function
  - Activation functions: $\sigma(z) = \max\{0, z\}$ (ReLU), $\sigma(z) = \frac{1}{1+e^{-z}}$ (sigmoid)
  - Forward propagation: Compute layer outputs from input to output
  - Backpropagation: Compute gradients using chain rule, update weights
  - Architecture: Input layer $\rightarrow$ Hidden layers $\rightarrow$ Output layer
  - Layers represent multiple layers of abstraction
- How to prevent overfitting?
  - Reduce Dimensionality by removing features
  - Regularize: $\min_w TrainLoss(w) + \lambda/2 \cdot ||w||^2$

Search: No uncertainty

Definitions: b: branching factor, m: maximum depth
- Components of a search problem:
  - State space: All possible states of an environment
  - Successor function: function that maps actions to consequences
  - Start state and end goal state
- DFS:
  - Idea: Expand one node as much as possible before exploring another path
  - $O(b^m)$ time complexity
  - $O(bm)$ space complexity
  - Complete if infinite tree size prevented and not optimal
- BFS:
  - Idea: prioritize nodes closest to start
  - $O(b^d)$ time complexity where d is depth of solution
  - $O(b^d)$ space complexity
  - Complete and optimal if step costs are equal
- Dijkstra's (Uniform Cost Search):
  - Idea: prioritize expansion of low-cost paths
  - $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ time complexity where C^* is optimal cost, ϵ is minimum step cost
  - $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ space complexity
  - Complete if step costs $\geq \epsilon > 0$ and optimal
- Greedy Search
  - Idea: Expand a node that we think is closest to the goal state
  - Utilize a heuristic
- A* Informed Search Algorithm
  - Idea: use a heuristic to improve efficiency and make informed decisions
  - Heuristic: A function that estimates how close a state is to the goal
  - A heuristic is admissible if it never overestimates the cost of reaching a goal: $0 \leq h(n) \leq h*(n)$ where $h(n)$ is the cost to the nearest goal
  - A heuristic is consistent if $h(A) - h(C) \leq cost(A \rightarrow C)$. If it is consistent then it is optimal
  - $f(n) = g(n) + h(n)$

Markov Decision Processes
- MDPs are defined by:
  - set of states $s \in S$ and actions $a \in A$
  - transition functions $T(s, a, s')$ and reward functions $R(s, a, s')$
  - Start state and terminal state
- Utility: sum of rewards recieved
- Expected utility: $EU(a) = \sum_{a'} P(RES(a) = s')U(s')$
- Policies: the plan to get from start to goal: $\pi* : S \rightarrow A$
- With infinite utility we can mitigate with a finite horizon (limiting depth) or utilizing discounting: $U([r_0, \cdots, r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{max}/(1-\gamma)$
- Discounting: Sooner rewards have higher utility than later rewards
- The value of a state s is: $V*(s) = \max_a Q*(s, a) : Q*(s, a) =$ expected utility starting out having taken action a from state s and acting optimally
- $\pi^*(s) = argmax_a Q*(s, a)$
- Bellman Equations: Take correct first action $\rightarrow$ continue acting optimally
  - $V*(s) = \max_a Q*(s, a)$
  - $Q*(s, a) = \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V*(s')]$
- $V(s)$: Calculated value of a state
- $Q(s, a)$: Calculated value of an action from a state
- $\pi(s)$: What action we should take at a state
- Alternatively, we can perform policy iteration where we calculate values for some fixed policy until convergence, then update the policy using resulting converged values as future values

Reinforcement Learning MDP's without T or R!
- MDPs, but we have to learn T and R
- Agent tries action A, gets $\{s, r\}$
- Model-Based Reinforcement Learning: Learn an approximate model based on experiences
  - STEP 1: Learn empirical MDP Model
  - Count $s' : \forall s, a$
  - Normalize to estimate $\hat{T}(s, a, s')$
  - Doscover $\hat{R}(s, a, s')$ when we experience s, a, s'
  - STEP 2: Solve MDP, then run learned policy
- Model-Free Reinforcement Learning: Learn policies/values directly from experience
  - Passive RL: Execute a fixed policy $\pi(s)$ and learn state values
  - Direct Evaluation: Average sample values for each state
  - Temporal Difference Learning: Update $V^\pi(s)$ after each transition
  - TD Update: $V^\pi(s) \leftarrow (1-\alpha)V^\pi(s) + \alpha[R(s, \pi(s), s') + \gamma V^\pi(s')]$
  - Active RL: Learn optimal policy while acting
  - Q-Learning: Learn Q-values without knowing T or R
  - Q-Learning Update: $Q(s, a) \leftarrow (1-\alpha)Q(s, a) + \alpha[R(s, a, s') + \gamma \max_{a'} Q(s', a')]$
  - Exploration vs Exploitation: Balance trying new actions vs using known good actions
  - ϵ-greedy: With probability ϵ choose random action, else choose $argmax_a Q(s, a)$
  - We can approximate Q-learning with functions $f_1, f_2, \cdots$
  - $V(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$
  - $Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \cdots + w_n f_n(s, a)$

Games
- Types of games: Zero-sum: Competition (agents have opposite utilities), General-sum games: Agents have independent utilities
- We have states $s \in S$, players $p \in P$, actions $a \in A$, functions $SxA \rightarrow S$, terminal test $S \rightarrow \{true, false\}$, terminal utilities: $SxP \rightarrow R$
- Minimax: Time: $O(b^m)$, Space: $O(bm)$
- Alpha-Beta Pruning: Prune branches in a minimax tree to increase efficiency. Time: $O(b^{m/2})$.
- Multi-agent minimax: Terminals have utility tuples, node values are also utility tuples, each player maximizes its own component.
- Monte-Carlo Tree Search
  - Evaluation by rollouts: play multiple games to termination from a state s and count wins and losses
  - Selective search: explore parts of the tree that will help improve the decision at the root, regardless of depth
- Rollout: for each rollout, repeat until terminal: play a move according to a fixed, fast rollout policy. The fraction of wins correlates with the true value of the position
- MCTS V1: Allocate rollouts to promising and uncertain nodes.
- promicing and uncertain are defined by: $UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(PARENT(n))}{N(n)}}$
  - $N(n)$ number of rollouts from node n
  - $U(n)$ total utility of rollouts for Player(Parent(n))
- Theorem: As $N \rightarrow \infty$ UCT selects the minimax move

Bayesian Networks
- $P(a|b) = \frac{P(a,b)}{P(b)}$
- $P(Y|e) = \frac{P(Y,e)}{P(e)} = \alpha P(Y, e) = \alpha \sum_{h \in H} P(Y, e, h) \rightarrow \alpha = \sum_{y \in Y} P(Y, e)$
- Normalization: multiple each entry by $\alpha = 1/$(Sum of all entries)
- Chain Rule: $P(x_1, x_2, \cdots, x_n) = \Pi_i P(x_i | x_1, \cdots, x_{i-1})$
- Bayesian Statistics: Probabilities represent my uncertainty about the world
- Frequentist Statistics: Probabilities represent frequencies of real random outcomes. Properties of the world are fixed
- Probabalistic Inference: compute a desired probability from a probability model:
  - Enumerate options with sun
  - Sum out irrelevant variables
  - Normalize
- Time, Space, Data points: $O(d^n)$
- Bayes Rule: $p(H = h|Y = y) = \frac{P(H=h)P(Y=y|H=h)}{P(Y=y)}$, $P(a|b) = \frac{P(b|a)P(a)}{P(b)}$
- $P(Y = y) = \sum_{h' \in H} p(H = h')P(Y = y|H = h')$
- Two variables X and Y are independent if $\forall x, y : P(x, y) = P(x)P(y)$
- This implies that $P(x|y) = P(x), P(y|x) = P(y)$
- Conditional Independence: X is conditionally independent of Y given Z iff:
  - $\forall x, y, z : P(x|y, z) = P(x|z)$ or $\forall x, y, z : P(x, y|z) = P(x|z)P(y|z)$
- Bayes' Nets: Complex joint distributions using conditional distributions
  - Nodes: Variables(with domains). Can be assigned (observed) or unassigned (unobserved)
  - Arcs: Interactions that indicate direct influence between variables
  - For a Bayesian net, we have a set of nodes: X_i for each variable, a directed acyclic graph, and a conditional distribution for each node given its parent variables in the graph
  - A conditional probability table is a table where each row is a distribution for the child given values of its parents
  - $P(X_1, \cdots X_n) = \Pi_i P(X_i | Parents(X_i))$

Markov Models
- Markov Models have an initial distribution $P(x_0)$, transition model $P(X_t|X_{t-1})$
- Hidden Markov Models have an additional sensor model with observations $P(E_t|X_t)$
  - Filtering $P(X_t|e_{1:t})$
  - Prediction $P(X_{t+k}|e_{1:t})$
  - Smoothing: $P(X_k|e_{1:t}) : k < t$
  - Explanation: $P(X_{1:t}|e_{1:t})$
- Markov Assumption: $P(X_t|X_0, \ldots, X_{t-1}) = P(X_t|X_{t-1})$
- Stationary Assumption: Transition probabilities don't change over time
- Joint Distribution: $P(X_0, X_1, \ldots, X_T) = P(X_0) \prod_{t=1}^{T} P(X_t|X_{t-1})$
- Filtering: $P(X_t|e_{1:t}) = \alpha P(e_t|X_t) \sum_{x_{t-1}} P(X_t|x_{t-1})P(x_{t-1}|e_{1:t-1})$
- (α represents the normalization operator here)
- Prediction: $P(X_{t+1}|e_{1:t}) = \sum_{x_t} P(X_{t+1}|x_t)P(x_t|e_{1:t})$
- Most Likely Explanation: $\text{argmax}_{x_{1:t}} P(x_{1:t}|e_{1:t})$ using Viterbi algorithm
- Viterbi: $m_{t+1}[x_{t+1}] = P(e_{t+1}|x_{t+1}) \max_{x_t} P(x_{t+1}|x_t)m_t[x_t]$
- Time steps: Elapse time (predict forward), observe (weight particles), resample

Algorithms:

- **K-nearest Neighbours**

```
function KNN(x, data, k):
    distances = []
    for each (x_i, y_i) in data:
        d = distance(x, x_i)
        distances.append((d, y_i))
    distances.sort()
    neighbors = distances[0:k]
    return mode([y for (d, y) in neighbors])
```

- **Learning a Decision Tree**

```
function LearnDecisionTree(data, attributes):
    if all examples have same label:
        return Leaf(label)
    if attributes is empty:
        return Leaf(majority_label(data))
    best_attr = choose_best_attribute(data, attributes)
    tree = Node(best_attr)
    for each value v of best_attr:
        subset = {x in data : x[best_attr] = v}
        if subset is empty:
            tree.add_branch(v, Leaf(majority_label(data)))
        else:
            remaining = attributes - {best_attr}
            subtree = LearnDecisionTree(subset, remaining)
            tree.add_branch(v, subtree)
    return tree

function choose_best_attribute(data, attributes):
    return argmax over attributes of information_gain
```

- **K-Means**

```
function KMeans(data, k):
    centroids = randomly_select_k_points(data)
    repeat until convergence:
        clusters = [[] for i in range(k)]
        for each point x in data:
            closest = argmin_i distance(x, centroids[i])
            clusters[closest].append(x)
        for i in range(k):
            centroids[i] = mean(clusters[i])
    return centroids, clusters
```

- **Stochastic Gradient Descent**

```
function SGD(data, w_init, eta, num_epochs):
    w = w_init # [0,...,0]
    for epoch in range(num_epochs):
        shuffle(data)
        for each (x, y) in data:
            gradient = compute_gradient(Loss(x, y, w), w)
            w = w - eta * gradient
    return w
```

- **DFS**

```
function DFS(problem):
    frontier = Stack()
    frontier.push(start_state)
    explored = set()
    while frontier is not empty:
        node = frontier.pop()
        if node is goal:
            return solution
        explored.add(node)
        for each successor of node:
            if successor not in explored and not in frontier:
                frontier.push(successor)
    return failure
```

- **BFS**

```
function BFS(problem):
    frontier = Queue()
    frontier.enqueue(start_state)
    explored = set()
    while frontier is not empty:
        node = frontier.dequeue()
        if node is goal:
            return solution
        explored.add(node)
        for each successor of node:
            if successor not in explored and not in frontier:
                frontier.enqueue(successor)
    return failure
```

- **Djikstra's Uniform Cost Search Algorithm**

```
function Dijkstra(problem):
    frontier = PriorityQueue()
    frontier.push(start_state, 0)
    explored = set()
    cost = {start_state: 0}
    while frontier is not empty:
        node = frontier.pop()
        if node is goal:
            return solution
        explored.add(node)
        for each successor of node:
            new_cost = cost[node] + step_cost(node, successor)
            if successor not in explored:
                if successor not in cost or new_cost < cost[successor]:
                    cost[successor] = new_cost
                    frontier.push(successor, new_cost)
    return failure
```

- **A* Algorithm**

```
function AStar(problem):
    frontier = PriorityQueue()
    frontier.push(start_state, h(start_state))
    explored = set()
    g_cost = {start_state: 0}
    while frontier is not empty:
        node = frontier.pop()
        if node is goal:
            return solution
        explored.add(node)
        for each successor of node:
            new_cost = g_cost[node] + step_cost(node, successor)
            if successor not in explored:
                if successor not in g_cost or new_cost < g_cost[successor]:
                    g_cost[successor] = new_cost
                    f_cost = new_cost + h(successor)
                    frontier.push(successor, f_cost)
    return failure
```

- **Expectimax: For games with chance nodes, compute expected values instead of min/max**

```
function EM(state):
    if state is terminal:
        return utility(state)
    if state is max node:
        return max over actions of EM(successor(state, action))
    if state is chance node:
        return sum outcomes of P(outcome) * EM(successor(state, outcome))
```

- **Minimax**

```
function Minimax(state):
    if state is terminal:
        return utility(state)
    if state is max node:
        return max over actions of Minimax(successor(state, action))
    if state is min node:
        return min over actions of Minimax(successor(state, action))
```

- **Alpha Beta Pruning**

```
function ABP(state, alpha, beta):
    if state is terminal:
        return utility(state)
    if state is max node:
        v = -infinity
        for each action:
            v = max(v, ABP(successor(state, action), alpha, beta))
            if v >= beta:
                return v  # Beta cutoff
            alpha = max(alpha, v)
        return v
    if state is min node:
        v = +infinity
        for each action:
            v = min(v, ABP(successor(state, action), alpha, beta))
            if v <= alpha:
                return v  # Alpha cutoff
            beta = min(beta, v)
        return v
```

- **MCTS Version 2.0 UCT**

```
function MCTS(root_state):
    tree = {root_state}
    N = {root_state: 0}
    U = {root_state: 0}
    repeat until out of time:
        # Selection: traverse tree using UCB1
        node = root_state
        path = [node]
        while node is fully expanded and not terminal:
            node = argmax_child UCB1(child)
            path.append(node)
        # Expansion: add new child
        if node is not terminal:
            child = unexplored_child(node)
            tree.add(child)
            N[child] = 0
            U[child] = 0
            path.append(child)
        # Simulation: rollout from new node
        result = rollout(child)
        # Backpropagation: update counts
        for n in path:
            N[n] += 1
            U[n] += result
    # Return action with highest visit count
    return argmax_child N[child]
```

- **Value Iteration**

```
function ValueIteration(S, A, T, R, gamma):
    Initialize V(s) = 0 for all s
    repeat until convergence:
        V_new = {}
        for each state s in S:
            V_new[s] = max_a sum_s' T(s,a,s')[R(s,a,s') + gamma*V(s')]
        V = V_new
    return V, extract_policy(V)
```

- **Policy Iteration**

```
function PolicyIteration(S, A, T, R, gamma):
    Initialize random policy pi
    repeat until policy stable:
        # Policy Evaluation
        V = solve V(s) = sum_s' T(s,pi(s),s')[R(s,pi(s),s') + gamma*V(s')]
        # Policy Improvement
        pi_new(s) = argmax_a sum_s' T(s,a,s')[R(s,a,s') + gamma*V(s')]
        if pi_new == pi: break
        pi = pi_new
    return pi
```

- **Forward Algorithm for Hidden Markov Models**

```
function Forward(observations, T, E, pi):
    alpha_0 = pi * E[obs_0]
    for t = 1 to T:
        alpha_t = E[obs_t] * (T^T * alpha_{t-1})
        normalize(alpha_t)
    return alpha_T
```

- **Backward Algorithm**

```
function Backward(observations, T, E):
    beta_T = [1, 1, ..., 1]
    for t = T-1 down to 0:
        beta_t = T * (E[obs_{t+1}] * beta_{t+1})
        normalize(beta_t)
    return beta_0
```

Equations

- Exponential Moving Average: $x_n = \alpha x_{n-1} + (1-\alpha)x_n$
- Log-Based Normalization: $\frac{1}{T}\log(P(w_0,\ldots,w_T)) = \frac{1}{T}\sum_{t-1}^{T}\log(P(w_t|w_{t-1}))$
- HMM Forward Pass: $\alpha_t(j) = P(O_t|X_t = j)\sum_i \alpha_{t-1}(i)\cdot P(X_t = j|X_{t-1} = i)$
- Entropy: $H(X) = -\sum_i P(x_i)\log_2 P(x_i)$
- Information Gain: $IG(Y|X) = H(Y) - \sum_{x\in X} P(x)H(Y|X = x)$
- Learning rate decay: $\alpha_t = \frac{1}{1+t}$
- L2 Regularization: $\lambda||w||_2^2 = \lambda\sum_i w_i^2$
- Sum of geometric series: $\sum_{t=0}^{\infty}\gamma^t = \frac{1}{1-\gamma}$ for $|\gamma| < 1$
- Conditional probability expansion: $P(A,B|C) = P(A|B,C)P(B|C)$
- Law of Total Probability: $P(A) = \sum_i P(A|B_i)P(B_i)$
- Marginalization: $P(X) = \sum_y P(X, Y = y)$
- Product Rule: $P(X,Y) = P(X|Y)P(Y) = P(Y|X)P(X)$
- Total Probability: $P(Y) = \sum_x P(Y|X = x)P(X = x)$
- Independence: $P(X,Y) = P(X)P(Y) \iff P(X|Y) = P(X)$
- Conditional Independence: $P(X,Y|Z) = P(X|Z)P(Y|Z)$
- Complete Bellman Equation: $V^*(s) = \max_a\sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$
- Policy Extraction via Q-Values: $\pi^*(s) = argmax_a Q^*(s,a)$
- Q-Value Iteration: $Q_{k+1}(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma\max_{a'} Q_k(s',a')]$