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 node based on best-split feature, 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:
  - Hypothesis: $f(x) = w_1 + w_2 x$ 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
  - Hypothesis: $f(x) = sign(w \cdot \phi(x)) : sign(z) = 1 : z > 0, -1 : z < 0, 0 : z = 0$
  - 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
  - Hypothesis: $f(x) = V\sigma(W\phi(x))$ 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
  - 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
  - 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 $\geq \epsilon > 0$
  - 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
  - $V*(s) = \max_a Q*(s, a)$
  - $Q*(s, a) = \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V*(s')]$
- 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: $Q_{k+1}(s, a) = \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma \max_{a'} Q_k(s', a')]$
- 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!

-

Bayesian Networks

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**

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