PEAS & Problem Formulation

- **P**erformance Measure: Parameters used to measure the performance/efficacy of an AI model (e.g. safety of a self-driving car)
- **E**nvironment: Surrounding of an agent

Fully Observable Agent has access to complete state of the environment at each point in time (e.g. chess board)	**Partially Observable** Agent does not have complete info about the current state of the environment (e.g. self-driving cars)
Deterministic Next state can be determined fully based on current state and action (e.g. chess)	**Stochastic** Next state has some randomness that cannot be completely determined by the agent (e.g. self driving cars)
Episodic The choice of an agent's action is independent of other actions (e.g pick and place robot)	**Sequential** The choice of an agent's action is dependent on previous actions (e.g. chess and self-driving cars)
Static Environment is unchanged while an agent is still deciding on an action (e.g. chess with no clock) **Semi-Dynamic** Environment does not change, but agent's performance score does	**Dynamic** Environment constantly changes, even when agent is still deciding on an action (e.g. self-driving cars)
Discrete Actions and percepts are clearly defined and of finite in number (e.g. chess)	**Continuous** Actions and percepts cannot be clearly defined (e.g. self-driving cars)
Single Agent Single agent operating by itself in an environment (e.g pick and place robot)	**Multi-Agent** Multiple agents operating in an environment (e.g. self-driving cars on a busy road)

- **A**ctuator: The part of an agent that delivers the output of an action to the environment (e.g. steering wheel of a self-driving car)
- **S**ensor: Parts of an agent that receives the input (e.g. camera & speedometer of a self-driving car)

Problem Formulation

- **State/State Space**: State/Set of all states reachable from initial state (e.g. all cities reachable from starting city)
- **Initial State**: e.g. starting city
- **Goal State(s)/Test**: Desired outcome (e.g. destination city)
- **Actions**: Things an agent can do (e.g. move to neighbouring city)
  - Explicitly list all of them, if exam asks
- **Transition Model**: The effect of performing an action on a state (e.g. arrive at next city)
- **Action Cost Function**: Cost of performing an action (e.g. distance)

Uninformed Search Algorithms

Breadth-First Search (BFS)

- Time: $1 + b + b^2 + \cdots + b^d = \boldsymbol{O(b^d)}$
- Space: $\boldsymbol{O(b^d)}$ (expand last child)
- Complete: Yes, if b is finite. If d is infinite, then yes if ∃ solution
- Optimal: Yes, if step cost is uniform
- In this module, next successor is at the start of the queue

```
create frontier: queue
create visited
insert initial state to queue and visited
while frontier is not empty:
    state = frontier.pop()
    for action in actions(state):
        next state = transition(state, action)
        if next state in visited: continue
        if next state is goal: return solution
        frontier.add(next state)
        visited.add(next state)
return failure
```

Uniform-Cost Search

- Tier: Group of paths with the same cost.
- # Tiers ("depth") = C^*/ε, C^* = cost of optimal solution, ε = min. edge cost
- Time: $\boldsymbol{O(b^{C^*/\varepsilon})}$
- Space: $\boldsymbol{O(b^{C^*/\varepsilon})}$
- Complete: Yes, if # tiers is finite i.e. $\varepsilon > 0$ and C^* is finite
- Optimal: Yes, if $\varepsilon > 0$
- Goal test is performed after POP, because the priority

```
create frontier: priority queue (PATH-COST)
create visited
insert initial state to queue and visited
while frontier is not empty:
    state = frontier.pop()
    if state is goal: return solution
    visited.add(state)
    for action in actions(state):
        next state = transition(state, action)
        if next state in visited: continue
        if next state not in frontier:
            frontier.add(next state)
        else if next state has lower COST:
            update state in frontier
return failure
```

Depth-First Search (DFS)

- Time: $\boldsymbol{O(b^m)}$, m = max depth
- Space: $\boldsymbol{O(bm)}$ (everytime we traverse down one level, b nodes are added)
- Complete: No, when depth is infinite
- Optimal: No
- In this module, next successor is at the top of the stack

```
create frontier: stack
create visited
insert initial state to queue and visited
while frontier is not empty:
    state = frontier.pop()
    for action in actions(state):
        next state = transition(state, action)
        if next state in visited: continue
        if next state is goal: return solution
        frontier.add(next state)
        visited.add(next state)
return failure
```

Depth-Limited Search (DLS)

- Variant of DFS; backtracks whenever d = depth limit (l)
- Time: $b^0 + b^1 + b^2 + \cdots + b^l = \boldsymbol{O(b^l)}$
- Space: $\boldsymbol{O(b)}$
- Complete: No, solution may lie in greater depths
- Optimal: No

Iterative-Deepening Search (IDS)

- Perform DLS with depth limit l = 0 … N until solution is found
- Time: $b^0 + (b^0 + b^1) + (b^0 + b^1 + b^2) + \cdots + (b^0 + b^1 + \cdots + b^d) = (d+1)b^0 + db^1 + (d-1)b^2 + \cdots + b^d = \boldsymbol{O(b^d)}$, d = depth where solution is found
- Space: $\boldsymbol{O(bd)}$
- Complete: Yes, if ∃ solution
- Optimal: Yes, if step cost is uniform. IDS is like "BFS-ing" a DFS
- Generates overhead = $\frac{N_{IDS} - N_{DLS}}{N_{(DLS)}}$

Bidirectional Search

- Perform BFS from each goal state, and BFS from start until the searches both search trees contain some common node(s)
- Operators need to be reversible (i.e. need to know what is the parent of each state)
- Time: $\boldsymbol{O(b^{d/2})} + \boldsymbol{O(b'^{d/2})}$, b = branching factor for forward BFS, b' = branching factor for backward BFS
- Space: max{ $\boldsymbol{O(b^{d/2})}$, $\boldsymbol{O(b'^{d/2})}$ }
- Complete: Yes
- Optimal: Yes

Informed Search Algorithms

Greedy Best-First Search

- Evaluation function only considers the heuristic, i.e. $f(n) = h(n)$
- Time: $\boldsymbol{O(b^m)}$ with good heuristic
- Space: $\boldsymbol{O(bm)}$ with good heuristic
- Complete: No
- Optimal: No

```
create frontier: priority queue
insert initial state to queue and visited
while frontier is not empty:
    state = frontier.pop()
    if state is goal: return solution
    for action in actions(state):
        next state = transition(state, action)
        frontier.add(next state)
return failure
```

A* Search

- Evaluation function considers the cost so far, g(n), together with heuristic, i.e. $f(n) = g(n) + h(n)$
- Time: $\boldsymbol{O(b^m)}$ with good heuristic
- Space: $\boldsymbol{O(bm)}$ with good heuristic
- Complete: Yes. g(n) monotonically increases along a path → if loop back to visited node, it won't explore that node again
- Optimal: Yes, depending on heuristics

Heuristics

- Admissibility: $h(n) \leq h^*(n)$ for all n
  - If $h(n)$ is **admissible**, A* using **tree** search is optimal
  - How?: relax the problem by removing certain restrictions
- Consistency: $h(n) \leq cost(n, n') + h(n')$
  - Triangle-inequality: x + y > z, for any sides of a triangle
  - If $h(n)$ is **consistent**, A* using **graph** search is optimal
  - Consistency implies admissibility, if $h(G) = 0$
- Dominance: If $h_1(n) \geq h_2(n)$ for all n, then $h_1(n)$ dominates $h_2(n)$
  - Dominant → closer to the true cost → incur less search cost

Variants of A* Search

- **Iterative Deepening A***: Cutoff using $f(n)$ cost (instead of depth)
- **Simplified Memory Bound A***: Drop nodes in PQ with worst $f(n)$ if PQ is full

Local Search

- Start with initial solution, move from one solution to another in solution space by applying local changes until an optimal solution is found

Trivial Algorithms

- **Random Sampling**: Generate states randomly until solution is found
- **Random Walk**: Randomly pick a neighbour from current state

Hill Climbing Algorithm

- Can get stuck at local optimum
- **Simulated Annealing**
  - With some probability *P*, pick a random neighbour instead of the best one
  - $P = e^{\frac{value(next) - valu\ (current)}{T}}$
  - If *T* decreases slowly enough, simulated annealing will find global optimum with high probability

```
current = initial state
loop:
    next = successor with highest value
    if value(current) ≥ value(next):
        return current
    current = neighbour
```

```
curr = initial state
for t = 1 … ∞:
    T = schedule(t)
    if T = 0: return current
    nxt = random successor of current
    if value(nxt) > value(curr) or P(nxt, curr, T):
        curr = next
```

Beam Search

- Perform *k* hill-climbing in parallel. i.e. *k* successors selected at each level
- **Local Beam Search**: Choose *k* successors deterministically (i.e. best *k* values)
- **Stochastic Beam Search**: Choose *k* successors probabilistically

Informed Search	Local Search
Costly if there's no solution as it explores the entire state space	Better if there's no solution, and a good enough state is acceptable
Slower if there's many solutions	Faster if there's many solutions
Slow if search space is large	Faster is search space is large

Adversarial Search

Minimax

- Alternate min-max at each level Max nodes pick max successor; Min nodes pick min successor
- Time: $\boldsymbol{O(b^m)}$
- Space: $\boldsymbol{O(bm)}$
- Optimal: Yes, if players are playing optimally

```
minimax(S, α, β, maximizer):
    if terminal(S): return utility(S)
    if maximizer:
        v = -∞
        for each action in actions(S):
            if α ≥ β: break
            S' = transition(action, S)
            v = max(v, minimax(S', α, β, False)
            α = max(α, v)
    else:
        v = ∞
        for each action in actions(S):
            if α ≥ β: break
            S' = transition(action, S)
            v = min(v, minimax(S', α, β, True)
            β = min(β, v)
    return v
```

Minimax with alpha-beta pruning

- α: min score the max player can guarantee
- β: max player the min player can guarantee
- At max level: update α only.
- At min level: update β only.
- Initially, (α, β) = (-∞, ∞)
- Branches are pruned whenever **α ≥ β**
- Ordering at the leaves can affect pruning efficiency
  - With perfect ordering time is reduced to : $\boldsymbol{O\left(b^{\frac{m}{2}}\right)}$

Supervised Learning

- Learns from labelled data
- Formal definition: Find a hypothesis $h: x \to \hat{y}$ from a hypothesis class H s.t. $h \approx f$, where $f: x \to y$ is a true mapping function
- **Classification**: predict discrete output
- **Regression**: predict continuous output

Performance Measure

- Split data into training set and test set
  - Training set: Train learning algorithm to generate hypothesis
  - Test set: Apply hypothesis on test set to analyse performance

Regression

- Absolute Error = $|\hat{y} - y|$
- Squared Error = $(\hat{y} - y)^2$
- Mean Squared Error = $\frac{1}{N}\sum_{i=1}^{N} (\hat{y}_i - y_i)^2$

Classification

- Accuracy = $\frac{1}{N}\sum_{i=1}^{N} 1_{\hat{y}_i = y_i}$
- Confusion Matrix

Predicted \ Actual	T	F
T	True +ve	False +ve
F	False -ve	True -ve

$Precision = \frac{TP}{TP + FP}$ $Recall = \frac{TP}{TP + FN}$

$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$

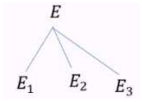
$F1 = \frac{2}{\frac{1}{P} + \frac{1}{R}}$

Decision Tree

- A tree-like structure with each internal node as a decision, each branch is an outcome of the decision and each leaf is the final prediction/decision
- Recursively pick an attribute to split remaining data until no more attributes to split OR all rows accounted for
- "Best" attributes are greedily chosen first

Chosing "best" attribute

- **Entropy** I
  - Indicates the "purity" of a decision – lower entropy indicates less noise and increased certainty
  - $\boldsymbol{I(P(v_1), \dots, P(v_n)) = -\sum_{i=1}^{n} P(v_i) log_2 P(v_i)}$
  - If binary decisions, then $I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{p}{p+n} log_2\left(\frac{p}{p+n}\right) - \frac{n}{p+n} log_2\left(\frac{n}{p+n}\right)$
- **Information Gain** IG
  - Difference in I between current and remainder
  - Remainder = sum of I of children nodes
  - This is used to decide the best attribute to use for splitting
- **Gain Ratio & Split Information**
  - Attributes with very specific values (e.g. HP number) can result in very high IG → overfitting
  - $GainRatio(A) = \frac{IG(A)}{SplitInformation(A)}$
  - $SplitInformation(A) = -\sum_{i=1}^{d} \frac{|E_i|}{|E|} log_2\left(\frac{|E_i|}{|E|}\right)$



- **Cost-Normalized-Gain**
  - Factor in the cost of attributes and choosing low-costs onces
  - **Cost-Normalized-Gain** = $\frac{IG^2(A)}{Cost(A)}$ or $\frac{2^{IG(A)} - 1}{(Cost(A)+1)^w}, 0 \le w \le 1$
    - w is adjusted on how important the cost is (larger w → more impt)
- **Continuous-Valued Attributes**
  - Can be discretised into a set of intervals
- **Attributes with missing values**
  - Assign most common value, drop the attribute, drop the rows etc.

Overfitting

- DTs performance is perfect on training data
- **Occam's Razor**
  - Prefeer short/simple hypotheses
  - Prefer shorter decision trees
- **Min-sample Pruning**
  - All leafs must have sample size greater than the min-sample
  - If sample size is smaller, then prune the parent decision node by making it a leaf with the majority decision as the value
- **Max-depth Pruning**
  - Depth = # decisions in one traversal
  - Prune all decision nodes exceeding the max-depth, picking majority

Linear Regression

- H is a set of line formulas
- **Regression**: Predicts a value with discrete inputs

Loss Function

- Notations:
  - Given a set of *m* (x, y) examples $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$
  - $x^{(i)} = (x_1, \dots, x_n)$, each x_i is a feature. Sometimes, we can prepend with the bias, x_0
  - h_w is a hypothesis $\in H$, $h_w(x^{(i)}) = w_0 x_0 + w_1 x_1^{(i)} + \cdots + w_n x_n^{(i)} = \hat{y}^{(i)}$
  - $w_0 x_0$ forms the y-intercept, so that the line does not always need to intersect at $y = 0$

Mean Squared Error (**MSE**) $\boldsymbol{J_{MSE} = \frac{1}{2m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})^2}$

Mean Absolute Error (**MAE**) $\boldsymbol{J_{MAE} = \frac{1}{2m}\sum_{i=1}^{m}|h_w(x^{(i)}) - y^{(i)}|}$

Gradient Descent

- Minimizes J_{MSE}
- Main Idea: Update each weight w_i in $w = (w_1, w_2, \dots, w_k)$ iteratively until $J_{MSE}(w)$ is minimum (local/global)
  - How do update? Check the slope of $J(w)$ against each $\boldsymbol{w_j}$ (using derivative), then update each $\boldsymbol{w_j}$

$$\boldsymbol{w_j \leftarrow w_j - \gamma \frac{\delta J(w_0, w_1, \dots)}{\delta w_j} = w_j - \gamma \frac{1}{m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})x_j^{(i)}}$$

	Batch GD	Mini-batch GD	Stochastic GD
# samples	All	Random subset	One random
Speed	Slow	Faster	Fastest
May escape local minima (for non-LR)?	Less likely	Yes	Yes

- MSE is **convex** for linear regression: it only has 1 minima (the global)

Features of different scales

- Features with large scales can dominate learning process → GD may not converge
- **Mean Normalization**: Normalize each feature i of input x to

$$x_i \leftarrow \frac{x_i - \mu_i}{\sigma_i}$$

- **Different learning rate** for each weight (smaller for features with larger scales)

Non-Linear Relationships

- Polynomial Regression: $h_w(x^{(i)}) = w_0 + w_1 f_1(x_1^{(i)}) + \cdots + w_n f_n(x_n^{(i)})$
- Max degree: $n - 1$; any larger can lead to overfitting
- Requires appropriate scaling otherwise it can lead to the same issues encountered by features of different scales

Normal Equation

$$X = \begin{bmatrix} 1 & x_1^{(1)} & \dots & x_n^{(1)} \\ 1 & x_1^{(2)} & \cdots & x_n^{(2)} \\ 1 & \dots & \dots & \dots \\ 1 & x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} \quad w = \begin{bmatrix} w_0 \\ w_1 \\ \dots \\ w_n \end{bmatrix} \quad \gamma = \begin{bmatrix} \gamma^{(1)} \\ \gamma^{(2)} \\ \dots \\ \gamma^{(m)} \end{bmatrix}$$

$$h_w(X) = Xw = w^T X^T$$

$\boldsymbol{w = (X^T X)^{-1} X^T Y}$, assuming $X^T X$ is invertible

- We can compute the weights using Normal Equation without going through gradient descent if $X^T X$ is invertible
- Advantage(s)
  - Directly finds w that minimises J_{MSE}
  - No feature scaling and playing around with γ is required
- Disadvantage(s)
  - Very slow with large number of features ($O(n^3)$)
  - Requires $X^T X$ to be invertible

Logistic Regression

- **Binary Classification** with continuous, based on some probability

Logistic Function

- $h_w(x)$ outputs the probability of x being part of the positive class
- $h_w(x) = \sigma(w_0 + w_1 x_1 + \cdots + w_n x_n)$, where σ is the logistic function:

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

- A decision threshold (of usually 0.5) is used as a cut-off for classification
- The decision boundary separates the set of points in the negative and positive class on the plane. It intersects the decision threshold
  - Decision boundary is perpendicular to the vector w

Loss Function

- Not ideal to use J_{MSE} – it will be non-convex
- **Cross-entropy** (CE) for C classes, $\hat{y}$ = predicted, y = actual

Cross-entropy (**CE**)

$$\boldsymbol{CE}(y, \hat{y}) = \sum_{i=1}^{C} -y_i log(\hat{y}_i)$$

Binary Cross-entropy (**BCE**)

$$\boldsymbol{BCE}(y, \hat{y}) = -y log(\hat{y}) - (1 - y) log(1 - \hat{y})$$

- BCE is CE but with only 2 classes (1 or 0)
- Use BCE J_{BCE} Loss as the loss function: sum BCE acoss all samples

$$\boldsymbol{J_{BCE}} = \frac{1}{m} \sum_{i=1}^{m} BCE(y^{(i)}, h_w(x^{(i)}))$$

- J_{BCE} is **convex** for logistic regression

Gradient Descent

- Minimizes J_{BCE}
- Weight update: (exactly the same as that in Linear Regression)

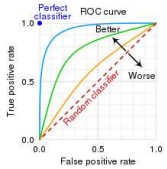
$$\boldsymbol{w_j} \leftarrow w_j - \gamma \frac{\delta J(w_0, w_1, \dots)}{\delta w_j} = \boldsymbol{w_j} - \gamma \frac{1}{m} \sum_{i=1}^{m} \left(h_w(x^{(i)}) - y^{(i)}\right) x_j^{(i)}$$

Non-Linear Decision Boundaries

- Transform features: $h_w(x^{(i)}) = \sigma(w_0 + w_1 f_1(x_1^{(i)}) + \cdots + w_n f_n(x_n^{(i)}))$

Performance Measure

- **Receiver Operator Characteristic (ROC) Curve**
  - TPR against FPR graph
    - TPR = TP / (TP + FN)
    - FPR = FP / (FP + TN)
  - TPR against FPR graph
- **Area Under Curve (AUC) of ROC Curve**
  - 0 ≤ AUC ≤1
  - AUC = 0.5 → random chance; The closer to 1, the better

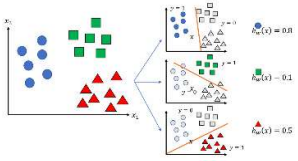


Multi-class Classification

- Run logistic regression multiple times, each on different classes

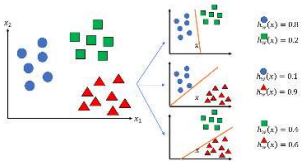
One-vs-All

- For each class C_i, run logistic regression on C_i and $C - C_i$
- Pick class with highest probability



One-vs-One

- For each pair of classes (C_i, C_j), run logistic regression on C_i and C_j
- Pick class with most wins



Model Evaluation

- Given dataset D and an error function (e.g. MSE, CE), $J_D(h)$ is the expected error of a model/hypothesis h

$$\boldsymbol{J_D(h)} = \frac{1}{N} \sum_{i=1}^{N} error\left(h(x^{(i)}), y^{(i)}\right)$$

- The lower the J_D, the better

Train-Validation-Test sets

- **Training** set D_{train}: Used to train the models $h_1, h_2, \dots h_k$
- **Validation** set D_{val}: Used to run on each of the models h_1 to h_k after training, then pick the best model h_i, that is, the model with least $J_{D_{val}}$
- **Test** set D_{test}: Used to run on chosen h_i to report h_i's performance =

Underfitting & Overfitting

- If $J_{D_{val}} \approx J_{D_{train}}$: High bias → Underfit and high training error
  - Increasing the size of the training set will not lower bias
  - Caused by inability of model to capture certain relationships. Increase model complexity to mitigate
- If $J_{D_{val}} \gg J_{D_{train}}$: High variance → Overfit and low training error
  - Increase size of training set to mitigate

Regularization

- Penalizes large weight values, favouring simpler hypothesis to avoid overfitting

L1 Regularization (Lasso)

Linear Regression

$$\boldsymbol{J_{MSE}} = \frac{1}{2m} \left[\sum_{i=1}^{m} \left(h_w(x^{(i)}) - y^{(i)}\right)^2 + \lambda \sum_{i=1}^{n} |w_i| \right]$$

Logistic Regression

$$\boldsymbol{J_{BCE}} = \frac{1}{m} \left[\sum_{i=1}^{m} BCE(y^{(i)}, h_w(x^{(i)})) + \lambda \sum_{i=1}^{n} |w_i| \right]$$

L2 Regularization (Ridge)

Linear Regression

$$\boldsymbol{J_{MSE}} = \frac{1}{2m} \left[\sum_{i=1}^{m} \left(h_w(x^{(i)}) - y^{(i)}\right)^2 + \lambda \sum_{i=1}^{n} w_i^2 \right]$$

Logistic Regression

$$\boldsymbol{J_{BCE}} = \frac{1}{m} \left[\sum_{i=1}^{m} BCE\left(y^{(i)}, h_w(x^{(i)})\right) + \frac{\lambda}{2} \sum_{i=1}^{n} w_i^2 \right]$$

- λ determines how much to penalize
  - Too large: Weights become too small → Underfit
  - Too small: Penalty is too little → Overfit
- Perform feature scaling before regularization

Linear Regression with Regularization: Normal Equation

$$w = \left(X^T X + \lambda \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 1 \end{bmatrix} \right)^{-1} X^T Y$$

$X^T X$ need not be invertible if $\lambda > 0$

Support Vector Machines

- Aim: Find a decision boundary that maximises the distance between it and the closest data points of both classes (i.e. the margin)

- Objective: $min\frac{1}{2}\|w\|^2$ s.t. all training data points are classified correctly
- Alternative objective:

$$max_\alpha \sum_i \alpha^{(i)} - \frac{1}{2}\sum_i \sum_j \alpha^{(i)}\alpha^{(j)}\bar{y}^{(i)}x^{(i)} \cdot x^{(j)}$$

- **Soft-Margin**
  - Allow some number of misclassifications for non-linearly separable data (whil assigning some penalty for misclassification)
- **Kernel Tricks**
  - Maps data from input space to a higher-dimensional feature space, without explicitly computing the transformation to that space, allowing data to be more likely linearly-separable
  - Explicitly computing the transformed features with ϕ is computationally expensive:

$$max_\alpha \sum_i \alpha^{(i)} - \frac{1}{2}\sum_i \sum_j \alpha^{(i)}\alpha^{(j)}\bar{y}^{(i)}\phi(x^{(i)}) \cdot \phi(x^{(j)})$$

  - Kernel tricks bypass this explicit computation by calculating the inner product between transformed vectors w/o actually transforming them:

$$max_\alpha \sum_i \alpha^{(i)} - \frac{1}{2}\sum_i \sum_j \alpha^{(i)}\alpha^{(j)}\bar{y}^{(i)}K(x^{(i)}, x^{(j)})$$

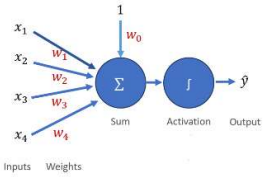
  - Gaussian (RBF) Kernel:

$$K(x^{(i)}, x^{(j)}) = \phi(x^{(i)}) \cdot \phi(x^{(j)}) = e^{\frac{\|x^{(i)}-x^{(j)}\|^2}{2\sigma^2}}$$

Neural Networks

Perceptron

- Perceptron = single-layer neural network; the simplest form of a NN
- Equivalent: $\hat{y} = h_w(x) = g(\sum_{i=0}^n w_i x_i)$, where g is an activation function
- **Activation function**: Transforms the weighted sum of inputs (+bias) into an output



Perceptron Learning Algorithm

1. Initialise all weights to 0
2. Loop (until convergence or max iterations):
   2.1. For each data point $(x^{(i)}, y^{(i)})$, classify $\hat{y}^{(i)} = h_w(x^{(i)}) = g(\sum_{i=0}^n w_i x_i)$,
   2.2. Choose one misclassified instance $(x^{(j)}, y^{(j)})$
   2.3. Update weights:

$$w \leftarrow w + \gamma(y^{(j)} - \hat{y}^{(j)})x^{(j)}$$

- Uses **step/sign function** as the activation function:

$$g(z) = \begin{cases} +1, z \geq 0 \\ -1, z < 0 \end{cases}$$

- **Does not converge** if data is not linearly separable

Activation Functions

- Without an activation function, the entire NN behaves like a single-layer model since the activation would just be a linear combination of inputs
- Linear Function: NN models linear regression
- Sigmoid Function: NN models logistic regression
- Softmax Function: NN models logistic regression; outputs from output layer form a probability distribution (i.e. sum of outputs = 1)

Multi-layer Neural Networks

- Capable of modelling non-linearly separable data
- Universal Function Approximation Theorem: NNs can represent a wide variety of interesting functions with appropriate weights
  - A single hidden layer can approximate any continuous function within a specific range

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad W^{[i]} = \begin{bmatrix} w_{11} & \cdots & w_{1m} \\ \vdots & \ddots & \vdots \\ w_{n1} & \cdots & w_{nm} \end{bmatrix}$$

$$\hat{y} = g^{[L]}\left(W^{[L]^T}\left(g^{[L-1]}\left(W^{[L-1]^T}\left(\dots g^{[1]}(W^{[1]^T}(x))\right)\right)\right)\right) = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_c \end{bmatrix}$$