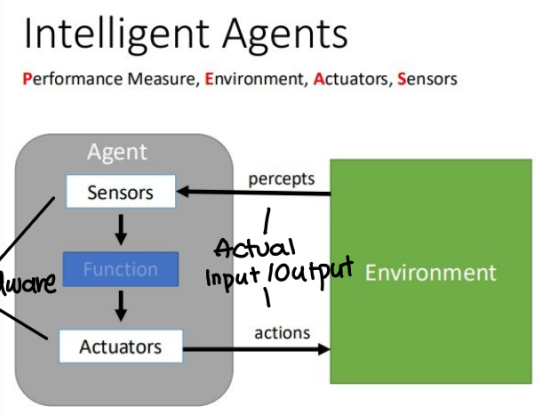
"Classical" AI : General



How do we design an AI, an "Intelligent Agent" ?

Goal Formulation → Problem Formulation ↑ Dependent on the problem type

Algorithm → Execute

Framework :

- Performance Measure
- Environment
- Actuators
- Sensors

Example : AI Fitness Health App that guides users through daily workouts and adapts routines based on their performance.

Performance Measure :

- Improvement in user's health metrics
- User consistency in the training program
- Minimise user injury and maximise fitness

Environment :

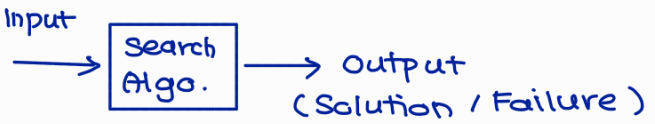
- User health status
- User lifestyle constraints
- Device used

Environment Types

- Fully Observable v Partially Observable (whether complete state information is known to agents)
  - Chess / Go
  - Poker / Self-Driving Car
- Single v Multi-Agent
- Deterministic v Stochastic (whether the next state is determined solely by the current state and actions executed by the agent i.e. whether there is a level of randomness)
  - + Strategic : If another intelligent agent / human is involved
- Episodic v Sequential (whether each state is independent or if past decisions can affect the current state)
- Static v Dynamic (whether the environment changes while the agent is deliberating)
  - + Semi-dynamic : If the environment doesn't change but the agent's score does
- Discrete v Continuous (Limited / 'Unlimited' no. of clearly defined, distinct actions)
  - Discrete ↑ Chess
  - Continuous ↑ Self-Driving Car (i.e. degree to turn the steering wheel right / left, etc.)

"Classical AI" : Search Problems

Search Problem : Type of problem where the goal is to find a state (or a path to a state) from a set of possible states by exploring various possibilities



$\forall$ state S_i, $|A(S_i)| \leq$ branching factor b
$\Rightarrow$ No. of valid actions at each state S_i is upperbounded by no. of possible actions

Representation Invariant : All Abstract states have a corresponding Concrete State

---

Algorithm Properties

- Time Complexity : No. of nodes generated / expanded
- Space Complexity : Max. no. of nodes in memory at a time
- Completeness : As long as one exists, the algo should return a solution
- Optimality : The algo should always return the optimal solution

Search Algorithms (+ Adversarial Search!)

1. Systematic Search :
   - Traverses the state space systematically, exploring all possibilities
   - Typically Complete & Optimal (under certain conditions)
   - Intractable (not solvable in a reasonable amount of time)
     ↳ many search problems are NP-Hard (i.e. TSP)

> Problem Formulation :
> - State Representation
> - Initial & Goal state / Test
> - (Valid) Actions ← actions(state)
> - Transition Model ← transition(s, a)
> - Action-Cost Function

2. Local Search
   - Traverses the state space by considering only locally-reachable states
   - Solution is the best state we've seen after n steps
   - Typically incomplete & suboptimal
   - Anytime-Property : Longer runtime $\propto$ More optimal solution
     By configuring n, we can get a 'Good-Enough' solution

> Problem Formulation :
> - State Representation
> - Initial State ← NO GOAL STATE! (optional Goal Test)
> - Successor Function : Generate Neighboring States
> - Evaluation Function : Evaluate the desirability / quality of a state

"Classical AI" : Local & Systematic Search

Uninformed Search Algo : We have no info about how good a state is

Informed Search Algo : We have some idea of how good a state is

1. Relaxed Problem
2. Solution for Relaxed Problem = Heuristic for original Problem

Heuristic : Estimate of the optimal path cost from a state s_i to the goal state, calculated by solving a relaxed problem when we generate our next state

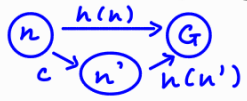
(The state space of the relaxed problem is a superset of the original)

$f(n) = g(n) + h(n)$ = Path cost to reach state n + Heuristic of state n

- Admissibility : $\forall$ nodes n, $h(n) \leq h^*(n)$ ← Optimal Path Cost

  Checks whether $h(n)$ overestimates the true cost
  i.e. Est. cost > True cost

- Consistency : $\forall$ nodes n, $\forall$ successors n' of n generated by any action a,
  $h(n) \leq c(n, a, n') + h(n')$ ← $h(\text{Goal}) = 0$



* If $h(n)$ is consistent → $h(n)$ is admissible

- Dominance : $\forall n, h_1(n) \geq h_2(n)$ → $h_1(n)$ dominates $h_2(n)$ (reflects original problem better)

  ⇒ As long as h_1 is admissible, h_1 is a better heuristic function

Systematic Search Algo :

Name	Time Complexity[1]	Space Complexity[1]	Complete?	Optimal?
Breadth-first Search	Exponential	Exponential	Yes	Yes
Uniform-cost Search	Exponential	Exponential	Yes[3]	Yes[3]
Depth-first Search	Exponential	Polynomial	No | Yes[4]	No
Depth-limited Search	Exponential	Polynomial[2]	No	No[2]
Iterative Deepening Search	Exponential	Polynomial[2]	Yes	Yes ← If constant step cost
A* Search	Exponential	Exponential	Yes[3]	Yes if admissible
A* Search with visited memory	Exponential	Exponential	Yes[3]	Yes if consistent

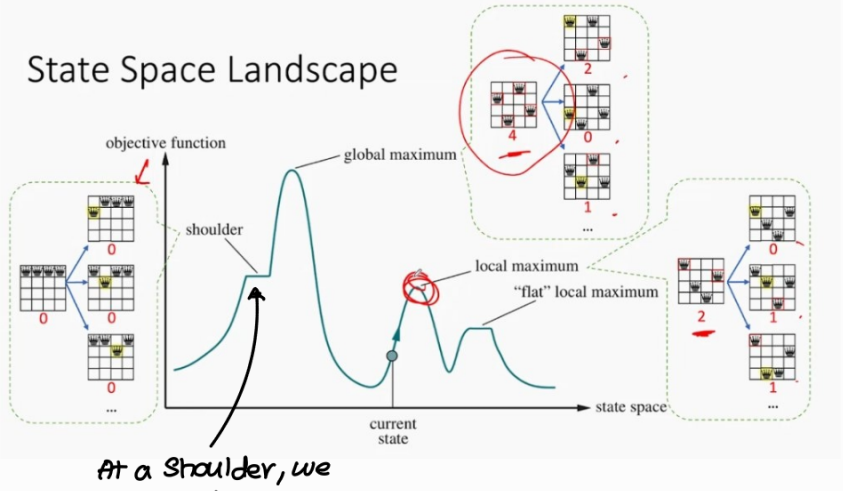
2: If used with DFS
3: If edge costs are positive
4: With visited memory

BFS : w.r.t. depth of optimal solution
UCS : w.r.t. 'tier' of optimal solution
DFS : w.r.t. max depth of the search tree
DLS : w.r.t. depth limit
IDS : w.r.t. depth of optimal solution

Systematic Search :

- Requires BOTH Initial & Goal states / Test
- If the search tree is infinite, we do not necessarily always terminate especially if there is not always a solution.

Local Search : Depending on the initial state, the solution that arises is different



At a shoulder, we return this state

* If no solution is possible / the search space is very large,
  Local Search > Systematic Search

"Classical AI" : Adversarial Search

Two players compete against each other, with competing goals

'Classical Adversarial Search' : Competitive, Two-Player Zero-Sum, Turn-Taking

Typically, we use a Game Tree to represent all possible states & moves in a game

- Nodes : States
- Edges : (Legal) Moves from 1 state to another

* The outcome (from the perspective of player A) is given at each leaf node.

Problem Formulation :

- State Representation
- Initial / Terminal States
- Actions
- Transitions
- Utility Function →

Utility Function

state → □ → Value (from perspective of A)

- Generally, A wants to maximise while B minimises
- Typically, terminal states : WIN / -WIN

⇒ non-terminal state values have to be calculated

Theorem : If B plays sub-optimally, ∀ actions taken by A, utility(state) of A ≥ utility(state) of A if playing against an optimal opponent

Minimax Algorithm

- For Classical Adversarial Search
- Assumes that all players play optimally ← If B plays sub-optimally, there may be other better strategies depending on B's strat.
- Computes outcome in a Depth-First manner

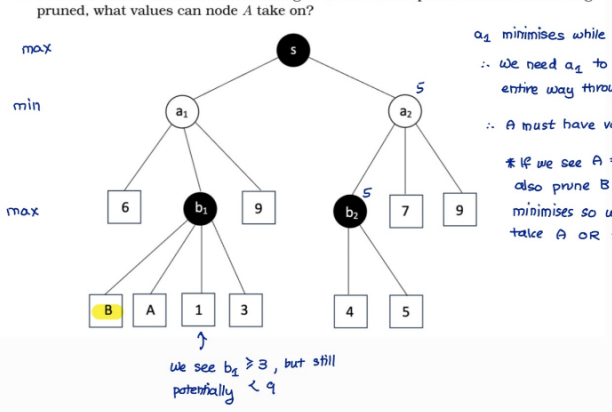
* Explores the entire Game Tree even if it may not be needed

↳ Alpha-Beta Pruning

- Maintains the best outcome so far for A, α, and B, β (highest & lowest respectively)
- Prunes unnecessary nodes

Example : Right to left evaluation

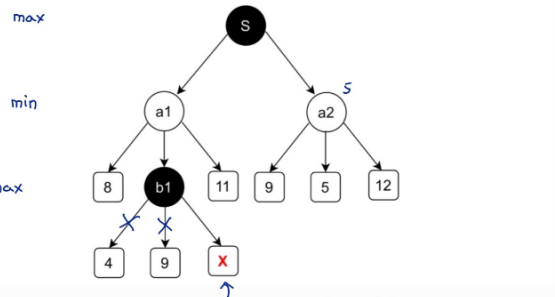
1. Consider the minimax search tree in Figure 8. In order to prevent node B from being pruned, what values can node A take on?



a_1 minimises while b_1 maximises.
∴ we need a_1 to see $b_1 < 9$ the entire way through
∴ A must have values < 9 ✱
✱ If we see A = 9, we can also prune B since a_1 minimises so we will either take A OR the leaf node 9

we see $b_1 \geq 3$, but still potentially < 9

2. Consider the minimax search tree in Figure 9. In order to prune the two leftmost children of node $b1$, what values can node X take on?



a_1 minimises while b_1 maximises
So we just need to see $b_1 \geq 11$ to prune the rest of b_1's nodes
⇒ We need to see $X \geq 11$ ✱

Cutoff Strategy

- Instead of evaluating the full Game Tree, cut off at some depth d (Think DLS) and estimate the value of this state by :

$$\text{eval(state)} = \begin{cases} \text{utility(state)} & \text{, if terminal} \\ \text{heuristic(state)} & \text{, otherwise} \end{cases}$$

* We do not consider Admissibility and Consistency for Adversarial Search

Theorem : (Finite, 2-Player, 0-Sum) Minimax w/ Cutoff returns a move optimal w.r.t eval function @ cutoff, BUT suboptimal w.r.t True Minimax Val

Machine Learning : General

Some problems are difficult to formulate
Some problems are inherently intractable
} Rather than an 'Intelligent Agent', create a 'Learning Agent' that learns a function that identifies patterns in the data & makes prediction

- Supervised v Unsupervised Learning : Whether we have the truth label
  - Supervised → Learn mapping from input to output
  - Unsupervised → Finds pattern / structure in unlabelled data
  - + Semi-Supervised Learning (some labelled, some unlabelled)
- Reinforcement Learning

Supervised Learning → 2 Types :

Components :
- Data
- Model
- Loss Function
- Learning Algorithm

1. Classification (Binary / Multi-Class / Multi-Label)
   - Predict a discrete label / class based on a set of pre-defined ones
2. Regression
   - Predict a continuous, numerical (real) value

↓

Takes in a training set D_{train} and finds a model h that approximates the true relationship between inputs and outputs

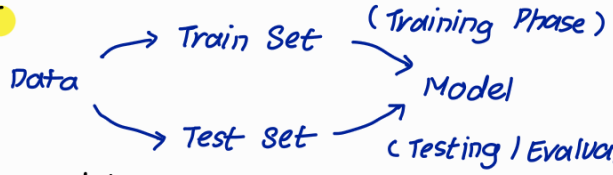
Usually quite general and can work for different data, model and loss

Dataset

Quality of data
- Relevance of features
- Noise (irrelevant / incorrect data)
- Balance (for Classification Tasks)
  - Could have high accuracy even on naive model / Give

}
- More data ⇒ Better Model Performance
- Garbage In → Garbage Out

Train-Test Split

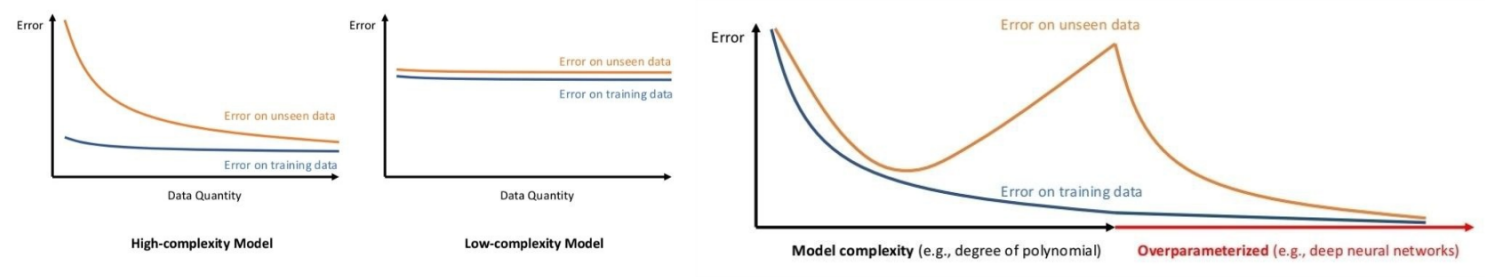


When training a model, we want a more general one that works well even on unseen data (i.e., in the real world when deployed)

1. Underfitting : When the model does not work well even on the training data ← High bias ; Low Variance
2. Overfitting : When the model works well on training data, but scores very poorly for test data ← Low bias ; High Variance

Bias : $\text{Bias}(\hat{Y}) = E(\hat{Y}) - Y$
- low → Fewer assumptions to build the target function

Variance :
- low → less sensitive to changes in training data



Hyperparameter Tuning : Process of optimising hyperparameter to improve its performance

Hyperparameter : Predefined and adjusted manually
Parameter : Learned during training (i.e. weights in a linear model)

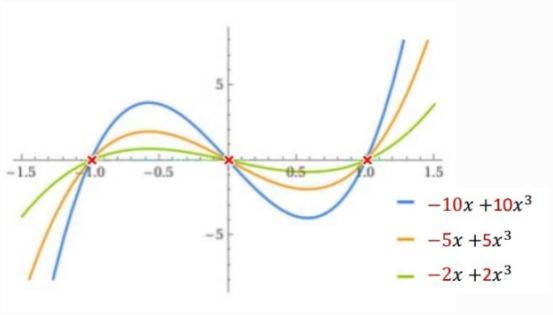
Seperate evaluation data for tuning, 'Validation Set'

Techniques : Grid Search, Random Search, Local Search, etc.

Regularization

We actually observe that overfitting is often associated with large-magnitude weights

Even if the model fits all points perfectly, there is still variation in the line in gives ⇒



∴ To discourage large-magnitude weights, we use <u>Regularization</u> :

Introducing constraints to the loss function to prevent the model from being too complex

Specifically, we augment the loss function $J(w)$ with a penalty function or Regularizer $R(w)$

$$J_{reg}(w) = J(w) + \lambda R(w)$$

- $J_{reg}(w)$: Regularized loss function
- $J(w)$: normal loss function
- $R(w)$: Regularizer
- λ : Regularization parameter

Typically, the Regularizer used is that of the L_p-norm function which measures the size of the vector

For $p \geq 1$, and a vector w of dimension $d+1$,

$$w = [w_0, w_1, \dots, w_d]^T,$$

the L_p-norm is defined as :

$$\|w\|_p = \left(\sum_{j=0}^{d} |w_j|^p \right)^{1/p}$$

* L_2-norm is actually Euclidean norm

$$\sqrt[2]{\sum_{j=0}^{d} w_j^2}$$

Common Use Cases :

- Lasso Regression → Linear Regression + L_1-norm
- Ridge Regression → Linear Regression + L_2-norm
- Elastic Net → Linear Regression + L_1-norm + L_2-norm

L_1 – Regularization

L_1-penalty : If a feature has non-zero weight, the L_1 Regularizer will penalise the feature $\bar{w}$ a constant penalty of λ (in terms of gradient)

L_1 induces Sparsity : For a feature to be included in the model, its contribution to reducing MSE loss must be exactly equal to the Regularization penalty λ (in terms of gradient)

If the feature's contribution is less than the penalty λ, the optimisation will prefer set its weight to exactly zero rather than pay the L_1 penalty, inducing sparsity

L_2 - Regularization

Invertibility : Let X be a $n \times d$ matrix and $\mathbb{I}$ be the $d \times d$ Identity Matrix. For any real number $\lambda > 0$, the matrix $X^TX + \lambda\mathbb{I}$ is invertible

Solution Uniqueness : For any $\lambda > 0$, Ridge Regression has a unique closed-form solution given by $w = (X^TX + \lambda\mathbb{I})^{-1} X^TY$

L_2 induces Weight Shrinkage : For any $\lambda = 0$, the L_2-norm of the L_2 regularized weight vector is strictly smaller than the non-regularized weight vector, assuming the solution is non-zero.

Machine Learning : Performance Measurement & Loss Function

Performance Measure & Loss Function :

Functions that take in a dataset $D = \{(x_1, y_1), \dots (x_n, y_n)\}$ and a model / hypothesis h and return a number representing how well h predicts the mapping $x_i \to y_i$

Performance Measure : Metric that evaluates how well the model performs a task
Used during evaluation / testing

Loss Function : Quantifies the difference between the model's prediction $\hat{y}_i$ and the true label y_i
Used during training to guide learning

Classification Task

Performance Measures

- Accuracy : $\frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{\hat{y}_i = y_i} = \frac{TP+TN}{Total}$

Confusion Matrix :

	True	
Predict	TP	FP
	FN	TN

- Precision : $TP/(TP+FP)$ ← If FP is worse (E.g. Spam Prediction)
- Recall : $TP/(TP+FN)$ ← If FN is worse (E.g. Cancer Prediction)
- F1 Score : $2/(\frac{1}{P} + \frac{1}{R})$ ← To balance Precision & Recall

Loss Function

Cross - Entropy :
① "True" Probability Distribution, P
② Probability Distribution, Q, that represents the model prediction

Measures the difference between two probability distributions over the same set of outcomes

$$H(P,Q) = -\sum_x P(x) \log(Q(x)) \xrightarrow{\text{For Binary Cross Entropy}} H(P,Q) = -\sum_x P(x)\log(Q(x))$$
$$= -P(1)\log(Q(1)) - P(0)\log(Q(0))$$

Let the "true" distribution P be described by a single value y (the true label : 1 / 0)
and the predicted distribution Q be described by a single value p (the model's predicted probability that the outcome is Class 1)
then, $BCE(y,p) = -y\log(p) - (1-y)\log(1-p)$

∴ **Binary Cross - Entropy Loss** : $J_{BCE}(w) = \frac{1}{n}\sum_{i=1}^{n} BCE(y_i, h_w(x_i))$

Regression Task

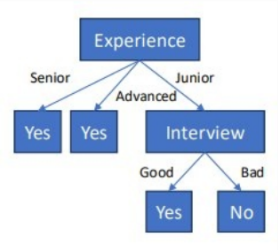
Performance Measure / Loss Functions

- Mean Squared Error : $\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$ * MSE is more sensitive to outliers (penalises them more)
- Mean Absolute Error : $\frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$ ⇒ which to use depends on whether outliers are considered important

Use either MSE / MAE based on ε →
- Huber Loss / log-cosh loss

∴ **Mean Squared Error Loss** (for Linear Regression) : $J_{MSE} = \frac{1}{2n}\sum_{i=1}^{n}(h_w(x_i) - y_i)^2$

Machine Learning : Decision Trees



For Classification Tasks, where
- Internal Nodes : Tests on features
- Branches : Outcome of said test
- Leaf Nodes : Assign Predictions

* Building of Decision Tree Algo is:
- Top-Down
- Greedy
- Recursive

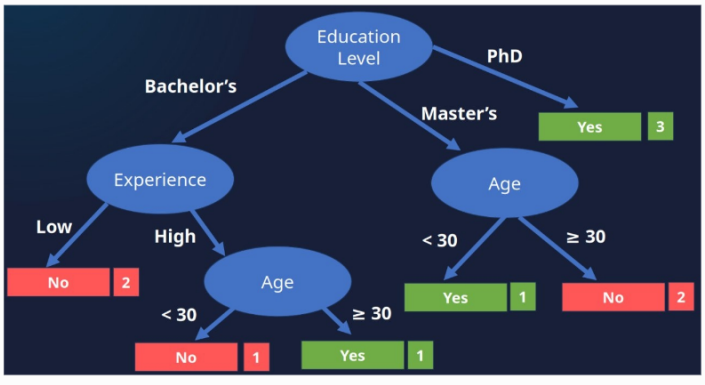
NP-Hard!! → It is possible to have multiple distinct decision trees for the same set of data we typically want 'compact' decision trees w less nodes to prevent overfitting.

∴ we make the problem intractable by using Greedy Heuristics (Information Gain): ← (Next Page)

Example

Experience	Education Level	Age	Hire?
Low	Bachelor's	Less than 30	No
Low	Master's	Less than 30	Yes
Low	PhD	Less than 30	Yes
Low	Bachelor's	Greater than or equal to 30	No
Low	Master's	Greater than or equal to 30	No
Low	PhD	Greater than or equal to 30	Yes
High	Bachelor's	Less than 30	No
High	PhD	Less than 30	Yes
High	Master's	Greater than or equal to 30	No
High	Bachelor's	Greater than or equal to 30	Yes
6 v 4	4 v 3 v 3	5 v 5	5 v 5

→ Full Decision Tree



$H(\text{Hire}) = H(5/10; 5/10) = 1$

$H(\text{Hire} \mid \text{Exp.}) = 6/10\, H(3/6; 3/6) + 4/10\, H(2/4; 2/4) = 1$ IG = 0

$H(\text{Hire} \mid \text{Edu.}) = 4/10\, H(1/4; 3/4) + 3/10\, H(1/3; 2/3) + 3/10\, H(1, 0) = 0.6$ IG = 0.4

$H(\text{Hire} \mid \text{Age}) = 5/10\, H(3/5; 2/5) + 5/10\, H(2/5; 3/5) = 0.971$ IG = 0.29

∴ 1st level : IG(Edu.) is greatest → choose Education

$H(\text{Bachelor}) = H(1/4; 3/4) = 0.811$

$H(\text{Hire} \mid \text{Bachelor, Exp.}) = 2/4\, H(1, 0) + 2/4\, H(1/2; 1/2) = 0.5$

$H(\text{Hire} \mid \text{Bachelor, Age}) = 2/4\, H(0, 1) + 2/4\, H(1/2; 1/2) = 0.5$

∴ Both IG = 0.811 − 0.5 = 0.311
Break tie using Experience for Bachelors

$H(\text{Masters}) = H(1/3; 2/3) = 0.918$

$H(\text{Hire} \mid \text{Masters, Exp.}) = 2/3\, H(1/2; 1/2) + 1/3\, H(0, 1) = 0.667$ IG = 0.251

$H(\text{Hire} \mid \text{Masters, Age}) = 1/3\, H(0, 1) + 2/3\, H(0, 1) = 0$ IG = 0.918

∴ Choose Age as next node for Exp. = Masters

* After this, fill in since only 1 possible feature left for both

---

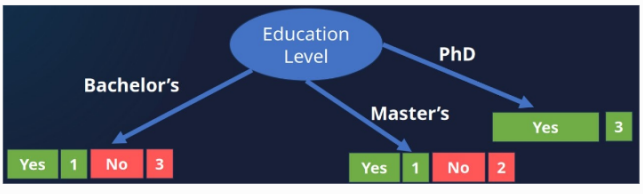
Theorem: Representational Completeness of Decision Trees

∀ finite set of consistent examples w discrete features and a finite set of label classes,
∃ decision tree that is consistent with the examples

* consistency : Suppose lbl = student pass / fail but we only have Gender. we CANNOT perfectly label this dataset.

∴ To avoid overfitting, we prune the tree

1. By min-sample leaves (i.e. 3) →
2. By max-depth (i.e. 1) →



Typically, we then either take the majority or some pre-defined default value

Information Gain

Entropy: Quantifies the uncertainty / randomness of a Random Variable ← If outcomes are decisions, the level of uncertainty of the decision

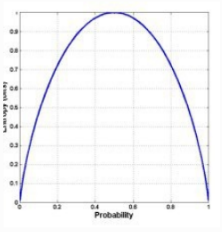
Let Y be a R.V w/ outcome values $y_1, \dots, y_k$ and probabilities $P(y_1), \dots, P(y_k)$

Then, the entropy is defined as: $H(Y) = H(P(y_1), \dots, P(y_k))$

$$= -\sum_{i=1}^{k} P(y_i) \log_2 P(y_i)$$

$P(0)$ OR $P(1) \rightarrow$ Entropy $= 0$

$P(0.5) \rightarrow$ Entropy $= 1$



For Binary Outcomes: ← Easily extendable to ≥ 3 outcomes

$P(pos) = p/(p+n)$

$P(neg) = n/(p+n)$

$$H(Y) = H(P(pos), P(neg)) = -\left(\frac{p}{p+n} \log_2 \frac{p}{p+n}\right) - \left(\frac{n}{p+n} \log_2 \frac{n}{p+n}\right)$$

Specific-Conditional Entropy

Entropy of a R.V. Y conditioned on the Discrete R.V. X taking some fixed value x

$$H(Y|X=x) = H(P(y_1|x), \dots, P(y_k|x))$$

In ctxt. of decision trees, the entropy of a single branch v the entropy of all branches of a decision

Conditional Entropy

Quantifies the entropy that remains of a R.V. Y when another R.V. X is known

$$H(Y|X) = \sum_{x \in X} P(X=x) H(Y|X=x)$$

Information Gain

Measure of the reduction in uncertainty after a decision was made

$$IG(Y; X) = H(Y) - H(Y|X)$$

Common Entropy:

$H(1/2; 1/2) = 1$

$H(1/3; 2/3) = 0.918$

$H(1/4; 3/4) = 0.811$

$H(1/5; 4/5) = 0.722$

$H(2/5; 3/5) = 0.971$

$H(1/6; 5/6) = 0.650$

$H(1) = 0$

$H(1/7; 6/7) = 0.591$

$H(2/7; 5/7) = 0.863$

$H(3/7; 4/7) = 0.985$

$H(1/8; 7/8) = 0.543$

$H(3/8; 5/8) = 0.954$

Machine Learning : Linear Regression

Regression : Based on the dataset, find a function that predicts the target $y \in \mathbb{R}$ for a new data point $x \in \mathbb{R}^d$

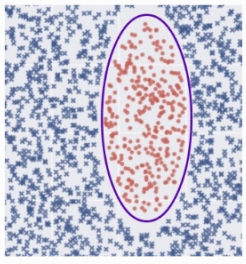
Linear Model

Given an input vector x of dimension d, a linear model is a function with the following form:

$$h_w(x) = w_0 + w_1 x_1 + \dots + w_d x_d$$ ← That is, we assign weights to each feature $x_i \in x$

w_0 : y-intercept

$$= w^T x = \sum_{j=0}^{d} w_j x_j$$



E.g.
- x & y : offset from $(0, 0)$
- x^2 & y^2 : elipsis shape
- xy : rotation

Feature Transformation

Not all data can be fitted perfectly with just a linear, straight line ⇒ Transform features
- Polynomial
- Log
- Exponential

Since this is a Regression task, we use MSE / MAE / Huber Loss / etc.

Regardless of the loss function used, we want to minimise our loss, i.e.

$$J_{MSE} = \frac{1}{2n} \sum_{i=1}^{n} (h_w(x_i) - y_i)^2$$ ⟶ To minimise ⟶ i.e. Gradient $\nabla J_{MSE}(w) = 0$

$$\forall w_i, \frac{\partial J_{MSE}(w)}{\partial w_i} = 0$$

Normal Equation

When using Normal Equation, we want to find the best w, that minimises loss, by just solving an equation:

$$w = (X^T X)^{-1} X^T y$$ →

$$\frac{\partial J_{MSE}(w)}{\partial w_j} = \frac{1}{n} \sum_{i=1}^{n} (w^T x_i - y_i) x_i = 0$$

↓ Expressed in vectors & matrices

$$X^T (Xw - y) = 0$$

↓ Assume invertibility of X

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

Pros :
- will always produce the same, optimal solution for the same set of data

Limitations :
- Time Complexity = $O(d^3)$, to invert & multiply matrix
- Only works if $(X^T X)^{-1}$ is invertible
- Only works for linear models

Gradient Descent

Gradient Descent

1. Start at some randomly initialised ω
2. Update ω with a step in the opposite direction of the gradient ← Repeat until termination criterion is satisfied
   E.g. $|\omega_{j+1} - \omega_j| \leq |\delta|$
   max. no. of steps

$$\omega_j \leftarrow \omega_j - \gamma \frac{\partial J(\omega_0, \omega_1, \dots, \omega_d)}{\partial \omega_j}$$

γ ; learning rate hyperparameter

* When setting γ,
- Too Large : May step over minimum
- Too Small : Takes a long time to converge

Learning Rate Scheduling
Change the value of γ depending on current epoch
Larger at the start, smaller as we iterate

Variants :
1. Batch Gradient Descent : Uses all samples in each iteration
   - Slower iterations
   - Smoother Gradient
2. Mini-Batch Gradient Descent : Use small, different, batches of d_{train} in each iteration
   - Faster iterations
   - Noisy Gradient
3. Stochastic Gradient Descent : Use 1 random training sample per iteration
   - Fastest iterations
   - Noisiest Gradient

Pros :
- Can be used for all datasets
- Faster on datasets w/ large no. of features

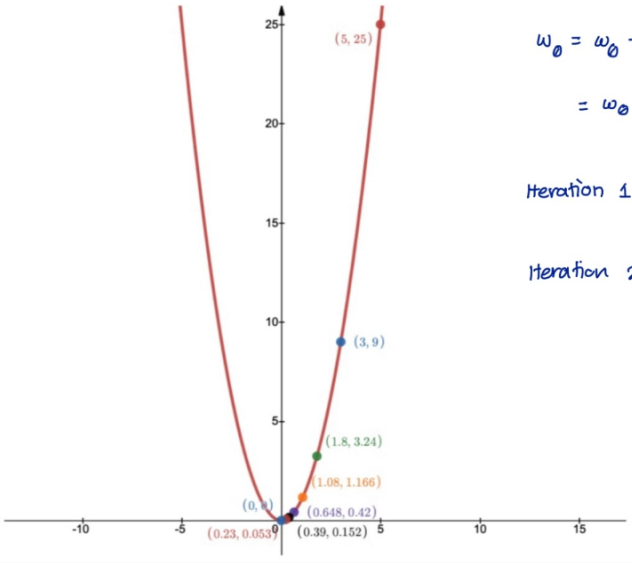
Limitations / Cons :
- Sensitive to magnitude of data ← scaled data may be given more importance ⇒ **Feature Scaling**
  - Min-Max Scaling : $[0, 1]$
    $x_i' = \frac{x_i - \min(x_i)}{\max(x_i) - \min(x_i)}$
  - Standardisation
    $x_i' = \frac{x_i - M_i}{\sigma_i}$
    Alternatively, diff. γ_j for each weight ω_j
- Requires tuning of hyperparameters
- Many iterations
- May get stuck on local minimum / plateau on non-convex optimisation

Examples - Gradient Descent

1)

Given a simple function $y = x^2$, we know the gradient is $\frac{dy}{dx} = 2x$. As such, the minimum of this function is 0.

Starting from the point $p = (5, 25)$ and using different γ values in $\{10, 1, 0.1, 0.01\}$, perform gradient descent on values of x. Tabulate the values of $p = (x, y)$ (where $y = x^2$) over **5 iterations** for each learning rate. State which γ value gives the best convergence by observing the oscillations of the values and the convergence to $(0, 0)$. An example is shown below for $\gamma = 0.2$ over 7 steps:



$$w_0 = w_0 - \gamma \frac{\partial J(w_0)}{\partial w_0}$$

$$= w_0 - 0.2(2x)$$

Iteration 1 : $x = x - 2 = 5 - 2$

$y = 3^2$

Iteration 2 : $x = x - 0.2(2x)$

$= 3 - 0.2(6)$

$x = 1.8 - 0.2(2 \times 1.8)$

$= 1.08$

2) Given the Linear Regression Hypothesis $h(x) = w_0 + w_1 x$, where $\begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 1 \end{bmatrix}$ and two datapoints : $(x_0, y_0) = (0, 0)$, $(x_1, y_1) = (2, 1)$ as well as the loss function : $\frac{1}{2} \sum_{x_i} (h(x_i) - y_i)$ and learning rate $\gamma = \frac{1}{4} = 0.25$, perform a single update to $\begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$ using gradient descent.

$$\frac{\partial J(w)}{\partial w_0} = (h(x_0) - y_0) + (h(x_1) - y_1)$$

$$= 2$$

$$\frac{\partial J(w)}{\partial w_1} = (h(x_0) - y_0)x_0 + (h(x_1) - y_1)x_1$$

$$= 3$$

$$\therefore \begin{bmatrix} w_{0,new} \\ w_{1,new} \end{bmatrix} = \begin{bmatrix} 0.5 \\ 1 \end{bmatrix} - \frac{1}{4} \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

$$= \begin{bmatrix} 0 \\ 0.25 \end{bmatrix} \#$$

Machine Learning : Logistic Regression

Logistic Model ← Binary Classification : 1 / 0

$$h_w(x) = \sigma(w_0 + w_1 x_1 + \dots + w_d x_d) \text{ where } \sigma(z) = \frac{1}{1+e^{-z}}$$

← Sigmoid Function

$$= \sigma(w^T x) = \sigma\left(\sum_{j=0}^{d} w_j x_j\right)$$

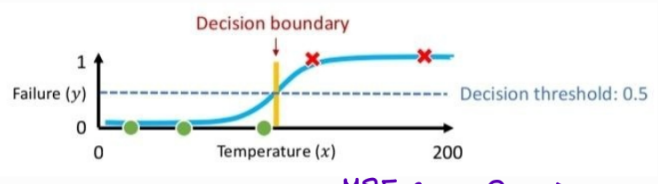
$\sigma'(x) = \sigma(x)(1-\sigma(x))$

↑ Logistic Function

* Outputs the probability that a given datapoint is of Class 1, aka. the Confidence Score of model

Typically, we then set some threshold value τ, s.t. if $P(Y|X=x) \geq \tau$, then Class 1
Otherwise, Class 0

Decision Boundary : Seperates the two classes in the feature space, defining the point at which the model changes its predicted class
Intersection btw the model's function and the decision threshold



* Instead of using MSE for loss function, we use (Binary) Cross - Entropy
Similar to Linear Regression, we also use Gradient Descent (See : Previous)

← MSE loss function is non-convex for logistic regression → stuck on local minimum

Binary Cross - Entropy Loss : (See : Performance Measure and Loss Functions)

$$J_{BCE}(w) = \frac{1}{n}\sum_{i=1}^{n} BCE(y_i, h_w(x_i))$$

* Remember $h_w(x) = \sigma(w^T x)$

$z = X @ \text{weights}$

$\hat{y} = \sigma(z)$

$$\frac{\partial J_{BCE}}{\partial w_j} = \frac{1}{n}\left(X^T \cdot (\hat{y} - y)\right)$$

Multi-Class Classification

- One - vs - One : Each Classifier votes for a class and the class with the most votes is selected
  - I.e. Classifier 1 : Class 1 v. Class 2 ; Classifier 2 : Class 1 v Class 3 ; Classifier 3 : Class 2 v Class 3
  - No. of Binary Classifiers : ${}^{K}C_2 = \frac{K(K-1)}{2}$
- One - vs - Rest : The Classifier with the highest probability output determines the class
  - I.e. Classifier 1 : Class 1 v. NOT Class 1 ; Classifier 2 : Class 2 v. NOT Class 2 ; Classifier 3 : Class 3 v. NOT Class 3
  - No. of Binary Classifiers : K

Multi - Label Classification

- Transformation into Binary Classification : ← Ignores correlation & constraints btw labels
  - Binary Relevance : Train one independent (Binary) Classifier per label
  - Classifier Chain : Binary Relevance, but feed predictions of previous label as features for subsequent labels ← Sensitive to chain order
- Transformation to Multi - Class Classification :
  - Label Powerset : Treat each unique label-set as a single multi-class label

Preserves correlation but can quickly explode in classes and be data sparse (little / no info) for some classes

Examples – Logistic Regression

1) Given the following weight vectors for a "one v one" logistic Regression model :

$$\omega_{\text{Dog / Rabbit}} = \begin{bmatrix} 0 \\ 0.4 \\ 0.1 \end{bmatrix}, \quad \omega_{\text{Rabbit / Cat}} = \begin{bmatrix} -1 \\ 0.2 \\ -0.4 \end{bmatrix}, \quad \omega_{\text{Cat / Dog}} = \begin{bmatrix} 2 \\ -0.5 \\ 0.3 \end{bmatrix}$$

and the input $x = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$, determine which class the model predicts

$$\text{Dog / Rabbit} \rightarrow \begin{bmatrix} 0 & 0.4 & 0.1 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} = 1 \quad \Rightarrow \text{Dog}$$

$$\text{Rabbit / Cat} \rightarrow \begin{bmatrix} -1 & 0.2 & -0.4 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} = -1.2 \quad \Rightarrow \text{Cat}$$

$$\text{Cat / Dog} \rightarrow \begin{bmatrix} 2 & -0.5 & 0.3 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} = 1.1 \quad \Rightarrow \text{Cat}$$

$\therefore$ 2 v 1 v 0 $\Rightarrow$ The model predicts cat #

Feature Transformation

Take a d-dimensional feature vector and transform it into a M-dimensional feature vector.

⇒ $M \geq d$ if we are creating new features from the original data
⇒ $M < d$ if we are performing some dimensionality reduction feature transformation

* With more weights, the model becomes more complex (overly) and is very prone to overfitting as well as taking longer to train.

Possible Solutions:
- Dimensionality Reduction
- Using Dual Formulation of the model (Next page)
  - ⇒ Based on Primal Formulation, we need 1 weight / feature
  - ⇒ Based on Dual Formulation, we need 1 weight / training data

For Dimensionality Reduction, the main idea is to
Remove Redundant Features

To do this, we make use of SVD Decomposition:

$$A = U \Sigma V^T$$

$U : d \times d$ ← Left Singular Vectors
$\Sigma : d \times n$ ← Diagonal Matrix where σ_j are the Singular Values
$V : n \times n$ ← Right Singular Vectors

where the n Singular Values tell us the importance of the new basis vectors

Curse of Dimensionality
The number of samples n needed to learn a hypothesis class increases exponentially with the number of features d

$$n = O(2^d)$$

Since the Singular Values in Σ are ordered in Descending Order, it suffices to just keep the top k vectors and Singular Values ← Notice that this essentially removes some 'redundant' features and compresses the data

↓ How to decide the value of k?

Choose the value of k such that we still retain enough information, and equivalently, x variance in the data

Thus, calculate the minimum k that is still able to retain x variance →

THEOREM: Given a mean-centered data matrix $\hat{X}^T$, the values $\sigma_j^2/(n-1)$ are variances of the data in the basis defined by $u^{(j)}$

We can also use Principal Component Analysis (PCA) which can either:
a) Performed directly using Covariance Matrix of the data (then calculating eigenvalues & eigenvectors) ← Also explains variance
b) Performed using SVD

Alternatively, we can also use an Encoder-Decoder NN Architecture (Autoencoder)
⇒ The model is trained to be able to reconstruct its input after compressing it

Input → Encoder → Compressed Data → Decoder → Output

↕ Minimise difference
Input (aka. Reconstruction Loss)

Primal & Dual Formulation (++ Kernels)

Representer Theorem

For a model parameterized w a weight vector w whose prediction is of the form $f(w^T x)$, the best weight vector w that minimises certain regularized loss $J_{reg}(w)$ can be expressed as:

$$w = \sum_{j=1}^{n} \alpha_j x^{(j)}$$ where α is a vector of real numbers $\alpha_1, \alpha_2, \dots, \alpha_n$ (n-dimensional)

We can thus express our Linear Models equivalently as follows:

$$\underbrace{h_w(x) = w^T x}_{\text{Primal formulation (Linear model expressed in terms of } w)} = \underbrace{\sum_{j=1}^{n} \alpha_j x^{(j)T} x = h_\alpha(x)}_{\text{Dual Formulation (Linear model expressed in terms of } \alpha)}$$

α ← Dual coefficient

* Thus, for our optimal model, the solution for $h_w(x)$ = the solution for $h_\alpha(x)$

* Minimising $J^{MSE}_{L_2}(w)$ is equal to minimising $J^{MSE}_{L_2}(\alpha)$

Of course, for our linear models, we want to minimise our loss functions

Primal Objective

To minimise loss function in our Primal Formulation, we want to minimise over w (this should be as per normal)

$$J^{MSE}_{L_2}(w) = \frac{1}{2n} \sum_{i=1}^{n} (w^T x^{(i)} - y^{(i)})^2 + \lambda \sum_{i=0}^{d} w_i^2$$

Dual Objective

Similarly, we want to minimise loss, but in our Dual Formulation, we instead minimise over α

$$J^{MSE}_{L_2}(\alpha) = \frac{1}{2n} \sum_{i=1}^{n} \left(\sum_{j=1}^{n} \alpha_j x^{(j)T} x^{(i)} - y^{(i)} \right)^2 + \lambda \sum_{i=0}^{d} \left(\sum_{j=0}^{d} \alpha_j x^{(j)} \right)^2$$

The Dual Formulation of a Linear Model can also be written as follows:

$$h_\alpha(x) = \sum_{j=1}^{n} \alpha_j x^{(j)T} x = \sum_{j=1}^{n} \alpha_j \langle x^{(j)}, x \rangle$$

← Inner Product ⇒ (In Euclidean Space) $\langle u, v \rangle = u^T v = u \cdot v$

* For any feature mapping function ϕ, there must exist a kernel function $K_\phi(u, v)$ that computes $\langle \phi(u), \phi(v) \rangle$ for any vectors u, v

↳ Using this, our model becomes: $h_\alpha(\phi(x)) = \sum_{j=1}^{n} \alpha_j \langle \phi(x^{(j)}), \phi(x) \rangle$

$$= \sum_{j=1}^{n} \alpha_j K_\phi(x^{(j)}, x) = h^\phi_\alpha(x)$$

Kernel Trick: we do not have to expand $\phi(x)$ explicitly, we only have to compute $K_\phi(x^{(j)}, x)$
Any model that uses a kernel trick is called a **Kernel Machine**.

* Kernel Functions calculate a similarity between 2 data points in a high dimension space

There are many common kernel functions that are efficient to compute
i.e. Polynomial degree k kernel
Gaussian Kernel (aka. Radial Basis Function Kernel)
etc.

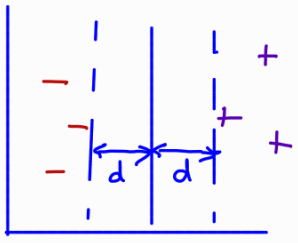
MERCER'S THEOREM: For any valid kernel K_ϕ, there must exist a vector space w feature mapping ϕ

↑ Continuous symmetric positive-definite kernel (idk bro)

Support Vector Machines

The main idea for SVMs is to find the most optimal decision boundary (aka Hyperplane) by maximising the 'margin' between the classes

Doing so improves generalisation of the model



Dist. from a point x to the hyperplane : $\frac{|w^T x_i + b|}{\|w\|}$

Given an input vector x of dimension d, an SVM model is a function w the following Primal Formulation :

$$h_w(x) = \text{sign}(w^T x + b)$$

$$\text{sign}(z, b) = \begin{cases} 1, & z > b \\ 0, & z = b \\ -1, & z < b \end{cases}$$

default = 0 (offset of hyperplane)

Primal Objective

$$\max_w \frac{1}{\|w\|} \quad \text{OR} \quad \min_w \frac{\|w\|^2}{2}$$

⇐ we want to maximise the margin d

Using Kernel Trick, we can classify even if not linearly seperable otherwise

The Dual Formulation (w kernel) of the model :

$$h_\alpha^\phi(x) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i k_\phi(x^{(i)}, x)\right)$$

Dual Objective

$$\max_\alpha \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y^{(i)} y^{(j)} k_\phi(x^{(i)}, x^{(j)})$$

$$\text{s.t. } \forall_i, \alpha_i \geq 0 \text{ and } \sum_{i=1}^{n} \alpha_i y^{(i)} = 0$$

For SVMs, we only care about the Support Vectors $x^{(i)}$ which lie directly on the margin boundaries

⇒ $\alpha_i > b$ iff $x^{(i)}$ lies directly on the margin boundaries

⇒ $\alpha_i = b$ iff $x^{(i)}$ is strictly outside the margin boundaries

* We should have no point for which $\alpha_i < b$

THEOREM : For linearly seperable data with r "effective" dimensions, where $r \leq d$, the number of support vectors is at least $r + 1$

Practically speaking, the number of support vectors are much less than the number of data points

⇒ Sparsity → we actually only need the support vectors and their corresponding dual coefficients α_i.

⇒ Results in a more compact and fast model (inference)

$$h_\alpha(x) = \text{sign}\left(\sum_{x^{(i)} \in S} \alpha_i k(x^{(i)}, x)\right)$$

SVM Problem : we want to try and maximise the smallest distance of all points to the hyperplane

$$\max_{w,b}\left[\min_i \left(\frac{|w^T x_i + b|}{\|w\|}\right)\right] = \max_{w,b}\left(\frac{1}{\|w\|} \min_i |w^T x_i + b|\right)$$

$\forall i, y_i(w^T x_i + b) \geq 1$

Since y_i is the class / label, it must have value 1 or -1.

Then, for this constraint to be satisfied, we have that

if $y_i = -1$, $(w^T x_i + b) \leq -1$

otherwise, $(w^T x_i + b) \geq 1$

In both cases, we can have some value of $(w^T x_i + b)$

s.t. $|w^T x_i + b| = 1$

Therefore, since freely scaling w & b does not affect the hyperplane,

Let $\min_i |w^T x_i + b| = 1$,

our SVM problem becomes : $\max_{w,b}\left(\frac{1}{\|w\|}\right)$

This is equivalent to minimising $\|w\|$ and thus also minimising $\frac{1}{2}\|w\|^2$ since $\frac{d}{dw}\left(\frac{1}{2}\|w\|^2\right) = \|w\|$

∴ The optimisation problem can also be stated as $\min_{w,b}\left(\frac{1}{2}\|w\|^2\right)$ #

shit formatting ...

Unsupervised Learning (Clustering)

With Unsupervised Learning, we have the data, but no labels. That is,
Given a set of n data points, our model should learn patterns in the data.

Clustering : Group similar data points into clusters or groups
* The number of clusters is not defined by the data (but we can tweak as a hyperparameter)

Centroid : The average of the set of points in a cluster → $\mu_j = \frac{1}{n_j} \sum_{i=1}^{n_j} x^{(i)}$

There are a few different clustering algorithms :

1. K-Means Clustering → Randomly initialise the starting centroids
2. K-Medoids Clustering → Randomly initialise one of the points to be a centroid
3. K-Means++ Clustering → Randomly initialise the first centroid then choose the rest w/ a weighted probability distribution $P(x) \propto D(x)^2$
   i.e. Choose further points to be the other centroids

Regardless, each algorithm performs the update similarly :

1. For each data point, assign it to the cluster formed by the closest centroid
2. Update the position of each centroid
   ⇒ If K-Means : Mean of all the data points
   ⇒ If K-Medoids : Data point closest to the mean of all the data points
3. Repeat until Convergence (i.e. all centroids did not change)

To see the 'goodness' of our centroids, we can measure the **Distortion** of the clusters :

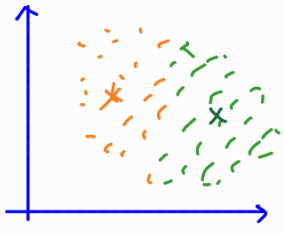
$$J(c^{(1)}, c^{(2)}, \dots, c^{(n)}, \mu_1, \mu_2, \dots, \mu_k) = \frac{1}{n} \sum_{i=1}^{n} \| x^{(i)} - \mu_{c^{(i)}} \|^2$$

that is, the average square distance of each sample to its assigned centroid

> **THEOREM** : Each step in the K-Means algorithm never increases distortion, and always converges

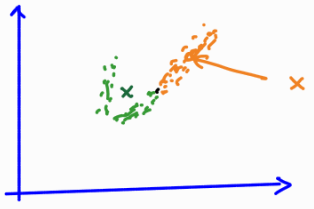
LIMITATIONS :

Non-Linearly Seperable Data



i.e. Some uniform data

Non-Spherical Data



i.e. Since the centroid found based on minimising average distance

Possible Solutions :

1. Feature Transformation
2. Gaussian Mixture Model
3. Spectral Clustering

Neural Networks

Neural Networks are composed of **Layers**, which are itself composed of **Neurons**

1. Fully Connected Layers
2. Convolutional Layer
3. Pooling Layers

Computes a weighted sum of input features and passes the result through an Activation Function

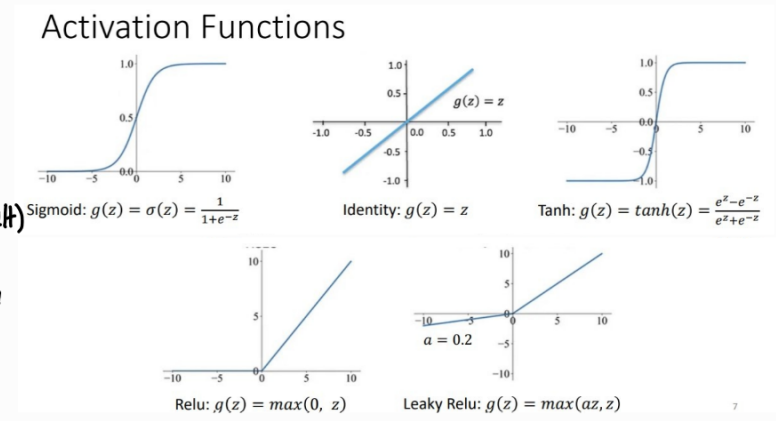
$$z = \sum_{j=0}^{d} w_j x_j$$

$$\hat{y} = g(z)$$

(Activation Function)

Types:

- Multi-Layer Perceptron (MLP)
- Convolutional Neural Network (CNN)
- Recurrent Neural Network (RNN)
  - ⇒ Also keep track of a Hidden State (ē prev. result) to make prediction for ē nxt input
  - ⇒ To prevent degradation of information with large sequential data, we use Attention



Query : Current Element
Key : Information keywords
Value : Actual Information

$$\text{Attention Score} = \frac{\text{weighted sum of all Values}}{\text{Relevant ctxt}}$$

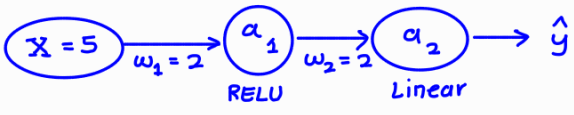
Forward Propagation : The process by which input data passes through a neural network to generate the output

Backward Propagation : The process by which the weights of each neuron in each layer is updated to better fit the data

1. Calculate $\hat{y}$
2. Compute the loss L
3. Let $z = \sum_{j=0}^{d} w_j x_j$ and $\hat{y} = g(z)$, then the gradient of the loss function w.r.t w_j is:

$$\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w_j}$$

Example



True Label : $y = 18$
Learning Rate : $\rho = 0.1$
Loss Function : MSE = $\frac{1}{2}(\hat{y} - y)^2$
$a_1 = 10$; $a_2 = 20$ → $\hat{y} = 20$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2}$$

$$= (\hat{y} - y)\left(\frac{\partial}{\partial w_2} w_2 a_1\right)$$

$$= (20 - 18)(a_1)$$

$$= 2(10)$$

$$= 20$$

$$\therefore w_2 \leftarrow w_2 - \rho\left(\frac{\partial L}{\partial w_2}\right)$$

$$\leftarrow 2 - (0.1 \times 20)$$

$$\leftarrow 0$$

MSE Loss : $\frac{\partial L}{\partial \hat{y}} = \hat{y} - y$

BCE Loss : $\frac{\partial L}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$

Batching

When training / during inference, we can also batch our input data to :

1) Shorten training time
2) Improve performance
   ⇒ Identify similar features across multiple images

Limitations / Weaknesses

1. Overfitting → **Early Stopping** when ē model's performance on ē validation set begins to worsen
   → **Dropout** : Randomly sets the output of neurons to 0
   Makes the model less reliant on specific neurons
2. Vanishing Gradient → Use Leaky ReLU, etc.
3. Exploding Gradient → Clip gradients

Layers in Neural Networks

Fully Connected Layers

Every neuron in each layer is connected to every other neuron in the next layer
Typically, the input layer & the output layer are Fully Connected.

Convolutional Layers

Uses Convolution Operation to extract features from the input data
Typically for Image data

0.5. If padding = k, add k pixels around the input data (all sides) w default value 0
1. Overlay the kernel (starting from the top left corner)
2. Compute the weighted sum
3. Slide the kernel (based on stride) & repeat until we go through the entire input

If stride, size, etc. is s.t. not exact for that row, ignore stride & fill to last position

lowers computational cost while preserving features

The resulting size of the data after applying a kernel of size $H_k \times W_k \times C_k$ on input data of size $H_I \times W_I \times C_I$ is:

$$\left\lceil \frac{H_I - H_k}{\text{Stride}} \right\rceil \times \left\lceil \frac{W_I \times W_k}{\text{Stride}} \right\rceil \times \text{num\_kernels}$$

Pooling Layers

Downsamples the feature maps
1. Max Pool: Keep the maximum value in the window ← Keeps most prominent features But loses small details
2. Average Pool: Keep the average value in the window ← Get a more big-picture view

RNN Layer

Introduces a Hidden State H for which $h^{[i]}$ is used as part of the input to the neurons (& have weights)
At an arbitrary time step t,

$$h^{[t]} = g^{[h]}\left((W^{[xh]})^T x^{[t]} + (W^{[hh]})^T h^{[t-1]}\right)$$
$$\hat{y}^{[t]} = g^{[y]}\left((W^{[hy]})^T h^{[t]}\right)$$

where $h^{[t-1]} = \begin{bmatrix} h_1^{[t-1]} \\ \vdots \\ h_n^{[t-1]} \end{bmatrix}$

Self-Attention Layers

Each input element can find relevant context from all elements in the same sequence it belongs to

Steps

1. Transforms the input vectors into Query, key and Value using 3 learnable matrices (shared across inputs)

$$Q = W^q X \;;\; K = W^k X \;;\; V = W^v X$$

2. Compute the Attention Score (using Softmax) $A = \frac{K^T Q}{\sqrt{d_k}} \rightarrow A'$

$$a_{it} = \frac{(k^i)^T q^{[t]}}{\sqrt{d_k}} \rightarrow a'_{it} = \frac{e^{a_{it}}}{\sum_{i=1}^{t} e^{a_{it}}}$$

Just Softmax function to calc. probability

3. Compute the weighted sum of all Values $H = VA'$

$$h^{[t]} = \sum_{i=1}^{T} a'_{it} v^{[i]}$$

* d is the dimension of the vector / matrix
i.e. $W^q \in \mathbb{R}^{d_q \times d}$
$W^k \in \mathbb{R}^{d_k \times d}$
$W^v \in \mathbb{R}^{d_v \times d}$

(Variant) Masked Self-Attention Layer

Each input element can find relevant context only from previous elements and itself

Same Steps but in Step 2, $a_{it} = \begin{cases} a_{it} & , i \le 1 \\ -\inf & , i > 1 \end{cases}$ ← All elements after the current element is effectively ignored

(Variant) Cross-Attention Layer

Elements in one sequence can find relevant context from another sequence

same Steps except $K = W^k Z$ where $Z = [z^{[1]}, z^{[2]}, \ldots, z^{[n]}]$
$V = W^v Z$ $z^{[i]} = [x^{[1]}, x^{[2]}, \ldots, x^{[T]}]$

In other words, Key and Value now contain information about other sequences

Various NN Tricks & Strategies

Positional Encoding

Explicitly inject positional information into the original input vectors

$$x'^{[t]} = x^{[t]} + PE^{[t]}$$

3 Requirements for PE:

1) Unique : Each position has a distinct encoding
2) Consistent : Can be generalised across all seq. lengths (Not dependent on T)
3) Bounded : within the range $[-1, 1]$

Using sin and cos to generate the k-th element,

$$PE_k^{[t]} = \begin{cases} \sin\left(\frac{t}{C^{k/d}}\right), & k \% 2 = 0 \\ \cos\left(\frac{t}{C^{(k-1)/d}}\right), & k \% 2 = 1 \end{cases}$$

where C is a constant

Teacher Forcing

Use the true label as the input for the next step, not the model's prediction
Used during training for sequence prediction

✓ Can easily parallelize training
✓ Prevent confusion & inaccuracies due to initial random predictions

X Can cause "train-test-mismatch" when inference
⇒ Possible solution: Randomly decide whether to feed model's prediction or true label (Gradually reduce true label % over time)

Transfer Learning

Use the weights of another model to be the initial weights of our actual model (instead of random initialisation)
Saves time and Computational Resources when training the actual model

1. Train another model on a seperate but related task (often w the same / similar data)
2. Use the weights from this model ^

* If we have unlabelled data, this can be used to train the first model

a) Rotation Prediction : Predict rotation of the image
b) Contrastive learning : Predict similar vectors / inputs
c) Image Inpainting : Fill in masked input
d) Next-word Prediction : Predict next word in sequence

} when generating the data, we can label them according to these tasks