CS2109S Finals

AY24/25 SEM 1

github/SelwynAng

1 Solving Problems by Searching

1.1 Designing an Agent

- **Assumptions:** Goal-based agent | Env is fully observable, deterministic, static, discrete
- **Problem-solving Agent:** Agent that plans ahead (considers a seq. of actions that form a path to a goal state), undertakes SEARCH process
- **Steps:**
  1. Goal Formulation: *(What do we want?)*
  2. Problem Formulation: *(How the world works?)* $\rightarrow$ States (state space), Initial State(initial state of agent), Goal State/Test (goal state of agent), Actions (things that agent can do in a given state), Transition Model (specifies outcome of an action to a given state & how it leads to new states), Action Cost Function (cost of performing an action)
  3. Search: *(How to achieve it?)* $\rightarrow$ Path (seq. of actions), Solution (path to a goal)
  4. Execute
- **Representation Invariant:** A condition that must be true over all valid concrete representations of a class

1.2 Search Algorithms (Introduction)

- **Search Algorithm:** Takes in search problem (input), returns solution/failure (output) | Defined by Order of Expansion (FRONTIER)
- **Evaluation Criteria:**
  1. Time Complexity: # of nodes generated/expanded
  2. Space Complexity: Max # of nodes in memory
  3. Completeness: Does it return solution if it exists?
  4. Optimality: Does it always find least cost solution?
- **Checking of Goal State:**
  - New state is checked for goal state before new states are PUSHED to frontier $\rightarrow$ Expand less states, may skip states with less cost
  - State is checked for goal state after state is POPPED from frontier $\rightarrow$ Expand more states, will not skip states with less cost

1.3 Search Algorithms (Uninformed Search)

- **Key Idea:** Search Algo is given no clue about how close a state is to the goal | Can be Tree or Graph Search
- **BFS:** Queue Frontier | Time Complexity: $O(b^d) = 1 + b + b^2 + \ldots + b^d$, where b is branching factor, d is depth of optimal solution | Space Complexity: $O(b^d)$ when expanded until last child in worst case | Completeness: Complete if b is finite | Optimality: Optimal if step cost is same everywhere
- **UCS:** Priority Queue (path cost) Frontier, where path cost == cost from root to a state | Time Complexity: $O(b^{C^*/\epsilon})$, where C^* is cost of optimal solution, ϵ is minimum edge cost $\rightarrow C^*/\epsilon$ is est. depth of optimal solution in worst case | Completeness: Complete if $\epsilon > 0$ and C^* is finite (if $\epsilon = 0$, zero cost cycle may occur) | Optimality: Optimal if $\epsilon > 0$
  Note: BFS is special case of UCS where step cost == 1 for every edge
- **DFS:** Stack Frontier | Time Complexity: $O(b^m)$ where b is branching factor, m is max depth | Space Complexity: $O(bm)$ as only 1 path is expanded at one time | Completeness: Not complete (when depth is infinite or can go back or forth) | Optimality: Not optimal (there can be paths with less cost not explored yet)
- **DLS (Depth Limited Search):** Limit the search depth to l where $l <= m$, backtrack once depth limit is reached | Time Complexity: $O(b^l)$ | Space Complexity: $O(bl)$ | Completeness: Not complete when soln lies deeper l | Optimality: Not optimal when soln lies deeper than l
  Note: We dk the depth of solution, which is a downside
- **IDS (Iterative Deepening Search):** Do DLS with max depth of $0, .., \infty$ $\rightarrow$ return soln if found, otherwise increase depth | Time Complexity: $O(b^d), Overhead = (n_{IDS} - n_{DLS})/n_{DLS}$ | Space Complexity: $O(bd)$ | Completeness: Complete | Optimality: Optimal if step cost is same everywhere
  Note: IDS is not always faster than DFS $\rightarrow$ Consider state space s.t. each state have only single successor & goal node is at depth n $\rightarrow$ IDS will run in $O(n^2)$, DFS will run in $O(n)$
- **Backward Search:** Search from goal
- **Bidirectional Search:** Combine forward search & backward search, stop when 2 searches meet | Time Complexity: $2 * O(b^{d/2}) < O(b^d)$

1.4 Search Algorithms (Informed Search)

- **Key Idea:** Search Algo has a clue on how close a state is to the goal
- **Best First Search:** Priority Queue ($f(n)$) Frontier, where $f(n)$ estimates the goodness of a state (Node with lowest $f(n)$ is selected first to be expanded) | $f(n)$ can be purely heuristic (estimated cost from n to goal) or a combi of path cost & heuristic
- **Greedy Best First Search:** Priority Queue ($f(n) = h(n)$) Frontier, where $h(n)$ is heuristic function that est. cost from n to goal (Expands node that seems closest to goal according to $h(n)$ without considering path cost so far) | Time Complexity: $O(b^m)$ | Space Complexity: $O(b^m)$ | Completeness: Not complete since GBFS might keep expanding nodes based on $h(n)$ without ever finding goal | Optimality: Not optimal since GBFS selects nodes based on $h(n)$ without considering path cost
- A^* **Search:** Priority Queue ($f(n) = g(n) + h(n)$) where $g(n)$ is cost so far to reach n | Time Complexity: $O(b^m)$ | Space Complexity: $O(b^m)$ | Completeness: Complete | Optimality: Optimal
  - If $h(n)$ is admissible $\rightarrow A^*$ using Tree search is optimal
  - If $h(n)$ is consistent $\rightarrow A^*$ using Graph search is optimal
  - *Note: UCS is special case of A^* search where $h(n) = 0$*

1.5 Heuristics

- Estimate cost from node n to goal
- **Admissible Heuristics:** For every node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is true cost to reach goal state from n (Never over-estimate)
- **Consistent Heuristics:** For every node n, every successor n' generated by action a, $h(n) \leq c(n, a, n') + h(n')$ and $h(G) = 0$ (Proof $h(n) - h(n') \leq c(n, a, n')$)
  Note: If $h(n)$ is consistent, $f(n') \geq f(n) \rightarrow f(n)$ is non-decreasing along any path $\rightarrow$ Nodes are expanded in order of increasing f cost
- **Dominance:** If $h_2(n) \geq h_1(n)$ for all $n \rightarrow h_2$ dominates h_1 | If h_2 is admissible $\rightarrow h_2$ is better for search
- **Creating Admissible Heuristics:**
  - Problem with fewer restrictions on actions is called a relaxed problem
  - Cost of an optimal soln to a relaxed problem is an admissible h for original problem
- Consistent heuristic is admissible, Non-admissible heuristic is not consistent

2 Local Search & Adversarial Search

2.1 Local Search

- **Assumptions:** Agent is a Goal/Utility-based agent, Env has a very large state space
- **Informed & Uninformed Search VS Local Search**
  1. IUS: Low to moderate state space | Optimal or no soln | Search path is usually the soln
  2. LS: Very large state space | Good enuf soln is preferable rather than no soln | State is the soln (don't care about search path)
- **Local Search Overview:**
  - Basic Idea: Start somewhere in state space, move towards a better spot
  - Problem Formulation: States(state space), Initial State(initial state of agent), Goal test (optional, coz we actually dk the goal state, rely on eval function instead), Successor Function (possible states from a state), Evaluation Function (Output value/goodness of a state)
- **Hill Climbing Algorithm**

```
current = initial state
while True:
    neighbor = a highest-valued successor of current
    if value(neighbor) <= value(current):
        return current
    current = neighbor
```

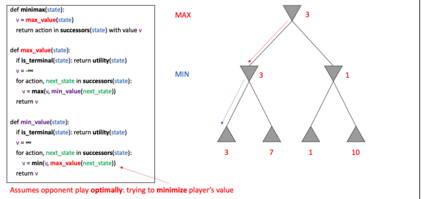
  - Known as Greedy Local Search (pick best amongst neighbors, repeat)
  - Best Soln: State space where eval. function has a max value (global max)
  - Disadvantages: Cannot reach global max if it enters local max, plateau | Sensitive to choice of initial state, poor initial state may result in poor final state (Can overcome with random restarts, walks)
- **Simulated Annealing**

```
current = initial state
T = a large positive value
while T > 0:
    next = a randomly selected successor of current
    if value(next) > value(current): current = next
    else with probability P(current, next , T): current = next
    decrease T
return current
```

  - P(current, next, T) = $e^{(value(next) - value(current))/T}$
  - More exploration of bad states is allowed when T is high, more exploitation is done when T is low $\rightarrow$ basically choosing worse successor may lead to a better max
  - Theorem: If T decreases slowly enough, SA will find global optimum with high probability

2.2 Adversarial Search

- **Assumptions:** Agent is Utility-based | Env is a game (game cannot be single player, partially observable, stochastic, but must be fully observable, deterministic, discrete, terminal states exist, 2 players, zero-sum, turn taking)
- **Minimax Algorithm:**



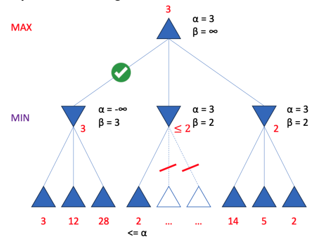
```
def minimax(state):
    v = max_value(state)
    return action in successors(state) with value v

def max_value(state):
    if is_terminal(state): return utility(state)
    v = -∞
    for action, next_state in successors(state):
        v = max(v, min_value(next_state))
    return v

def min_value(state):
    if is_terminal(state): return utility(state)
    v = ∞
    for action, next_state in successors(state):
        v = min(v, max_value(next_state))
    return v
```

Assumes opponent play **optimally**: trying to **minimize** player's value

  - Intuition: MAX wins when utility is high, MIN wins when utility is low | Assign utility values to all terminal states & start tracing from terminal states $\rightarrow$ Eventually, all states will have utility values, starting player can choose a state that will max/min his utility
  - Analysis: Completeness: Complete if tree is finite | Time Complexity: $O(b^m)$ | Space Complexity: $O(bm)$ depth first exploration | Optimality: Optimal against optimal opponent
- **Alpha-beta Pruning**

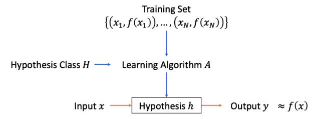


  - Definitions: α is best explored option to the root for MAX player (Highest value for MAX) | β is best explored option along path to the root for MIN player (Lowest value for MIN)
  - Procedure: 1. Assign $\alpha = -\infty$, $\beta = \infty$ for root 2. Propagate values down to the terminal node 3. Update α value at MAX node, β value at MIN node 4. Propagate values up 5. Prune branches of nodes where $\alpha \geq \beta$
- **Minimax with Cutoff**
  - Instead of calling $is\_terminal$, call is_cutoff which returns TRUE if (1): State is terminal or (2): Cut-off is reached
  - Instead of using $utility$, call $eval$ which is an eval. function that returns (1): Utility for terminal states or (2): Heuristic value for non-terminal states

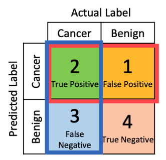
3 Introduction to ML & Decision Trees

3.1 Introduction to ML

- **Definitions:** Computer program is said to learn from experience E w.r.t. some class of tasks T & performance measure P, if its performance at tasks in T, as measured by P, improves with experience E
- **Types of Feedback:**
  1. Supervised Learning: Involves training a model on a labeled dataset, where input data is paired with correct output $\rightarrow$ Model learns to map inputs to outputs based on this labeled data, allowing it to make predictions on new data
     - Regression: Predict continuous input
     - Classification: Predict discrete input
  2. Unsupervised Learning: Deals with dataset that do not have labeled outputs $\rightarrow$ Goal is to identify patterns & structures within data
  3. Reinforcement Learning: Agent learns to make decisions by interacting with an environment $\rightarrow$ Agent receives feedback in the form of rewards or penalties based on its actions $\rightarrow$ Learns optimal behaviors over time
- **Formal Definitions:**



3.2 Performance Measure



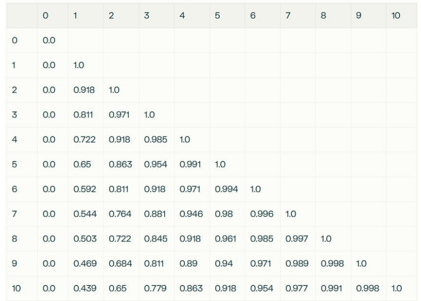
- **Accuracy:** $\frac{TP+TN}{TP+FN+FP+TN}$
- **Precision:** $\frac{TP}{TP+FP}$ (How many selected items are relevant, maximise if FP is costly)
- **Recall:** $\frac{TP}{TP+FN}$ (How many relevant items are selected, maximise if FN is dangerous)
- **F1 Score:** $\frac{2}{1/precision + 1/recall}$

3.3 Decision Trees

- **Traits of Decision Trees:**
  - Decision Trees can express any function of input attributes
  - Consistent Decision Tree for any training set, but probably will not generalize to new examples
  - # of distinct decision trees with n boolean attributes = 2^{2^n}
- **Decision Tree Learning Algorithm**

```
def DTL(examples, attributes, default):
    if examples is empty: return default
    if examples have the same classification:
        return classification
    if attributes is empty:
        return mode(examples)
    best = choose_attribute(attributes, examples)
    tree = a new decision tree with root best
    for each value $v_i$ of best:
        $examples_i$ = {rows in examples with best = $v_i$}
        subtree = DTL($examples_i$, attributes – best, mode(examples))
        add a branch to tree with label $v_i$ and subtree subtree
```

  - mode: Category with the highest number
  - choose attribute: Chooses attribute with the highest information gain
- **Choosing an attribute:**
  - Ideally select an attribute that splits examples into "all positive" or "all negative"
  - Entropy (Measure of randomness):
    $I(P(v_1), \ldots, P(v_n)) = -\sum_{i=1}^{n} P(v_i) log_2 P(v_i)$, where for data set containing p positive & n negative examples, $I(\frac{p}{p+n}, \frac{n}{p+n}) = -\frac{p}{p+n} log_2 \frac{p}{p+n} - \frac{n}{p+n} log_2 \frac{n}{p+n}$
    Note: I(1,0) = I(0,1) = 0, I(0.5, 0,5) = 1
  - Information Gain (Entropy of curr. node - Total Entropy of children nodes):
    $IG(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - remainder(A)$
    remainder(A) = $\sum_{i=1}^{v} \frac{p_i + n_i}{p+n} I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$, where examples are split into v subsets by attribute A



	0	1	2	3	4	5	6	7	8	9	10
0	0.0										
1	0.0	1.0									
2	0.0	0.918	1.0								
3	0.0	0.811	0.971	1.0							
4	0.0	0.722	0.918	0.985	1.0						
5	0.0	0.65	0.863	0.954	0.991	1.0					
6	0.0	0.592	0.811	0.918	0.971	0.994	1.0				
7	0.0	0.544	0.764	0.881	0.946	0.98	0.996	1.0			
8	0.0	0.503	0.722	0.845	0.918	0.961	0.985	0.997	1.0		
9	0.0	0.469	0.684	0.811	0.89	0.94	0.971	0.989	0.998	1.0	
10	0.0	0.439	0.65	0.779	0.863	0.918	0.954	0.977	0.991	0.998	1.0

- **Dealing with continuous valued attributes:** Define a discrete valued input attribute to partition values into discrete set of intervals
- **Dealing with missing values:** Assign most common value of attribute, assign probability to each value and sample, drop attribute, drop rows
- **Overfitting:** Decision Tree is perfect on training data, but worse on test data
- **Occam Razor:** Prefer short/simple hypothesis (long/complex hypothesis that fits data may be coincidence)
- **Pruning:** Prevents nodes from being split even when it fails to cleanly separate examples (Min samples leaf: Merge until leaf node is above min. samples number | Max depth: Merge until leaf nodes are at depth less than max depth)

4 Linear Regression

4.1 Linear Regression Basics

- **Purpose:** Given $h_w(x) = w_0 + w_1 x + \ldots$, find a w that fits the data well
- **Mean Squared Error (Loss Function):**
  $J_{MSE}(w) = \frac{1}{2m} \sum_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)})^2$
  1. m is the number of training examples
  2. $h_w(x^{(i)}) == \hat{y}^{(i)}$
  3. $y^{(i)}$ is actual y value for the ith training example
- **Partial Derivative:**
  $\frac{\partial J_{MSE}(w)}{\partial w_1} = \frac{1}{m} \sum_{i=1}^{m} (w_1 x^{(i)} - y^{(i)}) x^{(i)}$ (when we differentiate $J_{MSE}(w)$ wrt. particular weight, i.e. w_1)

4.2 Gradient Descent

- **Gradient Definition:** $\begin{bmatrix} \frac{\partial J(w)}{\partial w_0} \\ \frac{\partial J(w)}{\partial w_1} \\ \ldots \end{bmatrix}$, where each expression is a partial derivative wrt. different weights
- **Gradient Descent Algorithm:**
  1. Start at some w
  2. Pick a nearby w that reduces $J(w)$:
     $w_j \leftarrow w_j - \gamma \frac{\partial J(w_0, w_1, \ldots)}{\partial w_j}$
  3. Repeat until minimum is reached
- **Gradient Descent with 2 parameters:** Always store gradient in intermediate variable, then conduct gradient descent on each weight separately $\rightarrow$ Do not conduct gradient descent on all weights concurrently (or else a new weight will be updating another weight unknowingly)
- **Theorem:** $J_{MSE}(w)$ is convex for linear regression $\rightarrow$ 1 minimum, global minimum only

4.3 Gradient Descent Variants

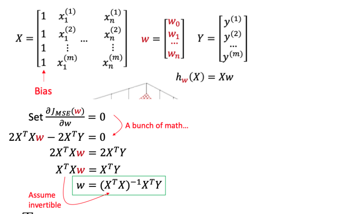
1. **Batch Gradient Descent:** All training examples involved
2. **Mini-batch Gradient Descent:** Subset of training examples at a time; Cheaper (faster); Random, may escape local minima for non-convex function
3. **Stochastic Gradient Descent:** 1 random data point at a time (Order which we use the samples should be randomly decided); Cheapest (fastest); More random, may escape local minima for non-convex function

4.4 Extension of Linear Regression

- **Features of different scales:** A feature with smaller magnitude will take much smaller steps each update than another feature with larger magnitude $\rightarrow$ Converging becomes slower
  1. Solution 1: Have different learning rates for each weight
  2. Solution 2: Conduct mean normalization $x_j \leftarrow \frac{x_j - \mu_j}{\sigma_j}$, where σ_j is the standard deviation of the feature across all training examples & μ_j is the mean of the feature across all training examples
- **Non-linear relationship:** Use polynomial regression for non-linear relationship (transform features) | Terms that are raised to a power more than 1 might need to be scaled as they can become too big | Max degree of polynomial needed to fit any set of n points is $n - 1$ (otherwise will overfit)

4.5 Normal Equation

- **Normal Equation Procedure:**



$$X = \begin{bmatrix} 1 & x_1^{(1)} & \ldots & x_n^{(1)} \\ 1 & x_1^{(2)} & \ldots & x_n^{(2)} \\ 1 & \vdots & & \vdots \\ 1 & x_1^{(m)} & & x_n^{(m)} \end{bmatrix} \quad w = \begin{bmatrix} w_0 \\ w_1 \\ \ldots \\ w_n \end{bmatrix} \quad Y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \ldots \\ y^{(m)} \end{bmatrix}$$

$$h_w(X) = Xw$$

Bias

$$\text{Set } \frac{\partial J_{MSE}(w)}{\partial w} = 0$$

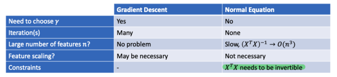
A bunch of math...

$$2X^TXw - 2X^TY = 0$$
$$2X^TXw = 2X^TY$$
$$X^TXw = X^TY$$
$$\boxed{w = (X^TX)^{-1}X^TY}$$

Assume invertible

X^TX becomes non-invertible when it is singular, which occurs when its rows (repeated or identical data points) or columns (redundant or highly correlated features) are linearly dependent

- **Gradient Descent VS Normal Equation:**



	Gradient Descent	Normal Equation
Need to choose γ	Yes	No
Iteration(s)	Many	None
Large number of features n?	No problem	Slow, $(X^TX)^{-1} \rightarrow O(n^3)$
Feature scaling?	May be necessary	Not necessary
Constraints	-	X^TX needs to be invertible

5 Logistic Regression

5.1 Logistic Regression Basics

- Used for classification problems
- **Logistic Regression (1D)**
  - Logistic Function: $\sigma(z) = \frac{1}{1+e^{-z}}$, where $z = wx$
  - Output of $\sigma(z)$ aka. $h_w(x)$ is in [0,1] and treated as a probability $\rightarrow$ If $\sigma(z) > \alpha$, then label as 1
- **Logistic Regression (2D):** Decision boundary is a line of intersection between prediction boundary plane and plane containing all the prediction points $\rightarrow$ Decision boundary is perpendicular to w

5.2 Measuring Fit

- **Why MSE is bad for Logistic Regression:** J_{MSE} for logistic regression would not work well as it is non-linear $\rightarrow$ non-convex $\rightarrow$ multiple local minima
- **Cross Entropy for C classes:** $CE(y, \hat{y}) = \sum_{i=1}^{C} -y_i log(\hat{y}_i)$ (Measures the average number of bits required to identify an event from 1 probability distribution $\rightarrow$ measures the difference between discovered probability distribution of a classification model & predicted values)
- **Binary Cross Entropy for 2 classes:**
  $BCE(y, \hat{y}) = -ylog(\hat{y}) - (1 - y)log(1 - \hat{y})$
  1. Given $y = 1$ & value of $\hat{y}$ is high, model will be rewarded for making a correct prediction
  2. Given $y = 1$ & value of $\hat{y}$ is low, model will be penalized for making a wrong prediction
- **BCE Loss Function:**
  $J_{BCE}(w) = \frac{1}{m} \sum_{i=1}^{m} BCE(y^{(i)}, h_w(x^{(i)}))$, which is convex for logistic regression $\rightarrow$ Can find global minimum during gradient descent

5.3 Gradient Descent

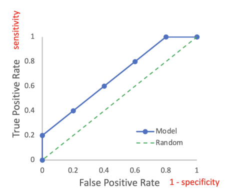
- **Partial Derivative:**
  $\frac{\partial J_{BCE}(w)}{\partial w_0} = \frac{1}{m} \sum_{i=1}^{m} \left(h_w(x^{(i)}) - y^{(i)}\right)$
  $\frac{\partial J_{BCE}(w)}{\partial w_1} = \frac{1}{m} \sum_{i=1}^{m} \left(h_w(x^{(i)}) - y^{(i)}\right) x_1^{(i)}$

5.4 Multi-Class Classification

- Split the multi-class dataset into multiple binary classification problems $\rightarrow$ Binary classifier is then trained on each binary classification problem & predictions are made using the model that is the most confident
- **One VS All**: Fit 1 classifier per class, fit against all other classes $\rightarrow$ Pick the highest probability (Eg. 3 classes of cat, dog, rabbit $\rightarrow$ Classifier of cat against all other classes gives the highest probability $\rightarrow$ Predict as cat)
- **One VS One**: Fit 1 classifier per class pair $\rightarrow$ Pick the most wins (Eg. Cat wins Dog, Rabbit wins Cat, Rabbit wins Dog $\rightarrow$ Predict as Rabbit)

5.5 Performance Measure

- **True Positive Rate & False Positive Rate:**
  $TPR = \frac{TP}{TP+FN}$, $FPR = \frac{FP}{FP+TN}$
- **Receiver Operator Characteristics (ROC) Curve:**
  - Graphical plot that illustrates performance of a binary classifier
  - Different data plots are for different α threshold values
  - If ROC curve is above diagonal random line $\rightarrow$ Model is more accurate than random chance
  - Area under curve (AUC) of ROC: AUC $>$ 0.5 means model is better than chance, AUC $\approx$ 1 means model is very accurate, AUC = 0 means model is predicting opposite class for each data



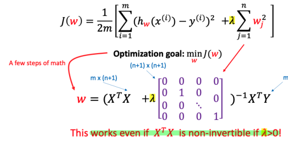
5.6 Model Evaluation & Selection, Hyper-parameter Tuning

- **Model Selection (based on error function):**
  1. Train each model h_i on training set D_{train}
  2. Run the models on validation set D_{val} $\rightarrow$ Choose model with MINIMUM $J_{D_{val}}$
  3. Test that model on a test set D_{test} & assess the model by computing $J_{D_{test}}$
- **Model Evaluation:** When comparing a logistic regression model's output with target label, use Accuracy or AUC-ROC, but not loss functions such as MSE, MAE, BCE
- **Bias:** Error introduced by approximating a real-world problem (which may be very complex) with a simplified model $\longrightarrow$ High bias is associated with Under-fitting & perform poorly on both training and test data, Bias does not improve with an increased number of data points
- **Variance:** Error introduced by the model's sensitivity to small fluctuations in the training data $\longrightarrow$ High variance is associated with Over-fitting & Perform well on the training data but poorly on test data, Obtaining more data points is likely to decrease variance
- **Hyperparameter Tuning:** To find the best model by looping through:
  1. Pick hyperparameters (Eg. Learning rate of gradient descent, max depth of decision tree, min sample of decision tree, loss function of regression model)
  2. Train model with hyperparameters
  3. Evaluate model

6 Support Vector Machine

6.1 Regularisation

- **Methods to address overfitting**
  1. Reduce number of features from high degree to low degree polynomial
  2. Regularisation: Keep all features but reduce magnitude of w_j $\longrightarrow$ Done by adding the weights to cost function, but since we do not know which w_j to penalise, we penalise all w_j instead
- **Linear Regression with Regularisation**
  - Cost Function: $J(w) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_w(x^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$
  - Gradient Descent: $w_j \leftarrow w_j - \gamma\frac{1}{m}\sum_{i=1}^{m}\left(h_w(x^{(i)}) - y^{(i)}\right)x_j^{(i)} - \gamma\frac{\lambda}{m}w_j$
  - When $\lambda = 0$, there is no regularisation, overfitting occurs
  - When $\lambda =$ high value, there is too much regularisation, under fitting occurs (line equation reduces to just w_0)
- **Linear Regression with Regularisation (Normal Equation)**

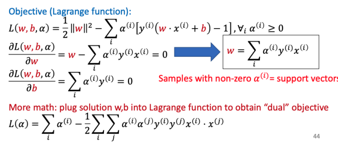


$J(w) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n} w_j^2\right]$
Optimization goal: $\min_w J(w)$
A few steps of math
(n+1) x (n+1)
m x (n+1)
m
$w = (X^TX + \lambda\begin{bmatrix}0&0&0&0\\0&1&0&0\\0&0&\ddots&0\\0&0&0&1\end{bmatrix})^{-1}X^TY$
This works even if X^TX is non-invertible if λ>0!

  - Modify the regularisation matrix such that the 1st element (corresponding to bias term) is 0 instead of 1 $\longrightarrow$ 1st row and column correspond to bias term w_0 and are set to 0 to not penalise the bias term
- **Logistic Regression with Regularisation**
  - Cost Function: $J(w) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h_w(x^{(i)}) + \left(1 - y^{(i)}\right)\log\left(1 - h_w(x^{(i)})\right)] + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$
  - Gradient Descent: $w_j \leftarrow w_j - \gamma\frac{1}{m}\sum_{i=1}^{m}\left(h_w(x^{(i)}) - y^{(i)}\right)x_j^{(i)} - \gamma\frac{\lambda}{m}w_j$

6.2 Support Vector Machine

- **Background:** We want to maximise the margin between all positive & negative points
- **Procedure (Hard Margin SVM):**
  1. Define decision rule (what is positive point, what is negative point): $\vec{w}\cdot\vec{x} + b \geq 0$ then positive
  2. Equation of margin: $margin = \frac{2}{||w||}$
  3. Constrained Optimization Problem: $\min\frac{1}{2}\|\mathbf{w}\|^2$ s.t. $y^{(i)}\left(\mathbf{w}\cdot\mathbf{x}^{(i)} + b\right) - 1 \geq 0$ or $\max_{\alpha\geq 0}\sum_i\alpha^{(i)} - \frac{1}{2}\sum_i\sum_j\alpha^{(i)}\alpha^{(j)}y^{(i)}y^{(j)}\left(\mathbf{x}^{(i)}\cdot\mathbf{x}^{(j)}\right)$
  4. Parameterization of decision boundary: Form system of equations using constraints and equation for the weights ($\sum_{i=1}^{n}\alpha^{(i)}y^{(i)} = 0$ and $w = \sum_{i=1}^{n}\alpha^{(i)}y^{(i)}x^{(i)}$) $\rightarrow$ Obtain values of $\alpha^{(i)}$ with GJE



Objective (Lagrange function):

$L(w,b,\alpha) = \frac{1}{2}\|w\|^2 - \sum_i\alpha^{(i)}[y^{(i)}(w\cdot x^{(i)} + b) - 1], \forall_i\,\alpha^{(i)} \geq 0$

$\frac{\partial L(w,b,\alpha)}{\partial w} = w - \sum_i\alpha^{(i)}y^{(i)}x^{(i)} = 0 \Rightarrow \boxed{w = \sum_i\alpha^{(i)}y^{(i)}x^{(i)}}$

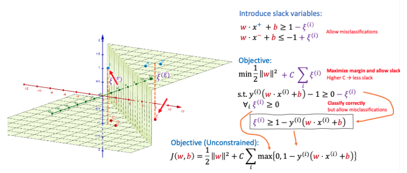
$\frac{\partial L(w,b,\alpha)}{\partial b} = \sum_i\alpha^{(i)}y^{(i)} = 0$ Samples with non-zero $\alpha^{(i)}$= support vectors

More math: plug solution w,b into Lagrange function to obtain "dual" objective

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

44

- **Soft Margin SVM**



Introduce slack variables:
$w\cdot x^+ + b \geq 1 - \xi^{(i)}$
$w\cdot x^- + b \leq -1 + \xi^{(i)}$
Allow misclassifications
Objective:
$\min\frac{1}{2}\|w\|^2 + C\sum_i\xi^{(i)}$
Maximize margin and allow slack
Higher C → less slack
s.t. $y^{(i)}(w\cdot x^{(i)} + b) - 1 \geq 0 - \xi^{(i)}$
$\forall_i\,\xi^{(i)} \geq 0$
Classify correctly but allow misclassifications
$\xi^{(i)} \geq 1 - y^{(i)}(w\cdot x^{(i)} + b)$
Objective (Unconstrained):
$J(w,b) = \frac{1}{2}\|w\|^2 + C\sum_i\max\{0, 1 - y^{(i)}(w\cdot x^{(i)} + b)\}$

  - Soft margin SVM is used when data is not linearly separable (allows for some misclassification of training data)
  - Slack variable is introduced to handle instances where data points are either on wrong side of margin or even misclassified
    1. $\xi_i = 0$ means data point lies either on or outside margin & is correctly classified
    2. $0 < \xi_i \leq 1$ means data point is within margin, but still correctly classified
    3. $\xi_i > 1$ means data point is on wrong side of decision boundary (misclassified)
  - C is regularisation parameter that controls the trade-off between maximising the margin and minimising classification errors $\longrightarrow$ Larger C gives more importance to minimising slack variables (fewer misclassifications are allowed), while smaller C allows for more margin violations
- **Kernel Methods & Kernel Tricks**
  - Used when data is truly not linearly separable
  - Use a kernel to transform data into higher dimensional space for non-linearly separable data

7 Introduction to Neural Networks

7.1 Perceptron

- **Formula:** $\hat{y} = h_w(x) = g(\sum_{i=0}^{n} w_ix_i)$, where g is the activation function & g is normally the sign function ($g(z) = +1 if z \geq 0, else - 1$) for perceptrons
- **Perceptron Learning Algorithm:**
  1. Initialise $\forall_i w_i$ (set random initial weights)
  2. Loop (until convergence aka. no misclassification or max steps reached)
     - For each instance $(x^{(i)}, y^{(i)})$, classify $\hat{y}^{(i)} = h_w(x^{(i)})$
     - Select 1 misclassified instance $(x^{(j)}, y^{(j)})$
     - Update weights: $w \leftarrow w + \gamma(y^{(j)} - \hat{y}^{(j)})x^{(j)}$ (remember to add bias term for x)

7.2 Neural Networks

- **Single Layer Neural Network**
  - Same like perceptron, except g is normally non-linear (Eg. σ, tanh, ReLU, Leaky ReLU)
  - NOT, AND, OR, NOR functions can be modelled by single layer, but XOR, XNOR require multi-layer (data is not linearly seprable)
- **Matrix multiplication in Neural Network:** $\hat{y} = g(W^Tx)$ in a single layer, where # of rows in W == # of inputs per perceptron, # of columns in W == # of perceptrons in layer
- **Forward Propagation across multi-layer:** Basically repeated matrix multiplication from right to left
- **Regression & Classification**
  - Activation function of perceptron(s) in final layer determines if we are doing regression (linear/no activation), binary classification (sigmoid) or multi-class classification (softmax)
  - Softmax activation: $g(z) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$, where sum of outputs of all the final layer perceptrons is 1

7.3 Gradient Descent with Neural Network

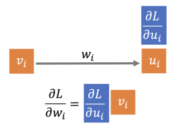
- **Derivative of Sigmoid function:** $\sigma(x) = \frac{1}{1+e^{-x}}$, $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- **Gradient Descent on Single layer Neural Network:** Given $a = \sum_{i=0}^{n} w_ix_i, \hat{y} = \sigma(a), L = \frac{1}{2}(\hat{y} - y)^2$ (MSE), Weight update is $w_i \leftarrow w_i - \gamma\frac{dL}{dw_i}$ == $w_i \leftarrow w_i - \gamma(\hat{y} - y)\hat{y}(1 - \hat{y})x_i$ (Derived with chain rule)

7.4 Neural Network VS Other models

1. **Logistic Regression:** Has Linear, Non-robust decision boundary (prone to misclassification since decision boundary can be too close to data points)
2. **Logistic Regression with Feature Mapping:** Has Non-linear, Non-robust decision boundary (prone to misclassification since decision boundary can be too close to data points)
3. **Support Vector Machine:** Has Non-linear, Robust decision boundary (decision boundary is guaranteed to be far from data points)
4. **Neural Networks:** Has Non-linear, non-robust decision boundary (prone to misclassification since decision boundary can be too close to data points)

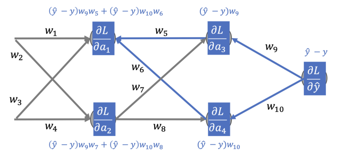
7.5 Back Propagation

- **Chain Rule Background:**
  1. 1 intermediate variable: Given $a = f(x), z = g(a)$, then $\frac{dz}{dx} = \frac{dz}{da}\frac{da}{dx}$
  2. 2 intermediate variables: Given $a = f(x), b = g(x), z = h(a, b)$, then $\frac{dz}{dx} = \frac{dz}{da}\frac{da}{dx} + \frac{dz}{db}\frac{db}{dx}$
- **Back Propagation with Linear Activation:**
  - Given linear activation $g(x) = x$ and loss function $L = \frac{1}{2}(\hat{y} - y)^2$
  - Do 1 round of forward propagation first, get the all the nodes' values (v_i, u_i) first
  - Do 1 round of back propagation next (RIGHT TO LEFT), calculating all the $\frac{dL}{da_i}$ then $\frac{dL}{dw_i}$ and then updating the weights via $w_i \leftarrow w_i - \gamma\frac{dL}{dw_i}$



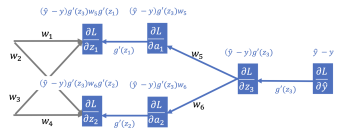
$\frac{\partial L}{\partial u_i}$
v_i
w_i
u_i
$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial u_i} v_i$

  - $\frac{dL}{dw_i} = \frac{dL}{du_i}\frac{du_i}{dw_i} = \frac{dL}{du_i}v_i$



$(\hat{y} - y)w_9w_5 + (\hat{y} - y)w_{10}w_6$
$(\hat{y} - y)w_9$
w_1
w_2
w_3
w_4
$\frac{\partial L}{\partial a_1}$
w_5
$\frac{\partial L}{\partial a_3}$
w_6
w_7
w_9
$\hat{y} - y$
$\frac{\partial L}{\partial \hat{y}}$
w_{10}
$\frac{\partial L}{\partial a_2}$
w_8
$\frac{\partial L}{\partial a_4}$
$(\hat{y} - y)w_9w_7 + (\hat{y} - y)w_{10}w_8$
$(\hat{y} - y)w_{10}$

- **Back Propagation with Non-linear Activation:**
  - Given non-linear activation $g(x)$ and loss function $L = \frac{1}{2}(\hat{y} - y)^2$
  - Same as linear activation, except need to account for $g'(z_i)$



$(\hat{y} - y)g'(z_3)w_5g'(z_1)$
$(\hat{y} - y)g'(z_3)w_5$
w_1
w_2
$\frac{\partial L}{\partial z_1}$
$g'(z_1)$
$\frac{\partial L}{\partial a_1}$
w_5
$(\hat{y} - y)g'(z_3)$
$\hat{y} - y$
$\frac{\partial L}{\partial z_3}$
$g'(z_3)$
$\frac{\partial L}{\partial \hat{y}}$
$(\hat{y} - y)g'(z_3)w_6g'(z_2)$
$(\hat{y} - y)g'(z_3)w_6$
w_3
w_4
$\frac{\partial L}{\partial z_2}$
$g'(z_2)$
$\frac{\partial L}{\partial a_2}$
w_6

8 Convolution Neural Network

8.1 Purpose

- Automatically & efficiently learn features from data while maintaining spatial relationships between pixels
- Does not make sense to have $800 \times 800 \times 100$ = 64 million weights for a neural network with 100 neurons for a 800×800 input pixel image $\longrightarrow$ Use CNN instead

8.2 Convolution:2D

- Given image input X and kernel W $\longrightarrow$ Multiply sliding input window with kernel then sum $\longrightarrow$ Output a feature map
- If we want to detect 1 more feature, we use another kernel and form another feature map

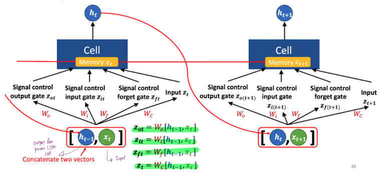
8.3 Convolution Tricks

- **Add Padding:** Add padding of 0s to image input $\longrightarrow$ Resulting feature map can be same size as original image input $\longrightarrow$ Without padding, size of feature maps would shrink with each layer, which could lead to loss of important information, especially near the edges of the image
- **Add Strides:** Increase stride length, so that we slide the input window multiple pixels over a single step $\longrightarrow$ Reduces the size of feature maps (manage computational costs) $\longrightarrow$ Larger stride focuses on more global patterns rather than small, localized details (reduces risk of overfitting)
- **Formula to get dimension of feature map:** Output Dimension = [(Input Dimension - Kernel Dimension + 2 * Padding) / Stride] + 1
- **Pooling layer:** Down-samples feature maps via Max pool, Average pool, Sum pool
- **Receptive Field:** Area of the input that contributes the output of a neuron in a convolutional layer (gives us an idea of where we are getting our results from as data flows through the layers of the network) $\longrightarrow$ Larger receptive field enables network to capture more global features
  1. Let $r_0 = 1, j_0 = 1$
  2. $r_i = r_{i-1} + (kernel_i - 1) \times j_{i-1}$
  3. $j_i = j_{i-1} \times stride_i$

9 Neural Network on Sequential Data

9.1 Recurrent Neural Network (RNN)

- **Sequential Data:** Data where order of elements is essential for capturing patterns & relationships $\longrightarrow$ Each data element depends on previous elements in the sequence (Eg. text, audio, video)
- **One-hot Encoding:** Transforms each element in the sequence into unique binary vector, where only position representing element is set to 1 & all other positions set to 0
- **RNN:**
  - x_i are inputs, h_i are hidden states, $\hat{y}_i$ are outputs, h_0 is set to 0
  - $h_t = g^{[h]}\left(\left(W^{[xh]}\right)^\top x_t + \left(W^{[hh]}\right)^\top h_{t-1}\right)$
  - $\hat{y}_t = g^{[y]}\left(\left(W^{[hy]}\right)^\top h_t\right)$
- **Deep RNN:** RNN with multiple layers
- **Bidirectional RNN:** 2 unidirectional RNN layers chained together in opposite directions & acting on same input $\longrightarrow$ Concatenate together corresponding outputs of 2 underlying unidirectional RNN layers
- **Long Short Term Memory (LSTM)**
  - Specialised RNN designed to overcome limitations of traditional RNNs, especially their inability to capture long-range dependencies effectively
  - Key Components of LSTM: (1): Cell State (memory pipeline that runs through entire LSTM), (2): Forget Gate (decides what information from cell state should be discarded), (3): Input Gate (determines what new info should be added to cell state), (4): Output Gate (decides what next hidden state should be)
  - Procedure in LSTM cell:
    1. Given $g^{[1]}$ is sigmoid activation function, $g^{[2]}$ is tanh activation function, z is input, z_i is input gate weight, z_f is forget gate weight, z_o is output gate weight
    2. At input gate: Obtain $a = g^{[1]}(z_i) \times g^{[2]}(z)$
    3. At forget gate: Obtain $b = c \times g^{[1]}(z_f)$, where c is from memory state
    4. Update memory state $c \leftarrow a + b$
    5. At output gate: Obtain $h = g^{[1]}(z_o) \times g^{[2]}(c)$, where h is the next hidden state
  - LSTM over multiple cells: Previous hidden state h_{t-1} || Current input x_t $\longrightarrow$ Multiplied with W_i to produce input z_t, and gate weights

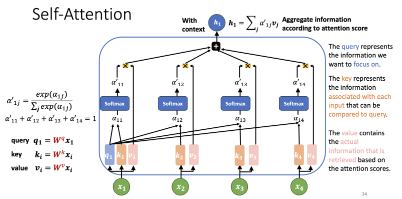


h_t
h_{t+1}
Cell
Memory c_t
Cell
Memory c_{t+1}
Signal control output gate z_{ot}
Signal control input gate z_{it}
Signal control forget gate z_{ft}
Input z_t
Signal control output gate $z_{o(t+1)}$
input gate $z_{i(t+1)}$
Signal control forget gate $z_{f(t+1)}$
Input z_{t+1}
W_o
W_i
W_f
W_c
$[h_{t-1}, x_t]$
$[h_t, x_{t+1}]$
$z_{ot} = W_o[h_{t-1}, x_t]$
$z_{it} = W_i[h_{t-1}, x_t]$
$z_{ft} = W_f[h_{t-1}, x_t]$
$z_t = W_c[h_{t-1}, x_t]$
Concatenate two vectors
20

- **Types of RNN:** (1): One-to-many (Image captioning), (2): Many-to-one (Sentiment analysis), (3): Many-to-many (Video classification with a label for each frame), (4): Encoder-decoder (Language Translation)

9.2 Self-attention

- **Motivation:** Unlike RNN which needs to wait for previous time steps, Self-attention model can capture context information without waiting for previous time steps
- **Procedure:** To calculate h_i



Self-Attention
With context
$h_1 = \sum_j \alpha'_{1j}v_j$
Aggregate information according to attention score
$\alpha'_{1j} = \frac{exp(\alpha_{1j})}{\sum_j exp(\alpha_{1j})}$
$\alpha'_{11} + \alpha'_{12} + \alpha'_{13} + \alpha'_{14} = 1$
query $q_1 = W^qx_1$
key $k_i = W^kx_i$
value $v_i = W^vx_i$
Softmax
The query represents the information we want to focus on.
The key represents the information associated with each input that can be compared to query.
The value contains the actual information that is retrieved based on the attention scores.
34

1. Multiply q_i (i = 1 in this case) with all k_i $\longrightarrow$ Obtain α_{ij}
2. Apply soft-max function to all α_{ij} and multiply resulting α'_{ij} with v_j
3. Sum up all $\alpha'_{ij}v_j$ to obtain h_i
4. Repeat 1-3 to obtain other h_i values

9.3 Issues with Deep Learning

1. **Overfitting:** Addressed by dropout (randomly set some activations to 0 during neural network training, not the weights), early stopping (so that $J_{val}(w)$ does not increase)
2. **Vanishing/Exploding Gradient:** Vanishing (small gradients multiplied again and again until it reaches almost 0), Exploding (Large gradients multiplied again and again until it overflows) $\longrightarrow$ Addressed by non-saturating activation functions (ReLU) or Gradient Clipping (clip gradient within $[min, max]$)

10 Unsupervised Learning

10.1 Supervised Learning VS Unsupervised Learning

- **Supervised Learning:** Given a set of m input-output pairs (training samples), learn to make a prediction
- **Unsupervised Learning:** Learning happens by experiencing data $\longrightarrow$ Given a set of m data points, learn patterns in data

10.2 K-means Clustering

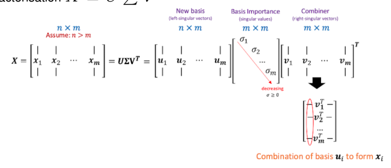
- **Centroid:** Let points $x^{(i)}$ for $i = 1, \ldots, m_1$ be assigned to cluster 1 $\longrightarrow$ For these points, Cluster Centroid is $\mu_1 = \frac{1}{m_1}\sum_{i=1}^{m_1} x^{(i)}$ ($\mu_1, x^{(i)}$ are vectors)
- **K-means Algorithm (m data points, K centroids):**
  1. Randomly initialise K centroids $\mu_1, \ldots, \mu_K$
  2. Repeat until convergence:
     - For $i = 1, \ldots, m$: $c^{(i)} \leftarrow$ Index of cluster centroid $(\mu_1, \ldots, \mu_K)$ closest to $x^{(i)}$ (Assign data points to centroid)
     - For $k = 1, \ldots, K : \mu_k \leftarrow$ Centroid of data points $x^{(i)}$ assigned to cluster k (Reposition centroid based on data points)
  3. No more changes in centroid positions $\longrightarrow$ Converged (Can be shown that each step in algorithm never increases loss function)
- **Local Optima:** Possible for K-means to reach a local optimum where centroids do not move anymore, but loss might be high
- **K-means Loss Function:** Average distance of each sample to its centroid
- **K-Medoids:** Pick K initial centroids randomly from points in data $\longrightarrow$ Pick data points that is closest to centroids & use them as centroids
- **Notes about K-means:**
  1. Centroids are usually not located at 1 of the data points
  2. K parameter is chosen by the user
  3. Multiple restarts of the K-means give different solutions

10.3 Hierarchical Clustering

- **Purpose:** In a situation where we cannot decide on a fixed number of clusters, we want a hierarchy of clusters
- **Algorithm:**
  1. Every data point is in a cluster
  2. Loop (until all points are in 1 cluster): Find a pair of cluster that "is nearest", merge them together (NOTE: We have $N = 2^n$ data points $\longrightarrow$ Hierarchical Clustering algorithm produces a sequence of clusterings $\longrightarrow$ Number of clusters in sequence is $N, N - 1, \ldots, 1$ (Basically merge 2 clusters at 1 single time only))
- **Variants (Pick the minimum)**
  1. Single Linkage: Distance between closest elements in 2 clusters
  2. Complete Linkage: Distance between furthest elements in 2 clusters
  3. Average Linkage: Average of distances of all pairs in 2 clusters
  4. Centroid Method: Distance between centroids of 2 clusters
- **Analysis:** It doesn't require # of clusters to be pre-defined, handles noise well, works well with non-linearly separable data, can be computationally expensive for large data sets ($O(n^3)$ to merge every cluster, where $O(n^2)$ to calculate distance for every pair & $O(n)$ to find nearest pair)

10.4 Dimensionality Reduction

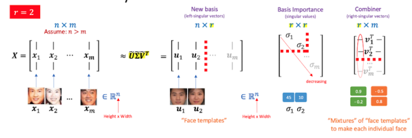
- **Overview:** Many ML problems have data with high dimensional features $\longrightarrow$ Number of samples to learn a hypothesis class increases exponentially with the number of features
  1. Reduction: Change basis of vector remove dependence between components, Remove non-important components that do not cause variations in data
  2. Reconstruction: Add non-important components back, Restore basis
- **Singular Value Decomposition (SVD):** Take without loss of generality $n > m$, for any $n \times m$ rectangular real-valued matrix X, there exists a factorisation $X = U\sum V^T$



New basis (left-singular vectors)
Basis Importance (singular values)
Combiner (right-singular vectors)
$n \times m$
Assume: $n > m$
$n \times m$
$m \times m$
$m \times m$
$X = [x_1\ x_2\ \cdots\ x_m] = U\Sigma V^T = [u_1\ u_2\ \cdots\ u_m]\,\mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_m)\,[v_1\ v_2\ \cdots\ v_m]^T$
decreasing $\sigma \geq 0$
v_1^T
v_2^T
v_m^T
Combination of basis u_i to form x_i

1. U is $n \times m$ and has m orthonormal columns
2. $\sum$ is $m \times m$, diagonal with $\sigma_j \geq 0$ ordered from largest to smallest
3. V is $m \times m$ and has m orthonormal columns and rows

- **Dimensionality Reduction via SVD:** Instead of m singular values, we set all singular values except the first r to 0 $\longrightarrow$ Resulting matrix is the "best" approximation with new basis size of r to the original matrix Transforms from $n \times m = n \times m, m \times m, m \times m$ to $n \times m = n \times r, r \times r, r \times m$, where $r < m$



$r = 2$
$n \times m$
Assume: $n > m$
New basis (left-singular vectors)
$n \times r$
Basis Importance (singular values)
$r \times r$
Combiner (right-singular vectors)
$r \times m$
$X = [x_1\ x_2\ \cdots\ x_m] \approx \tilde{U}\tilde{\Sigma}\tilde{V}^T = [u_1\ u_2\ \cdots\ u_m]$
$\in \mathbb{R}^n$
Height x Width
"Face templates"
decreasing
45
10
$\sigma_1\ \sigma_2$
0.9
-0.5
-0.2
0.8
"Mixtures" of "face templates" to make each individual face

- **Reduction & Reconstruction via SVD:**
  1. Data Matrix ($n \times m$): $X = U\sum V^T \approx \tilde{U}\tilde{\sum}\tilde{V}^T$
  2. Reduced Data ($r \times m$): $Z = \tilde{U}^TX$
  3. Reconstructed Data ($n \times m$): $\tilde{X} = \tilde{U}Z$
- **r-SVD**: defines an encoder-decoder structure for reduction and reconstruction of high-dimensional data
- **Principal Component Analysis (PCA):** Captures components that max the statistical variations of the data
  1. Given the data matrix $X = (x^{(1)}, \ldots, x^{(m)})$, where $x^{(i)}$ is the i^{th} row of X
  2. Compute mean over samples: $\overline{x} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$
  3. Compute mean-centred data: $\hat{x}^{(i)} = x^{(i)} - \overline{x}$
  4. Define data matrix $\widehat{X} = (\hat{x}^{(1)}, \ldots, \hat{x}^{(m)})$
  5. Create covariance matrix of data: $Cov(X) = \frac{1}{m}\widehat{X}\widehat{X}^T$
  6. Compute SVD on $Cov(X)$ to obtain the U matrix
  7. Reduce to r components to obtain $\widetilde{U}$ (choose minimum r s.t. $\frac{\sum_{i=1}^{r}\sigma_i^2}{\sum_{i=1}^{m}\sigma_i^2} \geq 0.99$, where $\geq$ 99% of variance in data is retained)