CS2109S Finals

by Zong Xun

Introduction to AI

We specify a task environment with the **P.E.A.S** framework. **Rational agents** aim to maximize expected performance regardless of environment, whereas **Omniscience** maximizes actual performance.

Nature of Environments

- Fully observable vs Partially Observable
- Single-agent vs Multi-agent
- Deterministic vs Strategic vs Stochastic
- Episodic vs Sequential
- Static vs Dynamic
- Discrete vs Continuous

Notes

1. Agent program takes in **current percept**, whereas the agent function takes in **entire percept history**.
2. Not all state abstractions can be mapped back to the real world i.e inaccurate representation.
3. All agents are rational if permutations of actions are reward-invariant i.e. random selection
4. Performance measure should indicate **correctness** and **efficiency**. It is generally hard to define, and should focus on what one wants to achieve in the environment.

Solving problems by searching

Types of problems: Single state, sensorless (conformant), contingency, exploration.

A single state search problem can be defined formally as:

- **State space.** Legal states and possible dimensions.
- **Successor function.** Set of actions that can be executed.
- **Transition model.** New state.
- **Action cost.** Cost of applying action a in state s.
- **Goal test.** May be explicit or implicit.

Search Strategies

The order of node expansion. They are evaluated based on:

- **Completeness**: Does it find a solution if it exists?
- **Time Complexity**: Number of nodes generated
- **Space Complexity**: Number of nodes in memory
- **Optimality**: Does it find a least cost solution?

General Tree Search - Best First Search

Pick a node from the **open list** with min value of an evaluation function, *f(n)*. Child nodes are added only if they are not visited or previously visited with a higher path cost. By employing different *f(n)* functions, we get specific algorithms.

Uninformed Search

An uninformed search algorithm is given no clue how close a state is to the goal(s).

- **Breath first search**. Use early goal test for optimization. Complete if finite branching factor. Not optimal if nodes on the same level have different "costs" (2008 MT)
  - Optimal if all nodes on **same level** have same cost. **Intuition**: all the nodes at depth d is generated before nodes at depth d + 1.
- **Uniform cost search** (Best first search with f(n) = path cost) - Special case of A*, $h(n) = 0 \implies$ always Optimal
- **Depth first search** - Special case of best first search. Optimal if all solutions have the same depth. Complete if there exists some time/cost limit.
- **Depth limited search** (DFS with depth limit, *l*)

$$N_{DLS} = b^0 + b^1 + \cdots + b^d$$

- **Iterative deepening search** (Slightly more overhead compared to DLS, but with DFS memory)

$$N_{IDS} = (d+1)b^0 + db^1 + \cdots + 2b^{d-1} + b^d$$

of depth d and branching factor, b.

Bidirectional search

Intuition: $2O(b^{\frac{d}{2}})$ is smaller than $O(b^d)$. Different search strategies can be employed for either half. However, *pred(succ(n))* and *succ(pred(n))* must be equal. Optimal if the evaluation function is the path cost.

Resolving redundant paths

(1) Memoize (2) Better heuristic (3) Pruning

Informed Search

Informed search strategy uses domain-specific hints about the location of goals to find solutions more efficiently. These "hints" are known as heuristics.

Greedy Best First Search

Uses the function $f(n) = h(n)$. Expand the first node that appears to be the closest to goal. Tree: not complete, Graph: Complete only if finite search spaces.

A* Search

Uses the function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost so far to reach n. Assuming all action costs are greater than $\epsilon > 0$ and state space is finite, A* search is complete. If there are infinitely many nodes with $f(n) \leq f(goal)$, then it is not complete. Whether it is **cost-optimal** depends on:

- **Admissibility.** $h(n) \leq h^*(n)$. If $h(n)$ is admissible, then A* using TREE-SEARCH is optimal.
- **Consistency.** $h(n) \leq c(n, a, n') + h(n')$ i.e. the triangle inequality holds. If h(n) is consistent, A* using GRAPH-SEARCH is optimal.

$$Consistent(h_1) \Rightarrow f(n') \geq f(n)$$

  This means that f(n) is **non-decreasing** along any path.
- **Dominance.** h_1 if $h_2(n) \geq h_1(n)$ for all $n \Rightarrow h_2$ is better for search. Intuition: "Larger" steps towards the goal state, visiting fewer nodes, $\implies$ faster and quicker.

Notes on A*

1. Admissibility is a sufficient condition for optimality, but not necessary. If $h(n)$ overestimates by a constant factor, A* search is still optimal.
2. If the optimal solution has cost $C*$ and the second-best has cost C_1 and if h(n) overestimates some costs, but never by more than $C* - C_1$ then A* is guaranteed to return cost-optimal solution.
3. All nodes with $f(n) < C*$ is expanded. No nodes with $f(n) > C*$ is expanded. A* is efficient because it prunes away nodes that are not necessary for finding an optimal solution.
4. If $h(n) = -2g(n)$, then $f(n) = -g(n)$ which means $A* = DFS \implies$ not complete.

Memory Bounded Algorithms

The main issue with A* is its use of memory.

- **IDA***. Use f-cost for cutoff. Min f-cost among unexplored nodes is used as the next cutoff.
- **Recursive Best-First Search**. Tracks f-value of the best alternative path. If the current node exceeds this value, the recursion unwinds back to the alternative path. Optimal if h(n) is admissible. Space complexity is $O(d)$.

  Complain: Both IDA* and RBFS use too little memory.
- **Simplified MA* (SMA*).** Expand newest best leaf until memory is full. Then drop the oldest node with the worst f-value and backup the forgotten node's value to its parent. SMA* will only regenerate the forgotten sub-tree when all other paths have been shown to look worse.

Heuristics

- **Effective branching factor,** b. The smaller b is, the better.
- **Effective depth,** d. The smaller d is, the better.
- The cost of an **optimal solution** to a **relaxed problem** is an **admissible heuristic** for the original problem.

Local Search Algorithms

Find best state according to **objective function**.

Hill Climbing Search Find local maxima by travelling to neighbouring states with steepest ascent. It can get stuck on **local maxima** and **plateaus**. Solutions are:

- **K-Sideways Move**.
- **Stochastic**. Chooses a random uphill move.
- **Random-Neighbours**. Useful when a state has thousands of successors. May escape shoulders, but little chance of escaping local optima.
- **Random-Restart**. Randomly generated initial steps, then restart if no maximum found. Can escape shoulders, and high chance of escaping local optima.

Simulated Annealing Escape local maxima by allowing some bad moves, but gradually reduce their frequency.

Beam Search Perform k hill-climbing searches in parallel with information sharing, abandoning unfruitful searches. The k states may end up clustering, making it k-times slower. **Stochastic beam search** alleviate this problem by choosing successors with probability proportional to their value.

Genetic algorithms Successor state generated by combining two parent states. (1) selection (who gets to be parent), (2) crossover point (recombination procedure), (3) mutation (randomly flip bits).

Formulating local search problems

- Define an **objective function** (e.g. min-conflicts)
  - can be a linear combination of constraints
- Define an initial candidate solution (greedily/randomized)
- Define transition function (e.g. swapping conflicts)
- Goal test (e.g. total conflicts = 0)

Notes on Local vs Informed

- Informed search is preferred if the search space is small, and there is **always** a solution.
- Local search is preferred in constraint satisfaction problems, and that **path to goal does not matter**.

Adversarial Search

Adversarial search problems arise from competitive environments wherein two or more agents have conflicting goals, such as in games.

Minimax Search Algorithm

Depth first exploration. Includes the state of the game, and the player's turn.

$$\text{MM(s)} = \begin{cases} \text{UTILITY(s, MAX)}, & \text{if IS-TERMINAL(s)} \\ \max_{a \in A(s)} \text{MM}(R(s,a)), & \text{if TM(s)=MAX} \\ \min_{a \in A(s)} \text{MM}(R(s,a)), & \text{if TM(s)=MIN} \end{cases}$$

Alpha-Beta Pruning

Pruning occurs when $\alpha \geq \beta$. **Pruning does not affect the final outcome**. With good move ordering, time complexity can be reduced to $O(b^{m/2})$, allowing us to examine twice as deep given the same computation power. Generally, the "best" nodes should be visited first. We can use **transpositions table** to cache previously seen states.

Making use of heuristics

Computation of the entire search space is not practical. Utility of states is estimated with **evaluation functions**. We can also make use of **cut-off depth** for the terminal test. This allows us to do iterative deepening, where each iteration gives rise to a better move ordering.

Notes on Minimax

1. It is optimal only against an optimal opponent.
2. As long as the **relationship between leaves are retained**, a translation or scale will not affect the final outcome (not true if multiplying by negative value).
3. Assume left-to-right ordering for $\alpha - \beta$ pruning, the leftmost branch is always evaluated.

Machine Learning

Decision Trees

A **decision tree** is a representation of a function that maps a vector of attributes to a single output value.

A **boolean decision tree** is equivalent to

$$output \leftrightarrow (Path_1 \vee Path_2 \vee \dots)$$

where each $Path_i$ is a conjunction of the form $(A_i = a_i \wedge A_n)$ of attribute value tests corresponding to a path from the root to a true leaf. We want to find the most compact decision tree. However, it is computationally intractable to construct all 2^{2^n} distinct decision trees with n Boolean attributes.

There are a total of 3^n distinct conjunctive hypotheses where each attribute in n is either in (positive), in (negative), or out.

A more **expressive hypothesis** space

- increases the chance that target function can be expressed
- increases the number of hypotheses consistent with training set

but may also result in worse predictions, due to "overfitting".

Information Theory

Choose a root that offers the most information gain. Entropy is defined to be the measure of **impurity**. Entropy can be found using the following:

$$I(P(v_1),\dots,P(v_n)) = -\sum_{i=1}^{n} P(v_i) log_2 P(v_i)$$

A chosen attribute, A, splits the training set E into subsets $E_1,\dots,E_v$, where A has v distinct vales.

$$remainder(A) = \sum_{i=1}^{v} \frac{p_i+n_i}{p+n} I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$$

Information gain is given by

$$IG(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - remainder(A)$$

The main intuition is that even partitions offer more information gain than uneven partitions.

Generalisation and Overfitting

Overfitting becomes more likely as the number of attribute grows, and less likely as we increase the number of training examples. We can prune certain nodes in the decision tree by removing those that are clearly irrelevant. This can be done via χ^2 pruning. Another way would be **cut-off sample**.

1. **Continuous Valued Attributes**: partition the values into a discrete set of intervals.
2. **Many values**: Use $GainRatio = \frac{Gain(C,A)}{SplitInInfo(C,A)}$ where $SplitInInfo(C,A) = -\sum_{i=1}^{d} \frac{|E_i|}{|E|} log_2 \frac{|E_i|}{|E|}$.
3. **Attribute with Differing Cost**: Replace `Gain` with $\frac{Gain^2(C,A)}{Cost(A)}$ or $\frac{2^{Gain(C,A)}-1}{(Cost(A)+1)^w}$ where $w \in [0,1]$ determines importance of cost.
4. **Missing values:** Assign most common value with/without same output, or assign each possible value of A some probability, then split according to this.

Notes on Decision Trees

1. Generating a decision tree, t_2 from a training set formed by another decision tree, t_1 will generate a tree that is logically equivalent, not equal! Two attributes might have the same IG, so its 50/50 who gets picked as root.
2. If the training set is small, even a reasonable hypothesis may overfit.
3. The simplest and smallest consistent DT is preferred.
4. Wrong predictions might indicate errors in some of the observations or missing a critical attribute. Another reason is some of the attributes may not be relevant.

Linear Regression and Classification

Types of loss functions

Loss function gives us an indication of how far off our predictions are compared to the actual values.

- Mean Squared Error

$$J(w_o, w_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)})^2$$

- Mean Absolute Error

$$J(w_o, w_1) = \frac{1}{m} \sum_{i=1}^{m} |h_w(x^{(i)}) - y^{(i)}|$$

Univariate Linear Regression

Form of $y = w_1 x + w_0$, where x is the input and y is the output. The loss function is minimized when its partial derivatives w.r.t w_0, w_1 is 0.

Gradient Descent

We can't always find partial derivatives that are zero. Gradient descent allows us to compute an estimate of the gradient at each point and move a small amount in the steepest downhill direction, until we converge on a point with minimum loss. Since the loss surface is convex, we will always arrive at the global minimum.

$$w_0 = w_0 - \alpha \frac{1}{m} \sum_{j=1}^{m} (w_0 + w_1 x_1^{(j)} + \dots - y_j)$$

$$w_i = w_i - \alpha \frac{1}{m} \sum_{j=1}^{m} (w_0 + w_1 x_1^{(j)} + \dots - y_j) \times x_i^{(j)}$$

This is otherwise known as **batch gradient descent**. Stochastic gradient descent updates weights by considering one point at a time, which is faster but does not guarantee global minima due to its randomness.

Multivariate linear regression

Gradient descent does not work well if the features have significantly different scales. We can perform

- **mean normalisation**, $x_i^j = \frac{x_i - \mu_i}{\sigma_i}$
- **min-max normalisation**, $x_i^j = \frac{x_i^j - min(x_i)}{max(x_i) - min(x_i)}$
- **median normalisation**, $x_i^j = \frac{x_i^j - median(x_i)}{IQR(x_i)}$

Without normalization, the model needs to perform large update to one weight compared to the other weight. This can cause the gradient descent trajectory to oscillate back and forth along one dimension, thus taking more steps to reach the minimum.

Normal Equation

Non-iterative approach, that works well when the number of features, n is small.

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

where $X^T X$ is invertible. Suppose not, perform the ERO/ECO and identify the linear dependency. Remove the necessary rows and cols and recompute the result.

Logistic Regression

Suppose we have our classification hypothesis as $h_w = 1$ if $w_0 + w_1 x_1 > 0$ or 0 otherwise. While capable of classification, Linear regression (1) does not generalize well to new data (2) shift in threshold when new data is added.

Since the threshold function is **not continuous and differentiable**, we approximate it with the logistic function.

$$h_w(x) = g(w^T x) \text{ where } g(z) = \frac{1}{1+e^{-z}}$$

where h_w estimates the probability that $y = 1$ on input x.

The *Binary Cross Entropy Loss Function* is given by:

$$-\frac{1}{m} \sum_{i=1}^{m} y^{(i)} log h_w(x^{(i)}) + (1-y^{(i)}) log(1 - h_w(x^{(i)}))$$

Note: MSE cannot be used due to non-convexity. The update function is the **same** as linear regression.

Regularization

Two ways to address overfitting: (1) reduce the number of features (2) reduce magnitude of weights (regularization)

L1-norm, Lasso Regression

$$J(w) = \frac{1}{2m} [\sum_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{i=1}^{n} |w_i|]$$

L2-norm, Ridge Regression

$$J(w) = \frac{1}{2m} [\sum_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{i=1}^{n} w_i^2]$$

The update function (**gradient descent**):

$$w_n = (1 - \frac{a\lambda}{m}) w_n - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)}) \cdot x_n^{(i)}$$

Remove α from λ for logistic regression.

The update function (**normal equation**):

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

This works even if $X^T X$ is non-invertible if $\lambda > 0$.

Note:

1. Too large λ can cause your model to underfit. Regularization assure that your model performs better on unseen data, sacrificing performance on training data. (higher bias but low variance)
2. Lasso shrinks the less important feature's coefficient to zero, removing some feature altogether. Works well for feature selection in case we have a huge number of features.
3. Ridge penalizes larger parameters, attempting to pull all parameters towards small values.

Support Vector Machines

Find a hyperplane in an N-dimensional space that distinctly classifies the data points. We aim to maximize the margin distance to provide some reinforcement that future data points can be classified with more confidence.

$$J(w) = C \sum_{i=1}^{m} y^{(i)} c_1 + (1-y^{(i)}) c_0 + \frac{1}{2} \sum_{i=1}^{n} w_i^2$$

where

$$c_1 = -log(\frac{1}{1+e^{-w^T x}}), c_0 = -log(1 - \frac{1}{1+e^{-w^T x}})$$

This is a constrained optimization problem to minimize the number of misclassifications and maximize correct classifications ("soft" margin).

Properties:

1. SVM is robust to outliers (increase regularization).
2. Support Vectors influence the position and orientation of the hyperplane.
3. SVM is a binary classifier, for n-class problems, n models is required to correctly classify.
4. Increasing the value of C reduces the effect of regularization, allowing the data to fit more (lower bias).

Kernel tricks

Kernels can be thought of as a bijective function which maps the data points to a higher dimensional plane so that the data is linearly separable. **Kernel tricks** allows us to operate in the original feature space without computing the coordinates of the data in a higher dimensional space, which is costly.

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

is known as the **guassian kernel**.

1. Given a data point x, compute new features based on the proximity to landmarks. Features $f_1, f_2, \dots, f_n$ will be ≈ 1 if x is close, else ≈ 0. Replace all x_i's with f_i's.
2. Apply feature scaling.
3. Some kernels may not converge. Need to satisfy Mercer's theorem.
4. $(n >> m)$ SVM with no kernel
   $(n < m)$ SVM with guassian kernel
   $(m >> n)$ SVM with no kernel

Hypothesis Evaluation

Split dataset into **training**, **validation** and **test** set. Train each model on D_{train}, compute $J_{D_{val}}(w)$, pick the model with the lowest $J_{D_{val}}(w)$, and use $J_{D_{test}}(w)$ to estimate the performance on unseen samples.

Bias and Variance Tradeoff

A model with a **high bias** makes more assumptions, and unable to capture the important features of our dataset. A high bias model cannot perform well on new data. To reduce, (1) increase the input features (2) decrease regularization term (3) use more complex models.

A model that shows **high variance** learns a lot and perform well with the training dataset, but does not generalize well with the unseen dataset. To reduce, (1) decrease the input features (2) do not use more complex model (3) increase training data (4) increase regularization term.

Neural networks

Neural networks are capable of learning complex non-linear functions.

Motivation

Regression and SVM can have exploding number of terms. With 100 features, and max degree 3, we have 3(100) + $3\binom{100}{2}$ + $\binom{100}{3}$ = 176, 850 terms.

Perceptron

The perceptron is a **linear classifier**.For each point, the algorithm predicts labels via $\hat{y} = \sigma(w^T x)$, where σ is any **non-linear** activation function.

Perceptron Learning Algorithm

1. Initialize weights, w_i (can be zero or random small values).
2. For each instance i with features $x^{(i)}$, classify $\hat{y}^{(i)} = sgn(w^T x^{(i)})$.
3. Select **one** misclassified instance, and update weights using
$$w \leftarrow w + \alpha(y - \hat{y})x$$
The key intuition is that when the labels are misclassified, we need to update the $cosine$ angle between the weight and x vector.
4. Repeat step 2 and 3 until convergence or maximum number of iterations.

Properties

- Not robust: can select any model to linear, **not deterministic**.
- Cannot converge on non-linearly separable data.

Single Layer Perceptron

After a single forward pass, we need to backpropagate to update the weights for the model to learn. In a single layer perceptron, the weight update is as follows

$$w_i \leftarrow w_i - \alpha \frac{d\epsilon}{dw_i} = w_i - \alpha(\hat{y} - y)g'(f)x_i$$

The above equation is derived using MSE = $\frac{1}{2}(\hat{y} - y)^2$ where n = 1 and chain rule, where $\frac{d\epsilon}{dw_i} = \frac{d\epsilon}{d\hat{y}}\frac{d\hat{y}}{df}\frac{df}{dw_i}$. If the sigmoid activation function is used, replace $g'(f)$ with $\hat{y}(1 - \hat{y})$.

Multi Layer Perceptron

A multi layer perceptron is formed by combining many single layer perceptrons. This allows the model to learn more complex functions.

Notations

- Denote $a_i^{[l]}$ to represent the ith perceptron on layer l.
- Denote $w_{ji}^{[l]}$ to represent the weight of the ith feature pointing to the jth perceptron on layer l. Weights for a single layer is usually represented as a row-major matrix, where each row represents the weights for one perceptron.
- Denote $a^{[l]} = g^{[l]}(f^{[l]})$ and $f^{[l]} = W^{[l]}a^{[l-1]}$ as the layer activations, where g is the activation function.

Layer Activation

With the above notations, we have

$$a^{[l]} = g^{[l]}(W^{[l]}a^{[l-1]})$$

Forward propagation works by applying these notations recursively all the way up to the input layer.
For **backpropagation**, it works similarly as SLP. The idea is to calculate the $\frac{d\epsilon}{dW^{[l]}}$ instead of $\frac{d\epsilon}{dW_i}$. We denote $\hat{y}'(W^{[l]})$ as the gradient relative to $W^{[l]}$. Then,

$$\hat{y}'(W^{[l]}) = \frac{df^{[l]}}{dW^{[l]}}(\delta^{[l]})^T$$

is a recursive notation for backpropagation, where

$$\delta^{[l]} = \frac{dg^{[l]}}{df^{[l]}}\frac{df^{[l+1]}}{dg^{[l]}}\delta^{[l+1]}$$

Simplifying, we get $\delta^{[l]} = g'(f^{[l]}) \circ W^{[l+1]}\delta^{[l+1]}$. Note that this is a V-shaped recursion, the gradient of the weighted input of each layer is calculated from back [$l + 1$] to front [l].
Note: When implementing custom functions/layers, make sure they are **differentiable** (to calculate their gradients).

Performance Measures

Confusion matrix is a useful visualization to measure the accuracy, precision and recall of the model.

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

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

Using precision and recall, we can compute the **F1 score**.

$$F1 = \frac{2TP}{2TP + FN + FP}$$

ROC curve is constructed by plotting the True Positive Rate (TPR) against the False Positive Rate (FPR). AUC ≈ 1 implies that the model is very accurate, there is a clear segregation between the positive and negative samples.
Note: Evaluation metrics explain the performance of a model, and loss functions optimize the performance of a model.

Deep Learning

Convolutional Neural Network

Goal: make use of spatial locality for feature extraction. Each layer of the model learns features in increasing complexity.
Motivation: (1) **leverage on spatial context** to extract local features, (2) enjoy **translation invariance**, features can also be recognized elsewhere, (3) **fewer parameters**, kernels share weights, which means training is faster.
Number of params in a convolutional layer (with **bias**):

$$\#params = (kernel\_size \times |input\_channels| + 1) \times |output\_channels|$$

Layers in a CNN

Three main types: Convolutional, Pooling, Fully Connected.

Convolution

An element-wise multiply-sum operation between an image section (dependent on stride and padding) and the **kernel**. The dimensions of the output **for one kernel** is given by

$$\lfloor \frac{n + 2p - K}{s} \rfloor + 1$$

Note: Kernels are learned automatically through backpropagation. If a kernel is given, be sure to rotate ($180°$) first, to avoid performing 2D correlation instead.

Pooling

Extract the most relevant features, and reduce dimensionality, parameters, and noise from previous convolutional layers.
Note: the output from the final pooling layer is flattened and fed to a MLP that can classify the final pooled feature map into a class label.

Fully connected

There can be multiple fully connected layers. An activation function is used in each fully connected layer.

Softmax Layer

The softmax activation is used for classification tasks. By normalizing the results to follow a probability distribution, it allows us to avoid binary classification and accommodate as many classes as needed.

Recurrent Neural Network

Goal: make use of temporal locality for feature extraction. Recurrent neural network (RNN) take information from prior inputs to influence the current input and output.

Types of RNN

1. *one-to-one:* Each I/O is independent. Not RNN.
2. *one-to-many:* Image captioning, text generation.
3. *many-to-one:* Text classification, sentiment recognition.
4. *many-to-many:* Language translation, music generation.
   - Process all inputs first, and then predict later.
   - Process 1 at a time.

Common problems

- **Overfitting:** (1) Regularization (Dropout - randomly setting activations to 0) (2) Early stopping.
- **Vanishing/Exploding gradient:** With saturating activations (like sigmoid), the gradient decreases exponentially as we propagate down (due to small derivatives). Consequently, change in weights becomes very small, causing convergence to be slow, if not impossible.
  To solve this: (1) Proper weight Initialization (2) Use non-saturating activations e.g. ReLU (Leaky ReLU) (3) Feature scaling (4) Gradient clipping.
  Note: Normal ReLU faces the **dying relu** problem when the majority of the activations are 0.

Unsupervised Learning

Try to derive structure by clustering the **unlabelled** data based on relationships among the variables in the data.

K-Means Algorithm

The algorithm is **not deterministic** due to its random initialization. It does not know what is the "best" solution, simply converging to a *possible* solution. Repeat the algorithm multiple times, and pick the clusters with the minimum cost, or use a **probabilistic method** of choosing the centroids (choose points with probability proportional to square of distance apart to avoid picking centroids close to one another).

The proof of convergence depends on the fact that every iteration produces a partition of lower loss $\Rightarrow$ prove that there is a finite number of such partitions, k^n.

Hierarchical Clustering

Plot a dendrogram. Bad choice due to (1) computational complexity (2) long training times (3) inaccurate results (4) lack of transparency (5) human validation.

1. Every data point is a one-point cluster.
2. Find a pair of points that is "nearest" according to some distance scheme and merge them.
3. Repeat until the data points are in one cluster.

Principal Component Analysis

Motivation: Higher dimensional data is more computationally costly $\implies$ reduce dimensions while maintaining as much information as possible.

Steps:

1. **Calculate the covariance matrix**
$$cov(X) = \frac{1}{m}XX^T$$
The covariance matrix allows us to understand the relationship between each variable $\implies$ identify any redundant variables.
2. **Perform SVD to obtain principal components**
$$U, E, V^T = X$$
where X is a n x m matrix, and m is the # of samples. Principal components represent the directions of the data that explain a maximal amount of variance, a.k.a the lines that capture most information of the data.
3. **Casting the data along the principal component axis**
Use the feature vector formed using the eigenvectors of the covariance matrix, to reorient the data from the original axes to the ones represented by the principal components.
$$Z^{(i)} = U_{reduce}^T X^{(i)}$$

Note: Do not use PCA for overfitting. The reason is that PCA is not aware of the labels in the training set, hence it might just throw away useful information.

Drawbacks:

Lossy compression as we do not regain 100% of the variance (**no redundancy**). For data with less redundancy, the value of k might be too large such that we would actually end requiring more space since decompression matrix U_{red}, $Z_{(i)}$ and the **row means** needs to be stored as well.