# COMP5318 Course Notes

Jazlyn Lin

July, 2021

## Contents

# 1 W1

## 1.1 Data Preprocessing

**Definition 1.1. Normalization and Standardization**

1. Normalization (MinMax-Scaling). Already explained in W2 - 2.2
$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$
2. Standardization
$$x' = \frac{x - \mu(x)}{\sigma(x)}$$

# 2 W2

## 2.1 Nearest Neighbor Algorithms

In order to implement Nearest Neighbor algorithms and say predict label of the new example, we need to first choose a distance measure to measure the "closeness"/distance of the neighbors.

**Definition 2.1. Distance Measures**

When $a_i$ and $b_i$ are **numeric**, to measure the distance $Dist(A, B)$ between $A = (a_1, ..., a_n)$ and $B = (b_1, ..., b_n)$, we have the following distance measures

1. Euclidean Distance (aka $\ell_2$ norm, most frequently used)
$$Dist(A, B) = \|A - B\|_2 = (\sum_{i=1}^{n} |a_i - b_i|^2)^{\frac{1}{2}} = \sqrt{(a_1 - b_1)^2 + ... + (a_n - b_n)^2}$$
2. Manhattan Distance (aka $\ell_1$ norm)
$$Dist(A, B) = \|A - B\|_1 = (\sum_{i=1}^{n} |a_i - b_i|^1)^1 = |a_1 - b_1| + ... + |a_n - b_n|$$
3. $\ell_p$ norm
$$Dist(A, B) = \|A - B\|_p = (\sum_{i=1}^{n} |a_i - b_i|^p)^{\frac{1}{p}}$$

If $a_i$ and $b_i$ are **categorical/nominal**, we can first use 0-1 loss to measure $a_i - b_i$ (e.g. $a_1 \neq b_1$ then $a_i - b_i = 1$ else $a_i - b_i = 0$) and then apply the previous distance measure.

**Example.** .

1. Numeric case. bla
2. Categorical case. Temperature and Play. A = ['high', 'no'], B = ['high', 'yes'] then $a_1 - b_1 = 0$, $a_2 - b_2 = 1$. Use $\ell_2$ norm $D(A, B) = \sqrt{0^2 + 1^2} = 1$

**Definition 2.2. Normalization (i.e. MinMaxScaler) before KNN**

- Definition: Transform **each attribute** by scaling them into a given range, which is by default [0,1]
- Reason: Some attributes are huge and can dominant other smaller attributes. For example, for income and age, income is the dominant attribute. If we don't normalise and scale them appropriately, when apply distance-based ML algorithms such as KNN, the dominant attribute will degrade the algorithms performance.
- Example:
  - age and income. A = [20, 40K], B = [40, 60K]. Without vs with normalisation, $\ell_1$ distance 20020 vs 0.4.
- Relevant algorithms: KNN, SVM, PCA
- Formula:
$$\text{attribute X: } X' = \frac{X - min(X)}{max(X) - min(X)}$$
- Code: MinMaxScaler

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler() # create an object of MinMaxScaler class
scaler.fit(X_train) #grab some min max information of training data
X_train_norm = scaler.transform(X_train)
#IMPORTANT: apply the same normalisation to testing data using training data's min and max
X_test_norm = scaler.transform(X_test)

#After normalisation ,apply KNN
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train_norm, y_train)
print("Accuracy on test set: {:.2f}".format(knn.score(X_test_norm, y_test)))
```

**Definition 2.3. KNN**

1. Procedure:
   (a) Select number of neighbors K
   (b) Select a distance measure, could be $\ell_1$ Manhattan or $\ell_2$ Euclidean
   (c) For each new example, calculate the distance between the new example and all other existing examples.
   (d) Find the K nearest neighbors based on the distance.
   (e) If classification task, apply *majority voting* and assign the most popular labels of K nearest neighbors to the new example. If regression task, assign the *average* target value of K nearest neighbors as the new target value.
2. Hyperparameters
   (a) $K$, number of the neighbors.
   - Rule of thumb: $K \leq \sqrt{m}$ where $m$ is the number of training examples.
   - Commercial case: $K = 10$
   - Larger $K$ makes KNN more robust to noisy examples
3. Complexity
   - Computational Complexity:
     – For inference, $O(mn)$ if $m$ is the number of training examples and $n$ is the example feature dimension.
     – to classify each new example's label, we need to go through all $m$ training examples with $n$ feature dimension
     – less complexity if more efficient data structure such as KD tree is used
   - Space Complexity:
     – $O(mn)$
     – Need to store all training examples, which is $m * n$ space
4. Decision Boundary
   - KNN produces decision boundary with arbitrary shapes
   - In 1NN's case
     – the decision boundaries are the edges of Voronoi diagram
     – For each example, it has a Voronoi region where all points closest to this example (1NN) are located.
5. Pros and Cons:
   - Pros:
     (a) Only calculate distance without model building hence could be fast for training
   - Cons:
     (a) slow for large data because of the large $O(mn)$ time and space complexity. But as solution we can use efficient data structures such as KD-trees and ball-trees to improve the time and space complexity.
     (b) very sensitive to value $K$
     (c) usually requires feature normalization in advance
     (d) For high-dimensional data with many features, usually not very effective and requires dimension reduction and feature selection in advance.

**Definition 2.4. Weighted KNN**

1. Similar to KNN but when performing classification or regression for the new example, the voting contribution of K neighbors are weighted by their distance. The closer neighbors will be given larger weight for voting labels/target values, i.e. their own labels/values will contribute more in the voting.
2. Example: when a new example is surrounded by multiple blue examples that are far away but very few red examples that are nearby. If we only apply KNN without weighting, the new example is likely to be assigned blue label due to the majority of blue examples. But if we apply weighted KNN, the new example could be assigned red label.
3. Weighting heuristic: $w = \frac{1}{d^2}$. So if neighbors are far away, it will be assign much small weights.

## 2.2 Rule-Based Algorithms

**Definition 2.5. 1R and the best 1R**

1. Definition: generate a single rule based on a single attribute to predict the target label
2. Example: 1R for 'Outlook' attribute:
   - sunny -> 2y3n -> play: no
   - overcast -> 4y0n -> play: yes
   - rainy -> 3y2n -> play: yes

   'Outlook' attribute has 3 distinct values: sunny, overcast, rainy and the target label we want to use the existing data of 'Outlook' - Play relationship to generate a rule. However, not all values associate with the same target label so we can use majority voting to pick the most common label for each attribute value. If frequency has a tie, randomly choose the label.
3. Best 1R
   - Definition: 1R is associated with only one attribute. In order to select the best rule, we can pick the attribute/1R that has the smallest training error.
   - Procedure
     (a) loop through all attributes and generate 1R for each attribute. If we have $n$ attributes then we have $n$ 1Rs.
     (b) evaluate the training error of all 1Rs and pick the 1R with the smallest training error as the best rule.
   - Example: W2 - p24
4. Pros and Cons:
   - Pros: simple algorithms, easy to understand and interpret; useful as a baseline of complex algorithms
   - Cons: Numerical attributes requires discretization first, which is built-in for 1R

**Definition 2.6. PRISM**

1. Definition: A rule-based *covering* algorithm that aims to generate class-specific rules where each rule covers all examples in each class but not a single example in other classes.
2. Procedure:
   (a) Starting from a class say $a$ and generate a empty rule for this class
   (b) Keep adding tests/conditions to this rule to cover all examples in class $a$ but exclude all examples from other classes.
   (c) For each rule, at each step of adding test, add the one that maximize $\frac{p}{t}$ (examples covered in the target class (e.g. $a$) over total number of examples covered) and stop adding tests to the current rule when $\frac{p}{t} = 1$.
   (d) If there's remaining uncovered target examples, create a new partial rule (non-overlapping with previous rule but still part of class $a$'s rule) and keep adding tests like before until all examples in $a$ are covered and no other classes' examples are included. Note that at this point the training data accuracy should be 100%.
   (e) Starting from another class in the remaining class set and create a rule for as previous steps.
3. Example: W2 - P29-37, focus on the class hard-lense example
   - check each attribute value and their target class coverage ratio $\frac{p}{t}$. astigmatism = yes has the highest $\frac{p}{t}$ hence add this test to the rule as the 1st test
   - After filtering astigmatism = yes to all examples, searching for the 2nd test with highest $\frac{p}{t}$ ratio, which is tear preduction = normal. Add this test to the rule as the 2nd test.
   - After filtering astigmatism = yes and tear preduction = normal, searching for the 2nd test with highest $\frac{p}{t}$ ratio, which is prescription = myope. Add this test to the rule as the 3rd test.
   - At this point, all 3 selected examples are from hard class and no examples have other class labels. However, we didn't cover all examples from hard class. One example is missing from our current rule. So we need to stop adding new tests to the current rule and start another rule (still belongs to class 'hard') to cover the remaining one hard example.
   - **Delete** the 3 selected examples from 'hard' class and start a new rule. Similar to previous steps, searching for a test with highest $\frac{p}{t}$. As a result, three tests found to cover the remaining 1 hard example: age = young, astigmatism = yes and tear = normal.

- Now all hard examples are covered by the following two rules and no examples from other classes are covered. Hence stop the rule creation for 'hard'.
  - partial rule 1: astigmatism = yes, tear prediction = normal, prescription = myope
  - partial rule 2: age = young, astigmatism = yes and tear production = normal
- Move on to the rule creation for next class 'soft' then 'none' until all classes have rules.

4. Remarks:
   - adding tests to the rule shrinks the rule coverage
   - each test/condition test 1 attribute value (e.g. age = young, age = pre-presbyopic, astigmatism = yes, astigmatism = no, $x > 1.2$, $y > 2.6$, ...)
5. Pros and Cons:
   - Cons: some test examples will not be covered by PRISM rules and hence will have no classification labels or regression prediction. So a default rule is applied to assign them to the most popular labels; need discretization for numeric attribute values.
6. Comparison with decision tree:
   - Similarity:
     - Each step add the best tests to best split or with highest contain ratio
     - Top down approach
   - Difference:
     - PRISM is covering approach while decision tree is divide-and-conquer approach. PRISM aims to cover all target examples but nothing else. decision tree aims to find attributes that best split all classes but not focusing on any target classes.
     - Different criterion for adding tests: PRISM maximizes for target example containing ratio $\frac{p}{t}$ while decision trees maximizes the class separation such as information gain/entropy reduction.

# 3 Week 3

## 3.1 Linear Regression

**Definition 3.1. Generalised Linear Model (GLM)**

The GLM generalizes linear regression by allowing the linear model to be related to the response variable via a link function and by allowing the magnitude of the variance of each measurement to be a function of its predicted value. It has three elements

1. An exponential family of probability distributions.
2. A linear predictor $Z = Xw$
3. A link function $g$ such that $E(Y \mid X) = \mu = g^{-1}(Z)$
   - usually $g = \sigma^{-1}$ so $g^{-1}(Z) = \sigma(Z) = E(Y \mid X)$

Classic GLMs include

1. Least square regression ($Xw$ linear predictor, including Lasso and Ridge)
2. Logistic regression ($Xw$ linear predictor)
3. Binomial Regression
4. etc

L1 L2

**Definition 3.2. Ordinary Least Square Regression**

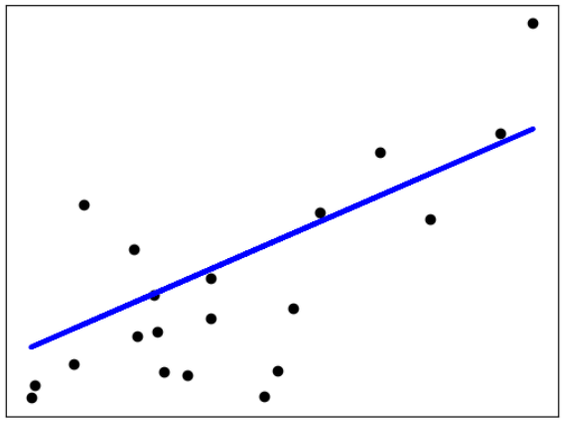


Figure 1: Ordinary Least square Regression

1. The objective function:
$$\min_w \|Xw - y\|_2^2$$
   - minimise RSS
   - use linear approximation $Xw$ to approximate $y$
2. Assumption: feature independence.

   If it's violated i.e. feature columns in design matrix $X$ are linearly dependent, then
   - Collinearity issue arises! (sol: good experiment design to avoid it in first place)
   - $X$ would be close to singular ($det(X) \approx 0$)
   - Least square estimate $w_{ls} = (X^T X)^{-1} X^T y$ would be highly sensitive to label errors/noise and will have overfitting issue
3. Closed-form solution (it exists here)

- Formula
  - $w_{ls} = (X^T X)^{-1} X^T y$
  - where $X \in \mathbb{R}^{n \times k}$, $y, \hat{y} \in \mathbb{R}^{n \times 1}$, $w \in \mathbb{R}^{k \times 1}$, $\hat{y} = Xw$
- Complexity
  - Time complexity: $O(nk^2 + k^3)$ operations
    - If $n >> k$, $O(nk^2)$
    - If $k >> n$, $O(k^3)$
    - Reason: matrix multiplication of $X^T X$ takes $O(nk^2)$ operations; matrix ($X^T X \in \mathbb{R}^{k \times k}$) inverse takes $O(k^3)$ operations $\Rightarrow O(nk^2 + k^3)$
  - Space complexity: $O(nk + k^2)$ floats
- Suitable case: when $n$ and $k$ are small as it's still cheap to calculate closed-form solution

4. Gradient descent (i.e. full batch) solution

- Applicable case: when $n$ and $k$ are too large and it's way to expensive to calculate closed-form solution
- Complexity
  - Time complexity: $O(nk)$ per iteration (here given its full-batch GD one iteration = one epoch and we need to process the whole training data)
  - Space complexity: $O(k)$ per iteration

**Definition 3.3. Ridge Regression ($\ell_2$ Regularised LS)**

1. The objective function:

$$\min_w \|Xw - y\|_2^2 + \lambda \|w\|_2^2 \text{ where } \lambda \geq 0$$

- minimise penalised RSS
- $\ell_2$ regularization
- $\lambda$ controls the weight shrinkage. The larger the $\lambda$ the larger the weight shrinkage $\Rightarrow$ less overfitting and more robust to collinearity

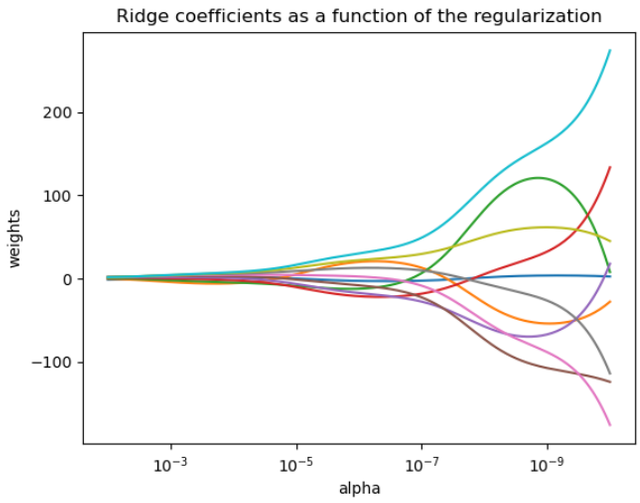



Figure 2: Relationship between regularization parameter and weights

2. Closed-form solution

- Formula
  - $w_{Ridge} = (X^T X + \lambda I_k)^{-1} X^T y$ with an extra term
  - where $X \in \mathbb{R}^{n \times k}$, $y, \hat{y} \in \mathbb{R}^{n \times 1}$, $w \in \mathbb{R}^{k \times 1}$, $\hat{y} = Xw$
- Complexity: same as OLS

**Definition 3.4. Lasso Regression ($\ell_1$ Regularised LS)**

1. The objective function:

$$\min_w \|Xw - y\|_2^2 + \lambda \|w\|_1 \text{ where } \lambda \geq 0$$

2. good for estimating sparse coefficients

**Definition 3.5. Logistic Regression**

1. The main idea of logistic regression
   - all about modelling posterior probability $P(Y = i|x, w)$ without doing any classification
   - obtain posterior probability from mapping the linear predictor $z = \hat{y} = w^T x$ from real number to the probability range [0,1] using logistic function (aka sigmoid function)
   - do some classification with probability using some prespecified criteria

   .
   In binary case, it is
   - $P(y = 1|x, w) = \sigma(z)$ where $y = 1$ usually represents success/target
   - $P(y = 0|x, w) = 1 - \sigma(z)$
   - Compact expression: $P(y|x, w) = \sigma(w^T x)^y [1 - \sigma(w^T x)]^{1-y}$

   where
   $$\begin{aligned} & P(y = 1|x, w) \\ &= \sigma(z) \\ &= \frac{1}{1 + e^{-z}} \\ &= \frac{1}{1 + e^{-w^T x}} \in [0, 1] \end{aligned}$$
   and we can use it to calculate the probability of "success"/"pass the exam"/etc given input features

2. The objective function
   $$\min_w \sum_{i=1}^{n} \ell_{log}(y_i \hat{y}_i)$$
   $$\Leftrightarrow \min_w \sum_{i=1}^{n} \ell_{log}(y_i w^T x_i)$$
   where logistic loss $\ell_{log}(y_i w^T x_i) = \log\left(1 + e^{-y_i w^T x_i}\right)$ is convex

3. No closed-form solution exists. But we can use some descent method to find an iterative solution for the Empirical Risk.

4. Update rule:
   $$w_{k+1} = w_k - \alpha_k \sum_{i=1}^{n} [\sigma(w^T x_i) - y_i] x_i \text{ with no regularization}$$
   $$w_{k+1} = w_k - \alpha_k \left[\sum_{i=1}^{n} [\sigma(w^T x_i) - y_i] x_i + \lambda w_k\right] \text{ with regularization}$$

5. Log-likelihood
   $$\log P(D) = \sum_{i=1}^{n} y_i \log \sigma(w^T x_i) + (1 - y_i) \log[1 - \sigma(w^T x_i)]$$
   more convenient to calculate cross-entropy error
   $$\begin{aligned} f(w) &= -\log P(D) \\ &= -\sum_{i=1}^{n} y_i \log \sigma(w^T x_i) + (1 - y_i) \log[1 - \sigma(w^T x_i)] \end{aligned}$$
   $$\nabla f(w) = \sum_{i=1}^{n} [\sigma(w^T x_i) - y_i] x_i$$
   if regularized then
   $$f(w) = \{-\sum_{i=1}^{n} y_i \log \sigma(w^T x_i) + (1 - y_i) \log[1 - \sigma(w^T x_i)]\} + \lambda \|w\|_2^2 \text{ Cross-entropy loss + penalisation}$$
   $$\nabla f(w) = \sum_{i=1}^{n} [\sigma(w^T x_i) - y_i] x_i + \lambda w$$

6. Multi-class Classification: key idea is the same, find some function that helps map $w^T x$ to [0,1] and obtain posterior probability estimate for each class

- In binary classification, we use sigmoid function $\sigma()$ to map $w^T x$ to [0,1]
- In multi-class classification, we use softmax function instead to map $w^T x$ to [0,1]

$$P(c_k|x) = \frac{e^{w_k^T x}}{\sum_i e^{w_i^T x}}$$

where we form a probability vector for all classes and $P(c_k|x)$ is an entry denoting the probability we predict that $x$ belongs to $c_k$ class. We then use argmax to find the class associated with highest posterior probability and use that as the predicted class for $x$

## 3.2 Descent Algorithms

This is more like a theoretical analysis. In Keras, for sgd optimiser, whether we adopt full-batch GD, SGD or mini-batch SGD totally depends on the batch size we specify. Say if we specify batch size to be 1, then it is equivalent to SGD; if $n$ then full-batch GD; anything in between, mini-batch GD.

**Definition 3.6. (Full Batch) Gradient Descent (GD)**

1. Definition: compute the gradient of the cost function wrt parameters over the whole training data set per iteration (i.e batch)
2. Start at a random point
3. Update rule:

$$\begin{aligned} w_{i+1} &= w_i - \alpha_i \nabla f(w_i) \\ w_{i+1} &= w_i - \alpha_i \sum_{i=1}^{n} (wx_i - y_i)x_i \\ &= w_i - \alpha_i \sum_{i=1}^{n} g_i \end{aligned}$$

$w_{i+1} = w_i - \alpha_i \sum_{i=1}^{n} (wx_i - y_i)x_i$ (least square regression's case)

$$\begin{aligned} f(w) &= \frac{1}{n} \sum_{i=1}^{n} (wx_i - y_i)^2 \\ \nabla f(w_i) &= \frac{2}{n} \sum_{j=1}^{n} (wx_j - y_j)x_i \\ w_{i+1} &= w_i - \alpha_i \nabla f(w_i) \\ &= w_i - \frac{2\alpha_i}{n} \sum_{i=1}^{n} (wx_i - y_i)x_i \end{aligned}$$

usually $n$ is a fixed constant so $\frac{2}{n}$ can be absorbed into $\alpha_i$

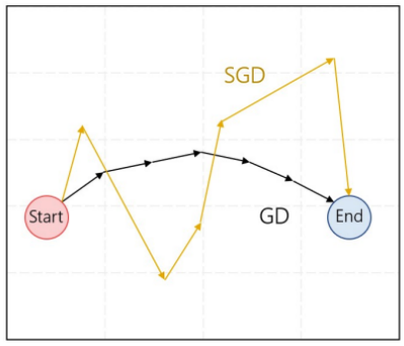
$$\Leftrightarrow w_{i+1} = w_i - \alpha_i \sum_{i=1}^{n} (wx_i - y_i)x_i = w_i - \alpha_i \sum_{i=1}^{n} g_i$$

4. Descent direction: -$\nabla f(w_i) = -\sum_{i=1}^{n} g_i$
5. Step size: commonly, $\alpha_i = \frac{\alpha}{n\sqrt{i}}$ where $i$ is the number of iterations and $n$ is the number of training points.
6. Pros and Cons
   - Pros: uses full batch to calculate gradient hence more information, faster convergence and less iterations
   - Cons: when the size of training samples $n$ is too large, it is computationally expensive to update the $w$ each iteration by calculating $\nabla f(w_i)$, which is a sum over $n$ samples

**Definition 3.7. Stochastic Gradient Descent (SGD)**

1. Goal: address GD's expensive computational cost
2. Approach: approximate and replace the full gradient (sum of $n$ examples) with only one random example per epoch/iteration
3. Update rule:

$$\begin{aligned} w_{i+1} &= w_i - \alpha_i \nabla f_r(w_i) \\ &= w_i - \alpha_i (wx_r - y_r)x_r \\ &= w_i - \alpha_i g_r \end{aligned}$$




where the sum is gone and $x_r, y_r$ is just one random example out of all $n$ examples

4. Pros and Cons:
   - Pros:
     - less computational cost per iteration
     - might end up with faster convergence as each iteration takes less time now
     - on average SGD approximates the full batch GD
   - Cons:
     - much slower convergence
     - more random and unstable convergence (zigzag)
     - more iterations involved

**Definition 3.8. Minibatch SGD**

1. Aim to improve the drawbacks of SGD
2. During each iteration, sample a minibatch $B_i$ and calculate the gradient over the minibatch instead just one sample
3. Update rule:
$$w_{i+1} = w_i - \alpha_i \nabla f_{B_i}(w_i)$$
$$w_{i+1} = w_i - \alpha_i \sum_{i=j}^{|B_i|} (wx_j - y_j)x_i$$
4. Pros and Cons
   - Pros
     - more parallelizable compared to SGD
     - less computational cost compare to GD ($|B_i| < n$)
     - can tune $|B_i|$ to improve the trade-off between computational cost and convergence rate/randomness in convergence
   - Cons
     - slower convergence than GD
     - extra tuning of $|B_i|$
     - still have randomness compared to GD

# 4 Week 4

## 4.1 Naive Bayes

**Definition 4.1. Naive Bayes Classifier**

1. Type: probabilistic classification method.
2. Theoretical foundation: Bayes Theorem
3. Two assumptions, which are both unrealistic/naive
   (a) Attributes are conditionally independent of each other. Given the class label, each attribute value is independent of each other, i.e.
   $$P(E|Yes) = P(E_1 \cap E_2 \cap E_3|Yes) = P(E_1|Yes)P(E_2|Yes)P(E_3|Yes)$$
   (b) Attributes are equally important.
4. Two cases dependent on the attribute types
   (a) Categorical attributes: just count the frequency as probability and calculate $P(E_i|Yes)$.
   - Examples: W3 - p11-18

   (b) Numerical attributes: can't count anymore. But can assume attributes follow normal distribution and use the pdf as a proxy of probability.
   - Example: W3 - p22-25. Essentially, calculate yes&attribute set's mean and std and no&attribute set's mean and std. Assume all yes&attribute or no&attribute value follows normal distribution and substitute the new example's attribute value into the normal pdf to calculate probability proxy.

   $$f(x) = \frac{1}{\delta\sqrt{2\pi}} exp(-\frac{(x-\mu)^2}{2\delta^2})$$
   $$P(temperature = 66|Yes) \approx f(temperature = 66|Yes) = \frac{1}{\delta_{yes,temp}\sqrt{2\pi}} exp(-\frac{(66-\mu_{yes,temp})^2}{2\delta^2_{y,temp}})$$
5. Edge cases and solutions
   (a) $P(E_1|Yes) = 0$, some attribute value of new example may not exist in the data. For example, Outlook = sunny is one of new example's attribute but in existing data $P(E_1|Yes) = (Outlook = sunny|play = Yes) = 0$. It will make the whole $P(E|Yes) = 0$ then always classify a new example as No. We need to do Laplace correction/m-estimate for all probabilities, including NO too. It is done by adding 1 to nominator and adding the number of distinct attribute values in to the denominator.
   (b) some attribute value of new example is missing. For example, in new example $E_1$ = (Outlook = ?). Solution is to exclude the missing value's posterior probability $P(E_1|Yes)$ in the calculation
6. Pros and Cons
   - Pros:
     – easy to calculate probability and then classify new examples due to the naive assumptions (ind attribute assumption)
     – easy to handle missing values as we just exclude them from calculation
     – can handle both numeric and categorical attribute values and calculate probabilities
     – robust to isolated noise points, which has small impact on the resulting posterior probabilities
   - Cons:
     – One of the naive assumption - independent attributes/features assumption is unrealistic since real-world attributes are commonly correlated (think about housing price example where housing areas and number of car parks are easily correlated). Solution is to apply feature selections to identify and discard redundant correlated features.
     – For numerical attributes, the normal distribution assumption is unrealistic too. Solution is to discretize numerical attributes to categorical ones first or adopt other more suitable distribution.

## 4.2 Evaluating ML Classifier's Performance

**Definition 4.2. Background Knowledge**

1. Parameter: Model parameter, part of the model such as $w_i$ the coefficient of linear regression model. Trained from data and are not controlled/tuned by us.

2. Hyperparameter: free parameters, can be tuned by us. Examples include $K$ in KNN, number of training epochs, SVM kernel type, etc.

3. Stratification: make sure that when splitting data into different sets, target classes ratio are not unbalanced in all sets. Otherwise it will confound our training and testing performance (e.g. training set has no class = no examples, which all assigned to testing set. Classifier trained over training set will definitely perform badly over testing set.

**Definition 4.3. Performance Evaluation Procedure**

1. **Holdout Method**

   - Intuitively, holdout here is referring to reserve partial original dataset as the testing set and keep it separate from the training set, so we wouldn't train on data we need to test for. This is because we want to use test set to mimic unseen data such that the testing error we obtained from testing set approximates the generalisation error well.
   - Procedure:
     (a) Split the original data into training and testing sets (randomly, split ratio given)
     (b) Train classifier over training set
     (c) Evaluate the model over testing set

2. **Repeated Holdout Method**

   - Definition: Repeat the splitting of original data $n$ times, obtain multiple accuracy and calculate the average accuracy
   - Pros and Cons:
     – Pros: Repeating holdout multiple times and then average the result can address the randomness introduced to some extent
     – Cons: The splitting is still random and training-testing sets across $n$ repeated runs may overlap with each other, e.g. running $n$ times but half of them the data splitting are similar hence the extra run of accuracy could be redundant, which defeats the whole purpose of repeated runs and wastes time and efforts. Solution: Use Cross-validation (CV) to avoid training-testing sets overlapping.

3. **Cross-Validation**

   - Motivation
     The motivation of using CV instead of training-testing split sets are as follows. If we only split original data into training-testing sets and use such combo for model training and parameter tuning, we are basically hacking the testing procedure as we are trying to utilize testing set to tune for the best hyper-parameters. In such way, the resultant model can overfit the testing set and may not generalise well if it's tested on another testing set. The solution is to holdout another part of original dataset as **validation set** and we basically train models over training set, tune hyperparameters over validation set and do final evaluation of the best tuned model over testing set and accept whatever result it gives from testing set. However, by splitting original set into 3 sets we greatly reduce the size of training set available and also makes the training result dependent on the specific training-testing split (which repeated holdout method can't fully solve either).
   - Solution: CV or K-fold CV. No need for specific standalone validation set anymore. Keep only training and testing set and further split training set into multiple non-overlapping folds, where validation set is embedded as one fold in each split and can be $K$ different folds.

   (a) **K-fold Cross Validation**

   - Procedure: Reserve/holdout testing set and further split **training set** into K non-overlapping folds. There would be $K$ total splits/runs and for each run, $K-1$ folds will be used for training and the remaining 1 fold will be used for *validation or "testing" (frequently called this, which is different from the final testing)*. The illustration is crystal-clear.
   - Best K: K=10 is the best by practice
   - **Special case: Leave-one-out Cross-Validation**
     – Definition: when $K = n$, we will use $K - 1 = n - 1$ samples for training and 1 sample for validation/'testing'.
     – Pros and Cons:
       ∗ Pros:
         i. Extremely effective use of training data as much more training examples $(n-1)$ are devoted for training and much more models were trained ($n$ models from $n$ runs) and evaluated for calculating average accuracy.

ii. There are always only $n$ ways to split the training data hence the result will be deterministic and no more randomness just like in the general K-fold CV case (where there could be many ways to do K-splitting and have to reduce such randomness by methods like Repeated K-fold CV)

* Cons:

i. Much larger computational cost. Assuming large dataset, n » K, then we need to train $n$ classifiers over with $n-1$-size training set, much larger computational cost compared to $K$ classifiers over $\frac{n(K-1)}{K}$ training examples

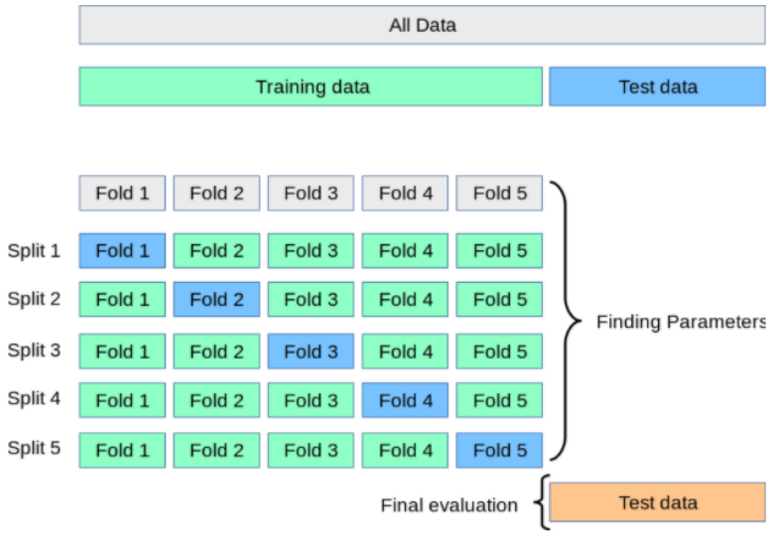



Figure 3: K-fold CV illustration

(b) **Stratified K-fold Cross-Validation**

- Each fold is stratified and the percentage of examples of different target classes are balanced. Stratification is crucial as mentioned in Background section.

(c) **Repeated Stratified K-fold Cross-Validation**

- Definition: run Stratified K-fold Cross-Validation $n$ times and average the result.
- Intuition: Even though we solved the overlapping issue from holdout method by using non-overlapping folds in K-fold Cross-Validation, we still face the randomness issue introduced by the split randomness of K-folds. To alleviate this issue, we can repeat the K-fold splitting of original data multiple times and average the result. Similar to LLN/sample avearge to expectation idea, as long as $n$ is big, the randomness caused would be small after averaging.

4. **Hyperparameter Tuning using Cross-Validation**

(a) Procedure: GridSearchCV

i. Create hyperparameter grid
ii. Loop through each hyperparameter combo, train model with such combo using K-fold CV (so not the full training set, only K-1 folds) and compute the cross-validation accuracy
iii. Select the best parameter combo with highest cross-validation accuracy
iv. Rebuild the model with such best hyperparameter combo over the whole training set
v. Evaluate the rebuilt model over the actual testing set
vi. Report the result

(b) Remarks: GridSearchCV with K=10 can be slow. Say we have 10 different combos of hyperparameters, and for each combo we need to run 10 models where each model trained over 90% training data and evaluated over 10% remaining validation/'testing' data. So in total 100 models to run before selecting the best combo!

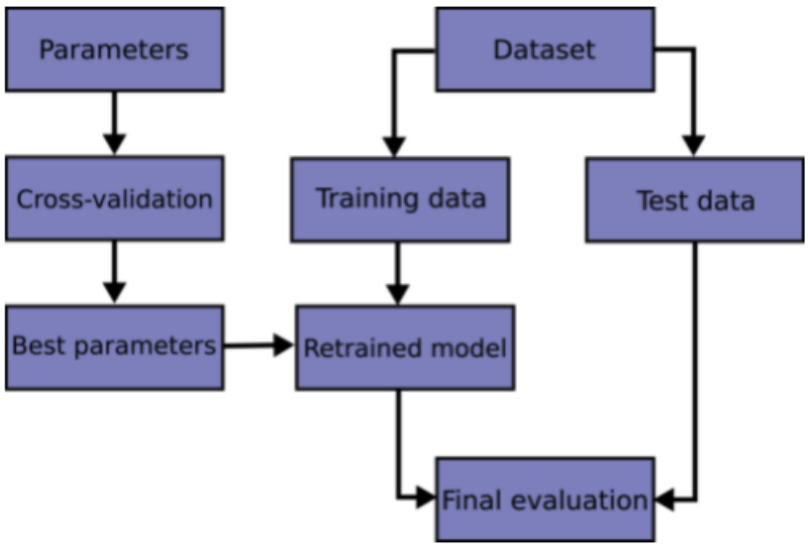



Figure 4: Parameter Tuning Pipeline

**Definition 4.4. Performance Measures**

1. Confusion matrix
   - not a performance measure, it is just a matrix output that is used for calculating performance measure.
2. Precision, Recall, $F_1$ Score, Accuracy
   - Precision: with regard to the prediction, the denominator is $\hat{y}$. TP/(TP+FP)
   - Recall: with regard to the true label, the denominator is $y$. TP/(TP+FN)
   - F1 Measure: harmonic mean of Precision and Recall, 2*Precision*Recall/(Precision+Recall)
   - Accuracy: TP/ALL = TP/(TP+FP+TN+FN)

# 5 Week 5

## 5.1 Decision Trees

1. Definition
2. Construct a decision tree
   (a) Pick an attribute as root node
   (b) Create branches for all attribute values and split examples into branches by the attribute values
   (c) For each attribute value branch, pick an attribute as the node to split the subset further
   (d) Terminate splitting until all examples in the branch have the same class and create a leaf node for that class
3. Which attribute as the root node, or non-leaf node?
   (a) Discrete Attribute (e.g. Outlook)
   - Example in the slide p10, can split can outlook, temp, humidity, windy. 4 options.
   - need to find a measure of purity such as information gain or entropy reduction
   - calculate the entropy reduction/information gain for all 4 option splits, pick the attribute that produces the highest entropy reduction/information gain.
   - Split the dataset $S$ further by attribute values (outlook-sunny, outlook-overcast, outlook-rainy). Trying to find new attribute to split $S_i$ using the same way, until each subset is pure.

   (b) Numerical Attribute (e.g. Temperature)
   - p27, Essentially discretization, split numerical values into "nominal" groups (e.g. group 1: temperature $<$ 70.5; group 2: temperature $\geq$ 70.5)
   - find all the split point where class changes and calculate the entropy reduction/information gain for each split point.
   - choose the split point with highest information gain/entropy reduction
4. Entropy
   - measure of the example set $S$'s uncertainty/impurity w.r.t. some target class. Usually measured in bits.
   - small entropy $\Leftrightarrow$ small uncertainty $\Leftrightarrow$ large purity
   - Formula
   $$H(S)/I(S) = -\sum_i P_i \log_2 P_i$$
   where $P_i$ is the class proportion for class value $i$ and $\log_2 0 = 0$ by assumption
   - Binary case
     – Two classes Yes and No. Assume $P_{Yes} = p$ then $P_{No} = 1 - p$. When set $S$ has either all Yes ($p = 1$) or all No ($p = 0$) class values, $S$ has highest purity and least entropy ($H(S) = 0$). When set $S$ has half-half Yes and No class values, $S$ has the largest entropy ($H(S) = 1$) and least purity.

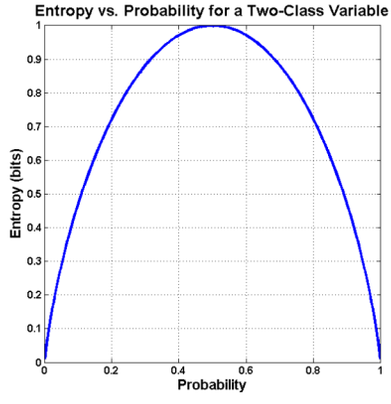



Figure 5: Caption

- Example: a 14-example set $S$ with 9 yes and 5 no. Then entropy of set $S$ is
$$\begin{aligned} H(S)/I(S) &= -\sum_i P_i \log_2 P_i \\ &= -P_{Yes} \log_2 P_{Yes} - P_{No} \log_2 P_{No} \\ &= -\frac{9}{14} \log_2 \frac{9}{14} - \frac{5}{14} \log_2 \frac{5}{14} \\ &= 0.94\text{bits} \end{aligned}$$

5. Information Gain
    - Definition: reduction of entropy
    - Formula: Gain = $T_1$ - $T_2$ where $T_1$ and $T_2$ are the entropy before and after the attribute split
    - Application: choose an attribute that produce the highest information gain aka highest entropy reduction as the next/best attribute to split
    - Example to calculate information gain: check slides.
6. Problems
    (a) Overfitting tree
        - context: tree can easily memorize the data, create super specific split and overfit the data. This happens often especially when training data is small or too noisy (labels don't provide general and accurate information. Tree will be confused and then just memorizes the data instead of extracting the patterns).
        - Solution: prune the tree to reduce model complexity
            i. Pre-pruning: stop growing the tree at some point
            ii. Post-pruning: fully growing the tree and then prune it later (preferred in practice)
                A. subtree replacement (compare the accuracy of replace or not replace situations, if higher or the same -> replace)
                B. subtree raising
                C. converting the tree to rules then prune them
        - Prune extent: build tree using training set; use validation set performance to evaluate the best pruning extent; test the pruning/the tree over test set

    (b) information gain metric may not work for certain attributes
        - context: highly-branching attributes such as SID will be selected as the split under information gain evaluation system. As a result, one-level tree with #SID branches.
        - solution: use gain ratio, an improved information gain metric, which penalises highly branching attributes
7. Pros and Cons
    - Pros:
        (a) easy to implement, visualise and interpret
        (b) can use pruning to avoid overfitting
        (c) applicable to both categorical and numerical attributes
    - Cons:
        (a) easy to overfit

## 5.2 Ensemble Method

**Definition 5.1. Ensemble Method**

1. Definition: a method that combines multiple base classifiers to make a prediction, where each base classifier's prediction result can be taken into the final prediction through majority voting, weighted voting, etc. Classifier using ensemble method tends to work better than each base classifier alone.
2. Goal: produce better performance/robustness over a single base classifier
3. Only possible under two conditions:
    (a) Base classifier is accurate. Error rate $\epsilon < 0.5$, i.e. base classifier's prediciton is better than random guess
    (b) Base classifiers are independent and diverse. Diversity can be created over training data (bagging and boosting), attributes (random forest) and learning algorithms.

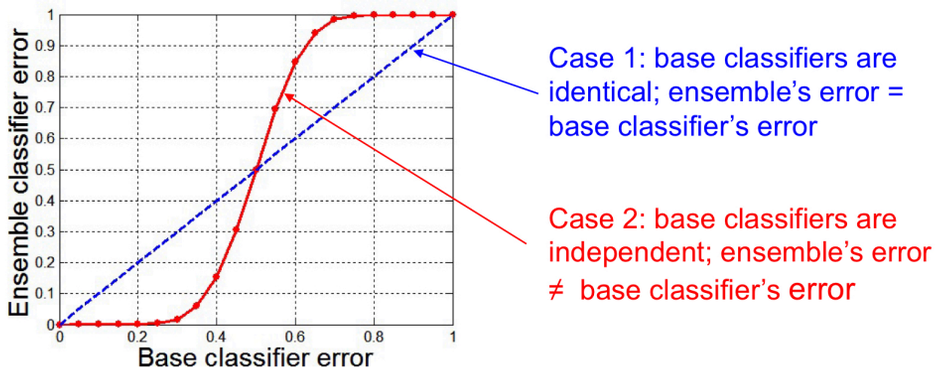



Figure 6: Caption

4. Two families by classifier building principle
   (a) Averaging method
      - Build several base classifier *independently* and then average/majority-voting the prediction results to produce the final prediction
      - Goal: reduce variance of a single classifier
      - Example: Bagging, Random Forest
   (b) Boosting method
      - Build several base classifiers *sequentially*
      - Goal: reduce bias and improve fitting; most suitable for combining weak base classifiers and produce a powerful ensemble
      - Example: AdaBoost, GradientBoosting
5. Methods to construct ensembles: essentially create disagreements between base classifiers
   (a) Data-focused: resample the original dataset and create multiple training sets. Train different classifiers using those training sets. Examples: bagging, boosting.
   (b) Feature-focused: use subset of attributes to train classifiers. Examples: random forest, random subspace
   (c) Label-focused
   (d) Algorithm-focused: use different parameters for the classifiers

**Definition 5.2. Bagging (Bootsrap Aggregation)**

1. Type: data-focused ensemble method. It creates disagreements/randomness between the training datasets of base classifiers
2. Goal: reduce variance and overfitting of a single base classifier such as a fully grown complex decision tree. Check `https://scikit-learn.org/stable/auto_examples/ensemble/plot_bias_variance.html#sphx-glr-auto-examples-`
3. Base classifiers: works best with strong and complex base classifiers such as fully developed decision trees
4. Sampling method: bootstrapping, which means samples were drawn from the original data *with replacement*
5. Voting:
   - majority voting for classification where each classifier is equally weighted
   - averaging for regression
6. Procedure:
   (a) Create $M$ bootstrapped datasets, where each dataset has the the same number of samples as the original dataset but it was sampled from the original dataset with replacement. This means $D'$ will have repetitive examples.
   (b) Train/build $M$ classifiers over $M$ bootstrapped datasets
   (c) Apply $M$ trained classifiers to the test set and obtain $M$ predictions and the most frequent prediction will be selected as the final prediction based on majority voting
7. Pros and Cons

- Pros:
  - Typically much better ensemble performance compared to the base one
  - Typically effective for unstable base classifiers (i.e. sensitive to data perturbation, e.g. decision trees, neural networks)
- Cons:
  -

**Definition 5.3. Random Forest**

1. Type: feature-focused and also data-focused ensemble. Each base classifier only uses a subset of features ($L < K$ features, usually the most important features) and was trained over a bootstrapped set of original training set.
2. Goal: Reduce variance of a single decision tree classifier and Introducing two sources of randomness and create an overall better performance model. This is done by creating more diverse trees in the forest, using feature subset and bootstrap data, which helps solve the overfitting issue of individual decision tree and reduce model variance over testing data.
3. Sampling method: same as bagging, bootstrapping aka sample with replacement from the original dataset
4. Features: a random subset of features is used for each tree
5. Base classifier: decision tree, built over a bootstrapped set
6. Tasks: can be used for classification or regression (RandomForestClassifier or RandomForestRegressor)
7. Procedure:
   (a) Create $M$ bootstrapped datasets like in bagging where each dataset contains the the same number of examples but just sampled with replacement
   (b) For each of the $M$ dataset, do random feature selection and filtered for a subset of features
   (c) Build classifiers over those filtered training sets
   (d) Combine classifier together using majority vote
8. Key hyperparameter to tune:
   (a) $n\_estimator$: the number of base classifiers/trees. Usually the larger the better but there would exist some number that over that the accuracy has no big difference. Also larger number of trees takes longer to train and more computationally expensive.
   (b) $max\_features$: the number of random features selected into the subset. Less features can reduce model variance but also increase bias (less fitting). In heuristic for classification task, $max\_features =' sqrt'$ i.e. sqrt of total number of features.
9. Application: feature selection based on the feature importance (i.e. entropy reductions). Usually the more important features are on the top of the tree as they produce the maximum entropy reduction and were split earlier.
10. Pros and Cons
    - Pros
      - The two sources of randomness (data bootstrapping (bagging) and random feature selections) help reduce tree variance and hence more robust to overfitting
      - Can be faster if only a subset of features is used
    - Cons
      - Can take a bit long when there are many trees
    - Remarks: suitable for accurate base trees with less correlation

**Definition 5.4. Adapted Boosting (AdaBoost)**

1. Key idea: boost weak learner's performance by having more focus on hard examples. Build a team of weak classifiers where subsequent classifiers "adapt" the difficult examples passed by previous classifiers and spend more efforts classifying them well.
2. Base classifier: weak learner with slightly better than random guessing performance, usually decision trees.
3. Voting: weighted majority vote, where more accurate classifiers are given more voting weight
4. Procedure

(a) Start a training set where each sample is weighted equally $\frac{1}{N}$ and train a weak classifier (e.g. small decision tree) $h_1$ over it.

(b) For each successive iteration, the sample weights are individually modified to create a reweighted data and $h_2$ is trained over the reweighted data. To be specific about the reweighting procedure, previously misclassified examples by $h_1$ will be assigned higher weight in the current iteration while correctly classified examples will be assigned lower weight. This forces the current or subsequent weak learners to spend more effort in classifying previously difficult training examples correctly.

(c) In the end, after specified number of base classifiers (i.e. $M$) is trained, combine the $M$ classifier with weighted voting to make a final prediction. Classifier with higher training accuracy will be given more weight.

**Definition 5.5. Gradient Boosting**

1. Key idea: similar to AdaBoost
2. Base classifier: weak learner, usually decision tree.

**Definition 5.6. Gradient Tree Boosting or Gradient Boosted Decision Trees (GBDT)**

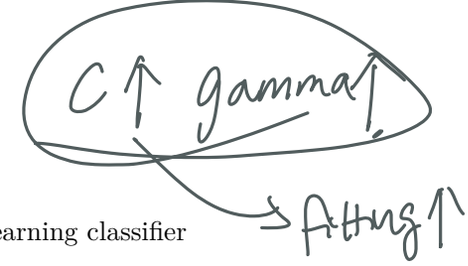
1. Base classifier: DT
2. Tasks: both classification and regression. $GradientBoostingClassifier$ or $GradientBoostingRegressor$
3. Key hyperparameters:
   - $n\_estimators$: the number of trees/weak learners to fit in the model
   - $learning\_rate$: in the range of (0,1]. It controls model overfitting by shrinking the contribution of each tree. Proven to boost accuracy with shrinkage/learning rate controls. There's usually a trade-off between $learning\_rate$ (more controls, less fitting) and $n\_estimators$ (more fitting). A small learning rate favors better test error.
   - $max\_depth$: control the tree size through depth
   - $max\_leaf\_nodes$: control the tree size through width
4. Pros and Cons:
   - Pros: fast, boost weak learner's performance
   - Cons: hard to interpret

**Definition 5.7. Compare Bagging and Boosting**

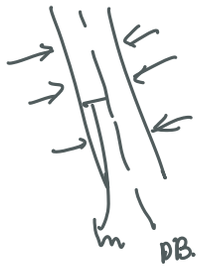
- Similarity:
  - Majority voting for classification and averaging for regression
  - Base classifiers are from the same type
- Difference:
  - Procedure: in bagging base classifiers are more independent and make predictions separately; In boosting classifiers are closely related and predict iteratively - the previous classifier pass misclassified and difficult examples to the next classifier
  - Weight: bagging has equal weight for all base classifier; boosting has different weights for classifiers depending on their performance - more accurate classifier has higher weight in the final prediction

# 6 Week 6

## 6.1 SVM




1. Type: supervised learning classifier
2. Support vectors: data points (also vectors) that lie closest to the decision hyperplane/surface and "support" the separating hyperplanes. These are the most difficult points to classify.
3. Margin: the street width between separating hyperplanes




4. Kernel types
   - Linear kernel
   - Polynomial kernel
   - Gaussian RBF kernel
5. Classifier Input and Out
   - Input: A set of training examples $\{(x_i, y_i)\}_{i=1}^{n}$
   - Output: A set of weights $\{(w_i)\}_{i=1}^{n}$ for each training examples.
6. Optimisation: Maximise margin to reduce the number of non-zero weights that are corresponding to the support vectors.
7. Pros and Cons
   - Pros
     (a) Effective for high-dimensional data/problem, even if the number of dimension is larger than the number of samples ($m > n$, but not $m >> n$)
     (b) memory efficient as it only utilises support vectors, which are a subset of training examples
     (c) Many types of kernel functions available to specify for the decision boundary
   - Cons
     (a) When the number of dimension is much larger than the number of samples ($m >> n$), it requires the use of kernel functions and regularization paramaters $C$ to avoid overfitting
     (b) SVM itself doesn't provide probability estimates and we have to calculate through CV(??????)
8. Difference compared to other
9. Time complexity

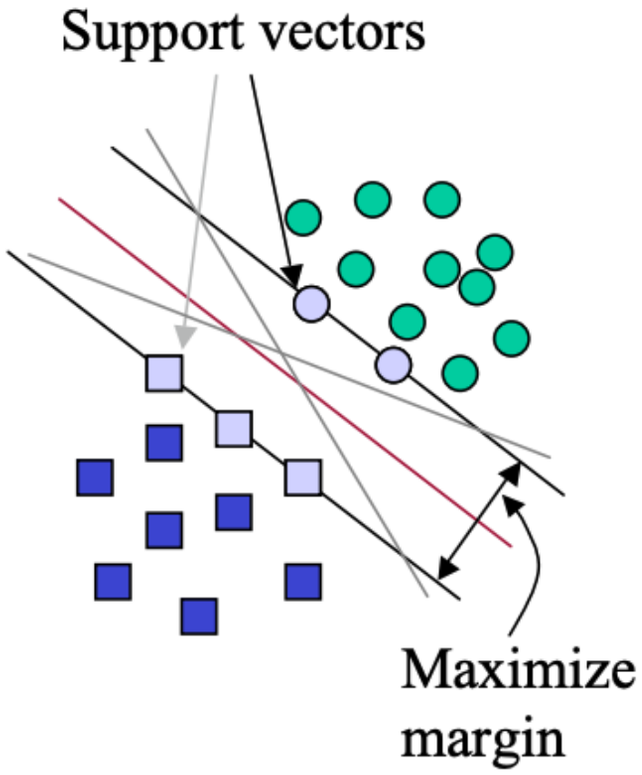



Figure 7: Caption

### 6.1.1 Maximum Margin Supporting Hyperplane

### 6.1.2 Linear SVM

### 6.1.3 Non-linear SVM

## 6.2 Dimension Reduction

**Definition 6.1. Dimension Reduction**

1. Motivation
   (a) In practice we usually have high-dimensional data with huge number of features (e.g. an $28 \times 28$ grey scale image has 784 features)
   (b) Training with such high-dim data would be super slow
   (c) Classifier is easy to overfit as it just memorizes patterns
   (d) Model is hard to interpret and visualise (e.g. CNN)
2. Philosophy: Not all features are important and many features are redundant, noisy (i.e. useless) and highly correlated (hence repetitive). Most variance can actually be captured by a small fraction of original feature dimensions.
3. Solution
   - Dimension reduction or namely feature extraction! This might even improve the accuracy as now the data representation is better.

**Definition 6.2. Principal Component Analysis (PCA)**

1. Type: dimension reduction or feature extraction method. Unsupervised learning method as no labels/values are provided.
2. Main idea: construct a new coordinate system spanned by *principal components* (usually way less than original) and then project original data into such new space. The projected data can then be used as ML algorithms input.
3. Characteristics:
   - Assume $Z_1$,..,$Z_k$ are $k$ largest principal components found where $k < m$

- PCs are usually ordered by the variance they capture from high to low
- PCs are also orthogonal to each other

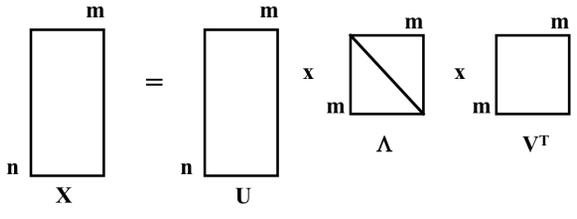
4. Choose the right number of PC
   - Find it according to the min variance we want to preserve
   - Elbow method, find subjectively from the variance-dim plot
5. Find Principal components: SVD

**Definition 6.3. Singular Value Decomposition (SVD)**

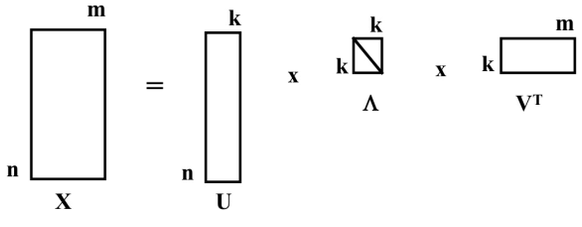
1. Motivation: Find principle components in $V$
2. Method: matrix factorization. Factorize the matrix $X$ into 3 matrices
$$X = U \times \Lambda \times V^T$$
where
   - $X \in \mathbb{R}^{n \times m}$ where $n \geq m$ and $X$ is the original data in the original space
   - $U \in \mathbb{R}^{n \times m}$ where $U$ is orthogonal matrix and also transformed data in the new coordinate system. Row in $U$ matches row in $X$.
   - $\Lambda \in \mathbb{R}^{m \times m}$ where $\Lambda$ is diagonal matrix and each entry in the diagonal is an eigenvalue $\lambda_i$/singular value (as $\geq 0$), sorted in a decreasing order
   - $V, V^T \in \mathbb{R}^{m \times m}$ where $V$ is orthogonal matrix. Each column of $V$ or each row of $V^T$ is an **eigenvector or principal components**
   - orthogonal matrix $A \Leftrightarrow AA^T = I$
3. Data Decomposition and Reduction:
   - $X$ can always be decomposed and represented by linear combination. If we think the first $k$ components are sufficient to preserve enough variance, represent the reduced $X$ with the first strongest $k$ terms and ignore the remaining weaker components.
   - As a result, $U$ becomes narrower.
   - In Python, $SVD(X) = ...$

   Data Decomposition: $X = \lambda_1 u_1 v_1^T + ... + \lambda_m u_m v_m^T$
   Data Reduction: $X_{reduced} = \lambda_1 u_1 v_1^T + ... + \lambda_k u_k v_k^T$ where $k < m$

   **Without data reduction:**




   **With data reduction:**




4. Applications
   (a) Image compression
      - Use the first $k$ terms to compress image but still remain most variance

- Compression ratio $r$

$$\begin{aligned} r &= \frac{\text{\#pixels after compression}}{\text{\#pixels before compression}} \\ &= \frac{k(1+n+m)}{nm} \\ &= \frac{k}{m} \text{ if } n >> m > k \end{aligned}$$

(b) Face recognition

- Use PCA first to extract main features with $k$ PCs then use KNN to find the "nearest neighbor"(i.e. most similar image) and see if they are the same person.
- PCA can improve accuracy for KNN

# 7 Week 7 Neural Networks

**Definition 7.1. Artifical Neural Networks**

Some classic ones

1. MLP
2. CNN
3. RNN

**Definition 7.2. (True) Perceptron and Neurons**

1. True Perceptron
   - Definition: (True) Perceptron, or single-layer perceptron, is a binary classifier where the activation function is chosen to be a step function. If weighted sum of input $< 0$ then output 0, otherwise 1.
   - Output: binary, either 0 or 1
   - Activation function: Step function
   - Use case: when data are linearly separable with binary classes
   - Cons: can't handle linearly inseparable data

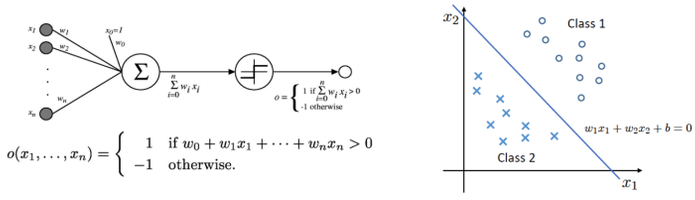



Figure 8: True Perceptron

2. Neuron
   - Definition: also known as artificial neuron, is a generalisation of perceptron
   - Output: can be binary or non-binary (continuous, a value in a range) dependent on the activation function
   - Activation function: can choose anyone

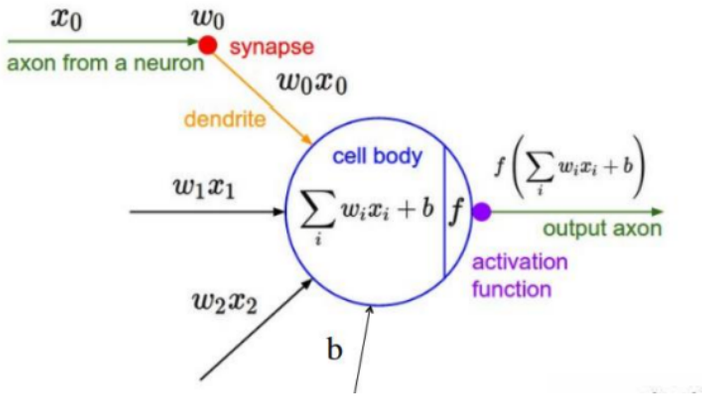



Figure 9: Artificial Neuron

**Definition 7.3. Layers**

1. Input layer: a row of input neurons that represents input data
2. Hidden layer: hidden as it's not exposed as input layer. The more hidden layers we have, the deeper the network is.

3. Output layer: the number of neurons dependent on the problem. Regression problem often has single output neuron with no activation function; Binary classification problem may have a single neuron with sigmoid function to denote the probability of the one class (e.g. class 1); Multiclass classfication problem may have one neuron for each class with softmax activation function to denote the probability of each class.

4. Naming convention: Does not include input layer into the number of layers. N-layer NN means there are N-1 hidden layers and 1 output layer.

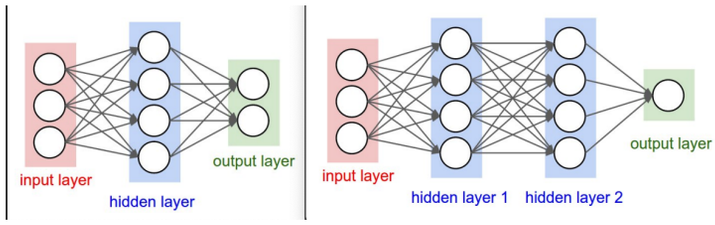



Figure 10: Left: 2-layer NN; Right: 3-layer NN

**Definition 7.4. Activation Function**

1. Definition: a simple mapping of weighted sum of neuron input to the neuron output

2. Assumption: often assume non-linearity, as it is preferred

3. Goal: introduce non-linearity into the neural networks, allowing the NN to be more expressive and approximates more complex functions

4. Why not linear activation?

   - A multilayer NN with linear activation is equivalent to a linear regression model. It can't learn from more complex and linearly inseparable data and has little power.

5. Non-linear Activation Functions:

   (a) A two-layer NN with non-linear activation is proven to be a *universal approximator* (known as the *Universal Approximation Theorem*

   - Traditional: Sigmoid, Tanh (i.e. hyperbolic tangent)
   - Modern: ReLU, LeakyReLU, ELU

   (b) Sigmoid

   - also known as logistic function
   - $\sigma(z) = \frac{1}{1+e^{-z}}$, $\sigma'(z) = \sigma(z)(1-\sigma(z))$
   - Output: (0,1)
   - Cons:
     i. Output easy to saturate given very positive or negative input, which leads to vanishing gradient problem especially with DNN (i.e. f(f(f(...))))
     ii. exp function costly to compute
     iii. function not 0 centered (i.e. slower convergence)

   (c) Tanh

   - Output: (-1,1)
   - Pros: 0-centered
   - Cons: easy to saturate hence vanishing gradient

   (d) ReLU

   - Pros: does not saturate when >0; faster convergence than sigmoid and tanh in practice
   - Cons: not 0-centered; <0 zone never gets activated and hence weight never gets updated

   (e) Leaky ReLU

   - Pros: does not saturate everywhere; faster convergence than sigmoid and tanh; no dead and non-activated gradient like ReLU

   (f) ELU

   - Pros: all pros of ReLU with extra robustness to noise
   - Cons: exp compuation can be costly

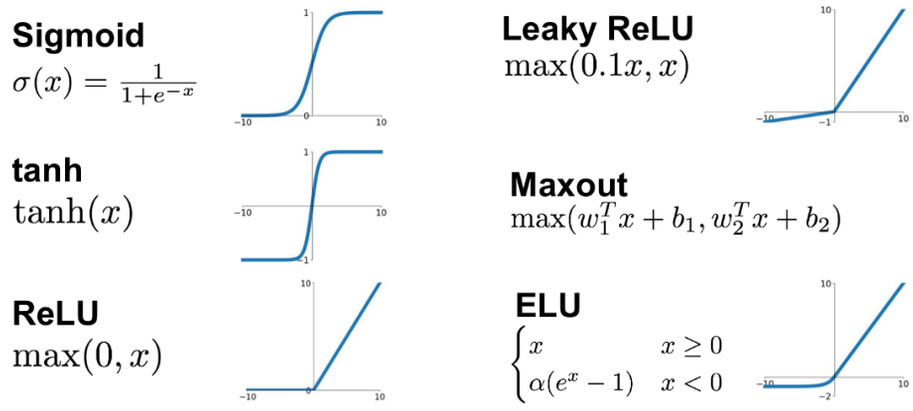



Figure 11: Non-linear Activation Functions

**Definition 7.5. Perceptron and Multilayer Perceptron (MLP)**

1. Type: feedforward neutral network
2. Layers: at least 3 layers of nodes
   - 1 input layer
   - at least 1 hidden layer
   - 1 output layer
3. Node: except for the input layer, each node is a neuron with a non-linear activation function
4. Difference between Single-Layer Perceptron/Perceptron and Multilayer Perceptron (MLP):
   - Perceptron is a component of MLP, where many "perceptrons" are organized into layers
   - MLP "perceptrons" here are not true perceptrons (i.e. step activation and binary only)

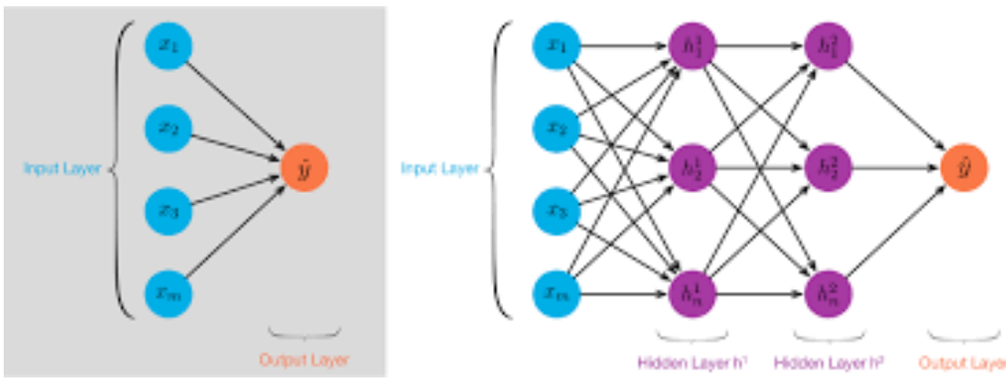



Figure 12: Left: Perceptron; Right: MLP

**Definition 7.6. Forward and Backward Propagation**

1. Forward Propagation/Forward Pass
   - Goal: feed forward the data to compute NN prediction and then calculate loss
2. Backward Propagation
   - Goal: compute gradient of the loss function with respect to each weight
   - Method: Dynamic programming. The gradient is calculated one layer at a time, iterating backward from the last layer to the input layer. This avoids redundant calculations of intermediate terms in the chain rule, which is very efficient.
   - Limitation:
     (a) Gradient descent with backprop is not guaranteed to find the global minimum of the error function, but only a local minimum
     (b) does not require normalization of input vectors while normalization actually can improve the performance

**Definition 7.7. Cost Function**

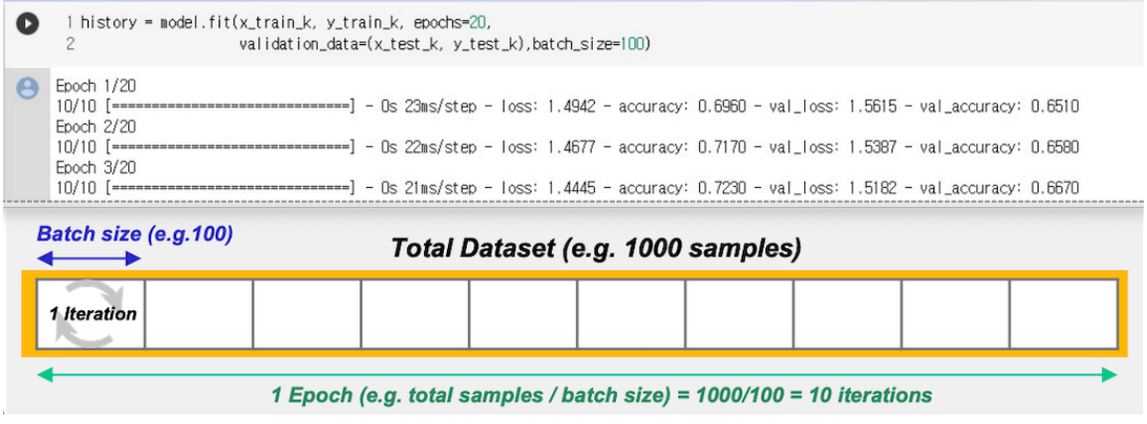
1. MSE: used for regression
2. Cross-entropy Loss: used for classification.

**Definition 7.8. Learning Rate**

1. Definition: also known as step size, controls the amount of weight that is updated during training. Instead of updating the weight with the full amount, it is scaled by the learning rate.
2. In general, larger learning rate may cause a very oscillating search, diverge, end up with a suboptimal solution; small learning rate, slower and may never converge
3. Need to find a learning rate that is not too large but also not too small, just good enough

**Definition 7.9. Epoch, Batch Size, Iteration and Pass**

1. Epoch
   - Definition: one forward pass and backward pass of the whole training dataset
   - Epoch size: often large
2. Batch
   - Definition: a subset of training data to work through before updating the model paramaters. Larger the batch size takes more memory.
   - Motivation: often can't fit in all training examples into memory in one go
   - Batch Size
     (a) (Full Batch) Gradient Descent: size of training data
     (b) Minibatch SGD: >=1 and <=size of training data. Often 32, 64, 128 and 256
     (c) SGD: 1
3. Iteration
   (a) Definition: number of iterations = number of passes
4. Pass
   (a) Definition: one pass includes both forward and backward passes
5. Example: 1000 examples in one epoch, split into 10 batches/iterations where each batch has 100 examples.




**Definition 7.10. Gradient Descent Algorithms / Optimiser**

The goal of gradient descent algorithm is to find the optimal point. "Gradient" refers to calculating gradient, "descent" refers to the moving down along that slope towards some minimum level of error. The gradient descent algorithm is iterative. I.e. forward pass with old weight to calculate the loss, backward pass to update the weight, then keep repeating both passes until it converges.

1. Full Batch Gradient Descent
2. Stochastic Gradient Descent (SGD)

3. Minibatch SGD
    - Cons: the same learning rate for each input variable
4. AdaGrad (learning rate)
    - Extension of SGD through learning rate
    - feature: automatically adapt the learning rate based on all previous seen partial gradients for each variable
    - Cons: can have a very small learning rate for each parameter by the end of the search. Slow down the search too soon before the minimal can be located
5. RMSProp (learning rate)
    - Extension of SGD and AdaGrad through learning rate
    - feature: automatically adapt the learning rate for each variable based on an exponentially decaying average of the past gradients. This allows the algorithm to forget early gradients and focus on the most recently observed partial gradients.
6. SGD with momentum (gradient)
    - Extension of SGD through gradient
    - Motivation: the search under SGD can oscillate a lot and even get stuck in flat spots that have no gradient.
    - feature: SGD with momentum incorporates an exponentially weighted average of previous gradients. This helps to build momentum and continue searching in the previous direction instead of getting stuck or oscillating.
    - Cons: acceleration can sometimes cause the search to overshoot the minima at the bottom of a basin
7. NAG (Nesterov's Accelerated Gradient)
    - Motivation: a modification to momentum to overcome this problem of overshooting the minima.
    - feature: like traditional momentum except it calculates the decaying moving average of the gradients of projected positions
8. Adam
    - Extension of SGD through both learning rate and gradient (?slide)
    - feature: combine the benefits of both AdaGrad and RMSProp that automatically adapts a learning rate for each input variable for the objective function

**Definition 7.11. Data Augmentation**

1. Definition: a technique to artificially create new training data from existing training data
2. Goal: improve the generalisation capability of the model and reduce overfitting
3. Example: image data augmentation creates transformed versions of training images through operations such as rotating, cropping and flipping.
4. Remarks:
    - the choice of the specific data augmentation techniques must be within the context of the training dataset. For example, flipping a cat image vertically wouldn't be realistic and appropriate.
    - only applied to the training dataset, and not to the validation or test dataset

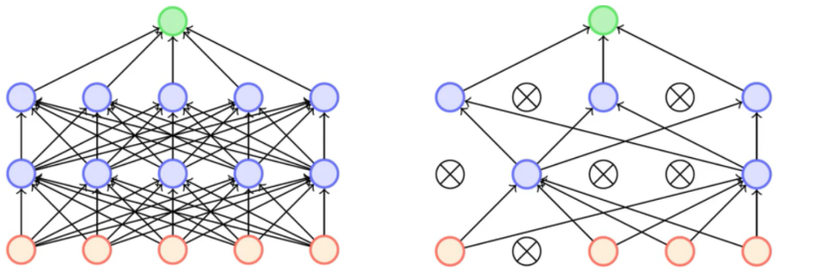
**Definition 7.12. Weight Decay**

1. Definition: penalise large weights through regulariser or constraints using either l2 or l1 regularisation
2. Purpose: reduce overfitting and improve generalised performance

**Definition 7.13. Dropout**

1. Definition: Dropout is a regularisation technique. By randomly dropping out a certain percentage of nodes in each layer per iteration, it approximates training a large number of neural networks with different architectures.
2. Training time only
3. Suitable layers: fully connected layers, conv layers.

4. Hyperparameter: retaining or dropout probability
5. Remarks
    - more effective than weight decay in general
    - often 0.5-0.8 for hidden layer and larger rate such as 0.8 for input layer
    - most effective for larger networks with more layers and nodes and limited data




**Definition 7.14. Batch Normalisation**

1. Definition: BN is a technique to standardize layer's input in a network.
2. Position: before or after the activation layer. For tanh and simoid, better after; for relu, better before.
3. Motivation: The layers' input distributions changes over time during training, which leads to internal covariate shift problem
4. Procedure
    (a) normalise the value to z
    (b) z*g
    (c) z*g + b
5. Happens at batch/iteration level
6. Pros:
    - stabilise and speed up training
    - reduce vanishing or exploding gradient issue
    - some regularisation effect
    - more robust and less sensitive to different learning rates or initialisation schemes

**Definition 7.15. Early Stopping**

1. Definition: a regularisation technique to reduce overfitting, where training is stopped once the validation performance stops improving (e.g. increase in loss or drop in accuracy)

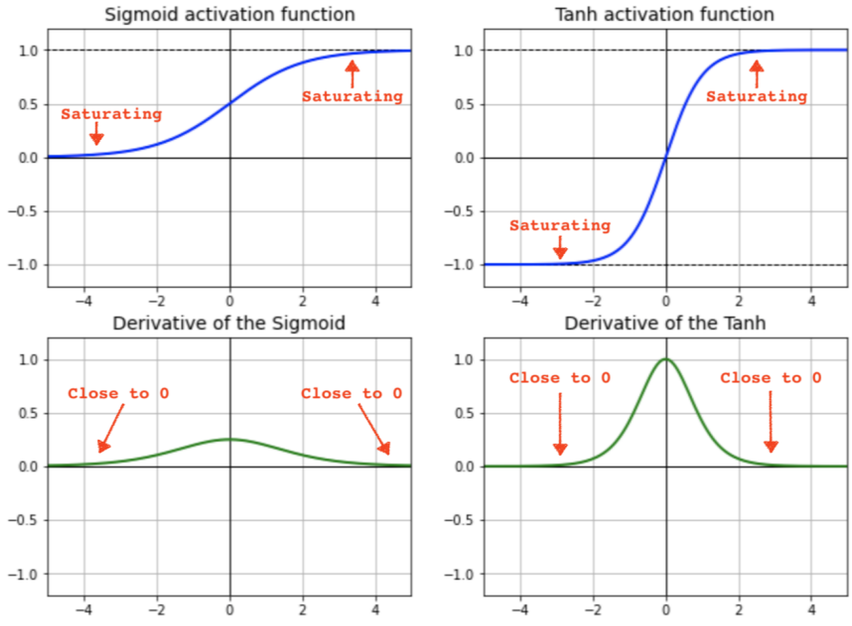
**Definition 7.16. Weight Initialisation**

1. Definition: initialise the weights of neural networks to small random numbers such that the optimisation process has a starting point
2. Pros: speed up training
3. Methods
    (a) Xavier Initialisation
        - Uniform distribution
        - Sigmoid activation
    (b) He Initialisation
        - Gaussian distribution
        - ReLU activation
4. Why we can't intialise big weight?
    - if sigmoid or tanh are used (i.e. easy to saturate ones) then the gradient can vanish

**Definition 7.17. Vanishing and Exploding Gradient**

During backprop, when calculating the gradient of loss wrt the weight, the gradient can increase or decrease dramatically when passing through layers.

1. Vanishing Gradient
   - Definition
     - Activation functions such as sigmoid and tanh are easy to be saturated for large and small input
     - The gradient for those saturated values are close to 0
     - During backprop, these small gradients keep getting smaller and smaller and vanish near the input layer.
     - This cause almost no weight updates and makes training slow or stopped
   - Symptom: The parameters of the higher layers change significantly while the parameters of lower layers does not change much; learning is very slow
   - Solution
     - Activation: Use activation functions such as ReLU (suffer from one side) and LeakyReLU (almost no suffer) which suffer less from saturating problems
     - Network: Use residual networks where skip connections allow gradient information to be passed through layers




2. Exploding Gradient
   - Definition: some large gradients keep on getting larger and larger as the backprop algorithm progresses from output to input layer. This in turn causes very large weight updates and causes the gradient descent to diverge.
   - Symptom: weights become NaN or very large (i.e. overflow)
   - Solution
     - Gradient Clipping: clip the gradients during backprop so that they never exceed some threshold
     - Batch Normalisation
     - Weight Decay/Regularization
     - LSTM (if applicable as in if it's sequential data)

# 8 Week 8 Deep Neural Networks

## 8.1 Convolutional Neural Networks (CNN)

**Definition 8.1. Context**

1. Assumption: inputs are images, which allow to encode image properties into the architecture
2. Motivation: Regular fully-connected Neural Networks such as MLP don't scale well to larger images. For example, a 200x200x3 image would lead to 120,000 weights for EACH neuron in a hidden layer. There will be even more weights given there are multiple neurons, which is expensive and can cause overfitting.

**Definition 8.2. CNN Architecture**

Take CIFAR-10 as example

1. Input Layer
   (a) Hold raw pixels of an input image, with dimension [32x32x3]. That is, width 32, height 32, and with three color channels R,G,B.
2. Conv Layer (CONV)
   (a) Definition: for each neuron in the conv layer, calculate the dot product between weights (i.e. filter) and the local region each neuron connected to in the previous layer
   (b) Input Volume (light pink square): 32x32x3
   (c) Filter (dark pink square)
   - Dimension of each filter: WxHxD = 5x5x3, where depth D is always equal to the depth of input volumes (D=3)
   - Number of filters: K = 5
   - Local connectivity: each neuron in the output volume only connects to a small local region of the input volume. Such connectivity is only local spatial-wise (WxH) but always full depth-wise (D).
   - Parameter sharing
     – Assumption: similar types of features exist widely in the image
     – Purpose: reduce the number of learning parameters
     – each neuron in the same depth slice of output volume (blue slice) shares the same weights (5x5x3) and bias (1). Sliding/Convolving a single filter (5x5x3) over the input volume produces a depth slice (i.e. output neurons) in the output volume.

   (d) Output Volume:
   - Depth (K): equal to the number of filters (K=5), where each filter looks for different things such as various edges, or blobs of color
   - Stride (S): defines the way we slide the filter. Stride = 1 means that we move the filters one pixel at a time. A large stride produces smaller spatial dimension of the output volume.
   - Padding (P): pad the input volume with zeros around the border such that we can control the spatial dimension of the output volumes (e.g. the same as previous layer)
   - Calculate spatial dimension (WxH)
   $$\frac{W - F + 2P}{S} + 1$$

   (e) Learnable paramaters: Since paramaters are shared for each depth slice, per filter there are FxFxD weights and 1 bias, while in total there are FxFxDxK weights and K biases.
   (f) Hyperparameters: K, F, S, P

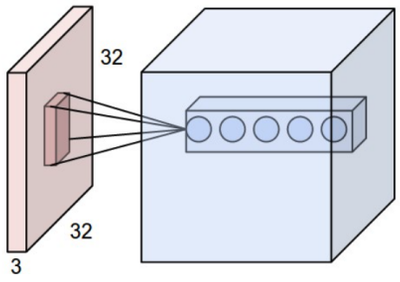



Figure 13: Input and Output Volume (can be considered as conv layer)

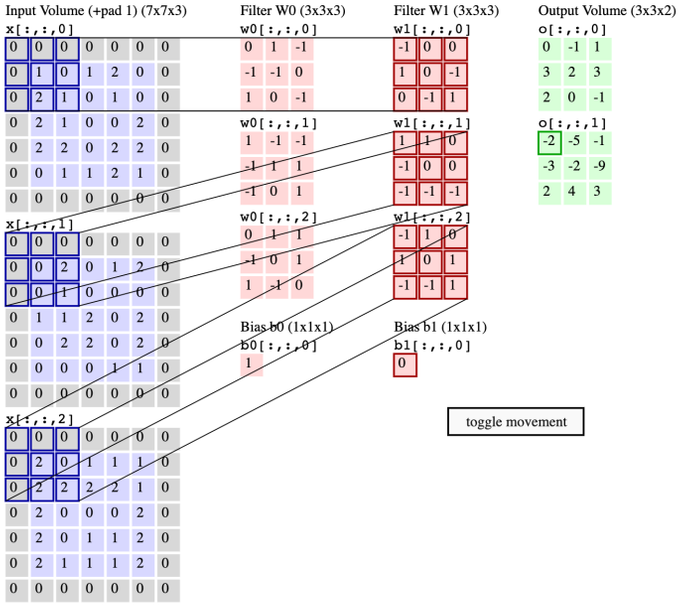



Figure 14: Calculating Output Volume of Conv Layer

3. ReLU Layer (RELU)
   (a) Definition: apply element-wise activation function to the dot product
   (b) Parameters (i.e. those learnable ones in the model): N/A, simply apply fixed activation functions
   (c) Hyperparameters: N/A
4. Pooling Layer (POOL)
   (a) Definition: downsample the 3D volume of neurons along the spatial dimension (i.e. weight and height). It reduces the number of parameters and can reduce overfitting.
   (b) Position: inserted in-between conv layers
   (c) Parameters (i.e. learnable ones in the model): N/A, simply apply fixed pooling function
   (d) Hyperparameters: S, F, Pooling Types
5. Flattening Layer
   (a) Definition: flattening the 3D output volume of neurons from the previous layer into an vector array, which can then be fed into the fully-connected layer
   (b) Position: between pool layer and fc layer
6. Fully Connected Layer (FC)
   (a) Definition: compute the class scores/class probabilities. Note each neuron in this layer is fully connected to all neurons in the previous layer. Size [1x1x10]
   (b) Paramaters: weights and biases

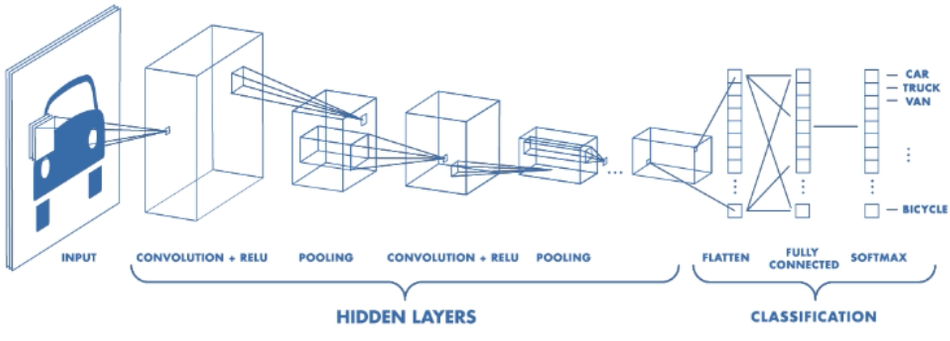



Figure 15: CNN General Architecture

**Definition 8.3. Similarity and Difference Between MLP and CNN**

1. Similarity between CNN and Fully-connected NN (e.g. MLP)
   - made up of neurons with learnable weights and biases
   - each neuron perform dot product of input and optionally follows a non-linearity
   - whole network still receives raw image pixels from one end and output class scores on the other end
   - tricks developed in NN still applies
2. Difference between CNN and Fully-connected NN (e.g. MLP)
   (a) 3D volume of neurons. In CNN, each layer transforms a 3D input volume to a 3D output volume of neurons. While in MLP, neurons are all organised in a 2D manner.
   (b) Local connectivity. In CNN, the neuron in a layer will only be connected to a small local region of a previous layer, instead of being fully-connected in MLP.
   (c) Parameter sharing. Neurons in each depth slice of the output volume are related as they shares the same weight and bias as other neurons in the same slice. By constrast, in MLP, neurons in the output layer are independent and do not share weights.

## 8.2 Recurrent Neural Networks (RNN)

**Definition 8.4. RNN**

1. Goal: model sequential data
2. Motivation: traditional NN such as MLP doesn't satisfy the 4 modelling criteria of sequential data
3. Four criteria for modelling sequence data
   (a) Weight Sharing
   (b) Track long-term dependencies
   (c) Handle varying length of input sequence
   (d) Maintain order information in sequence
   - Food was good, not bad at all
   - Food was bad, not good at all
4. Solution: RNN with the network in a loop design, which allows information to be passed from one time step to the next
5. Suitable data: sequential data
6. Forward Pass
   (a) **Update Hidden State**. Apply the recurrence relation at every time step to process a sequence and update the hidden state
   $$\begin{aligned} h_t &= f_W(h_{t-1}, x_t) \\ &= tanh(W_{hh}h_{t-1}, W_{xh}x_t) \end{aligned}$$
   where
   - $h_{t-1}$: previous hidden state
   - $x_t$: current input vector
   - $f_W$: a function parametrised with W, which is essentially activation function tanh with weights $W_{hh}$ and $W_{xh}$

   (b) **Output Prediction**. Output prediction vector for the current time step
   $$\hat{y}_t = W_{hy}h_t$$
   where
   - $h_t$: current hidden state (after update)

   (c) **Calculate Loss**. Calculate the loss $L_t$ at each time step. The final loss $L$ is a sum of all individual losses over all time steps.

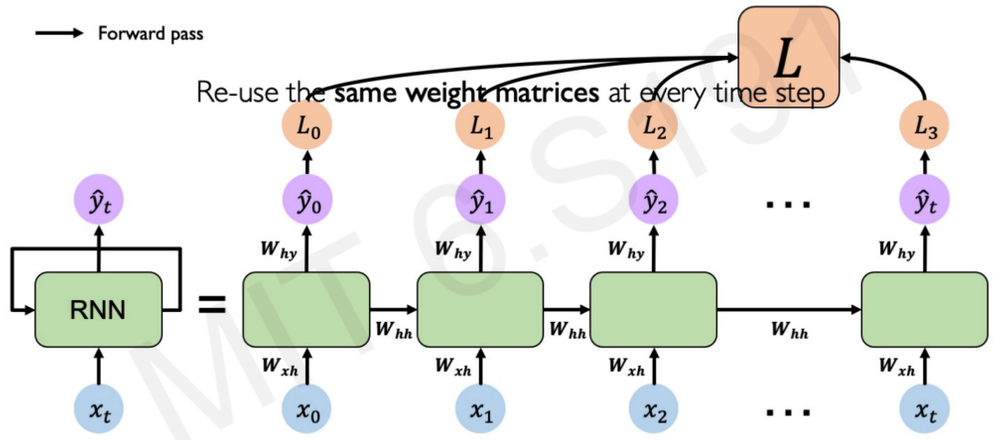



Figure 16: RNN and its unfold version

7. Backward pass

- Backpropagation through time (BPTT)
- Calculate $\frac{\partial L}{\partial W_{hh}}$, $\frac{\partial L}{\partial W_{xh}}$, $\frac{\partial L}{\partial W_{hy}}$

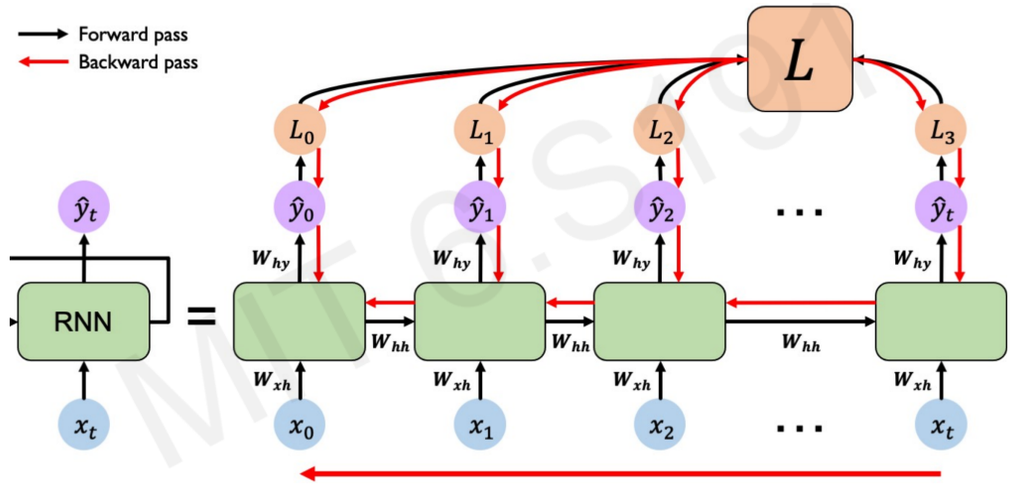



Figure 17: Backprop Through Time

8. Application:

- Image captioning (One to Many)
- Sentiment classification (Many to One)
- Machine translation (Many to Many)
- Music generation (Many to Many)

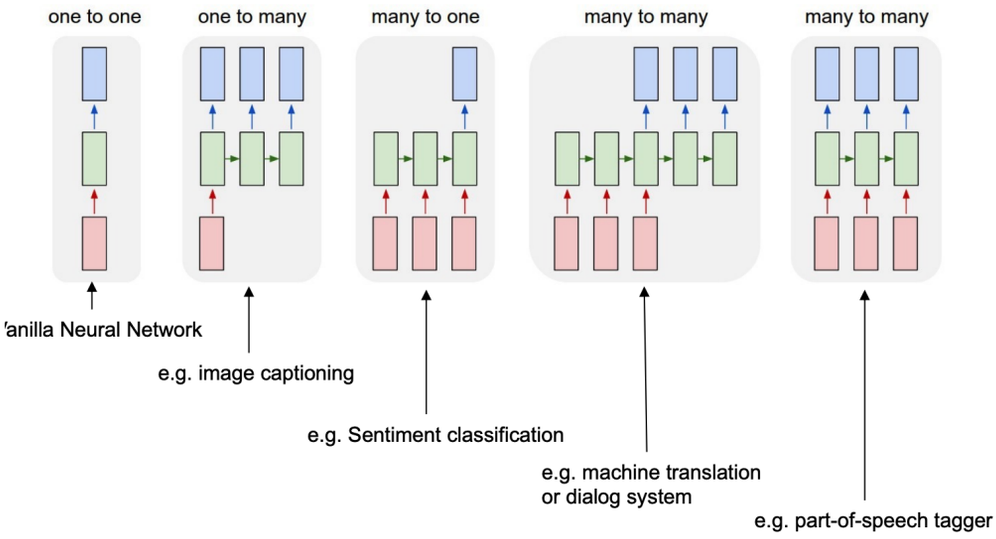



Figure 18: RNN Variants and their Applications

**Definition 8.5. LSTM**

1. Motivation:
   (a) Simple RNN cannot ensure long-term dependency when gap is too large
      - Small gap is ok: "the cloud is in the _" where key context word cloud is close.
      - Large gap is not ok: "I grew up in France with two sisters and one dog and I can speak _". France is too far away from french.
   (b) many repeated gradient computation ends up leading to vanishing or exploding gradient
      - value $> 1$, exploding gradient. Solution: gradient clipping
      - Value $< 1$, vanishing gradient. Solution: use less saturating activation function, weight initialisation or use better network.
2. Solution: use gated cell, where recurrent cell with gate is used to control the information flow. Both LSTM and GRU are gated cells and LSTM is the most widely used one.
3. LSTM Goal: explicitly designed for solving the long-term dependency issue of simple RNN; keep track of important information through gating over all time steps
4. LSTM Architecture
   (a) Cell State $C_{t-1}$ to $C_t$
      - Similar to conveyor belt where information flows from one end to the other. Information can be removed from or added to the cell state through gates.

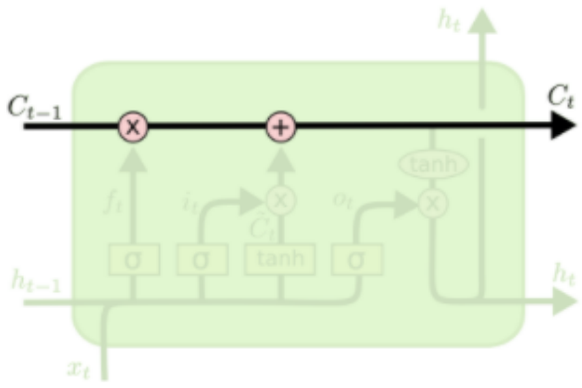



Figure 19: Cell State

   (b) Gate Definition
      - Literally like a gate, designed to *optimally* let information through.

- Two components:
  - a sigmoid NN layer: 0 - nothing through; 1 - everything through
  - a pointwise multiplication operation
- 3 gates in total to "gate" the cell state

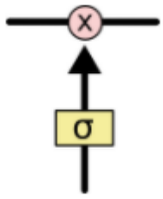
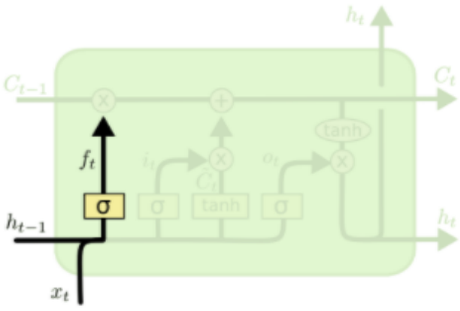



Figure 20: Gate

(c) Forget Gate: $f_t * C_{t-1}$

- Definition: Decide what information to **forget** in the cell state
- Procedure: sigmoid layer output 0 or 1 for each element in the cell state vector which will then be multiplied to the cell state vector to filter the information.

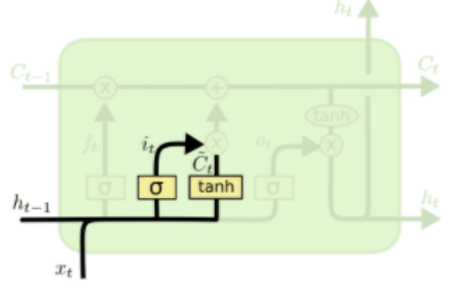



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figure 21: Forget Gate

(d) Input Gate: $i_t * \tilde{C}_t$

- Definition: Decide what new information to **store** in the cell state
- Procedure:
  i. Sigmoid layer outputs 0 or 1, which decides what tanh output to keep
  ii. Tanh layer helps regulate the value (i.e. between -1 and 1) and output a new candidate vector $\tilde{C}_t$
- Example: we want to add the gender information of a new subject to the cell state, which will replace the gender information of the previous subject as it is less relevant for prediction.

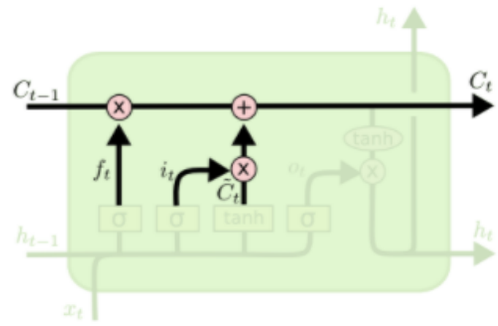



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Figure 22: Input Gate

(e) Update Cell State: $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$

- Definition: to actually implement the forget gate and input gate. That is, to remove or add information to the cell state and update the cell state

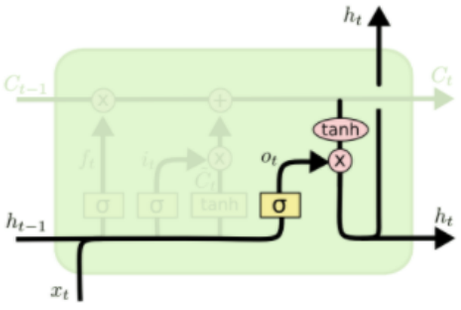



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 23: Update Cell State

(f) Output Gate: output $h_t$

- Definition: Decide what new information to **store** in the cell state
- Procedure:
  i. Sigmoid layer (receives input from $x_t$ and $h_{t-1}$) and outputs 0 or 1, which decides what information from the updated cell state we will output
  ii. Tanh layer helps regulate the updated cell state value (i.e. between -1 and 1)
  iii. Filter the tanh output by multiplying it with sigmoid output
- Example: we want to add the gender information of a new subject to the cell state, which will replace the gender information of the previous subject as it is less relevant for prediction.




$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 24: Output Gate

5. Similarity between Simple RNN and LSTM
   (a) Both have the repeating module design
6. Difference between Simple RNN and LSTM
   (a) The module structure is more complex for LSTM. LSTM has 4 NN layers (yellow blocks) while Simple RNN only has 1 NN layer (yellow blocks - activation layer: tanh)

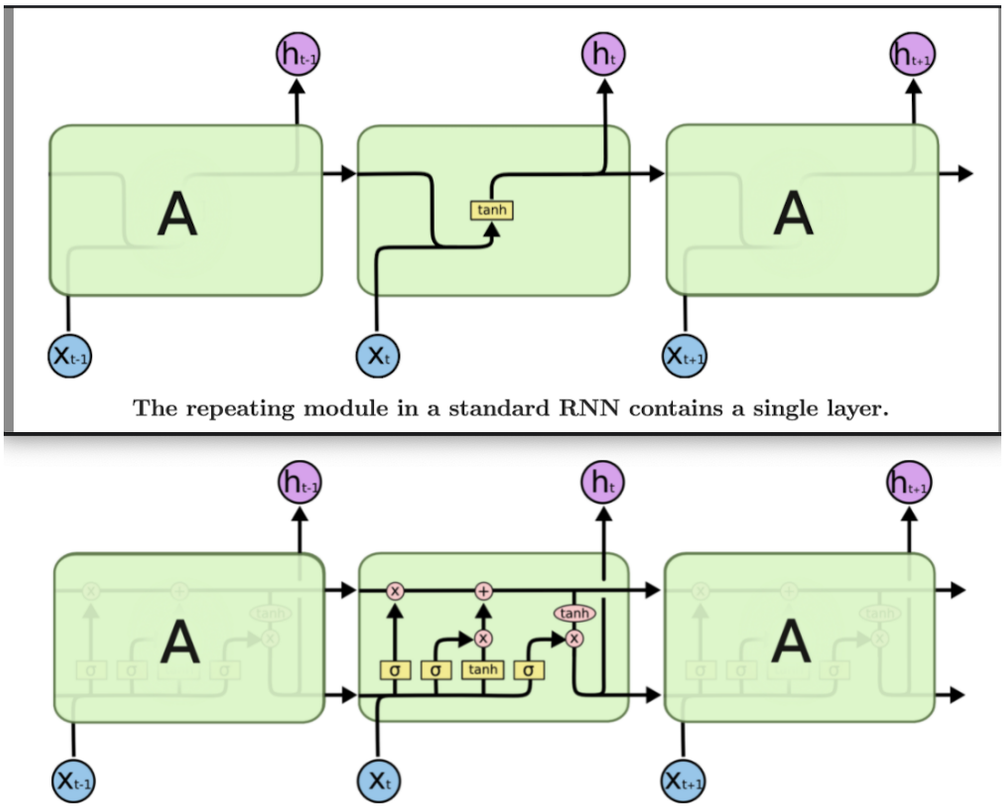
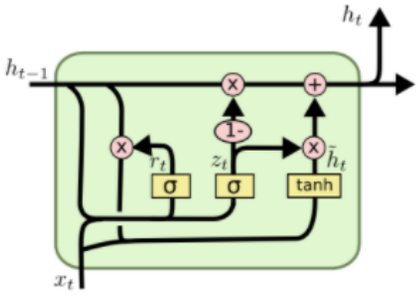



Figure 25: RNN Variants and their Applications

**Definition 8.6. GRU**

1. Definition: Gated Recurrrent Unit (GRU), which is a simpler version of gated cell model compared to LSTM
2. Key modifications:
   (a) Update gate
   (b) Reset gate



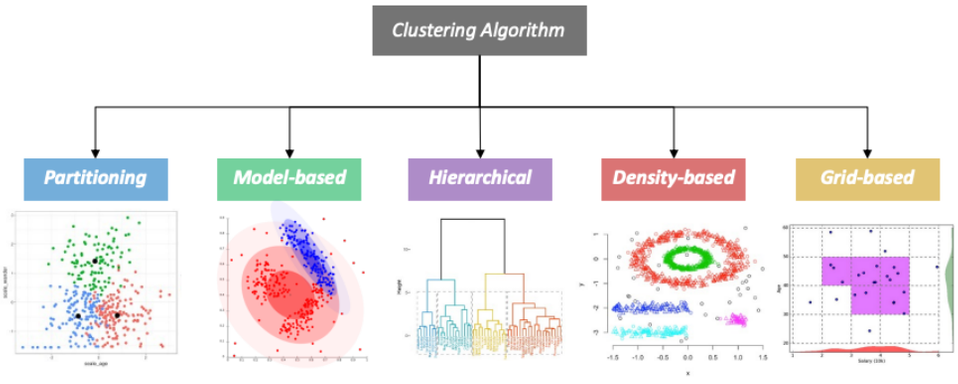

$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 26: GRU

# 9 Week 9 Clustering 1

**Definition 9.1. Clustering Context**

1. Type: unsupervised learning
2. Definition: group data into clusters such that the data points within the same cluster are *similar* and the data points from different clusters are different.
3. Good Clustering: high similarity within the same cluster and high dissimilarity across different clusters
4. Similarity measure (either distance or similarity)
   (a) Euclidean distance $l_2$
   (b) Manhattan distance $l_1$
   (c) Minkowski distance $l_p$
   (d) Cosine similarity (i.e. $cos\theta$).
5. 5 Types of Clustering Approaches
   (a) Partitioning clustering (i.e. K-Mean)
   (b) Model-based clustering (i.e. GMM with EM)
   (c) Hierachical clustering (i.e. Agglomerative and Divisive clustering)
   (d) Density-based clustering (i.e. DBSCAN)
   (e) Grid-based Clustering (i.e. STING and CLIQUE)




6. Application:
   - Image discretization
   - Image segmentation

## 9.1 Partitioning Clustering: K-Mean Clustering

**Definition 9.2. Partitioning Clustering: K-Mean Clustering**

1. **Partitioning Clustering**
   (a) Definition: *partition* a set of data points into non-overlapping subsets called clusters such that each data point is in exactly one subset
2. **K-Mean Clustering**
   (a) Definition:
      i. partitioning clustering algorithm
      ii. aim to find a prespecified number of clusters (i.e. K)
   (b) Centroid: mean of all data points in a cluster, which is not an actual data point.
   (c) Specific requirement:
      - specify $K$ in advance

- data points are continuous

(d) Algorithms:

i. Specify the number of clusters $K$
ii. Randomly select $K$ points as the initial centroids for each cluster
iii. For each point, calculate its distance to each centroid and assign it to the closest cluster
iv. After the reassignment of all points, recompute the cluster centroids (i.e. cluster mean)
v. Repeat step 3 and 4 (i.e. point reallocation and distance recomputation) until stopping condition satisfies

(e) Stopping condition

i. Minimum reallocation of data points
ii. Minimum change of cluster centroids
iii. Minimum decrease of SSE (aka sum of squared distances) where individual error/loss is defined as the distance between point and centroid

(f) Similarity measure: what we've mentioned, l1, l2, cosine similarity. K-Mean converge for most similarity measure.

(g) Time Complexity:

- $O(n * d * K * I)$ where $I$ is the number of iterations, $K$ is the number of clusters, $n$ is the number of data points and $d$ is the number of attributes.
- Reasoning: $n * d$ data points, $K$ centroids to calculate distance to each iteration, $I$ iterations

(h) Remarks

- Most converges happen early

3. **Bisecting K-Means**

(a) Type: an extension of basic K-Means

(b) Goal: obtain $K$ clusters

(c) Procedure:

- Short: split all data points into two clusters and then select one cluster to split until $K$ clusters are produced.
- Long: select one cluster based on some criterion, perform multiple bisection trials and select the bisection with lowest SSE. Then reselect a cluster based on some criterion, repeat the above steps until $K$ clusters are obtained.

(d) Pros: suffer less from the random initialisation problem. Two reasons. Firstly, it always performs several trials and select the bisection with least SSE. Secondly, every bisection only involves two centroids (less than $K$ which is more spread out) to initialise for.

4. **Strengths of K-Means (including Bisecting K-Means)**

(a) simple and efficient

(b) suitable for different data types

(c) Bisecting K-means suffers less from random initialisation problems

5. **Weakness of K-Means (including Bisecting K-Means)**

(a) Centroid Initialization

- Problem: Centroid initialisation is random. Bad initialisation can lead to suboptimal clusterings with high SSEs.
- Solutions:
  i. perform multiple runs and select the clustering with the least SSE
  ii. K-Mean++
    – select the initial centroid randomly
    – for each successive centroid, points farthest from the current centroid are most likely to be selected
    – Pros: ensure centroids are well separated

(b) Empty Cluster

- Problem: Sometimes no points are allocated to a cluster and this leads to a higher SSE (some other points in other clusters can be the centroid of empty cluster and its associated error could have been excluded)
- Solution: find a replacement centroid
  i. Global: Find a replacement point with global maximum error among all clusters
  ii. Local: Find a replacement point with local maximum error in the cluster with highest SSE

(c) Outliers

- Problem: the existence of outliers can degrade centroids selected (e.g. using K-Mean++ we can select outlier as centroid), which therefore causes the SSE to be higher than it should be
- Solution: remove outliers, either before clustering (know for sure that won't cluster well), or after clustering (after multiple runs, identify those which consistently contribute a lot to SSE)

(d) Update centroid incrementally (BOOK IS WRONG, NOT AN ISSUE, more like a centroid update scheme)

- Definition: centroid can be updated incremenally after each point assignment
- Pros: no empty cluster
- Cons: order dependency (cluster is dependent on the order of points being processed); more costly

(e) Clusters with different sizes, densities and non-spherical shapes.

- Problem: K-means have difficulty in detecting clusters with different sizes, densities and non-spherical shapes. The suboptimal clustering often splits the clusters into subclusters.
- Solution: choose other algorithms such as DBSCAN; still working if accept subclusters.

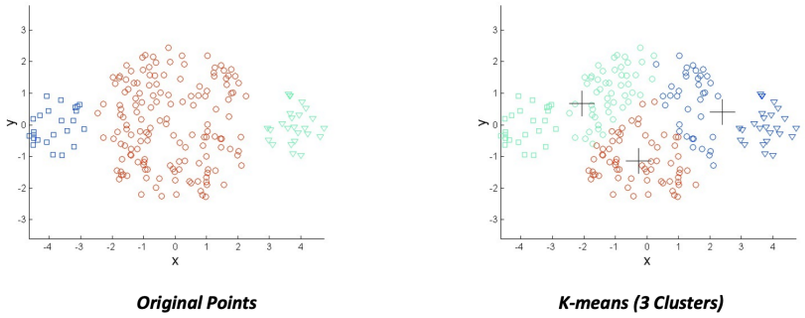



Figure 27: Different Sizes

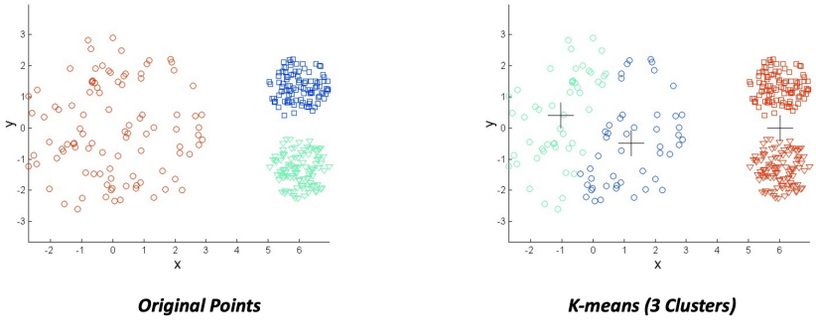



Figure 28: Different Densities

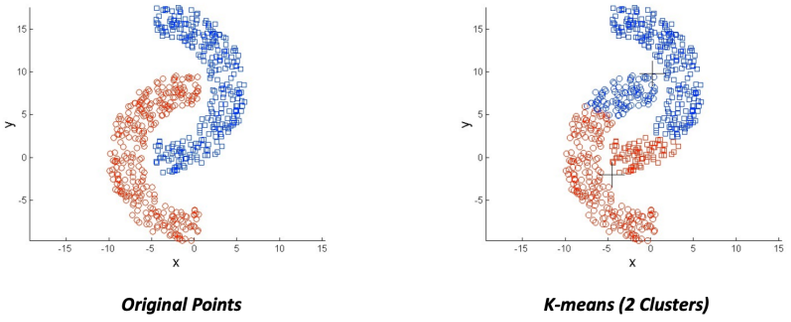



Figure 29: Non-spherical Shapes

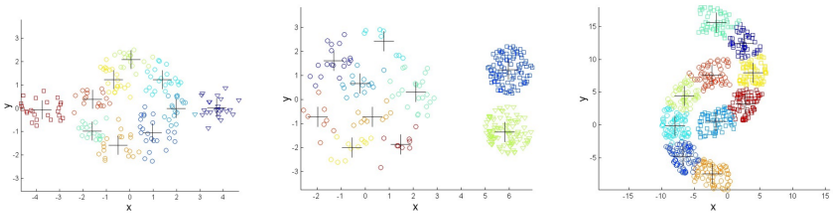


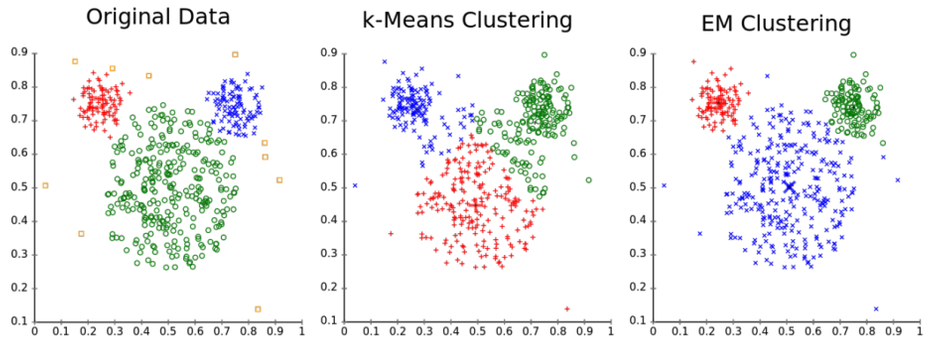
Figure 30: Subclusters

## 9.2 Model-based Clustering: GMM with EM

**Definition 9.3. Model-based Clustering: GMM with EM**

1. *Gaussian* Mixture Model (GMM)
   (a) Definition:
      i. Model type: A probabilistic model
      ii. Data generation process: Assumes that all the data points are generated from a mixture of Multivariate Gaussian distributions with unknown parameters.
   (b) Assumption:
      i. Data generating process is a mixture of multivariate normals
   (c) Remarks:
      - All observations from cluster $k$ are drawn from multivariate normal distribution $\mathcal{N}(\mu_k, \Sigma_k)$, where $\mu_k, \Sigma_k$ are mean vector and covariate matrix.
      - $\pi_k$ is the weight of each distribution, often calculated by $\frac{N_k}{N}$. The higher the weight, the larger percentage of observations from distribution $k$
   (d) Expectation-Maximisation Algorithm (EM Algorithm)
      i. Goal: estimate parameters for GMM
      ii. Procedure in short: make an initial guess for the parameters, and then iteratively improves these estimates
      iii. Procedure in long:
         A. Initialise model parameters $\Theta_k = (\mu_k, \Sigma_k, \pi_k)$ in whatever way
         B. For each point $x_n$, repeat two steps
            - Expectation-Step: calculate $\gamma(z_{nk})$, the probability $x_n$ belongs to distribution $k$
            $$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)}{\sum_{i=1}^{K} \pi_i \mathcal{N}(x_n|\mu_i, \Sigma_i)}$$
            where $\pi_i$, $\mu_i$ and $\Sigma_i$ for each cluster $i$ are obtained first through initialisation then updated by maximisation step. So in this step, we have to calculate $n * k$ probabilities
            - Maximisation-Step: update the parameter estimates which maximises the expected likelihood. For each distribution/cluster, we select all relevant $n$ probabilities and use them to weight each data point for calculation.
            $$\mu_k \leftarrow \frac{\sum_{n=1}^{N} \gamma(z_{nk}) x_n}{N_k}$$
            $$\Sigma_k \leftarrow \frac{\sum_{n=1}^{N} \gamma(z_{nk})(x_n - \mu_k)(x_n - \mu_k)^T}{N_k}$$
            $$\pi_k \leftarrow \frac{N_k}{N}$$
         C. stop until parameters does not change much
   (e) Difference between GMM and K-Means
      i. **Model Type**: K-mean is partitioning clustering and GMM is model-based clustering
      ii. **Point Assignment**: K-mean assigns each data point to a cluster based on its distance to the centroid, which is deterministic; GMM assigns each data point to the most probable distribution it belongs to, which is probabilistic
      iii. **Data Assumption**: K-mean does not assume data follows Gaussian distribution while GMM does
      iv. **Capability**: K-mean is less effective than GMM in handling complex clusters (e.g. clusters with different sizes)

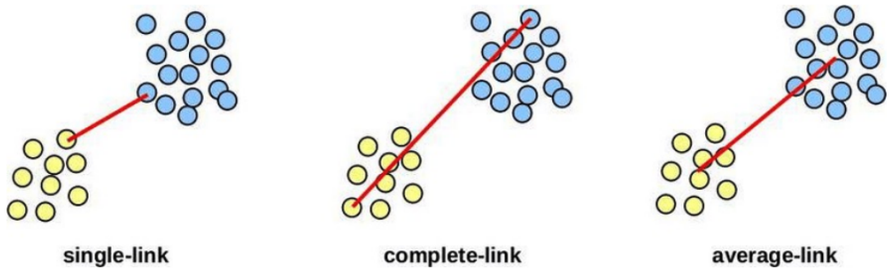
v. **Converge**: K-mean converges faster than GMM




## 9.3 Hierarchical Clustering: Agglomerative and Divisive Clustering

**Definition 9.4. Hierarchical Clustering: Agglomerative and Divisive Clustering**

1. Hierachical Clustering
   (a) Definition: produces a set of nested clusters organised as a hierachical tree
   (b) Two main types:
      i. **Agglomerative Clustering (bottom up)** (more popular)
      ii. Divisive Clustering (top down)
   (c) Pros:
      i. Visualisation: clustering structure can be visualised as dendrogram
      ii. Number of clusters: no need to assume the number of clusters as any number of clusters can be obtained by cutting dendrogram at certain level
      iii. Taxonomy: good for applications with hierarchies, such as creating taxonomy
2. Agglomerative Clustering
   (a) Definition: Bottom up. Start with $n$ clusters at bottom where each data point forms a cluster. At each step, merge closest pair of clusters (through distance matrix) until there is one cluster.
   (b) Requirement: **Distance matrix** for clusters
      i. Definition: define the distance between clusters instead of points
      ii. Three types
         A. Single link: smallest distance
         B. Complete link: largest distance
         C. Average link: average distance
      iii. As for the actual distance calculation, still need to decide. E.g. l1 or l2 or correlation (for time series)




   (c) Procedure
      i. Assign each data point to its own cluster
      ii. Calculate the distance matrix (up to what type we choose) between clusters
      iii. Merge the two clusters with smallest distance and update the distance matrix. If one step there are multiple pairs with the same smallest distance, for those non-overlapping pairs we merge all those pairs in one step, for those overlapping pairs pick a random one to merge.
      iv. Stop until there is a single cluster
   (d) Calculating Example: p66

(e) Pros:

i. Good for applications with hierarchies, such as creating taxonomy

ii. No need to assume the number of clusters as any number of clusters can be obtained by cutting dendrogram at certain level

(f) Cons:

i. Optimality: All merging decisions are locally optimal but not globally optimal. When data are noisy and high-dimensional (e.g. document data), such local-optimal optimisation can end up having bad clustering result.

ii. Cost: High computational and storage cost

3. Divisive Clustering

(a) Definition: Top down. Start with one cluster on the top level. At each step, split each cluster until there are $n$ singleton cluster where each cluster only contains one data point.

(b) Implementation: MST, which produces the same clustering as single link agglomerative clustering

(c) Procedure

i. Compute MST for the graph

ii. Keep cutting off the longest edge until each cluster contains one data point

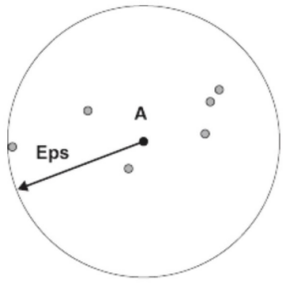
# 10 Week 10 Clustering 2

## 10.1 Density-based Clustering: DBSCAN

**Definition 10.1. Density-based Clustering: DBSCAN**

**DBSCAN**

1. Definition: a clustering algorithm that separates high-density regions from low-density regions
2. Some Relevant Terms
   (a) $\epsilon$: the radius of the neighborhood
   (b) $MinPts$: minimum number of points required within the $\epsilon$-neighborhood
   (c) Density: the number of data points within the $\epsilon$-neighborhood of some point $A$, including $A$ itself. For example, density for A's neighborhood is 7




3. Three types of points based on density
   (a) Core points
   - Definition and Condition
     i. at least $MinPts$ points within the $\epsilon$-neighborhood
   - Example: if $MinPts = 7$, then A is a core point as $Density(A) = 7 \geq MinPts$
   - Action: connect all core points with edges that are within $\epsilon$-neighborhoods. All connected core points will be in the same cluster.

   (b) Border points
   - Definition and Condition
     i. NOT a core point (i.e. $< MinPts$ points within the $\epsilon$-neighborhood)
     ii. but falls within the $\epsilon$-neighborhood of a core point
   - Example: if $MinPts = 7$, then B is NOT a core point as $Density(B) = 4 < MinPts$ but B is within A's $\epsilon$-neighborhood
   - Action: assign it to a cluster of its associated core point. For example, assign B to the cluster A associates with.
   - Remarks: literally the border of a cluster as no points are density reachable from the border points (end of the connected chain)

   (c) Noise points
   - Definition and Condition
     i. Not a core point AND not a border point
   - Example: if $MinPts = 7$, then C is not a core point as $Density(C) = 3 < MinPts$ and C is not in neighborhoods of core points.
   - Action: discard

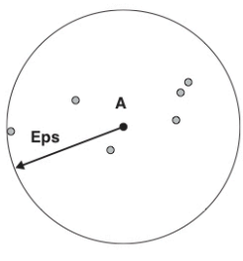



**Figure 8.20.** Center-based density.

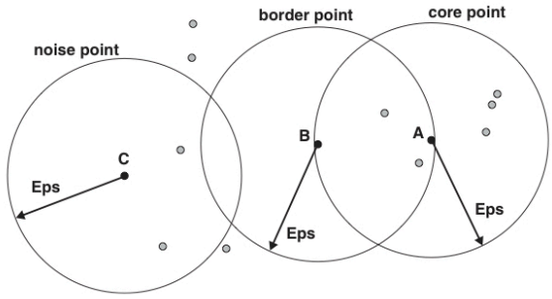


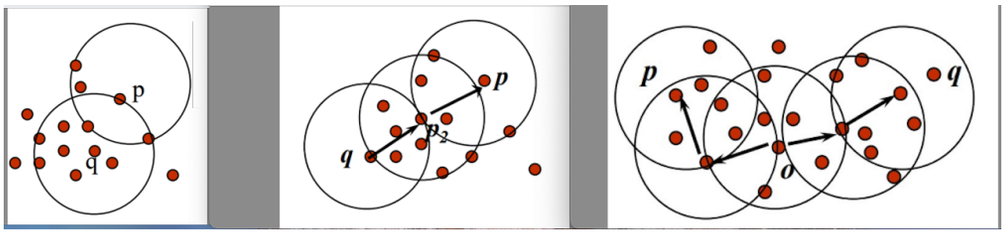

**Figure 8.21.** Core, border, and noise points.

4. Three Connectivity Properties

   (a) Directly Density-Reachable: p is directly density reachable from q if q is a **core point** and p is within the $\epsilon$-neighborhood of q

   (b) Density-Reachable: p is density reachable from q if all adjacent points during the middle are directly reachable. For example, p2 is directly reachable from q, p is directly reachable from p2 (inaccurate graph from slides)

   (c) Density-Connected: p and q are density connected if p and q are both density reachable from some middle point o.




5. DBSCAN Algorithm

   (a) Label each point as core, border or noise point.

   (b) For noise points, discard them all

   (c) For core points, put an edge between them if the distance is within $\epsilon$ (i.e. connect all density reachable points). For those edge-connected core points, group them into separate clusters

   (d) For border points, assign them to their associated core point's cluster

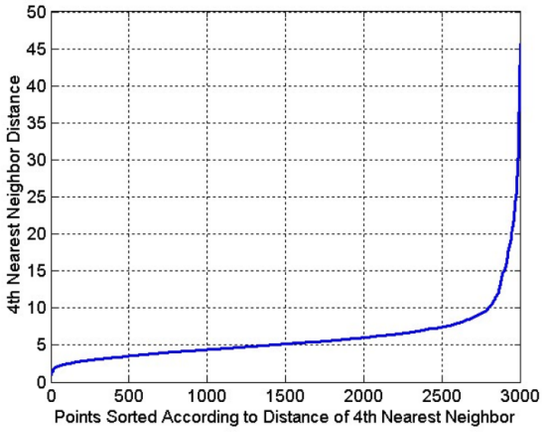
6. DBSCAN Complexity: $O(n \log n)$ with spatial index (e.g. kd-tree), otherwise $O(n^2)$ (worst case, for each point need to look for n points in the neighborhood)

7. Select $\epsilon$ and $MinPts$

   (a) Tool: k-dist graph, where k-dist is the distance to the k-th closest point.

   (b) Assumption: for points in a cluster, the k-dist should be similar or small; for noise points, the k-dist should be large.

   (c) Procedure: pick some k and calculate k-dist for all points. Plot the total number of points fall within all k-dist. The $\epsilon$ corresponds to the elbow point is the one to pick. For example, in below figure, given k=4, the best $\epsilon$ should be around 10.




8. Pros:

   (a) Resistant to noise points (discard)

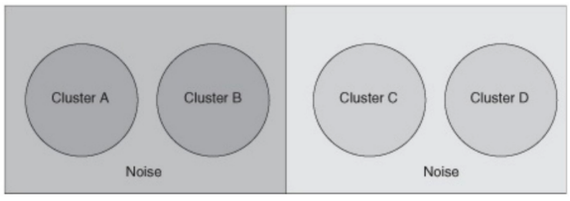
   (b) Handle clusters with arbitrary shapes and size better than K-Means (last figure)

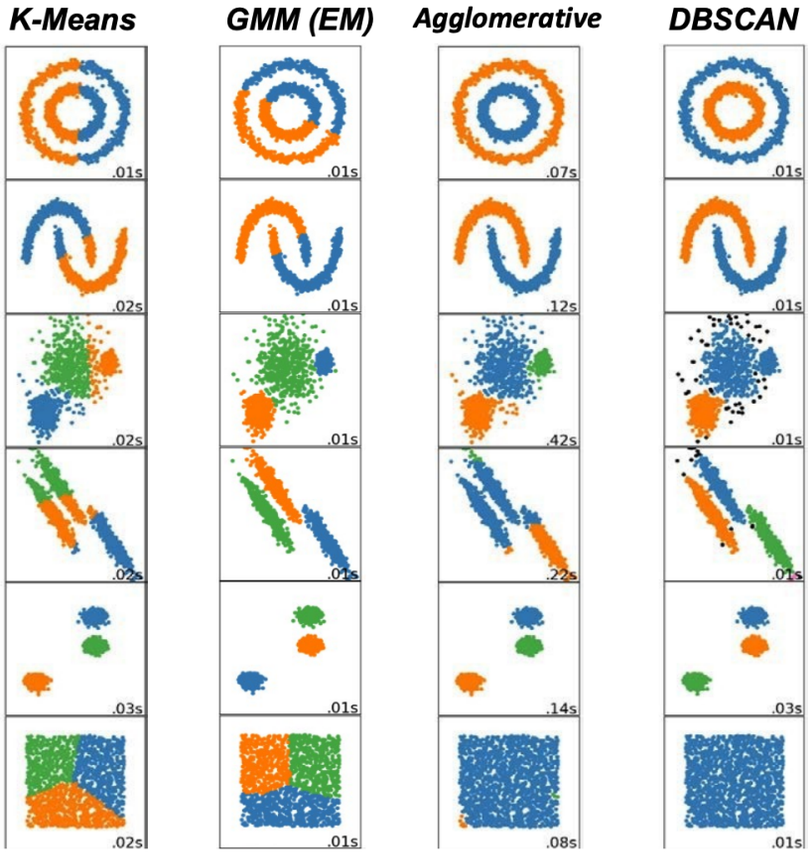
9. Cons:

   (a) Difficulty with clusters with widely varying densities (2-ball graphs)

   (b) Difficulty with high-dimensional data

(c) Costly to compute pairwise distance between points, especially for high-dimensional data




## 10.2 Grid-based Clustering: CLIQUE and STING

- Definition: breaks the data space into grid cells and then forms clusters from cells that are sufficiently dense
- Two types: CLIQUE and STING
- Cons:
  - Cell size and shape: he rectangular grid cells do not accurately capture the density of the circular boundary area
  - Curse of dimensionality: the number of potential grid cells increases rapidly when dimension increases and it would be hard to cluster high-dimensional data

## 10.3 Evaluation of Clustering Outcomes

**Definition 10.2. Evaluation of Clustering Outcomes**

1. External Measures:
   - Matching-based measures
     (a) Purity:
       - Definition: purity for the whole clustering
       - Example: Each cluster's purity is the majority parition percentage. Total purity is the weighted sum of individual purity.

     (b) Maximum Matching:

– Definition: Treat the clustering evaluation as a max matching problem and our goal is to maximise the edge weight sum of a matching
– For example, each number of the cell can be considered as an edge weight. We need to first find all matchings between C and T and then calculate $W(M) = \frac{\sum_e W(e)}{n}$. The match score with the maximum weight sum ratio $W(M)$ is selected

| $C \backslash T$ | $T_1$ | $T_2$ | $T_3$ | Sum |
|---|---|---|---|---|
| $C_1$ | 0 | 20 | 30 | 50 |
| $C_2$ | 0 | 20 | 5 | 25 |
| $C_3$ | 25 | 0 | 0 | 25 |
| $m_j$ | 25 | 40 | 35 | 100 |

(c) F-measures
– Precision: same formula as purity where individual purity = majority partition/cluster size
– Recall: Majority partition/total size of that partition
– F-measure: harmonic mean of precision and recall. Calculate individual F score then average them.

- Entropy-based measures
  (a) NMI: normalised shared info between clustering C and partition T
- Pairwise measures
  (a) TP: two data points x1 and x2 have the same partition ($y$) and the same cluster labels ($\hat{y}$)
  (b) TN, FP, FN

# 11 Week 12 Markov Model

1. Markov Model
2. Markov Chain
3. Hidden Markov Model (HMM): class of probabilistic model that allow us to predict a sequence of hidden events from a set of observed events

# 12 Week 12 Reinforcement Learning

**Definition 12.1. Reinforcement Learning**

1. Definition: one kind of machine learning paradigms where an agent in an environment aims to learn the best policy to follow (i.e. to guide its actions) through trial-and-error in order to maximise its expected cumulative rewards.
2. Elements
   (a) Policy: determines the agent's behavior
   (b) Reward: a function of ($s$, $a$). Immediate and short-term reward. Agent adjusts its policy depending on the reward it receives (e.g. too low this time, next time select other action)
   (c) Value function: the expected cumulative reward since the current state. Long-term and over the entire lifetime of the agent. Need to be estimated.
   (d) Model: a model of the environment, which mimics the behavior of the environment (i.e. given action, provide reward and new state). Markov Decision Process is a type of model. Depending on whether model is used, we have model-based methods and model-free methods.
3. Difference from other two ML paradigms (data, goal)
   - Supervised Learning
     - Data: each example is correctly labelled.
     - Goal: learn the correct mapping between example and its correct label that generalises over unseen examples
   - Unsupervised Learning
     - Data: examples without labels
     - Goal: learn hidden structure in the data
   - Reinforcement Learning
     - Data: state-action pairs
     - Goal: maximise the expected cumulative rewards/future rewards
     - Learn from interactions

## 12.1 Markov Decision Process (MDP)

1. Definition: an extension of Markov Process or Markov Chains, where there are more than one actions and there is reward associated with each action.
2. Purpose: used to model the RL environment
3. Assumption: Markov Property (since MARKOV decision process). The current state $s'$ only depends on the previous state $s$ and the previous action $a$.
4. Notations
   - $\mathcal{S}$: a set of all possible states
   - $\mathcal{A}$: a set of all possible actions
   - $r_t$: immediate reward at time step $t$ to reward the action $a_{t-1}$
   - $\pi$: agent's policy
     - for each state $s$, $\pi(|s)$ is a probability distribution over action. $\pi(a_{11}|s_1)$, $\pi(a_{12}|s_1)$ and $\pi(a_{13}|s_1)$ are probabilities over three possible actions under $s_1$ and they sum to 1. The probability distribution $\pi(|s)$ is specific to the state $s$ and varies with $s$.
     - $\pi(a_t|s_t)$ represents that if the agent follows $\pi$ as a policy, the agent will have $\pi(a_t|s_t)$ probability to take an action $a_t$ given state $s_t$
   - $\pi^*$: agent's optimal policy
   $$\pi^* = \text{argmax}_\pi \mathbb{E}[\sum_{t\geq 0} \gamma^t r_t|\pi] \tag{1}$$
   - $\gamma$: discount factor
     - obtain the present value of all rewards; small=short-sighted; large=long-sighted
5. Interacting Procedure:
   (a) At some time step $t$, the agent receives a state $S_t = s_t$ (from the environment) and a reward $R_t = r_t$ for its previous action. The agent interacts with the state by making an action $A_t = a_t$.

(b) At next time step $t+1$, based on $s_t$ and $a_t$, the environment sends the new reward $r_t$ (the lecture notation) as well as the new state $s_{t+1}$ for the agent to interact with.

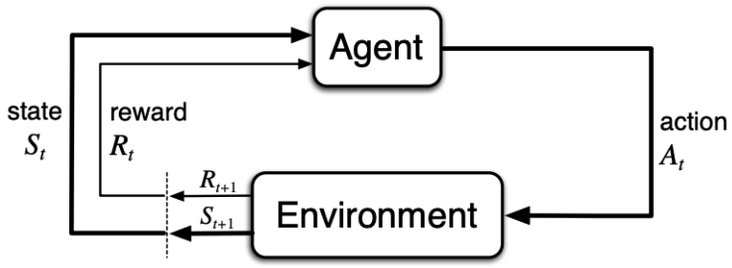



**Figure 3.1:** The agent–environment interaction in a Markov decision process.

6. Distributions
   (a) Initial state $s_0$
      - follows some random distribution
      - $s_0 \sim P(s_0)$
   (b) Action $a_t$
      - only depends on the current state $s_t$. NOT dependent on the reward as $r_t$ is irrelevant and only for previous action $a_{t-1}$. $r_t$ is obtained only after the action $a_t$.
      - Perform action according to the policy (i.e. probability distribution) which is state specific $\pi(a_t|s_t)$
   (c) Reward $r_t$ (this notation is different from the book but consistent with 5318 and 5328 lectures)
      - depend on $s_t$ and $a_t$. It's for the action $a_t$ in previous time step
      - $P(r_t|s_t, a_t)$
   (d) Next state $s_{t+1}$
      - depends on $s_t$ and $a_t$ (e.g. robot rides bike example)
      - $P(s_{t+1}|s_t, a_t)$, which is also a transition probability
7. Trajectory
   - by following a policy, produces state-action-reward sequences
   - $s_0, a_0, r_0, s_1, a_1, r_1, ....$
8. Episodic Vs. Continuing Tasks
   (a) Episodic Tasks: the agent-environment interaction can be broken down into different subsequences called episodes. For example, each new round of the Pong game is an episode, and the final time step of an episode occurs when a player scores a point. The environment then will be reset to some standard starting state or to a random sample from a distribution of possible starting states.
   (b) Continuing Tasks: the agent-environment interactions don't break up naturally into episodes, but instead continue without limit. If we don't discount future rewards, the agent will keep doing the task as the reward goes to infinity.
9. Discounted Expected Accumulative Reward (i.e. Expected Accumulative Reward)
   - to overcome the infinite accumulative reward problem from continuing tasks. Makes the expected accumulative reward converge to a finite number even the rewards are infinite
   - makes agent to focus more on immediate reward than future rewards

## 12.2 Q-Learning

**Definition 12.2. Value function**

1. Definition: given a state $s$ or a state-action pair ($s$, $a$), if the agent starts to follow the policy $\pi$ *from now on* (i.e. the policy does not influence $s$ or ($s$,$a$)), then the output of the value function is the expected cumulative reward.
2. Application: use to evaluate the expected cumulative reward of scenario when the agent starts to follow the policy $\pi$ from now on after being given state $s$ or given state-action pair ($s$, $a$)
3. **State-value function** $V^{\pi}(s)$

- Input: state $s$ only
- $V^{\pi}(s) = \mathbb{E}[\sum_{t\geq 0}\gamma^t r_t|s,\pi]$, where $s \sim P(S_{t+1}|s_t,a_t)$, $r_t \sim P(R_t|s_t,a_t)$ and $\gamma \in (0,1)$
- Interpretation: given state $s$, if the agent follows $\pi$ from now on, it can receive $V^{\pi}(s)$ expected cumulative reward
- Application: use to evaluate how good state $s$ is
- **Optimal V-value** $V^*(s)$: for a given state $s$, the optimal state-value function gives the largest expected cumulative reward under any policy $\pi$

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$

4. **Action-value function** or **Q-value Function** $Q^{\pi}(s,a)$
   - Input: state $s$ and action $a$
   - $Q^{\pi}(s,a) = \mathbb{E}[\sum_{t\geq 0}\gamma^t r_t|s,a,\pi]$, where $s \sim P(S_{t+1}|s_t,a_t)$, $a \sim P(A_t|s_t)$, $r_t \sim P(R_t|s_t,a_t)$ and $\gamma \in (0,1)$
   - Interpretation: given state $s$ and action $a$, if the agent follows $\pi$ from now on, it can receive $Q^{\pi}(s,a)$ expected cumulative reward
   - Application: use to evaluate how good the state-action pair $s$ and $a$ are
   - **Optimal Q-value** $Q^*(s,a)$: for a given state-action pair $(s,a)$, the optimal action-value function gives the largest expected cumulative reward under any policy $\pi$

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

**Definition 12.3. Policy**

1. Definition: policy $\pi$ is a function that maps a given state $s$ to a probability of taking each action $a$ that is possible to that state. For each state $s$, $\pi(|s)$ is a probability distribution over action and the probability distribution is specific to the state.
2. Example: $\pi(a_t|s_t)$ represents that at time step $t$, given state $s_t$, if the agent follows $\pi$ as a policy/function, then agent will have $\pi(a_t|s_t)$ probability to take an action $a_t$
3. $\pi^*$: agent's optimal policy
4. Difference between Policy and Value Function
   - Policy cares about the probabilities of each action. Following some policy $\pi$, given some state $s$, the policy $\pi(|s)$ evaluates how **probable** it is for an agent to take any action that is possible given $s$.
   - Value function cares about the expected cumulative rewards. Following some policy $\pi$, the function evaluates how **valuable** the state (if V-function) or state-action pair (if Q-function) is for an agent
5. Optimal Policy
   - Definition: Given any state $s$, optimal policy gives the largest (could tie) V-value (i.e. expected cumulative reward) among all other policies.

     That is, $\forall s \in \mathcal{S}$, we have $\pi \geq \pi' \Leftrightarrow V^{\pi}(s) \geq V^{\pi'}(s)$, which is equivalent to

$$\begin{aligned}\pi^* &= \text{argmax}_{\pi} V^{\pi}(s) \\ &= \text{argmax}_{\pi}\mathbb{E}[\sum_{t\geq 0}\gamma^t r_t|s,\pi]\end{aligned}$$

**Definition 12.4. Bellman Optimality Equation**

One property of the optimal action-value function $Q^*(s,a)$ is that it has to satisfy **Bellman Optimality Equation**, which can be defined as

$$Q^*(s,a) = r + \gamma\mathbb{E}_{s'\sim\mathcal{S}}[\max_{a'} Q^*(s',a')|s,a]$$

where the $r$ is the expected immediate reward for $r_t$ (rewarding current state-action $(s,a)$. That is, $\mathbb{E}(r_t) = r$.

It states that for any state-action pair (s,a) (i.e. $s_t$ and $a_t$ to be exact) at time t, the optimal Q-value $Q^*(s,a)$ we can get from such pair can be considered as the sum of two following components

1. the expected value of immediate reward $r$ (i.e. $\mathbb{E}(r_t) = r$ is a constant) for taking action $a$ in state $s$ at the current time $t$
2. the expected value of maximum discounted optimal Q-value from any next state-action pair $(s',a')$, We can't control $s'$ as it is given by the environment following some distribution $S$, but we can pick the action $a'$ that maximises the Q-value.

**Definition 12.5. Q-Learning**

1. Goal: find optimal Q-function $Q^*$ or Optimal Q-value $Q^*(s,a)$ for any given state-action pair. Afterwards, find the optimal policy
2. Philosophy: use the Bellman optimality equation to iteratively update Q-value for each state-action pair and eventually make every Q(s,a) converges to $Q^*(s,a)$. Once all estimated optimal Q-values are obtained, the optimal policy is also ready as we can simply lookup from the Q-table.
3. Value Iteration
   - Motivation: we want to know the optimal Q-function $Q^*$ such that given any state $s$ we can know what action $a$ to take that comes with the highest $Q^*$. Even though $Q^*$ satisfies Bellman Equation, we actually can't obtain $Q^*$ directly. Both LHS and RHS of Bellman Equation depends on the unknown $Q^*$ and the only reliable number we have is the immediate reward.
   - Goal: Estimate unknown $Q^*(s,a)$ iteratively
   - Solution: use **value iteration** method. Use the Bellman equation to iteratively updates the Q-values for each state-action pair until every $Q(s,a)$ converges to the optimal Q-function $Q^*(s,a)$.
   - Update Rule:

$$\begin{aligned} Q(s,a) &\leftarrow r + \gamma \mathbb{E}_{s' \sim \mathcal{S}}[\max_{a'} Q(s',a')] \\ &\leftarrow Q(s,a) + \mathbb{E}_{s' \sim \mathcal{S}}[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \\ &\leftarrow Q(s,a) + \{\mathbb{E}_{s' \sim \mathcal{S}}[r + \gamma \max_{a'} Q(s',a')] - Q(s,a)\} \\ &\leftarrow Q(s,a) + \{Q^*(s,a) - Q(s,a)\} \text{ where Q(s,a) is known but not the } Q^*(s,a) \end{aligned}$$

     Empirically, it can be written as

$$\begin{aligned} &\leftarrow Q(s,a) + \{\hat{Q}^*(s,a) - Q(s,a)\} \\ &\leftarrow Q(s,a) + \eta\{[r + \gamma \max_{a'} Q(s',a')] - Q(s,a)\} \\ Q_{i+1}(s,a) &\leftarrow Q_i(s,a) + \eta\{[r + \gamma \max_{a'} Q_i(s',a')] - Q_i(s,a)\} \\ &\leftarrow (1-\eta) Q_i(s,a) + \eta[r + \gamma \max_{a'} Q_i(s',a')] \end{aligned}$$

   - Q-table:
     - Definition: a look-up table used to update and track q-values for each state-action pair
     - Each table entry is the estimation of $Q^*(s,a)$ and we iteratively update the estimation to make it more accurate

     Some remarks:
     - Key equation:

$$\begin{aligned} Q_{i+1}(s,a) &\leftarrow Q_i(s,a) + \eta\{[r + \gamma \max_{a'} Q_i(s',a')] - Q_i(s,a)\} \\ &\leftarrow (1-\eta) Q_i(s,a) + \eta[r + \gamma \max_{a'} Q_i(s',a')] \end{aligned}$$

     - The difference between Q can be regarded as loss. Ideally, after many iterations, when the Q(s,a) is close enough to $Q^*(s,a)$, the loss will converge to 0 or equivalently value update difference will be very small between iterations. That's how we know we can stop updating and take it as optimal value.
     - The updated Q value in the (i+1) iteration is the weighted sum of old Q-value $Q_i(s,a)$ and **learned Q-value or target** $[r + \gamma \max_{a'} Q_i(s',a')]$. We don't want to discard the old value as there may be useful information to take from. By using the learning rate $\eta$ and gradually decay it, we can adjust how much old information to keep as we learn more and more about the environment.
     - when $i$ is sufficiently large, the Q value would converge to the optimal one.

**Definition 12.6. Q-learning Algorithm and Sarsa Algorithm**

1. Both are temporal difference methods
2. Target Policy vs Behavior Policy
   - Motivation: All learning control methods face a dilemma: They seek to learn action values conditional on subsequent optimal behavior, but they need to behave non-optimally in order to explore all actions to find the optimal actions. How can they learn about the optimal policy while behaving according to an exploratory policy?
   - One Solution: use two policies

  - Target Policy:
    * Update Q-value with the new chosen action but not update action
    * Obtain action through Q (i.e. argmaxQ)
    * Exploitation
    * Target policy is used as optimal policy
    * Example: **Q-Learning**, **Sarsa**
  - Behavior Policy:
    * Updates action after every state such that it can generate more behavior to explore around
    * Exploration
    * Behavior policy is not used as optimal policy
    * Example: **Sarsa**

3. On-Policy vs Off-Policy

- On-policy
  - Target policy is consistent with behavior policy
  - Example: **Sarsa**, which is both target and bahavior policy. Sarsa is not only a target policy by updating the Q-value with the new chosen action $a'$, but also a behavior policy by updating the new action $a'$ as next action.
- Off-policy
  - target policy is inconsistent with behavior policy
  - Example: **Q-Learning**, which is target policy only. It only updates the Q-value function with the newly chosen action $a$ from argmax, but it does not update the action as next action.

**Algorithm 1** Q-Learning

Initialize the Q value $Q(s, a)$ for all state-action pairs arbitrarily
**for** *each episode* **do**
  Initialize $s$
  **for** $s$ *is not a terminal state* **do**
    Choose $a = \arg\max_a Q(s, a)$
    Substitute action $a$ and $s$ into the environment and observe $r$, $s'$
    $Q(s, a) \leftarrow Q(s, a) + \eta\left(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right)$
    $s \leftarrow s'$ //only update the state but not action
  **end**
**end**

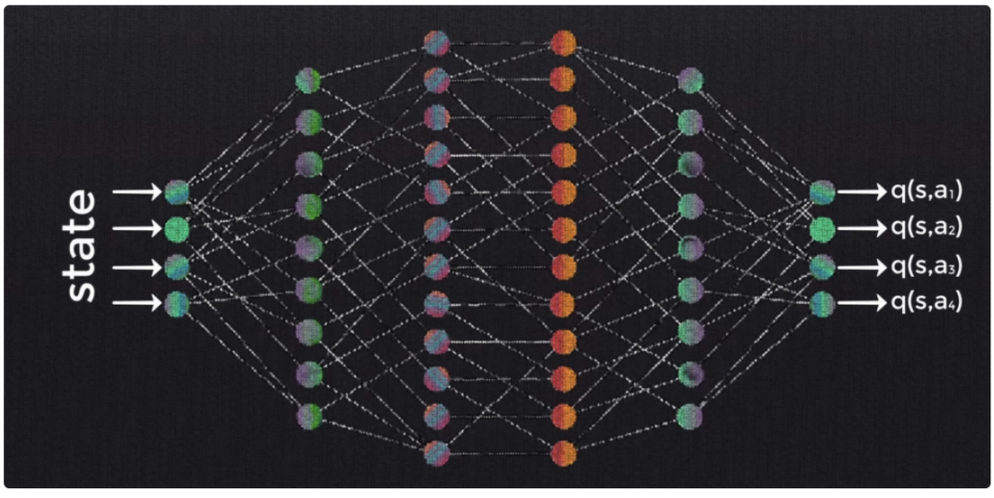
**Algorithm 2** Sarsa

Initialize the Q value $Q(s, a)$ for all state-action pairs arbitrarily
**for** *each episode* **do**
  Initialize $s$
  Choose $a = \arg\max_a Q(s, a)$
  **for** $s$ *is not a terminal state* **do**
    Substitute action $a$ and $s$ into the environment and observe $r$, $s'$
    Choose $a' = \arg\max_{a'} Q(s', a')$
    $Q(s, a) \leftarrow Q(s, a) + \eta\left(r + \gamma Q(s', a') - Q(s, a)\right)$
    $s \leftarrow s'$
    $a \leftarrow a'$ //action $a$ is also updated
  **end**
**end**

## 12.3 Deep Q-Learning

**Definition 12.7. Deep Q-Networks (DQN)**

1. Definition: a deep neural network (DNN) that approximates a Q-function
2. Motivation: some tasks have continuous state-action pairs (e.g. games) and it would be infeasible and inefficient to compute, store and update all those pairs in a Q-table
3. Solution: rather than using value iteration method to iteratively calculate the optimal Q-value and then obtain the optimal Q-function, we can use deep neural network (DNN) as a function approximation to estimate the optimal Q-function (Recall that a 3-layer DNN can approximate almost any functions). For example, $Q(s, a, w) \approx Q^*(s, a)$.
4. Input: a stack of preprocessed consecutive frames. This is because a single frame is not enough for the network to fully understand the state and more frames provide more information.
5. Output: Q-value for each possible action $a_i$ for the given state (the output layer is a fully connected layer. No softmax or any activation function as we want the raw Q-values)
6. The loss
$$L(w) = (r + \gamma \max_{a'} Q_w(s', a') - Q_w(s, a))^2$$
7. Update the weight through GD and BP.




**Definition 12.8. DQN with Experience Replay and Prioritised Experience Replay**

1. Motivation: learning from consecutive training examples (i.e. state-action pairs) can be problematic as there may be strong temporal relationship between those examples. DNN can capture them and then degrades the performance.
2. Approaches
   (a) **DQN with Experience Replay**
   - Solution: randomly sample examples from the **experience bank** for training.
   - Procedure

1. Initialize replay memory capacity.
2. Initialize the network with random weights.
3. *For each episode:*
    1. Initialize the starting state.
    2. *For each time step:*
        1. Select an action.
            - *Via exploration or exploitation*
        2. Execute selected action in an emulator.
        3. Observe reward and next state.
        4. Store experience in replay memory.
        5. Sample random batch from replay memory.
        6. Preprocess states from batch.
        7. Pass batch of preprocessed states to policy network.
        8. Calculate loss between output Q-values and target Q-values.
            - Requires a second pass to the network for the next state
        9. Gradient descent updates weights in the policy network to minimize loss.

   - Loss calculation: For each sample, two forward passes of states. First pass with state $s$. second pass with next state $s'$ to obtain max term in the RHS of Bellman Equation. Then calculate the loss between the output $Q(s)$

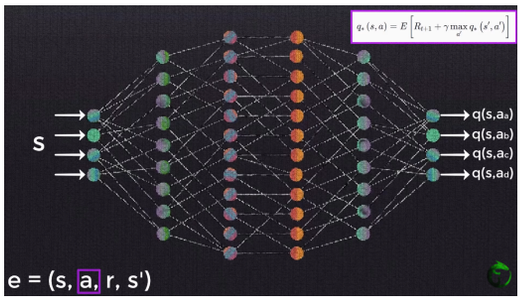



Figure 31: First Forward Pass

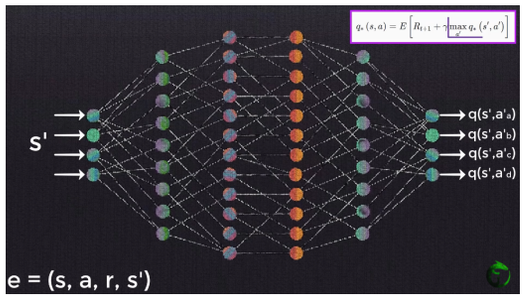



Figure 32: Second Forward Pass

- Pros: breaks the temporal correlation between consecutive samples and therefore makes the learning more effective.
- Cons: random sampling ignores the importance (i.e. update contribution) of individual samples

**Algorithm 1** Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
**for** episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  **for** $t = 1, T$ **do**
    With probability $\epsilon$ select a random action $a_t$ $\epsilon$-greedy
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$ experience
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to
  **end for** update w. loss
**end for**

From the *Tutorial: Deep Reinforcement Learning* by David Silver, Google DeepMind

(b) **DQN with Prioritised Experience Replay**

- Solution: weight example accordingly to their DNN weight update contribution
- Pros: take the sample update contribution into consideration and more contributing samples are more likely to be sampled
- Procedure: store experiences in a priority queue according to their error
  - DNN weight update is positively correlated with the loss value $([r + \gamma \max_{a'} Q_w(s', a')] - Q_w(s, a))^2$
  - example with a larger loss also has a larger DNN weight update and should be "prioritised" with higher assigned weight in sampling. In such a way they are more likely to be sampled.

  correlated with error

**Algorithm 3** DQN with Prioritised Experience Replay (simplified)

Initialize the Q value $Q(s, a)$ for all state-action pairs arbitrarily
Initialize experience bank $B = \emptyset$
Observe $s_0$
**for** $t = 1$ *to* $T$ **do**
  Choose $a = \arg\max_a Q_w(s_{i-1}, a)$
  Observe $s_t$, $r_t$ and $\gamma_t$
  Store the transition $(s_{t-1}, a_{t-1}, r_t, \gamma_t, s_t)$ in $B$ with maximal priority
  **if** $t \equiv 0 \mod K$ *to* **then**
    Sample a transition $(s, a, r, \gamma, s', a')$ according to the priority
    Compute error $\delta = (r + \gamma \max_{a'} Q_w(s', a') - Q_w(s, a))^2$
    Update the priority of current transition according to $|\delta|$
    Update weights $w \leftarrow w + \eta \nabla (r + \gamma \max_{a'} Q_w(s', a') - Q_w(s, a))^2$
  **end**
**end**

3. Notations and Terminology

- Experience: $e_t = (s, a, r, s') = (s_t, a_t, r_t, s_{t+1})$ represents a summary of agent's experience at time $t$
- Experience Replay: a technique

- Experience Bank: also called replay memory, is a database where agent's experiences are stored
- N: in practice only last N experiences are stored in the experience bank

**Definition 12.9. Target Network**

1. Purpose: deal with non-stationary learning targets (i.e. the learned Q-value $[\gamma + max_{a'}Q(s', a')]$) and improve the learning stability
2. Procedure: Fixed the parameter $\hat{w}$ for target network $\hat{w}$

**Definition 12.10. Double DQN**

1. Motivation: The max operator in DQN uses the same network to select and evaluate, which makes it more likely to overestimate target value (i.e. learnt Q-value) and create biased estimation.
2. Solution: decouple the selection from evaluation. Current Q-network is used to select actions and the older Q-network is used to evaluate actions.

**Definition 12.11. Policy Gradient**

1. Motivation: Q-function can be really complicated especially given high-dimensional state (i.e. images). Hard to learn every state-action pair for such scenario.
2. Solution: Instead of learn state-action pair in Q-learning, learn policy directly. Use DNN to model policy $\pi_\theta$.
3. Solve for the Gradient Estimator $\nabla\hat{J}(\theta)$
   (a) Expected cumulative reward for policy $\theta$ is $J(\theta)$ and we want to find the optimal policy $\theta^*$ that maximises $J(\theta)$.
   $$J(\theta) = \mathbb{E}[\sum_{t\geq 0} \gamma^t r_t | \pi_\theta]$$
   $$\theta^* = \text{argmax}_\theta J(\theta)$$
   (b) Define trajectory $\tau = (s_0, a_0, r_0, ...)$ and $r(\tau)$ as the cumulative reward of trajectory $\tau$, then we can rewrite

$J(\theta)$ as

$$\begin{aligned}
J(\theta) &= \mathbb{E}[\sum_{t\geq 0} \gamma^t r_t | \pi_\theta] \\
&= \mathbb{E}_{\tau \sim p(\tau;\theta)}[r(\tau)] \\
&= \int_{\tau \sim p(\tau;\theta)} r(\tau) p(\tau;\theta)\, d\tau
\end{aligned}$$

$$\begin{aligned}
\nabla_\theta J(\theta) &= \int_{\tau \sim p(\tau;\theta)} r(\tau) \nabla_\theta p(\tau;\theta)\, d\tau \\
&= \int_{\tau \sim p(\tau;\theta)} r(\tau) p(\tau;\theta) \frac{\nabla_\theta p(\tau;\theta)}{p(\tau;\theta)}\, d\tau \\
&= \int_{\tau \sim p(\tau;\theta)} r(\tau) p(\tau;\theta) \nabla_\theta \log p(\tau;\theta)\, d\tau \\
&= \mathbb{E}_{\tau \sim p(\tau;\theta)}[r(\tau) \nabla_\theta \log p(\tau;\theta)]
\end{aligned}$$

Suppose we have $n$ trajectories $\tau^1$, $\tau^2$, ..., $\tau^n$ then by plugging them in

$$\nabla_\theta \hat{J}(\theta) = \frac{1}{n} \sum_{i=1}^{n} r(\tau^i) \nabla_\theta \log p(\tau^i;\theta)$$

$$\begin{aligned}
p(\tau^i;\theta) &= [\pi_\theta(a_0^i|s_0^i) p(s_1^i|s_0^i, a_0^i)] \times [\pi_\theta(a_1^i|s_1^i) p(s_2^i|s_1^i, a_1^i)] ... \\
&= \Pi_{t\geq 0} \pi_\theta(a_t^i|s_t^i) p(s_{t+1}^i|s_t^i, a_t^i) \\
\log p(\tau^i;\theta) &= \sum_{t\geq 0} [\log \pi_\theta(a_t^i|s_t^i) + \log p(s_{t+1}^i|s_t^i, a_t^i)] \\
\nabla_\theta \log p(\tau^i;\theta) &= \nabla_\theta \sum_{t\geq 0} [\log \pi_\theta(a_t^i|s_t^i) + \log p(s_{t+1}^i|s_t^i, a_t^i)] \\
&= \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t^i|s_t^i) + \sum_{t\geq 0} \nabla_\theta \log p(s_{t+1}^i|s_t^i, a_t^i) \\
&= \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t^i|s_t^i) + 0 \\
&= \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t^i|s_t^i) \\
\nabla_\theta \hat{J}(\theta) &= \frac{1}{n} \sum_{i=1}^{n} r(\tau^i) \nabla_\theta \log p(\tau^i;\theta) \\
&= \frac{1}{n} \sum_{i=1}^{n} r(\tau^i) \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t^i|s_t^i) \\
&= \frac{1}{n} \sum_{i=1}^{n} \sum_{t\geq 0} r(\tau^i) \nabla_\theta \log \pi_\theta(a_t^i|s_t^i)
\end{aligned}$$

**Algorithm 4** Policy Gradient (simplified)

Initialize $\theta$ arbitrarily

**for** *each episode* $(s_0, a_0, r_0, \ldots, s_T, a_T, r_T) \sim \pi_\theta$ **do**

  $\Delta\theta \leftarrow 0$

  $r \leftarrow 0$

  **for** $t = 0$ *to* $T$ **do**

    $\Delta\theta \leftarrow \Delta\theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t)$

    $r \leftarrow r + \gamma r_t$

  **end**

  $\theta \leftarrow \theta + \eta\Delta\theta$

**end**

## 12.4 RL Application

1. Alpha Go
2. Self-play
   - Produce a large amount of training data for training DNN
   - Provide auto-curriculum for the agent to learn, from simple to hard opponents