Wang Xiyu

1 Overview

- Week 1-3: Classical AI, search algorithms
  1. Uninformed search
  2. Local search: hill climbing
  3. Informaed search: A*
  4. Adversarial search Minimax
- Week 4-7: Classical ML
  1. Decision trees
  2. Linear/Logistic regression
  3. Kernels and support vector machines
  4. "Classical" unsuperivese learning
- Week 10-12: Modern ML
  1. Neural networks
  2. Deep learning
  3. Sequential data
- Week 13: Misc.

2 AI: Computers Trying to Behave Like Humans

- **PEAS Framework:**
  - **Performance measure:** define "goodness" of a solution
  - **Environment:** define what the agent can and cannot do
  - **Actuators:** outputs
  - **Sensors:** inputs
- Agent function is sufficient.
- Common agent structures (to define an AI agent):
  - Reflex
  - Goal-based
  - Utility-based
  - Learning
  - (Others possible; can mix and match!)
- Exploration vs exploitation

3 Problem Statement

fully observable $\wedge$ deterministic $\wedge$ static $\wedge$ discrete $\implies$ only need to observe once
To solve a prob using search:

- A goal or a set of goals
- a model of the enironment
- a search algorithm

goal formulation -> problem formulation -> search -> execute

1. goal formulation
2. problem formulation, eg. path finding
   - states: nodes representation invariant:: abstract states should correspond to concrete states
   - initial state: starting node
   - goal states/test: dest node
     Goal test: define the goal using a function is_goal
   - actions: move along an edge :: $|actions(state)| \leq branching\_factor$
   - transition model: $f(curr\_state, action) \implies next\_state$
   - action cost function: see edges
3. Important facts:
   - Representation Invariant: ensure that the abstract states correspond to concrete states
   - Goal Test: Goal defined via a function is_goal
   - Action: a set of $action(state)$, $|actions(state)| \leq branching\_factor$
   - Transition model: $f(curr\_state, action) \implies next\_state$

Search

Uninformed search

No information that could guide the seaech: no clue how good a state is

```
create frontier
// create visited // with vsited memory
insert Node(initial_state) to frontier
while frontier is not empty:
    node = frontier.pop()
    if node.state is goal:
        return solution
//  if node.state in visited: // with vsited memory
//     continue
//  visited.add(state)
    for action in actions(node.state):
    next_state = transition(node.state, action)
    frontier.add(Node(next_state))
return failure
```

Different subvariant of tree search uses differen DS for the frontier.

Search Type	**Data Structure for Frontier**
BFS	Queue
DFS	Stack
UCS (Uniform-cost Search)	Priority Queue

Depth limited search

limit the search to depth l
backtrack when the limit is hit.
time complexity: exponential to search depth
space complexity: size of the frontier

```
create frontier
tier = 0
insert Node(initial_state) to frontier
while (!empty(frontier)) && (tier <= limit):
    node = frontier.pop()
    tier++
    if node.state is goal:
        return solution
    for action in actions(node.state):
    next_state = transition(node.state, action)
    frontier.add(Node(next_state))
return failure
```

Iterative deeptening search

search with depth from 0 to inf
return soln when found. Both complete

```
create frontier
tier = 0
insert Node(initial_state) to frontier
while (!empty(frontier)) && (tier <= limit):
    node = frontier.pop()
    tier++
    if node.state is goal:
        return solution
    for action in actions(node.state):
    next_state = transition(node.state, action)
    frontier.add(Node(next_state))
return failure
```

Summary

Name	Time Complexity*	Space Complexity*	Complete?	Optimal?
Breadth-first Search	Exponential	Exponential	Yes	Yes
Uniform-cost Search	Exponential	Exponential	Yes	Yes
Depth-first Search	Exponential	Polynomial	No#	No
Depth-limited Search(DFS)	Exponential	Polynomial	No	No
Depth-limited Search(BFS)	Exponential	Exponential	No	Yes
Iterative Deepening Search(DFS)	Exponential	Exponential	Yes	Yes
Iterative Deepening Search(BFS)	Exponential	Exponential	Yes	Yes

Not complete if not tracking visited nodes, search may stuck in loop before visiting all nodes.
* In terms of some notion of depth/tier

4 Local Search

Systematic search: typically complete and optimal under certain constraints. However intractable sometimes
Local search: typically incomplete and suboptimal, but has anytime property, ie. longer time -> better solution. Able to provide good enough solution under reasonable amount of time.

4.1

1. Start at random position in the state space

2. iteratively move from a state to another neighouuring state vie perturbation or construction
3. solution is the final state

$$\text{State space: all possible configuration} \tag{1}$$

$$\text{Search space: a subset of state space that will be explored} \tag{2}$$

4.1.1 Perturbation search

- Search space: complete candidate solutons
- search step: modification of one or more solution

For example: swap a path with another path

4.1.2 Constructive search

- partial candidate soluton
- extension of one or more solution

For example: path finding

4.2

goal formulation -> problem formulation -> search -> execute

1. goal formulation
2. problem formulation, eg. path finding
   - states: nodes representation invariant:: abstract states MAYNOT directly correspond to concrete states
   - initial state: starting node, a candidate solution
   - goal states/test: dest node [optional]
     Goal test: define the goal using a function $is\_goal\ f(curr\_state, action) \implies next\_state$
   - Successor function: a funciton that generates neighbouring states by applying modifications from the curren tstate. This defines the local search space

4.3 Evaluation function

A math function that assess the quality or desireability of the solution.
Some solutions may be unacceptable but there are some less bad than the others.

4.4 Hill Climbing/ Greedy Local Search

```
curr_state = init_state
while 1:
    best_succ = best(successor(curr_state))
    if (eval(best) <= eval(curr_state))
        return curr_state
    curr_state = best_succ
```

4.5 State space landscape

- Global max:
- Local max:
- shoulder:

5 Heuristics

Admissibility

A heuristic function $h(n)$ is **admissible** if it **never overestimates** the true cost $h^*(n)$ of reaching the goal state. Formally, for all nodes n:

$$h(n) \leq h^*(n)$$

where:

- $h(n)$ is the estimated cost from node n to the goal.
- $h^*(n)$ is the true minimum cost from n to the goal.

An admissible heuristic ensures that an A* search using it will always find an **optimal** solution.

Consistency (Monotonicity)

A heuristic $h(n)$ is **consistent** (or **monotonic**) if, for every node n and every successor node n' reached via action a, the estimated heuristic cost satisfies:

$$h(n) \leq h(n') + c(n, n')$$

where:

- $c(n, n')$ is the actual cost of moving from n to n'.

Consistency implies that the estimated cost of a node never increases by more than the cost of an actual move, ensuring that the total estimated cost $f(n) = g(n) + h(n)$ is always non-decreasing along a path.

Relationship Between Admissibility and Consistency

- If a heuristic is **consistent**, it is also **admissible**.
- If a heuristic is **admissible**, it is **not necessarily consistent**.

Prove: Consistency implies admissibility

Admissibility

$$\forall n \in State, h(n) \leq \sum_{i=start}^{goal} c(i', a, i) \tag{3}$$

Consistency

$$\forall n \in State, \forall a \in Action, h(n) \leq h(n') + c(n', a, n) \tag{4}$$

Proof. Base case: $n_{start} \vdash n_{goal}$, $h(n_{goal}) = h^*(n_{goal})$,

$$h(n_{start}) \leq c(n_{start}, n_{goal}) + h^*(n_{goal}) = h^*(n_{start})$$

Inductive case: suppose n is of k distance away from n_{goal} on the optimal path from n_{start}. From the base case we have,

$$h(n_k) \leq h^*(n_k)$$

$$h(n_{k-1}) \leq h(n_k) + c(n_{k-1}, n_k) \leq h^*(n_k) + c(n_{k-1}, n_k) = h^*(n_{k-1})$$

□

6 Adversarial search

6.1 Classical adversarial games

- Fully observable
- Deterministic
- discrete
- No infinite run
- 2-player zero-sum
- turn taking

termns

- Player: agent
- Turn:
- Move
- End state
- winning conditon
-

6.2 Problem formulation in adversarial search

- states: nodes representation invariant:: abstract states MAYNOT directly correspond to concrete states
- initial state: starting node, a candidate solution
- Terminal State: state the outcome of the game when it terminates
- Utility function: output the value of a state from the perspetive of our agent

6.3 Minimax

In the view of A, A try to maximize the outcome of the game, B will try to minimize A's outcome, as the gam is zero sum

- expend(state) => [a]

```
max_value(state):
    if is_terminal(state): return utility(state)
    v = -∞
    for next_state in expand(state):
        v = max(v, min value for player A in next_state)
    return v

min_value(state):
    if is_terminal(state): return utility(state)
    v = ∞
    for next_state in expand(state):
        v = min(v, max value for player A in next_state)
    return v

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

6.4 Alpha-beta prunning

From the viewpoint of the MAX player:

- α: the value of the best choice for the MAX player so far
- β: the value of the best choice for the MIN player so far

An optimized version of the Minimax algorithm using prunning:

```
max_value(state, α, β):
    if is_terminal(state): return utility(state)
    v = -∞
    for next_state in expand(state):
        v = max(v, min(v, α, β))
    return v

min_value(state, α, β):
    if is_terminal(state): return utility(state)
    v = ∞
```

```
        for next_state in expand(state):
            v = min(v, max(v, α, β))
        return v

    alpha-beta search(state):
        v = max_value(state, -∞, ∞) // initialized α to be -∞, β to ∞
        return action in expand(state) with value v
```

---

7 Learning agent

For problems that the function are difficult to specify, solutions re intractale to compute in general. Typically episodic,

$$DL \subset ML \subset AL$$

7.1 supervised

Learn the mapping input, feedback given -> output, given a dataset, it minimizes the difference betwenn the prediction and the provided correct ans using a leanign algorithm e.g. image ientification

1. Train phase: try minimizing the diff between pred and correct ans given using the training set, resulting in a trainign agen function, known as the model/hypothesis
2. Testing/evaluation phase: using a test set to measure the performance of the model. The performance on unseen data measures the generalization of the model

Task

- Classification: to predict discreate labels or catagories on the input features
- Regression: predict continuous numetical value based on input features

Dataset

$$D = \bigcup_{[1,n]} \{(x^{(i)}, y^{(i)})\}$$

True data generation function

$$y = f^*(x) + \epsilon$$

where $f^*(x)$ is true but unknow, which generates the label from the input features; ϵ is some noise or error term, which account for the randomness or imperfection in the date generating process.
the goal is to find a function that best approximately $f^*(x)$

Hypothesis class

THE set of models or functions that maps from inputs to outputs $h : X \implies Y$ that can be learned by a learing algorithm. Each element of the hypo clas $h \in \mathcal{H}$

LEarnign algorithm

$$A(D_{train}, H_{hypo}) = h(x), h \in \mathcal{H} \approx f^*(x)$$

PErformance measure

$$h(x) \approx f^*(x)$$

$$PM(D_{test}, h \in H_{hypo}) \mapsto$$

Try the hypothesis h on a new set of examples (test data)

Regression: error

$$\text{Absolute error} = |\hat{y} - y| \tag{5}$$

$$\text{Squared error} = (\hat{y} - y)^2 \tag{6}$$

Mean squared error

$$MSE = \frac{1}{N}\sum_{i=0}^{N}(\hat{y}^{(i)} - y^{(i)})^2$$

Mean absolute error

$$MAE = \frac{1}{N}\sum_{i=0}^{N}|\hat{y}^{(i)} - y^{(i)}|$$

where

$$\hat{y}^{(i)} = h(x^{(i)})$$

Accuracy:

$$A = \frac{1}{N}\sum_{i=1}^{n}\mathbb{1}_{\hat{y}^{(i)}=y^{(i)}}$$

confusion matrix

True positive, false positive, false negatve, true negative (2, 1, 3, 4 quadrnt)

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

$$Precision = \frac{TP}{(TP + FP)}$$

maximize if FP is costly

$$Recall = \frac{TP}{(TP + FN)}$$

maximize recall if FN is dangerous

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

7.2 Decision Tree

Greedy, top-down, recursive, use

```
DTL(examples, attributes, default):
    if (examples = ∅): return default
    if (∀e ∈ example, e has the same classification c): return c
    if (attributes = ∅): return mode(examples)

    best = choose_attribute(attributes, examples)
    tree = new decision tree with root test best
    for v_i of best do:
        examples_i = {e|e ∈ examples, e.best = v_i}
        subtree = DTL(examples, attributes \ best, mode(examples))
        tree.add(v_i: subtree)
```

$$f : [\text{attribute vector}] \mapsto Boolean$$

Basically nested if-else

Expresssioveness

Decision trees can express any function of the input attributess. Trivially, Consistent trainign set → Consistent decision tree, but unlikely to generalize to new examples
Each row in the truth table is represented as a path in the decison tree

Size of the hypothesis class

For n boolean attributes, there are n boolean func, n distinct truth tables rach with 2^n rows → 2^{2^n} decision trees

Informativeness

Ideally we want to sekect an attibute that aplits the examples into all positive or all negative.
Entropy: The higher the more random, thus less informative

$$I(P(v_1)...P(v_k)) = -\sum_{i=1}^{k} P(v_i) log_2 P(v_i)$$

For data set contains boolean outputs,

$$I(P(+), P(-)) = -\frac{p}{p+n} log_2 \frac{p}{p+n} - \frac{n}{p+n} log_2 \frac{n}{p+n}$$

where $0 \leq \mathbb{R}_I \leq 1$. However for non-binary variables the entropy can be greater than 1

Information gain

Information gain = entropy of this node - entropy of children nodes

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

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

Decision tree pruning

- By sample size
- by max search depth

7.3 unsupervised

FInd pattern. No feedback given. e,g, group images by char

7.4 Reinforment

Trial and error, reward given based on observation adn action. eg. chess

8 Linear regression

8.1 LR

- Data: N data points: $([(feature\_vector, target)])$

  Regression: given $x \in \mathbb{R}^d$ and no target, find a function that predicts the target $y \in \mathbb{R}$
  the function:

  $$f : \mathbb{R}^d \mapsto \mathbb{R}$$

  Linear Model:

  $$h_w(x) = \sum_{i=1}^{d} w_i x_i = v_w^T x$$

Where w_i are the parameters or weights and $x_0 = 1$ is a dummy variable (always 1), w_0 is the bias. when d = 1, the model is linear

Feature Transformation

Modify the original features of a dataset to make them more suitable for modeling.

- Feature engineering
  - Polynomial features: $z = x^k, k$ is the polynomial degree.
  - log feature: $z = \log(x)$
  - Exp. feature: $z = e^x$
- Feature scale

  - Min-max scaling: $z_i = \frac{x_i - min(x_i)}{max(x_i) - min(x - i)}$, scales to [0, 1]
  - standardization: $z_i = \frac{x_i - \mu_i}{\sigma_i}$, transformed data has mean of 0 and SD of 1
  - robust scaling (not in syl)
- Feature encoding (not in syl)

Measruing Fit

Loss function (MSE of the for N samples):

$$J_{MSE}(w) = \frac{1}{2N} \sum_{i=1}^{N} (h_w(x^{(i)}) - y^{(i)})^2 \tag{7}$$

Since polynomial regression can be converted into linear regression, by theorem 7.2, the MSE loss function on linear/poly nomial regression is convex

8.2 Normal Equation

1. Take the partial derivative of the linear model

$$\frac{\partial(h_w(x^{(i)}))}{\partial w_j} = \frac{\partial(w^T x^{(i)})}{\partial w_j} = x_j^{(i)} \tag{8}$$

2. Take the par. deriv. of each term in the sum, with respect to w_0 and w_1

$$\frac{\partial((h_w(x^{(i)}) - y)^2)}{\partial(w_j)} = 2(h_w(x^{(i)}) - y)x_j^{(i)} \text{ (chain rule)} \tag{9}$$

3. hence

$$\frac{\partial}{\partial w_j} J_{MSE}(w) = \frac{1}{2N} \frac{\partial}{\partial w_j} \sum_{i=1}^{N} (h_w(x^{(i)}) - y)^2 = \frac{1}{N} \sum_{i=1}^{N} (h_w(x^{(i)}) - y)x_j^{(i)} \tag{10}$$

   As $h_w(x^{(i)}) = w_0 + w_1(x^{(i)})$ in linear regression,

$$\frac{\partial}{\partial w_0} J_{MSE}(w) = \frac{1}{N} \sum_{i=1}^{N} ((w_0 + w_1 x^{(i)}) - y^{(i)}) \quad (x_0 = 1)$$

$$\frac{\partial}{\partial w_1} J_{MSE}(w) = \frac{1}{N} \sum_{i=1}^{N} ((w_0 + w_1 x^{(i)}) - y^{(i)})(x^{(i)})$$

   In generalized form, where $N > 1$,

$$\frac{\partial}{\partial w_N} \left[\frac{1}{2N} \sum_{i=1}^{N} \left(h_w(x^{(i)}) - y^{(i)} \right)^2 \right] = \frac{1}{N} \sum_{i=1}^{N} (h_w(x^{(i)}) - y^{(i)})(x^{(i)^N})$$

Code example:

```
def gradient_descent_multi_variable(X, y, lr = 1e-5, number_of_epochs = 250):'''
    Approximate bias and weight that gave the best fitting line.
        Parameters
        ----------
            X (np.ndarray) : (m, n) numpy matrix representing feature matrix
            y (np.ndarray) : (m, 1) numpy matrix representing target values
            lr (float) : Learning rate
            number_of_epochs (int) : Number of gradient descent epochs
        Returns
        -------
            bias (float):
                The bias constant
            weights (np.ndarray):
                A (n, 1) numpy matrix that specifies the weight constants.
            loss (list):
                A list where the i-th element denotes the MSE score at i-th epoch.'''
    bias:number = 0
```

```
    weights = np.full((X.shape[1], 1), 0).astype(float)
    loss:List[number] = []
    N:number = X.shape[0]
    pred = X @ weights + bias    # pred:ℝ^{m×1} = X:ℝ^{m×n} × w:ℝ^{n×1}+ bias:number
    for e in range(number_of_epochs):
        pred = X @ weights + bias
        g_w = (1 / N) * (X.T @ (pred - y)) # ∂/∂w1 J_MSE(w) = 1/N X^T((Xw + bias) − y)
        g_b = (1 / N) * np.sum(pred - y) # ∂/∂w0 J_MSE(w) = 1/N Σ_{i=1}^N ((Xw + bias) − y)
        bias -= lr * g_b                 # w0 ← w0 − γ ∂J_MSE(w)/∂w0
        weights -= lr * g_w              # w1 ← w1 − γ ∂J_MSE(w)/∂w1
        loss.append(mean_squared_error(y, pred))
    return bias, weights, loss
```

Normal Equation

find w that minimizes J_{MSE}:

$$\frac{\partial(J_{MSE})}{\partial(w_j)} = \frac{1}{N}\sum_{i=1}^{N}(w^T x^{(i)} - y^{(i)})x_j^{(i)} = 0$$

Where $w = \begin{bmatrix} w_0 \\ w_1 \\ \cdots \\ w_N \end{bmatrix}$, $x = \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_N \end{bmatrix}$, $w^T x$ is just the linear combination of feature vector w on x

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

Suppose invertible

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

Derive Normal equation from MSE

Derive: $w = (X^T X)^{-1} X^T y$ From: $MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})^2$

Proof

We start with the Mean Squared Error (MSE) function:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})^2, \tag{11}$$

where $\hat{y}^{(i)} = h_w(x^{(i)}) = w^T x^{(i)}$, with $w \in \mathbb{R}^{d\times 1}$ and $y \in \mathbb{R}^{N\times 1}$. Expanding in terms of matrix notation, let $A = Xw - y$, where $X \in \mathbb{R}^{N\times d}$ with rows $x^{(i)}$, then:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(w^T x^{(i)} - y^{(i)})^2 = \frac{1}{N}(A^T A). \tag{12}$$

Rewriting:

$$MSE = \frac{1}{N}(Xw - y)^T(Xw - y), \tag{13}$$

which expands further as:

$$MSE = \frac{1}{N}\left(w^T X^T Xw - w^T X^T y - y^T Xw + y^T y\right). \tag{14}$$

Since $w^T X^T Xw$ and $y^T Xw$ are scalars, we simplify:

$$MSE = \frac{1}{N}\left(w^T X^T Xw - 2w^T X^T y + y^T y\right). \tag{15}$$

To minimize MSE, we take the partial derivative with respect to w:

$$\frac{\partial}{\partial w}\left[\frac{1}{N}\left(w^T X^T Xw - 2w^T X^T y + y^T y\right)\right]. \tag{16}$$

Differentiating term by term:

$$\frac{\partial}{\partial w}(w^T X^T Xw) = 2X^T Xw, \tag{17}$$

$$\frac{\partial}{\partial w}(-2w^T X^T y) = -2X^T y, \tag{18}$$

$$\frac{\partial}{\partial w}(y^T y) = 0. \tag{19}$$

Thus,

$$\frac{\partial}{\partial w}(MSE) = \frac{1}{N}(2X^T Xw - 2X^T y). \tag{20}$$

Setting the derivative to zero for minimization:

$$\frac{2}{N}X^T Xw - \frac{2}{N}X^T y = 0. \tag{21}$$

$$X^T Xw = X^T y. \tag{22}$$

Solving for w, assuming $X^T X$ is invertible:

$$w = (X^T X)^{-1} X^T y. \tag{23}$$

8.3 Gradient Descent

- GD Algorithm
- Variant: mini-batch, stochastic
- Problems and Solutions

Computing the paritial derivative to find the minimum is costly, use local search to find a local min

$$w_j \leftarrow w_j - \gamma \frac{\partial(J(w_0, w_1 \dots))}{\partial(w_j)}$$

Where γ is the learning rate.
Repeat until termination criterion is satisfied

Convexity

Theorem 8.1. *A convex function has one singular global minimum.*

Theorem 8.2. *MSE loss function is convex for linear regression*

Variants

- Batch: take all training samples
- Mini-batch: random subset
- Stochastic: select one random sample

Linear regression with Regularization

Knowing the hypothesis

$$h_w(x) : w^T x$$

Penalize weights in the loss fucntion

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

Optimization goal: $\min(J_{reg}(w))$
Penalty functions

1. L1 penalty (Laso Regression in Linear regression): $P(w) = \sum_{j=0}^{d} |w_j|$
2. L2 penalty (Ridge regression in linear regression): $P(w) = \sum_{j=0}^{d} \frac{1}{2} w_j^2$

Gradient descent of regularized loss function

$$\frac{\partial J_{reg}(w)}{\partial w} = \frac{\partial J(w)}{\partial w} + \frac{\partial \lambda P(w)}{\partial w} \tag{24}$$

Regularized cost function (Ridge)

$$J(w) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_w(x^{(i)}) - y(i)) + \lambda \sum_{i=0}^{n} w_i^2\right]$$

Where
$\sum_{i=1}^{m}(h_w(x^{(i)}) - y(i))$ fits the data "well", where $\lambda \sum_{i=0}^{m} w_i^2$ is the regularization parameter, that avoid "over-fitting" Regularized J_{MSE} using Ridge

$$\frac{\partial J}{\partial w_i} = \frac{1}{2m}\left[\frac{\partial}{\partial w_i}\sum_{i=1}^{m}(w_0 + w_1 x^{(i)} - y^{(i)})^2 + \lambda \frac{\partial}{\partial w_i}\sum_{i=0}^{n} w_i^2\right]$$

$$= \frac{1}{m}\left[\sum_{i=1}^{m}(w_0 + w_1 x^{(i)} - y^{(i)})(x^{(i)}) + \lambda w_i\right], \qquad x_0 = 1$$

Regularized J_{MSE} using Lasso

$$\frac{\partial J}{\partial w_i} = \frac{1}{2m}\left[\frac{\partial}{\partial w_i}\sum_{i=1}^{m}(w_0 + w_1 x^{(i)} - y^{(i)})^2 + \lambda \frac{\partial}{\partial w_i}\sum_{i=0}^{n} |w_i|\right]$$

$$= \frac{1}{m}\left[\sum_{i=1}^{m}(w_0 + w_1 x^{(i)} - y^{(i)})(x^{(i)}) + \lambda sgn(w_i)\right], \qquad x_0 = 1$$

Generalized Regularization formula:

$$\lambda \sum_{i}^{n} |w_i|^p$$

Lasso reg. is when $p = 1$; Ridge when $p = 2$

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

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

$\forall \lambda > 0, X^T X + \lambda I$ is invertible

As $w = \begin{bmatrix} w_0 \\ w_1 \\ \cdots \\ w_n \end{bmatrix}$, here in the regularized cost function, m indicates the size of the training set, while n indicates the size of the feature vector large $\lambda \to$ underfitting; $\lambda = 0 \to$ overfitting

9 Logistic Regression

Classification Task

$$x \in \mathbb{R}^d \mapsto void$$

Find a function thats

$$y \mapsto \{0, 1\}, \text{ for each } x$$

Sigmoid

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

$$\sigma'(x) = \frac{-(1 + e^{-x})'}{(1 + e^{-x})^2} = \frac{e^{-x}}{1 + 2e^{-x} + e^{-2x}}$$

$\sigma(f(x))$

$$\sigma(x^2) = \frac{1}{1 + e^{-x^2}}$$

Decision Boundary

A surface that separates the different classes in the featyre space
The intersection betweeen the hypothesis function and the decision threshold

Binary Cross Entropy

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

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

$\sigma(x) = \frac{1}{1+e^{-x}}$ is Not convex, but $-log(\sigma(x))$ is convex
Hypothesis function

$$h_w(x) = \sigma(w_0, w_1x_1 + w_2x_2)$$

Weight Update

$$w_j \leftarrow w_j - \gamma\frac{\partial J_{BCE}(w_0, w_1, \dots)}{\partial w_j}$$

Loss function derivative

$$\frac{\partial J_{BCE}(w_j)}{\partial w} = \frac{\partial}{\partial w_j}\frac{1}{N}\sum_{i=1}^{N} BCE(y^{(i)}, h_w(x^{(i)})) = \frac{1}{N}\sum_{i=1}^{N}(h_w(x^{(i)}) - y^{(i)})x_j^{(i)}$$

9.1 Logistic regression cost function

$$\begin{cases} -log(h_w(x)), \text{ if y = 1} \\ -log(1-h_w(x)), \text{if y = 0} \end{cases} \tag{25}$$

Multiclass classification

One-to-one

$$\{0, 1, ...C\} \times \{0, 1, ...C\} \rightarrow (C^2 - C) \text{ BCE}$$

One-vs-One is a technique where a separate binary classifier is $\mathcal{C}$ trained for every pair of classes. For $\mathcal{C}$ classes, this results in $\frac{\mathcal{C}(\mathcal{C}-1)}{2}$ classifiers. Each classifier distinguishes between two specific classes.

One-vs-Rest

$$\{0, 1, ...C\}.map(c \rightarrow \{0, 1, ...C\} \backslash \{c\})$$

One-vs-Rest is a technique where a separate binary classifier is trained for each class, treating all other classes as a single combined class. For $\mathcal{C}$ classes, this results in $\mathcal{C}$ classifiers.

Generalization

- In supervised learning, and machine learning in general, the model's performance on unseen data is all we care about. This ability to perform well on new, unseen data is known as the model's generalization capability.
- Measuring a model's error is a common practice to quantify the performance of the model. This error, when evaluated on unseen data, is known as the generalization error.
- There are two factors that affect generalization:
  - Dataset quality
    - Relevance: Dataset should contain relevant data, i.e., features that are relevant for solving the problem.
    - Noise: Dataset may contain noise (irrelevant or incorrect data), which can hinder the model's learning process and reduce generalization.
    - Balance (for classification): Balanced datasets ensure that all classes are adequately represented, helping the model learn to generalize well across different classes.
  - Data quantity
    - In general, having more data typically leads to better model performance, provided that the model is expressive enough to accurately capture the underlying patterns in the data

    - Extreme case: if the dataset contains every possible data point, the model would no longer need to "guess" or make predictions. Instead, it would only need to simply memorize all the data!
  - Model complexity
    - Refers to the size and expressiveness of the hypothesis class.
    - Indicates how intricate the relationships between input and output variables that the model can capture are.
    - Higher model complexity allows for more sophisticated modeling of input-output relationships

Hyperparameter

Hyperparameters are settings that control the behavior of the training algorithm and model but are not learned from the data. They need to be set before the training process begins. Such as

- Learning rate
- Feature transformations
- Batch size and iterations in mini-batch gradient descent

Hyperparameter Tuning

Hyperparameter tuning is the process of hyperparameters optimizing the of a machine learning model to improve its performance. It is also known as hyperparameter search.

Objective

The goal is to find the best combination of hyperparameters that maximize the model's performance.
High bias: under-fitting High variance: over-fittingx

K-means Clustering

```
// Initialize K cluster centroids randomly from the dataset
for k = 1 in K clusters:
    μ_k = random(x ∈ D) // Pick a random data point x from dataset D as the initial centroid

// Repeat until convergence (e.g., centroids stop moving)
while (! convergence):
    // Assignment step: assign each point to the nearest centroid
    for i = 1 in m:
        c_i = argmin_k ||x^(i) − μ_k||^2 // Assign point x^(i) to the closest cluster (minimizing squared distance)

    // Update step: move centroids to the mean of assigned points
    for k = 1 in K:
        μ_k = 1/|{x^(i)|c^(i)=k}| Σ_{x∈{x^(i)|c^(i)=k}} x
        // Recompute centroid μ_k as the mean of all points assigned to cluster k
```

Dimentsion reduction

Theorem 9.1. *To learn a hypothesis class with d feature, $N = O(2^d)$ samples are needed. (The curse of dimensinality)*

In unsuperised leanring we have a massive data matrix $X^{N\times d}$. Consider X^T

SVD decomposition

$$A^{d\times N} = U^{d\times d}\Sigma^{d\times N}V^{T(N\times N)}$$

Where U contains d orthonormal columns,
Σ is a diagnol matrix where $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_N$
V contains N orthonormal columns and N orthonormal rows
(Supposing r < N < d)

$$X^T = \begin{bmatrix} | & | & ... & | \\ x^{(1)} & x^{(2)} & ... & x^{(N)} \\ | & | & ... & | \end{bmatrix}_{d\times N} = \begin{bmatrix} | & | & ... & | \\ u^{(1)} & u^{(2)} & ... & u^{(d)} \\ | & | & ... & | \end{bmatrix}_{d\times d} \begin{bmatrix} \sigma_1 & 0 & ... & 0 \\ 0 & \sigma_2 & ... & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & \sigma_N \\ 0 & 0 & ... & 0 \end{bmatrix}_{d\times N} \begin{bmatrix} | & | & ... & | \\ v^{(1)} & v^{(2)} & ... & v^{(N)} \\ | & | & ... & | \end{bmatrix}^T_{N\times N}$$

$$= \begin{bmatrix} \sigma_1 \sum_{i=1}^{N} u_{i1}v_{i1} & \sigma_2 \sum_{i=1}^{N} u_{i1}v_{i2} & \dots & \sigma_N \sum_{i=1}^{N} u_{i1}v_{iN} \\ \sigma_1 \sum_{i=1}^{N} u_{i2}v_{i1} & \sigma_2 \sum_{i=1}^{N} u_{i2}v_{i2} & \dots & \sigma_N \sum_{i=1}^{N} u_{i2}v_{iN} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_1 \sum_{i=1}^{N} u_{id}v_{i1} & \sigma_2 \sum_{i=1}^{N} u_{id}v_{i2} & \dots & \sigma_N \sum_{i=1}^{N} wu_{id}v_{iN} \end{bmatrix} = \begin{bmatrix} | & | & \dots & | \\ \sum_{i=1}^{N} \sigma_i v_1^{(i)} u^{(i)} & \sum_{i=1}^{N} \sigma_i v_2^{(i)} u^{(i)} & \dots & \sum_{i=1}^{N} \sigma_i v_N^{(i)} u^{(i)} \\ | & | & \dots & | \end{bmatrix}$$

By setting $r < N$, we keep the most significant r dimensions:

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_N \\ 0 & 0 & \dots & 0 \end{bmatrix} \to \begin{bmatrix} \sigma_1 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \dots & \vdots \\ 0 & \dots & \sigma_r & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix} \implies \tilde{U} = \begin{bmatrix} | & | & \dots & | \\ u^{(1)} & u^{(2)} & \dots & u^{(r)} \\ | & | & \dots & | \end{bmatrix}, \text{where } r < N$$

$$\tilde{X}^T =$$

After SVD Decomposition, U represents the new orthonormal basis, Σ represents the importance, V^T represents the linear combination coefficient.
From Σ we can tell the significance of the new basis vecotr, we can selectively remove less important ones to reduce the number of dimensions.

Theorem 9.2. *given a mean-centred data matrix $\hat{X}^T$, the values $\frac{\sigma_j^2}{N-1}$ is the variance of*

Therefore to retain a of variance in the data,

$$\text{Explained variance} = \frac{\sum_{i=1}^{r} \sigma_i^2}{\sum_{i=1}^{N} \sigma_i^2} \geq a$$

$$\dots = \frac{\sum_{i=1}^{N} ||\hat{x_i}^{(i)} - \tilde{x_i}^{(i)}||^2}{\sum_{i=1}^{N} ||\hat{x}^{(i)}||^2}$$

Perceptron Learning algorithm

Assuming linearly separable:

```
initialize w
while (i < max_iteration | !convergence):
    for (x_i, y_i) in D, if x_i is misclassified:
        ŷ = h_w(x)
        learning_error = (y_i − ŷ_i)
        w += γ(y_i − ŷ_i)x_i
```

Proof.

$$\hat{y} = sgn(\mathbf{w}^T x)$$

Case 1: $y = +1, \hat{y} = -1$

$\hat{y} < 1 \implies \mathbf{w}^T x < 0 \implies \mathbf{w}x < 0 \implies |\mathbf{w}||x| \cos\theta < 0 \implies \theta \in (\frac{\pi}{2}, \pi)$

But we want the opposite, ie $\hat{y} = +1 \implies \theta \in [0, \frac{\pi}{2}] \implies$ Reduce θ

$\mathbf{w}' = \mathbf{w} + x \implies \theta' < \theta$

Case 2: $y = -1, \hat{y} = +1$

Similarly to case1, but the other way round. $\mathbf{w}' = \mathbf{w} - x \implies \theta' > \theta$

$$\mathbf{w}' \leftarrow \mathbf{w} + 2\gamma x$$

$$\mathbf{w}' \leftarrow \mathbf{w} + \gamma(+1 - (-1))x$$

$$\mathbf{w}' \leftarrow \mathbf{w} + \gamma(y - \hat{y})x$$

Trivial to show, both cases arrive at the same update rule. □

If the data samples are not linearly separable, such method is not applicable since it does not guarentee convergence.

Multilayer neural network

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \mathbf{W}^{[1]} = \begin{bmatrix} W_{11}^{[1]} & W_{12}^{[1]} \\ W_{21}^{[1]} & W_{22}^{[1]} \\ W_{31}^{[1]} & W_{32}^{[1]} \end{bmatrix}$$

$$\hat{y}^{[1]} = g^{[1]}(\mathbf{W}^{[1]T} x)$$

For n layers,

$$\hat{y}^{[n]} = g^{[n]}({W^{[n]}}^T \hat{y}^{[n-1]})$$

$$\hat{y} = g^{[n]}({W^{[n]}}^T (g^{[n-1]}({W^{[n-1]}}^T ... g^{[2]}({W^{[2]}}^T g^{[1]}({W^{[1]}}^T \mathbf{x})))))$$

This is forward propagation

Multi class classification

Softmax function

$$\mathbf{x}^{N \times 1} \rightarrow \mathbf{z}^{C \times 1}$$

$\forall z_i \in \mathbf{z}, i \in [1, C], z_i \in [0, 1]$, as z_i indicates the probability of $\mathbf{x}$ belonging to class C_i. To map the neuron network output from $\mathbb{R}$ to $[0, 1]$, we use softmax function:

$$g(z_i) = \frac{e^{z_i}}{\sum_{i=1}^{C} z_i}$$

where $\mathbb{R}_g = [0, 1]$, and $\sum_{i=1}^{C} g(z_i) = 1$, indicating a complete probability distributon.

10 Torch

Torch library example:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

class CNNModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Sequential(
            # nn.Conv2d(in_chnl, out_chnl, kernel_size, stride, padding)
            nn.Conv2d(12, 16, kernel_size=3, padding=1),
            nn.BatchNorm2d(16),
            nn.GELU()
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.GELU()
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(32, 16, kernel_size=3, padding=0),
            nn.BatchNorm2d(16),
            nn.GELU()
        )
        # pooling layer
        self.pool = nn.AdaptiveAvgPool2d((3, 3))
        self.fc1 = nn.Linear(16 * 3 * 3, 32)
        self.fc2 = nn.Linear(32, 1)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.pool(x)
        x = x.view(-1, 16 * 3 * 3)
        x = self.dropout(F.relu(self.fc1(x)))
        return (self.fc2(x))

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)

train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNNModel().to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
```

Sequential data

Recurrent Neural Network (RNN) is a multilayer neural network, which can capture sequential features, by having hidden states

$$\text{Consider a sequence of elements} : x^1 = \begin{bmatrix} x_1^1 \\ x_2^1 \end{bmatrix}, x^2 = \begin{bmatrix} x_1^2 \\ x_2^2 \end{bmatrix}, x^3 = \begin{bmatrix} x_1^3 \\ x_2^3 \end{bmatrix}$$

$$\text{Initial hidden state} h_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Where $\mathbf{W}^{[hy]} = \begin{bmatrix} w_1 \\ \vdots \\ w_r \end{bmatrix}$, $\hat{y}^t = \mathbf{W}^{[hy]}$

$\mathbf{W}^{[hh]} = \begin{bmatrix} w_{11} & ... & w_{1h} \\ \vdots & \ddots & \vdots \\ w_{h1} & ... & w_{hh} \end{bmatrix}, \mathbf{W}^{[xh]} = \begin{bmatrix} w_{11} & ... & w_{1h} \\ \vdots & \ddots & \vdots \\ w_{x1} & ... & w_{xh} \end{bmatrix}$

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

$$\hat{y}^t = g^{[y]}\left((\mathbf{W}^{[hy]})^T\mathbf{h}_t\right)$$

$$\mathbf{h}^1 = g^{[h]}\left((\mathbf{W}^{[xh]})^T\mathbf{x}^1 + (\mathbf{W}^{[hh]})^T\mathbf{h}_0\right)$$

$$\text{Update } h_1 = \begin{bmatrix} h_1^1 \\ h_2^1 \end{bmatrix}$$

$$\mathbf{h}^2 = g^{[h]}\left((\mathbf{W}^{[xh]})^T\mathbf{x}^2 + (\mathbf{W}^{[hh]})^T\mathbf{h}_1\right)$$

$$\hat{y}^2 = \left((\mathbf{W}^{[hy]})^T\mathbf{h}^2\right)$$