

## Math

**Chain Rule.**  $h(x) = g(f(x)). \quad h'(x) = g'(f(x))f'(x).$

$$\frac{dh}{dx} = \frac{dg}{df} \frac{df}{dx}.$$

**Composition.**  $z(x) = h(g(f(x))). \quad \frac{dz}{dx} = \frac{dh}{dg} \frac{dg}{df} \frac{df}{dx}.$

**Hadamard Product.**  $\mathbf{W} \circ \mathbf{X} = \begin{pmatrix} w_{11}x_{11} & w_{12}x_{12} \\ w_{21}x_{21} & w_{22}x_{22} \end{pmatrix}.$

**S-by-V.**  $\frac{\delta y}{\delta \mathbf{x}} = \begin{pmatrix} \frac{\delta y}{\delta x_1} \\ \frac{\delta y}{\delta x_n} \end{pmatrix}.$  **V-by-V.**  $\frac{\delta \mathbf{y}}{\delta \mathbf{x}} = \begin{pmatrix} \frac{\delta y_1}{\delta x_1} & \frac{\delta y_n}{\delta x_n} \\ \frac{\delta x_1}{\delta x_n} & \frac{\delta x_n}{\delta x_n} \end{pmatrix}.$

**S-by-Mat.**  $\frac{\delta \mathbf{y}}{\delta \mathbf{X}} = \begin{pmatrix} \frac{\delta y}{\delta x_{11}} & \frac{\delta y}{\delta x_{1m}} \\ \frac{\delta y}{\delta x_{n1}} & \frac{\delta y}{\delta x_{nm}} \end{pmatrix}.$

**V-by-Mat.**  $\frac{\delta \mathbf{y}}{\delta \mathbf{X}} = \begin{pmatrix} \frac{\delta y_1}{\delta x_{11}} & \frac{\delta y_1}{\delta x_{1m}} \\ \frac{\delta y_1}{\delta x_{n1}} & \frac{\delta y_1}{\delta x_{nm}} \end{pmatrix} \dots \begin{pmatrix} \frac{\delta y_n}{\delta x_{11}} & \frac{\delta y_n}{\delta x_{1m}} \\ \frac{\delta y_n}{\delta x_{n1}} & \frac{\delta y_n}{\delta x_{nm}} \end{pmatrix}.$

**Invertibility.** A square matrix **A** is invertible if there exists another square matrix **B** such that **AB = I** and **BA=I**.

**Determinant.**  $\det(\mathbf{A}) = a_{11}$  if  $n = 1$  else  $a_{11}\mathbf{A}_{11} + a_{12}\mathbf{A}_{12} + \dots + a_{1n}\mathbf{A}_{1n}$  where  $\mathbf{A}_{ij} = (-1)^{i+j}\det(\mathbf{M}_{ij})$  and  $\mathbf{M}_{ij}$  is a  $(n-1) \times (n-1)$  matrix obtained from **A** by deleting the  $i$ -th row and  $j$ -th column. A square matrix is invertible iff determinant  $\neq 0$ .

**Eigens.** Nonzero column vector **u** is an eigenvector of **A** if **Au = λu** for some scalar  $\lambda$  (eigenvalue).  $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$ .

**Logs.**  $\log \frac{a}{b} = \log a - \log b$ .

## General Search Strategy

**Best-First Search.** Take node from frontier with lowest  $f(n)$ .

## Uninformed Search Strategies

**Breadth-First Search.** T/S:  $1 + b + b^2 + \dots + b^d = O(b^{d+1})$ . Complete if  $b$  is finite. Optimal if cost is 1 per step.

**Uniform-Cost Search.** Let  $C^*$  be the cost of the optimal solution, and  $\epsilon > 0$  be a lower bound on the cost of each action. T/S:  $O(b^{1+\lceil C^*/\epsilon \rceil})$ , which can be much greater than  $b^d$ . Complete if all action costs are  $> \epsilon > 0$ . Cost-optimal.

**Depth-First Search.** T:  $O(b^m)$  where  $m$  is maximum depth. S:  $O(bm)$ . Can be incomplete if there is an infinite path. Not optimal generally.

**Depth-Limited Search.** T:  $O(b^l)$  where  $l$  is limit. S:  $O(bl)$ . Incomplete if  $l$  is poorly selected. Good limit is diameter. Num nodes generated  $N(DLS) = b^0 + b^1 + \dots + b^d$ .

**Iterative Deepening Search.** T:  $O(b^m)$ . S:  $O(bm)$ . Complete. Cost-optimal if step cost is equal throughout. Num nodes generated  $N(IDS) = (d+1)b^0 + (d)b^1 + (d-1)b^2 + \dots + (1)b^d$ .

**Bidirectional Search.** Search from both initial state and goal state(s). Motivation:  $b^{d/2} + b^{d/2} \ll b^d$ . Track multiple frontiers and must be able to “traverse backwards”.

## Informed (Heuristic) Search

**Greedy Best-FS.**  $f(n) = h(n)$ . T/S:  $O(b^m)$  or  $O(|V|)$  (or  $O(bm)$  with good heuristic function. Not complete as can go into loops. Not cost-optimal.

**A\* Search.**  $f(n) = g(n) + h(n)$ , where  $g(n)$  is path cost from initial node to node  $n$ . T/S:  $O(b^d)$  for poor heuristic. If all action costs are  $> \epsilon > 0$  and state space is finite, A\* search is complete. Cost-optimal depends on heuristic properties.

**Memory-Bounded Search.** Save space by: store a node either in frontier or in *reached*, to remove states from *reached* when we can prove they are no longer needed e.g. by prohibiting U-turns, or by reference counting. Other new algorithms include:

- **Beam search.** Frontier = top  $k$   $f$ -scores, or limit to  $f$ -scores within  $\delta$  of best  $f$ -score. May be incomplete and suboptimal.
- **Iterative-deepening A\* search (IDA\*).** IDS but limit = smallest  $f$ -cost of any node that exceeded the cutoff on the previous iteration.

By Hanming Zhu

- **Recursive best-first search (RBFS).** Tracks the  $f$ -value of next best path from any ancestor of current node. If curr node  $i$  this value, unwinds to that path. As it unwinds, RBFS replaces the  $f$ -value of nodes along path with the backed-up value. Optimal if  $h(n)$  is admissible. S:  $O(d)$ ,  $d$  = depth of deepest optimal solution. IDA\* and RBFS use too little memory, even if memory is available. They may end up reexploring the same states many times over. We can thus employ:

- **Memory-bounded A\* (MA\*).** Not covered.
- **Simplified MA\* (SMA\*).** Expand newest best leaf until memory is full. Then drop oldest node with worst  $f$ -value and backup the forgotten node's value to its parent. SMA\* will regenerate that subtree only when all other paths look worse than the forgotten path.

## Heuristic Properties

**Admissibility.** Never overestimates the cost to reach a goal. If admissible, then A\* using **tree-like search** is cost-optimal.

**Consistency.** Stronger than admissibility.  $h(n)$  is consistent if, for every node  $n$  and every successor  $n'$  of  $n$  generated by an action  $a$ , we have  $h(n) \leq c(n, a, n') + h(n')$ , i.e. triangle inequality. If consistent, then A\* using **graph search** is cost-optimal.

**Effective branching factor.** Compute  $b^*$  from  $N$  nodes generated and solution depth  $b$  using  $N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$ . The closer to 1  $b^*$  is, the better.

**Effective depth.** Reduce depth by constant  $k_h$ , i.e.  $O(b^{d-k_h})$  vs  $O(b^d)$  for uninformed search.

If  $h_2$  dominates  $h_1$ , i.e. for any node  $n$ ,  $h_2(n) \geq h_1(n)$ , then A\* using  $h_2$  will never expand more nodes than using  $h_1$ . To generate heuristics, we can **relax** the problem, i.e. remove certain restrictions on the actions.

## Local Search

Aim is to find the best state according to an **objective function**.

**Hill-Climbing Search / Greedy Local Search.** Travel to better neighbours. Easy objective function is to negate the heuristic function. Suffers from local maxima, ridges (sequence of local maxima), plateaus (sequence of same values, but is local maxima) and shoulders (same values, but progress is possible). Solutions are:

- **Sideways Move.** Do a limited no. of sideways moves, see if the plateau is actually a shoulder.
- **Stochastic Hill Climbing.** Choose a random uphill move, probabilities based on steepness of move.
- **First-choice Hill Climbing.** Randomly generate successors until one is better than current. Useful if state has e.g. thousands of succs.
- **Random-Restart Hill Climbing.** Perform a fixed no. of steps from some randomly generated initial steps, then restart if no maximum found.
- **Simulated Annealing.** Randomly pick a next move. If better, go, else accept it with a probability  $< 1$  that decreases exponentially with the “badness” of the move. Escape local maxima by allowing some random moves.

**Local Beam Search.** Pick  $k$  random initial states, then generate successors. If goal is found, terminate, else pick best  $k$  successors and repeat. Like  $k$  random restarts but info is shared. May suffer from a lack of diversity if  $k$  states start to cluster.

**Stochastic Beam Search** alleviates clustering by choosing successors with probability proportional to their value. **Genetic Algorithms.** Each individual is a string over a finite alphabet (often a Boolean string). Mixing number = no. of parents that combine. Select parents, e.g. probability proportional to fitness score. Combine the parents, e.g. randomly select a crossover point to split the parent strings, then mix the four parts to form two children. Once offspring is generated, every bit in its composition is flipped

with probability equal to the mutation rate. The next generation can be purely offspring, or may also include parents (elitism).

## Adversarial Search

**Minimax Search.** Utility of state in game tree, *assuming both players play optimally* from there.

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

T:  $O(b^m)$ . S:  $O(bm)$ . Complete if tree is finite.

Can prune via **Alpha-Beta pruning**.  $\alpha$  is best choice found so far along search path for MAX,  $\beta$  is best choice for MIN. If at MIN node, can stop checking successors once we find a node  $\leq \alpha$ , since that upper bounds all other values. If at MAX node, can stop if we find a node that  $\geq \beta$ .

Hence, **move ordering** matters. If best successor is generated first, then we can prune the rest. If done perfectly, alpha-beta would examine  $O(b^{m/2})$  nodes instead of  $O(b^m)$ , i.e. effective branching factor becomes  $\sqrt{b}$ , or that we can examine a tree twice as deep in the same time. With **random move** ordering, we have roughly  $O(b^{3m/4})$  for moderate  $b$ .

Another optimisation is **transpositions** — cache previously seen states in a transposition table, since different permutations of move sequence can end up in the same position.

**Heuristic Alpha-Beta Tree Search.** Go wide but shallow. Replace utility function with **evaluation function** that estimates the expected utility of a state. For terminal states,  $\text{EVAL}(s, p) = \text{UTILITY}(s, p)$ , and for non-terminal states,  $\text{UTILITY}(\text{loss}, p) \leq \text{EVAL}(s, p) \leq \text{UTILITY}(\text{win}, p)$ . Also replace IS-TERMINAL( $s$ ) with IS-CUTOFF( $s, d$ ), where  $d$  is search depth of  $s$ , and returns true if  $s$  is terminal, and either true or false otherwise.

The **evaluation function** uses features of the state to compute an expected utility value. For chess, it's typically the linear weighted sum:  $\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$ . Features are e.g. number of rooks, etc. Assumes that features are independent.

**Cutting Off Search.** Heuristic A/B tree search but we cut off past a certain depth. Can do iterative deepening. If transposition table is used, subsequent rounds will get faster and can use the past evaluations to improve move ordering.

## Machine Learning

**Classification.** Output is one of a finite set of values.

**Regression.** Output is a number.

**Supervised Learning.** Observe input-output pairs and learns a function that maps input to output. **Unsupervised Learning.** Learns patterns in the input without explicit feedback. Most common task is clustering. **Reinforcement Learning.** Agent learns from a series of reinforcements: rewards and punishments.

## Linear Regression

$y = \theta_0 + \theta_1 x$ , where  $x$  is input,  $y$  is output, and  $\theta_0, \theta_1$  are real-valued coefficients to be learnt.  $w$  may also be used instead of  $\theta$ .

**Squared-Error Loss.**  $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$  where  $m$  is the number of examples. Minimized when its partial derivatives with respect to  $\theta_0$  and  $\theta_1$  are zero:  $\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = 0$  and  $\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = 0$ . Unique solution if these are zero (not always the case):  $\theta_1 = \frac{m(\sum x^{(i)} y^{(i)}) - (\sum x^{(i)})(\sum y^{(i)})}{m(\sum (x^{(i)})^2) - (\sum x^{(i)})^2}$  and  $\theta_0 = \frac{\sum y^{(i)} - \theta_1 \sum x^{(i)}}{m}$ .

**Batch Gradient Descent.** Compute an estimate of gradient at each point and move a small amount in the steepest downhill direction, until we converge on a point with min loss. Hypothesis =  $h_\theta(x^{(i)}) = \theta^T x^{(i)}$ .  $\theta_0 \leftarrow \theta_0 -$

$\alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})$ .  $\theta_1 \leftarrow \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \times x^{(i)}$ .  $\alpha$  is the learning rate.

**Stochastic Gradient Descent.** Randomly select a small no. of examples at each step, so it's faster.

**Multivariate Regression.** Extend the above to  $n$ -element vectors:  $h_\theta(x^{(i)}) = \theta_0 + \sum_{n=1}^n \theta_n x_n^{(i)}$ . Loss function is  $J(\theta)$ .  $\theta_n \leftarrow \theta_n - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \times x_n^{(i)}$ .

**Normal Equation.** Possible to solve analytically, without gradient descent.  $\mathbf{y}$  = output vector.  $\mathbf{X}$  = data matrix. Vector of predicted outputs:  $\hat{\mathbf{y}} = \mathbf{X}\theta$ . Squared-error loss is  $L(\theta) = \|\hat{\mathbf{y}} - \mathbf{y}\|^2 = \|\mathbf{X}\theta - \mathbf{y}\|^2$ . Set gradient to 0 and rearrange to get the minimum-loss weight vector  $\theta = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$ .  $\mathbf{X}^T \mathbf{X}$  needs to be invertible. T:  $O(n^3)$ .

**Feature Scaling.** Via mean normalisation, i.e.  $x_i \leftarrow \frac{x_i - \mu_i}{\sigma_i}$ .

**Linear Classification.**  $h_\theta(x) = 1$  if  $\theta^T x > 0$  (or 0.5) else 0 otherwise.  $\theta_i \leftarrow \theta_i + \alpha(y - h_\theta(x)) \times x_i$ . Determine linear decision boundary / separator.

## Polynomial Regression

Handles non-linear relationships. Let  $x_2 \leftarrow x^2$ ,  $x_3 \leftarrow x^3$  and so on. Or even  $\sqrt{x}$ . Need to do feature scaling.

## Decision Trees

Represents a function that maps a vector of attribute values to an outcome. Internal node = test. Leaf = decision.

**Expressiveness.**  $\text{Output} \Leftrightarrow (\text{Path}_1 \vee \text{Path}_2 \vee \dots)$ ,  $\text{Path}_i$  is a conjunction ( $A_m = v_x \wedge A_n = v_y \wedge \dots$ ) of attribute-value tests corresponding to path from root to leaf.

**Hypothesis Space.**  $n$  attributes = truth table has  $2^n$  rows. Each row outputs true or false, so  $2^{2^n}$  possible functions / distinct decision trees.

## Learning

1. If no eggs, return default decision e.g. most common decision from parent's examples.
2. If eggs are all true or false, return that value.
3. If no attributes left for splitting, return most common decision of current eggs.
4. Else, then split on best attribute and recurse.

Choosing attributes: use **information gain**, defined in **entropy**. For a random variable  $V$  with values  $v_n$  having probability  $P(v_n)$ ,  $I(P(v_1), \dots, P(v_n)) = -\sum_{i=1}^n P(v_i) \log_2 P(v_i)$ . Unit is number of bits. For training set with  $p$  positive and  $n$  negative eggs,  $I(\frac{p}{p+n}, \frac{n}{p+n}) = -\frac{p}{p+n} \log_2 \frac{p}{p+n} - \frac{n}{p+n} \log_2 \frac{n}{p+n}$ .

An attribute  $A$  with  $v$  distinct values divides the training set  $E$  into subsets  $E_1, \dots, E_v$ . We define  $\text{remainder}(A) = \sum_{i=1}^v \frac{p_i + n_i}{p+n} I(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i})$ .

The **information gain** from  $A$  attribute test is expected reduction in entropy:  $IG(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - \text{remainder}(A)$ . We choose the attribute with largest  $IG$ .

**Overfitting.** Becomes likelier as num attribute grows, and less likely as num training eggs grow.

**Pruning.** Look at test node that only has leaf descendants nodes. If test deemed statistically insignificant, replace it with a leaf node, and repeat recursively. Assume there is no underlying pattern (null hypothesis), then calculate the deviation from an absence of pattern and see if unlikely (5% or less). Can be done via  $\chi^2$  tests ( $\chi^2$  pruning).

**Missing Data.** Fill in with most common value at node. Or most common value amongst eggs with same output. Or assign based on some probability.

**Continuous and Multi-Valued Input.** Use inequality tests. For attributes with a large number of possible values but are not continuous nor have a meaningful ordering, we can use IG ratio instead, where  $\text{SplitInformation}(A) = -\sum_{i=1}^d \frac{|E_i|}{|E|} \log_2 \frac{|E_i|}{|E|}$  and  $\text{GainRatio}(A) = \frac{\text{Gain}(A)}{\text{SplitInformation}(A)}$ .

**Differing Costs.** If there are costs incurred to obtain some attributes, learn a consistent DT with low expected cost by

using  $\frac{Gain^2(A)}{Cost(A)}$  or  $\frac{Gain(A)-1}{(Cost(A)+1)}$  where  $\omega \in [0, 1]$  = relative importance of cost vs IG.

**Continuous-Valued Output.** Need regression tree.

**Logistic Regression**

Classification with linear regression has problems, since it's not differentiable and is discontinuous. We can approximate using the logistic function  $Logistic(z) = \frac{1}{1+e^{-z}}$ .

$$h_{\theta}(x) = \frac{1}{1+e^{-\theta^T x}}. \quad J(\theta) = \frac{1}{m} \sum_{j=1}^m Cost(h_{\theta}(x), y_i)$$

where  $Cost(h_{\theta}(x), y) = -y \log h_{\theta}(x) - (1-y) \log(1-h_{\theta}(x))$  (natural log). Gradient descent no change. We interpret  $h_{\theta}(x)$  as the estimated probability that  $y = 1$  on input  $x$ .

**Multi-class.** Train  $n$  different logistic classifiers for all  $n$  classes. For each input, pick the class  $i$  with max output.

**Model Selection**

**Hyperparameters.** Parameters of the model class, not the individual model e.g. max degree for polynomial regression, threshold for  $\chi^2$  pruning.

**Cross-Validation.** Training set is used to train multiple candidate models. Validation (dev) set is used to evaluate candidate models (may be using different hyperparameters) to get the best one. Test set tests the best model. Error used is the loss function.

**Bias vs Variance.** High bias = underfit, more data points won't help.  $J_{train}(\theta) \approx J_{cv}(\theta)$  will be high. High variance = overfit, more data points will help.  $J_{train}(\theta)$  will be low and  $J_{cv}(\theta) \gg J_{train}(\theta)$ .

**Performance Measures.** For test set.

- **Correctness.** Percentage correct.
- **Accuracy.** Average correctness across  $m$  instances.
- **Confusion Matrix.** From top left to bottom right, TP, FP, FN, TN. Accuracy =  $\frac{TP+TN}{TP+FP+FN+TN}$ . Precision =  $\frac{TP}{TP+FP}$  i.e. how many selected items are correct. Recall =  $\frac{TP}{TP+FN}$  i.e. how many correct items are selected.
- **F1 Score.**  $(\frac{P^{-1}+R^{-1}}{2})^{-1} = \frac{2TP}{2TP+FP+FN}$ . More robust (less sensitive to extreme values).
- **Receiver Operator Characteristic (ROC) Curve.** For each value of a hyperparameter, plot a curve of true positive rate (y) (sensitivity) vs false positive rate (x) (1 - specificity). Better than random if ROC curve is above random line  $y = x$ .
- **Area Under Curve (AUC) of ROC.** Concise metric. AUC > 0.5 means model is better than chance. AOC  $\approx 1$  means very accurate.
- **Log Loss.** Negative average of log of corrected predicted probabilities for each instance. Smaller = better.
- **MSE.** Only for regression.

**Regularisation**

Alternative to reducing the no. of features.

**Linear Regression.** Penalise complex hypotheses.  $J(\theta) = \frac{1}{2m} (\sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{i=1}^n \theta_i^2)$  where  $n$  is no. features and  $\lambda$  is the regularisation parameter (hyperparameter).  $\theta_n \leftarrow (1 - \frac{\alpha \lambda}{m}) \theta_n - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) \times x_n^{(i)}$ . Need to do feature scaling. L2 Norm:  $\theta^2$ , L1 Norm:  $||\theta||$ .  
**Normal Equation.**  $\theta = (X^T X + \lambda I_m)^{-1} X^T Y$ . Works even if  $X^T X$  is non-invertible if  $\lambda > 0$ .  
**Logistic Regression.**  $J(\theta) = \frac{1}{m} \sum_{j=1}^m Cost(h_{\theta}(x), y_i) + \frac{\lambda}{2m} \sum_{i=1}^n \theta_i^2$ . Gradient descent same as for linear regression regularisation.

**Support Vector Machines**

Constructs a **maximum margin separator**. Closest vectors to separator are support vectors as they "hold up" the separating plane.

$J(\theta) = C(\sum_{i=1}^m y^{(i)} cost_1(\theta^T x) + (1 - y^{(i)}) cost_0(\theta^T x)) + \frac{1}{2} \sum_{i=1}^n \theta_i^2$ .  $cost_1$  decreases linearly until  $x = 1$  then = 0 after.  $cost_0$  is the reverse.  $C \approx \frac{1}{\lambda}$ .  $h_{\theta}(x) = 1$  if  $\theta^T x \geq 0$  else 0. First term also called hinge loss and also expressed as  $y \max(0, 1 - \theta^T x) + (1 - y) \max(0, 1 + \theta^T x)$ .

**Kernel tricks.** If data (size  $N$ ) is mapped into a space of sufficiently high dimension (e.g.  $N - 1$ ), almost always linearly separable. First set landmarks  $l^{(i)} = x^{(i)}$ . Using

**Gaussian kernel,**  $f_i = similarity(x, l^{(i)}) = e^{-\frac{|x-l^{(i)}|^2}{2\sigma^2}}$ .  $h_{\theta}(x) = 1$  if  $\theta_0 + \theta_1 f_1 + \dots + \theta_n f_n \geq 0$  else 0. We minimise  $C(\sum_{i=1}^m y^{(i)} cost_1(\theta^T f) + (1 - y^{(i)}) cost_0(\theta^T f)) + \frac{1}{2} \sum_{i=1}^n \theta_i^2$ . Need to apply feature scaling. Use for small  $n$  and intermediate  $m$ , otherwise for  $n \gg m$  or  $m \gg n$ , use logistic regression or linear (no) kernel.

**Parameters.** Large C: Low bias, high variance. Small C: High bias, low variance. Large  $\sigma^2$ : Smoother variation for features. High bias, low variance. Small  $\sigma^2$ : Low bias, high variance.

**Other Kernels.** Need to satisfy Mercer's theorem. Polynomial, String,  $\chi^2$ , tanh.

**Multi-Class.** Train one SVM for each class. Pick class  $i$  with max  $(\theta^{(i)})^T x$ .

**Soft Margin.** For non-linearly separable. Allow eggs to fall on wrong side but assign penalty based on distance.

**Perceptrons**

**Activation Functions.** Perceptrons compute  $\hat{y} = g(\sum_{i=0}^n w_i x_i)$ , where  $x_0 = 1$  and  $g$  is a non-linear activation function. **Step.**  $sgn(x) = 1$  if  $x \geq 0$  else -1. **Sigmoid (Logistic Reg).**  $\sigma(x) = \frac{1}{1+e^{-x}}$ . **SILU.**

$x \times \sigma(x)$ . **tanh.**  $\tanh x$ . **ReLU.**  $\max(0, x)$ . **XOR Gate.**  $XOR(x_1, x_2) = AND(NOT(AND(x_1, x_2)), OR(x_1, x_2))$ . **Perceptron Learning Algorithm.** Initialize weights  $w$  (all zero or random small values). For each instance  $i$  classify  $\hat{y}^{(i)}$  with features  $x^{(i)}$ . Select **one** misclassified instance and update weights with  $w \leftarrow w + \eta(y - \hat{y}) \times x$ . Note that  $y$  and  $\hat{y}$  are both either 1 or 0. This is  $\equiv$  stochastic gradient descent.

**Perceptron vs Linear SVM.** Not robust and cannot converge for non-linearly separable. SVM can converge + maximises margin.

**Gradient Descent.** General: Compute  $\frac{\delta \epsilon}{\delta w}$  where  $\epsilon$  is the loss function. Iterate  $w \leftarrow w - \eta \frac{\delta \epsilon}{\delta w} = w - \eta \frac{\delta \epsilon}{\delta \hat{y}} \cdot \frac{\delta \hat{y}}{\delta w}$  until convergence. MSE:  $\frac{1}{2}(\hat{y} - y)^2$ .  $w_i \leftarrow w_i + \eta(y - \hat{y}) y' g'(f(x)) x_i$ . Sigmoid:  $g'(x) = \sigma'(x) = \sigma(x)(1 - \sigma(x)) \rightarrow w \leftarrow w + \eta(\hat{y} - y)\hat{y}(1 - \hat{y})x$ . **tanh:**  $g'(x) = \tanh' x = 1 - \tanh^2 x$ . **ReLU:**  $g'(x) = 0$  if  $x < 0$  else 1 if  $x > 0$ , undefined for 0.

**Multi-Layer Neural Network.**

**Layer Activation.** At layer  $l - 1$  (1-based), we have perceptrons feeding values 1, and  $\mathbf{a}_1^{[l-1]}$  to  $\mathbf{a}_m^{[l-1]}$ . To compute  $\mathbf{a}_j^{[l]}$ , we multiply 1 by  $\mathbf{w}_{j0}^{[l]}$ ,  $\mathbf{a}_1^{[l-1]}$  by  $\mathbf{w}_{j1}^{[l]}$ , until  $\mathbf{a}_m^{[l-1]} \times \mathbf{w}_{jm}^{[l]}$ , then apply the activation function at  $l$ :  $\mathbf{a}^{[l]} = g^{[l]}((\mathbf{W}^{[l]})^T \mathbf{a}^{[l-1]})$ .

**Forward Propagation / Feedforward.** As  $\mathbf{a}^{[l]} \equiv g^{[l]}(f^{[l]})$ ,  $f^{[l]} \equiv (\mathbf{W}^{[l]})^T \mathbf{a}^{[l-1]}$ , hence  $\hat{y}(\mathbf{x}) = g^{[L]}(f^{[L]}(g^{[L-1]}(\dots(g^{[1]}(f^{[1]}(\mathbf{x})))\dots)))$ .

**Multi-Layer Gradient Descent.** Formulas same as single-layer. Assume we have two layers,  $\hat{y} = \mathbf{w}_2 \mathbf{a}_1$  where  $\mathbf{a}_1 = \sigma(\mathbf{w}_1 \mathbf{x})$ .  $\frac{\delta \epsilon}{\delta \mathbf{w}_2} = \frac{\delta \epsilon}{\delta \hat{y}} \frac{\delta \hat{y}}{\delta \mathbf{w}_2}$  (Chain Rule).  $\frac{\delta \epsilon}{\delta \mathbf{w}_1} = \frac{\delta \epsilon}{\delta \hat{y}} \frac{\delta \hat{y}}{\delta \mathbf{a}_1} \frac{\delta \mathbf{a}_1}{\delta \mathbf{w}_1}$  (Chain Rule). Key insight is that once we solve for all the  $\frac{\delta \hat{y}}{\delta \mathbf{w}_i}$ s, we can compute any  $\frac{\delta \epsilon}{\delta \mathbf{w}}$  for use with gradient descent.

**Generalise to Matrix Form.**  $\frac{\delta \hat{y}}{\delta \mathbf{w}^{[l]}} = \mathbf{a}^{[l-1]} \delta^{[l]}$  and  $\delta^{[l]} = g'^{[l]}(f^{[l]}) \mathbf{W}^{[l+1]} \delta^{[l+1]}$ .

**Backpropagation.** Efficiently computes the gradient. Avoids duplicate calculations and unnecessary intermediate values. Calculated from back  $[l + 1]$  to front  $[l]$ :  $\frac{\delta \hat{y}}{\delta \mathbf{w}^{[l]}} = \mathbf{a}^{[l-1]} (\delta^{[l]})^T$  and  $\delta^{[l]} = [g'^{[l]}(f^{[l]})] \circ (\mathbf{W}^{[l+1]} \delta^{[l+1]})$ . If we want to use custom layers, they need to be differentiable.

**Deep Neural Networks**

Refers to NN with  $\geq 3$  layers. Goal is to have long computation paths with input variables interacting in complex ways.

**Convolutional Networks.** Adjacency of pixels matters + expensive to work with raw images (where each pixel is a matrix). Idea is to dot product every  $l$  by  $l$  "submatrix" in the image with a **flipped** size  $l$  **kernel** and store that in a resultant feature map. We use the  $*$  symbol,  $\mathbf{z} = \mathbf{x} * \mathbf{k}$ , defined as  $z_i = \sum_{j=1}^l k_j x_{j+1-(l+1)/2}$ . Using different kernels we can extract different features. Add **paddings** around image to avoid losing pixels at edges. Increase **stride** to move more "cells" right and down i.e. faster.  $h = \lfloor \frac{H-K+2P}{S} \rfloor + 1$

**Convolutional Layer.**  $\mathbf{A}^{[l]} = g^{[l]}(\mathbf{W}^{[l]} * \mathbf{A}^{[l-1]})$ , where  $\mathbf{W}^{[l]}$  is a "list" of kernels, i.e. a 3D matrix, and  $\mathbf{A}^{[l-1]}$  is a "list" of feature maps, i.e. also 3D matrix.  $g^{[l]}$  could be same as for a linear layer.

**MLP vs CNN.** MLP has multiple neurons per layer (1D vector), which take in 1D vector of activations and return 0D scalar activation. CNN has multiple kernels per layer (3D matrix), which take in 3D matrix of feature maps (stack of filters) and return 2D matrix feature map.

**Intuition.** First layer responds to simple shapes like edges. Higher layers respond to more complex structures, like tail/leg of animal. Top layer responds to complex/abstract concepts, e.g. what we identify as animals.

**Pooling.** Layer that summarises the preceding layer. Similar to convolutional layer (with size and stride), but operation is fixed and not learnt. Typically no activation function is applied.

- **Average-Pooling.** Computes avg of  $l$  inputs. This downsamples the resolution (i.e. coarsen the resolution) by a factor of  $l$ .
- **Max-Pooling.** Though also downsamples, semantics differ. It acts as a logical disjunction and highlights features e.g. darker spots in the receptive field.

Downsampling reduces dimensionality and facilitates multiscale recognition. Also reduces num weights needed later, thus lower costs and faster learning.

**Softmax.** Outputs a categorical distribution, i.e.  $d$  values with probabilities summing to 1.  $P(y = j | \theta^{(i)}) = \frac{e^{\theta_j^{(i)}}}{\sum_{d=0}^d e^{\theta_d^{(j)}}}$  where  $\theta^{(i)}$  is the  $i$ -th input value.

**CNN Architecture and Usage.** Feature learning phase repeats (Convolution + ReLU + Pooling). Classification phase does (Flatten + Softmax). Used for image classification and segmentation, and object detection.

**Recurrent NN.** Allow cycles in computation graphs with a delay. It also stores in memory the past inputs. Detects long-distance dependencies.  $\mathbf{a}_t^{[l]} = g^{[l]}((\mathbf{W}_x^{[l]})^T \mathbf{a}_t^{[l-1]} + (\mathbf{W}_h^{[l]})^T \mathbf{a}_{t-1}^{[l]})$ . Uses backpropagation through time.

**RNN Usage.** One-to-many: one input for many outputs over time, text generation, image captioning. Many-to-one: Text/sentiment classification. Many-to-many: Language translation, music generation, processes all inputs first then predicts output. Many-to-many with only time dependency: Plain RNN, 1 input processed at a time.

**Overfitting.** Can regularize with **drop out**, i.e. randomly set some activations to 0 during training. Can also do **early stopping**, where training stops once the validation set loss stops decreasing and increases instead.

**Vanishing / Exploding Gradients.** When gradients approach 0 during backpropagation and weights stop updating, or when gradients keep getting larger, causing gradient descent to diverge.

- **Proper Weight Initialization.** Also called Glorot initialization.
- **Use Non-Saturating Activation Fns.** Can use ReLU, but it suffers from dying ReLU. Alternatives are Leaky ReLU, Exponential Linear Unit (ELU), etc.
- **Batch Normalization.** Each layer does feature scaling.

- **Gradient Clipping.** Clip gradients to  $[-1.0, 1.0]$ .

**Clustering (Unsupervised Learning)**

**K-Means.** Randomly initialise  $K$  centroids  $\mu_1, \mu_2, \dots, \mu_K$ . Repeat until convergence:

- For  $i = 1$  to  $m$ :  $c^{(i)} \leftarrow$  index of centroid closest to  $\mathbf{x}^{(i)}$ .
- For  $k = 1$  to  $K$ :  $\mu_k \leftarrow$  centroid of points assigned to cluster  $k$ .

Loss:  $J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K) = \frac{1}{m} \sum_{i=1}^m ||\mathbf{x}^{(i)} - \mu_{c^{(i)}}||^2$ . Affected by initial  $K$  centroids. Solutions: try many times and pick the one with the lowest loss, or K-Means++.

**K-Medoids.** K-Means but now we pick the data point that has min avg distance to all other points in the cluster. Final centroids would now be interpretable.

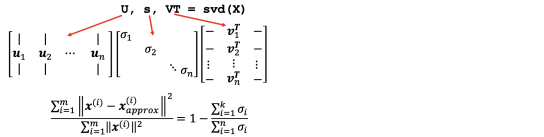
**Selecting K.** In **elbow method**, we plot loss against  $K$ , and pick the  $K$  where after that point, loss decreases linearly. Another way is via **business need**, e.g. T-shirt sizes.

**Hierarchical (Agglomerative) Clustering.** Avoid picking  $K$ . Like UFDs. Every point is a cluster, then we merge the two nearest clusters until all points are in one cluster. Likely need feature scaling. A **dendrogram** shows how sample merge, plotted against distance. To calculate distance, we can use max, min, group avg, centroid distance, Euclidean distance, Manhattan distance. High complexity. Applied for news, medical imaging, recommendation engines, social network analysis.

**Dimensionality Reduction.** Complexity of unsupervised learning is high. Aim is to reduce dimensionality (fewer features) but capture most of the meaningful properties.

**Principal Component Analysis.**  $\mathbf{A}\mathbf{X} = \mathbf{X}\mathbf{\Lambda}$ , where  $\mathbf{X}$  is a matrix of column eigenvectors  $v_1, \dots, v_n$  and  $\mathbf{\Lambda}$  is a diagonal matrix of eigenvalues  $\lambda_1, \dots, \lambda_n$ . We thus have  $\mathbf{A} = \mathbf{X}\mathbf{\Lambda}\mathbf{X}^{-1}$ . Using **singular value decomposition**, we can get  $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ , where  $\mathbf{\Sigma}$  is a diagonal matrix of singular values of  $\mathbf{X}$   $\sigma_1, \dots, \sigma_n$  and the columns of  $\mathbf{U}$  and  $\mathbf{V}$  are the left and right singular vectors respectively.

1. **Compute Covariance.**  $\mathbf{\Sigma} = \frac{1}{m} \sum_{i=1}^m (x^{(i)})(x^{(i)})^T$ .
2. **Apply SVD.** Get  $\mathbf{U} = [\mathbf{u}_1, \dots, \mathbf{u}_n]$ .
3. **Extract first k vectors.**  $\mathbf{U}_{reduce} = [\mathbf{u}_1, \dots, \mathbf{u}_k]$ .
4. **Multiply.**  $\mathbf{z}^{(i)} = \mathbf{U}_{reduce}^T \mathbf{x}^{(i)}$  for  $0 \leq i \leq m$ .



$\mathbf{x}^{(i)} \in \mathbb{R}^n \rightarrow \mathbf{z}^{(i)} \in \mathbb{R}^k$ . Achieves compression and visualization ( $k = 2$  or  $3$ ). Don't use for overfitting. Pick  $k$  and retain 99% of variance: minimum  $k$  s.t.  $\frac{\sum_{i=1}^k \sigma_i}{\sum_{i=1}^n \sigma_i} \geq 0.99$ .

**AI Ethics**

**Training Data Collection.** Data is often bad (manipulated, incomplete, biased), missing (representation bias) or tainted. Need to scrub data to neutral and mirror demographics.

**Model Training.** Metrics calculated against an entire set of data are not always accurate, e.g. overall accuracy for male and female patients is okay, but individually, accuracy for male patients does not meet the cut.

**Replicable Biased Decisions.** Harm reinforcement occurs when algorithms cause allocation and representation issues to be further entrenched. People assume that they are objective.

**Final Concepts.** Utilitarianism. Kantianism. Subjective Relativism (what's right for you may not be for me). Code of conduct: have reasonable diligence, use scientific method, no cherry picking, transparency about data collection and usage/sharing. Garbage in, garbage out.