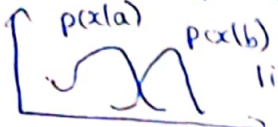
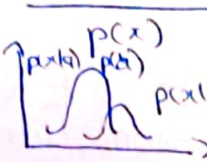
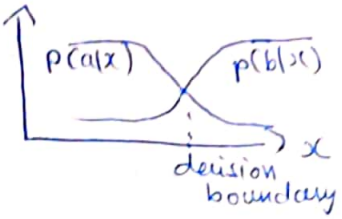
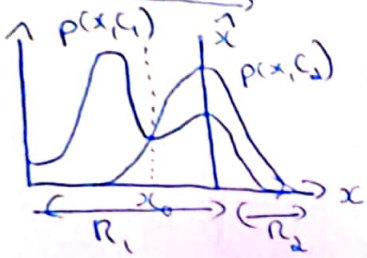
Bayes. $p(C_k|x) = \frac{p(x|C_k)\,p(C_k)}{p(x)} = \frac{p(x|C_k)\,p(C_k)}{\sum_i p(x|C_i)\,p(C_i)}$









p(mistake) $= p(x \in R_1, C_2) + p(x \in R_2, C_1)$

$= \int_{R_1} p(x, C_2)\,dx + \int_{R_2} p(x, C_1)\,dx$

$= \int_{R_1} p(C_2|x)\,p(x)\,dx + \int_{R_2} p(C_1|x)\,p(x)\,dx$

Decide for C_1 if

$p(C_1|x) > p(C_2|x) \Rightarrow \frac{p(x|C_1)}{p(x|C_2)} > \frac{p(C_2)}{p(C_1)}$ } - decision threshold θ

Likelihood-ratio test for k class

$\frac{p(x|C_k)}{p(x|C_j)} > \frac{p(C_j)}{p(C_k)} \quad \forall j \neq k$

Minimize the expected loss

$E(L) = \sum_k \sum_j \int_{R_j} L_{kj}\, p(x, C_k)\,dx$

Generalisation with loss function

We can formalize this by introducing a loss matrix L_{kj} (decision)

L_{cancer} = (Truth) cancer / normal × (decision) cancer / normal = $\begin{bmatrix} 0 & 1000 \\ 1 & 0 \end{bmatrix}$

2 class: C_1, C_2, 2 decision: α_1, α_2

Loss function: $L(\alpha_j | C_k) = L_{kj}$

Expected loss (= risk R) for two decisions:

$E_{\alpha_1}(L) = R(\alpha_1|x) = L_{11}\,p(C_1|x) + L_{21}\,p(C_2|x)$

$E_{\alpha_2}(L) = R(\alpha_2|x) = L_{12}\,p(C_1|x) + L_{22}\,p(C_2|x)$

Goal: decide such that expected loss is minimized

i.e decide α_1 if $R(\alpha_2|x) > R(\alpha_1|x)$

Discriminant function

$y_1(x), \ldots, y_K(x)$ ← $\frac{p(x|C_1)}{p(x|C_2)} > \frac{(L_{21} - L_{22})\,p(C_2)}{(L_{12} - L_{11})\,p(C_1)}$ ←

classify x as class C_k if $y_k(x) > y_j(x) \quad \forall j \neq j$

$y_k(x) = \log p(x|C_k) + \log p(C_k)$

+++++→ x Data: $x_1 \ldots x_n$ estimate: $p(x)$

Method:
- Parametric representation
- Non parametric representation
- Mixture models

$N(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\}$

$N(x|\mu, \Sigma) = \frac{1}{(2\pi)^{D/2}\,|\Sigma|^{1/2}} \exp\left\{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right\}$

Given data $X = \{x_1, x_2, \ldots x_N\}$. Parametric form of the distribution with parameters θ.

E.g for Gaussian distribution $\theta = (\mu, \sigma)$

Learning - Estimate parameter θ

Likelihood of θ

Probability that the data X have indeed been generated from a probability density with parameter θ. $L(\theta) = p(X|\theta)$

MLE - Computation of the likelihood

- single data point: $p(x_n|\theta)$
- Assumption: all data points are independent

$L(\theta) = p(X|\theta) = \prod_{n=1}^{N} p(x_n|\theta)$

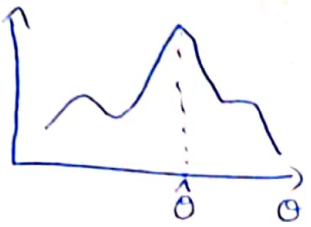
- Log likelihood $E(\theta) = -\ln L(\theta) = -\sum_{n=1}^{N} \ln p(x_n|\theta)$

- Estimate of the parameters θ
  - Maximize the likelihood
  - Minimize the negative log-likelihood

likelihood : $L(\theta) = p(X|\theta) = \prod_{i=1}^{N} p(x_n|\theta)$

- we want to obtain $\hat{\theta}$ such that $L(\hat{\theta})$ is maximized.

$p(X|\theta)$



Minimizing the log-likelihood.

$$\frac{\partial E(\theta)}{\partial \theta} = -\frac{\partial}{\partial \theta} \sum_{i=1}^{N} \ln p(x_n|\theta) = -\sum_{n=1}^{N} \frac{\frac{\partial}{\partial \theta} p(x_n|\theta)}{p(x_n|\theta)} \stackrel{!}{=} 0$$

Log likelihood for normal distribution.

$$E(\theta) = -\sum_{i=1}^{N} \ln p(x_n|\mu,\sigma) = -\sum_{n=1}^{N} \ln\left(\frac{1}{\sqrt{2\pi\sigma}} \cdot \exp\left\{-\frac{\|x_n-\mu\|^2}{2\sigma^2}\right\}\right)$$

Thus $\hat{\mu} = \frac{1}{N}\sum_{n=1}^{N} x_n \quad \hat{\sigma}^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n-\hat{\mu})^2 \quad \hat{\theta} = (\hat{\mu}, \hat{\sigma}^2)$

$$E(\mu_{ML}) = \mu \qquad E(\sigma^2_{ML}) = \left(\frac{N-1}{N}\right)\sigma^2$$

MLE
- → underestimate the variance of the distribution.
- → overfits the observed data.

Probability that x falls into small region R.

$P = \int_R p(y)dy$ $\quad P \approx p(x)V$ $\quad P = \frac{K}{N} \Rightarrow p(x) \approx \frac{K}{NV}$

Kernel method $\quad$ K nearest neighbour.

Parzen window

$$k(u) = \begin{cases} 1 & , |u_i| \leq \frac{1}{2}h \;, i = 1, \ldots D \\ 0 & , \text{else} \end{cases}$$

$$K = \sum_{n=1}^{N} k(x - x_n) \qquad V = \int k(u)du = h^D$$

Probability density estimate $\quad p(x) \approx \frac{K}{NV} = \frac{1}{Nh^D} \sum_{n=1}^{N} k(x - x_n)$

Gaussian kernel $\quad k(u) = \frac{1}{(2\pi h^2)^{1/2}} \exp\left\{-\frac{u^2}{2h^2}\right\}$

$$K = \sum_{n=1}^{N} k(x - x_n) \qquad V = \int k(u)du = 1$$

$$p(x) \approx \frac{1}{N} \sum_{n=1}^{N} \frac{1}{(2\pi)^{D/2} h} \exp\left\{-\frac{\|x - x_n\|^2}{2h^2}\right\}$$

K-nearest neighbour.

$p(C_j|x) = \frac{p(x|C_j)p(C_j)}{p(x)}$ $\quad p(x) \propto \frac{K}{NV}$ $\quad p(x|C_j) \approx \frac{K_j}{N_j V} \Rightarrow$

$p(C_j|x) \approx \frac{K_j}{N_j V} \cdot \frac{N_j}{N} \cdot \frac{NV}{K} = \frac{K_j}{K}$

$p(C_j) \approx \frac{N_j}{N}$

$$p(x|X) = \int \frac{p(x|\theta)L(\theta)p(\theta)}{\int p(X|\theta)p(\theta)d\theta} d\theta$$

Mixture models

- hard clustering
  - element either belong to cluster or not
- soft clustering
  - soft assignment
- Mixture models are probabilistically doing soft clustering.
- each cluster: a generative model (Gaussian or multinomial)
- parameters (e.g. mean / covariance are unknown)

⇒ EM algo used to find the parameter for k "sources"

- k = 2, Gaussian with unknown μ_i, σ_i^2
- If I know the source i can find μ_i, σ_i^2

If we know the source of the gaussian (μ_i, σ_i^2) (parameters) we can guess more likely from 'a' or 'b' using bayes

$$p(b|x_i) = \frac{p(x_i|b)\,p(b)}{p(x_i)}$$

$$p(x_i|b) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left\{-\frac{(x_i-\mu_b)^2}{2\sigma_b^2}\right\}$$

we can use EM

- start with random Gaussian (μ_a, σ_a^2) (μ_b, σ_b^2)
- for each point: $p(b|x_i)$ = does it look like it came from b?
- adjust (μ_a, σ_a^2) and (μ_b, σ_b^2) to fit points assigned to them

↳ E-step: soft assignment

$$\gamma_j(x_n) \leftarrow \frac{\pi_j N(x_n|\mu_j, \Sigma_j)}{\sum_{k=1}^{K} \pi_k \cdot N(x_n|\mu_k, \Sigma_k)}$$

↳ M-step: re-estimate the parameter (σ_a^2, μ_a)

$$\hat{N}_j \leftarrow \sum_{n=1}^{N} \gamma_j(x_n)$$

$$\mu_{j\,new} \rightarrow \frac{1}{\hat{N}_j} \cdot \sum_{n=1}^{N} \gamma_j(x_n)\, x_n$$

$$\Sigma_{j\,new} \rightarrow \frac{1}{\hat{N}_j} \cdot \sum_{n=1}^{N} \gamma_j(x_n)(x_n - \hat{\mu}_j^{new})(x_n - \hat{\mu}_j^{new})^T$$

GMM or MOG

$$p(x|\theta) = \sum_{j=1}^{M} p(x|\theta_j)\,p(j)$$

$$p(x|\theta_j) = N(x|\mu_j, \sigma_j^2) = \frac{1}{\sqrt{2\pi\sigma_j}} \exp\left\{-\frac{(x-\mu_j)^2}{2\sigma_j^2}\right\}$$

$$p(j) = \pi_j \quad 0 \le \pi_j \le 1$$

prior of component "j"

MLE: $\theta = -\ln L(\theta) = \sum_{n=1}^{N} \ln p(x_n|\theta)$

$$\mu_j = \frac{\sum_{n=1}^{N} \gamma_j(x_n)\, x_n}{\sum_{n=1}^{N} \gamma_j(x_n)}$$

$$\gamma_j(x_n) = \frac{\pi_j N(x_n|\mu_j, \Sigma_j)}{\sum_{k=1}^{K} \pi_k N(x_n|\mu_k, \Sigma_k)}$$

responsibility of component j for x_n (1)

k-mean

It optimise the following

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|x_n - \mu_k\|^2$$

$$r_{nk} = \begin{cases} 1 & \text{if } k = \arg\min_k \|x_n - \mu_k\|^2 \\ 0 & \text{otherwise.} \end{cases}$$

1. Initialisation: pick k arbitary centroids (cluster means)
2. Assign each sample to the closest centroid.
3. Adjust the centroid to be the means of the sample assigned to them
4. Go to step 2 (until no change)

Pros

→ simple & fast to compute
→ converges to local minimum of within-cluster squared error

Problem case

→ settings k?
→ sensitive to initial centres
→ Sensitive to outliers
→ Detect only spherical clusters

GMM

Properties

→ Very general, can represent any distribution
→ once trained, very fast evaluate
→ can be updated online

Problems

→ some numerical issues in implementation
⇒ need to apply regularisation to avoid singularities.
→ EM of MOG is computationally expensive
- especially for high-dimensional problems
- slower convergence than k-means
- Result sensitive to initialisation.

→ Need to select number of mixture component K.

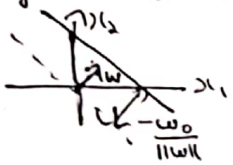
• Discriminant functions
$y_1(x), \dots y_K(x)$
$y_k(x) > y_j(x) \quad \forall j \neq j$
$y_k(x) = \log p(x|C_k) + \log p(C_k)$

$y_1(x) > y_0(x)$
$y(x) > 0$

2-class problem

$y(x) > 0$: decide for C_1 & C_2

$y(x) = w^T x + w_0 = \sum_{i=0}^{D} w_i x_i$
$y(x) = 0 \rightarrow$ decision boundary $x_0 = 1$



One-vs-all classifiers

heuristic method for using binary classification algorithm for multi-class.

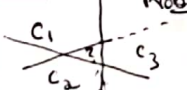
Ex:- R, B, G

Binary classifier: R V [B(G)]
" " 2 : B V [R(G)]
" " 3 : G V [R, B]

It require one model to be created for each class.

Solve
- Take K linear function of form $y_k(x) = w_k^T x + w_{k0}$
- decision boundary C_k iff $y_k > y_j \; \forall j \neq k$

Problem: 1 of K coding scheme
$t_n = (0,1,0,0,0)^T$



$C_{kj} = (w_j - w_k)^T x + (w_{j0} - w_{k0}) = 0$

one vs one multiclass

Ex: R, B, G, Y

Binary class 1 : R vs B
2 : R V G
3 : B V G
4 : R V Y
⋮

No. of binary dataset $\frac{N_{class} \times (N_{class}-1)}{2}$

Least square

- very sensitive to outliers
- Least square corresponds to Maximum likelihood under the assumption of a Gaussian conditional distribution.

K class

$y_k(x) = w_k^T x + w_{k0}, \quad k = 1, \dots K.$
$y_k(x) = \tilde{w}^T x$
$\tilde{w} = [\tilde{w}_1, \dots \tilde{w}_K]$
target $t = [t_1, \dots t_K]^T$

$Y(\tilde{x}) = \tilde{x}\tilde{w}$

$\tilde{x} = \begin{bmatrix} x_1^T \\ \vdots \\ x_N^T \end{bmatrix} \quad T = \begin{bmatrix} t_1^T \\ \vdots \\ t_N^T \end{bmatrix}$

$\tilde{x}\tilde{w} - T$

Least-square

Sum of square error

$E(w) = \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{K} (w_k^T x_n - t_{kn})^2$

Multiclass

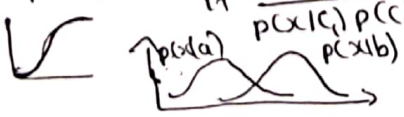
$E_D(\tilde{w}) = \frac{1}{2} Tr\{(\tilde{x}\tilde{w} - T)^T (\tilde{x}\tilde{w} - T)\}$

$\frac{2 E_D(\tilde{w})}{\partial \tilde{w}} = \tilde{x}^T (\tilde{x}\tilde{w} - T) \Rightarrow y(x) = \tilde{w}^T \tilde{x} = T^T (\tilde{x}^\dagger)^T \tilde{x}$

$\tilde{x}^\dagger$: pseudo inverse

$y(x) = g(w^T x + w_0)$ ← activation function maybe nonlinear.

$p(C_1|x) = \frac{1}{1 + \frac{p(x|C_2)p(C_2)}{p(x|C_1)p(C_1)}} = \frac{1}{1 + \exp(-a)} = g(a)$ → logistic sigmoid activation function



LDA II

If data is not linearly seperable, a linear decision boundary is still optimal.
Choice of discriminant function is based on
- Prior knowledge - empirical comparison of alternative method.

Generalized linear discriminants

Transform vector x with M nonlinear basis function $\phi_j(x)$: This basic function allows nonlinear boundaries.

$$y_k(x) = \sum_{j=1}^{M} w_{kj}\phi_j(x) + w_{k0} = y_k(x;w)$$

Single-layer network: ϕ_j is fixed, only weight w is learned.
Multi-layer network: both the w and the ϕ_j are learned.

GD (Gradient descent) → feature

Sample: $x = [x_1, x_2, \dots x_n]$ → single data point y [Training: find θ that maximize seeing training data] (MLE)

$x \in \mathbb{R}^d \quad y \in \{0,1\}$

$D = \{(x_i, y_i)\}_{i=1}^{n}$ we want to learn $\theta = [\theta_0, \theta_1, \dots \theta_d]$

$\hat{\theta}_{MLE} = \arg\max_{\theta} P(D;\theta) = \arg\max P(Y|X;\theta) = \arg\max \prod_{i=1}^{N} P(y_i|x_i;\theta)$

For logistic $P(y_i|x_i;\theta) = \text{Function}(\theta^T x_i)$ → Sigmoid function → $(-\infty,\infty) \to (0,1)$ (→ clink function)

$P(y_i=1|x_i;\theta) = \underbrace{\sigma(\theta^T x_i)}_{\text{sigmoid}} \to L(\theta) = \max_{\theta} \prod_{i=1}^{n} P(y_i|x_i;\theta) = \max_{\theta} \prod_{i:y_i=1} \sigma(\theta^T x_i) \prod_{i:y_i=0} (1-\sigma(\theta^T x_i))$

$$L(\theta) = \prod_{i=1}^{N} \sigma(\theta^T x_i)^{y_i} * (1-\sigma(\theta^T x_i))^{1-y_i}$$

$$l(\theta) = \max \sum_{i=1}^{N} y_i \log(\sigma(\theta^T x_i)) + (1-y_i)\log(1-\sigma(\theta^T x_i))$$

(negative log likelihood)
$NLL(\theta) = -l(\theta)$
we need to minize the log likelihood (loss minize)

$$NLL(\theta) = \min_{\theta} \sum_{i=1}^{N} y_i \log(\sigma(\theta^T x_i)) + (1-y_i)\log(1-\sigma(\theta^T x_i)) \Rightarrow \text{loss function} \quad (1)$$

optimizer used to find θ that minizes loss function. Here it is GD.
1st ordn optimers: GD, require first order derivative, jacobian matrix, name is based on jacobian
2nd ordn optimizers: Newton-Raphson, 2nd order derivative, Hessian matrix, name is based on Hessian ie move.

GD

$$\theta = \theta + \alpha \frac{\delta NLL}{\delta \theta^T} \to \text{loss change / unit change in weights.} \qquad \frac{d(\sigma(x))}{dx} = \sigma(x).(1-\sigma(x))$$

chain rule to (1) → $\frac{\delta NLL}{\delta \theta^T} = \sum_{i=1}^{n} (y_i - \sigma(\theta^T x_i)).x_i$, $\hat{\theta}^{MLE} = \theta$.

GD → iterate over m-iterations $\theta = \theta + \alpha \sum_{i=1}^{M} (y_i - \sigma(\theta^T x_i)) x_i$

$P(y=1|x) = \sigma(x;\theta=\hat{\theta}^{MLE}) = \frac{1}{1+e^{-(\hat{\theta}^{MLE})^T.x}}$ $\quad y = \begin{cases} 1; & P(y=1|x) \geq 0.5 \\ 0; & P(y=1|x) < 0.5 \end{cases}$

---

error function: $E(w) = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K} (y_k(x_n;w) - t_{kn})^2 = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K}\left(\sum_{j=1}^{M} w_{kj}\phi_j(x_n) - t_{kn}\right)^2$

$$w_{kj}^{(\tau+1)} = w_{kj}^{(\tau)} - \eta \left.\frac{\partial E(w)}{\partial w_{kj}}\right|_{w^{(\tau)}} \qquad \frac{\partial E(w)}{\partial w_{kj}} = (y_k(x_n;w) - t_{kn})\phi_j(x_n)$$

↳ learning rate

$$w_{kj}^{(\tau+1)} = w_{kj}^{(\tau)} - \eta \delta_{kn}\phi_j(x_n) \qquad \delta_{kn} = y_k(x_n;w) - t_{kn}$$

with activation function

$$\frac{\partial E_n(w)}{\partial w_{kj}} = (\ ) \Rightarrow w_{kj}^{(\tau+1)} = w_{kj}^{(\tau)} - \eta\, \delta_{kn}\phi_j(x_n) \quad | \quad \delta_{kn} = \frac{\partial g(a_k)}{\partial w_{kj}}(y_k(x_n;w) - t_{kn})$$

Probabilistic linear discriminad and gaussian method of modelling.

no. of parameter for gaussian → mean: M + covariance $\frac{M(M+1)}{2}$ → total $\frac{M(M+5)}{2}+1$ parameter (M dimension)

logistic regression parameter: just the values of $w \Rightarrow$ M parameters

consider dataset, $\{\phi_n, t_n\}$ with $n=1,\ldots,N$, where $\phi_n = \phi(x_n)$ and $t_n \in \{0,1\}$ $t=[t_1,\ldots t_N]^T$

Logistic regression

Classify the weather as rainy, sunny and partially cloudy.

$P(y=1|x) = p(x) \quad x \in R \quad p(x) \in [0,1]. \quad p(x) = \frac{1}{1+e^{-\beta x}}$

$\log\left(\frac{p(x)}{1-p(x)}\right) = \beta x$ goal → estimate $\hat{\beta}$

For class 1: estimate $\hat{\beta}$ such that $p(\beta)$ is close to 1 as possible.

For class 0: estimate $\hat{\beta}$ such that per $1-p(\hat{\beta})$ is close to 1 as possible.

$L(\beta) = \prod_{y_i=1} p(x_i) \times \prod_{y_i=0}(1-p(x_i)) \quad l(\beta) = \sum_{i=1}^{n} y_i \log\left(\frac{1}{1+e^{-\beta x_i}}\right) + (1-y_i)\log\left(\frac{e^{-\beta x_i}}{1+e^{-\beta x_i}}\right)$

maximize

log likelihood function $l(\beta)$ that needs to be optimized

$l(\beta) = \sum_{i=1}^{n} \cdot y_i \beta x_i - \log(1+e^{\beta x_i}) \quad \beta = \arg\max_{\beta} l(\beta)$

we consider Newton Raphson to compute β. First two terms of taylor series expansion.

$\nabla_\beta l(\beta) = \nabla_\beta l(\beta^*) + (\beta-\beta^*)\nabla_{\beta\beta} l(\beta^*)$ → Gradient w.r.t β.

$\nabla_\beta l(\beta^*) + (\beta-\beta^*)\nabla_{\beta\beta} l(\beta^*) = 0 \Rightarrow \beta^{t+1} = \beta^t - \frac{\nabla_\beta l(\beta^t)}{\nabla_{\beta\beta} l(\beta^t)}$ → Hessian matrix

$\nabla_\beta l = \sum_{i=1}^{n} \cdot [y_i - p(x_i)] x_i$

$\nabla_\beta l = x^T(y-\hat{y})$

$\nabla_{\beta\beta} l = -x^T p(1-p) x$

$= -x^T w x$

$\nabla_{\beta\beta} l(\beta^t) = \nabla_\beta \cdot \sum_{i=1}^{n} [y_i - p(x_i)] x_i$

$\nabla_{\beta\beta} l(\beta^t) = -\sum_{i=1}^{n} \cdot p(x_i)[1-p(x_i)] x_i^T x_i$

$\beta^{(t+1)} = \beta^{(t)} + (x^T w^{(t)} x)^{-1} \times (y - \hat{y}^{(t)})$ → Repeat for n times.

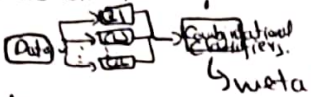
$p(x) = \frac{1}{1+e^{-\beta x}}$

Summary: Logistic regression

- Directly represent posterior distribution $p(\phi|C_k)$
- Require fewer parameters than modelling the likelihood + prior

Ensemble - Use same training algorithm several times on subset of data.
Well suited for unstable learning algorithms. Eg:- Decision trees, neural networks
stable learning algo - Nearest neighbour, linear regression.
5 B 5-fold cross-validation.. - Bagging - Bootstrap aggregation
$M=N$, 63.2%, Perform multiple runs of the algo with different random initialisation.

Stacking



↳ meta classifier.

- simplicity
- correlation b/w classifiers
- Feature combination

Model combination

Eg:- Mixture of models

$$p(x)=\sum_{k=1}^{K}\pi_k N(x|\mu_k,\Sigma_k)$$

Uncertainity which mixture component responsible for

Marginal probability of a data set $p(X)=\prod_{n=1}^{N}\sum_{k}^{K}\pi_k N(x_n|\mu_k,\Sigma_k)$

Bayesian model averaging:- Suppose we have H diffent models $h=1,\dots H$

Marginal distribution $p(x)=\sum_{h=1}^{H} p(x|h)\,p(h)$ $p(h)$- prior probability

Model combination → Different data generated by different model components.
Uncertainity is about which component created which data points
one latent variable z_n for each data point.

Bayesian model averaging $p(X)=\prod_{n=1}^{N} p(x_n)=\prod_{n=1}^{M}\sum_{z_n} p(x_n,z_n)$

→ The whole data set is generated by a single model.
→ Uncertainity is about which model was responsible.
→ one latent variable z for the entire data set.

$$y_{com}(x)=\frac{1}{M}\sum_{m=1}^{M} y_m(x) \qquad y(x)=h(x)+\epsilon(x) \quad \text{error, true value}$$

$$p(X)=\sum_{z} p(X,z)$$

Sum of square errors $E_x=[\{y_m(x)-h(x)\}^2]=E_x[\epsilon_m(x)^2]$

Average error individual models $E_{AV}=\frac{1}{M}\sum_{m=1}^{M} E_x[\epsilon_m(x)]^2$

Average error of committe $E_{com}=E_x\left[\left\{\frac{1}{M}\sum_{m=1}^{M} y_m(x)-h(x)\right\}^2\right]=E_x\left[\left\{\frac{1}{M}\sum_{m=1}^{M}\epsilon_m(x)\right\}^2\right]$

$E_{com}=\frac{1}{M}E_{AV}$ assumption: $E_x[\epsilon_m(x)]=0$ → zero mean

$E_x\{\epsilon_m(x)\epsilon_j(x)\}=0$ errors are uncorrelated

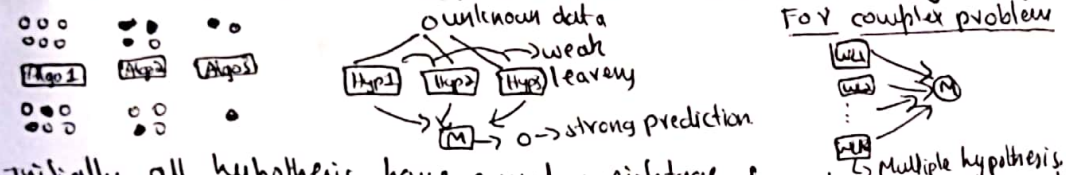
---

Ensemble method - simple method for improving classification

Apparent contradiction
- classifiers should be trained on sample from the distribution on which it will be tested.
- Resampling seems to violate this recommendation.

Explanation
- we do not attempt to model the full category distribution here.
- Try to find the decision boundry more directly.
- Increasing number of component classifiers broadens the class of implementable decision functions.

Boosting - combine multiple rules of thumb to make accurate decision.
Is problem learnable? - Probably Approximately Correct mode (PAC)
chances of getting bad data? lower than threshold $1-\delta$?
PAC - A problem is PAC learnable, if learning algorithm can find solution with some error ϵ with probability greater than $(1-\delta)$
Strong learner - error < threshold (ϵ), probability $> 1-\delta$.
Ex:- sound recognition, lot's of data, lot's of parameter and hardware requirement.
what if we don't have good hardware, or less dataset and how do we learn now?
That's where weak learners comes into picture.
If a strong learner can solve a problem, a collection of weak learners can do it too,
↳ This is based on hypothesis boosting mechanism.
Hypothesis: Represent what model has learned.
For: Ex:- in supervised learning; final equation with weights $\cdot y = b + w_1x_1 + \dots + w_px_p$
Instead we construct 3 hypothesis and finall do majority vote.



Initially all hypothesis have equal weightage to make a majority voting.
But what if each hypothesis added their <u>important changes</u> or adapt depending on the <u>errors of new hypothesis added</u>. This is where adaptive boosting comes in. (Adaboost)
Normal hypothesis boosting - No weight associated with it.
Adaboost - weighted.
Step1 :- Initialise weight of each sample to be the same.
Step2 : Construct the first hypothesis by training on these datapoints.
Step3 : Determine training error of these hypothesis and update the weights.
weight update - Increase weight of misclassified sample,
- Decrease weight of correctly classified sample.
update the weight of hypothesis itself.
First iteration - since first hypothesis weight = 1., As iteration goes more hypothesis is added and similarly weights are updated.
If not enough hypothesis is added then partially boosted ~~bagg~~ model won't have good performance to capture pattern. so boosting prone to underfitting.

Adaboost

1) Initialise weights to all training points. $w = \frac{1}{N}$ E.g:- If I have 10K datapoint each datapoint have weight 1/10K
2) Calculate error rate for each weak classifier $\epsilon = \sum_{\text{wrong}} w_i$ It will grow multiple decision stumps or small decision trees calculate error by each decision tree.
3) Pick classifier or decision stump with the lowest error rate.
4) Now ~~voting~~ compute voting power for the classifier $\alpha = \frac{1}{2} \log \frac{1-\epsilon}{\epsilon}$
(This gives the classifier the weak classifier strength in terms of how much is it more accurate as compared to the other weak classifier.)
5) Append the classifier into my ensemble classifier of $f(x)$. Then question is asked is $f(x)$ good enough for making a strong prediction or have I gone around multiple rounds to create a very strong classifier. or are there any good classifier left that I can create. To check whether I have completed enough round is to check if the best classifier has an error rate of one by two. (1/2)

- once it is evaluated and you feel that there are multiple more rounds that you have to conduct in term of creating more weak classifiers with some voting power. Then next step is to update the weights.

↓

6) Update weights of each point where the previous classifier went wrong.

$$w_{new} = \begin{cases} \frac{w_{old}}{2(1-\epsilon)} & \text{if point classified correctly (weight} \downarrow) \\ \frac{w_{old}}{2\epsilon} & \text{if point classified wrongly (wait} \uparrow) \end{cases}$$

7) Go back to step 2.

$X = \{x_1, \dots x_N\}$ → Training data
$T = \{t_1, \dots t_N\}$, $t_n \in \{-1, 1\}$ target value
$w = \{w_1, \dots w_N\}$

1. Initialisation: Set $w_n^{(1)} = \frac{1}{N}$ for $n = 1, \dots N$
2. For $m = 1, \dots M$ iterations.

a) Train a new weak classifiers $h_m(x)$ using the current weighting coefficients $W^{(m)}$ by minimizing the weighted error function.

$$J_m = \sum_{n=1}^{N} w_n^{(m)} I(h_m(x) \neq t_n)$$

$h_m(x)$ → weak classifier

$$I(A) = \begin{cases} 1 & \text{if A is true} \\ 0 & \text{else} \end{cases}$$

b) Estimate the weighted error of this classifier on X:

$$\epsilon_m = \frac{\sum_{n=1}^{N} w_n^{(m)} \cdot I(h_m(x) \neq t_n)}{\sum_{n=1}^{N} w_n^{(m)}}$$

c) Calculate a weighting coefficients for $h_m(x)$: $\alpha_m = \ln\left(\frac{1-\epsilon_m}{\epsilon_m}\right)$

d) Update the weighting coefficients:

$$w_n^{(m+1)} = w_n^{(m)} \exp\{\alpha_m I(h_m(x_n) \neq t_n)\}$$

e) Make final prediction using the final combined model $H(x) = \text{sign}\left\{\sum_{m=1}^{M} \alpha_m h_m(x)\right\}$

Exponential error function $E = \sum_{n=1}^{N} \exp\{-t_n f_m(x_n)\}$

where $f_m(x)$ is a classifier (linear combination of base classifier $h_l(x)$)

$$f_m(x) = \frac{1}{2} \sum_{l=1}^{m} \alpha_l h_l(x)$$

Goal:- Minimize E w.r.t α_l and $h_l(x)$

$$\frac{dE}{d h_m(x_n)} = 0$$

$$E = \sum_{n=1}^{N} w_n^{(m)} \exp\left\{-\frac{1}{2} t_n \alpha_m h_m(x_n)\right\}$$

correctly classified points: $t_n h_m(x_n) = +1$
$t_n h_m(x_n) = -1$ → misclassified

Adaboost minimizes exponential error in a sequential fashion.

Limitations:

Original Adaboost sensitive to misclassified training data points.
- Because of exponential error function • Improved by GentleBoost.

Two class classifier.
- Multiclass extensions available.

Decision tree:-
Each node specifies a test for some attribute.
Each branch corresponds to a possible value of the attributes.

Limitations of decision trees
⇒often produce noisy or weak classifiers.
⇒Do not generalize too well.
⇒Training data fragmentation:
- As tree progresses, splits
⇒Overtraining and undertraining
- Deep trees: fit the training data well, will not generalize well to new test data.
- Shallow trees: not sufficiently refined.
⇒ Stability
- Trees are sensitive to details of the training points.
- If a single data point is only slightly shifted a radically different tree may come out.
⇒Expensive learning step.
- due to costly selection of optimal split.

Ensemble Combination
Combine o/p of several trees by averaging their posteriors

$$p(C|x) = \frac{1}{L} \sum_{l=1}^{L} \cdot P_{l,\eta}(C|x)$$

(l, η) → Tree leaves

CART framework.
→ 6 Q → Binary or multi? → which property
→ when node is leaf? → how tree can be pruned?
→ How to deal impure nodes → how missing attributes handled?

Decision tree complexity
Data points $\{x_1, \cdots x_N\}$ - dimensionality D
Storage : $O(N)$, Test runtime : $O(\log N)$
Training runtime : $O(DN^2 \log N)$
- Most expensive part.
- Critical step: selecting the optimal splitting point.

Best way to split the data.
→ Don't look for global optimal split.
→ Instead randomly use subset of K attributes on which to base the split.
→ Choose best splitting attribute.
Eg maximizing the information gain

$$\Delta E = \sum_{k=1}^{K} \cdot \frac{|S_k|}{|S|} \cdot \sum_{j=1}^{N} p_j \log_2(p_j)$$

Bootstrap + Decision tree → Bagged decision tree → Random forest.

Bootstrap- subsampling with replacement.

Ex:- [1, 2, 3, ... N] · M_1, M_2 (sample of subsample)

$M_b = \frac{\sum_{i=1}^{n} M_i}{N}$ (N-subsample)

Why bootstrap? average over average reduces variance of the predicted mean.

B Decreasing variance of prediction will reduce overfitting.

Decision tree → i/p sample
categorical o/p → classification tree
continuous o/p → Regression trees

CART → classification and regression trees helps to construct predictive model of data.

Bagging:
"Bootstraped aggregation."
"Bootstraped to high variance algorithm.
high variance- Different training samples leads to different results.

high variance model is decision trees
Different data fed into can change the structure of the data drastically.
we can reduce it with the help of bagging

Bagged decision tree

Consider 1000 samples let's say we want to train B classifier. For each of B classifier we split the dataset with replacement (Bootstrap). Then each classifier get 300 samples. Now construct B decision tree using corresponding samples. We pass the data to all tree and majority o/p rules.

In case of regression, it takes the mean of all decision tree outputs.

Why are we not pruning individual trees? won't they exhibit a high variance. Yes they will we don't care because results are aggregated anyways. This aggregation lead to decrease in variance.

Why use bagging in high variance ML algo?
High variance ML algo the B decision trees will have different structures and hence will predict differently. Hence aggregate prediction will decrease the variance of prediction.
However if we apply it to low variance ML algo like logistic regression then all B classifier look the same.

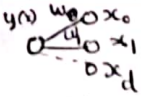
Random forest

The problem with bagged decision tree each B node can be split ~~into~~ on the same P features. This could lead to structural similarities blw the trees. and hence correlation. However we want our model has to be different and d correlated as possible.
Hence random forest are like bagged decision trees. but every tree can split based on subset of features M. this subset of features is different for each classifier. For classification tree with P features every tree should be able to split on $\sqrt{P}$ features.
For aggression trees each tree should be able to split on P/3 features.

$m = \begin{cases} \sqrt{P} & \text{classification} \\ P/3 & \text{regression} \end{cases}$

Suppose we have 1000 samples and we pick 600 of them then we left with 400 datapoints. that decision tree haven't seen. These 400 samples are called out of bag samples. (OOB).
OOB can be used to validate the random forest classifier.

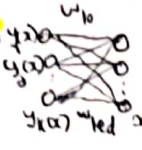
Standard perceptron



- o/p layer
- weights
- i/p layer

i/p layer → Hand designed features.
o/p layer → Linear o/p → $y(x) = w^T x + w_0$
Logistic o/p → $y(x) = \sigma(w^T x + w_0)$

Multi-class perceptron



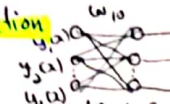
o/p layer
- weights
- i/p layer

Learning = Determining the weights w.

o/p layer. Linear o/p → $y_k(x) = \sum_{i=0}^{d} w_{ki} x_i$

Logistic o/p → $y_k(x) = \sigma\left(\sum_{i=0}^{d} w_{ki} x_i\right)$

Non-linear basis function



- o/p layer
- weights
- feature layer
- mapping
- i/p layer

o/p → Linear o/p $y_k(x) = \sum_{i=0}^{d} w_{ki} \phi(x_i)$

Logistic o/p $y_k(x) = \sigma\left(\sum_{i=0}^{d} w_{ki} \phi(x_i)\right)$

Perceptron learning: → If o/p unit is correct, leave the weights alone.
→ If the o/p unit incorrectly o/p a zero, add the i/p vector to the weight vector.
→ If the o/p unit incorrectly o/p a one, subtract the i/p vector from the weight vector.

This is guaranteed to converge to a correct solution if such solution exist.

$w_{kj}^{(\tau+1)} = w_{kj}^{(\tau)} - \eta(y_k(x_n; w) - t_{kn})\phi_j(x_n)$ → Delta rule a.k.a LMS rule

Loss function:

L2 Loss $L(t, y(x)) = \sum_n (y(x_n) - t_n)^2$ Least square regression

L1 loss $L(t, y(x)) = \sum_n |y(x_n) - t_n|$ Median regression

Cross-entropy loss $L(t, y(x)) = -\sum \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\}$

Hinge loss $L(t, y(x)) = -\sum_n [1 - t_n y(x_n)]_+$ SVM classification.

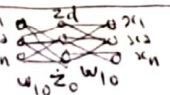
Softmax loss $L(t, y(x)) = -\sum_n \sum_k \{I(t_n - k) \ln \frac{\exp(y_k(x))}{\sum_j \exp(y_j(x))}\}$

Regularization: $E(w) = \sum_n L(t_n, y(x_n; w)) + \lambda \|w\|^2$ → weight decay → Avoid overfitting

Limitation of perceptron

→ Given right i/p feature perceptron can train any separable function.
→ For some task exponential number of i/p feature is required.
→ XOR problem.
→ Non linear classifier based on the right kind of features, the problem become solvable.

Multi-layer perceptron:



- o/p layer
- hidden layer
- mapping
- i/p layer

o/p $y_k(x) = g^{(2)}\left(\sum_{i=0}^{d} w_{ki}^{(2)} g^{(1)}\left(\sum_{i=0}^{d} w_{ij}^{(1)} x_j\right)\right)$

Activation function: $g^{(2)}(a) = \sigma(a)$, $g^{(1)}(a) = a$.

Need an efficient way of adapting all weights, not just the last layer.
Learning weights of hidden layer = learning features. → Gradient descent.

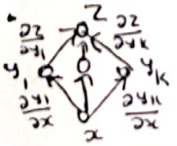
Error function: $E(w) = \sum_n L(t_n, y(x_n; w)) + \lambda \Omega(w)$ $L(.)$ → loss
$\Omega(.)$ → Regularize.

GD
1) obtain gradient for each weight
2) Adjust to new weight

$L(t_n, y(x_n; w)) = \sum_n (y(x_n; w) - t_n)^2$ → L2-loss.

$\Omega(w) = \|w\|_F^2$ → L2-regularizer → update weight $\frac{\partial E(w)}{\partial w_{ij}^{(1)}}$

Compute gradient for each weight $\left(\frac{\partial E(w)}{\partial w_{10}^{(2)}} \cdots \frac{\partial E(w)}{\partial w_{1n}^{(2)}}\right) \left(\frac{\partial E(w)}{\partial w_{10}^{(1)}} \cdots \frac{\partial E(w)}{\partial w_{nd}^{(1)}}\right)$ → Naive analytical differentiation



$\frac{\partial z}{\partial x} = \sum_{i=1}^{k} \frac{\partial z}{\partial y_i} \cdot \frac{\partial y_i}{\partial x}$

→ With increase in depth, there will be exponentially many paths.
→ infeasible to compute this way

So we go for 3rd approach → Backpropagation.

Numerical differentiation

Given current state $w^{(\tau)}$ make small changes to $w^{(\tau)}$ and accept those that improve $E(w^{(\tau)})$

Horribly inefficient! Need several forward passes for each weight. Each forward pass is one run over the entire dataset.

Gradient descent

Two main steps

1. Computing the gradient for each weight
2. Adjusting the weights in the direction of the gradient

$$w_{ij}^{(\tau+1)} = w_{ij}^{(\tau)} - \eta \left.\frac{\partial E(w)}{\partial w_{ij}}\right|_{w(\tau)}$$

Batch learning vs stochastic learning.

Batch learning

- Process the full dataset at once to compute the gradient.

Stochastic learning

- Choose a single example from the training set
- Compute the gradient only based on this example
- This estimate will generally be noisy, which has some advantage.

Batch learning advantage

- Conditions of convergence are well understood.
- Many acceleration techniques (e.g. conjugate gradient) only operate in batch learning
- Theoretical analysis of the weight dynamics and convergence rates are simpler.

Stochastic learning advantage

- Usually much faster than batch learning
- often results in better solutions.
- Can be used for tracking changes.

Minibatches

Idea

- Process only a small batch of training example together.
- Start with a small batch size & increase it as training proceeds.

Advantage:
- Gradient will more stable than for stochastic gradient descent, but still faster to compute than with batch learning.
- Take advantage of redundancies in the training set.
- Matrix operations are more efficient than vector operations.

Caveats:
- Error function should be normalised by the minibatch size, s.t. we can keep the same learning rate b/w minibatches.

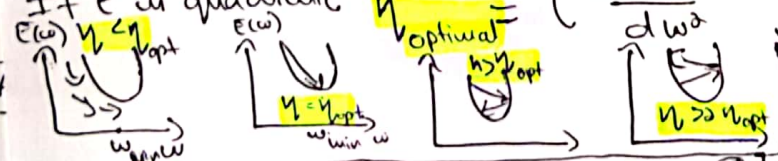
$$E(w) = -\frac{1}{N}\sum_n \cdot L(t_n, y(x_n; w)) + \frac{\lambda}{N}\Omega(w)$$

Choose right learning rate

analysing the convergence of Gradient descent.

Ex:- $w^{(\tau-1)} = w^{(\tau)} - \eta \frac{dE(w)}{dw}$ what is the optimal η_{opt}?

If E is quadratic $\eta_{optimal} = \left(\frac{d^2 E(w^{(\tau)})}{dw^2}\right)^{-1}$ (inverse of hessian)



Why learning is slow?

If the i/p are correlated.
- The ellipse will be very elongated
- The direction of steepest descent is almost ⊥ to the direction towards minimum.

The momentum method

- **Idea:** Instead of using the gradient to change the position of the weight "particle" use it to change velocity.

Intuition: Ball rolling on error surface.
- It starts off by following the error surface, but once it has accumulated momentum, it no longer does steepest descent.

Implementation of Momentum method:

Change in the update equations
- Effect of the gradient: increased the previous velocity, subject to a decay by $\alpha < 1$

$$v(t) = \alpha v(t-1) - \epsilon \frac{\partial E}{\partial w}(t)$$

set weight change to the current velocity

$$\Delta w = v(t) = \alpha v(t-1) - \epsilon \frac{\partial E}{\partial w}(t)$$
$$= \alpha \Delta w(t-1) - \epsilon \frac{\partial E}{\partial w}(t)$$

Idea of RMSdrop

- Divide the gradient by a running average of its recent magnitude

$$\text{MeanSq}(w_{ij}; t) = 0.9\,\text{MeanSq}(w_{ij}; t-1) + 0.1\left(\frac{\partial E}{\partial w_{ij}}(t)\right)^2$$

- Divide the gradient by sqrt(MeanSq(w_ij; t))

Backpropagation → Idea: Compute the gradient layer by layer.
→ Each layer below builds upon the results of the layer above.
→ The gradient is propagated backwards through the layers.

Step:
1) Convert the discrepancy b/w each o/p and its target value into an error derivate

$$E = \frac{1}{2} \sum_{j \in o/p} (t_j - y_j)^2$$

2) Compute error derivative in each hidden layer from error derivatives in the layer above. $\frac{\partial E}{\partial y_i} = -(t_i - y_i)$

3) Use error derivatives w.r.t activities to get error derivatives w.r.t the incoming weights.

y_i → o/p of layer j
z_j → i/p of layer j

connection $z_j = \sum_i w_{ij} y_i$

$y_i = g(z_j)$

$\frac{\partial E}{\partial y_i} \rightarrow \frac{\partial E}{\partial w_{ik}}$

$\frac{\partial E}{\partial z_j} = \frac{\partial y_i}{\partial z_j} \frac{\partial E}{\partial y_j} = y_j(1-y_j) \frac{\partial E}{\partial y_i}$

$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial z_j}{\partial y_i} \frac{\partial E}{\partial z_j} = \sum_j w_{ij} \frac{\partial E}{\partial z_j}$

Forward pass

$y^{(0)} = x$
for $k = 1, \ldots, l$ do
$z^{(k)} = w^{(k)} y^{(k-1)}$
$y^{(k)} = g_k(z^{(k)})$
end for
$y = y^{(l)}$
$E = L(t, y) + \lambda \Omega(w)$

Backward pass

$h \leftarrow \frac{\partial E}{\partial y} = \frac{\partial}{\partial y} L(t, y) + \lambda \frac{\partial \Omega}{\partial y}$
for $k = l, l-1, \ldots, 1$ do
$h \leftarrow \frac{\partial E}{\partial z^{(k)}} = h \odot g'(y^{(k)})$
$\frac{\partial E}{\partial w^{(k)}} = h y^{(k-1)T} + \lambda \frac{\partial \Omega}{\partial w^{(k)}}$
$h \leftarrow \frac{\partial E}{\partial y^{(k-1)}} = w^{(k)T} h$
end for

Factoring paths

Problem: Combinatorial explosion

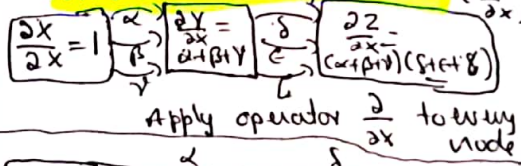
Ex:

$\frac{\partial Z}{\partial X} = (\alpha+\beta+\gamma) * (\delta+\epsilon+\zeta)$

Efficient algorithms for computing the sum.
- Instead of summing over all the paths explicitly, compute the sum more efficiently by merging paths back together at every node.

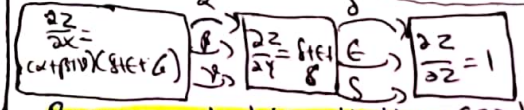
This is what we want in backpropagation!
- Forward differentiation needs one pass per node. With backward differentiation can compute all derivatives in one single pass.
- Speed up in O(#inputs) compared to forward differentiation

Forward mode differentiation $\left(\frac{\partial}{\partial x}\right)$



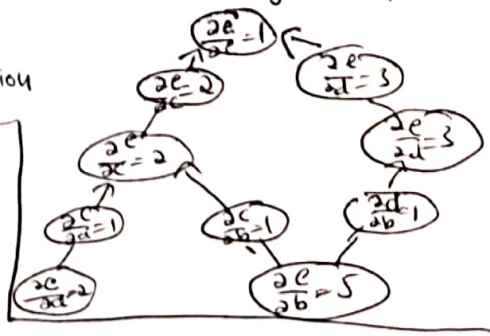
Apply operator $\frac{\partial}{\partial x}$ to every node

Reverse mode differentiation $\left(\frac{\partial Z}{\partial}\right)$



Apply operator $\frac{\partial Z}{\partial}$ to every node

Why do we care?
Let's consider the example again
- Using reverse-mode differentiation from e up ....
- Runtime: O(#edges)
- Result: derivative of e with respect to every node.



Automatic differentiation:-

Softmax output

$$E(w) = -\sum_{n=1}^{N} \sum_{k=1}^{K} \{ I(t_n = k) \ln \frac{\exp(w_k^T x)}{\sum_{j=1}^{K} \exp(w_j^T x)} \}$$

Practical issues

1) Exponentials get very big and can have vastly different magnitude
- Trick 1: Do not compute first softmax, then log, but instead directly evaluate log exp in the denominator.
- Trick 2: Softmax has the property that for a fixed vector b
softmax(a+b) = softmax(a)

RL: An agent by interacting with the environment and obtaining a reward signal for its actions.
Cart Pole balancing, context-Aware object hand-over, opening doors.
Supervised learning:- gwin i/p x and labels/outputs y. Learn: a mapping $f(x)=y$.
unsupervised learning:- i/p x, some Learn: some structure about the data
Reinforcement learning:- Given: a set of experiences Learn: how to behave so that the reward is maximised.
A reinforcement modelled as Markov decision Process.

MDP(S, A, T, R), return R, $J = E[\sum_t r(a_t, s_t)]$

Policy $\pi : S \to A$, parameter π_θ, trajectory $\tau = (s_0, a_0, s_1, \ldots a_n, s_{n+1})$

$P_\pi(\tau) = P(s_0) \prod_{i=0}^{n} P_\pi(a_i|s_i)\, P(s_{i+1}|s_i, a_i)$ (episode or rollout)

Stochastic policy -> sampling from distribution of action state $a_t \sim \pi(a|s_t)$

Deterministic policy -> Fixed, $a_t = \pi(s_t)$ Eg:- Grasping a door. Eg:- stroke in tennis

Finite & infinite horizon tasks -> finite tasks/episodic ends after a certain number of time steps or when a goal is reached. Eg: Navigation

Infinite task -> goes on (ex)y

$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)$ discontinued by a factor $\gamma \in [0,1]$

Return, $R(\tau) = \sum_{t=0}^{T} r(s_t, a_t)$

RL objective -> Maximize the agents expected return $\pi^* = \arg\max_\pi E_{\tau\sim\pi}\left[\sum_t r(s_t, a_t)\right]$

↓ find: optimal policy for agent. Maximize the parameterised policy

$\theta^* = \arg\max_\theta E_{\tau\sim\pi_\theta}\left[\sum r(s_t, a_t)\right] = \arg\max_\theta \int P(\tau|\pi) R(\tau)\, d\tau$

Trasition function T + reward function r = model

-> If T and r are known in advance, model-based RL (exploit the model knowledge)

-> If T and r are unknown, model-free RL (learn using trial error)

we often prefer combination of both

Exploration: Acting by trying out action that may not be optimal one.

Exploitation: Acting awarding to the best policy.

Tradeoff: if the agent exploits too much, it risk of converging to sub-optimal policy. (may not see entire environment)

if the agent do exploration, it may not known when to stop or converge.

Off policy: choose policy different from what it has learned. (Randomly choose policy)(slow)

on policy: an agent sample actions from the policy that it is trying to learn. (

Value function: (It tells if it's good to be in this state and not the other)

State value function: $V^\pi(s)$ => It is the expected return if we start in state s_0 and then act awarding to the policy. $V^\pi(s) = E[R(\tau)|s_0 = s]$

State action value function: $Q^\pi(s,a)$ (evaluates both state and action) (Q-function)

=> Expected return for agent starting at state s_0 and action a and act subsequently.

$Q^\pi(s,a) = E_{\tau\sim\pi}[R(\tau)|s_0=s, a_0=a]$.

Optimal state value function (w.r.t policy π): $V^*(s) = \max_\pi V^\pi(s)\ \forall s \in S$

optimal state-action value function: $Q^*(s,a) = \arg\max_\pi Q^\pi(s,a)\ \forall s \in S,\ \forall a \in A(s)$

$V^*(s) = \max_{a\in A(s)} Q^*(s,a)$. $a^* = \arg\max_a Q^*(s,a)$ => consequence.

By having optimal state-action we know how good each state is and how good a policy is.

Advantage function: $A(s,a) = Q(s,a) - V(s)$ -> Tells how good specific action is.

used to reduce variance of exploration.

Model based learning: Aim: find optimal value function

$$V^*(s) = \max_\pi E\left[\sum_{t=0}^{\infty} \cdot \gamma^t \cdot r(s_t, a_t)\right]$$

We can find **optimal value function** by solving **Bellman equations**.

$$V^*(s) = \max_a \underbrace{[r(s,a)}_{①} + \gamma \sum_{s'} \underbrace{P(s'|s,a) V^*(s')}_{②}] \quad ②$$

we need to find how good a state is. First we need **immediate reward** ① then we have **discounted expectation of** how good the **subsequent states** are. ②

Optimal policy is then $\pi^*(s) = \arg\max_a \left(r(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s')\right)$

Two algorithms: 1) Value iterations 2) Policy iterations. (solving algorithm for optimal value function)

Value iteration: uses directly bellman equation and solves for them. (iterative algorithm) start with random initialisation of $V(s)$, $\forall s \in S$, then iterate over the equations. Every state we maximize over the action. solve until convergence → Eg:- $\forall s \in S, |V^i(s) - V^{i-1}(s)| \leq \epsilon$ for small ϵ.
It is guranteed to converge to an optimal policy.

Policy iteration: Here we modify the policy directly. so it is called policy iteration.
Here we randomly initialise a policy π and then solve the system of equation.

$$V^{\pi_i}(s) = r(s, \pi_i(s)) + \gamma \sum_{s'} P(s'|s,a) V^{\pi_i}(s')$$

Update the policy

$$\pi_{i+1}(s) = \arg\max_a \left(r(s,a) + \gamma \sum_{s'} P(s'|s,a) V^{\pi_i}(s')\right)$$

Model free learning:-
Setup: while learning collect trajectories τ and the anompaning rewards.
use this setup to find optimal policy.
Two types of algorithms: **Temporal difference learning**: perform value/policy update at every step.
Montecarlo learning: estimate the return and then perform value/policy updates.

Temporal difference - TD(λ)-Learning: In TD(λ) we try to bring value function closer to the reward function.
The parameter λ controls the amount of prediction during learning.

$V(s_t) = V(s_t) + \alpha (r(s_t, a_t) + \gamma V(s_{t+1}) - V(s_t))$, where α is a learning rate.

Q learning: Here we try to estimate the state-action value function.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left(r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right)$$

Policy search (PS)
Here we don't need to find value function explicitly rather we would optimise policy space directly. Useful for parameterised policies. They help to incorporate prior knowledge about the policy.
Three type of PS algorithms:

Policy gradient method: we represent policy as differentiable function then we use gradient based method to optimise.

Information-theoretic algo: we want to do policy updates donot destroy the previous policies. We try to bound the policy updates.

Expectation-maximisation-based method: Frame policy search as expectation maximization and then solve EM.

Policy gradients

Modify policy parameter θ by calculating the gradient of the expected result. $\theta \leftarrow \theta + \nabla J(\theta)$ $J(\theta) = \int R(\theta) P(\tau|\theta) d\theta$

Policy gradient algorithms make use of likelihood ratio trick.

$$\nabla_\theta P(\tau|\theta) = P(\tau|\theta) \nabla_\theta \log P(\tau|\theta)$$

Reinforce algorithm

Initialise policy parameter θ randomly

for $i \leftarrow 1$ to N do (let agent explore for n iteration)
  $\tau \rightarrow \{\}$ collect trajectories.
  for $j \leftarrow 1$ to M do
    $\tau \leftarrow$ sample(π_θ) sample M trajectories
    $\tau \leftarrow \tau \cup \tau$ → append

$J_\theta \leftarrow \frac{1}{M} \sum_{j=1}^{M} \sum_{t} r_t^j$

$\theta \leftarrow \theta + \nabla J_\theta$

on ith iteration we estimate expected return
collect M trajectories,
estimate expected return.
estimate gradient of return.
update policy parameters
repeat for N iteration

Actor-Critic Learning

Value based algorithms - critic based
Policy search algorithms - actor based
we combine both of them together to have actor-critic family of RL.
which estimate the value function and maintain a policy at the same time.
Actor critic algorithms make use of baseline b when estimating the gradient of J.

$$\nabla_\theta J(\theta) = E\left[\sum_{t=0}^{b} \nabla_\theta \log P_\theta(a_t|s_t)(R_t - b_t)\right]$$

why do we do baseline → we usually observe high variance of policy. baseline try to reduce b variance.