Linear Regression:
- Normal eqns, L_2/L_1 regularization
- Auto Regressive models
- Convex linear/logistic objective

Classification:
- K-NN
- logistic regression; convex objective

Optimisation:
- Gradient descent, momentum, nesterov
- Newton
- Conjugate gradients

Neural Networks:
- ~~Auto diff~~ Autoencoder, PCA
- LSTM
- Initialisation

Auto diff:
- forward; complex & dual
- Backward
- Backprop & parameter tying

Dimension Reduction:
- PCA; eigendecomposition & SVD, ICA
- Fishers LDA

Clustering:
- K-means
- Mixture models; GMM
- Spectral clustering

Fast Nearest Neighbours:
- distance metric
- Orchard, AESA
- buoys
- K-d tree

Linear Regression

$$E(\underline{w}) = \sum_i (y_i - \underline{w}^T \underline{x}_i)^2$$ ← convex as $\underline{\underline{H}}$ positive definite ($\underline{z}\underline{\underline{H}}\underline{z} > 0$)

$O(D^3)$ time, $O(D^2)$ space → $\underline{\underline{X}}^T \underline{\underline{X}}\, \underline{w} = \underline{\underline{X}}^T \underline{y}$

$$\hat{\underline{w}}_{L_2} = (\underline{\underline{X}}^T \underline{\underline{X}} + \lambda \underline{\underline{I}})\, \underline{\underline{X}}^T \underline{y}$$ + analytical form solution; - not sparse; try L_1

- Auto-regressive models:
  - time series $v_1, v_2, \ldots, v_t$
  - predict v_t using $v_{t-1:t-L} = \underline{\phi}_t$
  - can use least squares / L_1/L_2 regularisation

$$E(\underline{\alpha}) = \sum_t (v_t - \underline{\phi}_t^T \underline{\alpha})^2$$

Classification

- K-NN: vote of k closest using some distance metric
- Logistic regression:

$$P(c=1 \mid \underline{x}) = \sigma(\underline{w}^T \underline{x}), \quad \sigma(x) = \frac{e^x}{1+e^x}, \quad \sigma(-x) = 1 - \sigma(x), \quad \sigma'(x) = \sigma(x)(1-\sigma(x))$$

$$L(\underline{w}) = \sum \log P(c_i \mid \underline{x}_i) = \sum (c_i \log \sigma(\underline{x}_i^T \underline{w}) + (1-c_i) \log (1 - \sigma(\underline{w}^T \underline{x}_i)))$$

- optimise w/ gradient descent methods

Optimisation

$O(DN)$ time, $O(D+N)$ space

- Gradient Descent: $\underline{w}_{k+1} = \underline{w}_k - \varepsilon \nabla E(\underline{w})\big|_{\underline{w}_k}$
  - error falls as $1/T$
- Momentum: $\underline{w}_{k+1} = \underline{w}_k + \tilde{g}_{k+1}$, $\tilde{g}_{k+1} = \mu_k g_k - \varepsilon \nabla E(\underline{w})\big|_{\underline{w}_k}$
  - moving average gradient
- Nesterov:
  - momentum, but 'smarter' gradient MA update

$$\tilde{g}_{k+1} = \mu_k \tilde{g}_k - \varepsilon \nabla E(\underline{w})\big|_{\underline{w}_k + \mu_k \tilde{g}_k}$$

simple, but coord system dependent

$O(D^2)$ per iteration

- Newton: $\underline{w}_{k+1} = \underline{w}_k - \varepsilon \underline{\underline{H}}^{-1}_{E(\underline{w})} \nabla E(\underline{w})|_{\underline{w}_k}$
  - invariant under coord transform, but $\underline{\underline{H}}$ expensive to compute & invert ($O(D^3)$)
  - 1 step convergence for quadratic $E(\underline{w})$

- Conjugate gradients:
  $\underline{p}_i \underline{\underline{A}} \underline{p}_j^T = \delta_{ij} \underline{p}_1 \underline{\underline{A}} \underline{p}_1^T \Rightarrow \underline{p}_1, \underline{p}_2, \ldots$ conjugate vectors for $\underline{\underline{A}}$

can do analytically for quadratic objective

$k = 1$

$\underline{w}_k = \ldots, \quad \underline{p}_k = -\nabla E(\underline{w})|_{\underline{w}_k}$

while $\nabla E(\underline{w})|_{\underline{w}_k} \neq 0$:

line search along $\underline{p}_k$ → $\alpha_k = \arg\min_\alpha E(\underline{w} + \alpha \underline{p}_k)$

update $\underline{w}$ ← $\underline{w}_{k+1} = \underline{w}_k + \alpha_k \underline{p}_k$

find new conjugate gradient → $\underline{p}_{k+1} = -\nabla E(\underline{w})|_{\underline{w}_{k+1}} + \beta_k \underline{p}_k$

$k += k+1$

- max D iterations

Neural Networks

$\underline{x} \to \underline{h}_1 \to \underline{h}_2 \to \ldots \to \underline{y}$

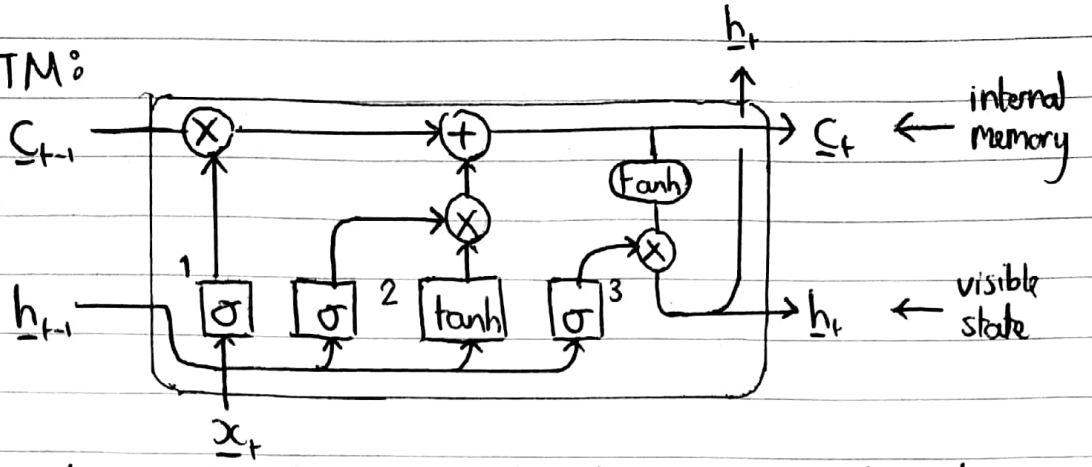
- $\underline{y} = \sigma_L \xi(\underline{x}|\underline{w}) = \sigma_L(\underline{\underline{W}}_L \underline{h}_{L-1})$, $\underline{h}_l = \sigma_l(\underline{\underline{W}}_l \underline{h}_{l-1}), l = 2, \ldots, L-1$
  $\underline{h}_1 = \sigma_1(\underline{\underline{W}}_1 \underline{x})$
  - optimize by gradient descent
    - compute gradient w/ autodiff
  - Autoencoder:
    - bottleneck hidden layer w/ few nodes
    - goal is to reconstruct input
    - with single hidden layer & σ_1 linear equivalent to PCA

$\hat{\underline{y}} = \underline{\underline{W}}_2 \underline{\underline{W}}_1 \underline{y} = \underline{\underline{W}}_2 \underline{\underline{W}}_1 \underline{\underline{U}} \underline{\underline{\Sigma}} \underline{\underline{V}}^T$ ← SVD of $\underline{y}$

hidden layer size K

to minimize square loss, take largest K singular values of $\underline{\underline{U}}$ → $\Rightarrow \underline{\underline{W}}_1 \underline{\underline{W}}_2 = \underline{\underline{U}}_K \underline{\underline{U}}_K^T$, $\hat{\underline{y}} = \underline{\underline{U}}_K \underline{\underline{U}}_K^T \underline{y}$, same as PCA

- LSTM:



1: forget gate, 2: input gate, 3: output gate

- Special architecture for RNN, to help deal with exploding/vanishing gradients & long term memory
- train w/ backpropagation through time

- Initialisation:
  - aim to keep error $O(1)$
  - scale layer inputs to mean 0, variance 1 $\left(x_i = \frac{x_i - \mu_i}{\sigma_i}\right)$
  - sample weights w/ mean 0, variance $= 1/D$ ← size of input $\underline{x}$
    - keep output mean 0, 1 variance
  - Decorrelating:
    - makes opt Hessian approx diagonal, can ease optimisation

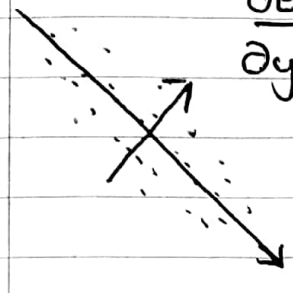
$\underline{x}^n = \underline{\underline{S}}^{-1} \underline{\underline{U}}^T \underline{x}^n$, where $\underline{\underline{X}}\,\underline{\underline{X}}^T = \underline{\underline{U}}\,\underline{\underline{S}}^2\,\underline{\underline{U}}^T$ (so SVD of $\underline{\underline{X}}$)

Auto Diff: given $f(\underline{x})$, find $\left.\frac{\partial f}{\partial x_i}\right|_{\underline{x}}$

- Forward:
  - Complex arithmetic: approximate
    $$f'(\underline{x}) = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \mathrm{Im}(f(x + i\epsilon))$$
  - Dual arithmetic: exact
    $$f'(\underline{x}) = \text{Dual Part}(f(x + \epsilon))$$
  - both numerically inefficient & need to overload functions to work w/ dual/complex arithmetic
- Backward:
  - Find reverse schedule of nodes
  - Starting from first node N, (in reverse), $t_N = 1$
  - Recurse back, $t_n = \sum_{c \in ch(n)} \frac{\partial f_c}{\partial f_n} t_c$ ($ch(n)$ = children of n)
  - t at root nodes gives total derivative wrt these variables
  - efficient; $\leq 5\times$ complexity of $f(\underline{x})$!

Dimension Reduction:

· PCA:



- Minimise $E(\underline{\underline{B}}) = \sum_{n,d} (x_d^n - \sum_{j=1}^{M} y_j^n b_i^j)^2$

$$\frac{\partial E}{\partial y_R^n} = 0 \Rightarrow y_R^n = \sum_d x_d^n b_d^n \Rightarrow \underline{\underline{y}} = \underline{\underline{B}}^T \underline{\underline{X}}$$

$$E(\underline{\underline{B}}) = tr((\underline{\underline{X}} - \underline{\underline{B}}\,\underline{\underline{y}})^T (\underline{\underline{X}} - \underline{\underline{B}}\,\underline{\underline{y}})), \quad \underline{\underline{B}}^T\underline{\underline{B}} = \underline{\underline{I}}$$

$$= tr(X^T(\underline{\underline{I}} - \underline{\underline{B}}\underline{\underline{B}}^T)^2 X) = (XX^T(\underline{\underline{I}} - \underline{\underline{B}}\underline{\underline{B}}^T))$$

$$E(\underline{\underline{B}}) = tr(\underline{\underline{S}}) - tr(\underline{\underline{S}}\,\underline{\underline{B}}\,\underline{\underline{B}}^T)(N-1)$$

- Minimise $-tr(\underline{\underline{S}}\,\underline{\underline{B}}\,\underline{\underline{B}}^T)$, enforce $\underline{\underline{B}}^T\underline{\underline{B}} = \underline{\underline{I}}$ w/ Lagrange Multipliers

$$\Rightarrow \underline{\underline{S}}\,\underline{\underline{B}} = \underline{\underline{B}}\,\underline{\underline{L}}$$

- enforce $\underline{\underline{L}}$ diagonal $\Rightarrow$ cols of $\underline{\underline{B}}$ are eigenvectors of $\underline{\underline{S}} = \frac{1}{N-1}\underline{\underline{X}}\underline{\underline{X}}^T$
- eigendecomposition of $\underline{\underline{S}}$ $O(D^3)$, faster to do:

$$\underline{\underline{X}} = \underline{\underline{U}}\,\underline{\underline{D}}\,\underline{\underline{V}}^T, \quad \underline{\underline{B}} = \underline{\underline{U}} \qquad (\text{SVD is } \min(O(DN^2), O(D^2N))$$

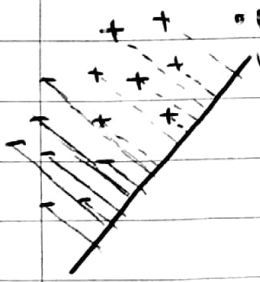
- for sparse data, can iterate

$$\underline{x} = \underline{\underline{S}}\,\underline{x}, \text{ converges at principle eigenvector}$$

- for missing data,

$$E(\underline{\underline{B}}) = \sum_{nd} \gamma_d^n \left(x_d^n - \sum_{j=1}^{M} y_j^n b_i^j\right)^2$$

· Fisher's Linear Discriminant:



- $y^n = \underline{w}^T \underline{x}^n$, project to minimise class overlap

(c: mean of class)

$$y_c \sim N(\mu_c, \sigma_c^2) \Rightarrow F(\underline{w}) = \frac{(\mu_1 - \mu_2)^2}{\pi_1\sigma_1^2 + \pi_2\sigma_2^2}$$

$$\mu_c = \underline{w}^T \underline{m}_c$$
$$\sigma_c^2 = \underline{w}^T \underline{\underline{S}}_c \underline{w}$$
$$\pi_c = N_c/N$$

$$F(\underline{w}) = \frac{\underline{w}^T \underline{\underline{A}}\,\underline{w}}{\underline{w}^T \underline{\underline{B}}\,\underline{w}} \qquad \left(\begin{array}{l}\underline{\underline{A}} = (\underline{m}_1 - \underline{m}_2)^T(\underline{m}_1 - \underline{m}_2) \\ \underline{\underline{B}} = \pi_1 \underline{\underline{S}}_1 + \pi_2 \underline{\underline{S}}_2\end{array}\right)$$

$$\Rightarrow \frac{\partial F}{\partial \underline{w}} = 0 \Rightarrow (\underline{w}^T \underline{\underline{B}}\,\underline{w})\,\underline{\underline{A}}\,\underline{w} = (\underline{w}^T \underline{\underline{A}}\,\underline{w})\,\underline{\underline{B}}\,\underline{w}$$

$$\Rightarrow \underline{w} \propto \underline{\underline{B}}^{-1}(\underline{m}_1 - \underline{m}_2)$$

Clustering

- Can use k-means:
  - hard clustering, scales poorly, requires spherical clusters w/ equal density
- Mixture models:
  - $p(\underline{x}) = \sum_c p(\underline{x}|c) \cdot p(c)$
  - eg GMM: $\underline{x}|c \sim N(\underline{\mu}_c, \underline{\underline{S}}_c^2)$ ($\underline{\underline{S}}$ (PSD, symmetric))
  - learn $\underline{\mu}_c, \sigma_c^2$ by maximising likelihood (EM algorithm)

$$L = \sum_n \log \sum_c p(c) \cdot p(\underline{x}|c) \qquad \left(p(\underline{x}|c) = \frac{1}{\sqrt{\det(2\pi \underline{\underline{S}}_c)}} \exp\left(\frac{1}{2} (\underline{x}^n - \underline{m}_c)^T \underline{\underline{S}}_c^{-1} (\underline{x}^n - \underline{m}_c)\right)\right.$$

  - need to stop $\underline{\underline{S}}_c$ being too small, else $p(\underline{x}|c)$ can become infinite for $\underline{x}_c = \underline{\mu}_c$
  - Gaussians sensitive to outliers; use t-distributions
- Spectral clustering:
  - data as markov chain

$$p(i|j) = \frac{A_{ji}}{\sum_k A_{jk}} \qquad (A_{ji} \text{ is similarity of } j \text{ \& } i)$$

  - find equilibrium dist & cluster with this (k-means)

Fast Nearest Neighbour

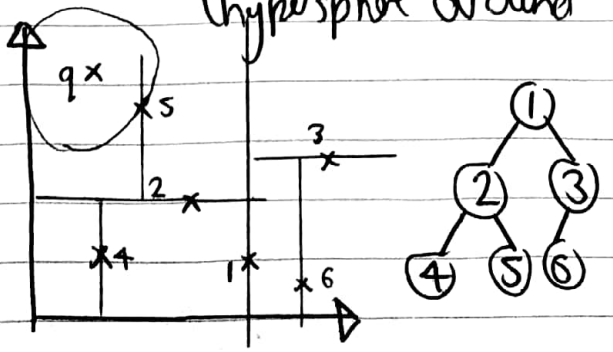
- Metric distance:
  - $d(x,y) > 0$
  - $d(x,y) = d(y,x)$
  - $d(x,y) = 0 \Rightarrow y = x$
  - $d(x,y) \leq d(x,z) + d(z,y)$
- Orchard:
  - precompute distance matrix & ordered list of neighbours for each $\underline{x}^i$
  - for query $\underline{q}$:
    - pick initial $\underline{x}^i$, step along it's list to $\underline{x}^j$
    - if $d(\underline{x}^j, \underline{q}) < d(\underline{x}^i, \underline{q})$, go to $\underline{x}^j$'s list
    - if $d(\underline{x}^j, \underline{q}) \geq \frac{1}{2} d(\underline{x}_i, \underline{q})$, stop early, return $\underline{x}^i$
- AESA:
  - precompute distance matrix
  - $\mathcal{I}$ = set of $d(\underline{q}, \underline{x}^i)$ calculated already
  - pre-eliminate $\underline{x}^j$ if: (lower bound on $d(\underline{q}, \underline{x}^j)$)

$$\min\{d(\underline{q},\underline{x}^i)\} \leq \max_{i \in \mathcal{I}} \{d(\underline{q},\underline{x}^i) - d(\underline{x}^i,\underline{x}^j)\}$$

- Buoys:
  - small set of points $\underline{b}^1, \ldots, \underline{b}^B$
  - compute distance matrix only for buoys
  - for $\underline{x}^j$:

$$U_j = \min_m \{d(\underline{q},\underline{b}^m) + d(\underline{b}^m,\underline{x}^j)\}$$
$$L_j = \max_m \{d(\underline{q},\underline{b}^m) - d(\underline{b}^m,\underline{x}^j)\}$$

  - pre-eliminate all points with $U_j > \min_i\{L_i\}$
  - then feed to AESA / Orchard
- k-d tree:
  - recursively divide space, alternating dimensions
  - query given root up, checking if need to check siblings (hypersphere around ~~sibling already seen~~ query)



q → check 5
skip 4
check 2
skip 3
check 1 ⇒ return 5

COMPGI09 - Applied ML

1a: Intro to Supervised Learning

utility ↓ loss ↓ (U = −L)

producing a class label given x^* → decision function, $c(x^*)$

- Loss: cost of making an error
- for input x^*, we learn predictive distribution $P(c|x^*)$
- Expected utility $= U(c(x^*)) = \sum_{c^{true}} U(c^{true}, c(x^*)) \cdot P(c^{true}|x^*)$
  - try to maximise this
- How to define $P(c|x^*)$ & $c(x^*)$:
  - Empirical Risk: empirical dist $P(c|x^*)$, Parametric $c(x^*|\theta)$
  - Bayesian Decision: non-trivial $P(c|x^*)$, unconstrained $c(x^*)$
- Utility functions:
  - ~~Utility~~ General utility matrix:

$$U_{ij} = U(c^{true} = i, c^{pred} = j)$$

can be v. asymmetric; eg cancer diagnosis

$$U(c^* = j) = \sum_i U_{ij} \cdot P(c^{true} = i | x^*)$$

- Squared Loss:

$$L(y^{true}, y^{pred}) = (y^{true} - y^{pred})^2$$

$$L(y^{pred}) = \int (y^{true} - y^{pred})^2 P(y^{true}|x) dy^{true}$$

(maximised by $y^{pred} = \int y^{true} P(y^{true}|x) dy^{true}$)

- Choice of loss can influence ease of training; logistic regression model for classification w/ log loss is convex, squared loss is non-convex

- Empirical Risk Approach:
  $D = \{(x^n, c^n), n = 1 \dots N\}$
  - approximate $P(c|x)$ by empirical dist, $P(c,x|D) = \frac{1}{N}\sum_n \delta(c,c^n)\delta(x,x_n)$

$$\Rightarrow \langle U(c,c(x))\rangle_{P(c,x|D)} = \frac{1}{N}\sum_n U(c_n, c(x_n))$$ ← Empirical expected utility

- For x_n^* in train set, $c(x_n) = x_n$, but this x have $c(x)$ undefined! outside train set
- For x outside train set:
  - use parametric function $c(x|\theta)$
    - eg 2 class problem, linear classifier

$$c(x|\theta) = \begin{cases} 1 & \text{if } \vec{\theta}^T\vec{x} + \theta_0 \geq 0 \\ 2 & \text{o/w} \end{cases}$$

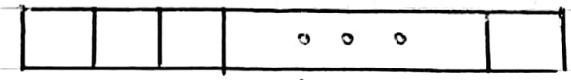
$$R(\theta|D) = \frac{1}{N}\sum_n L(c_n, c(x_n|\theta))$$

$$\theta_{opt} = \arg\min_\theta R(\theta|D)$$

- Overfitting:
  - too flexible $c(x|\theta)$ overfits training data
  - add penalty term, $P(\theta)$ (eg $P(\theta) = \|\vec{\theta}\|^2$)
    - minimise $R'(\theta|D) = R(\theta|D) + \lambda P(\theta)$

λ set by validation

- Cross-validation:



sp

  - split data into k folds
  - train on k-1 folds, test on held out fold
  - repeat k times & average loss/accuracy/... ~~to close~~

- Pros & Cons:
  - \+ large training data, empirical dist tends towards true
  - \+ discriminant function chosen to minimise risk
  - − poor for smaller amount of train data
  - − need to retrain for a different loss function
  - − no inherent confidence in prediction

- Bayesian Decision Approach:
  - fit model $P(c, x|\theta)$ to train data (eg maximum likelihood)
    - could overfit, maybe use regularization
  - determine $c(x)$ ~~gives~~ by maximizing expected utility

$$U(c(x^*)) = \sum_c U(c, c(x^*)) P(c|x, \theta)$$
$$c(x^*) = \underset{c(x^*)}{\operatorname{argmax}}\ U(c(x^*))$$

- ~~Discriminative~~ Generative Approach:

$$p(c, x|\theta) = P(x|c, \theta_{x|c}) \cdot P(c|\theta_c)$$ (class generates x)

  + easy to specify prior in $P(x|c)$
  - doesn't directly target classification model, $P(c|x)$; can be non-trivial to find $P(c|x)$ now if data is complex

- Discriminative Approach:

$$P(c, x|\theta) = P(c|x, \theta_{c|x}) \cdot P(x|\theta_x)$$

  + directly addresses $P(c|x)$, modelling decision boundary (which may be far easier to model than distribution of data within a class)
  - hard to include prior knowledge

- Pros & Cons:
  + conceptually very good, trying to model environment independent of decision process
  + if $P(x, c|\theta)$ correct, optimal
  - if $P(x, c|\theta)$ poor, prediction could be highly inaccurate
    - often use regularization for $P(x, c|\theta)$ to help avoid this

16: Maths Regresher

- Linear Algebra $A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$

$[A+B]_{ij} = a_{ij} + b_{ij}$

$[AB]_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$ (if $AB=BA$, A & B commute)

$I_{ij} = \delta_{ij}$, $\delta_{ij} = \begin{cases} 1 & \text{if } i=j \\ 0 & \text{o/w} \end{cases}$ $I = \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix}$

$[B^T]_{kj} = b_{jk}$ $(ABC)^T = C^T B^T A^T$

- Vectors: can be $n \times 1$ column matrix $\vec{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$

$\vec{w} \cdot \vec{x} = \sum w_i x_i = \vec{w}^T \vec{x}$

$|\vec{x}|^2 = \vec{x}^T \vec{x} = \sum_i x_i^2$

- $\det(A^T) = \det(A)$ $\det(I) = 1$
  $\det(AB) = \det(A)\det(B)$ $\det(A^{-1}) = \frac{1}{\det(A)}$

Inverse → $A^{-1}A = I = AA^{-1}$ (if A^{-1} doesn't exist, A is singular)
$(AB)^{-1} = B^{-1}A^{-1}$

Linear system → $A\vec{x} = \vec{b} \Rightarrow \vec{x} = A^{-1}\vec{b}$

$X = [\vec{x}_1, \cdots, \vec{x}_n]$, Rank is max # linearly independent cols or rows

- $\vec{x}_1, \cdots, \vec{x}_m$ linearly independent if only solution to $\sum_i \alpha_i \vec{x}_i = \vec{0}$ is $\alpha_i = 0$ for all i

$A^T A = I = AA^T \Rightarrow A^T = A^{-1}$, A is orthogonal

eigenvectors ↓ eigenvalues

$A\vec{e} = \lambda\vec{e}$, for $n \times n$ matrix, n eigenvalues & vectors

$(A - \lambda I)\vec{e} = \vec{0}$

$\Rightarrow \lambda$ is eigenvalue if $\det(A - \lambda I) = 0$

$A = \sum_{i=1}^{n} \lambda_i \vec{e}_i \vec{e}_i^T$ eigenvectors orthogonal to each other

$A = E \Lambda E$ (E matrix of eigenvectors, $\Lambda = \text{diag}(\lambda_i)$

Singular value decomposition $\rightarrow X = USV^T$ $U: n \times n$, $U^T U = I_n$ $X: n \times p$

$V: p \times p$, $V^T V = I_p$ $S: n \times p$ & diagonal, all +ve, & ordered (largest top-left)

if $\forall \vec{x}. \vec{x}^T A \vec{x} > 0 \Rightarrow A$ positive definite
($\vec{x} \neq \vec{0}$)

$\text{trace}(A) = \sum a_{ii} = \sum \lambda_i$ ~~$\forall \log\det(A)$~~

$\det(A) = \prod \lambda_i$

$\text{trace}(\log A) = \log\det(A)$

1b: Maths Refresher

- Calculus:

$$\frac{df}{dx} = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta} = f'(x)$$

numerical approximation ↓

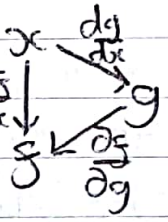
$$\left(\frac{df}{dx} \approx \frac{f(x+\delta)-f(x)}{\delta} + O(\delta^2) \approx \frac{f(x+\delta)-f(x-\delta)}{2\delta} + O(\delta^3)\right)$$

Taylor series → $f(x) = f(0) + x\frac{df}{dx} + \frac{x^2}{2}\frac{d^2f}{dx^2} + \cdots$

chain rule → $\frac{df(g(x))}{dx} = \frac{df(y)}{dy}\Big|_{y=f(x)} \frac{dg}{dx} = \frac{df}{dx}\frac{dg}{dx}$

$$\frac{d}{dx}(f+g) = \frac{df}{dx} + \frac{dg}{dx}, \quad \frac{d}{dx}(fg) = f\frac{dg}{dx} + g\frac{df}{dx}$$

$$\frac{df}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial g}\frac{dg}{dx}$$



, where $f(x)$ includes $g(x)$

Jacobian → $\nabla f(\vec{x}) = \vec{g}(\vec{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$, points along direction of max change

Hessian → $H_f(\vec{x}) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$

$$\frac{\partial}{\partial A}\operatorname{trace}(AB) = B^T$$

$$\partial \log\det A = \partial \operatorname{trace}(\log A) = \operatorname{trace}(A^{-1}\partial A)$$

$$\frac{\partial}{\partial A}\log\det A = A^{-T}$$

$$\partial A^{-1} = -A^{-T}\partial A A^{-1}$$

- Convex analysis:
  - $f(\vec{x})$ convex if, for any $\vec{x}, \vec{y}$, $0<\lambda<1$,

$$f(\lambda\vec{x}+(1-\lambda)\vec{y}) \leq \lambda f(\vec{x})+(1-\lambda)f(\vec{y})$$

(or if Hessian is positive definite for all points)

  - only have 1 minimum & that is only stat point
  - f, g convex $\Rightarrow$ $f+g$, ~~convex~~ $f(Ax+b)$ convex
    $f(g(x))$ convex if f increasing function

- Numerical issues:
  - mathematical identities may not hold on computer, need to check & fix at key points in code
  - eg $S=\exp(a)+\exp(b)$ may well overflow, leaving $\log(S)$ undefined
    - use logsum exp, ~~approx~~

$$\log(S)=a+\log(1+\exp(b-a)) \quad \text{(if } a \geq b\text{, in general take the max)}$$

$$\text{logsumexp}(\vec{x})=m+\log\left(\sum \exp(x_i-m)\right), \quad m=\max(\vec{x})$$

- Distributions:
  - Multi-variate Gaussian

$$p(\vec{x}\mid\vec{\mu},\Sigma)=\frac{1}{\sqrt{\det(2\pi\Sigma)}}\exp\left[\frac{-1}{2}(\vec{x}-\vec{\mu})^T\Sigma^{-1}(\vec{x}-\vec{\mu})\right]$$

    - Σ is covariance, & symmetric
      $\Rightarrow \Sigma=E\Lambda E^T$ (eigendecomposition)
    - can view multivariate Gaussian as shifted, scaled & rotated standard Gaussian

decorrelating → $\vec{y}=\Sigma_x^{-1/2}(\vec{x}-\vec{\mu}_x)\sim N(\vec{0}, I)$

2: Regression

- $S = \{(\vec{x}_i, y_i), i = 1, \dots, N\}$ sampled iid from $P(\vec{x}, y)$
  want to learn mapping from $\vec{x}$ to y

- Least Squares:
  $$E(\vec{w}) = \sum_i (y_i - \vec{w}^T \vec{\phi}_i), \quad \vec{\phi}_i = (1, \vec{x}_i)^T = \vec{\phi}(\vec{x}_i)$$

  $\hat{w}$, minimiser of $E(\vec{w})$, satisfies $(X^T X)\hat{w} = X^T \vec{y}$

  $$X = \begin{pmatrix} \vec{x}_1^T \\ \vdots \\ \vec{x}_n^T \end{pmatrix}, \quad \vec{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

  - can fit polynomial w/ different basis functions, eg for cubic $\vec{\phi}(x) = (1, x, x^2, x^3)^T$

- Regularisation:
  - should first scale input dimensions to unit variance
    $$x_{id} \leftarrow \frac{x_{id}}{\sigma_d}, \quad \sigma_d^2 = \frac{1}{N}\sum_i (x_{id} - \mu_d)^2, \quad \mu_d = \frac{1}{N}\sum_i x_{id}$$

  - now minimise $E'(\vec{w}) = E(\vec{w}) + \lambda R(\vec{w})$
  - L_2-Regularisation:
    $$R(\vec{w}) = |\vec{w}|^2 = \vec{w}^T\vec{w} = \sum w_i^2$$
    $$\hat{w} = \left(\sum_i \phi_i (\phi_i)^T + \lambda I\right)^{-1} \sum_i y_i \phi_i$$

    + easy analytic solution
    - doesn't encourage sparse solutions; prefers many small coefficients (eg colinear x, keeps both)

  - L_1-Regularisation
    $$R(\vec{w}) = |\vec{w}| = \sum w_i$$

    + encourages sparse solutions
    - non-differentiable; need special optimisation routines (only at origin, can just proceed & not worry about it sometimes)
    - doesn't deal w/ co-linear x, but not usually an issue

- Auto-Regressive models:
  - time series of $v_1, \ldots, v_{t-1}$
  - predict v_t using $v_{t-1}, \ldots, v_{t-L}$

$$v_t \approx \sum_{i=1}^{L} a_i v_{t-i} = \vec{a}^T \vec{\phi}_t, \quad \vec{\phi}_t = (v_{t-1}, \ldots, v_{t-L})^T$$

  - can train as before, w/ LS & L_1/L_2-regularisation
  - Probabilistic view:

$$v_t = \vec{a}^T \vec{\phi}_t + \eta_t, \quad \eta_t \sim N(\eta_t | \mu, \sigma^2)$$ 'innovation level'

  - L^{th} order Markov model

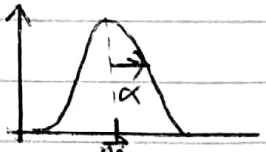
$$P(v_{1:T}) = \prod_{t=1}^{T} P(v_t | v_{t-1}, \ldots, v_{t-L})$$

$$P(v_t | v_{t-1}, \ldots, v_{t-L}) \sim N(v_t | \vec{a}^T \vec{\phi}_t, \sigma^2)$$

  - $L(\vec{a}) = \sum_{t=1}^{T} P(v_t | \vec{\phi}_t) =$

maximised by $\vec{a} = \left(\sum_t \vec{\phi}_t \vec{\phi}_t^T\right)^{-1} \sum_t v_t \vec{\phi}_t$

- Radial basis functions:
  - eg $\phi_i(\vec{x}) = \exp\left(\frac{-1}{2\alpha^2}(\vec{x} - \vec{m}_i)^2\right)$



  - adding together these, with weights, allows function fitting
  - curse of dimensionality:
    - if in 1D need 16 RBF to work well, need 16^n in n-dimensions for same performance
    - solutions: let centres move (similar to neural networks
      - or use Gaussian processes

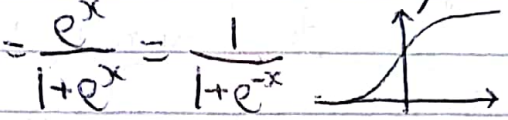
3: Classification

- $S = \{(\vec{x}_i, y_i), i = 1, \dots, n\}$, y_i discrete class labels
- Generative models:
  - k-NN:
    - to classify $\vec{x}^*$, look at k nearest points in train set & do majority vote
    - distance
      - squared Euclidean: $d(\vec{x}, \vec{x}') = (\vec{x} - \vec{x}')^T (\vec{x} - \vec{x}')$
        - if length scales of dif components of $\vec{x}$ vary largely, will be dominated by one, losing information
      - Mahalanobis distance: $d(\vec{x}, \vec{x}') = (\vec{x} - \vec{x}') \Sigma^{-1} (\vec{x} - \vec{x}')$ (Σ: covariance of all inputs)
        - + rescales components of $\vec{x}$
    - + simple, smooth decision boundary
    - − need to store whole training set, distance calculation slow, unclear how to deal w/ missing data or incorporate priors
  - Probabilistic view:
    - each train point is centre of Gaussian w/ width σ^2
    - $\sigma^2 \to 0 \Rightarrow$ only nearest neighbour has effect, $\sigma^2 > 0 \Rightarrow$ others have effect
    - training point classified by looking at three Gaussians generated by training points
    - gives even smoother k-NN
    - ~~if neighbour far,~~
    - − far neighbours classified very confidently
  - Naive Bayes: $P(c|\vec{x}^*) = \frac{P(\vec{x}^*|c)P(c)}{P(\vec{x}^*)} = \frac{P(\vec{x}^*|c)P(c)}{\sum_c P(\vec{x}^*|c)P(c)}$
    - + very fast to train; can just count for discrete attributes. Deals with missing data easily
    - − overly confident with low counts (solution: add pseudo counts for each class)

- Discriminative Models:
  - parametric classification: classify $\vec{x}$ with $c(\vec{x}|\vec{\theta})$, learn $\vec{\theta}$ from data
    - set $\vec{\theta}$ to minimize loss on train set,
      eg $L(c,\hat{c}) = \begin{cases} 0 & \text{if } c=\hat{c} \\ 1 & \text{otherwise} \end{cases}$ (zero-one loss) (complex error surface, hard to train)
  - Logistic Regression:

$$P(c=1|\vec{x}) = \sigma(b+\vec{x}^T\vec{w}), \quad P(c=0|\vec{x}) = 1-P(c=1|\vec{x})$$

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



for multiclass, change to softmax $g(x) = \frac{e^x}{\sum_j e^{x_j}}$

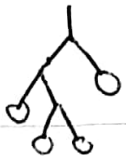
- Decision boundary:
  - $\vec{x}$ where $P(c=1|\vec{x}) = P(c=0|\vec{x}) \Rightarrow b+\vec{x}^T\vec{w} = 0$
  - linearly separable if all of each class lies on either side of decision boundary
    - if not linearly separable, use of basis function can help

log likelihood (assume data iid) →

$$L(\vec{w},b) = \sum_i c_i \log \sigma(b+\vec{w}^T\vec{x}_i) + (1-c_i)\log(1-\sigma(b+\vec{w}^T\vec{x}_i))$$

- optimize by gradient ascent or Newton (faster)
  - concave, so will reach global optimum
- if linearly separable, weights will forever grow; need to stop early or add penalty term

- Support Vector Machines:
  - classes $\in \{-1, 1\}$, $c(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w}^T\vec{x}+b \geq 0 \\ -1 & \text{if } \vec{w}^T\vec{x}+b < 0 \end{cases}$
  - goal: minimise $\frac{1}{2}\vec{w}^T\vec{w}$
    subject to $y_i(\vec{w}^T\vec{x}_i+b) \geq 1, \quad i=1,\dots,n$
  - if data not linearly separable:
    - minimise $\frac{1}{2}\vec{w}^T\vec{w} + \frac{1}{2}C\sum_i \xi_i^2$
    subject to $y_i(\vec{w}^T\vec{x}+b) \geq 1-\xi_i, \quad i=1,\dots,n$
- Kernels:
  - if use basis function $\phi(\vec{x})$, only need to be able to calculate $\langle \phi(\vec{x}_1), \phi(\vec{x}_2) \rangle$; can skip the basis function and just use a kernel; ie an implicit basis function,
  - effectively pergams SVM in higher (or infinite) dimensional space



- Decision trees:
  - learn series of decisions, such that data at each node as 'pure' as possible
  + fast to train & interpretable
  - unstable & limited
- Ensemble:
  - combine predictions of several classifiers
  - Bagging:
    - generate B datasets by sampling (with replacement) from original
    - train B models on these, & average or vote for predictions
    - Random forest is roughly bagging applied to decision trees

- Summary:
  - ~~Distr~~ Generative:
    - K-NN + simple, intuitive
      - slow (O(n)), need good distance function, struggle w/ missing data
    - Naive Bayes + fast to train & apply, easily deals w/ missing data
      - weak
  - Discriminative:
    - Logistic Regression + simple & fast to train, fast to apply (O(p)), concave optimisation, can use kernels, scales well
      - struggles w/ missing data
    - SVM (non-differentiable objective fn) + good generalisation, fast to apply, concave
      → doesn't scale well, struggles w/ missing data
    - Decision Tree + fast to train, interpretable,
      - ~~str~~ weak, struggles w/ missing data
    - Random Forest + fast to train, good performance
      - less interpretable, struggle w/ missing data

4: Optimisation

- Gradient Descent & first order methods:

$$\vec{x}_{k+1} = \vec{x}_k + \epsilon \nabla f(\vec{x}_k) \qquad \text{(decrease } f \text{ as } f(\vec{x}_{k+1}) \approx f(\vec{x}_k) - \epsilon \|\nabla f(\vec{x}_k)\|^2\text{)}$$

  - can get stuck in local minima, & learning rate too big may diverge
  - Convergence:
    - assume f convex, optimum is finite, f twice differentiable
    - $f(\vec{x}_T) - f(\vec{x}^*) \leq \frac{1}{2cT}(\vec{x}_1 - \vec{x}^*)^2$

$\Rightarrow$ error falls as $1/T$ ← iterations of gradient descent

- Momentum:
  - reduce zig-zag behaviour by updating $\vec{x}$ w/ moving average of gradients

$$\tilde{g}_{k+1} = \mu_k \tilde{g}_k - \epsilon g_k(\vec{x}_k)$$
$$\vec{x}_{k+1} = \vec{x}_k + \tilde{g}_{k+1}$$

  - converges faster, helps carry us across flat regions too (ie saddles)
  - very good for noisy gradients
- Nesterov:
  - Same as momentum, except

$$\tilde{g}_{k+1} = \mu_k \tilde{g}_k - \epsilon g(\vec{x}_k + \mu_k \tilde{g}_k)$$

  - ie use gradient of point we will move to, rather than current point (effectively, do momentum, then think 'how can I get an even better update using gradient at current point')
  - $\mu_k = 1 - 3/(k+5)$, convergence $f(\vec{x}_k) - f^* \leq \frac{c}{k^2}$ (comp. w/ $\frac{1}{k}$)

+ very simple & easy to implement
− need suitably small ϵ to guarantee convergence
− coordinate system dependant; transformation of coordinates (eg changing units) leads to different path

- Line Search & Conjugate Gradients:
  - Line search:
    - pick direction, find optimum in this direction, repeat
    - if quadratic function, $\vec{d} = -\nabla f(\vec{x}_k)$ zig-zags; better to line-search independently along each axis
  - Conjugate Gradients:
    - perform line search updates in conjugate directions sequentially, building up set of conjugate vectors
    - after d iterations, will have found overall optimum (to a quadratic function)
    - $k=1$; $\vec{x}_1 = \infty$; $\vec{p}_1 = -\vec{g}_1$
      while $\vec{g}_k \neq 0$:
      $\alpha_k = \underset{\alpha}{\operatorname{argmin}}\, f(\vec{x}_k + \alpha_k \vec{p}_k)$ ← line search
      $\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{p}_k$ ← update x
      $\beta_k = \vec{g}_{k+1}^T \vec{g}_{k+1} / (\vec{g}_k^T \vec{g}_k)$] find new conjugate direction
      $\vec{p}_{k+1} = -\vec{g}_{k+1} + \beta_k \vec{p}_k$ ← $\vec{p}_{k+1}$ conjugate to $\vec{p}_1, \dots, \vec{p}_k$
      k ++
  - Higher order methods:
    - Newton's method:

Taylor expansion to 2nd order →

$$f(x+\Delta) = f(x) + \Delta^T \nabla f + \frac{1}{2}\Delta^T H_f \Delta + O(|\Delta|)^3$$

$$\Rightarrow x_{k+1} = x_k - \varepsilon H_f^{-1} \nabla f$$ (converge in 1 step for quadric functions, $\varepsilon = 1$)

+ invariant under coord transform
− not storing Hessian & calculating $H^{-1}\nabla f$ expensive
− not guaranteed to produce downhill step; can do line search in $H^{-1}\nabla f$ direction instead

- Gauss-Newton:
  - for objective function $f(\vec{x}) = L(\vec{h}(\vec{x}))$,

$$G_{ij} = \sum_k \frac{\partial h_k}{\partial x_i} \frac{\partial^2 L}{\partial h_k \partial h_i} \frac{\partial h_i}{\partial x_j}$$ can approximate H, & is positive semidefinite

$\Rightarrow$ update $\vec{x}_{t+1} = \vec{x}_t - \varepsilon G^{-1} \vec{g}$, with $g_i = \frac{\partial f}{\partial x_i}$

approaches Newton as we approach optimum, as G approaches H

- Levenberg-Marquardt algorithm:
  - Gauss-Newton needs to solve $C\Delta = \nabla E$
  - C can be rank deficient or ill ~~defined~~ conditioned
  - regularize, & instead solve

$$(C + \lambda I)\Delta = \nabla E$$

Gauss-Newton for KL loss

- Natural Gradient:

$$E(\theta) = KL(p(x)|q(x|\theta)) = -\langle \log q(x|\theta) \rangle + \text{const}$$

gradient → $g_i = -\left\langle \frac{\partial}{\partial \theta_i} \log q(x|\theta) \right\rangle$, $C_{ij} = \left\langle \frac{\partial}{\partial \theta_i} \log q(x|\theta) \cdot \frac{\partial}{\partial \theta_j} \log q(x|\theta) \right\rangle$

curvature matrix

update → $\theta(t+1) = \theta(t) - \varepsilon C^{-1} g$

+ invariant under parameter transforms

5a: Large Scale Learning

- Linear Models:
  - least squares is $O(\dim(\vec{\theta})N^3)$ time to solve, & requires $O(D^2)$ space
    - ~~can't~~ don't need much space to store & compute gradient
    - $O(dDN)$ time, $O(N+D)$ space

$E(\vec{\theta}) = \sum (y_n - \vec{\theta}^T \vec{x}_n)^2 + \lambda \vec{\theta}^T \vec{\theta}$ (where d is density factor of sparse $\vec{x}$, ie each $\vec{x}$ has max dD non-zero elements)

$\frac{\partial E}{\partial \theta_i} = -2\sum (y_n - \vec{\theta}^T \vec{x}_n) x_{ni} + 2\lambda\theta_i$

- Online stochastic gradient descent:
  - scale objective $E(\vec{\theta}) = \frac{1}{N}\sum_n (y_n - \vec{\theta}^T \vec{x}_n)^2 + \gamma \vec{\theta}^T \vec{\theta}$

$$\frac{\partial E}{\partial \theta_i} = -2\frac{1}{N}\sum_n (y_n - \vec{\theta}^T \vec{x}_n) x_{ni} + 2\gamma\theta_i$$

batch size $N' < N$

$$\approx -2\frac{1}{N'}\sum_{n'} (y_{n'} - \theta^T x_{n'}) x_{n'i} + 2\gamma\theta_i$$

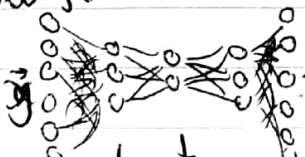
  - extreme case $N'=1$; treat inputs as a stream
  - + robust to synchronisation issues in parallel implementations
- Higher order methods:
  - use better ~~search~~ search directions + line search to converge faster
  - eg conjugate gradients

5b: Neural Net Introduction

- Vector input $\vec{x}$, $\vec{y} = g(\vec{x}|W) = \sigma_L(W_L \vec{h}_{L-1})$
  - where $\vec{h}_l = \sigma_l(W_l \vec{h}_{l-1})$, $l = 2, \dots, L-1$, $\vec{h}_1 = \sigma_1(W_1 \vec{x})$
  - σ_l is the transfer function (normally invertible & monotonic)
  - common to include bias terms too (so $h_l = \sigma(W_l h_{l-1} + b_l)$)

$$\vec{x} \to \vec{h}_1 \to \cdots \to \vec{h}_{L-1} \to \vec{y}$$

  - if σ is linear, whole network is linear & additional layers play no role
- Regression:
  - minimise loss function (eg squared loss)
  - optimise using SGD or higher order methods
    - gradient easy to calculate w/ AutoDiff
  - hard to initialise weights well; variety of approaches including layerwise training
  - common to include penalty (eg L_2-regularisation) on weights
    - otherwise very easy to overfit
- Specific architectures:
  - Autoencoder:



$$E(W) = \sum_n (\vec{y}_n - g(\vec{y}_n|W))^2$$

    - try to reconstruct input
    - bottleneck layer forces network to find low-dimensional representation of data
    - equivalent to PCA when using network $y \to h \to \tilde{y}$ w/ linear transfer functions
    - can only beat PCA with a non-linear function at output
    - if N = size dataset units in bottom hidden layer can always perfectly reconstruct output even w/ bottleneck size = 1

- Convolutional NNs:
  - allows for translational (& higher forms) of invariance; good for image processing
  - usually then max-pool after conv layer, to detect if any of neurons responded strongly
- NNs in NLP:
  - Word embeddings:
    - replace 1-hot euclidean embeddings w/ embeddings in lower dimensional space
    - can learn diff embeddings dependent on goal, eg next word prediction
    - natural geometry tends to appear, eg $V_{WOMAN} - V_{MAN} \approx V_{QUEEN} - V_{KING}$
- Time Series Models:
  - Recurrent nets:
    - hidden activation at time t (h_t) (& possibly y_t) depend on h_{t-1}, x_{t-1} & y_{t-1}
    - gradient computation:

$$E(A,B,C) = \sum_t (y_t - f(h_t; C))^2, \quad h_t = g(x_t, h_{t-1}; A, B$$

RTRL - Real time recurrent learning:
- compute gradient w/ single forward pass in time ($\frac{\partial h_t}{\partial \theta} = h(x_t, h_{t-1})$)
- $O(tH^3)$ space (H = size of hidden layer)
  $\Rightarrow$ normally impractical

BPTT - Backprop Through Time:
- use parameter tying; calculate gradient treating each layers params as independent, then evaluate expression by setting all to the same value
- efficient & exact

- hard to train, usually use special architectures (eg LSTM)

56: Neural Net Introduction

Generative Models:

- parameterise joint distribution (ie describes how points are generated)
  $$P(x, y) = P(x|y) \cdot P(y)$$
- can then find information about y from $P(y|x)$
- Variational inference:
  $$P(x|\theta) = \int_y P(x|y, \theta) \cdot P(y)$$
  - We want to find θ to maximise this
  - use a variational distribution $q(y|x, \phi)$ to help estimate $P(y)$, & optimise wrt θ & ϕ

Training nets:

- $E(\vec{w})$ usually complex, not convex, many non-trivial local optima
- Solution quality depends lots on weight initialisation
- + Very powerful when trained well & right

7a: Automatic Differentiation

- take $f(\vec{x})$ & return exact value for $\frac{\partial f}{\partial x_i}\Big|_{\vec{x}}$
  - can do in less than 5× time to calculate $f(\vec{x})$
- Symbolic differentiation:
  - can give computationally inefficient results, eg need to repeat calculation unnecessarily for $\frac{\partial f}{\partial x_1}\Big|_{\vec{x}}$ & $\frac{\partial f}{\partial x_2}\Big|_{\vec{x}}$
- Forward differentiation:
  - ~~Complex~~ dual arithmetic: ← can do similar w/ complex arithmetic, keeping the imaginary part

~~$f(x+\varepsilon$~~

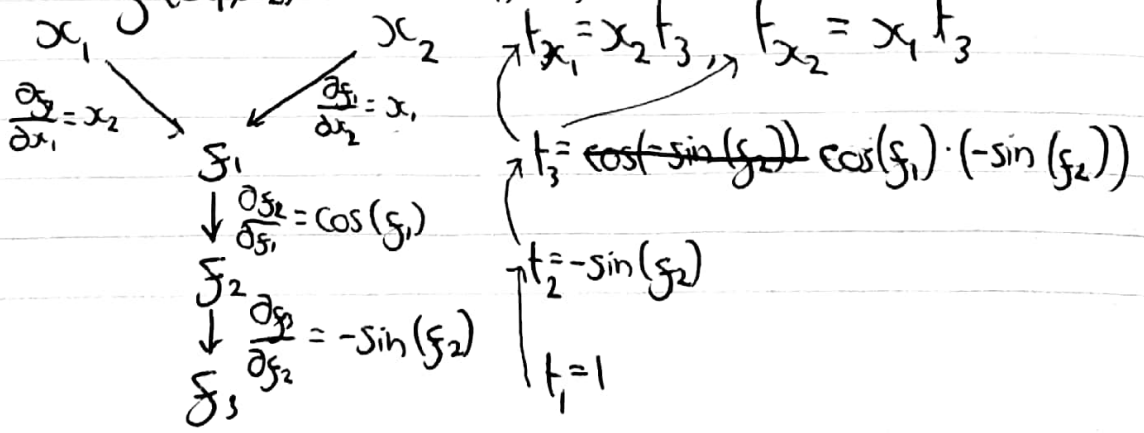
Dual exact / Comple approx: $f'(x) = \text{DualPart}(f(x+\varepsilon))$

(eg $f(x) = x^2 \Rightarrow f(x+\varepsilon) = x^2 + \boxed{2\varepsilon x}$)

- holds for any smooth function, need to overload every function to work with dual arithmetic
- not numerically efficient

- Reverse differentiation:
  1. Find reverse ancestral schedule of nodes
  2. Start from first node n_1 in reverse schedule, $t_{n_1} = 1$
  3. For next node n in schedule:
     $t_n = \sum_{c \in ch(n)} \frac{\partial f_c}{\partial f_n} t_c$, where $ch(n)$ is children of n
  4. Total derivatives wrt root nodes are given by values of t at these nodes

eg: $f(x_1, x_2) = \cos(\sin(x_1, x_2))$



- Hessian-Vector Product:
  - in Newton optimisation, update using $\vec{x} = H^{-1}\vec{g}$
  - in practice, find update by solving $H\vec{x} = \vec{g}$
    $\Rightarrow$ need $H\vec{x}$

derivative in direction of $\vec{v}$ $\searrow$ $D_{\vec{v}}(f) = \lim_{\delta \to 0} \frac{f(\vec{x} + \delta\vec{v}) - f(\vec{x})}{\delta} = \sum_i v_i \frac{\partial f}{\partial x_i}$

Can show $D_{\vec{v}}\left(\frac{\partial E}{\partial \theta_i}\right) = [H\vec{v}]_i$

- In AutoDiff:
  1. Forward pass, ordering nodes
  2. Backward pass, calculate t_n
  3. Directional derivative forward pass
     $D_{\vec{v}}(f_i) = \sum_k \frac{\partial f_i}{\partial \pi_k^i} D_{\vec{v}}(\pi_k^i)$ ($\vec{\pi}^i$ are parameters of f_i)
  4. Directional derivative reverse pass:
     $D_{\vec{v}}(t_n) = \sum_{c \in ch(n)} \left(\frac{\partial f_c}{\partial f_n} D_{\vec{v}}(t_c) + t_c \frac{\partial^2 f_c}{\partial f_n^2} D_{\vec{v}}(f_n) \right)$

7c: Neural Net Initialisation

- First, scale inputs to 0 mean, unit variance

$$x_i^n \leftarrow \frac{x_i^n - \mu_i}{\sigma_i}$$

- Aim to keep output 0 mean, unit variance ← to keep error $O(1)$

$\Rightarrow$ Sample weights from 0 mean, $1/D$ variance ← size of input $\vec{x}$

- can do for multiple layers

- Decorrelating inputs
  - approximate Hessian, & apply

$$\vec{x}^n \leftarrow S^{-1} U^T \vec{x}^n, \text{ where } XX^T = U S^2 U^T$$

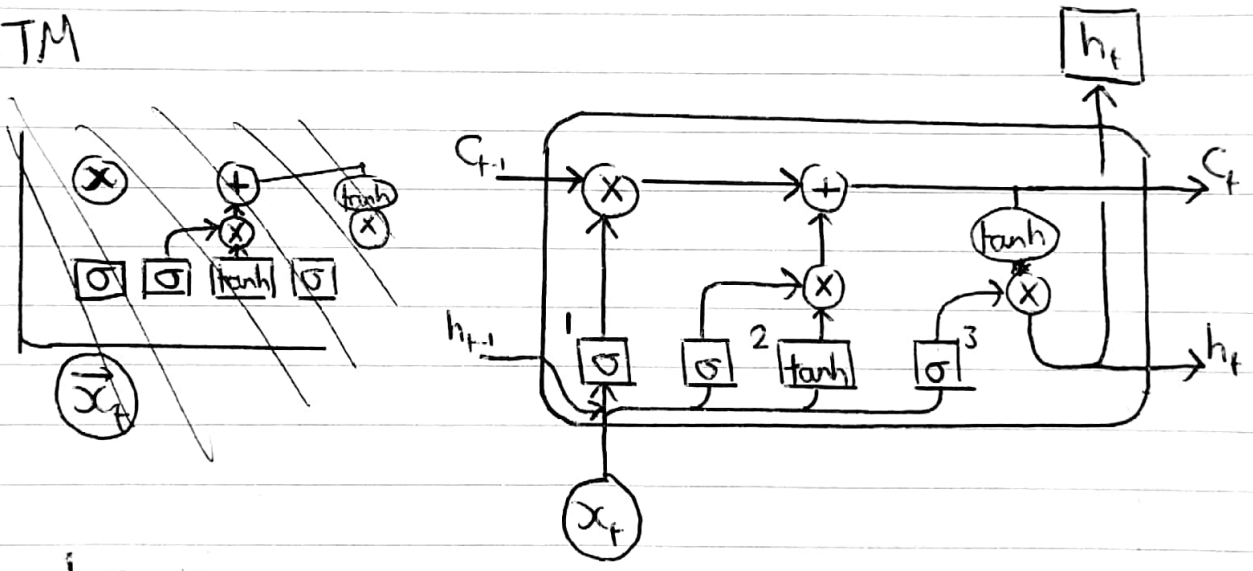
(ie SVD of X)

- make approximate Hessian diagonal & can be useful to ease optimisation

8a: Recurrent Networks

- Gradient decay / explosion:
  - for a very deep network, gradients grow exponentially w/ no. layers ⇒ can explode or decay to 0
  - need special architectures to deal w/ this
- Memory decay:
  - information typically lost over time steps in traditional RNN
  - need some form of 'running memory', $m(t)$
    - can be 'gated' using $\alpha(t) = \sigma(W_{in}\, x(t) + B_{in})$

      $m(t) = \alpha(t)\, x(t) + (1-\alpha(t))\, m(t-1)$
    - works well when α_t close to 1; gradient decays slowly
- LSTM



  - top is memory
  - 1: forget gate; allow or forgetting memory
  - 2: input gate; update memory
  - 3: output gate; decide what bit of memory to output
- GRU:
  - similar, but h_t acts as memory & is fully exposed as output

8b: Unsupervised Dimension Reduction

- In high dimensional data with some structure, it will tend to lie close to some lower dimensional manifold

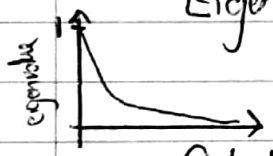
- Principle Component Analysis:
  - Full rank representation: $\vec{x} = \sum_{d=1}^{D} y_d \vec{b}^d$, $x \in \mathbb{R}^D$ (basis vectors (linearly independent))
  - Reduced rank approximation: $\vec{x} \approx \sum_{d=1}^{M} (y_d \vec{b}^d) + \vec{c}$, ie $\tilde{x} = \vec{c} + B\vec{y}$
  - aim to minimise $E(B, Y, \vec{c}) = \sum_n \sum_d (x_d^n - \tilde{x}_d^n)^2$

note: to give unique solution, enforce all $\vec{b}^i$ orthogonal & unit length

  - can show $\vec{c} = \frac{1}{N} \sum_n \vec{x}^n$, and $\vec{b}^d$ are eigenvectors of sample covariance matrix, $\frac{1}{N-1} \sum_n (\vec{x}^n - \vec{c})^T (\vec{x}^n - \vec{c})$

due to this, usually assume data centred on $\vec{0}$

  - can reduce dimensionality by only taking top M eigenvectors (by sorted eigenvalue)
  - Eigen-spectrum:



    - ~~Error~~ squared error of approximation $E = (N-1) \sum_{n=M+1}^{D} \lambda_d$ ie, sum neglected eigenvalues
  - ~~Rotational invariance~~
  - PCA Maths:

minimise $E(B, Y) = \sum_{n,d} (x_d^n - \sum_{j=1}^{M} y_j^n b_i^j)^2 = \text{Trace}((X - BY)^T (X - BY))$

$\frac{\partial E}{\partial y_k^n} = 0 \Rightarrow y_k^n = \sum_d b_d^n x_d^n$

$X = (\vec{x}^1, \ldots, \vec{x}^N)$

$B^T B = I$ (ie basis vectors = orthogonal & unit length)

$Y = B^T X$

$\Rightarrow E(B) = \text{trace}(X^T (I - BB^T)^2 X) = \text{Trace}(XX^T (1 - BB^T))$

Sample Covariance $= [\text{trace}(S) - \text{trace}(SBB^T)](N-1)$

To enforce $B^T B = I$, use lagrange multiplies L

$\Rightarrow$ minimise $-\text{trace}(SBB^T) + \text{trace}(L(B^T B - I))$

$\Rightarrow$ ~~ab optimum~~

$\Rightarrow$ at optimum $SB = BL$

- enforce L to be diagonal $\Rightarrow$ cols of B are corresponding eigenvectors of S

Squared residual error!

- Now, $\text{trace}(SBB^T) = \text{trace}(L)$

$$\Rightarrow E(B) \frac{1}{N-1} E(B) = Tr(S) - tr(L) = \sum_{d=1}^{D} \lambda_d - \sum_{d=1}^{M} \lambda_d = \sum_{d=M+1}^{D} \lambda_d$$

- PCA w/ missing data:
  - some x_i^n missing; aim to infer them
  - first, find B & Y by ~~only~~ minimising error wrt only known x_i^n
    - need to iterate B for fixed Y & vice versa, decreasing loss until minimum reached
  - can be used eg for Movie recommendations from a matrix of liked/disliked movies

- Matrix decompositions

$$\underset{(D\times N)}{X} \approx \underset{(D\times M)}{B} \underset{(M\times N)}{Y}$$

Data; basis; weights or components

(PCA is a type of this, where $M \leq D$)

- Under-complete: $M < D$, eg PCA
  - Y is lower dimensional approximate representation of X

- Over-complete: $M > D$
  - need to introduce additional constraints
  - can lead to sparse representations of data

8b: Unsupervised Dimension Reduction

- Non-Negative Matrix Factorisation: (~~NN~~NNMF)
  - $X_{ij} \approx \sum_k B_{ik} Y_{kj}$, where X_{ij}, ~~+ve~~ B_{ik} & Y_{kj} all +ve
    - eg X columns are images, B are 'basis' vectors for these images & Y weights for basis vectors
  - Training:
    - use an EM algorithm
    - aim to approximate $P(x,y) \approx \hat{P}(x,y) = \sum_z \underbrace{P(x|z)}_{X\,B} \underbrace{P(y|z)}_{Y} P(z)$
    - M: fix $P(x|z)$, $P(y|z)$, $P(z)$
      - find $P(z)$ to minimise $KL(P(x,y), \hat{P}(x,y))$
    - ~~E: fix $P(z)$~~
      - Similarly, update $P(x|z)$ & $P(y|z)$
    - E: update $q(z|x,y)$ used in M step
    - guaranteed to increase likelihood $\Rightarrow$ converges
  - + interpretable basis vectors

9a: Independent Component Analysis

- Find coordinate system where coords are independent
  - compared w/ PCA, where ensured to be orthogonal
  - can be better than PCA, for eg, finding directions along which data was generated (which not necessarily orthogonal)

- Train:
  - maximise likelihood

$$L(A^{-1}) = L(B) = N \log \det(B) + \sum_{n,i} \log P([B\vec{v}^n]_i)$$

where $\vec{v}^1 \ldots \vec{v}^N$ are points & A being our ICA estimate

- derive a gradient ascent rule (Ng can apply Newton's method to get natural gradient update)

9b: Supervised Dimension Reduction

- Aim to find linear projection of data to lower dimensional space that allows us best classification ~~accuracy~~ performance

- Fisher's Linear Discriminant:
  - model data as produced by Gaussian per each class
  - projected distributions still Gaussian
  - project such that there is minimal overlap

$$p(\vec{x}_{n|i}) = N(\vec{x}_i | \vec{m}_i, S_i)$$

$\vec{m}_1$ = sample mean of group 1

S_1 = ~~co~~ sample covariance

$$y_i^n = \vec{w}^T \vec{x}_i^n$$

Find $\vec{w}$ such that minimal overlap in ys

$$\min\left(\frac{(\mu_1 - \mu_2)^2}{\pi_1 \sigma_1^2 + \pi_2 \sigma_2^2}\right) = \frac{w^T (m_1 - m_2)(m_1 - m_2)^T w}{w^T (\pi_1 S_1 + \pi_2 S_2) w} = \frac{w^T A w}{w^T B w}$$

$$\Rightarrow \vec{w} \propto B^{-1}(\vec{m}_1 - \vec{m}_2)$$

- If $N_1 + N_2 < D$, B ~~*~~ not ~~d~~ invertible (~~fet~~ features which never change across all $\vec{x}$ can also make objective ill defined)

- Canonical Variates:
  - Generalise Fisher's method to >1 dimension & >2 classes

$$P(\vec{x}) = N(\vec{x} | \vec{m}_c, S_c),$$

$$\vec{y} = W^T \vec{x} \Rightarrow P(\vec{y}) = N(\vec{y} | W^T \vec{m}_c, W^T S_c W)$$

class count, class c mean, mean of all data

- Compute $\underline{A} = \sum_c N_c (\vec{m}_c - \vec{m})(\vec{m}_c - \vec{m})^T$

$$\underline{B} = \sum_c N_c \underline{S}_c$$

- Let $\tilde{\underline{B}}^T \tilde{\underline{B}} = \underline{B}$ (assuming $\underline{B}$ invertible)

- Maximize $F(\underline{W}) = \frac{tr(\underline{W}^T \tilde{\underline{B}}^{-T} \underline{A} \tilde{\underline{B}}^{-1} \underline{W})}{tr(\underline{W}^T \underline{W})} = tr(\underline{W}^T \underline{\hat{B}}^{-T} \underline{A} \tilde{\underline{B}}^{-1} \underline{W} \cdot \frac{1}{D})$ ($\underline{W}$ orthonormal)

$\underline{C}$

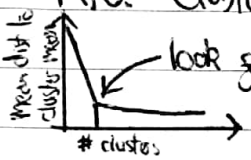
- $\underline{C} = \underline{E} \underline{\Lambda} \underline{E}^T$ (PSD & symmetric)

$$F(\underline{W}) = tr(\underline{W}^T \underline{E} \underline{\Lambda} \underline{E}^T \underline{W})$$

$\Rightarrow$ maximized by $\underline{W} = [\vec{e}_1, \ldots, \vec{e}_L]$, [eigenvectors of $\underline{C}$]

9c: Clustering

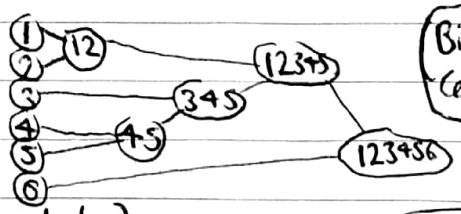
- K-means:
  - minimise sum of squared ~~lens~~ distances of points from cluster mean
  - Initialise means randomly
  - until converge:
    - assign points to closest cluster centre
    - recalculate cluster centres (mean of points in cluster)
  - decreases loss each iteration ⇒ will converge
  - non-deterministic; repeat with different initialisations
  - No. clusters:



  - Limitations:
    - only 1 cluster per datapoint
    - sensitive to outliers (k-medians better)
    - requires spherical clusters of equal density
    - scales poorly with dimensionality
    - sensitive to dimension scale (~~data~~* should be rescaled)
    - requires numerical data; encode categorical w/ one-hot vectors
    - how to deal with missing fields?

* each dimension

- Hierarchical K-means:
  - can do bottom up (merge from single points)
  - or top down (remove 'outsides' from least cohesive cluster)



Binary tree example, can be n-ary

- Spectral Clustering:
  - model data as Markov chain, with

$$p(x'|x) = \frac{A_{x,x'}}{\sum_{x''} A_{x,x''}}$$, where $A_{i,j}$ is similarity of i & j

result is very dependent on choice of this! Bad?

  - find equilibrium distribution, & can cluster this with K-means

- Gaussian Mixture Models:
  - model data generation; first sample cluster, then sample point from that cluster

$$P(v) = \sum_h P(v|h)P(h)$$

cluster h ('hidden')
point v ('visible' or 'observable')

  - Learn parameters by maximising likelihood (use EM algorithm)
  - Cluster by computing $P(h|v) \propto P(v|h) \cdot P(h)$ (find max $P(h|v)$)
  - $P(v|h)$ need not be Gaussian; ~~could eg be Bernoulli (for clustering binary~~ (eg some features from categorical distribution)

$$LL = \sum_n \log \sum_c P(c) \times \frac{1}{\sqrt{\det(2\pi \underline{\underline{S}}_c)}} \exp\left(\frac{-1}{2}(\underline{x}^n - \underline{m}_c)^T \underline{\underline{S}}_c^{-1} (\underline{x}^n - \underline{m}_c)\right)$$

    - $\underline{\underline{S}}_c$ must be symmetric, PSD
    - if $\underline{m}_c = \underline{x}^n$ for some c, n, as $\underline{\underline{S}}_c \to 0$, $P(\underline{x}^n | \underline{m}_c, \underline{\underline{S}}_c)$ becomes infinite $\Rightarrow$ need to limit how small $\underline{\underline{S}}_c$ ('width' of Gaussians) can be
  - k-means:
    - if constrain all $\underline{\underline{S}}_c$ to be $\sigma^2 \underline{\underline{I}}$, as $\sigma^2 \to 0$, GMM becomes equivalent to k-means
    - $\sigma^2 > 0$ give 'softer' clustering than k-means
  - Gaussian not too resistant to outliers; use t-distribution
  - Find number of pure clusters by cross-validation (likelihood of test data)
  - Easy to train using EM

10: Visualisation

- Aim to find low dimensional representation (2 or 3) of data
  - no correct or perfect method; all lose information
- Methods:
  - PCA & Sammon 'mappings'; old, less preferred now
  - Autoencoders; good, very expensive to train
- Stochastic Neighbour Encoding:
  - aim to transform $\underline{x}$ to y (high → low dimensional)

$$p_{j|i} = \frac{\exp(-(\underline{x}_i - \underline{x}_j)^2/(2\sigma_i^2))}{\sum_{j \neq i} \exp(-(\underline{x}_i - \underline{x}_j)^2/(2\sigma_i^2))} \qquad (p_{i|i} = 0)$$

$$q_{i|j} = \frac{\exp(-(y_i - y_j)^2)}{\sum_{j \neq i} \exp(-(y_i - y_j)^2)} \qquad (q_{i|i} = 0)$$

Do so by gradient descent; treat y_i's as parameters to optimize

  - aim to find Y such that q similar to p
    - minimize $\sum_i KL(p_i|q_i) = \sum_{ij} p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$
  - Problems:
    - KL divergence not symmetric; leads to good local structure but poor global
    - Gaussian $p_{i|j}$ = distant points have too little effect
- t-SNE:
  - use 'symmetric' loss

$$KL(p|q) = \sum_{ij} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

$$\left(\text{where } p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N},\ p_{i,i} = 0\right)$$

  - let q_{ij} be a student t-distribution

$$q_{ij} = \frac{(1+(y_i - y_j)^2)^{-1}}{\sum_{i \neq j} (1+(y_i - y_j)^2)^{-1}},\ q_{i,i} = 0$$

⇒ very distant points similar contribution to somewhat distant points

- Scaling:
  - $O(N^2)$ to compute t-SNE objective; so $O(N^2)$ per iteration of gradient descent!
  - Alternative:
    - select a few landmark points, i', randomly
    - compute $p_{j'|i'}$ for all pairs of landmarks by doing many Markov chains from i' until it reaches another landmark, j'
    - now do t-SNE on smaller $p_{j'|i'}$
      - expensive to compute $p_{j'|i'}$, but only needs to be done once

11: Fast Nearest Neighbours Calculation

- Finding nearest neighbour naively is $O(DN)$
- Orchard:
  - good when distance computation expensive (eg high D)
  - Triangle inequality:
    $|\underline{x}-\underline{y}| \leq |\underline{x}-\underline{z}| + |\underline{z}-\underline{y}|$ (ie faster to go to $\underline{y}$ directly rather than via $\underline{z}$)
  - Metric:
    - $d(\underline{x},\underline{y}) \geq 0$, $d(\underline{x},\underline{y}) = 0 \Rightarrow \underline{x} = \underline{y}$,
      $d(\underline{x},\underline{y}) = d(\underline{y},\underline{x})$ (& $d(\underline{x},\underline{y}) \leq d(\underline{x},\underline{z}) + d(\underline{z},\underline{y})$)
    - Euclidean distance is a metric
  - Exploiting triangle inequality:
    - if $d(\underline{x},\underline{y}) \leq \frac{1}{2} d(\underline{z},\underline{y}) \Rightarrow d(\underline{x},\underline{y}) \leq d(\underline{x},\underline{z})$
      $\Rightarrow$ don't need to calculate $d(\underline{x},\underline{z})$
  - Algorithm:
    - precompute all $d_{ij} = dist(\underline{x}_i, \underline{x}_j)$ — $O(DN^2)$ time! $O(N^2)$ space!
    - for each i, compute ordered list of $\underline{x}_j$ closest to $\underline{x}_i$
    - for query point q: — $O(D)$, repeat $M \leq N$ times
      - pick $\underline{x}_i$, initial guess at q's NN
      - go along $\underline{x}_i$'s list, ~~finding closest point to q~~
      - ~~can stop early~~
      - if $d(\underline{x}_j, q) < d(\underline{x}_i, q)$, go to $\underline{x}_j$'s list
      - if $d(\underline{x}_j, q) \geq d\frac{1}{2} d(\underline{x}_i, q)$ can stop early!
- Approximating & Eliminating Search Algorithm (AESA):
  - again, precompute distance matrix
  - $d(q, \underline{x}^j) \geq d(q, \underline{x}^i) - d(\underline{x}^i, \underline{x}^j)$ (triangle inequality)
  - $\Rightarrow$ ~~compute~~ let $I = \{$points for which $d(q, \underline{x}^i)$ calculated$\}$
  - $d(q, \underline{x}^j) \geq \max_{i \in I} \{d(q, \underline{x}^i) - d(\underline{x}^i, \underline{x}^j)\}$ — $O(M(N-M))$, repeat $M \leq N$ times
  - eliminate all j where this bound is greater than $\min d(q, \underline{x}^i)$

$O(N^2)$ space

↓

- Buoys:
  - Storing full distance matrix can be prohibitive
  - instead, store distances to set of buoys, $\underline{b}^1, \ldots, \underline{b}^B$
  - Pre-elimination:
    - ~~$d(\underline{q}, \underline{x}^n)$~~

$$\max_m \{d(\underline{q}, \underline{b}^m) - d(\underline{b}^m, \underline{x}^n)\} \leq d(\underline{q}, \underline{x}^n) \leq \min_m \{d(\underline{q}, \underline{b}^m) + d(\underline{b}^m, \underline{x}^n)\}$$

    - eliminate any points where lower bound is larger than smallest upperbound
    - then feed remainder to AESA or Orchard

AKA Linear AESA →
  - AESA with buoys:
    - order by lower bounds (lower bound for next point)
    - evaluate, if $d(\underline{q}, \underline{x}^n) \leq L_{next}$, terminate; $\underline{x}^n$ is NN
  - Orchard w/ buoys:
    - Worse than Linear-AESA
- k-d tree:
  - binary tree, recursively dividing space through medians, cycling through dimensions to divide along
  - Finding nearest neighbour:
    - traverse tree with $\underline{q}$ to root, a leaf, this is best guess
    - traverse up tree; if
      - check parent, if better guess then set this as best guess
      - check if need to traverse down other to other child of parent (hypersphere of radius = current best guess intersecting w/ hyperplane of parent = need to check other child)