

# Supervised Learning<sup>1</sup>

David Barber

University College London

---

<sup>1</sup> These slides accompany the book *Bayesian Reasoning and Machine Learning*. The book and demos can be downloaded from [www.cs.ucl.ac.uk/staff/D.Barber/bml](http://www.cs.ucl.ac.uk/staff/D.Barber/bml). Feedback and corrections are also available on the site. Feel free to adapt these slides for your own purposes, but please include a link to the above website.

## Utility and Loss

- Given an input  $x^*$ , the optimal prediction depends on how costly making an error is. This is quantified using a loss function  $L$  (or utility  $U = -L$ ).
- In forming a *decision function*  $c(x^*)$  that will produce a class label for the new input  $x^*$ , we don't know the true class, only our surrogate for this, the predictive distribution  $p(c|x^*)$ .
- If  $U(c^{true}, c^{pred})$  represents the utility of making a decision  $c^{pred}$  when the truth is  $c^{true}$ , the expected utility is

$$U(c(x^*)) = \sum_{c^{true}} U(c^{true}, c(x^*)) p(c^{true}|x^*)$$

For a parametric decision function  $c(x^*|\theta)$  we would have (summing over datapoints)

$$U(\theta) = \sum_n \sum_{c^{true}} U(c^{true}, c(x_n^*|\theta)) p(c^{true}|x_n^*)$$

One finds the decision (or parameter  $\theta$ ) that maximises the expected utility.

# The different philosophies

- In the above maximal expected utility framework, we need to define the utility,  $U$ , the class probability  $p(c|x)$  and the function  $c(x^*)$ .
- However, in practice we typically don't know the correct model underlying the data – all we have is a dataset of examples  $\mathcal{D} = \{(x^n, c^n), n = 1, \dots, N\}$  (and possibly our domain knowledge).
- There are different philosophies for how to determine  $p(c|x)$  and how to parameterise the decision function  $c(x^*)$ .
- Broadly, there are two main philosophies: Empirical Risk (empirical distribution for  $p(c|x)$  and parametric  $c(x^*|\theta)$ ) and Bayesian Decision (non-trivial  $p(c|x)$  and unconstrained  $c(x^*)$ ).
- We will discuss the benefits/drawbacks of each.
- Generally, there is no 'correct' approach and it's useful to potentially consider both, depending on the problem.

## Zero-one loss/utility

A ‘count the correct predictions’ measure of prediction performance is based on the zero-one utility:

$$U(c^{true}, c^*) = \begin{cases} 1 & \text{if } c^* = c^{true} \\ 0 & \text{if } c^* \neq c^{true} \end{cases}$$

For the two class case, we then have the expected utility

$$U(c(x^*)) = \begin{cases} p(c^{true} = 1|x^*) & \text{for } c(x^*) = 1 \\ p(c^{true} = 2|x^*) & \text{for } c(x^*) = 2 \end{cases}$$

Hence, in order to have the highest expected utility, the (unconstrained) decision function  $c(x^*)$  should correspond to selecting the highest class probability  $p(c|x^*)$ :

$$c(x^*) = \begin{cases} 1 & \text{if } p(c = 1|x^*) > 0.5 \\ 2 & \text{if } p(c = 2|x^*) > 0.5 \end{cases}$$

In the case of a tie, either class is selected at random with equal probability.

## General utility functions

- One can readily generalise this to multiple-class situations using a *utility matrix* with elements

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

where the  $i, j$  element of the matrix contains the utility of predicting class  $j$  when the true class is  $i$ .

- Then the expected utility is

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

and one then finds the class  $c^*$  that maximises this expected utility.

# Asymmetric Utility

- In some applications the utility matrix is highly non-symmetric.
- Consider a medical scenario in which we are asked to predict whether or not the patient has cancer  $\text{dom}(c) = \{\text{cancer}, \text{benign}\}$ .
- If the true class is cancer yet we predict benign, this could have terrible consequences for the patient. On the other hand, if the class is benign yet we predict cancer, this may be less disastrous for the patient.
- Such asymmetric utilities can favour conservative decisions – in the cancer case, we would be more inclined to decide the sample is cancerous than benign, even if the predictive probability of the two classes is equal.

## Squared loss/utility

- In regression problems, for a real-valued prediction  $y^{pred}$  and truth  $y^{true}$ , a common loss function is the squared loss

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

- The above decision framework then follows through, replacing summation with integration for the continuous variables.
- For an output prediction given an input  $x$ ,

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

In the unconstrained case, this loss is minimized by setting

$$y^{pred} = \int y^{true} p(y^{true}|x) dy^{true}$$

## Other loss functions

- It's worth noting that some regression problems have non-squared loss (for example the summed absolute loss) in which case optimal predictor won't necessarily be simply the average of the prediction distribution.
- It's very common to train models using the squared loss. However, if the task performance is measured by some other loss, it often makes more sense to train the model using the correct loss. One often sees users train a model using one kind of loss, but evaluate it using a very different loss – seems a bad idea in general.
- It's also worth noting that the loss can heavily effect how easy it is to train a model. For example, in classification, using the logistic regression model  $p(c = 1|x) = \sigma(\theta^T \mathbf{x})$  with a log-loss

$$U(\theta) = \frac{1}{N} \sum_n \sum_{c^{true}} p(c^{true}|n) \log \sigma((2c^{true} - 1)\theta^T \mathbf{x}^n)$$

gives a convex optimisation problem for  $\theta$ . Using a squared loss gives a non-convex problem.

## Empirical Risk Approach

## Using the empirical distribution

- Given a dataset of train data  $\mathcal{D} = \{(x^n, c^n), n = 1, \dots, N\}$ , a direct approach to not knowing the correct model  $p^{true}(c, x)$  is to replace it with the *empirical distribution*

$$p(c, x | \mathcal{D}) = \frac{1}{N} \sum_{n=1}^N \delta(c, c^n) \delta(x, x^n)$$

- Using this gives the empirical expected utility

$$\langle U(c, c(x)) \rangle_{p(c, x | \mathcal{D})} = \frac{1}{N} \sum_n U(c^n, c(x^n))$$

or conversely the *empirical risk*

$$R = \frac{1}{N} \sum_n L(c^n, c(x^n))$$

- Assuming the loss is minimal when the correct class is predicted, the optimal decision  $c(x)$  for any input in the train set is given by  $c(x^n) = c^n$ .
- However, for any new  $x^*$  not contained in  $\mathcal{D}$  then  $c(x^*)$  is undefined.

## Prediction beyond the train set

- To define the class of a novel input we use a parametric function  $c(x|\theta)$ . For example for a two class problem  $\text{dom}(c) = \{1, 2\}$ , a linear decision function is given by

$$c(\mathbf{x}|\theta) = \begin{cases} 1 & \text{if } \boldsymbol{\theta}^T \mathbf{x} + \theta_0 \geq 0 \\ 2 & \text{if } \boldsymbol{\theta}^T \mathbf{x} + \theta_0 < 0 \end{cases}$$

If the vector input  $\mathbf{x}$  is on the positive side of a hyperplane defined by the vector  $\boldsymbol{\theta}$  and bias  $\theta_0$ , we assign it to class 1, otherwise to class 2. The empirical risk then becomes a function of the parameters  $\theta = \{\boldsymbol{\theta}, \theta_0\}$ ,

$$R(\theta|\mathcal{D}) = \frac{1}{N} \sum_n L(c^n, c(x^n|\theta))$$

The optimal parameters  $\theta$  are given by minimising the empirical risk with respect to  $\theta$ ,

$$\theta_{opt} = \underset{\theta}{\operatorname{argmin}} R(\theta|\mathcal{D})$$

The decision for a new datapoint  $x^*$  is then given by  $c(x^*|\theta_{opt})$ .

## Example of the Empirical Risk approach

Let's consider a two class problem in which we have utility

$$U(c^{true} = 1, c^{pred} = 1) = 1, U(c^{true} = 2, c^{pred} = 2) = 1$$

$$U(c^{true} = 1, c^{pred} = 2) = 0, U(c^{true} = 2, c^{pred} = 1) = -20$$

and consider a classifier:

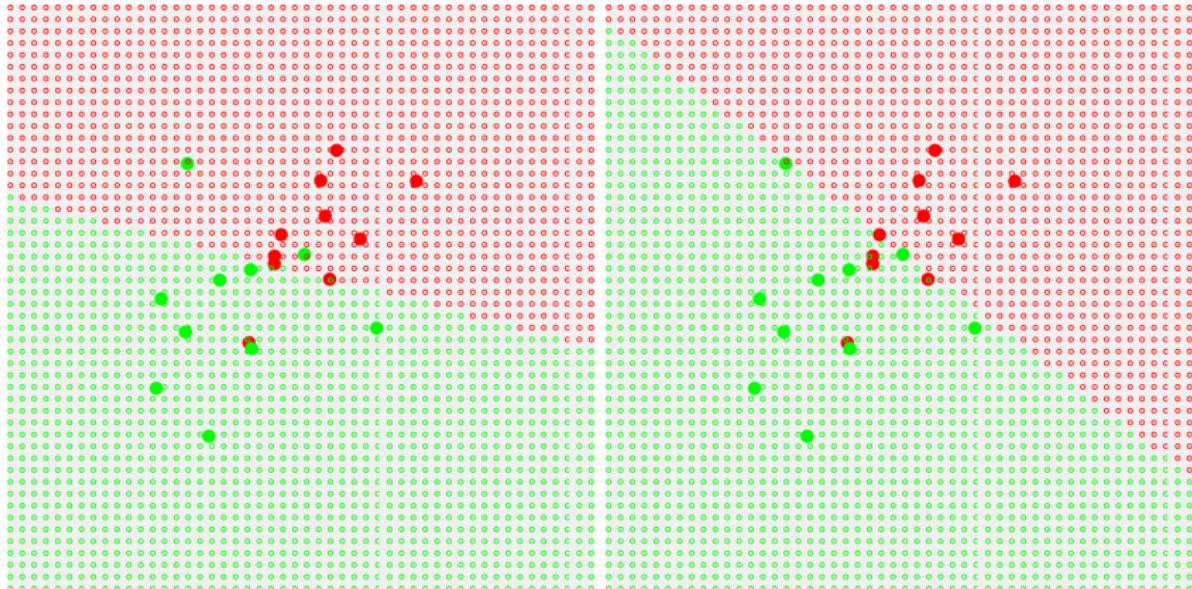
$$c(\mathbf{x}|\theta) = \begin{cases} 1 & \text{if } \boldsymbol{\theta}^\top \mathbf{x} + \theta_0 \geq 0 \\ 2 & \text{if } \boldsymbol{\theta}^\top \mathbf{x} + \theta_0 < 0 \end{cases}$$

Then the unpenalised utility  $U(\theta)$  is

$$\sum_{n|c^n=1} \mathbb{I}[\boldsymbol{\theta}^\top \mathbf{x}^n + \theta_0 \geq 0] + \sum_{n|c^n=2} \mathbb{I}[\boldsymbol{\theta}^\top \mathbf{x}^n + \theta_0 < 0] - 20 \sum_{n|c^n=2} \mathbb{I}[\boldsymbol{\theta}^\top \mathbf{x}^n + \theta_0 \geq 0]$$

Finding  $\theta$  that maximises this utility is difficult since this is a non-convex objective. Once learned we make a prediction  $c(x^*|\theta)$  for a novel  $x^*$ . The decision boundary is linear in this particular case.

# Empirical Risk Examples



(a) Using an identity utility matrix  $U = I$

(b) Using  $U(1,1) = U(2,2) = 1, U(2,1) = -20, U(1,2) = 0$

For each utility  $U$  we need to retrain the model. Note how changing the utility changes the decision boundary. The larger circle are the training points, and the smaller circles the test points.

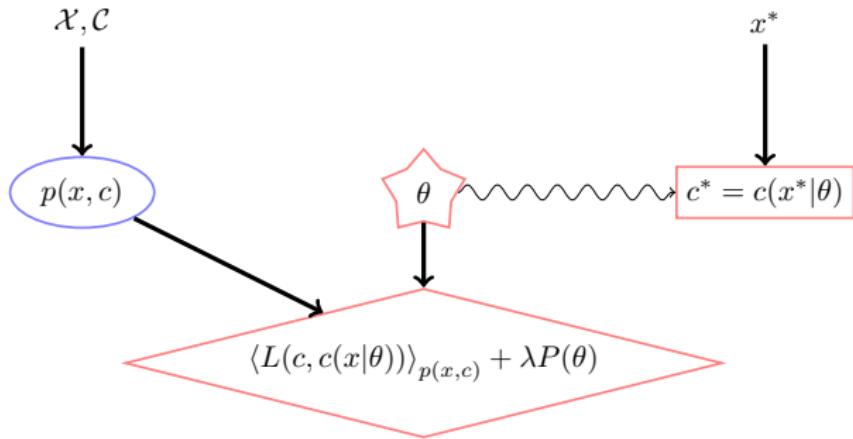
# Overfitting

- In this *empirical risk minimisation* approach, as we make the decision function  $c(x|\theta)$  more flexible, the empirical risk goes down.
- However, if we make  $c(x|\theta)$  too flexible we will have little confidence that  $c(x|\theta)$  will perform well on a novel input  $x^*$ . Since we only constrain the decision function on the known training points, a flexible  $c(x|\theta)$  may change rapidly as we move away from the train data, leading to poor generalisation.
- To constrain the complexity of  $c(x|\theta)$  we may minimise the penalised empirical risk

$$R'(\theta|\mathcal{D}) = R(\theta|\mathcal{D}) + \lambda P(\theta)$$

where  $P(\theta)$  is a function that penalises complex functions  $c(x|\theta)$ . The *regularisation constant*,  $\lambda$ , determines the strength of this penalty and is typically set by validation.

# Empirical Risk



Empirical risk approach. Given the dataset  $\mathcal{X}, \mathcal{C}$ , a model of the data  $p(x, c)$  is made, usually using the empirical distribution. For a classifier  $c(x|\theta)$ , the parameter  $\theta$  is learned by minimising the penalised empirical risk with respect to  $\theta$ . The penalty parameter  $\lambda$  is set by validation. A novel input  $x^*$  is then assigned to class  $c(x^*|\theta)$ , given this optimal  $\theta$ .

## Heuristic justification for squared Penalty

- For the linear decision function above, it is reasonable to penalise wildly changing classifications in the sense that if we change the input  $x$  by only a small amount we expect (on average) minimal change in the class label.
- The squared difference in  $\theta^T x + \theta_0$  for two inputs  $x_1$  and  $x_2$  is  $(\theta^T \Delta x)^2$  where  $\Delta x \equiv x_2 - x_1$ .
- By constraining the length of  $\theta$  to be small we limit the ability of the classifier to change class for only a small change in the input space.
- Assuming the distance between two datapoints is distributed according to an isotropic multivariate Gaussian with zero mean and covariance  $\sigma^2 I$ , the average squared change is  $\langle (\theta^T \Delta x)^2 \rangle = \sigma^2 \theta^T \theta$ , motivating the choice of the Euclidean squared length of the parameter  $\theta$  as the penalty term,  $P(\theta) = \theta^T \theta$ .

# Validation

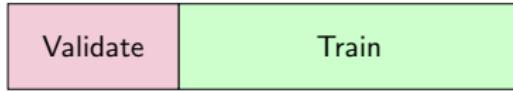
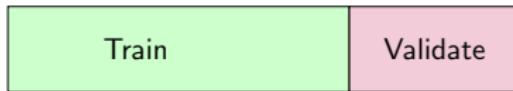


Models can be trained using the train data based on different regularisation parameters. The optimal regularisation parameter is determined by the empirical performance on the validation data. An independent measure of the generalisation performance is obtained by using a separate test set.

# Validation

- In penalised empirical risk minimisation we need to set the regularisation constant  $\lambda$ . This can be achieved by evaluating the performance of the learned classifier  $c(x|\theta)$  on validation data  $\mathcal{D}_{validate}$  for several different  $\lambda$  values, and choosing the  $\lambda$  which gave rise to the classifier with the best performance.
- It's important that the validation data is not the data on which the model was trained since we know that the optimal setting for  $\lambda$  in that case is zero, and again we will have little confidence in the generalisation ability.
- Given a dataset  $\mathcal{D}$  we split this into disjoint parts,  $\mathcal{D}_{train}, \mathcal{D}_{validate}$ , where the size of the validation set is usually chosen to be smaller than the train set. For each parameter  $\lambda_a$  one then finds the minimal empirical risk parameter  $\theta_a$ . The optimal  $\lambda$  is chosen as that which gives rise to the model with the minimal validation risk. Using the optimal regularisation parameter  $\lambda$ , many practitioners retrain  $\theta$  on the basis of the whole dataset  $\mathcal{D}$ .

# Cross Validation

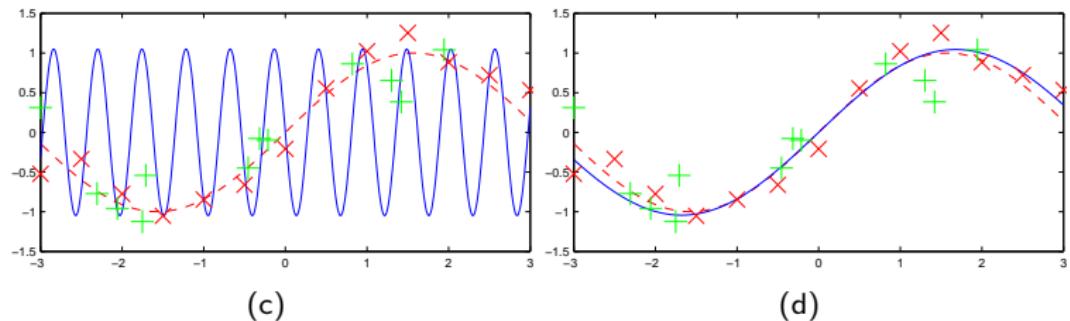


In cross-validation the dataset is split into several train-validation sets. Depicted is 3-fold cross-validation. For a range of regularisation parameters, the optimal regularisation parameter is found based on the empirical validation performance averaged across the different splits.

## Cross validation

- In cross-validation the dataset is partitioned into training and validation sets multiple times with validation results obtained for each partition.
- Each partition produces a different training  $\mathcal{D}_{train}^i$  and validation  $\mathcal{D}_{validate}^i$  set, along with an optimal penalised empirical risk parameter  $\theta_a^i$  and associated (unregularised) validation performance  $R(\theta_a^i | \mathcal{D}_{validate}^i)$ .
- The performance of regularisation parameter  $\lambda_a$  is taken as the average of the validation performances over  $i$ . The best regularisation parameter is then given as that with the minimal average validation error.
- In  $K$ -fold cross-validation the data  $\mathcal{D}$  is split into  $K$  equal sized disjoint parts  $\mathcal{D}_1, \dots, \mathcal{D}_K$ . Then  $\mathcal{D}_{validate}^i = \mathcal{D}_i$  and  $\mathcal{D}_{train}^i = \mathcal{D} \setminus \mathcal{D}_{validate}^i$ . This gives a total of  $K$  different training-validation sets over which performance is averaged. In practice 10-fold cross-validation is popular, as is leave-one-out cross-validation in which the validation sets consist of only a single example.

## Validation example



The true function which generated the noisy data is the dashed line; the function learned from the data is given by the solid line. **(a)**: The unregularised fit ( $\lambda = 0$ ) to training given by  $\times$ . Whilst the training data is well fitted, the error on the validation examples, denoted by  $+$ , is high. **(b)**: The regularised fit ( $\lambda = 0.5$ ). Whilst the train error is high, the validation error is low.

# Empirical risk approach

## Benefits

- In the limit of a large amount of training data the empirical distribution tends to the correct distribution.
- The discriminant function is chosen on the basis of minimal risk, which is the quantity we are ultimately interested in.
- The procedure is conceptually straightforward.

---

## Drawbacks

- It seems extreme to assume that the data follows the empirical distribution, particularly for small amounts of train data. More reasonable assumptions for  $p(x)$  would take into account likely  $x$  that could arise, not just those in the train data.
- If the loss function changes, the discriminant function needs to be retrained.
- Some problems require an estimate of the confidence of the prediction. Whilst there may be heuristic ways to evaluating confidence in the prediction, this is not inherent in the framework.

## Bayesian Decision Approach

## Bayesian decision approach

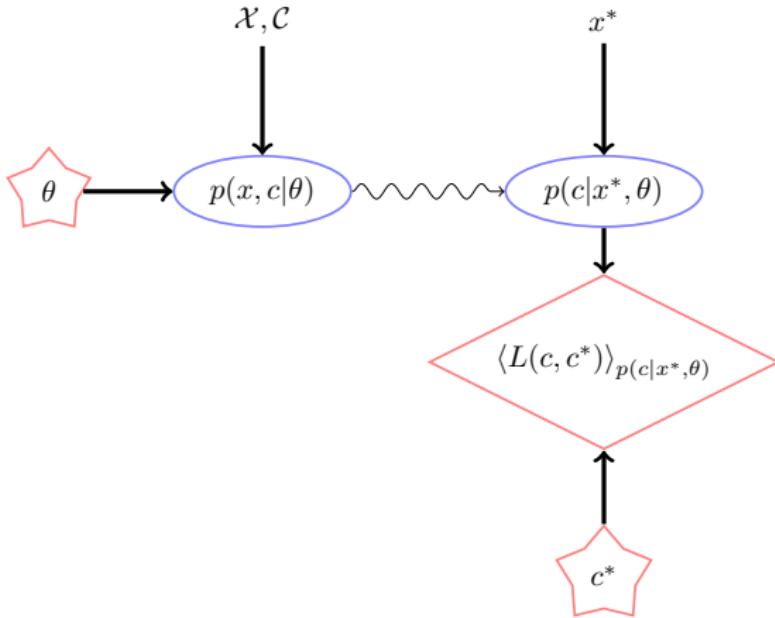
- An alternative to using the empirical distribution is to first fit a model  $p(c, x|\theta)$  to the train data  $\mathcal{D}$ .
- How the model  $p(c, x|\theta)$  is trained is a separate issue. This can be done for example using maximum likelihood. For example

$$\theta = \arg \max_{\theta} \sum_{n=1}^N \log p(c^n, x^n | \theta)$$

- Note that issues of overfitting also arise here and can be addressed using regularisation and validation. For example, adjust regularisation parameters to ensure that a validation likelihood is high.
- Given this model, the decision function  $c(x)$  is automatically determined from the maximal expected utility (or minimal risk) with respect to this model, in which the unknown  $p(c^{true}|x)$  is replaced with  $p(c|x, \theta)$ :

$$U(c(x^*)) = \sum_c U(c, c(x^*))p(c|x^*, \theta)$$

and  $c(x^*) = \arg \max_{c(x^*)} U(c(x^*)).$



Bayesian decision approach. A model  $p(x, c|\theta)$  is fitted to the data. After learning the optimal model parameters  $\theta$ , we compute  $p(c|x, \theta)$ . For a novel  $x^*$ , the distribution of the assumed ‘truth’ is  $p(c|x^*, \theta)$ . The prediction (decision) is then given by that  $c^*$  which minimises the expected risk  $\langle L(c, c^*) \rangle_{p(c|x^*, \theta)}$ .

# Discriminative versus Generative modelling

There are two main approaches to fitting  $p(c, x|\theta)$  to data  $\mathcal{D}$ . These boil down to the two ways to parameterise a joint distribution.

## Discriminative approach

In this case we write:

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

so that we explicitly parameterise how to form the class distribution.

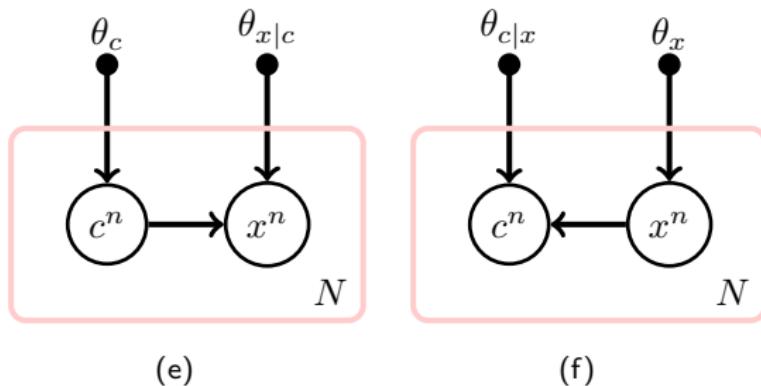
---

## Generative approach

In this case we make a model of  $x$  given the class:

$$p(c, x|\theta) = p(x|c, \theta_{x|c})p(c|\theta_c)$$

# Generative versus Discriminative modelling



Two generic strategies for probabilistic classification. **(a)**: Class dependent generative model of  $x$ . After learning parameters, classification is obtained by making  $x$  evidential and inferring  $p(c|x)$ . **(b)**: A discriminative classification method  $p(c|x)$ .

## Example of the Generative Modelling Approach

We first fit a model to data, for example:

$$p(x|c) = \mathcal{N}(x|\mu_c, \Sigma_c)$$

using say maximum likelihood. This is easy – just use the empirical mean and covariance. We set  $p(c = 1)$  to the fraction of training datapoints in class 1, and similarly for  $p(c = 2)$ . For any subsequent decision for class of input  $x^*$  we calculate:

$$p(c = 1|x^*) = \frac{p(x^*|c = 1)p(c = 1)}{p(x^*|c = 1)p(c = 1) + p(x^*|c = 2)p(c = 2)}$$

$$p(c = 2|x^*) = \frac{p(x^*|c = 2)p(c = 2)}{p(x^*|c = 1)p(c = 1) + p(x^*|c = 2)p(c = 2)}$$

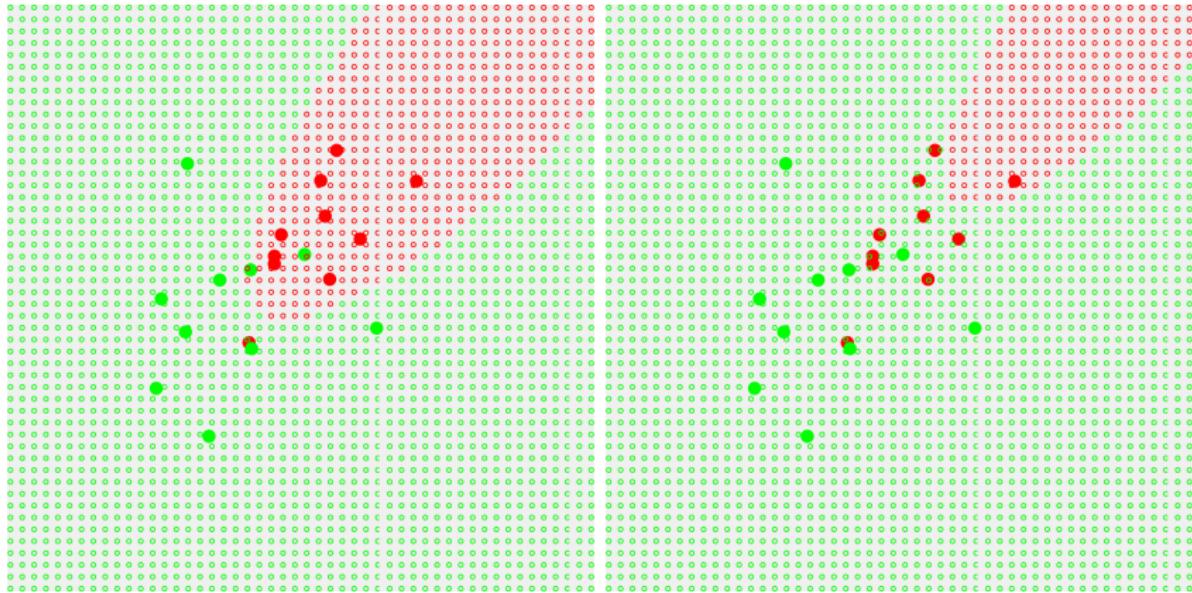
and

$$U(c^* = 1) = U(1, 1)p(c = 1|x^*) + U(2, 1)p(c = 2|x^*)$$

$$U(c^* = 2) = U(1, 2)p(c = 1|x^*) + U(2, 2)p(c = 2|x^*)$$

Then we set  $c^* = 1$  if  $U(c^* = 1) > U(c^* = 2)$ . In this case the resulting decision boundary is non-linear.

# Generative Modelling Examples



(g) Using an identity utility matrix  $U = I$

(h) Using  $U(1,1) = U(2,2) = 1, U(2,1) = -20, U(1,2) = 0$

The model is fitted independently of  $U$ . Note how changing the utility changes the decision boundary. The larger circle are the training points, and the smaller circles the test points.

# Generative approach

**Advantages** Prior information about the structure of the data is often most naturally specified through a generative model  $p(x|c)$ . For example, for male faces, we would expect to see heavier eyebrows, a squarer jaw, etc.

**Disadvantages** The generative approach does not directly target the classification model  $p(c|x)$  since the goal of generative training is rather to model  $p(x|c)$ . If the data  $x$  is complex, finding a suitable generative data model  $p(x|c)$  is a difficult task. Furthermore, since each generative model is separately trained for each class, there is no competition amongst the models to explain the  $x$  data. On the other hand it might be that making a model of  $p(c|x)$  is simpler, particularly if the decision boundary between the classes has a simple form, even if the data distribution of each class is complex.

## Example of the Discriminative Modelling Approach

Let's first fit a model to data:

$$p(c = 1|x) = \sigma(\theta^T \mathbf{x} + \theta_0)$$

using say maximum likelihood. This is easy since the log-likelihood is convex.  
For any subsequent decision for class of input  $x^*$  we calculate:

$$U(c^* = 1) = U(1, 1)\sigma(\theta^T \mathbf{x} + \theta_0) + U(2, 1)(1 - \sigma(\theta^T \mathbf{x} + \theta_0))$$

$$U(c^* = 2) = U(1, 2)\sigma(\theta^T \mathbf{x} + \theta_0) + U(2, 2)(1 - \sigma(\theta^T \mathbf{x} + \theta_0))$$

Then we set  $c^* = 1$  if  $U(c^* = 1) > U(c^* = 2)$ .

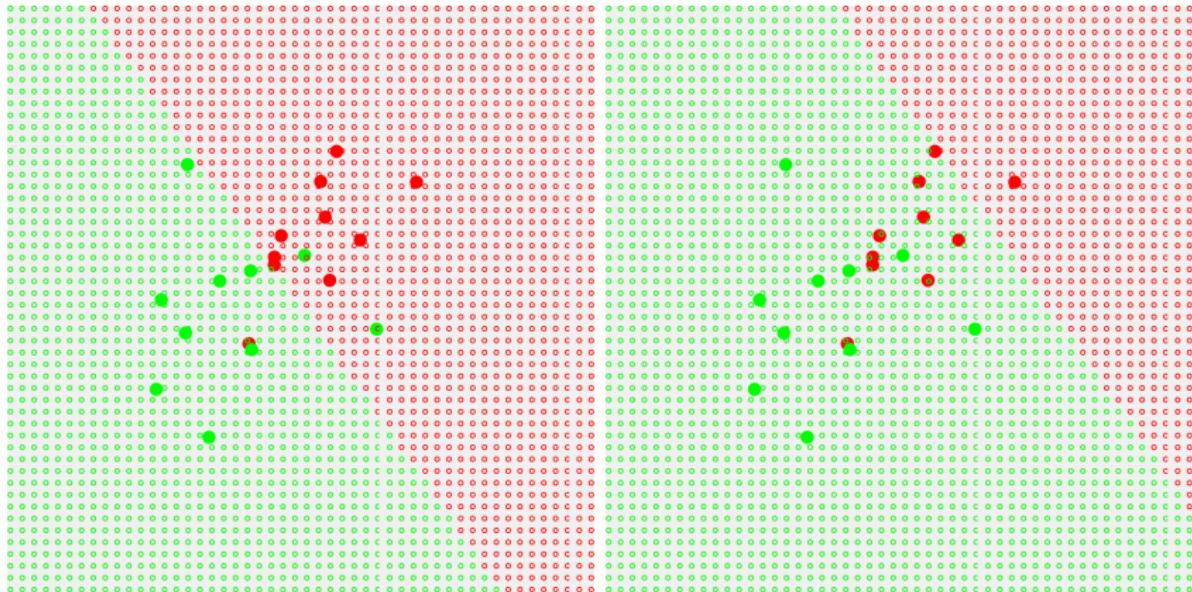
For our example the specific utilities give:

$$U(c^* = 1) = \sigma(\theta^T \mathbf{x} + \theta_0) - (1 - \sigma(\theta^T \mathbf{x} + \theta_0)) = 2\sigma(\theta^T \mathbf{x} + \theta_0) - 1$$

$$U(c^* = 2) = 1 - \sigma(\theta^T \mathbf{x} + \theta_0)$$

For this model the resulting decision boundary is linear.

# Discriminative Modelling Examples



(i) Using an identity utility matrix  $U = I$

(j) Using  $U(1, 1) = U(2, 2) = 1, U(2, 1) = -20, U(1, 2) = 0$

The model is fitted independently of  $U$ . Note how changing the utility changes the decision boundary. The larger circle are the training points, and the smaller circles the test points.

# Discriminative approach

**Advantages** The discriminative approach directly addresses finding an accurate classifier  $p(c|x)$  based on modelling the decision boundary, as opposed to the class conditional data distribution in the generative approach. Whilst the data from each class may be distributed in a complex way, it could be that the decision boundary between them is relatively easy to model.

**Disadvantages** Discriminative approaches are usually trained as 'black-box' classifiers, with little prior knowledge built used to describe how data for a given class is distributed. Domain knowledge is often more easily expressed using the generative framework.

# Bayesian decision approach

## Benefits

- This is a conceptually ‘clean’ approach, in which one tries ones best to model the environment (using either a generative or discriminative approach), independent of the subsequent decision process.
- In this case learning the environment is separated from the effect this will have on the expected utility.
- The decision  $c^*$  for a novel input  $x^*$  can be a highly complex function of  $x^*$  due to the maximisation operation.
- If  $p(x, c|\theta)$  is the ‘true’ model of the data, this approach is optimal.

---

## Drawbacks

- If the environment model  $p(c, x|\theta)$  is poor, the prediction  $c^*$  could be highly inaccurate since modelling the environment is divorced from prediction.
- To avoid divorcing the learning of the model  $p(c, x|\theta)$  from its effect on decisions, in practice one often includes regularisation terms in the environment model  $p(c, x|\theta)$  which are set by validation based on an empirical loss.

# Mathematics Refresher

David Barber

# Linear Algebra

# Matrices

An  $m \times n$  matrix  $\mathbf{A}$  is a collection of scalar values arranged in a rectangle of  $m$  rows and  $n$  columns.

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

The  $i, j$  element of matrix  $\mathbf{A}$  can be written  $A_{ij}$  or more conventionally  $a_{ij}$ . Where more clarity is required, one may write  $[\mathbf{A}]_{ij}$  (for example  $[\mathbf{A}^{-1}]_{ij}$ ).

---

## Matrix addition

For two matrix  $\mathbf{A}$  and  $\mathbf{B}$  of the same size,

$$[\mathbf{A} + \mathbf{B}]_{ij} = [\mathbf{A}]_{ij} + [\mathbf{B}]_{ij}$$

## Matrix multiplication

For an  $l$  by  $n$  matrix  $\mathbf{A}$  and an  $n$  by  $m$  matrix  $B$ , the product  $\mathbf{AB}$  is the  $l$  by  $m$  matrix with elements

$$[\mathbf{AB}]_{ik} = \sum_{j=1}^n [\mathbf{A}]_{ij} [\mathbf{B}]_{jk} ; \quad i = 1, \dots, l \quad k = 1, \dots, m .$$

In general  $\mathbf{BA} \neq \mathbf{AB}$ . When  $\mathbf{BA} = \mathbf{AB}$  we say they  $\mathbf{A}$  and  $\mathbf{B}$  commute.

$$\begin{aligned} & \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{pmatrix} \\ &= \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} \end{pmatrix} \end{aligned}$$

## Identity

The matrix  $\mathbf{I}$  is the identity matrix, necessarily square, with 1's on the diagonal and 0's everywhere else. For clarity we may also write  $\mathbf{I}_m$  for a square  $m \times m$  identity matrix. Then for an  $m \times n$  matrix  $\mathbf{A}$ ,  $\mathbf{I}_m \mathbf{A} = \mathbf{A} \mathbf{I}_n = \mathbf{A}$ . The identity matrix has elements  $[\mathbf{I}]_{ij} = \delta_{ij}$  given by the Kronecker delta:

$$\delta_{ij} \equiv \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

## Transpose

The transpose  $\mathbf{B}^T$  of the  $n$  by  $m$  matrix  $\mathbf{B}$  is the  $m$  by  $n$  matrix  $D$  with components

$$[\mathbf{B}^T]_{kj} = \mathbf{B}_{jk}; \quad k = 1, \dots, m \quad j = 1, \dots, n.$$

$(\mathbf{B}^T)^T = \mathbf{B}$  and  $(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$ . If the shapes of the matrices  $\mathbf{A}, \mathbf{B}$  and  $\mathbf{C}$  are such that it makes sense to calculate the product  $\mathbf{ABC}$ , then

$$(\mathbf{ABC})^T = \mathbf{C}^T \mathbf{B}^T \mathbf{A}^T$$

# Vector algebra

## Vectors

Let  $\mathbf{x}$  denote the  $n$ -dimensional column vector with components

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

A vector can be considered a  $n \times 1$  matrix.

---

## Addition

$$\mathbf{x} + \mathbf{y} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{pmatrix}$$

# Vectors

## Euclidean representation

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = x_1 \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + x_2 \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + x_3 \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

We can write this as

$$\mathbf{x} = x_1 \mathbf{e}^1 + x_2 \mathbf{e}^2 + x_3 \mathbf{e}^3$$

---

## Using a different basis

We can choose other basis vectors and then write the same vector

$$\mathbf{x} = y_1 \mathbf{b}^1 + y_2 \mathbf{b}^2 + y_3 \mathbf{b}^3$$

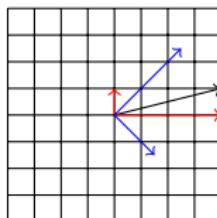
If these basis vectors are orthonormal

$$y_i = \mathbf{x}^\top \mathbf{b}^i$$

# Vectors: 2D example

Euclidean representation

$$\mathbf{x} = \begin{pmatrix} 4 \\ 1 \end{pmatrix}$$



---

Using a different basis

We can choose other basis vectors and then write the same vector

$$\mathbf{x} = y_1 \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} + y_2 \begin{pmatrix} -1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}$$

Since these basis vectors are orthonormal

$$y_1 = \left( \begin{pmatrix} 4 \\ 1 \end{pmatrix}^T \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} \right) = 5/\sqrt{2}, \quad y_2 = \left( \begin{pmatrix} 4 \\ 1 \end{pmatrix}^T \begin{pmatrix} -1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} \right) = -3/\sqrt{2}$$

## Scalar product

$$\mathbf{w} \cdot \mathbf{x} = \sum_{i=1}^n w_i x_i = \mathbf{w}^\top \mathbf{x}$$

The length of a vector is denoted  $|\mathbf{x}|$ , the squared length is given by

$$|\mathbf{x}|^2 = \mathbf{x}^\top \mathbf{x} = \mathbf{x}^2 = x_1^2 + x_2^2 + \cdots + x_n^2$$

A unit vector  $\mathbf{x}$  has  $\mathbf{x}^\top \mathbf{x} = 1$ . The scalar product has a natural geometric interpretation as:

$$\mathbf{w} \cdot \mathbf{x} = |\mathbf{w}| |\mathbf{x}| \cos(\theta)$$

where  $\theta$  is the angle between the two vectors. Thus if the lengths of two vectors are fixed their inner product is largest when  $\theta = 0$ , whereupon one vector is a constant multiple of the other. If the scalar product  $\mathbf{x}^\top \mathbf{y} = 0$ , then  $\mathbf{x}$  and  $\mathbf{y}$  are orthogonal.

## Linear dependence

- A set of vectors  $\mathbf{x}^1, \dots, \mathbf{x}^n$  is linearly dependent if there exists a vector  $\mathbf{x}^j$  that can be expressed as a linear combination of the other vectors.
- If the only solution to

$$\sum_{i=1}^n \alpha_i \mathbf{x}^i = \mathbf{0}$$

is for all  $\alpha_i = 0, i = 1, \dots, n$ , the vectors  $\mathbf{x}^1, \dots, \mathbf{x}^n$  are linearly independent.

## Determinant

For a square matrix  $\mathbf{A}$ , the determinant is the volume of the transformation of the matrix  $\mathbf{A}$  (up to a sign change). Writing  $[\mathbf{A}]_{ij} = a_{ij}$ ,

$$\det \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = a_{11}a_{22} - a_{21}a_{12}$$

The determinant in the  $(3 \times 3)$  case has the form

$$a_{11}\det \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix} - a_{12}\det \begin{pmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{pmatrix} + a_{13}\det \begin{pmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}$$

More generally, the determinant can be computed recursively as an expansion along the top row of determinants of reduced matrices.

$$\det(\mathbf{A}^T) = \det(\mathbf{A})$$

$$\det(\mathbf{AB}) = \det(\mathbf{A})\det(\mathbf{B}), \quad \det(\mathbf{I}) = 1 \Rightarrow \det(\mathbf{A}^{-1}) = 1/\det(\mathbf{A})$$

## Matrix inversion

For a square matrix  $\mathbf{A}$ , its inverse satisfies

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{I} = \mathbf{A}\mathbf{A}^{-1}$$

It is not always possible to find a matrix  $\mathbf{A}^{-1}$  such that  $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ , in which case  $\mathbf{A}$  singular. Geometrically, singular matrices correspond to projections. Provided the inverses exist

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$$

---

## Pseudo inverse

For a non-square matrix  $\mathbf{A}$  such that  $\mathbf{A}\mathbf{A}^T$  is invertible,

$$\mathbf{A}^\dagger = \mathbf{A}^T (\mathbf{A}\mathbf{A}^T)^{-1}$$

satisfies  $\mathbf{A}\mathbf{A}^\dagger = \mathbf{I}$ .

# Solving Linear Systems

## Problem

Given square  $N \times N$  matrix  $\mathbf{A}$  and vector  $\mathbf{b}$ , find the vector  $\mathbf{x}$  that satisfies

$$\mathbf{Ax} = \mathbf{b}$$

---

## Solution

Algebraically, we have the inverse:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

In practice, we solve solve for  $\mathbf{x}$  numerically using Gaussian Elimination – more stable numerically.

---

## Complexity

Solving a linear system is  $O(N^3)$  – can be very expensive for large  $N$ .  
Approximate methods include conjugate gradient and related approaches.

## Matrix rank

For an  $m \times n$  matrix  $\mathbf{X}$  with  $n$  columns, each written as an  $m$ -vector:

$$\mathbf{X} = [\mathbf{x}^1, \dots, \mathbf{x}^n]$$

the rank of  $\mathbf{X}$  is the maximum number of linearly independent columns (or equivalently rows).

---

### Full rank

An  $n \times n$  square matrix is full rank if the rank is  $n$ , in which case the matrix is must be non-singular. Otherwise the matrix is reduced rank and is singular.

## Orthogonal matrix

A square matrix  $\mathbf{A}$  is orthogonal if

$$\mathbf{AA}^T = \mathbf{I} = \mathbf{A}^T\mathbf{A}$$

From the properties of the determinant, we see therefore that an orthogonal matrix has determinant  $\pm 1$  and hence corresponds to a volume preserving transformation.

$$\det(\mathbf{AA}^T) = \det(\mathbf{I})$$

$$\det(\mathbf{A}) \det(\mathbf{A}^T) = 1$$

$$\det(\mathbf{A})^2 = 1$$

This means that the transformation that  $\mathbf{A}$  represents is something like a rotation, reflection or shear.

# Linear transformations

## Cartesian coordinate system

Define  $\mathbf{u}_i$  to be the vector with zeros everywhere except for the  $i^{th}$  entry, then a vector can be expressed as  $\mathbf{x} = \sum_i x_i \mathbf{u}_i$ .

---

## Linear transformation

- What does a matrix represent in terms of a transformation?

$$\mathbf{A}\mathbf{u}_i = \mathbf{a}_i$$

where  $\mathbf{a}_i$  is the  $i^{th}$  column of  $\mathbf{A}$ .

- That is, the columns of the matrix  $\mathbf{A}$  represent where the cartesian basis vectors get transformed to.
- More generally, a linear transformation of  $\mathbf{x}$  is given by matrix multiplication by some matrix  $\mathbf{A}$

$$\mathbf{Ax} = \sum_i x_i \mathbf{A}\mathbf{u}_i = \sum_i x_i \mathbf{a}_i$$

## Eigenvalues and eigenvectors

For an  $n \times n$  square matrix  $\mathbf{A}$ ,  $\mathbf{e}$  is an eigenvector of  $\mathbf{A}$  with eigenvalue  $\lambda$  if

$$\mathbf{A}\mathbf{e} = \lambda\mathbf{e}$$

For an  $(n \times n)$  dimensional matrix, there are (including repetitions)  $n$  eigenvalues, each with a corresponding eigenvector. We can write

$$\underbrace{(\mathbf{A} - \lambda\mathbf{I})}_{\mathbf{B}} \mathbf{e} = \mathbf{0}$$

If  $\mathbf{B}$  has an inverse, then the only solution is  $\mathbf{e} = \mathbf{B}^{-1}\mathbf{0} = \mathbf{0}$ , which trivially satisfies the eigen-equation. For any non-trivial solution we therefore need  $\mathbf{B}$  to be non-invertible. Hence  $\lambda$  is an eigenvalue of  $\mathbf{A}$  if

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0$$

It may be that for an eigenvalue  $\lambda$  the eigenvector is not unique and there is a space of corresponding vectors.

## Spectral decomposition

A real symmetric matrix  $N \times N$   $\mathbf{A}$  has an eigen-decomposition

$$\mathbf{A} = \sum_{i=1}^n \lambda_i \mathbf{e}_i \mathbf{e}_i^\top$$

where  $\lambda_i$  is the eigenvalue of eigenvector  $\mathbf{e}_i$  and the eigenvectors form an orthogonal set,

$$(\mathbf{e}^i)^\top \mathbf{e}^j = \delta_{ij} (\mathbf{e}^i)^\top \mathbf{e}^i$$

In matrix notation

$$\mathbf{A} = \mathbf{E} \Lambda \mathbf{E}^\top$$

where  $\mathbf{E}$  is the orthogonal matrix of eigenvectors and  $\Lambda$  the corresponding diagonal eigenvalue matrix.

---

### Computational Complexity

It generally takes  $O(N^3)$  time to compute the eigen-decomposition.

# Singular Value Decomposition

The SVD decomposition of a  $n \times p$  matrix  $\mathbf{X}$  is

$$\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^T$$

where  $\dim \mathbf{U} = n \times n$  with  $\mathbf{U}^T\mathbf{U} = \mathbf{I}_n$ . Also  $\dim \mathbf{V} = p \times p$  with  $\mathbf{V}^T\mathbf{V} = \mathbf{I}_p$ .

- The matrix  $\mathbf{S}$  has  $\dim \mathbf{S} = n \times p$  with zeros everywhere except on the diagonal entries.
- The singular values are the diagonal entries  $[\mathbf{S}]_{ii}$  and are positive.
- The singular values are ordered so that the upper left diagonal element of  $\mathbf{S}$  contains the largest singular value.

---

## Computational Complexity

It takes  $O\left(\max(n, p)(\min(n, p))^2\right)$  time to compute the SVD-decomposition.

# Positive definite matrix

- A symmetric matrix  $\mathbf{A}$  with the property that  $\mathbf{x}^T \mathbf{A} \mathbf{x} \geq 0$  for any vector  $\mathbf{x}$  is called positive semidefinite.
- A symmetric matrix  $\mathbf{A}$ , with the property that  $\mathbf{x}^T \mathbf{A} \mathbf{x} > 0$  for any vector  $\mathbf{x} \neq 0$  is called positive definite.
- A positive definite matrix has full rank and is thus invertible.

---

## Eigen-decomposition

Using the eigen-decomposition of  $\mathbf{A}$ ,

$$\mathbf{x}^T \mathbf{A} \mathbf{x} = \sum_i \lambda_i \mathbf{x}^T \mathbf{e}^i (\mathbf{e}^i)^T \mathbf{x} = \sum_i \lambda_i (\mathbf{x}^T \mathbf{e}^i)^2$$

which is greater than zero if and only if all the eigenvalues are positive. Hence  $\mathbf{A}$  is positive definite if and only if all its eigenvalues are positive.

## Trace and Det

$$\text{trace}(\mathbf{A}) = \sum_i A_{ii} = \sum_i \lambda_i$$

where  $\lambda_i$  are the eigenvalues of  $\mathbf{A}$ .

$$\det(\mathbf{A}) = \prod_{i=1}^n \lambda_i$$

A matrix is singular if it has a zero eigenvalue.

---

### Trace-Log formula

For a positive definite matrix  $\mathbf{A}$ ,

$$\text{trace}(\log \mathbf{A}) \equiv \log \det(\mathbf{A})$$

The above logarithm of a matrix is not the element-wise logarithm. In general for an analytic function  $f(x)$ ,  $f(\mathbf{M})$  is defined via the Taylor series expansion of the function. On the right, since  $\det(\mathbf{A})$  is a scalar, the logarithm is the standard logarithm of a scalar.

Calculus

# Calculus

For a function  $f(x)$ , the derivative is defined to be

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

This is also often written as  $f'(x)$  for convenience.

The second derivative is defined to be the derivative of the derivative:

$$\frac{d^2 f}{dx^2} = \lim_{\delta \rightarrow 0} \frac{\frac{df}{dx}(x + \delta) - \frac{df}{dx}(x)}{\delta}$$

also written as  $f''(x)$  for convenience.

Note that the funny notation is because one can think of the derivative as an operator  $\frac{d}{dx}$  that we apply to the function  $f(x)$ . The second derivative is given by applying this operator twice:  $(\frac{d}{dx})^2$  which is more conveniently written as  $\frac{d^2}{dx^2}$ .

---

## Taylor Series

Any smooth function can be written as

$$\begin{aligned} f(x) &= f(0) + \sum_{i=1}^{\infty} \frac{x^i}{i!} \left. \left( \frac{d}{dx} \right)^i f(x) \right|_{x=0} \\ &= f(0) + x \frac{df}{dx} + \frac{x^2}{2} \frac{d^2 f}{dx^2} + \dots \end{aligned}$$

# Some Calculus Rules

## Chain Rule

For a function of a function  $f(g(x))$  (e.g.  $\sin(\cos(x))$ )

$$\frac{d(f(g(x)))}{dx} = \left. \frac{df(y)}{dy} \right|_{y=f(x)} \frac{dg}{dx}$$

which is usually more conveniently written as

$$\frac{d(f(g(x)))}{dx} = \frac{df}{dg} \frac{dg}{dx}$$

---

## Sum Rule

The differential operator is a linear operator and therefore

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

---

## Product Rule

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

# Numerical Approximation

Take a finite (small value) for  $\delta$ . Then

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

---

## Central Difference

Using the Taylor series, we can write

$$f(x + \delta) \approx f(x) + \delta f'(x) + \frac{\delta^2}{2} f''(x) + O(\delta^3)$$

$$f(x - \delta) \approx f(x) - \delta f'(x) + \frac{\delta^2}{2} f''(x) + O(\delta^3)$$

Subtracting, we can rearrange to give

$$f'(x) \approx \frac{f(x + \delta) - f(x - \delta)}{2\delta} + O(\delta^3)$$

At the cost of an additional function evaluation, we therefore have a *much* more accurate approximation.

## Partial and Total Derivative

For a function that depends on two (or more) variables  $f(x, y)$ , the partial derivative with respect to  $x$  is defined as

$$\frac{\partial f}{\partial x} = \lim_{\delta \rightarrow 0} \frac{f(x + \delta, y) - f(x, y)}{\delta}$$

That is, the partial derivative with respect to  $x$  keeps the state of all other variables fixed.

- Consider a function  $f(x)$  that depends directly on  $x$  in some manner, and indirectly through another function. We want to find the change in  $f$  as we change  $x$ , accounting also for indirect changes.
- Consider, for example

$$f(x) = x^2 + xg, \quad \text{where } g(x) = x^2$$

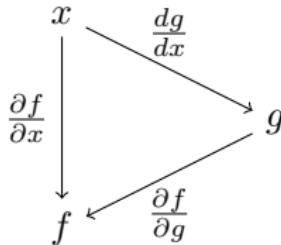
Then  $df/dx$  (the total derivative) is given by

$$\begin{aligned}\frac{df}{dx} &= \frac{\partial f}{\partial x} + \frac{\partial f}{\partial g} \frac{dg}{dx} \\ &= \underbrace{2x + g}_{\text{partial derivative}} + \underbrace{x}_{\text{p.d wrt } y} \underbrace{2x}_{\text{t.d of } g}\end{aligned}$$

## Partial and Total Derivative (Graphical Representation)

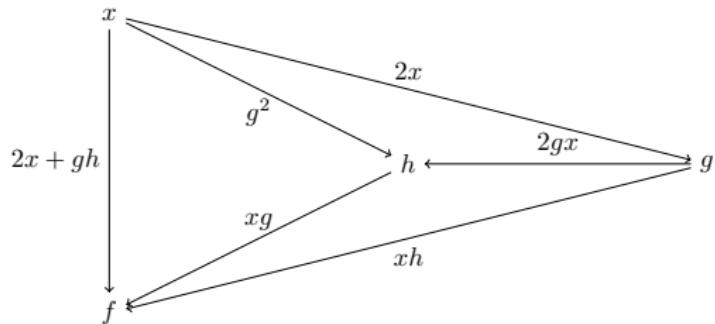
A useful graphical representation is that the total derivative of  $f$  with respect to  $x$  is given by the sum over all path values from  $x$  to  $f$ , where each path value is the product of the derivatives of the functions on the edges:

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



### Example

For  $f(x) = x^2 + xgh$ , where  $g = x^2$  and  $h = xg^2$



$$f'(x) = (2x + gh) + (g^2 xg) + (2x2gxxg) + (2xxh) = 2x + 8x^7$$

# Multivariate Calculus

## Partial derivative

Consider a function of  $n$  variables,  $f(x_1, x_2, \dots, x_n)$  or  $f(\mathbf{x})$ . The partial derivative of  $f$  w.r.t.  $x_i$  is defined as the following limit (when it exists)

$$\frac{\partial f}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_{i-1}, x_i + h, x_{i+1}, \dots, x_n) - f(\mathbf{x})}{h}$$

---

## Gradient vector

For function  $f$  the gradient is denoted  $\nabla f$  or  $\mathbf{g}$ :

$$\nabla f(\mathbf{x}) \equiv \mathbf{g}(\mathbf{x}) \equiv \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

## Interpreting the gradient vector

- Consider a function  $f(\mathbf{x})$  that depends on a vector  $\mathbf{x}$ .
- We are interested in how the function changes when the vector  $\mathbf{x}$  changes by a small amount :  $\mathbf{x} \rightarrow \mathbf{x} + \boldsymbol{\delta}$ , where  $\boldsymbol{\delta}$  is a vector whose length is very small:

$$f(\mathbf{x} + \boldsymbol{\delta}) = f(\mathbf{x}) + \sum_i \delta_i \frac{\partial f}{\partial x_i} + O(\boldsymbol{\delta}^2)$$

- We can interpret the summation above as the scalar product between the vector  $\nabla f$  with components  $[\nabla f]_i = \frac{\partial f}{\partial x_i}$  and  $\boldsymbol{\delta}$ .

$$f(\mathbf{x} + \boldsymbol{\delta}) = f(\mathbf{x}) + (\nabla f)^\top \boldsymbol{\delta} + O(\boldsymbol{\delta}^2)$$

# Interpreting the Gradient

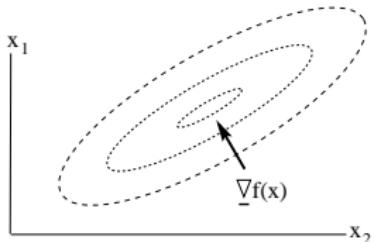


Figure : Interpreting the gradient. The ellipses are contours of constant function value,  $f = \text{const}$ . At any point  $\mathbf{x}$ , the gradient vector  $\nabla f(\mathbf{x})$  points along the direction of maximal increase of the function.

Consider a direction  $\hat{\mathbf{p}}$  (a unit length vector). Then a displacement,  $\delta$  units along this direction changes the function value to

$$f(\mathbf{x} + \delta \hat{\mathbf{p}}) \approx f(\mathbf{x}) + \delta \nabla f(\mathbf{x}) \cdot \hat{\mathbf{p}}$$

The direction  $\hat{\mathbf{p}}$  for which the function has the largest change is that which maximises the overlap

$$\nabla f(\mathbf{x}) \cdot \hat{\mathbf{p}} = |\nabla f(\mathbf{x})| |\hat{\mathbf{p}}| \cos \theta = |\nabla f(\mathbf{x})| \cos \theta$$

where  $\theta$  is the angle between  $\hat{\mathbf{p}}$  and  $\nabla f(\mathbf{x})$ . The overlap is maximised when  $\theta = 0$ , giving  $\hat{\mathbf{p}} = \nabla f(\mathbf{x}) / |\nabla f(\mathbf{x})|$ . Hence, the direction along which the function changes the most rapidly is along  $\nabla f(\mathbf{x})$ .

## Higher derivatives

The 'second-derivative' of an  $n$ -variable function is defined by

$$\frac{\partial}{\partial x_i} \left( \frac{\partial f}{\partial x_j} \right) \quad i = 1, \dots, n; \quad j = 1, \dots, n$$

which is usually written

$$\frac{\partial^2 f}{\partial x_i \partial x_j}, \quad i \neq j \quad \frac{\partial^2 f}{\partial x_i^2}, \quad i = j$$

If the partial derivatives  $\partial f / \partial x_i$ ,  $\partial f / \partial x_j$  and  $\partial^2 f / \partial x_i \partial x_j$  are continuous, then  $\partial^2 f / \partial x_i \partial x_j$  exists and

$$\partial^2 f / \partial x_i \partial x_j = \partial^2 f / \partial x_j \partial x_i.$$

This is also denoted by  $\nabla \nabla f$ . These  $n^2$  second partial derivatives are represented by a square, symmetric matrix called the Hessian matrix of  $f(\mathbf{x})$ .

$$\mathbf{H}_f(\mathbf{x}) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

## Vector Taylor Series

For a scalar function of a vector argument, the first terms of the expansion are

$$f(\mathbf{x} + \boldsymbol{\delta}) \approx f(\mathbf{x}) + \boldsymbol{\delta}^T \mathbf{g} + \frac{1}{2} \boldsymbol{\delta}^T \mathbf{H} \boldsymbol{\delta}$$

where  $\mathbf{g}$  is the gradient vector of  $f$ , evaluated at  $\mathbf{x}$  and  $\mathbf{H}$  is the Hessian of  $f$ , evaluated at  $\mathbf{x}$ .

- If  $\mathbf{H}$  is positive definite, the function looks locally like a bowl  $\cup$  around the point  $\mathbf{x}$ .
- If  $\mathbf{H}$  is negative definite, the function looks locally like an upturned bowl  $\cap$  around the point  $\mathbf{x}$ .
- If  $\mathbf{H}$  is non-definite (neither positive nor negative), there are directions through  $\mathbf{x}$  along which the function looks like  $\cup$  and others along which it looks like  $\cap$ .

# Matrix calculus

For matrices  $\mathbf{A}$  and  $\mathbf{B}$

$$\frac{\partial}{\partial \mathbf{A}} \text{trace}(\mathbf{AB}) \equiv \mathbf{B}^T$$

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

So that

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

For an invertible matrix  $\mathbf{A}$ ,

$$\partial \mathbf{A}^{-1} \equiv -\mathbf{A}^{-T} \partial \mathbf{A} \mathbf{A}^{-1}$$

# Convex Analysis

## Convex Function

- A function  $f(\mathbf{x})$  is convex if, for any two point  $\mathbf{x}$  and  $\mathbf{y}$  and  $0 < \lambda < 1$

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

- If  $f$  is twice differentiable,  $f(\mathbf{x})$  is convex if its Hessian  $\mathbf{H}(\mathbf{x})$  is positive definite for all points  $\mathbf{x}$ .

---

## Optimisation

- Geometrically, (strictly i.e. the above is  $<$  not  $\leq$ ) convex functions look like  and have only one minimum.
- Convex functions are very important since there are typically very efficient algorithms that guarantee to find the global minimum of the function.
- A function  $f(\mathbf{x})$  is concave if  $-f(\mathbf{x})$  is convex.
- In much of machine learning, we need to learn parameters through some form of optimisation. If the objective function is convex, this will make parameter learning straightforward.

# Properties of Convex functions

## Norms are convex

All norms are convex, in particular the  $p$ -norm

$$\|x\|_p \equiv \left( \sum_i |x_i|^p \right)^{1/p}, \quad p \geq 1$$

---

## Compositions

If  $f$  and  $g$  are convex then:

- $f + g$  is convex (positive sums of convex functions are convex)
- $f(Ax + b)$  is convex ('affine transformation')
- $f(g(x))$  is convex provided  $f$  is an increasing function

---

## Log convexity

- In machine learning we often encounter 'log convex' functions. This means a function  $g$  such that  $f$ , where  $f(x) = \log g(x)$ , is convex.
- For example  $g(x) = \exp(x^2)$  is log convex.

Exercises: Show the following functions are convex

$$f(x) = x^2$$

---

$$f(x) = -\log \sigma(x), \text{ where } \sigma(x) = 1/(1 + \exp(-x))$$

Exercises: Show the following functions are convex

$$f(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{x} \text{ for positive definite } \mathbf{A}$$

---

$$f(\mathbf{x}) = -\log \sigma(\mathbf{x}^T \mathbf{w}), \text{ where } \sigma(x) = 1/(1 + \exp(-x))$$

## Numerical Issues

## Numerical issues: rounding error

- Often in machine learning we have a large number of terms to sum, for example when computing the log likelihood for a large number of datapoints.
- It's good to be aware of potential numerical limitations and ways to improve accuracy, should this be a problem. Double floats have a relative error of around  $1 \times 10^{-16}$ .
- Operations that are mathematical identities may not remain so. For example

$$\sum_n x_i^n x_j^n$$

should give rise to a symmetric matrix. However, this symmetry can be lost due to roundoff.

- In general, it's worth checking key points in your code for such issues.

## Numerical issues: rounding error

- Consider

$$S = \sum_{i=1}^N x_i$$

If  $x_i$  cannot be represented exactly by your machine, round-off error will potentially accumulate in computing  $S$ .

- Let  $y$  be an ‘approximation’ to each  $x_i$ , then

$$S = \sum_{i=1}^N (x_i - y + y) = Ny + \sum_{i=1}^N (x_i - y)$$

If each  $x_i$  is close to  $y$ , then the term  $\sum_{i=1}^N (x_i - y)$  is small but not sensitive to round off error (since each term is small and has roughly the same value).

See `testacc.m` for an example.

## logsumexp

- It's common in ML to come across expressions such as

$$S = \exp(a) + \exp(b)$$

for large (in absolute value)  $a$  or  $b$ . If  $a = 1000$ , Matlab will return  $\infty$  (0 for  $a = -1000$ ). It's not sufficient to simply compute the log:

$$\log S = \log(\exp(a) + \exp(b))$$

since this requires the exponentiation still of each term.

- Let  $m = \max(a, b)$ .

$$\log S = m + \log(\exp(a - m) + \exp(b - m))$$

Let's say that  $m = a$ , then

$$\log S = a + \log(1 + \exp(b - a))$$

Since  $a > b$  then  $\exp(b - a) < 1$  and  $\log(1 + \exp(b - a)) < \log 2$ . We can compute  $\log S$  more accurately this way.

- More generally, we define the logsumexp function

$$\text{logsumexp}(\mathbf{x}) = m + \log\left(\sum_{i=1}^N \exp(x_i - m)\right), \quad m = \max(x_1, \dots, x_N)$$

## logsumexp: example

In a classification problem of a 100 dimensional vector  $\mathbf{x}$ ,

$$p(c = i|\mathbf{x}) = \frac{e^{-(\mathbf{x}-\mathbf{m}_i)^2}}{\sum_j e^{-(\mathbf{x}-\mathbf{m}_j)^2}}$$

A naive implementation of this is likely to lead to  $\frac{0}{0}$  and a numerical error.

---

## Using logsumexp

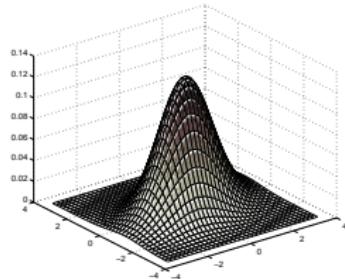
$$\log p(c = i|\mathbf{x}) = y_i - \text{logsumexp}(\mathbf{y})$$

where

$$y_j = -(\mathbf{x} - \mathbf{m}_j)^2$$

# Distributions

# Multivariate Gaussian



$$p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \equiv \frac{1}{\sqrt{\det(2\pi\boldsymbol{\Sigma})}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu})}$$

- $\boldsymbol{\mu}$  is the mean vector of the distribution:

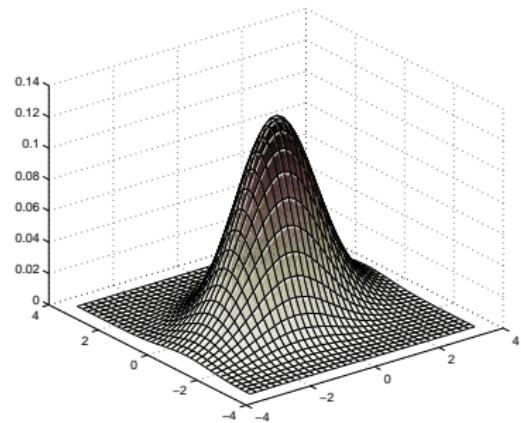
$$\boldsymbol{\mu} = \langle \mathbf{x} \rangle_{\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})}$$

- $\boldsymbol{\Sigma}$  is the covariance matrix of the distribution.

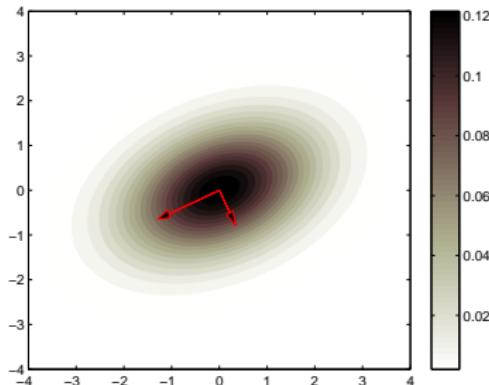
$$\boldsymbol{\Sigma} = \left\langle (\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^\top \right\rangle_{\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})}$$

- $\int_{-\infty}^{\infty} p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) d\mathbf{x} = 1$ .

# Geometric Picture



(a)



(b)

**Figure :** **(a)**: Bivariate Gaussian with mean  $(0, 0)$  and covariance  $[1, 0.5; 0.5, 1.75]$ . Plotted on the vertical axis is the probability density value  $p(x)$ . **(b)**: Probability density contours for the same bivariate Gaussian. Plotted are the unit eigenvectors scaled by the square root of their eigenvalues,  $\sqrt{\lambda_i}$ .

## Geometric Picture

Every real symmetric matrix  $D \times D$  has an eigen-decomposition

$$\Sigma = \mathbf{E}\Lambda\mathbf{E}^T$$

where  $\mathbf{E}^T\mathbf{E} = \mathbf{I}$  and  $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_D)$ . In the case of a covariance matrix, all the eigenvalues  $\lambda_i$  are positive. This means that one can use the transformation

$$\mathbf{y} = \Lambda^{-\frac{1}{2}}\mathbf{E}^T(\mathbf{x} - \boldsymbol{\mu})$$

so that

$$(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu}) = (\mathbf{x} - \boldsymbol{\mu})^T \mathbf{E} \Lambda^{-1} \mathbf{E}^T (\mathbf{x} - \boldsymbol{\mu}) = \mathbf{y}^T \mathbf{y}$$

Under this transformation, the multivariate Gaussian reduces to a product of  $D$  univariate zero-mean unit variance Gaussians. This means that we can view a multivariate Gaussian as a shifted, scaled and rotated version of a 'standard' (zero mean, unit covariance) Gaussian in which the centre is given by the mean, the rotation by the eigenvectors, and the scaling by the square root of the eigenvalues.

## Linear Transform of a Gaussian

- Let  $\mathbf{y}$  be linearly related to  $\mathbf{x}$  through

$$\mathbf{y} = \mathbf{M}\mathbf{x} + \boldsymbol{\eta}$$

where  $\boldsymbol{\eta} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ , and  $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}_x, \Sigma_x)$ .

- Then the marginal  $p(\mathbf{y}) = \int_{\mathbf{x}} p(\mathbf{y}|\mathbf{x})p(\mathbf{x})$  is a Gaussian

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{y} | \mathbf{M}\boldsymbol{\mu}_x + \boldsymbol{\mu}, \mathbf{M}\Sigma_x\mathbf{M}^T + \Sigma)$$

---

### Decorrelating (whitening)

If  $\mathbf{x}$  has covariance matrix  $\Sigma_x$  and mean  $\boldsymbol{\mu}_x$ , then

$$\mathbf{y} = \Sigma_x^{-1/2} (\mathbf{x} - \boldsymbol{\mu}_x)$$

has mean  $\mathbf{0}$  and identity covariance matrix. A commonly used initial transformation on data.

# Regression (Linear Models)

David Barber

# Regression

- This is a form of supervised learning.
  - We have a dataset of input-output examples  $(\mathbf{x}^n, y^n), n = 1, \dots, N$
  - We want to learn a mapping from inputs  $\mathbf{x}$  to output  $y$  such that when we get a new input  $\mathbf{x}$ , we will output the correct output.
- 

## Example

- Given a set of vectors with features (attributes)

$$\begin{pmatrix} \text{client age} \\ \text{client postcode indicator} \\ \text{client monthly tesco spend} \\ \text{client purchase types} \end{pmatrix}$$

predict their annual income.

- We may have a very large set of such features, only some of which may be relevant for the prediction.

# Generalisation

## Basic Assumptions

- The data we have (all inputs and corresponding outputs) is generated from the same unvarying mechanism.
- We will typically split our available data into three distinct parts: train data, validation data and test data.
- We want to find a model that will have good generalisation performance.

---

## Validation

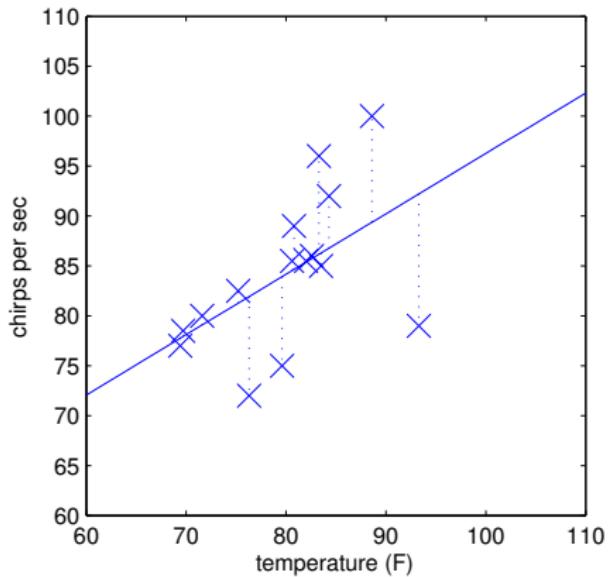


Different models can be trained using the train data. The optimal model is determined by the empirical performance on the validation data. An independent measure of the generalisation performance of this optimal model is obtained by using a separate test set.

## Fitting a Line

Given training data  $\{(x^n, y^n), n = 1, \dots, N\}$ , a linear regression fit is

$$y(x) = a + bx$$



Data from singbats – the number of chirps per second, versus the temperature in Fahrenheit. Straight line regression fit to the data.

## Least Squares fitting

$$E(a, b) = \sum_{n=1}^N [y^n - y(x^n)]^2 = \sum_{n=1}^N (y^n - a - bx^n)^2$$

To minimise  $E(a, b)$  we differentiate with respect to  $a$  and  $b$  we obtain

$$\frac{\partial}{\partial a} E(a, b) = -2 \sum_{n=1}^N (y^n - a - bx^n), \quad \frac{\partial}{\partial b} E(a, b) = -2 \sum_{n=1}^N (y^n - a - bx^n)x^n$$

- We then equate these two equations to zero to find the optimum.
- The resulting two linear equations can be solved to find  $a$  and  $b$ .

## Linear Parameter Models for Regression

We can generalise the above to fitting linear functions of vector inputs  $\mathbf{x}$ . For a dataset  $\{(\mathbf{x}^n, y^n), n = 1, \dots, N\}$ , a linear parameter model is

$$y(\mathbf{x}) = \mathbf{w}^\top \phi(\mathbf{x})$$

where  $\phi(\mathbf{x})$  is a vector valued function of the input vector  $\mathbf{x}$ . For example, in the case of a straight line fit, with scalar input and output, we have

$$\phi(x) = (1, x)^\top, \quad \mathbf{w} = (a, b)^\top,$$

We define the error as the sum of squared differences between the observed outputs and the predictions under the linear model:

$$E(\mathbf{w}) = \sum_{n=1}^N (y^n - \mathbf{w}^\top \phi^n)^2, \quad \text{where } \phi^n \equiv \phi(\mathbf{x}^n)$$

## Normal Equations

Differentiating  $E(\mathbf{w})$  with respect to  $w_k$ , and equating to zero gives

$$\sum_{n=1}^N y^n \phi_k^n = \sum_i w_i \sum_{n=1}^N \phi_i^n \phi_k^n$$

or, in matrix notation,

$$\sum_{n=1}^N y^n \phi^n = \sum_{n=1}^N \phi^n (\phi^n)^\top \mathbf{w}$$

These are called the normal equations, for which the solution is

$$\mathbf{w} = \left( \sum_{n=1}^N \phi^n (\phi^n)^\top \right)^{-1} \sum_{n=1}^N y^n \phi^n$$

Although we write the solution using matrix inversion, in practice one finds the numerical solution using Gaussian elimination since this is faster and numerically more stable.

# Polynomial Fitting

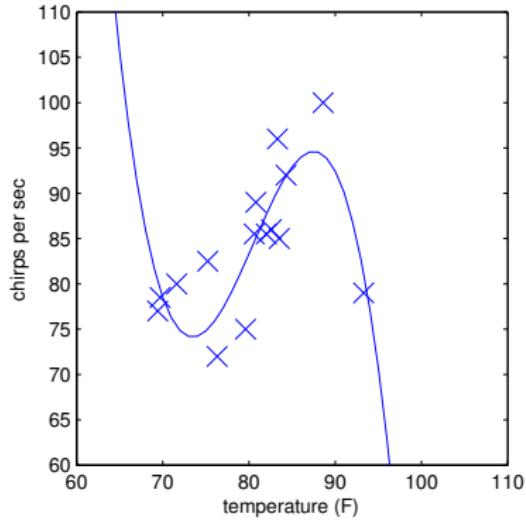


Figure : Cubic polynomial fit to the singbat data.

A cubic polynomial is given by

$$y(x) = w_1 + w_2x + w_3x^2 + w_4x^3$$

As a LPM, this can be expressed using

$$\phi(x) = (1, x, x^2, x^3)^T$$

# Regularisation

- Sensible to first scale each input dimension  $x_i$  so that it has unit variance:

$$x_i^n \rightarrow \frac{x_i^n}{\sigma_i}$$

where the variance in the  $i^{th}$  dimension is given by

$$\sigma_i^2 = \frac{1}{N} \sum_{n=1}^N (x_i^n - \mu_i)^2, \quad \mu_i = \frac{1}{N} \sum_{n=1}^N x_i^n$$

- Add an extra regularising term  $R(\mathbf{w})$  to penalise rapid changes in the output

$$E'(\mathbf{w}) = E(\mathbf{w}) + \lambda R(\mathbf{w})$$

where  $\lambda$  is a regularisation constant.

- We can determine  $\lambda$  using validation data (so that we find a  $\lambda$  that promotes good generalisation performance).

## $L_2$ regularisation or ‘Ridge Regression’

- Add a quadratic penalty to the objective. This prevents large weight values:

$$E'(\mathbf{w}) = \sum_{n=1}^N (y^n - \mathbf{w}^\top \phi^n)^2 + \lambda \mathbf{w}^\top \mathbf{w}$$

The optimal  $\mathbf{w}$  is given by

$$\mathbf{w} = \left( \sum_n \phi^n (\phi^n)^\top + \lambda \mathbf{I} \right)^{-1} \sum_{n=1}^N y^n \phi^n$$

- $\lambda$  may be determined using a validation set.
- Historically  $L_2$  regularisation is popular since there is an easy analytic (and computational) solution to finding the optimum.

## Issues with Ridge Regression

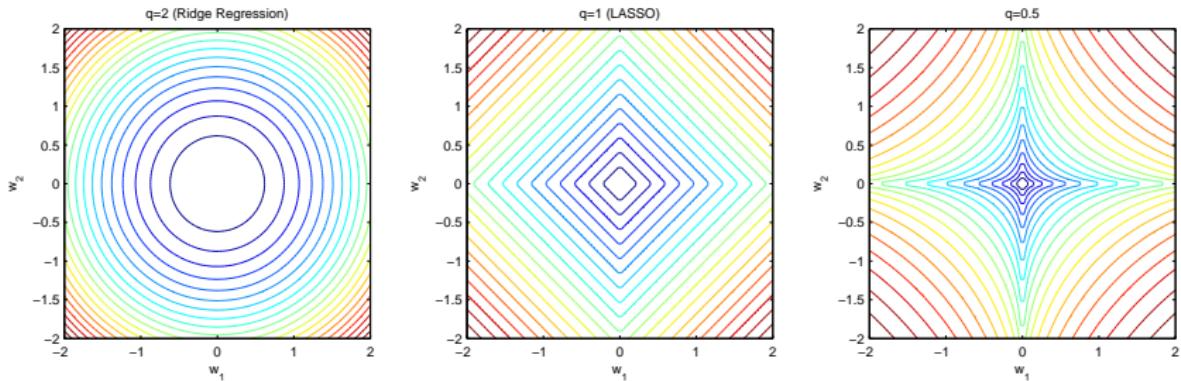
- Consider linear regression with

$$\frac{1}{N} \sum_{n=1}^N (y^n - w_1 x_1^n - w_2 x_2^n)^2 + w_1^2 + w_2^2$$

- Let  $x_1^n = x_2^n$  ('co-linear' case). Then either  $x_1$  or  $x_2$  is redundant.
- For  $w_1 = 1, w_2 = 0$  the ridge penalty is  $1^2 + 0^2 = 1$
- For  $w_1 = 1/2, w_2 = 1/2$  the ridge penalty is  $(1/2)^2 + (1/2)^2 = 1/2$
- Both cases above have the same squared loss, but the second one in which both weights are equally active is preferred by ridge regression.

# Different Penalty terms

Regularisation penalty  $\sum_i |w_i|^q$ .



- As we decrease  $q$  towards 0 we get the ‘ideal’ regulariser that selects weights which are 0 if they do not contribute.
- The objective function is complicated (non convex) for  $q < 1$  ●
- For  $q = 1$  the objective function for squared loss linear regression remains convex and easy to optimise ●

## $L_1$ verus $L_2$ regularisation

$$L_2(\mathbf{w}) = \sum_i w_i^2$$

- Can also be written as

$$\mathbf{w}^\top \mathbf{w} = \sum_i w_i^2$$

- This heavily penalises large  $w_i$  but only penalises by a small amount small  $w_i$ .
- The penalty is differentiable.
- Weights which are small will still persist.

---

$$L_1(\mathbf{w}) = \sum_i |w_i|$$

- This is the sum of the absolute values of the elements of  $\mathbf{w}$ .
- This heavily penalises large  $w_i$  and amount small  $w_i$  by a proportional amount.
- The penalty is non-differentiable.
- Need to use specialised optimisation routines.
- Only significant weights will remain – redundant parameters will be ‘pruned’ to zero.

## Understanding the $L_1$ penalty

- Consider a Loss function

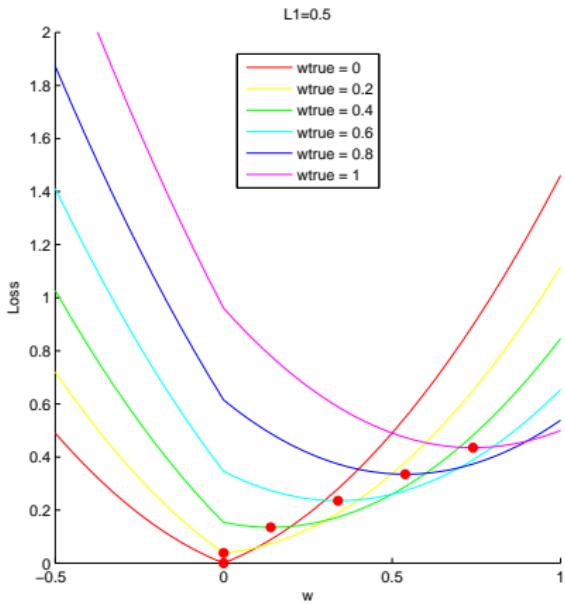
$$L(w) = \frac{1}{N} \sum_{n=1}^N (y^n - wx^n)^2 + \lambda|w|$$

where the data  $y^n$  is generated by

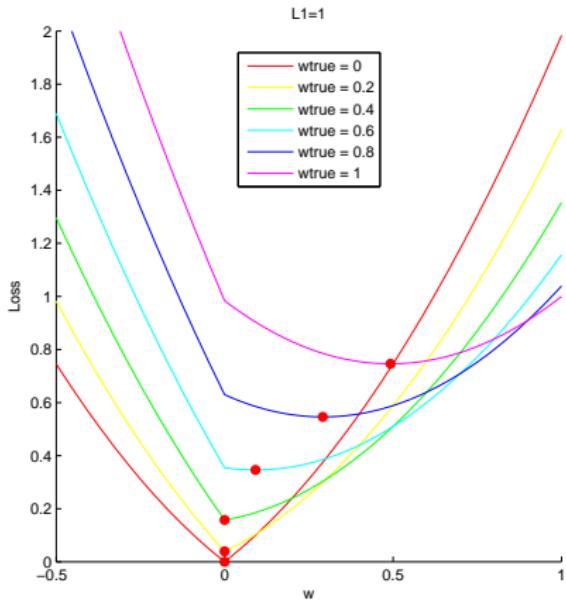
$$y^n = w_0 x^n$$

and the inputs  $x^n$  are randomly selected from the Gaussian.

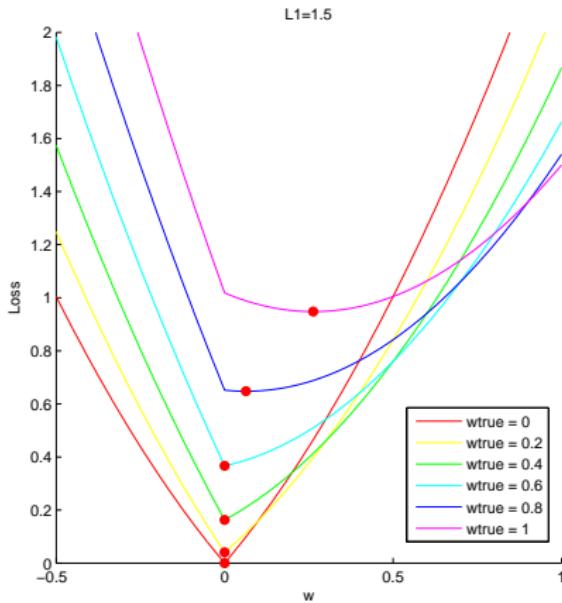
- Ideally we would like the minimum of  $L(w)$  to be at the point  $w = w_0$  since then we would recover the true data generating process.
- There is a complex interplay between minimising the squared loss term and the penalty.
- Only when the true  $w_0$  is large enough does the estimated  $w$  become non-zero.



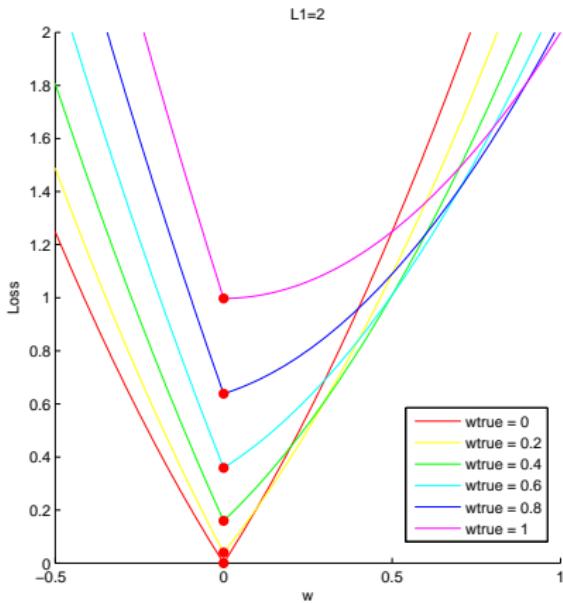
Here the amount of regularisation is small and, except for very small  $w_0$  we estimate non-zero  $w$ . The true values of  $w$  though are still underestimated.



As we increase the regularisation only for significantly non zero values of  $w_0$  is the estimated  $w$  non zero. The estimated  $w$  underestimates the true  $w_0$  more.



The regularisation penalty is so high that  $w$  is estimated to be zero, even when the true value is significantly non zero.



Here the amount of regularisation is small and, except for very small  $w_0$  we estimate non-zero  $w$ . The true values of  $w$  though are still underestimated.

## $L_1$ Pruning Effect

Consider a problem in which the train data is generated by  $y^n = w_0 x_1^n$ , and we try to fit a model

$$w_1 x_1 + w_2 x_2$$

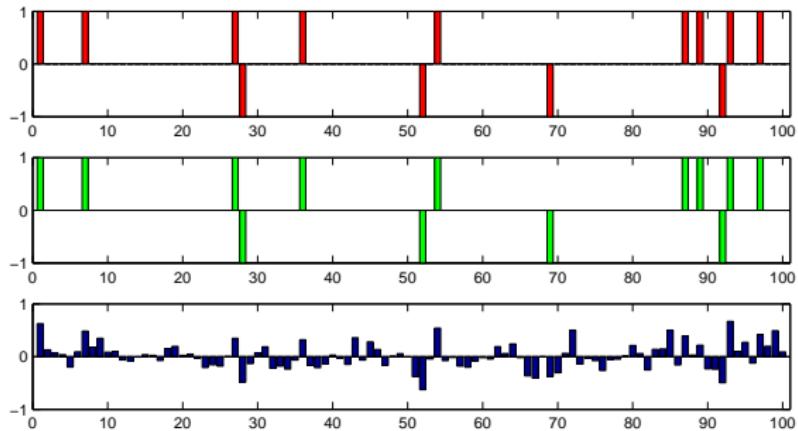
with  $x_2^n = x_1^n + \epsilon^n$ ,  $\epsilon^n \ll 1$ . In this case we don't need  $w_2$  and we'd like to see this set to 0 by the algorithm.

$$\begin{aligned} E(w_1, w_2) &\equiv \sum_n (y^n - w_1 x_1^n - w_2 x_2^n)^2 + \lambda(|w_1| + |w_2|) \\ &= \sum_n (w_0 x_1^n - w_1 x_1^n - w_2 x_2^n)^2 + \lambda(|w_1| + |w_2|) \\ &= \sum_n ((w_0 - w_1 - w_2) x_1^n - w_2 \epsilon^n)^2 + \lambda(|w_1| + |w_2|) \end{aligned}$$

$$E(w_0, 0) = \lambda |w_1|, \quad E(w_0/2, w_0/2) = w_2^2 \sum_n \epsilon_n^2 + \lambda |w_1|$$

Since  $E(w_0, 0) < E(w_0/2, w_0/2)$  the method prefers the sparse solution.

## $L_1$ versus $L_2$



- Linear regression based on 50 data points and 100 dimensional parameter vector (should be impossible?!)
- Top: true weights
- Middle:  $L_1$  regularised weights.
- Bottom:  $L_2$  regularised weights.
- Provided the true weight vector is sparse, even if we have fewer datapoints than parameters we can still recover the correct solution.

## Comments on $L_1$ regularisation

- $L_1$  is often used with linear parameter models since it results in simple objective functions that are easy to optimise.
- $L_1$  penalisation does not deal with the co-linear  $x$  case. This is not usually an issue in practice.
- To deal with co-linearity we need a non-convex objective function that is difficult to optimise.

---

### Expert feature selection versus $L_1$ penalisation

- The historical approach is to ask experts to decide which features (components of  $x$ ) should be included.
- The modern approach is to throw in all features that one might consider relevant and use  $L_1$  penalisation to determine which are relevant, with the non-relevant features being regularised to zero.

## Comments on $L_1$ regularisation

- For a non-linear regression model, the corresponding  $L_1$  regularised objective is more complex, typically non-convex.
- In this case need special optimisation methods to formally deal with strict  $L_1$  penalty.
- In practice (e.g. in deep learning), people often don't worry too much about the non-differentiability at the origin of the  $L_1$  penalty, and just proceed with gradient based training as normal.
- If the non-differentiability is a concern, simple differentiable approximations are popular, such as

$$|x| \leq \sqrt{x^2 + \epsilon^2}$$

and use a small  $\epsilon$ .

## Auto-Regressive Models

We can predict the future value  $v_t$  based on past values  $v_{t-1}, \dots, v_{t-L}$  using

$$v_t \approx \sum_{l=1}^L a_l v_{t-l}$$

where  $\mathbf{a} = (a_1, \dots, a_L)^\top$  are the 'AR coefficients'.

- The model predicts the future based on a linear combination of the previous  $L$  observations.
- This is exactly of the same form as our linear model

$$v_t \approx \mathbf{a}^\top \boldsymbol{\phi}^t$$

where  $\boldsymbol{\phi}^t = (v_{t-1}, v_{t-2}, \dots, v_{t-L})^\top$ .

- We can train this model as usual using the least squares objective with  $L_1$  or  $L_2$  regularisation.

## Auto-Regressive Models (Probabilistic Viewpoint)

We can predict the future value  $v_t$  based on past values  $v_{t-1}, \dots, v_{t-L}$  using

$$v_t = \sum_{l=1}^L a_l v_{t-l} + \eta_t, \quad \eta_t \sim \mathcal{N}(\eta_t | \mu, \sigma^2)$$

where  $\mathbf{a} = (a_1, \dots, a_L)^\top$  are the ‘AR coefficients’ and  $\sigma^2$  is the ‘innovation’ level.  
This is an  $L^{th}$  order Markov model:

$$p(v_{1:T}) = \prod_{t=1}^T p(v_t | v_{t-1}, \dots, v_{t-L}), \quad \text{with } v_i = \emptyset \text{ for } i \leq 0$$

with

$$p(v_t | v_{t-1}, \dots, v_{t-L}) = \mathcal{N}\left(v_t \left| \sum_{l=1}^L a_l v_{t-l}, \sigma^2\right.\right) = \mathcal{N}(v_t | \mathbf{a}^\top \hat{\mathbf{v}}_{t-1}, \sigma^2)$$

where

$$\hat{\mathbf{v}}_{t-1} \equiv [v_{t-1}, v_{t-2}, \dots, v_{t-L}]^\top$$

## Training an AR model (Probabilistic Viewpoint)

Maximum Likelihood training of the AR coefficients requires maximising

$$\log p(v_{1:T}) = \sum_{t=1}^T \log p(v_t | \hat{\mathbf{v}}_{t-1}) = -\frac{1}{2\sigma^2} \sum_{t=1}^T (v_t - \hat{\mathbf{v}}_{t-1}^\top \mathbf{a})^2 - \frac{T}{2} \log(2\pi\sigma^2)$$

Differentiating w.r.t.  $\mathbf{a}$  and equating to zero we arrive at

$$\sum_t (v_t - \hat{\mathbf{v}}_{t-1}^\top \mathbf{a}) \hat{\mathbf{v}}_{t-1} = 0$$

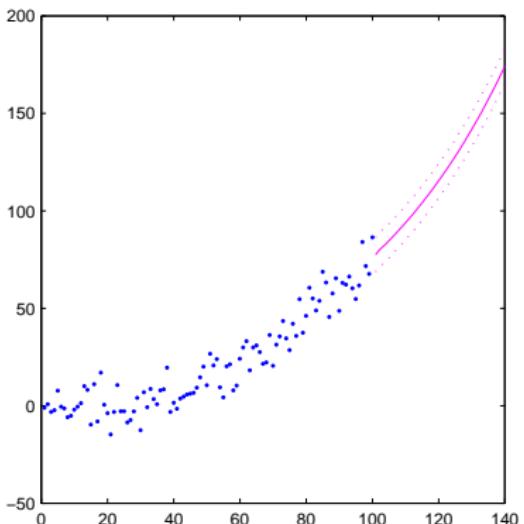
so that optimally

$$\mathbf{a} = \left( \sum_t \hat{\mathbf{v}}_{t-1} \hat{\mathbf{v}}_{t-1}^\top \right)^{-1} \sum_t v_t \hat{\mathbf{v}}_{t-1}$$

These equations can be solved by Gaussian elimination. Similarly, optimally,

$$\sigma^2 = \frac{1}{T} \sum_{t=1}^T (v_t - \hat{\mathbf{v}}_{t-1}^\top \mathbf{a})^2$$

## AR model: Fitting a trend



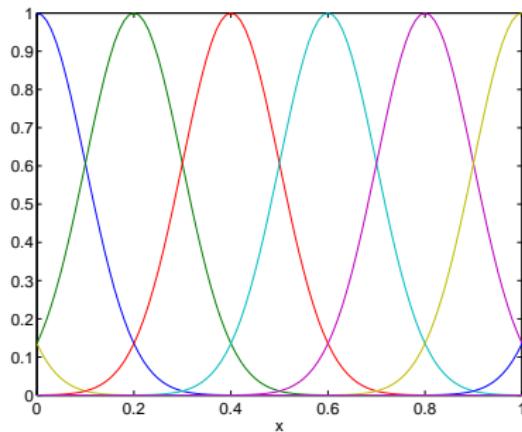
- Fitting an order 3 AR model to the training points.
- The  $x$  axis represents time, and the  $y$  axis the value of the timeseries.
- The solid line is the mean prediction and the dashed lines  $\pm$  one standard deviation.

## Radial basis functions

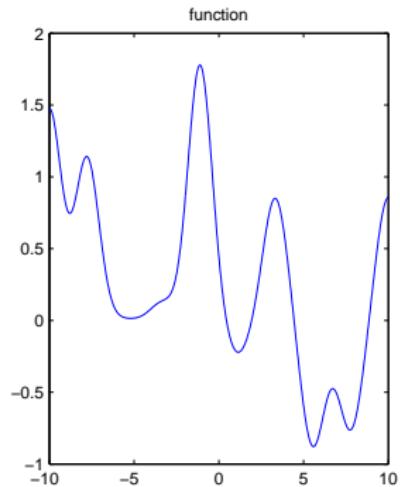
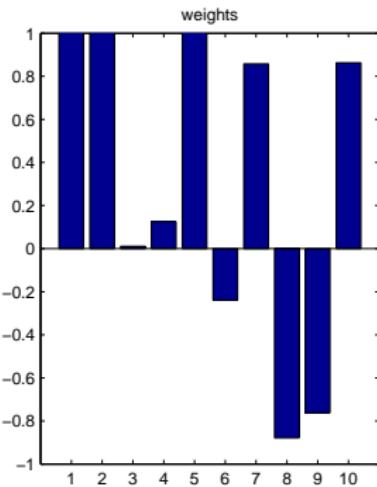
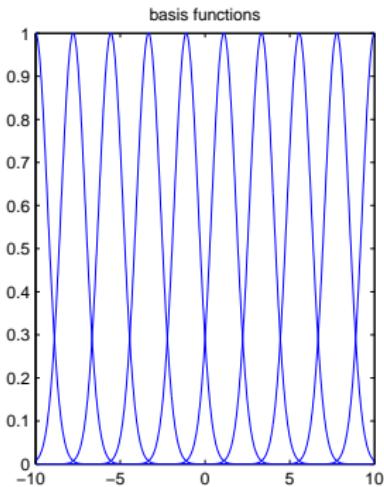
A popular LPM is given by the non-linear function

$$\phi_i(\mathbf{x}) = \exp\left(-\frac{1}{2\alpha^2} (\mathbf{x} - \mathbf{m}^i)^2\right)$$

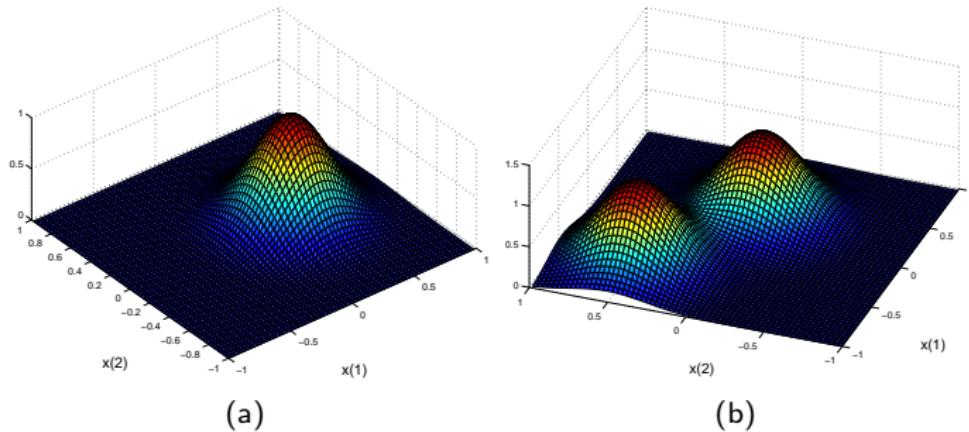
These basis functions are bump shaped, with the centre being given by  $\mathbf{m}^i$  and the width by  $\alpha$ .



# Radial basis functions

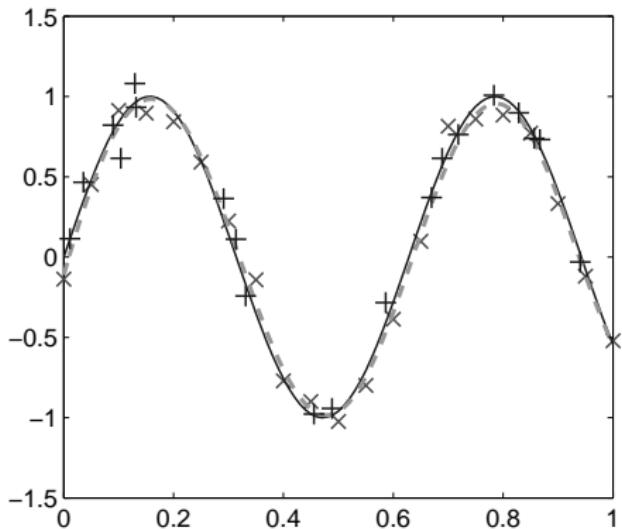


## RBFs with higher dimensional input



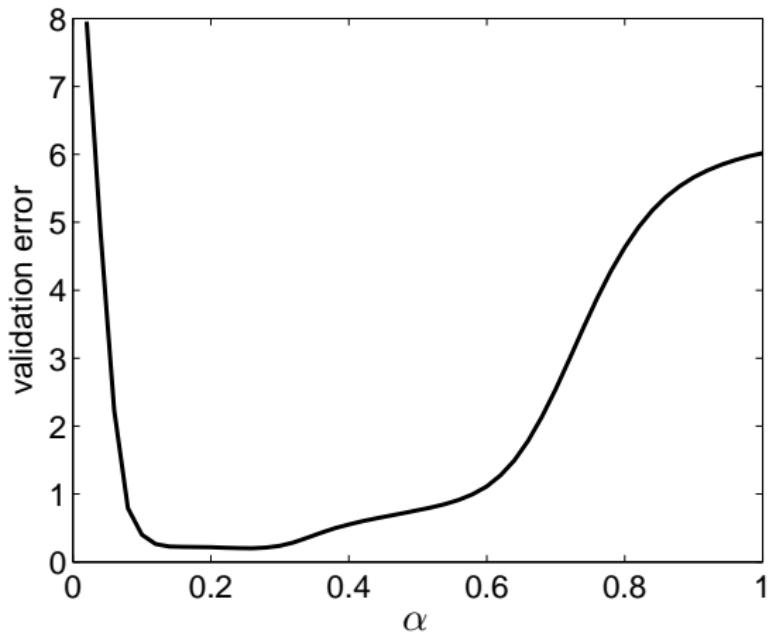
**Figure :** (a): The output of an RBF function  $\exp\left(-\frac{1}{2}\left(\mathbf{x} - \mathbf{m}^1\right)^2/\alpha^2\right)$ . Here  $\mathbf{m}^1 = (0, 0.3)^T$  and  $\alpha = 0.25$ . (b): The combined output for two RBFs with  $\mathbf{m}^1$  as above and  $\mathbf{m}^2 = (0.5, -0.5)^T$ .

## Fitting the RBF to data



The  $\times$  are the training points, and the  $+$  are the validation points. The solid line is the correct underlying function  $\sin(10x)$  which is corrupted with a small amount of additive noise to form the train data. The dashed line is the best predictor based on the validation set.

## Setting the RBF width $\alpha$



The validation error as a function of the basis function width for the validation data. Based on the validation error, the optimal setting of the basis function width parameter is  $\alpha = 0.25$ .

## A curse of dimensionality

- If the data has non-trivial behaviour over some input region, then we need to cover this region input space fairly densely with bump type functions.
- In the above case, we used 16 basis functions for a one dimensional input space.
- In 2 dimensions if we wish to cover each dimension to the same discretisation level, we would need  $16^2 = 256$  basis functions. Similarly, for 10 dimensions we would need  $16^{10} \approx 10^{12}$  functions. To fit such an LPM would require solving a linear system in more than  $10^{12}$  variables.
- This explosion in the number of basis functions with the input dimension is a ‘curse of dimensionality’.
- One way around this is to enable the centres of the RBF to move – this is similar to what happens in a neural network.
- Gaussian Processes are an alternative approach – see the BRML book.

## Summary

- Regression is about modelling inputs to (continuous) outputs.
- Linear Regression and linear parameter models are the most common and are very easy to train ●
- RBFs are also very common non-linear input-output mappings but suffer from the curse of dimensionality ●
- It is generally a better idea to use  $L_1$  (Lasso) regularisation than the more classical  $L_2$  (ridge regression) regularisation.
- Even for cases with less datapoints than numbers of parameters, provided the true underlying parameter vector is sparse, we can potentially correctly recover this using  $L_1$  regularisation ●
- Modern viewpoint is that we can do feature selection automatically – just throw in all features and use  $L_1$  regularisation to find those that are useful ●
- Most regression models can be used in time-series prediction ●

# Classification

David Barber

# Classification

- This is a form of supervised learning.
- We have a dataset of input-output examples  $(\mathbf{x}^n, c^n), n = 1, \dots, N$
- For example we might have an image, represented by a vector  $\mathbf{x}$  and a corresponding discrete labels:

$$\underbrace{\begin{matrix} 4 \\ | \\ 1 \end{matrix}}_{\mathbf{x}^1}, \quad c^1 = 4$$

$$\underbrace{\begin{matrix} 7 \\ | \\ 1 \end{matrix}}_{\mathbf{x}^2}, \quad c^2 = 7$$

⋮

- We want to learn a mapping from inputs  $\mathbf{x}$  to class label  $c$  such that when we get a new input  $\mathbf{x}$ , we will output the correct class label.

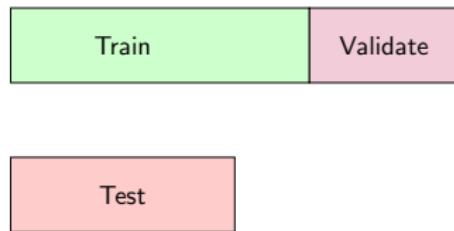
# Generalisation

## Basic Assumptions

- The data we have (all inputs and corresponding labels) is generated from the same unvarying mechanism.
- We will typically split our available data into three distinct parts: train data, validation data and test data.
- We want to find a model that will have good generalisation performance.

---

## Validation



Different models can be trained using the train data. The optimal model is determined by the empirical performance on the validation data. An independent measure of the generalisation performance of this optimal model is obtained by using a separate test set.

# Generative Models

# Do As Your Neighbour Does

- Each input vector  $\mathbf{x}$  has a corresponding class label,  $c^n \in \{1, \dots, C\}$ . Given a dataset of  $N$  train examples,  $\mathcal{D} = \{\mathbf{x}^n, c^n\}, n = 1, \dots, N$ , and a novel  $\mathbf{x}$ , we aim to return the correct class  $c(\mathbf{x})$ .
- For a classifier that works on unseen data, there must be some ‘smoothness’ in the underlying data generating process.
- If two points  $\mathbf{x}^1$  and  $\mathbf{x}^2$  that are very close can map to completely different class labels, there is no regularity in the process – the labels are essentially random.
- All classifiers assume some form of smoothness – the labels typically will not change much as we move a small distance in the input space

---

## A first classifier

- Nearest neighbour methods are a useful starting point since they readily encode basic smoothness intuitions and are easy to program.
- For novel  $\mathbf{x}$ , find the nearest input in the training set and use the class of this nearest input.

## Nearest Neighbour: Squared Euclidean Distance

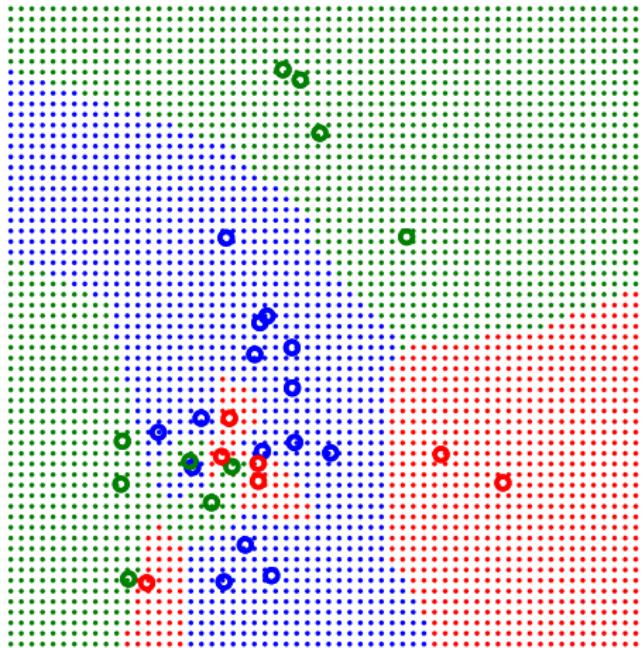
- For vectors  $\mathbf{x}$  and  $\mathbf{x}'$  representing two different datapoints, we measure 'nearness' using a dissimilarity function  $d(\mathbf{x}, \mathbf{x}')$ .
- A common dissimilarity is the squared Euclidean distance

$$d(\mathbf{x}, \mathbf{x}') = (\mathbf{x} - \mathbf{x}')^T (\mathbf{x} - \mathbf{x}') = \sum_{i=1}^D (x_i - x'_i)^2$$

which can be more conveniently written  $(\mathbf{x} - \mathbf{x}')^2$ .

- Based on the squared Euclidean distance, the decision boundary is determined by the perpendicular bisectors of the closest training points with different training labels.
- This partitions the input space into regions classified equally and is called a Voronoi tessellation.

## Nearest Neighbour: Voronoi tessellation



Here there are three classes, with training points given by the circles, along with their class. The dots indicate the class of the nearest training vector. The decision boundary bisects two datapoints belonging to different classes.

## Nearest Neighbour: taking care with distance

- If the length scales of the components of the vector  $\mathbf{x}$  vary greatly, the largest length scale will dominate the squared distance, with potentially useful class-specific information in other components of  $\mathbf{x}$  lost. ●
- For example, we might decide to represent a quantity in millimeters or in meters. This could heavily affect any distance calculations and completely change the classifier. ●
- The Mahalanobis distance

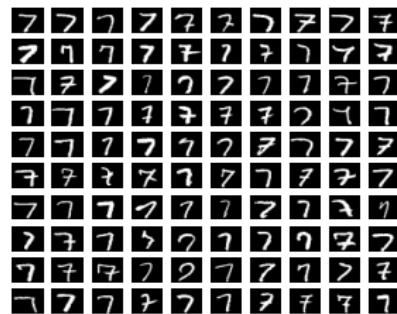
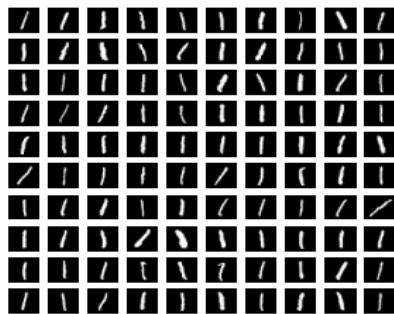
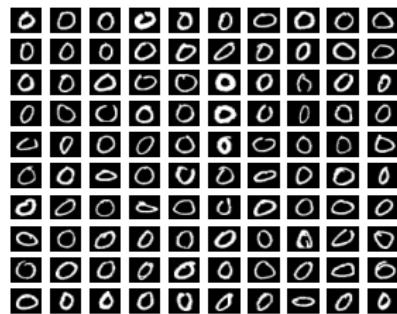
$$d(\mathbf{x}, \mathbf{x}') = (\mathbf{x} - \mathbf{x}')^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \mathbf{x}')$$

where  $\boldsymbol{\Sigma}$  is the covariance matrix of the inputs (from all classes) can overcome some of these problems since it effectively rescales the input vector components. ●

## Nearest Neighbour: Comments

- The whole dataset needs to be stored to make a classification since the novel point must be compared to all of the train points. This can be partially addressed by removing datapoints which have little or no effect on the decision boundary.
- Particularly for low dimensional data, finding the nearest neighbour can be speeded up by various techniques that exploit the geometry of the space (see later).
- Each distance calculation can be expensive if the datapoints are high dimensional. Principal Components Analysis, is one way to address this and replaces  $\mathbf{x}$  with a low dimensional projection  $\mathbf{p}$ . The Euclidean distance of two datapoints  $(\mathbf{x}^a - \mathbf{x}^b)^2$  is then approximately given by  $(\mathbf{p}^a - \mathbf{p}^b)^2$ . This is both faster to compute and can also improve classification accuracy since only the large scale characteristics of the data are retained in the PCA projections.
- It is not clear how to deal with missing data or incorporate prior beliefs and domain knowledge.

# Handwritten Digit Example



Some of the train examples of the digit zero, one and seven. There are 300 train examples of each of these three digit classes.

# KNN on handwritten digits (MNIST)

## One versus Zero

- Each digit contains  $28 \times 28 = 784$  pixels. The train data consists of 300 zeros, and 300 ones.
- To test the performance of the nearest neighbour method (based on Euclidean distance) we use an independent test set containing a further 600 digits.
- The nearest neighbour method, applied to this data, correctly predicts the class label of all 600 test points.
- The reason for the high success rate is that examples of zeros and ones are sufficiently different that they can be easily distinguished.

---

## One versus Seven

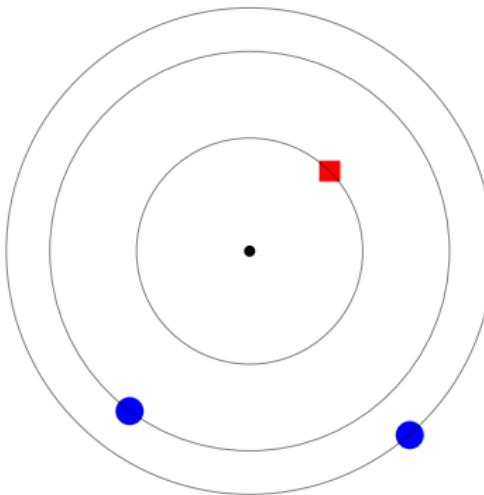
- This time, 18 errors are found using nearest neighbour classification – a 3% error rate for this two class problem.
- As an aside, the best methods classify real world digits (over all 10 classes) to an error of less than 1 percent – better than the performance of an ‘average’ human.

# $K$ -Nearest Neighbours

## Making Nearest Neighbours mode robust

- If your neighbour is simply mistaken (has an incorrect training class label), or is not a particularly representative example of his class, then these situations will typically result in an incorrect classification.
- By including more than the single nearest neighbour, we hope to make a more robust classifier with a smoother decision boundary (less swayed by single neighbour opinions).
- If we assume the Euclidean distance as the dissimilarity measure, the algorithm considers a hypersphere centred on the test point  $x$ . The radius of the hypersphere is increased until it contains exactly  $K$  train inputs.
- The class label  $c(x)$  is then given by the most numerous class within the hypersphere.

## *K*-Nearest Neighbours



In  $K$ -nearest neighbours, we centre a hypersphere around the point we wish to classify (here the central dot). The inner circle corresponds to the nearest neighbour, a square. However, using the 3 nearest neighbours, we find that there are two round-class neighbours and one square-class neighbour— and we would therefore classify the central point as round-class.

## Choosing $K$

- Whilst there is some sense in making  $K > 1$ , there is certainly little sense in making  $K = N$  ( $N$  being the number of training points).
- For  $K$  very large, all classifications will become the same – simply assign each novel  $\mathbf{x}$  to the most numerous class in the train data.
- This suggests that there is an optimal intermediate setting of  $K$  which gives the best generalisation performance.

---

### Using validation data

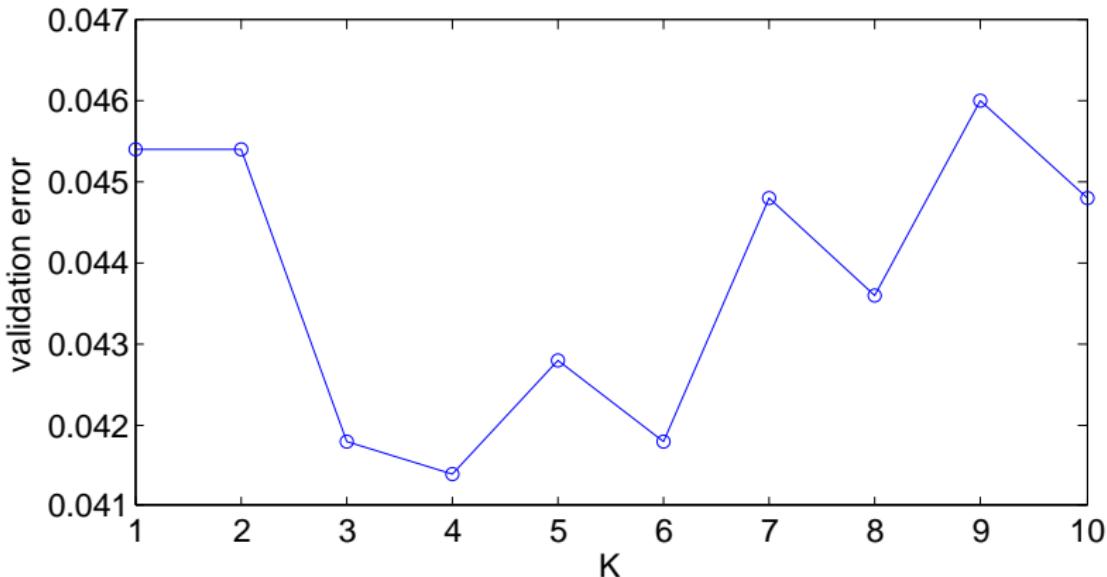
- Consider a validation set.
- For each  $K$  of interest, calculate the performance of the KNN classifier (whose class label is assigned on the basis of the train data).
- Choose  $K$  based on the best validation performance.

# KNN: Choosing $K$

MNIST Digits (all 10 classes)

There are 15,000 train datapoints and 5000 validation points.

`demoKNNlearnK.m`



$K = 4$  is the best choice, giving an error of around 4.1%.

## Probabilistic Interpretation of Nearest Neighbours

Consider the situation where we have data from two classes – class 0 and class 1. We make the following mixture model for data from class 0, placing a Gaussian on each datapoint:

$$p(\mathbf{x}|c=0) = \frac{1}{N_0} \sum_{n \in \text{class 0}} \mathcal{N}(\mathbf{x}|\mathbf{x}^n, \sigma^2 \mathbf{I})$$

where  $D$  is the dimension of a datapoint  $\mathbf{x}$  and  $N_0$  are the number of train points of class 0, and  $\sigma^2$  is the variance.

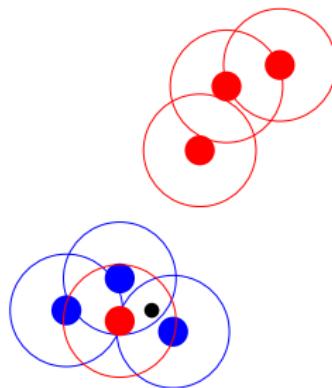
Similarly, for data from class 1:

$$p(\mathbf{x}|c=1) = \frac{1}{N_1} \sum_{n \in \text{class 1}} \mathcal{N}(\mathbf{x}|\mathbf{x}^n, \sigma^2 \mathbf{I})$$

To classify a new datapoint  $\mathbf{x}^*$ , we use Bayes' rule

$$p(c=0|\mathbf{x}^*) = \frac{p(\mathbf{x}^*|c=0)p(c=0)}{p(\mathbf{x}^*|c=0)p(c=0) + p(\mathbf{x}^*|c=1)p(c=1)}$$

## Probabilistic Interpretation



A probabilistic interpretation of nearest neighbours. For each class we use a mixture of Gaussians to model the data from that class  $p(\mathbf{x}|c)$ , placing at each training point an isotropic Gaussian of width  $\sigma^2$ . The width of each Gaussian is represented by the circle. In the limit  $\sigma^2 \rightarrow 0$  a novel point (small black dot) is assigned the class of its nearest neighbour. For finite  $\sigma^2 > 0$  the influence of non-nearest neighbours has an effect, resulting in a soft version of nearest neighbours.

## Maximum Likelihood

- The maximum likelihood setting of  $p(c = 0)$  is  $N_0/(N_0 + N_1)$ , and  $p(c = 1) = N_1/(N_0 + N_1)$ .
- An analogous expression holds for  $p(c = 1|\mathbf{x}^*)$ .
- To see which class is most likely we may use the ratio

$$\frac{p(c = 0|\mathbf{x}^*)}{p(c = 1|\mathbf{x}^*)} = \frac{p(\mathbf{x}^*|c = 0)p(c = 0)}{p(\mathbf{x}^*|c = 1)p(c = 1)}$$

If this ratio is greater than one, we classify  $\mathbf{x}^*$  as 0, otherwise 1.

## Limiting case

$$\frac{p(c=0|\mathbf{x}^*)}{p(c=1|\mathbf{x}^*)} = \frac{p(\mathbf{x}^*|c=0)p(c=0)}{p(\mathbf{x}^*|c=1)p(c=1)}$$

- If  $\sigma^2$  is very small, the numerator is dominated by that term for which datapoint  $\mathbf{x}^{n_0}$  in class 0 is closest to the point  $\mathbf{x}^*$ . Similarly, the denominator will be dominated by that datapoint  $\mathbf{x}^{n_1}$  in class 1 which is closest to  $\mathbf{x}^*$ .
- Taking the limit  $\sigma^2 \rightarrow 0$ , with certainty we classify  $\mathbf{x}^*$  as class 0 if  $\mathbf{x}^*$  is closer to  $\mathbf{x}^{n_0}$  than to  $\mathbf{x}^{n_1}$ .
- The nearest (single) neighbour method is therefore recovered as the limiting case of a probabilistic generative model.,

## Probabilistic Interpretation

- The motivation for  $K$  nearest neighbours is to produce a classification that is robust against unrepresentative single nearest neighbours.
- To ensure a similar kind of robustness in the probabilistic interpretation, we may use a finite value  $\sigma^2 > 0$ .
- This smooths the extreme probabilities of classification and means that more points (not just the nearest) will have an effective contribution.
- The extension to more than two classes is straightforward, requiring a class conditional generative model for each class.
- By using a richer generative model of the data we may go beyond the Parzen estimator approach.

## When your nearest neighbour is far away

- For a novel input  $x^*$  that is far from all training points, Nearest Neighbours, and its soft probabilistic variant will confidently classify  $x^*$  as belonging to the class of the nearest training point.
- This is arguably opposite to what we would like, namely that the classification should tend to the prior probabilities of the class based on the number of training data per class.
- A way to avoid this problem is, for each class, to include a fictitious large-variance mixture component at the mean of all the data, one for each class.
- For novel inputs close to the training data, this extra fictitious component will have no appreciable effect. However, as we move away from the high density regions of the training data, this additional fictitious component will dominate since it has larger variance than any of the other components.
- As the distance from  $x^*$  to each fictitious class point is the same, in the limit that  $x^*$  is far from the training data, the effect is that no class information from the position of  $x^*$  occurs.

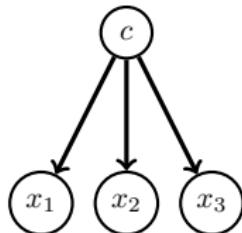
# Naive Bayes

We form a joint model of a  $D$ -dimensional attribute (input) vector  $\mathbf{x}$  and the corresponding class  $c$

$$p(\mathbf{x}, c) = p(c) \prod_{i=1}^D p(x_i|c)$$

Coupled with a suitable choice for each conditional distribution  $p(x_i|c)$ , we can then use Bayes' rule to form a classifier for a novel input vector  $\mathbf{x}^*$ :

$$p(c|\mathbf{x}^*) = \frac{p(\mathbf{x}^*|c)p(c)}{p(\mathbf{x}^*)} = \frac{p(\mathbf{x}^*|c)p(c)}{\sum_c p(\mathbf{x}^*|c)p(c)}$$



**Figure:** The central assumption is that given the class  $c$ , the attributes  $x_i$  are independent.

## Naive Bayes example

- Consider the following vector of binary attributes:

(shortbread, lager, whiskey, porridge, football)

$\mathbf{x} = (1, 0, 1, 1, 0)^T$  is a person that likes shortbread, does not like lager, drinks whiskey, eats porridge, and has not watched England play football.

- Together with each vector  $\mathbf{x}$ , there is a label  $nat$  describing the nationality of the person,  $\text{dom}(nat) = \{\text{scottish}, \text{english}\}$ .
- We wish to classify the vector  $\mathbf{x} = (1, 0, 1, 1, 0)^T$  as either scottish or english

---

### Training data

0	1	1	1	0	0
0	0	1	1	1	0
1	1	0	0	0	0
1	1	0	0	0	1
1	0	1	0	1	0

(a) English

1	1	1	1	1	1	1
0	1	1	1	1	0	0
0	0	1	0	0	1	1
1	0	1	1	1	1	0
1	1	0	0	1	0	0

(b) Scottish

## Naive Bayes example

$$p(\text{scottish}|\mathbf{x}) = \frac{p(\mathbf{x}|\text{scottish})p(\text{scottish})}{p(\mathbf{x}|\text{scottish})p(\text{scottish}) + p(\mathbf{x}|\text{english})p(\text{english})}$$

where  $p(\text{scottish}) = 7/13$  and  $p(\text{english}) = 6/13$ .

$$p(\mathbf{x}|nat) = p(x_1|nat)p(x_2|nat)p(x_3|nat)p(x_4|nat)p(x_5|nat)$$

$p(x_1 = 1 \text{english})$	$= 1/2$	$p(x_1 = 1 \text{scottish})$	$= 1$
$p(x_2 = 1 \text{english})$	$= 1/2$	$p(x_2 = 1 \text{scottish})$	$= 4/7$
$p(x_3 = 1 \text{english})$	$= 1/3$	$p(x_3 = 1 \text{scottish})$	$= 3/7$
$p(x_4 = 1 \text{english})$	$= 1/2$	$p(x_4 = 1 \text{scottish})$	$= 5/7$
$p(x_5 = 1 \text{english})$	$= 1/2$	$p(x_5 = 1 \text{scottish})$	$= 3/7$

For  $\mathbf{x} = (1, 0, 1, 1, 0)^T$ , we get

$$p(\text{scottish}|\mathbf{x}) = \frac{1 \times \frac{3}{7} \times \frac{3}{7} \times \frac{5}{7} \times \frac{4}{7} \times \frac{7}{13}}{1 \times \frac{3}{7} \times \frac{3}{7} \times \frac{5}{7} \times \frac{4}{7} \times \frac{7}{13} + \frac{1}{2} \times \frac{1}{2} \times \frac{1}{3} \times \frac{1}{2} \times \frac{1}{2} \times \frac{6}{13}} = 0.8076$$

## Naive Bayes

- Naive Bayes is very fast to train.
- For discrete attributes, we just count the data.
- In the case of low counts, the method can be overly confident. One approach to deal with this is to add a fixed number of pseudocounts to each variable attribute for each class.
- Naive Bayes deals easily with more than two classes.
- Missing data is easily dealt with.
- We can easily deal with attributes that take more than two states or are continuous.
- SpamBayes is a popular spam/ham junk mail classifier – lightening fast to train with low storage costs.

## Discriminative Models

## Parametric Classification

Parametric methods define a classifier directly as a function of the input  $\mathbf{x}$  and some (to be learned) parameters  $\theta$ .

---

Canonical example parametric two class method

$$c(\mathbf{x}; \theta) = \begin{cases} 1 & \text{if } \theta^T \mathbf{x} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

- This is called a perceptron and one of the earliest methods.
- It is called a linear classifier (it is a linear function of the parameters  $\theta$ ).
- Often consider extending  $\mathbf{x}$  to some non-linear feature vector with components  $\psi_i(\mathbf{x})$  and use  $\theta^T \psi(\mathbf{x})$  as the classifier.
- For example

$$\psi(x_1, x_2) = \begin{pmatrix} x_1 \\ x_1 x_2 \\ \sin(x_2) \\ x_2^3 \end{pmatrix}$$

## Loss functions

- For each input  $\mathbf{x}$  our predictor (which is a function  $c(\mathbf{x}; \boldsymbol{\theta})$  of this input and the parameter  $\boldsymbol{\theta}$ ) will produce a class label.
- We want this to match the train data label.
- If our predictor function has adjustable parameters, we will set them to minimise some loss criterion

$$\min_{\boldsymbol{\theta}} \sum_{n=1}^N L(c^n, c(\mathbf{x}^n; \boldsymbol{\theta}))$$

where  $L(c^{true}, c^{pred})$  is a loss function that specifies how bad it is if we get the class label incorrect.

---

### Zero-one loss

$$L(c^{true}, c^{pred}) = \begin{cases} 0 & \text{if } c^{pred} = c^{true} \\ 1 & \text{if } c^{pred} \neq c^{true} \end{cases}$$

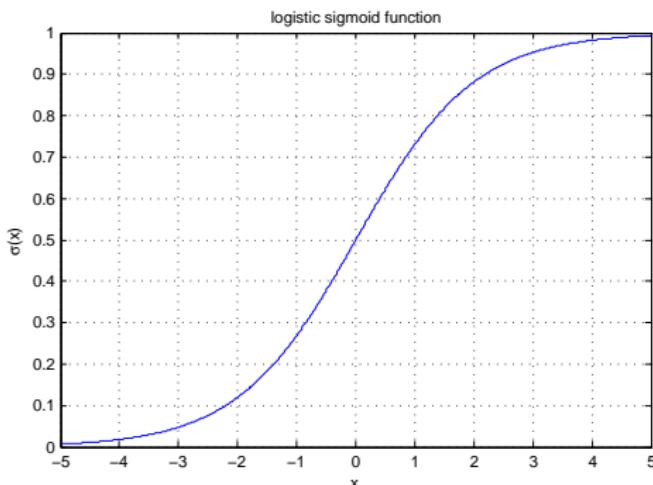
- This is in some sense an 'ideal' loss function.
- Error surface is very complex – difficult to train.

# Probabilistic Models for Binary classification, $c \in \{0, 1\}$

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

here the parameters are given by the ‘weight vector’  $\mathbf{w}$  ‘bias’  $b$  and the ‘logistic sigmoid’ is defined by

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

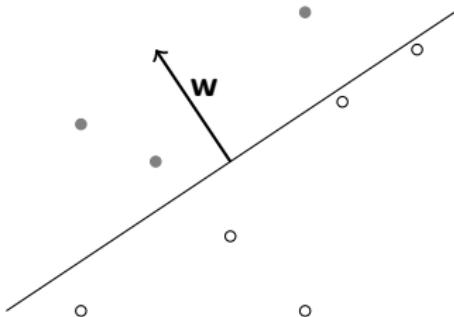


## The decision boundary

The decision boundary is defined as that set of  $\mathbf{x}$  for which  $p(c = 1|\mathbf{x}) = 0.5$ :

$$b + \mathbf{x}^T \mathbf{w} = 0$$

If  $\mathbf{x}$  is on the decision boundary, so is  $\mathbf{x}$  plus any vector perpendicular to  $\mathbf{w}$ . In  $D$  dimensions, the space of vectors that are perpendicular to  $\mathbf{w}$  occupy a  $D - 1$  dimensional hyperplane.



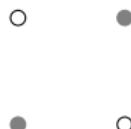
The decision boundary  $p(c = 1|\mathbf{x}) = 0.5$  (solid line). For two dimensional data, the decision boundary is a line. If all the training data for class 1 (filled circles) lie on one side of the line, and for class 0 (open circles) on the other, the data is said to be linearly separable. More generally,  $\mathbf{w}$  defines the normal to a hyperplane and data is linearly separable if data from each of the two classes lies on opposite sides of the hyperplane.

# Linear Separability

If all the data for class 1 lies on one side of a hyperplane, and for class 0 on the other, the data is said to be linearly separable.

---

## Non linearly separable example



The XOR problem with each component of the training inputs on the cube  $x_i \in \{+1, -1\}$ . This is not linearly separable.

---

## Making the data linearly separable

- We can map using a non-linear vector function  $\psi(\mathbf{x})$ .
- Eg

$$\psi(x_1, x_2) = \begin{pmatrix} (x_1 - 1)(x_2 - 1) \\ (x_1 + 1)(x_2 + 1) \end{pmatrix}, \quad \mathbf{w} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

- For the grey class data,  $\mathbf{w}^T \psi$  will be 4. For the other class it will be 0.
- In this case the  $\psi$  datapoints are linearly separable.
- Similarly we can map to a higher dimension so that the data becomes linearly separable.

# Maximum likelihood training

## IID assumption

Assuming the data is identically and independently drawn from the same mechanism, the likelihood of the data is

$$\prod_{n=1}^N p(c^n | \mathbf{x}^n, b, \mathbf{w}) = \prod_{n=1}^N p(c=1 | \mathbf{x}^n, b, \mathbf{w})^{c^n} (1 - p(c=1 | \mathbf{x}^n, b, \mathbf{w}))^{1-c^n}$$

where we have used the fact that  $c^n \in \{0, 1\}$ .

---

## Log likelihood function

For logistic regression this gives the log likelihood as

$$\mathcal{L}(\mathbf{w}, b) = \sum_{n=1}^N c^n \log \sigma(b + \mathbf{w}^\top \mathbf{x}^n) + (1 - c^n) \log (1 - \sigma(b + \mathbf{w}^\top \mathbf{x}^n))$$

---

## Optimisation

Maximise  $\mathcal{L}$  numerically with respect to the weights  $\mathbf{w}$  and bias  $b$ .

# Gradient ascent optimisation

One of the simplest methods is gradient ascent for which the gradient is given by

$$\nabla_{\mathbf{w}} \mathcal{L} = \sum_{n=1}^N (c^n - \sigma(\mathbf{w}^T \mathbf{x}^n + b)) \mathbf{x}^n$$

The gradient ascent procedure then corresponds to updating the weights using

$$\mathbf{w}^{new} = \mathbf{w} + \eta \nabla_{\mathbf{w}} \mathcal{L}$$

where the learning rate  $\eta$  is chosen small enough to ensure convergence. The application of the above rule will lead to a gradual increase in the log likelihood. The bias is updated similarly.

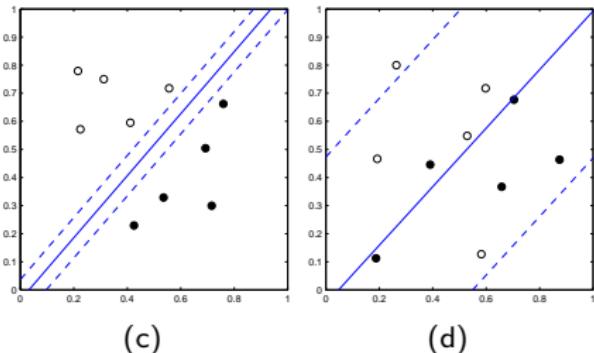
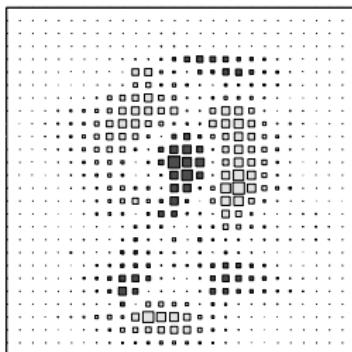


Figure: The decision boundary  $p(c=1|\mathbf{x}) = 0.5$  (solid line) and confidence boundaries  $p(c=1|\mathbf{x}) = 0.9$  and  $p(c=1|\mathbf{x}) = 0.1$  after 10000 iterations of batch gradient ascent with  $\eta = 0.1$ . **(a)**: Linearly separable data. **(b)**: Non-linearly separable data. Note how the confidence interval remains broad.

## Example: Classifying handwritten '1' versus '7'

- We apply logistic regression to the 600 handwritten digits in which there are 300 ones and 300 sevens in the data.
- Using gradient ascent training with a suitably chosen stopping criterion, the number of errors made on the 600 test points is 12, compared with 14 errors using Nearest Neighbour methods.



**Figure:** Logistic regression for classifying handwritten digits 1 and 7. Displayed is a Hinton diagram of the 784 learned weight vector  $w$ , plotted as a  $28 \times 28$  image for visual interpretation. Light squares are positive weights and an input  $x$  with a (positive) value in this component will tend to increase the probability that the input is classed as a 7. Similarly, inputs with positive contributions in the dark regions tend to increase the probability as being classed as a 1 digit. Note that the elements of each input  $x$  are either positive or zero.

# Geometry of the error surface

## Hessian

The Hessian of the log likelihood  $\mathcal{L}(\mathbf{w})$  is the matrix with elements

$$H_{ij} \equiv \frac{\partial^2 \mathcal{L}}{\partial w_i \partial w_j} = - \sum_n x_i^n x_j^n \sigma^n (1 - \sigma^n)$$

---

## Concavity

This is negative semidefinite since, for any  $\mathbf{z}$ ,

$$\sum_{ij} z_i H_{ij} z_j = - \sum_{i,j,n} z_i x_i^n z_j x_j^n \sigma^n (1 - \sigma^n) \leq - \sum_n \left( \sum_i z_i x_i^n \right)^2 \leq 0$$

This means that the error surface is concave (an upside down bowl) and batch gradient ascent converges to the optimal solution, provided the learning rate  $\eta$  is small enough.

## Beyond first order gradient ascent

- Gradient ascent is easy to implement but slow to converge.
- Since the surface has a single optimum, a Newton update

$$\mathbf{w}^{new} = \mathbf{w}^{old} + \eta \mathbf{H}^{-1} \mathbf{g}$$

where  $\mathbf{H}$  is the Hessian matrix as above and  $0 < \eta < 1$ , will typically converge much faster than gradient ascent.

- However, for large scale problems with  $\dim(\mathbf{w}) \gg 1$ , the inversion of the Hessian is computationally demanding and limited memory BFGS or conjugate gradient methods are more practical alternatives.

---

## Sparse data

- If only a small number of the elements of  $\mathbf{x}$  are non-zero, the data is called sparse.
- In this case we can implement conjugate gradient methods very efficiently (see notes).
- We can easily train logistic regression using millions of dimensions and numbers of training datapoints.

## Avoiding overconfident classification

- Provided the data is linearly separable the weights will continue to increase and the classifications will become extreme.
- This is undesirable since the resulting classifications will be over-confident.
- One way to prevent this is early stopping in which only a limited number of gradient updates are performed.
- An alternative method is to add a penalty term to the objective function

$$\mathcal{L}'(\mathbf{w}, b) = \mathcal{L}(\mathbf{w}, b) - \alpha \mathbf{w}^T \mathbf{w}.$$

The scalar constant  $\alpha > 0$  encourages smaller values of  $\mathbf{w}$  (remember that we wish to maximise the log likelihood). An appropriate value for  $\alpha$  can be determined using validation data.

- The objective  $\mathcal{L}'$  is still concave so that training is numerically easy.

## More than two classes

### Softmax regression

For more than two classes  $C > 2$  the logistic regression model is easily extendible using

$$p(c|x) = \frac{e^{\mathbf{w}_c^\top \phi(x)}}{\sum_d e^{\mathbf{w}_d^\top \phi(x)}}$$

where  $\phi$  is a defined vector function of  $x$  and we have a vector  $\mathbf{w}$  for each class.  
The function

$$f(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

is called the softmax function and is a smooth version of the max function.

---

### Training surface

One can show that the log likelihood is a jointly concave function of  $\mathbf{w}_1, \dots, \mathbf{w}_C$ .

# Support Vector Machines for binary classification

- SVMs are linear classifiers that encourage good generalisation performance.
  - SVMs do not fit comfortably within a probabilistic framework
- 

## Maximum margin linear classifier

In the SVM literature it is common to use  $+1$  and  $-1$  to denote the two classes.  
For a hyperplane defined by weight  $\mathbf{w}$  and bias  $b$ , the classifier is

$$\mathbf{w}^\top \mathbf{x} + b \begin{cases} \geq 0 & \text{class } +1 \\ < 0 & \text{class } -1 \end{cases}$$

To make the classifier robust we impose that the decision boundary should be separated from the data by some finite amount  $\epsilon^2$  (assuming in the first instance that the data is linearly separable):

$$\mathbf{w}^\top \mathbf{x} + b \begin{cases} \geq \epsilon^2 & \text{class } +1 \\ < -\epsilon^2 & \text{class } -1 \end{cases}$$

## SVM objective function

To classify the training labels correctly and maximise the margin, the optimisation problem is equivalent to:

$$\text{minimise } \frac{1}{2} \mathbf{w}^T \mathbf{w} \quad \text{subject to } y^n (\mathbf{w}^T \mathbf{x}^n + b) \geq 1, \quad n = 1, \dots, N$$

This is a quadratic programming problem.

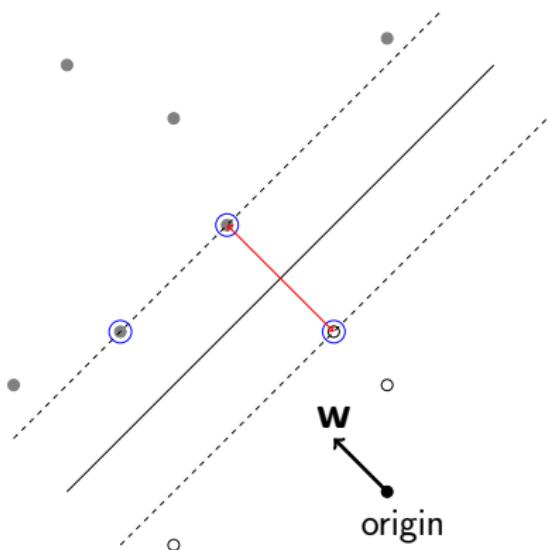


Figure: SVM classification of data from two classes (open circles and filled circles). The decision boundary  $\mathbf{w}^T \mathbf{x} + b = 0$  (solid line). For linearly separable data the maximum margin hyperplane is equidistant from the closest opposite class points. These support vectors are highlighted in blue and the margin in red. The distance of the decision boundary from the origin is  $-b/\sqrt{\mathbf{w}^T \mathbf{w}}$ , and the distance of a general point  $\mathbf{x}$  from the origin along the direction  $\mathbf{w}$  is  $\mathbf{x}^T \mathbf{w}/\sqrt{\mathbf{w}^T \mathbf{w}}$ .

## Non-linearly separable data

- To account for potentially mislabelled training points (or for data that is not linearly separable), we relax the exact classification constraint and use instead

$$y^n (\mathbf{w}^\top \mathbf{x}^n + b) \geq 1 - \xi^n$$

where the ‘slack variables’ are  $\xi^n \geq 0$ .

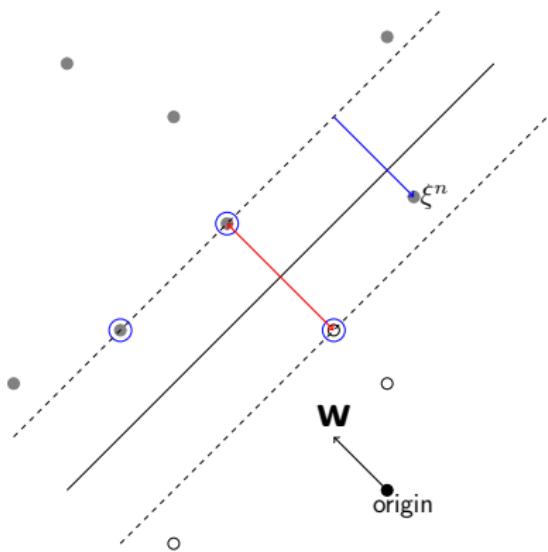
- Here each  $\xi^n$  measures how far  $\mathbf{x}^n$  is from the correct margin. For  $0 < \xi^n < 1$  datapoint  $\mathbf{x}^n$  is on the correct side of the decision boundary. However for  $\xi^n > 1$ , the datapoint is assigned the opposite class to its training label.
- Ideally we want to limit the size of these ‘violations’  $\xi^n$ .

The 2-norm soft-margin objective is

$$\text{minimise } \frac{1}{2} \mathbf{w}^\top \mathbf{w} + \frac{C}{2} \sum_n (\xi^n)^2 \quad \text{with } y^n (\mathbf{w}^\top \mathbf{x}^n + b) \geq 1 - \xi^n, \quad n = 1, \dots, N$$

where  $C$  controls the number of mislabellings of the data. The constant  $C$  needs to be determined empirically using a validation set.

## 2-Norm soft-margin



Slack margin. The term  $\xi^n$  measures how far a variable is from the correct side of the margin for its class. If  $\xi^n > 1$  then the point will be misclassified and treated as an outlier.

## Motivating Kernels

Consider a general objective of the form

$$E(\mathbf{w}) = \frac{1}{2} \mathbf{w}^T \mathbf{w} - \sum_n f(\mathbf{w}^T \mathbf{x}_n)$$

This has a minimum when  $\mathbf{w}$  lies in the span of the data  $\mathbf{x}_1, \dots, \mathbf{x}_N$

$$\mathbf{w} = \sum_n f'(\mathbf{w}^T \mathbf{x}_n) \mathbf{x}_n$$

We can therefore assume  $\mathbf{w} = \sum_n \alpha_n \mathbf{x}_n$  and form an equivalent objective

$$E(\boldsymbol{\alpha}) = \frac{1}{2} \sum_{m,n} \alpha_n \alpha_m \mathbf{x}_n^T \mathbf{x}_m - \sum_n f\left(\sum_m \alpha_m \mathbf{x}_m^T \mathbf{x}_n\right)$$

- This explains why  $E(\mathbf{w})$  can be expressed as only a function of the scalar product between datapoints. An alternative is therefore to define the scalar product via a kernel function.
- The ‘kernel’ trick applies to any objective of the above form. In addition to the SVM, one can ‘kernelise’ linear regression, logistic regression, etc.

# Using Kernels

- One can re-express the objective so that it depends on the inputs  $\mathbf{x}^n$  only via the scalar product  $(\mathbf{x}^n)^\top \mathbf{x}^n$ . If we map  $\mathbf{x}$  to a vector function of  $\mathbf{x}$ , then we can write

$$K(\mathbf{x}^n, \mathbf{x}^m) = \phi(\mathbf{x}^n)^\top \phi(\mathbf{x}^m)$$

- This means that we can use any positive semidefinite kernel  $K$  and make a non-linear classifier.

---

## Kernels

- A kernel (or covariance function)  $k$  is a special function such that for a collection of vectors  $\mathbf{x}^1, \dots, \mathbf{x}^N$  the matrix  $K$  with elements

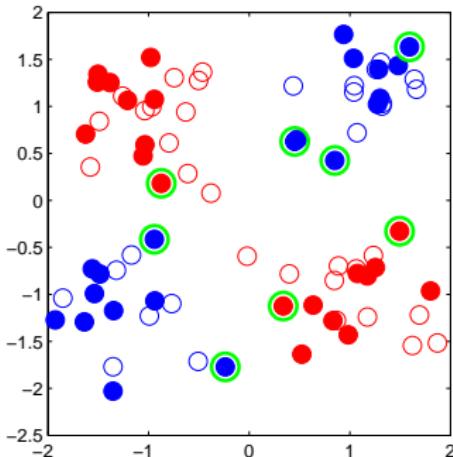
$$K_{mn} = k(\mathbf{x}^m, \mathbf{x}^n)$$

is positive semidefinite.

- The most well known is the ‘Gaussian’ or ‘squared exponential’

$$k(\mathbf{x}^m, \mathbf{x}^n) = e^{-(\mathbf{x}^m - \mathbf{x}^n)^2}$$

## SVM demo with a squared exponential kernel



- The solid red and solid blue circles represent data from different classes.
- The support vectors are highlighted in green.
- For the unfilled test points, the class assigned to them by the SVM is given by the colour.
- The computational complexity is  $O(N^3)$  where  $N$  is the number of training points.
- `demoSVM.m`

## Decision Trees

Consider data with the following attributes:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} \text{Salary} \\ \text{Permanent Job} \\ \text{Criminal Conviction} \end{pmatrix}$$

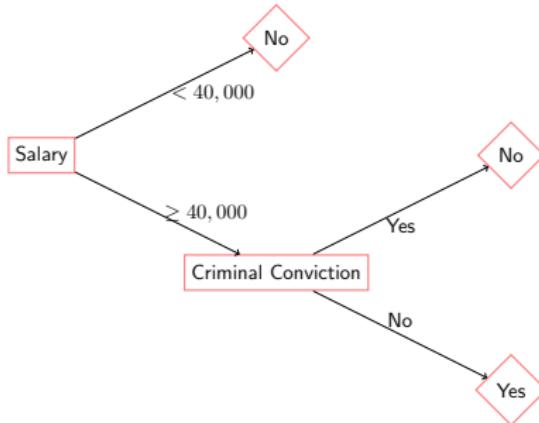
and that we have a training dataset in which bank managers have offered loans to

$$\begin{pmatrix} 100,000 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 50,000 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 75,000 \\ 1 \\ 0 \end{pmatrix}$$

and refused loans to

$$\begin{pmatrix} 50,000 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 30,000 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 10,000 \\ 1 \\ 0 \end{pmatrix}$$

# Decision Trees



- The decision tree is learned such that the data associated with each node is as 'pure' (contains the same class labels) as possible.
- At each node, we consider all the variables and choose that one which splits the data associated to that node into as pure children as possible.
- Fast to train and interpretable ●
- Somewhat limited — decisions typically based on simple single variable splits ●

## Ensemble Predictors

- Idea is to combine the predictions of several classifiers (works also for regression).
  - Many techniques based on different ways to construct predictors and also different ways to combine them.
  - Bagging is one of the simplest and most popular.
- 

### Bagging

- Bagging is a general ‘ensemble’ method.
- From our original input-output dataset, we generate  $B$  new datasets.
- Each dataset is generated by sampling (with replacement) data from the original dataset.
- For each new dataset we then train a model.
- The predictions from these  $B$  models are then combined, either by averaging or taking the majority prediction.

# Random Forest

- We generate a set of decision trees and take the majority prediction of the trees.
  - Slightly different approaches to generating the trees.
  - Very popular in prediction competitions and often does surprisingly well.
- 

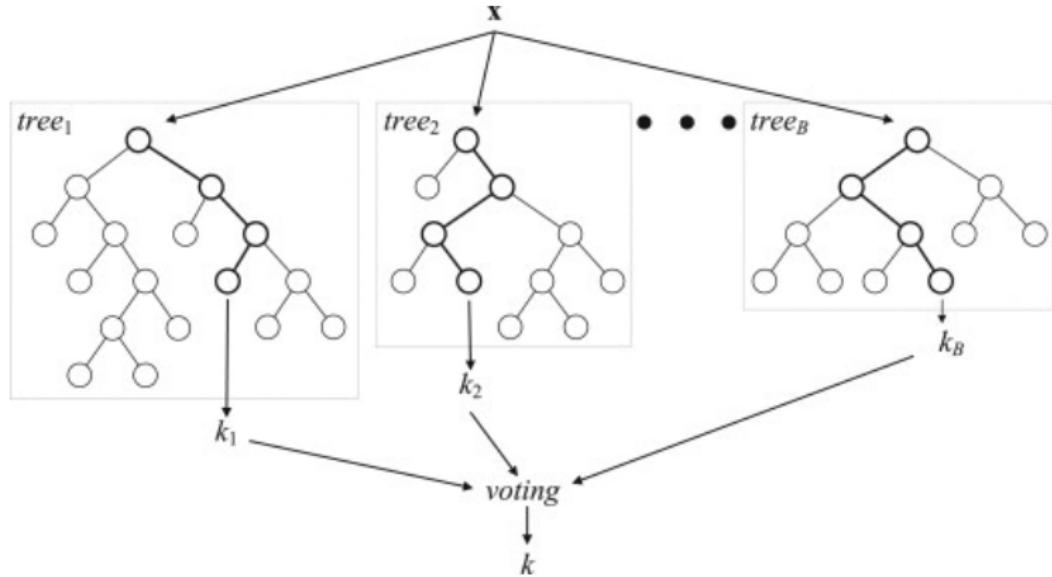
## Ho's approach

- For each tree decide on a fixed random subset of the variables on which to form the tree.
  - Usually take around  $\sqrt{D}$  number of variables, where  $D$  is the total number of variables in the input vector.
- 

## Breiman's approach

- This is bagging applied to decision trees.
- We form the dataset for each tree using sampling with replacement.
- At each node in the tree we randomly select only a small number of the dimensions of the variable  $x$  on which to find the best split.

# Random Forest



# Summary

## K-nearest neighbours

- Simple and intuitive ●
  - Slow to apply – typically scales  $O(N)$  ●
  - Heavily dependent on sensible distance function ●
  - Can't easily deal with missing data ●
- 

## Logistic Regression

- Simple and fast to train ●
- Fast to apply – typically scales  $O(D)$  ●
- Objective function is concave ●
- Can increase complexity by using kernels or non-linear mappings ●
- Can easily scale up to huge sparse data ●
- Can't easily deal with missing data ●

# Summary

## Support Vector Machine

- Has good generalisation performance ●
- Quite fast to apply ●
- Objective function is concave ●
- Objective function is not differentiable, with  $O(N^3)$  cost – does not scale well ●
- Can't easily deal with missing data ●

---

## Decision Tree

- Fast to train ●
- Highly interpretable ●
- Quite a weak classifier – decisions don't take simultaneous attributes into account.
- Random Forest extension has good empirical performance ●
- RF is less interpretable than the original DT ●
- Can't easily deal with missing data ●

# Summary

## Naive Bayes

- Fast to train and apply ●
- Can easily deal with missing data ●
- Classifier is quite weak ●

# State of the art

## Generative Approaches

- For  $p(\mathbf{x}|c, \theta)$  we can use more powerful models than Naive Bayes.
- For example one can use class conditional (correlated) distributions such as the multivariate Gaussian.

## Deep learning

- Neural Nets can be used to model for example the output probability for a binary classifier:

$$p(c|\mathbf{x}, \mathcal{W}) \propto e^{f_c(\mathbf{x}, \mathcal{W})}$$

where  $f_c(\mathbf{x}, \mathcal{W})$  is the scalar output of a network for class  $c$  and  $\mathcal{W}$  is the weights of the network.

- One can then use maximum likelihood to train the parameters  $\mathcal{W}$  of this discriminative classifier.
- These can be very powerful classifiers, but are hard to train.

# Optimisation<sup>1</sup>

David Barber

---

<sup>1</sup> These slides accompany the book *Bayesian Reasoning and Machine Learning*. The book and demos can be downloaded from [www.cs.ucl.ac.uk/staff/D.Barber\(brml](http://www.cs.ucl.ac.uk/staff/D.Barber(brml)). Feedback and corrections are also available on the site. Feel free to adapt these slides for your own purposes, but please include a link to the above website.

# The need for optimisation

Machine learning often requires fitting a model to data. Often this means finding the parameters  $\theta$  of the model that ‘best’ fit the data.

---

## Regression

For example, for regression based on training data  $(\mathbf{x}^n, y^n)$  we might have a model  $y(x|\theta)$  and wish to set  $\theta$  by minimising

$$E(\theta) = \sum_n (y^n - y(\mathbf{x}^n|\theta))^2$$

---

## Complexity

In all but very simple cases, it is extremely difficult to find an algorithm that will guarantee to find the optimal  $\theta$ .

## A simple case: Linear regression

For example for a linear predictor

$$y(\mathbf{x}|\boldsymbol{\theta}) \equiv \mathbf{x}^\top \boldsymbol{\theta}$$

$$E(\boldsymbol{\theta}) = \sum_n \left( y^n - \boldsymbol{\theta}^\top \mathbf{x}^n \right)^2$$

The optimum is given when the gradient wrt  $\boldsymbol{\theta}$  is zero:

$$\frac{\partial E}{\partial \theta_i} = 2 \sum_n \left( y^n - \boldsymbol{\theta}^\top \mathbf{x}^n \right) x_i^n = 0$$

$$\underbrace{\sum_n y^n x_i^n}_{b_i} = \sum_j \underbrace{\sum_n x_i^n x_j^n}_{X_{ij}} \theta_j$$

Hence, in matrix form, this is

$$\mathbf{b} = \mathbf{X}\boldsymbol{\theta}, \rightarrow \boldsymbol{\theta} = \mathbf{X}^{-1}\mathbf{b}$$

which is a simple linear system that can be solved in  $O\left((\dim \boldsymbol{\theta})^3\right)$  time.

## Quadratic functions

A class of simple functions to optimise is, for symmetric  $\mathbf{A}$ :

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{x}^T \mathbf{b}$$

This has a unique minimum if and only if  $\mathbf{A}$  is positive definite. In this case, at the optimum

$$\nabla f = \mathbf{A} \mathbf{x} - \mathbf{b} = \mathbf{0} \rightarrow \mathbf{x} = \mathbf{A}^{-1} \mathbf{b}$$

For  $\mathbf{x} + \boldsymbol{\delta}$ , the new value of the function is

$$f(\mathbf{x} + \boldsymbol{\delta}) = f(\mathbf{x}) + \underbrace{\boldsymbol{\delta}^T \nabla f}_{=0} + \frac{1}{2} \underbrace{\boldsymbol{\delta}^T \mathbf{A} \boldsymbol{\delta}}_{\geq 0}$$

Hence  $\mathbf{A}^{-1} \mathbf{b}$  is a minimum.

## Gradient Descent and First Order Methods

## Gradient Descent

We wish to find  $\mathbf{x}$  that minimises  $f(\mathbf{x})$ . For general  $f$  there is no closed-form solution to this problem and we typically resort to iterative methods.

For  $\mathbf{x}_{k+1} \approx \mathbf{x}_k$ ,

$$f(\mathbf{x}_{k+1}) \approx f(\mathbf{x}_k) + (\mathbf{x}_{k+1} - \mathbf{x}_k)^T \nabla f(\mathbf{x}_k)$$

setting

$$\mathbf{x}_{k+1} - \mathbf{x}_k = -\epsilon \nabla f(\mathbf{x}_k)$$

gives

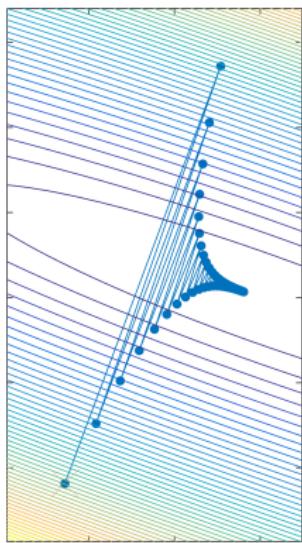
$$f(\mathbf{x}_{k+1}) \approx f(\mathbf{x}_k) - \epsilon |\nabla f(\mathbf{x}_k)|^2$$

Hence, for a small  $\epsilon$ , the algorithm

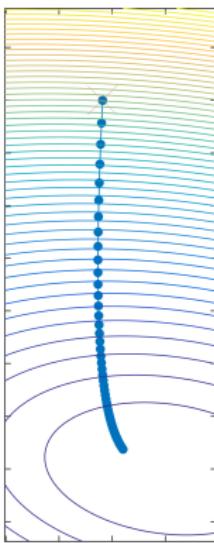
$$\mathbf{x}_{k+1} = \mathbf{x}_k - \epsilon \nabla f(\mathbf{x}_k)$$

decreases  $f$ . We iterate until convergence.

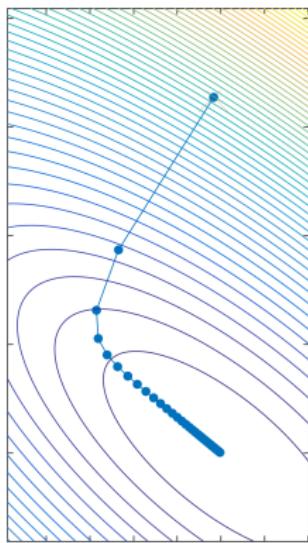
# Gradient Descent



(a) Learning rate too large



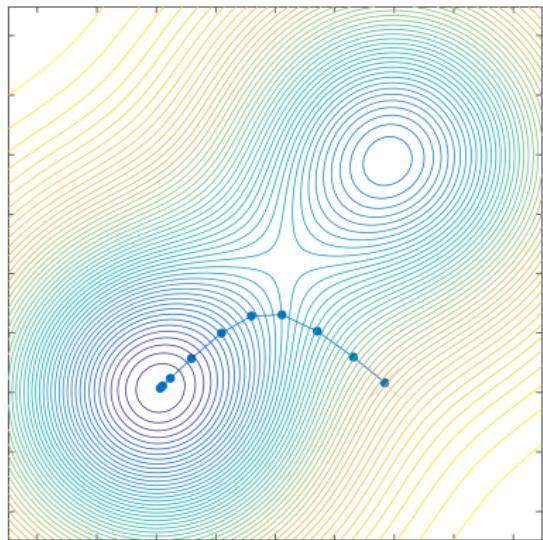
(b) Learning rate too small



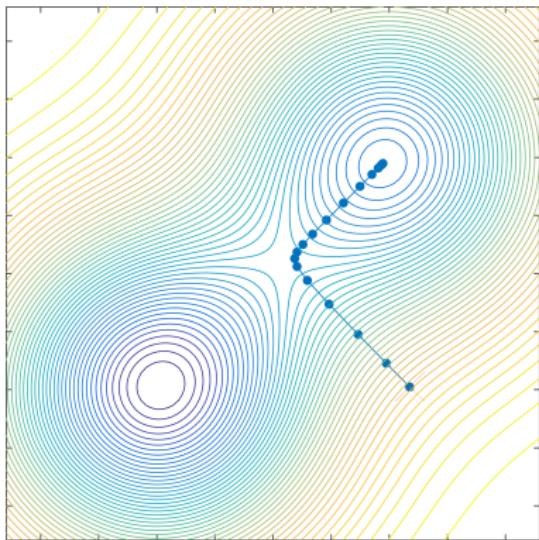
(c) Learning rate OK

Note the difference between converging to the minimal function value, and the optimum parameters; we might have almost converged to the minimal value but still be a long way from the optimum parameters. Common to consider adapting the learning rate. This is usually determined by experimenting with different values or learning schedules.

# Gradient Descent



(d) Found global minimum



(e) Found local minimum

Depending on the initial point, even for the same function and learning rate, we can converge to different solutions.

## Generalised gradient update

- Consider an update of the form using a ‘curvature matrix’  $\mathbf{C}$ ,

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \epsilon \mathbf{C}^{-1} \mathbf{g}$$

where  $\mathbf{C}$  is symmetric and positive definite,  $\epsilon > 0$ .

$$f(\mathbf{x}_{t+1}) \approx f(\mathbf{x}_t) - \epsilon \mathbf{g}^T \mathbf{C}^{-1} \mathbf{g} + \frac{\epsilon^2}{2} \mathbf{g}^T \mathbf{C}^{-1} \mathbf{H} \mathbf{C}^{-1} \mathbf{g}$$

- The first term  $-\epsilon \mathbf{g}^T \mathbf{C}^{-1} \mathbf{g}$  reduces the function value (the inverse of a positive definite matrix is positive definite). Provided  $\epsilon$  is small, the second term will be small and not affect the function reduction.
- Up to second order, we would get a reduction in the function value provided

$$\epsilon < \frac{2\mathbf{g}^T \mathbf{C}^{-1} \mathbf{g}}{\mathbf{g}^T \mathbf{C}^{-1} \mathbf{H} \mathbf{C}^{-1} \mathbf{g}}$$

- Will discuss later good ways to find suitable ‘curvature’ matrices.

# Convergence Rate for Convex Functions

Let's assume:

**Convexity**  $f$  is convex and finite over the path  $\mathbf{x}_1, \dots, \mathbf{x}_T$  that the gradient descent algorithm will take.

**Finite Solution** The optimum  $\mathbf{x}^*$  exists and is finite.

**$\nabla f$  Lipschitz continuous** The gradient is Lipschitz with constant  $L$ . We further assume  $f$  is twice differentiable. This means

$$\mathbf{H}(\mathbf{x}) \preceq L\mathbf{I}$$

so that all the eigenvalues of the Hessian are less than or equal to  $L$ .

**Learning Rate** At each step, the learning rate  $\epsilon \leq 1/L$ .

---

Convergence:

$$f(\mathbf{x}_T) - f(\mathbf{x}^*) \leq \frac{1}{2\epsilon T} (\mathbf{x}_1 - \mathbf{x}^*)^2$$

This means that the error goes down as  $O(1/T)$  where  $T$  is the number of iterations in the gradient descent algorithm.

## Proof (1)

Note that I'm using  $t$  here instead of  $k$  for the iteration index.

- $\mathbf{x}_{t+1} = \mathbf{x}_t - \epsilon \nabla f(\mathbf{x}_t)$
- Taylor's theorem (with residual term) says

$$f(\mathbf{y}) = f(\mathbf{x}) + (\mathbf{y} - \mathbf{x})^\top \nabla f(\mathbf{x}) + \frac{1}{2}(\mathbf{y} - \mathbf{x})^\top \mathbf{H}(\mathbf{z})(\mathbf{y} - \mathbf{x})$$

for some  $\mathbf{z}$  between  $\mathbf{x}$  and  $\mathbf{y}$ . Since  $\mathbf{H} \preceq L\mathbf{I}$  then (writing  $\mathbf{g}(\mathbf{x}) = \nabla f(\mathbf{x})$ )

$$f(\mathbf{x}_{t+1}) \leq f(\mathbf{x}_t) - \epsilon \mathbf{g}^2(\mathbf{x}_t) + \frac{L\epsilon^2}{2} \mathbf{g}^2(\mathbf{x}_t) = f(\mathbf{x}_t) - \epsilon \left(1 - \frac{L\epsilon}{2}\right) \mathbf{g}^2(\mathbf{x}_t)$$

If  $\epsilon \leq 2/L$ , then  $f(\mathbf{x}_{t+1}) \leq f(\mathbf{x}_t)$ . Optimising over  $\epsilon$  the biggest decrease is obtained using a learning rate  $\epsilon = 1/L$ . We assume throughout that  $\epsilon \leq 1/L$ , for which

$$f(\mathbf{x}_{t+1}) \leq f(\mathbf{x}_t) - \frac{\epsilon}{2} \mathbf{g}^2(\mathbf{x}_t)$$

- Hence, provided  $\epsilon$  is small enough, we will decrease the error with each update.

## Proof (2)

- Since  $f$  is convex,

$$f(\mathbf{y}) \geq f(\mathbf{x}) + (\mathbf{y} - \mathbf{x})^T \nabla f(\mathbf{x})$$

Which means

$$f(\mathbf{x}_t) \leq f(\mathbf{x}^*) + (\mathbf{x}_t - \mathbf{x}^*)^T \mathbf{g}(\mathbf{x}_t)$$

- Hence

$$f(\mathbf{x}_{t+1}) \leq f(\mathbf{x}_t) - \frac{\epsilon}{2} \mathbf{g}^2(\mathbf{x}_t) \leq f(\mathbf{x}^*) + (\mathbf{x}_t - \mathbf{x}^*)^T \mathbf{g}(\mathbf{x}_t) - \frac{\epsilon}{2} \mathbf{g}^2(\mathbf{x}_t)$$

- Write  $\mathbf{d} = \mathbf{x}_t - \mathbf{x}^*$ . Then

$$\mathbf{d}^T \mathbf{g} - \frac{\epsilon}{2} \mathbf{g}^2 = -\frac{1}{2\epsilon} \left[ (\epsilon \mathbf{g} - \mathbf{d})^2 - \mathbf{d}^2 \right]$$

- Hence, using  $\mathbf{d} - \epsilon \mathbf{g} = \mathbf{x}_t - \epsilon \mathbf{g} - \mathbf{x}^* = \mathbf{x}_{t+1} - \mathbf{x}^*$

$$f(\mathbf{x}_{t+1}) \leq f(\mathbf{x}^*) + \frac{1}{2\epsilon} \left[ (\mathbf{x}_t - \mathbf{x}^*)^2 - (\mathbf{x}_{t+1} - \mathbf{x}^*)^2 \right]$$

## Proof (3)

- Rearranging the above,

$$f(\mathbf{x}_{t+1}) - f(\mathbf{x}^*) \leq \frac{1}{2\epsilon} \left[ (\mathbf{x}_t - \mathbf{x}^*)^2 - (\mathbf{x}_{t+1} - \mathbf{x}^*)^2 \right]$$

$$f(\mathbf{x}_{t+2}) - f(\mathbf{x}^*) \leq \frac{1}{2\epsilon} \left[ (\mathbf{x}_{t+1} - \mathbf{x}^*)^2 - (\mathbf{x}_{t+2} - \mathbf{x}^*)^2 \right]$$

- Hence the sum ‘telescopes’ with terms cancelling:

$$\begin{aligned} \sum_{t=1}^T (f(\mathbf{x}_{t+1}) - f(\mathbf{x}^*)) &\leq \frac{1}{2\epsilon} \left[ (\mathbf{x}_1 - \mathbf{x}^*)^2 - (\mathbf{x}_{T+1} - \mathbf{x}^*)^2 \right] \\ &\leq \frac{1}{2\epsilon} (\mathbf{x}_1 - \mathbf{x}^*)^2 \end{aligned}$$

Since  $f(\mathbf{x}_T) \leq f(\mathbf{x}_t)$ ,

$$\begin{aligned} \sum_{t=1}^T (f(\mathbf{x}_T) - f(\mathbf{x}^*)) &\leq \sum_{t=1}^T (f(\mathbf{x}_t) - f(\mathbf{x}^*)) \\ T (f(\mathbf{x}_T) - f(\mathbf{x}^*)) &\leq \frac{1}{2\epsilon} (\mathbf{x}_1 - \mathbf{x}^*)^2 \quad \blacksquare \end{aligned}$$

## Momentum

One simple idea to limit the zig-zag behaviour is to make an update in the average direction of the previous updates.

---

### Moving average

Consider a set of numbers  $x_1, \dots, x_t$ . Then the average  $a_t$  is given by

$$a_t = \frac{1}{t} \sum_{\tau=1}^t x_\tau = \frac{1}{t} (x_t + (t-1)a_{t-1}) = \epsilon_t x_t + \mu_t a_{t-1}$$

for suitably chosen  $\epsilon_t$  and  $0 \leq \mu_t \leq 1$ . If  $\mu_t$  is small then the more recent  $x$  contribute more strongly to the moving average.

---

## Momentum

Idea is to use a form of moving average to the updates:

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

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \tilde{\mathbf{g}}_{k+1}$$

Hence, instead of using the update  $-\epsilon \mathbf{g}_k(\mathbf{x}_k)$ , we use the moving average of the update  $\tilde{\mathbf{g}}_{k+1}$  to form the new update.

## Momentum

- Momentum can increase the speed of convergence since, for smooth objectives, as we get close to the minimum the gradient decreases and standard gradient descent starts to slow down.
- If the learning rate is too large, standard gradient descent may oscillate, but momentum may reduce oscillations by going in the average direction.
- However, the momentum parameter  $\mu$  may need to be reduced with the iteration count to ensure convergence.
- Particularly useful when the gradient is noisy. By averaging over previous gradients, the noise ‘averages’ out and the moving average direction can be much less noisy.
- Momentum is also useful to avoid saddles (a point where the gradient is zero, but the objective function is not a minimum, such as the function  $x^3$  at the origin) since typically the momentum will carry you over the saddle.

# Nesterov's Accelerated Gradient

Nesterov

- This looks similar to momentum but has a slightly different update

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

That is, we use the gradient of the point we will move to, rather than the current point.

- Need to choose a schedule for  $\mu_k$ . Nesterov suggests

$$\mu_k = 1 - 3/(k + 5)$$

- For convex functions NAG has rate of convergence to the optimum

$$f(\mathbf{x}_k) - f^* \leq \frac{c}{k^2}$$

for some constant  $c$ , compared to  $1/k$  convergence for gradient descent.

## 'Understanding' Nesterov's Accelerated Gradient

Let's imagine that we've arrived at our current parameter  $\mathbf{x}_k$  based on an update

$$\mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{v}_{k-1} \tag{1}$$

where  $\mathbf{v}_{k-1}$  is the update. We now (retrospectively) ask: 'What would have been a better update than the one we actually made?'. Well, using the update we arrived at a function value

$$f(\mathbf{x}_k) = f(\mathbf{x}_{k-1} + \mathbf{v}_{k-1})$$

We could have got to a better value by changing  $\mathbf{v}_{k-1}$  to

$$\mathbf{v}_k = \mathbf{v}_{k-1} - \epsilon \frac{\partial}{\partial \mathbf{v}} f(\mathbf{x}_k) = \mathbf{v}_{k-1} - \epsilon \frac{\partial}{\partial \mathbf{x}} f(\mathbf{x}_{k-1} + \mathbf{v}_{k-1})$$

Hence, we define

$$\mathbf{v}_k = \mathbf{v}_{k-1} - \epsilon \mathbf{g}(\mathbf{x}_{k-1} + \mathbf{v}_{k-1})$$

and make the update (note the difference with (1))

$$\mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{v}_k$$

## 'Understanding' Nesterov's Accelerated Gradient

- The basic update

$$\mathbf{v}_k = \mathbf{v}_{k-1} - \epsilon \frac{\partial}{\partial \mathbf{x}} f(\mathbf{x}_{k-1} + \mathbf{v}_{k-1})$$

essentially is a form of momentum that pushes  $\mathbf{x}$  along the direction it was previously going.

- Even when we reach a point where the gradient is zero, then  $\mathbf{v}_k$  will be the same as  $\mathbf{v}_{k-1}$ .
- Without modification, this vanilla algorithm will diverge.
- We therefore introduce a term to dampen oscillations and ensure convergence. One can show that this gives the standard Nesterov update:

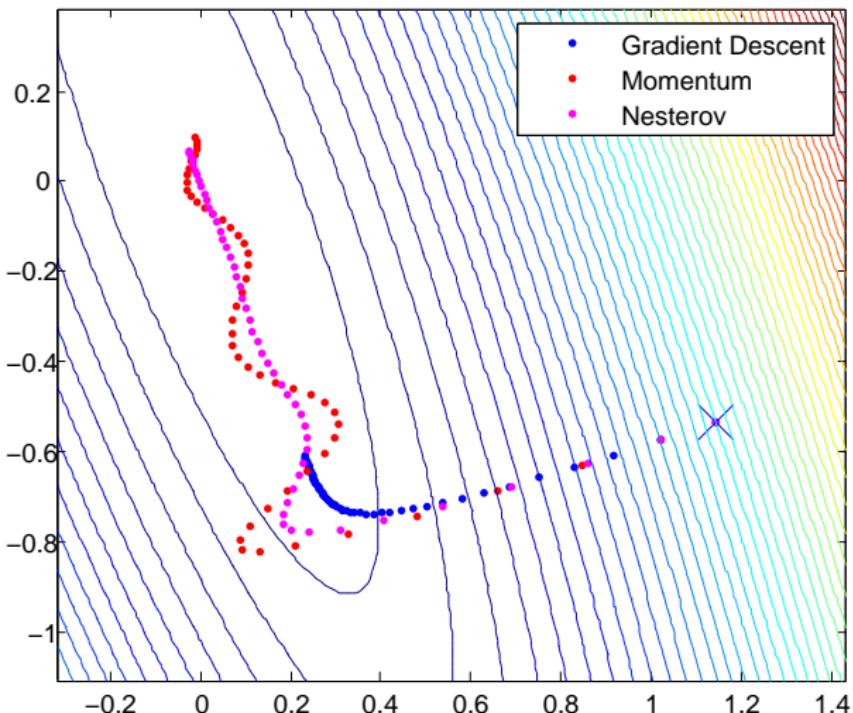
$$\mathbf{v}_k = \mu_{k-1} \mathbf{v}_{k-1} - \epsilon \frac{\partial}{\partial \mathbf{x}} f(\mathbf{x}_{k-1} + \mu_{k-1} \mathbf{v}_{k-1})$$

$$\mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{v}_k$$

Since  $\mu < 1$ ,  $\mathbf{v}$  will quickly converge to zero around the minimum.

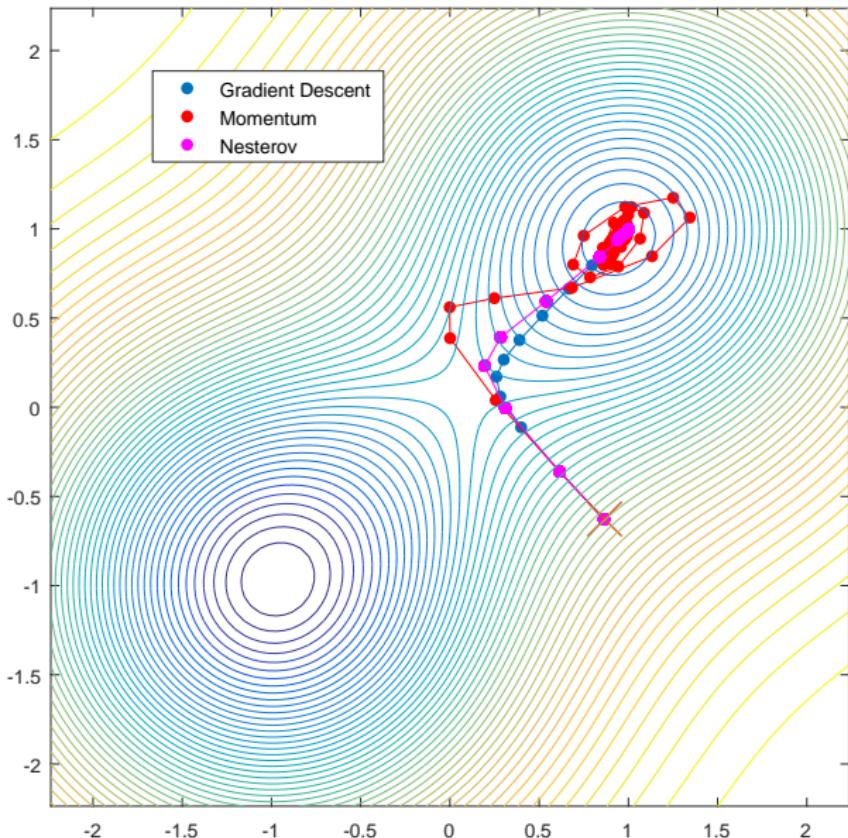
- One can also view this as a 'trust' region factor (see later) that says that we only trust the Taylor expansion provided that the update is not too large.

## Demo



$\epsilon = 0.1$ .  $\mu_t = 1 - 3/(t + 5)$  for both momentum and Nesterov. All trajectories start at the same point and have 50 updates. Nesterov oscillates much less than momentum. See `demoGradDescent.m`

# Demo



$\epsilon = 0.1$ .  $\mu_t = 1 - 3/(t + 5)$  for both momentum and Nesterov.

## Comments on gradient descent

### Good

This is a very simple algorithm that is easy to implement.

---

### Bad

Only improves the solution at each stage by a small amount. If  $\epsilon$  is not small enough, the function may not decrease in value. In practice one needs to find a suitably small  $\epsilon$  to guarantee convergence.

---

### Ugly

The method is coordinate system dependent. Let  $\mathbf{x} = \mathbf{M}\mathbf{y}$  and define

$$\hat{f}(\mathbf{y}) = f(\mathbf{x})$$

We now perform gradient descent on  $\hat{f}(\mathbf{y})$ :

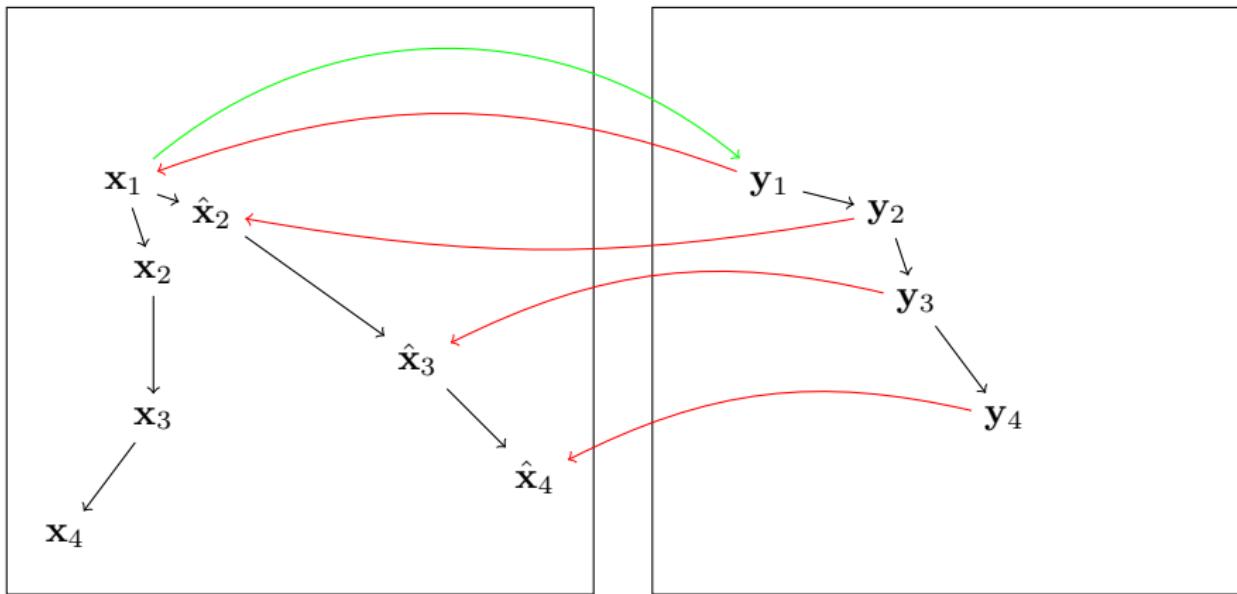
$$[\mathbf{y}_{k+1}]_i = [\mathbf{y}_k]_i - \epsilon \sum_j \frac{\partial f}{\partial x_j} \frac{\partial x_j}{\partial y_i} \rightarrow \mathbf{y}_{k+1} = \mathbf{y}_k - \epsilon \mathbf{M}^T \nabla f(\mathbf{x}_k)$$

Hence

$$\mathbf{M}\mathbf{y}_{k+1} = \mathbf{M}\mathbf{y}_k - \epsilon \mathbf{M}\mathbf{M}^T \nabla f(\mathbf{x}_k) \rightarrow \mathbf{x}_{k+1} = \mathbf{x}_k - \epsilon \mathbf{M}\mathbf{M}^T \nabla f(\mathbf{x}_k)$$

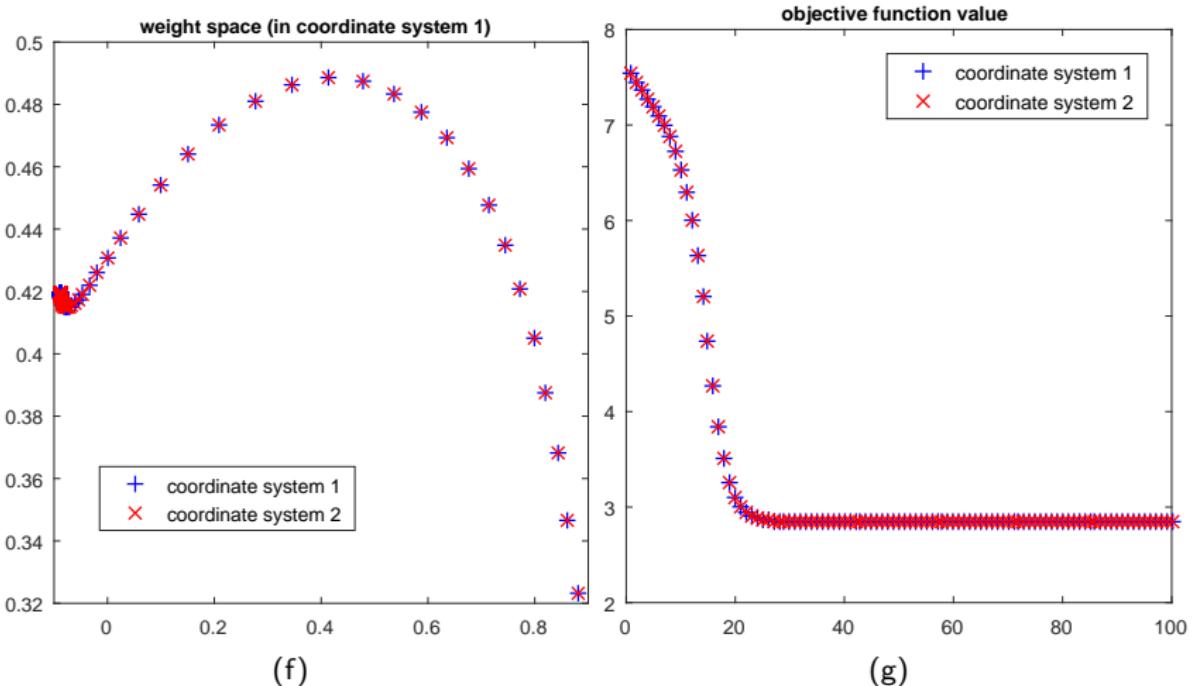
The algorithm is coordinate system dependent (except for orthogonal transformations).

## Coordinate System Dependence



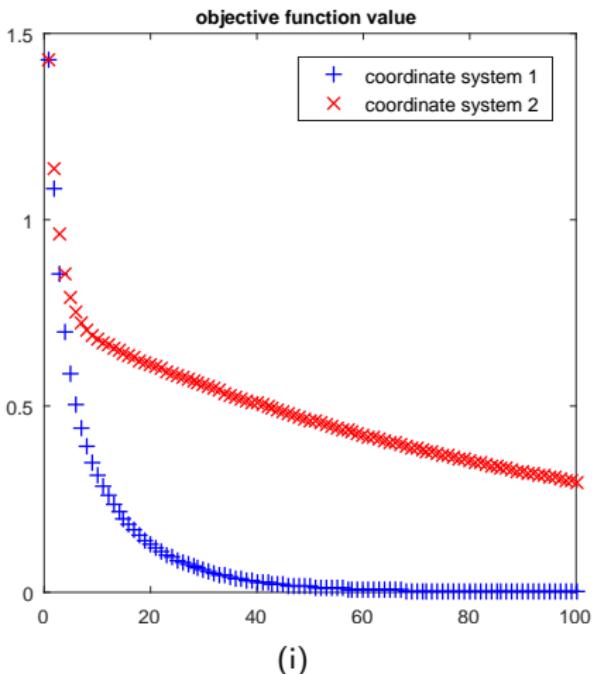
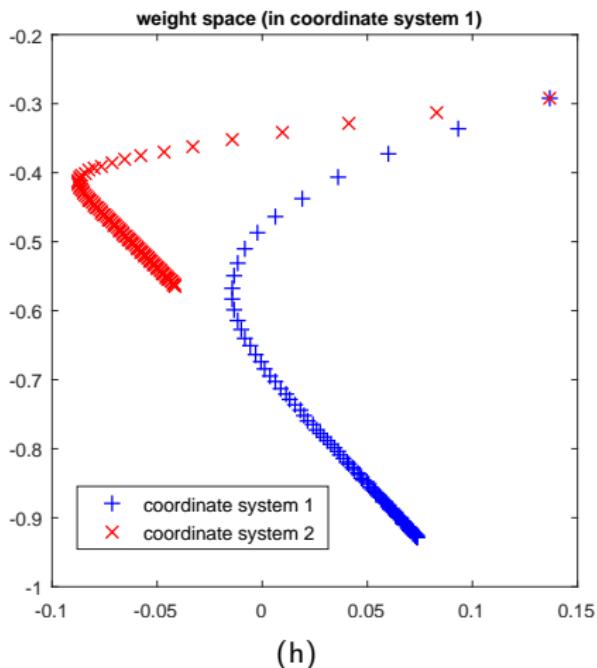
The  $\mathbf{x}_k$  points are the standard gradient descent vectors in the  $x$ -coordinate system. We then map the initial point  $\mathbf{x}_1$  to the corresponding point  $\mathbf{y}_1 = \mathbf{M}^{-1}\mathbf{x}_1$  and begin gradient descent in the  $y$  space. We can then map each point  $\mathbf{y}_k$  back to the corresponding point in the  $x$ -space using  $\hat{\mathbf{x}}_k = \mathbf{M}\mathbf{y}_k$ . In general (unless  $\mathbf{M}$  is orthogonal) the  $x$  and  $\hat{x}$  trajectories are different. See `demoGradDescentCoordTransform.m`.

# Demo



If we use an orthogonal  $M$  the gradient descent is invariant to the coordinate transformation.

# Demo



If we use a non-orthogonal  $M$  the gradient descent depends on the coordinate transformation.

## Line Search and Conjugate Gradients

## Line Search

One way to potentially improve on gradient descent is choose a particular direction  $\mathbf{p}_k$  and search along there. We then find the minimum of the one dimensional problem

$$F(\lambda) = f(\mathbf{x}_k + \lambda \mathbf{p}_k)$$

Finding the optimal  $\lambda^*$  can be achieved using a standard one-dimensional optimisation method. For example if we identify a bracket such that  $f(a) > f(b) < f(c)$  and then fit a polynomial to estimate a new  $x$  with  $f(x) < f(b)$ , this identifies a new bracket. One then repeats until convergence. Once found we set

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda^* \mathbf{p}_k$$

and then choose another search direction  $\mathbf{p}_{k+1}$  and iterate.

---

### Good

By reducing the problem to a sequence of one dimensional optimisation problems, at each stage the potential change in  $f$  is greater than would be typically achievable by gradient descent.

---

### Search directions

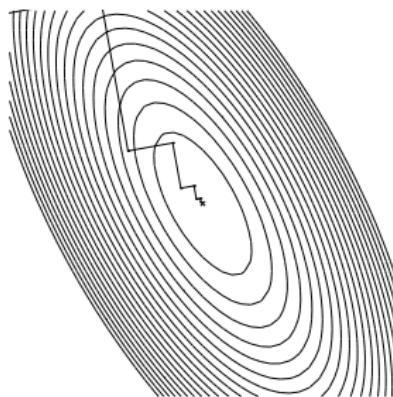
How do we choose a good search direction?

## Choosing the search directions

It would seem reasonable to choose a search direction that points ‘maximally downhill’ from the current point  $\mathbf{x}_k$ . That is, to set

$$\mathbf{p}_k = -\nabla f(\mathbf{x}_k)$$

However, at least for quadratic functions, this is not optimal and leads to potentially zig-zag behaviour:



This zig-zag behaviour can occur for non isotropic surfaces.

# Philosophy

Other ways to avoid problems of basic gradient descent:

---

## Use Quadratic functions to gain insight

Much of the theory of optimisation is based on the following: Even though the problem is not quadratic, let's pretend the problem is quadratic and see what we would do in that case. This will inspire the solution for the general case.

---

## Quadratic functions

In the quadratic function case, we get zig-zag behaviour if  $\mathbf{A}$  is not diagonal. If we could find a coordinate transform  $\mathbf{x} = \mathbf{P}\hat{\mathbf{x}}$

$$\hat{f}(\hat{\mathbf{x}}) = \frac{1}{2}\hat{\mathbf{x}}^T \mathbf{P}^T \mathbf{A} \mathbf{P} \hat{\mathbf{x}} - \mathbf{b}^T \mathbf{P} \hat{\mathbf{x}}$$

with

$$\mathbf{P}^T \mathbf{A} \mathbf{P}$$

being diagonal, then we can perform line-search independently along each axis of  $\hat{\mathbf{x}}$  and find the solution efficiently. This is achieved by the conjugate gradient algorithm.

## Conjugate vectors

The vectors  $\mathbf{p}_i$ ,  $i = 1, \dots, n$  are conjugate for the  $n \times n$  matrix  $\mathbf{A}$  if

$$\mathbf{p}_i^\top \mathbf{A} \mathbf{p}_j = \delta_{ij} \mathbf{p}_i^\top \mathbf{A} \mathbf{p}_i, \quad i, j = 1, \dots, n$$

---

### Finding the optimum of a quadratic

Searching along these conjugate directions effectively ‘diagonalises’ the problem.  
If we write  $\mathbf{x} = \sum_i \alpha_i \mathbf{p}_i$ , then

$$f(\mathbf{x}) = \frac{1}{2} \mathbf{x}^\top \mathbf{A} \mathbf{x} - \mathbf{b}^\top \mathbf{x} = \sum_{i=1}^n \left( \frac{1}{2} \alpha_i^2 \mathbf{p}_i^\top \mathbf{A} \mathbf{p}_i - \alpha_i \mathbf{b}^\top \mathbf{p}_i \right)$$

which gives a set of independent optimisation problems, one for each  $\alpha_i$ . Hence, if we find the optimum along  $n$  conjugate directions, we must have found the optimum  $\mathbf{x}$ .

# Conjugate Gradient

- The idea of conjugate gradients is to perform the independent optimisations along conjugate directions, but do these optimisations sequentially, building up the set of conjugate vectors as we do so.
- To keep the argument simple, set  $\mathbf{x}_1 = \mathbf{0}$  and in general

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

At each iteration we seek the optimum  $\alpha_k$ .

- Using this, we then build up the solution

$$\mathbf{x}_{k+1} = \sum_{i=1}^k \alpha_i \mathbf{p}_i$$

so that after  $n$  iterations, we will have found the overall optimum.

- It's enough at each iteration to search along a direction  $\mathbf{p}_k$  that is conjugate to all previous search directions. Writing  $\mathbf{a} \perp \mathbf{b}$  if  $\mathbf{a}$  is conjugate to  $\mathbf{b}$ . For example for  $n = 4$ , choose  $\mathbf{p}_1$ . Then find  $\mathbf{p}_2 \perp \mathbf{p}_1$ . Then find  $\mathbf{p}_3$  such that  $\mathbf{p}_3 \perp \mathbf{p}_2, \mathbf{p}_3 \perp \mathbf{p}_1$ . Then find  $\mathbf{p}_4$  such that  $\mathbf{p}_4 \perp \mathbf{p}_3, \mathbf{p}_4 \perp \mathbf{p}_2, \mathbf{p}_4 \perp \mathbf{p}_1$ . We now have a full conjugate set.

## Conjugate Gradients

The question now is how to find the conjugate directions. As we saw, we only require that  $\mathbf{p}_k$  is conjugate to  $\mathbf{p}_1, \dots, \mathbf{p}_{k-1}$ . Defining  $\mathbf{g}_k = \nabla f(\mathbf{x}_k)$ , and

$$\beta_k = \mathbf{g}_{k+1}^T \mathbf{g}_{k+1} / (\mathbf{g}_k^T \mathbf{g}_k)$$

one can show that for our quadratic function,

$$\mathbf{p}_{k+1} = -\mathbf{g}_{k+1} + \beta_k \mathbf{p}_k$$

is conjugate to all previous  $\mathbf{p}_i$ ,  $i = 1, \dots, k$ .

A more common alternative is the Polak-Ribière formula.

$$\beta_k = \frac{\mathbf{g}_{k+1}^T (\mathbf{g}_{k+1} - \mathbf{g}_k)}{\mathbf{g}_k^T \mathbf{g}_k}$$

For quadratic functions, with  $\dim \mathbf{x} = n$ , conjugate gradients is guaranteed to find the optimum in  $n$  iterations, each iteration taking  $O(n^2)$  operations. In a more general non-quadratic problem, no such guarantee exists.

# Conjugate Gradients

This gives rise to the conjugate gradients for minimising a function  $f(\mathbf{x})$

- 1:  $k = 1$
- 2: Choose  $\mathbf{x}_1$ .
- 3:  $\mathbf{p}_1 = -\mathbf{g}_1$
- 4: **while**  $\mathbf{g}_k \neq \mathbf{0}$  **do**
- 5:      $\alpha_k = \underset{\alpha_k}{\operatorname{argmin}} f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)$  ▷ Line Search
- 6:      $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$
- 7:      $\beta_k := \mathbf{g}_{k+1}^T \mathbf{g}_{k+1} / (\mathbf{g}_k^T \mathbf{g}_k)$
- 8:      $\mathbf{p}_{k+1} := -\mathbf{g}_{k+1} + \beta_k \mathbf{p}_k$
- 9:      $k = k + 1$
- 10: **end while**

## Higher Order Methods

## Newton's method

Consider a function  $f(\mathbf{x})$  that we wish to find the minimum of. A Taylor expansion up to second order gives

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

The matrix  $\mathbf{H}_f$  is the Hessian.

Differentiating the right hand side with respect to  $\Delta$  (or, equivalently, completing the square), we find that the right hand side has its lowest value when

$$\nabla f = -\mathbf{H}_f \Delta \Rightarrow \Delta = -\mathbf{H}_f^{-1} \nabla f$$

Hence, an optimisation routine to minimise  $f$  is given by the Newton update

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \epsilon \mathbf{H}_f^{-1} \nabla f$$

With this update, the change in the function is

$$\Delta^T \nabla f + \frac{1}{2} \Delta^T \mathbf{H}_f \Delta = \epsilon \left( -1 + \frac{\epsilon}{2} \right) \nabla f^T \mathbf{H}_f^{-1} \nabla f$$

For quadratic functions, Newton's method converges in one step (for  $\epsilon = 1$ ). More generally one uses a value  $\epsilon < 1$  to avoid overshooting effects.

## Comments on Newton's method

### Good

A benefit of the Newton method over gradient descent is that the decrease in the objective function is invariant under a linear change of co-ordinates,  $\mathbf{x} = \mathbf{M}\mathbf{y}$ . Defining  $\hat{f}(\mathbf{y}) \equiv f(\mathbf{x})$ , the change in  $\mathbf{y}$  under a Newton update is

$$-\mathbf{H}_{\hat{f}}^{-1} \nabla_{\mathbf{y}} \hat{f}$$

where  $\nabla_{\mathbf{y}} \hat{f} = \mathbf{M}^T \nabla_{\mathbf{x}} f$ ,  $\mathbf{H}_{\hat{f}} = \mathbf{M}^T \mathbf{H}_f \mathbf{M}$ . In terms of the  $\mathbf{x}$  coordinate system the change is

$$\mathbf{M}\Delta\mathbf{y} = -\mathbf{M}\mathbf{H}_{\hat{f}}^{-1} \nabla_{\mathbf{y}} \hat{f} = -\mathbf{M} (\mathbf{M}^T \mathbf{H}_f \mathbf{M})^{-1} \mathbf{M}^T \nabla_{\mathbf{x}} f = -\mathbf{H}_f^{-1} \nabla_{\mathbf{x}} f = \Delta\mathbf{x}$$

so that the change is independent of the coordinate system (up to linear transformations of the coordinates).

---

### Bad

Storing the Hessian and solving the linear system  $\mathbf{H}_f^{-1} \nabla f$  is very expensive.

## More comments on Newton's method

- Newton's method is not guaranteed to produce a downhill step!
- This only happens (for sure) if  $\epsilon$  is small and  $\mathbf{H}$  is positive definite.
- If  $\mathbf{H}$  is positive definite, one can use a line search in the direction  $\mathbf{H}^{-1}\nabla f$  to ensure we go downhill and make a non-trivial jump.

## Using Conjugate gradient

When we solve the linear system  $\mathbf{H}\Delta = \nabla f$ , a practical approach is to find  $\Delta$  by minimising

$$\Psi(\Delta) \equiv \frac{1}{2}\Delta^T \mathbf{H} \Delta - \Delta^T \nabla f$$

This is typically much faster than calling standard linear solvers (such as Gaussian elimination).

---

## Quasi-Newton

- In Quasi-Newton methods such as Broyden-Fletcher-Goldfarb-Shannon, an approximate inverse Hessian is formed iteratively.
- LBFGS is a popular practical method that limits the memory requirement of BFGS.

## Gauss Newton

Consider objectives of the form

$$E(\mathbf{w}) = \sum_{n=1}^N [y^n - f(\mathbf{x}^n, \mathbf{w})]^2$$

where  $n$  is for example a data index in a set of  $N$  datapoints. This is typical of square loss functions in regression for predictor function  $f$ .

---

## Hessian

$$\frac{\partial^2 E}{\partial w_i \partial w_j} = 2 \sum_n \left( -[y^n - f(\mathbf{x}^n, \mathbf{w})] \frac{\partial^2 f^n}{\partial w_i \partial w_j} + \frac{\partial f^n}{\partial w_i} \frac{\partial f^n}{\partial w_j} \right), \quad f^n \equiv f(\mathbf{x}^n, \mathbf{w})$$

- Close to the minimum of  $E$ , provided the function  $f$  is fitting the data well, then the residuals  $y^n - f(\mathbf{x}^n, \mathbf{w})$  will be small and we can ignore the first term.
- The second term is positive definite. We therefore define the matrix  $\mathbf{C} = 2 \sum_n \nabla f^n (\nabla f^n)^T$  and use this in place of  $\mathbf{H}$  in the Newton update.

## Generalised Gauss Newton

Consider an objective function

$$f(\mathbf{x}) = L(\mathbf{h}(\mathbf{x}))$$

where  $\mathbf{h}(\mathbf{x})$  is a vector-valued function and  $L$  is a convex loss function. Then

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial^2 h_k}{\partial x_i \partial x_j} + \sum_{k,l} \frac{\partial h_k}{\partial x_i} \frac{\partial^2 L}{\partial h_k \partial h_l} \frac{\partial h_l}{\partial x_j}$$

By neglecting the first term above, the matrix with elements

$$G_{i,j} = \sum_{k,l} \frac{\partial h_k}{\partial x_i} \frac{\partial^2 L}{\partial h_k \partial h_l} \frac{\partial h_l}{\partial x_j}$$

is positive semidefinite:

$$\sum_{i,j} z_i G_{i,j} z_j = \sum_{k,l} \underbrace{\sum_{i=1} \frac{\partial h_k}{\partial x_i}}_{\gamma_k} \underbrace{\frac{\partial^2 L}{\partial h_k \partial h_l}}_{M_{k,l}} \underbrace{\sum_j \frac{\partial h_l}{\partial x_j}}_{\gamma_l} = \sum_{k,l} \gamma_k M_{k,l} \gamma_l \geq 0$$

## Generalised Gauss Newton

- Since the Generalised Gauss-Newton matrix is positive semidefinite, the update

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \epsilon \mathbf{G}^{-1} \mathbf{g}$$

where

$$g_i \equiv \frac{\partial f}{\partial x_i}$$

is guaranteed to lower the function value  $f$  (provided  $\epsilon$  is sufficiently small).

- Since  $\frac{\partial L}{\partial h_k}$  will be small close to a minimum, this update will approach that of the Newton method as we converge.

## Levenberg-Marquardt algorithm

In the Gauss-Newton method the update requires us to solve the linear system

$$\mathbf{C}\Delta = \nabla E$$

- However, if we don't have many training datapoints, then  $\mathbf{C}$  will be rank deficient (non-invertible).
- Even then, in practice,  $\mathbf{C}$  may be ill-conditioned (it has a large range of eigenvalues) which makes solving the system numerically unstable.

---

## LM algorithm

The idea is to regularise the problem and solve (using conjugate gradients)

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

for a suitably chosen positive scalar  $\lambda$ .

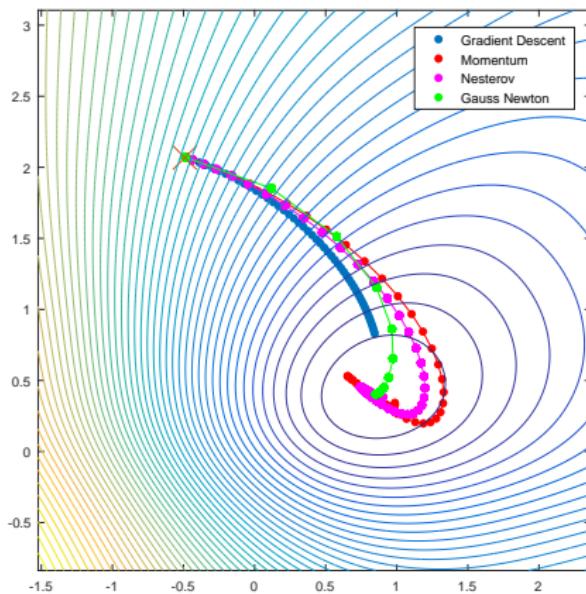
- One can show that this is effectively defining a 'trust region' since this is equivalent to minimising (with respect to  $\Delta$ )

$$E(\mathbf{w}) + \Delta^\top \nabla E + \frac{1}{2} \Delta^\top \mathbf{C} \Delta + \frac{\lambda}{2} \Delta^\top \Delta$$

where the final regularising terms constrains the scale of the update  $\Delta$ .

- This makes sense since we only trust the Taylor expansion in the vicinity of  $\mathbf{w}$ .

## Demo: Gauss Newton on a sum of squares loss



Gauss Newton using a learning rate of  $\epsilon = 0.95$  and regulariser  $\lambda = 0.05$ . The Gauss Newton method converges in fewer iterations, but each iteration is more costly.

## Natural Gradient (Gauss-Newton style for KL loss)

Consider an objective that we wish to minimise wrt  $\theta$  of the form

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

where expectation is wrt  $p(x)$ . The gradient wrt  $\theta$  is

$$g_i = -\left\langle \frac{\partial \log q(x|\theta)}{\partial \theta_i} \right\rangle$$

We define the matrix positive definite matrix curvature matrix

$$C_{i,j} = \left\langle \frac{\partial \log q(x|\theta)}{\partial \theta_i} \frac{\partial \log q(x|\theta)}{\partial \theta_j} \right\rangle$$

Then the update is  $C^{-1}g$  so that

$$\theta(t+1) = \theta(t) - \epsilon C^{-1}g$$

This is analogous to the Gauss-Newton method and guarantees a reduction in the KL (for small  $\epsilon$ ).

# Natural Gradient Invariance

Consider a transformation  $\theta_i = f_i(\phi)$ . Then

$$g_i^\phi = -\frac{\partial \langle \log q(x|f(\phi)) \rangle}{\partial \phi_i} = -\left\langle \sum_j \frac{\partial \log q(x|\theta)}{\partial \theta_j} \frac{\partial \theta_j}{\partial \phi_i} \right\rangle = [J^\top \langle g^\theta \rangle]_i$$

where  $J_{i,j} = \frac{\partial \theta_i}{\partial \phi_j}$ . Similarly,

$$C_{i,j}^\phi = \left\langle \frac{\partial \log q(x|f(\phi))}{\partial \phi_i} \frac{\partial \log q(x|f(\phi))}{\partial \phi_j} \right\rangle = \left\langle [J^\top g^\theta]_i [J^\top g^\theta]_j \right\rangle$$

In matrix notation

$$C^\phi = J^\top \left\langle g^\theta (g^\theta)^\top \right\rangle J$$

Hence the update

$$\delta^\phi = (C^\phi)^{-1} g^\phi = \left( J^\top \left\langle g^\theta (g^\theta)^\top \right\rangle J \right)^{-1} J^\top \langle g^\theta \rangle = J^{-1} \left( \left\langle g^\theta (g^\theta)^\top \right\rangle \right)^{-1} \langle g^\theta \rangle$$

- The  $\phi$  update corresponds to a new point in the  $\theta$  space

$$\begin{aligned}
 \theta_i^{new} &= f_i(\phi + \epsilon\delta^\phi) \\
 &\approx f_i(\phi) + \epsilon \sum_j \delta_j^\phi \frac{\partial f_i}{\partial \phi_j} \\
 &= \theta_i + \epsilon \sum_j J_{ij} [J^{-1}\delta^\theta]_j \\
 &= \theta_i + \epsilon\delta_i^\theta
 \end{aligned}$$

- The final expression is the update to  $\theta$  that we would have made in the  $\theta$  coordinate system.
- Hence, *up to first order* the parameter update is independent of the coordinate system.
- This is not the case for Newton's method (see next page).

In Newton's method, the update is (just for a scalar for simplicity)

$$\begin{aligned}\phi(t+1) &= \phi(t) - \epsilon \left( \frac{\partial^2 E}{\partial \phi^2} \right)^{-1} \frac{\partial E}{\partial \phi} \\ &= \phi(t) - \epsilon \left( \frac{\partial}{\partial \phi} \left( \frac{\partial E}{\partial \theta} \frac{\partial \theta}{\partial \phi} \right) \right)^{-1} \frac{\partial E}{\partial \theta} \frac{\partial \theta}{\partial \phi} \\ &= \phi(t) - \epsilon \left( \frac{\partial E}{\partial \theta} \frac{\partial^2 \theta}{\partial \phi^2} + \left( \frac{\partial \theta}{\partial \phi} \right)^2 \frac{\partial^2 E}{\partial \theta^2} \right)^{-1} \frac{\partial E}{\partial \theta} \frac{\partial \theta}{\partial \phi}\end{aligned}$$

The term  $\frac{\partial E}{\partial \theta} \frac{\partial^2 \theta}{\partial \phi^2}$  prevents the invariance. If this were not present, we would have

$$\phi(t+1) = \phi(t) - \epsilon \frac{\partial \phi}{\partial \theta} \left( \frac{\partial^2 E}{\partial \theta^2} \right)^{-1} \frac{\partial E}{\partial \theta}$$

and

$$f(\phi(t+1)) \approx f(\phi(t)) - \epsilon \frac{\partial f}{\partial \phi} \frac{\partial \phi}{\partial \theta} \left( \frac{\partial^2 E}{\partial \theta^2} \right)^{-1} \frac{\partial E}{\partial \theta}$$

which gives (to first order) the Newton update in the  $\theta$  coordinate system

$$\theta(t+1) \approx \theta(t) - \epsilon \left( \frac{\partial^2 E}{\partial \theta^2} \right)^{-1} \frac{\partial E}{\partial \theta}$$

Note that the invariance still holds for linear transformations and as we approach the optimum since then the term  $\frac{\partial E}{\partial \theta} \frac{\partial^2 \theta}{\partial \phi^2}$  becomes small.

# Large Scale Learning<sup>1</sup>

David Barber

---

<sup>1</sup> These slides accompany the book *Bayesian Reasoning and Machine Learning*. The book and demos can be downloaded from [www.cs.ucl.ac.uk/staff/D.Barber\(brml](http://www.cs.ucl.ac.uk/staff/D.Barber(brml)). Feedback and corrections are also available on the site. Feel free to adapt these slides for your own purposes, but please include a link to the above website.

# Large Scale Problems

## Stochastic Gradient Descent

SGD is useful in the situations:

- The dataset is too big to store all of it.
- The dataset is so big that it will take a long time to take an update

---

## Exploiting Sparsity

- We will also consider some special linear models that we can solve efficiently by exploiting any sparsity that might exist in the problem.

# Batch vs Online

## Batch

- Compute the gradient by first summing over all the training data inputs.
  - Then do a gradient update.
- 

## Online

- Datapoints arrive in a stream
- Compute approximate gradient by summing over a single datapoint.
- Then do a gradient update immediately for this datapoint.

# Linear Models

## Linear regression

Least squares error:

$$E(\boldsymbol{\theta}) = \sum_n \left( y^n - \boldsymbol{\theta}^\top \mathbf{x}^n \right)^2 + \lambda \boldsymbol{\theta}^\top \boldsymbol{\theta}$$

The optimum is given when the gradient wrt  $\boldsymbol{\theta}$  is zero:

$$\frac{\partial E}{\partial \theta_i} = -2 \sum_n \left( y^n - \boldsymbol{\theta}^\top \mathbf{x}^n \right) x_i^n + 2\lambda \theta_i = 0$$

$$\underbrace{\sum_n y^n x_i^n}_{b_i} = \sum_j \left( \underbrace{\lambda \delta_{i,j} + \sum_n x_i^n x_j^n}_{A_{ij}} \right) \theta_j$$

Hence, in matrix form, this is

$$\mathbf{b} = \mathbf{A}\boldsymbol{\theta}, \rightarrow \boldsymbol{\theta} = \mathbf{A}^{-1}\mathbf{b}$$

which is a simple linear system that can be solved in  $O\left((\dim \boldsymbol{\theta})^3\right)$  time.

## Sparse Data

- Let  $D = \dim \theta$ .
- Storing  $\mathbf{A}$  scales  $O(D^2)$  – very expensive for large  $D$ .
- Note that the gradient

$$\frac{\partial E}{\partial \theta_i} = -2 \sum_n \left( y^n - \boldsymbol{\theta}^\top \mathbf{x}^n \right) x_i^n + 2\lambda\theta_i$$

Can be computed without storing  $\mathbf{A}$ .

- Suggests that we can use gradient based methods to find  $\theta$ .
- If each  $\mathbf{x}$  is itself sparse, with density factor  $0 \leq d \leq 1$  (so that on average each  $\mathbf{x}$  contains only  $dD$  non zero entries), computing the gradient takes  $O(dDN)$  time and order  $O(N + D)$  storage.

## Online Stochastic Gradient Descent

We can consider the equivalent rescaled objective

$$\frac{1}{N} \sum_n \left( y^n - \boldsymbol{\theta}^T \mathbf{x}^n \right)^2 + \gamma \boldsymbol{\theta}^T \boldsymbol{\theta}$$

where  $\gamma = \lambda/N$ , with gradient

$$\mathbf{g} = -2 \frac{1}{N} \sum_n \left( y^n - \boldsymbol{\theta}^T \mathbf{x}^n \right) \mathbf{x}^n + 2\gamma \boldsymbol{\theta}$$

- We can view the first term as the expectation wrt the empirical distribution.
- By approximating the empirical distribution by a limited number of samples  $N'$ , we can form the approximation

$$\mathbf{g} \approx -2 \frac{1}{N'} \sum_{n'} \left( y^{n'} - \boldsymbol{\theta}^T \mathbf{x}^{n'} \right) \mathbf{x}^{n'} + 2\gamma \boldsymbol{\theta}$$

- Then do a gradient update.
- Only requires that  $N'$  samples from the dataset  $\mathbf{X}$  are available for each gradient calculation.

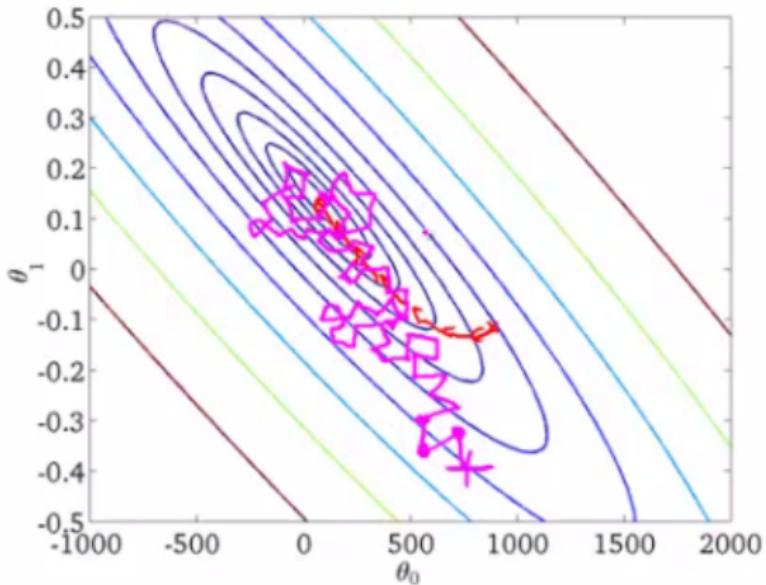
## Online Stochastic Gradient Descent

- In the extreme case we can assume that only a stream of input-output pairs  $\mathbf{x}^n, y^n$  is available (we don't store these pairs – they just 'appear').
- Then as each datapoint  $n$  arrives, we can compute a stochastic approximation to the gradient

$$\mathbf{g} \approx 2 \left( y^n - \boldsymbol{\theta}^T \mathbf{x}^n \right) \mathbf{x}^n + 2\gamma \boldsymbol{\theta}$$

- Then do a gradient update,  $\boldsymbol{\theta}^{new} = \boldsymbol{\theta} - \epsilon \mathbf{g}$ .
- Provided that the gradient step is small, on average we will tend to take a gradient in roughly the same direction as the batch update.
- Many very large (Google scale) 'Big Data' algorithms use Stochastic Gradient Descent with data stored on different machines.
- The stochastic nature makes the algorithm robust to synchronisation issues in parallel implementations.
- Vowpal Wabbit and similar tools use SGD to train very large linear predictors.

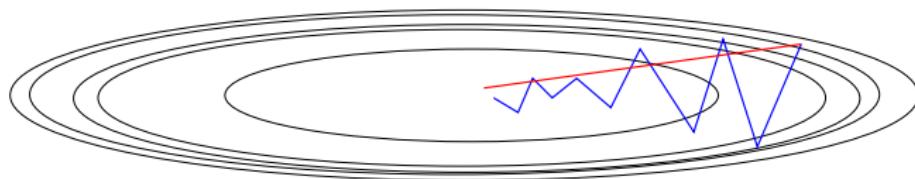
## Stochastic Gradient Descent



In red is the path of standard gradient descent, in magenta the path of stochastic gradient descent. Eventually they both converge to the true minimum. Thanks to Alex Holehouse for the picture.

## Higher Order Methods

- Gradient Descent only makes small local moves and is thus slow to converge.
- Higher order methods combine better search directions with line search to improve convergence.
- Efficient ways to exploit sparsity in the feature (input) vector  $x$  to perform regression on very large datasets with very high dimensional  $x$ .



- In blue is the standard gradient descent path and in red is the conjugate gradients (with line search) path.
- For a quadratic surface, conjugate gradients is much faster than standard gradient descent.

# Conjugate Gradients

Conjugate gradients is an efficient way to minimise a function  $f(\theta)$

- 1:  $k = 1$
- 2: Choose  $\theta_1$ .
- 3:  $p_1 = -g_1$
- 4: **while**  $g_k \neq 0$  **do**
- 5:      $\alpha_k = \underset{\alpha_k}{\operatorname{argmin}} f(\theta_k + \alpha_k p_k)$  ▷ Line Search
- 6:      $\theta_{k+1} := \theta_k + \alpha_k p_k$
- 7:      $\beta_k := g_{k+1}^T g_{k+1} / (g_k^T g_k)$
- 8:      $p_{k+1} := -g_{k+1} + \beta_k p_k$
- 9:      $k = k + 1$
- 10: **end while**

The algorithm is particularly useful in the case of quadratic functions  $f$  since then the line search can be carried out exactly.

# Line Search for a Quadratic

Given a quadratic

$$f(\mathbf{x}) \equiv \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{x}^T \mathbf{b}$$

we wish to find  $\lambda$  that minimises along the line direction  $\mathbf{p}$ :

$$\min_{\lambda} f(\mathbf{x} + \lambda \mathbf{p})$$

$$\begin{aligned} f(\mathbf{x} + \lambda \mathbf{p}) &= \frac{1}{2} (\mathbf{x} + \lambda \mathbf{p})^T \mathbf{A} (\mathbf{x} + \lambda \mathbf{p}) - \mathbf{b}^T (\mathbf{x} + \lambda \mathbf{p}) \\ &= \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{b}^T \mathbf{x} + \frac{1}{2} \lambda^2 \mathbf{p}^T \mathbf{A} \mathbf{p} + \lambda (\mathbf{p}^T \mathbf{A} \mathbf{x} - \mathbf{b}^T \mathbf{p}) \end{aligned}$$

Differentiating wrt  $\lambda$  and setting to 0, we obtain

$$\lambda = -\frac{\mathbf{p}^T (\mathbf{A} \mathbf{x} - \mathbf{b})}{\mathbf{p}^T \mathbf{A} \mathbf{p}}$$

## Conjugate Gradients for the quadratic form

```
function [x, val]=conjgrad(A,b,x0,opts)
%CONJGRAD conjugate gradients for minimising 0.5*x'*A*x-x'*b
% for positive definite A
% [x, val]=conjgrad(A,b,x0,opts)
% x0 is the initial starting solution
x=x0;
g=A*x-b;
p=-g;
valOld=realmax;
for loop=1:opts.maxits
    alpha=-(p'*g)/(p'*A*p);
    x=x+alpha*p;
    gnew=A*x-b;
    beta=(gnew'*gnew)/(g'*g);
    p=-gnew+beta*p;
    g=gnew;
    val = 0.5*x'*(g-b);
    if loop>1; if valOld-val<opts.tol; break; end; end
    valOld=val;
end
```

# Conjugate Gradients for Linear Regression

$$\frac{1}{2} \sum_n \left( y^n - \boldsymbol{\theta}^\top \mathbf{x}^n \right)^2 + \frac{1}{2} \lambda \boldsymbol{\theta}^\top \boldsymbol{\theta} = \frac{1}{2} \boldsymbol{\theta}^\top \mathbf{A} \boldsymbol{\theta} - \boldsymbol{\theta}^\top \mathbf{b}$$

where

$$\mathbf{A} = \lambda \mathbf{I} + \sum_n \mathbf{x}^n (\mathbf{x}^n)^\top, \quad \mathbf{b} = \sum_n y^n \mathbf{x}^n$$

- Note that only place the problematic  $\mathbf{A}$  appears in the CG algorithm is in the term

$$\mathbf{p}^\top \mathbf{A} \mathbf{p} = \lambda \mathbf{p}^2 + \sum_n (\mathbf{p}^\top \mathbf{x}^n)^2$$

- This means that we can run (batch) CG without computing  $\mathbf{A}$ .

# Conjugate Gradients for Sparse Linear Regression

```
function [w, valCG]=conjgradSparseLinReg(X,y,w0,lambda,opts)
%CONJGRADSPARSELINREG conjugate gradients for minimising:
% sum_n [(y(n)-w'*X(:,n))^2 +lambda*w'*w
% (Trivial Initialisation code lines removed)
% MAIN LOOP:
for loop=1:opts.maxits
    tmp=lambda*sum(p.*p);
    for n=1:N
        tmp=tmp+(sum(X(:,n).*p))^2;
    end
    alpha=-(p'*g)/tmp;
    w=w+alpha*p;
    gnew=-b+lambda*w;
    for n=1:N
        gnew=gnew+X(:,n)*(sum(X(:,n).*w));
    end
    beta=(gnew'*gnew)/(g'*g);
    p=-gnew+beta*p;
    g=gnew;
    val = 0.5*w'*(g-b);
    if loop>1; if valOld-val<opts.tol; break; end; end
    valOld=val;
end
```

# Sparse Linear Learning

- `demoLargeScaleLinearRegression.m` shows how to quickly perform linear regression for very high dimensional sparse data.
- Sparse Linear Learning can also easily be applied to classification, for example logistic regression ●
- An advantage of these simple models is that they are easily parallelisable ●
- Distributed data can be handled easily ●
- Stochastic Gradient Descent is a popular and simple mechanism to make parallel algorithms robust to synchronisation issues ●

# Neural Nets

David Barber

# Learning Objective

## Lectures

- Motivation for neural networks.
- Understanding of feedforward networks.
- How to train neural networks.
- Autoencoders for compression.

NN background

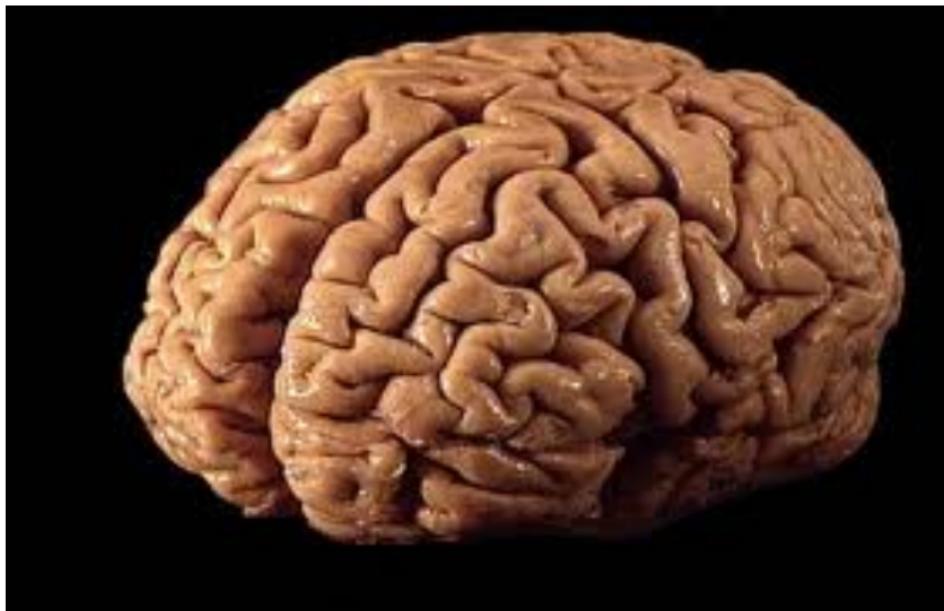
# Neural Networks

- “Neural Nets” traditionally refers to layered information processing models.
  - Inspiration comes from biology in which information is typically processed in layers and hierarchically represented.
  - We will focus here on traditional ‘feedforward’ models.
  - NNs can be problematic to train. However, once trained they are *lightening fast* to use on a novel datapoint.
- 

## Deep Learning

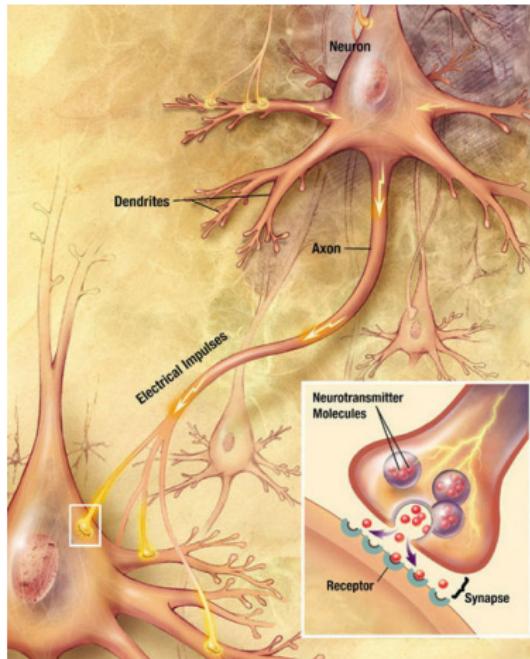
- NNs have resurged in interest in the last few years. Also called ‘deep learning’.
- Sense that very complex tasks (object recognition, learning complex structure in data) requires going beyond simple statistical techniques.
- Recent results using NNs are encouraging.
- Big Interest (!) in this – Google, Facebook, etc.

## Astonishing Hypothesis: Crick



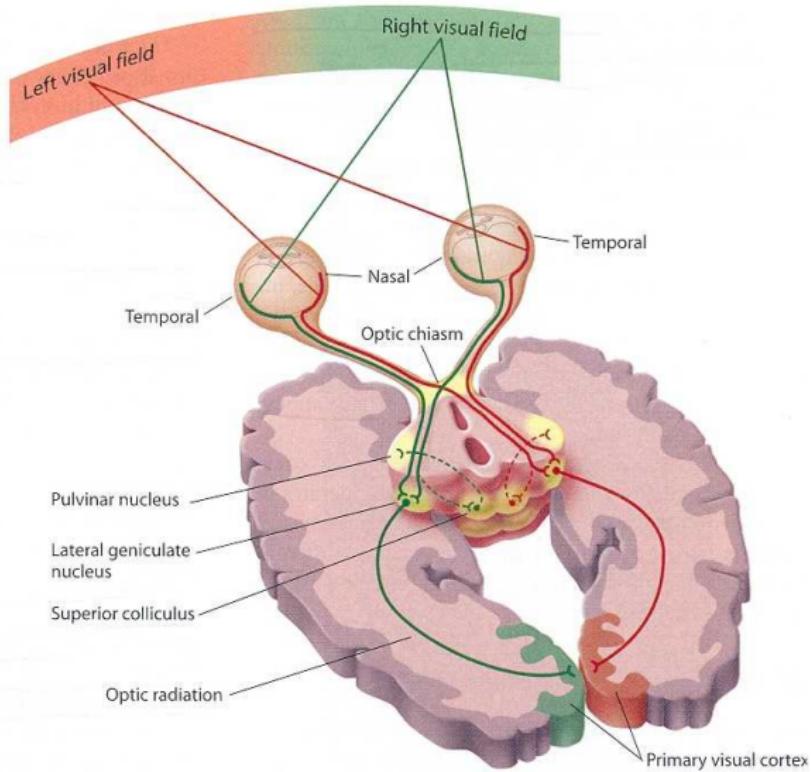
*"A person's mental activities are entirely due to the behaviour of nerve cells and the molecules that make them up and influence them."*

# Neurons

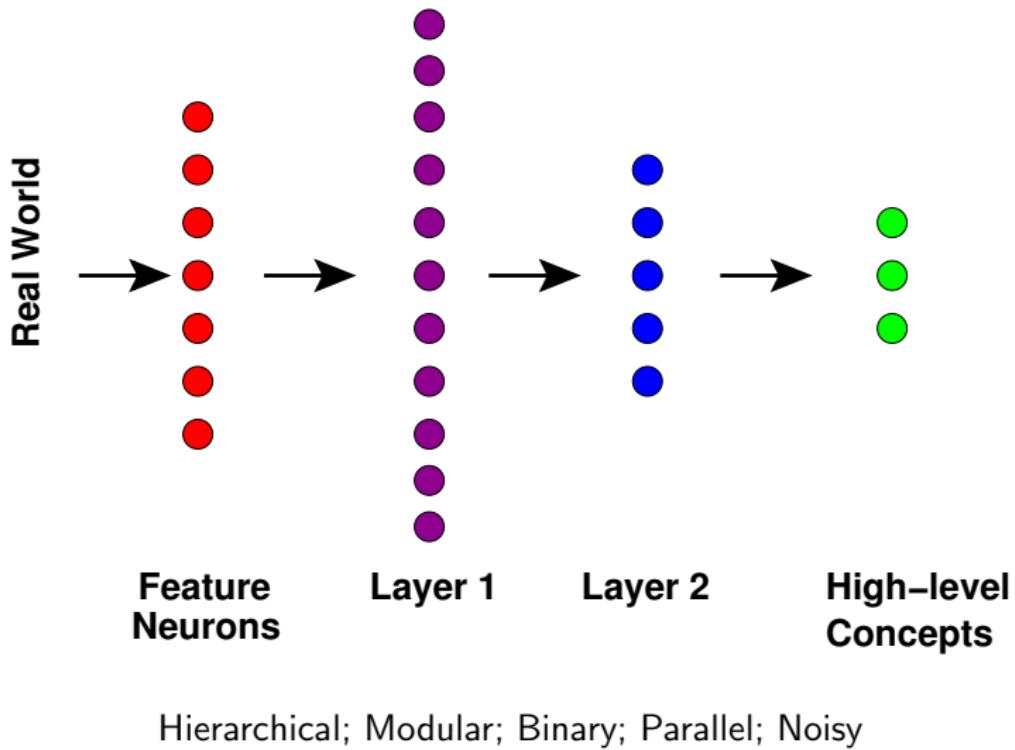


Learning = adjusting the strengths of the connections between neurons.

# Visual Pathway



# Information Processing in Brains

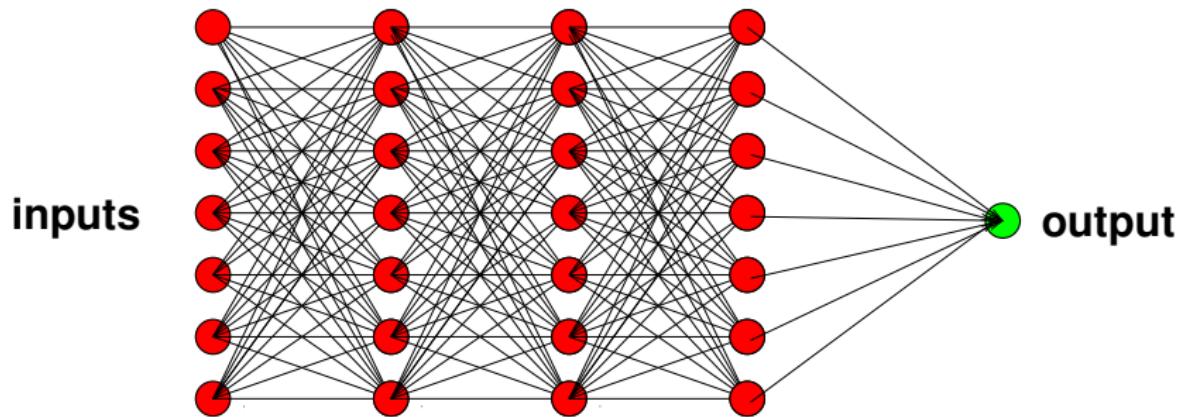


# Connectionism

1960 Realised a perceptron can only solve simple tasks.

1970 Decline in interest.

1980 New computing power made training multilayer networks feasible.



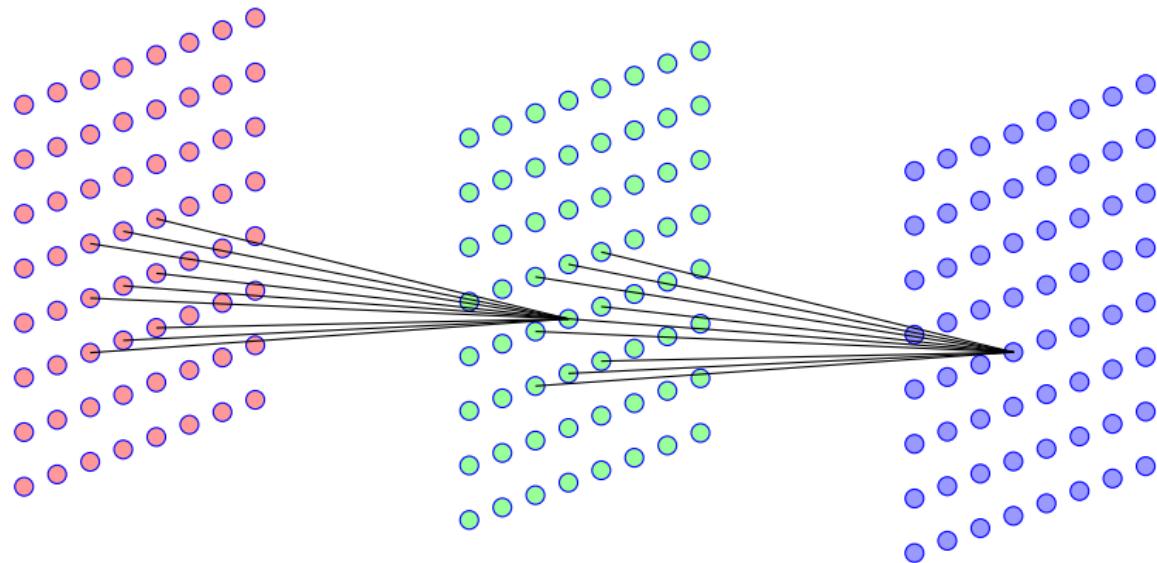
Each node (or 'neuron') computes a function of a weighted combination of parental nodes:  $h_j = \sigma(\sum_i w_{ij} h_i)$

## Digit Recognition using a Neural Network

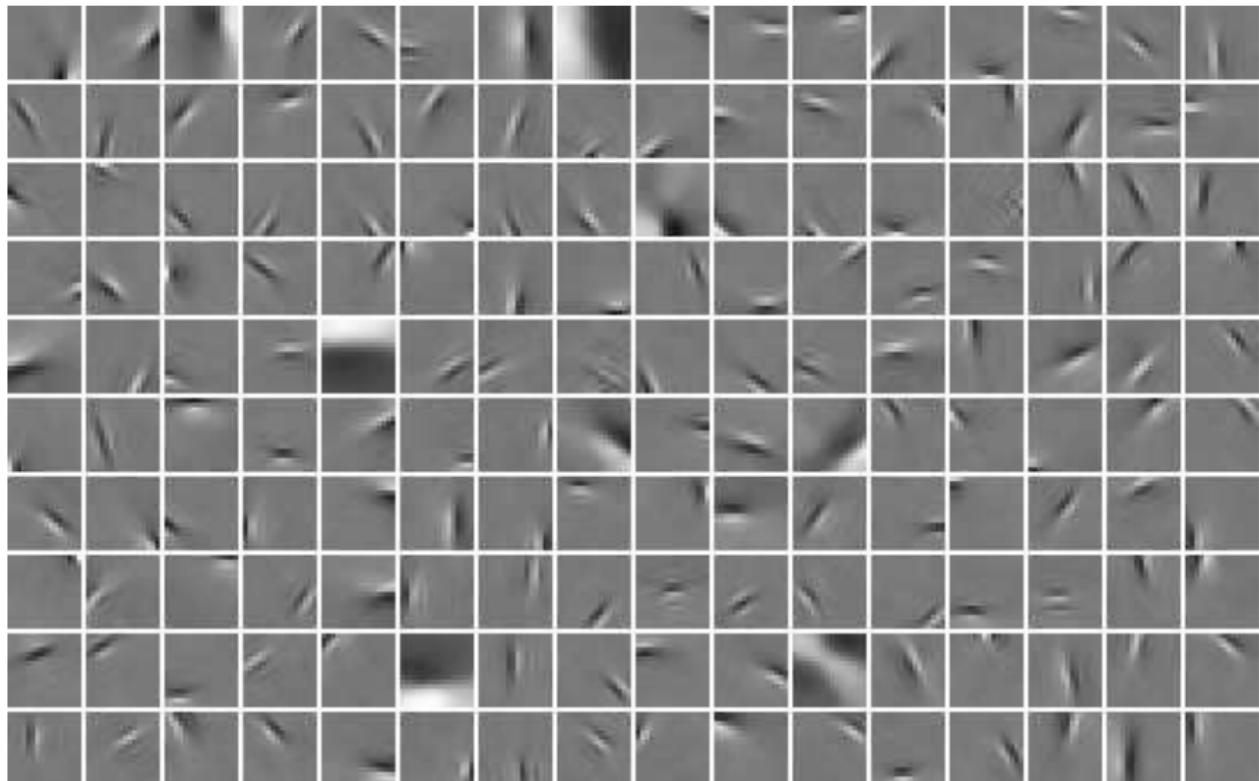
7210414959  
0690159734  
9665407401  
3134727121  
1742351244

Computer makes error less than 1/1000.

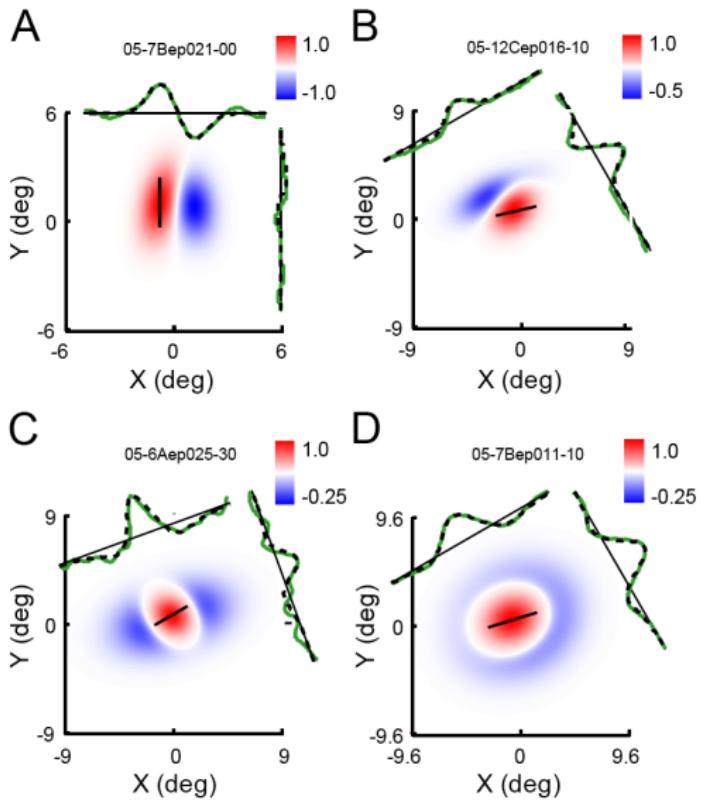
# Digit recognition using a neural network



## Feature neurons in the first layer



# Feature neurons in a real brain



## Definition

- Consider a vector input  $\mathbf{x}$  that gets mapped to an output  $\mathbf{y}$  through intermediate layers.
- For a network with  $L$  layers, we write the vector function that the network computes as

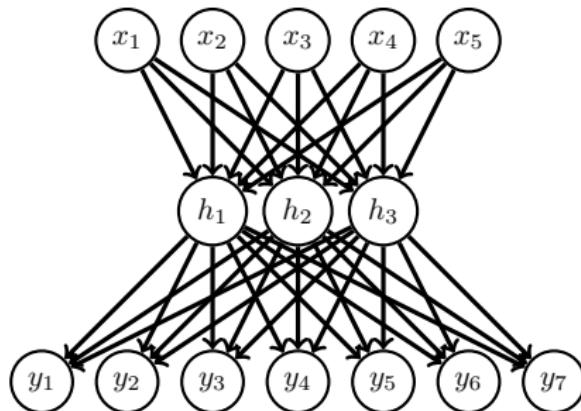
$$f(\mathbf{x}|\mathcal{W}) \equiv \sigma_L(\mathbf{W}_L \mathbf{h}_{L-1})$$

where

$$\mathbf{h}_l = \sigma_l(\mathbf{W}_l \mathbf{h}_{l-1}), \quad l = 2, \dots, L-1, \quad \mathbf{h}_1 = \sigma_1(\mathbf{W}_1 \mathbf{x})$$

- The dimension of each hidden layer is given by  $H_l \equiv \dim(\mathbf{h}_l)$ .
- The transfer functions  $\sigma_l(x)$  can be different for each layer (or even different for units within the same layer).
- Common also to include ‘bias’ terms  $\mathbf{b}$ , so that  $\mathbf{h}_l = \sigma_l(\mathbf{W}_l \mathbf{h}_{l-1} + \mathbf{b}_l)$ . The biases form part of the parameter set.
- Typical to use an invertible monotonic function,  $\sigma(x) = e^x/(1 + e^x)$  or  $\sigma(x) = \tanh(x)$ .

## The NN diagram



- Input-output neural network with a single hidden layer.
- Can write this more compactly as  $\mathbf{x} \rightarrow \mathbf{h} \rightarrow \mathbf{y}$ .
- Can have more than one hidden layer:  $\mathbf{x} \rightarrow \mathbf{h}_1 \rightarrow \mathbf{h}_2 \dots \rightarrow \mathbf{y}$ .
- This is a diagrammatic representation of a function  $\mathbf{y} = f(\mathbf{x}|\mathcal{W})$ .
- If the transfer functions are all linear, the function is linear and additional hidden layers play no role. Typically we are interested in the situation of using non-linear transfer functions – takes us away from classical statistical techniques.
- Very rich functional structure but hard to analyse.

# Regression

- Neural networks can be used to solve a regression problem by minimising the squared loss on a set of input-output example pairs,  $(\mathbf{x}^n, \mathbf{y}^n)$ ,  $n = 1, \dots, N$ :

$$E(\mathcal{W}) = \sum_{n=1}^N [\mathbf{y}^n - \mathbf{f}(\mathbf{x}^n | \mathcal{W})]^2$$

- Optimisation can be achieved using gradient descent (slow) or higher order methods such as BFGS or conjugate gradients.
- A well documented issue with training ‘deep’ networks is that it is difficult to find a good initialisation of the weights  $\mathcal{W}$ .
- For this reason, ‘layerwise’ training algorithms and unsupervised-initialisation approaches have been popular.
- Common to use distributed computing methods on large scale problems.
- Then use Stochastic Gradient Descent, which is robust to processor synchronisation issues.
- Note that the gradient can be computed efficiently using the ‘back-propagation’ algorithm, or more generally just using AutoDiff.

# Loss functions

## Regression

Squared loss  $E(\mathbf{y}, \mathbf{h}^L) = (\mathbf{y} - \mathbf{h}^L)^2$  is the most common.

---

## Classification

Logistic loss for binary classification  $y \in \{+1, -1\}$ . We have a single output with  $\sigma_L(x) \equiv \sigma(x)$ , where  $\sigma(x)$  is the logistic sigmoid.

$$E(y, h^L) = -\log (\mathbb{I}[y = +1] h^L + \mathbb{I}[y = -1] (1 - h^L))$$

Softmax for multiple classes. Have  $C$  outputs, one for each class with:

$$\sigma_L(x_c) \equiv \frac{e^{W_c^L x_c}}{\sum_c e^{W_c^L x_c}}$$

$$E(\mathbf{y}, \mathbf{h}^L) = -\log \sum_{c=1}^C \mathbb{I}[y = c] h_c^L$$

## Penalty Terms

- NNs are potentially very powerful functions.
- Given enough hidden layers, pretty much any function (including random noise!) can be modelled.
- To discourage overfitting, common to add penalty terms to the error.
- The most common regularisation term is

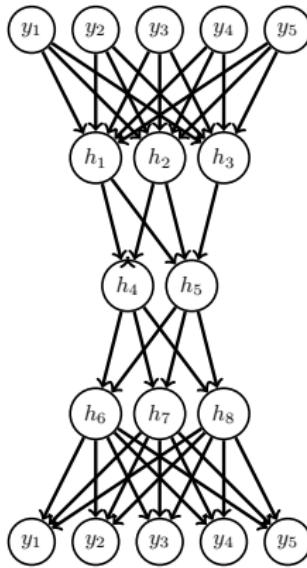
$$\lambda \sum_l \sum_{ij} (W_{ij}^l)^2$$

Easy to include this term in the gradient calculation.

- Note that usually no need to penalise bias terms.
- Regularisation parameter  $\lambda$  is set typically by cross validation.

## Specific NN Architectures

# Autoencoder



- Try to map the input back to itself.
- Hidden Layer with the smallest number of units is called the 'bottleneck'.
- The bottleneck forces the network to try to find a low dimensional representation of the data.
- Useful for unsupervised learning.

## Autoencoder

- Aim is to try to find a lower dimensional ('compressed') representation of higher dimensional data  $\mathbf{y}$ .
- When the output is set to the input, the objective is to minimise

$$E(\mathcal{W}) = \sum_{n=1}^N [\mathbf{y}^n - \mathbf{f}(\mathbf{y}^n | \mathcal{W})]^2$$

wrt  $\mathcal{W}$ .

- Typically use several hidden layers,  $\mathbf{y} \rightarrow \mathbf{h}^1 \rightarrow \dots \rightarrow \mathbf{h}^L$ .
- The 'bottleneck' (hidden layer with smallest dimension) effectively means that the outputs  $\mathbf{y}$  are then approximated by the lower dimensional representations in the bottleneck layer.
- These representations can be used for low dimensional encodings of the data  $\mathbf{y}$  or for subsequent processing (such as classification).

`demoAutoencoderMNIST.m`

## Autoencoder on MNIST digits

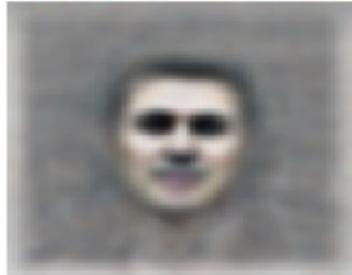
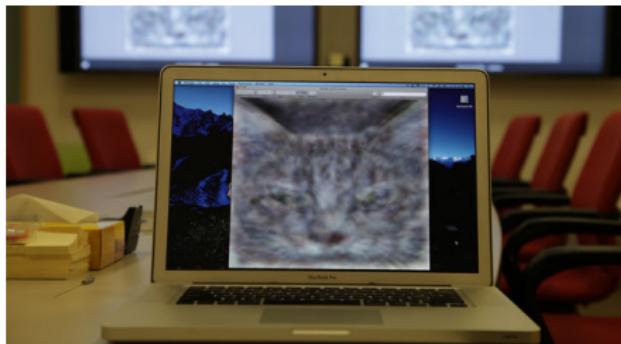
- 60,000 training images ( $28 \times 28 = 784$  pixels).
- Use a form of autoencoder to find a lower (30) dimensional representation.



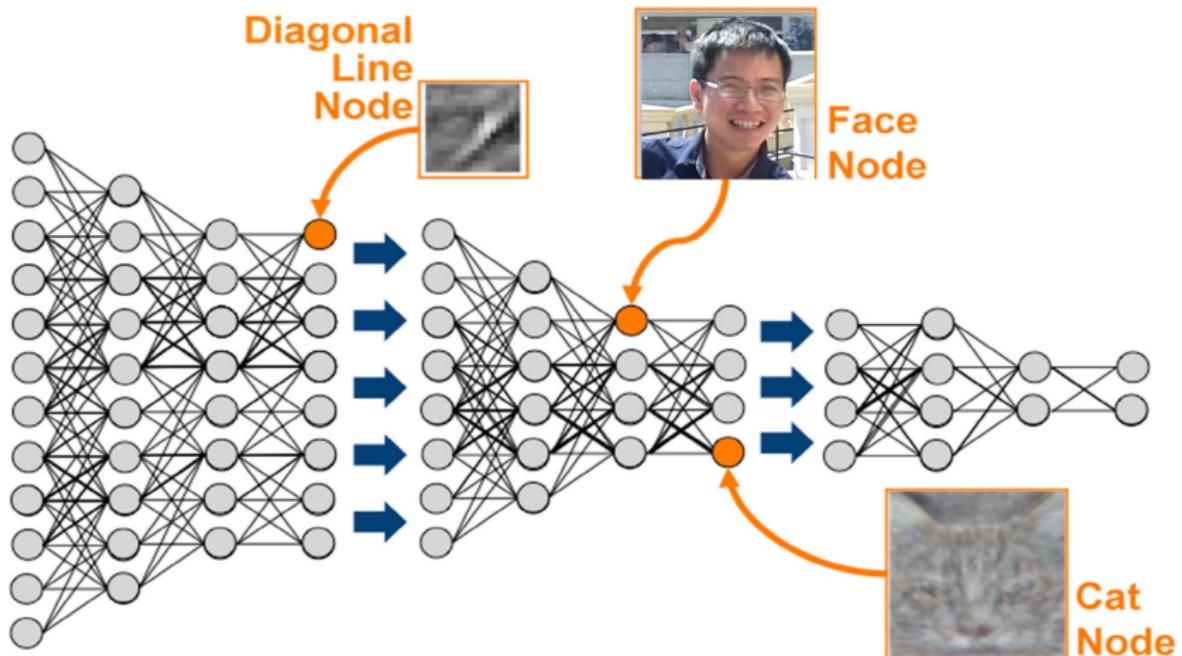
**Figure:** Reconstructions using  $H = 30$  components. From the Top: Original image, NN using 1 layer (reconstruction error=2.42); NN using  $H = [30, 100]$  (reconstruction error=2.38); PCA (reconstruction error=14.46)

# Google Cats

- 10 Million Youtube video frames (200x200 pixel images).
- Use a specialised autoencoder with 9 layers (1 billion weights).
- 2000 computers + two weeks of computing.
- Examine bottleneck units to see what images they most respond to.



# Google Autoencoder



From Nando De Freitas

# Autoencoders and PCA

## PCA

- Given a set of data  $Y = (\mathbf{y}^1, \dots, \mathbf{y}^N)$  consider trying to find a rank  $H$  approximation  $\tilde{Y}$  to  $Y$  that minimises the squared loss  $\|Y - \tilde{Y}\|^2$ .
- Well known that the minimal rank  $H$  loss is given by taking the singular value decomposition of  $Y = USV^T$ , taking the largest  $H$  singular values:

$$\tilde{Y} = U_H S_H V_H^T$$

- This is called Principal Components Analysis and is a classical algorithm.
- There is a polynomial algorithm for SVD and also very fast approximations.

---

## Autocencoder

- Consider  $\mathbf{y} \rightarrow \mathbf{h} \rightarrow \tilde{\mathbf{y}}$ , a single hidden layer net with linear transfer  $\sigma_2(x) = x$ . Let's call the output of the network  $\tilde{Y}$ . Then the squared loss is given by

$$E = \|Y - \tilde{Y}\|^2$$

where  $\tilde{Y} = W_2\sigma_1(W_1Y)$ . If  $\sigma_1(x) = x$  then  $\tilde{Y} = W_2W_1Y = W_2W_1USV^T$  which has rank  $H$ . But this is the same as PCA by setting  $W_2 = U_H$  and  $W_1 = U_H^T$  since then  $\tilde{Y} = U_H S_H V_H^T$ .

- Hence PCA is equivalent to an autoencoder  $\mathbf{y} \rightarrow \mathbf{h} \rightarrow \tilde{\mathbf{y}}$  with linear transfer functions.

## Autoencoders and PCA

- Let's try to make a more powerful autoencoder by using a non-linear transfer function  $\sigma_1(x)$ .
- Then  $\tilde{Y} = W_2\sigma_1(W_1Y)$ .
- This means that  $\tilde{Y}$  has still has rank  $H$ , the size of the hidden layer.
- Note that  $\sigma_1(W_1Y)$  is also some matrix.
- Hence, overall, this corresponds to trying to find a rank  $H$  matrix  $\tilde{Y}$ .
- But this *optimally* solved by PCA – the non-linearity cannot improve the power of this model.
- We arrive at an important conclusion, that for a linear output  $\sigma_2(x) = x$ , the optimal transfer function on the single-hidden layer autoencoder is given by  $\sigma_1(x) = x$ , with the weights set by the SVD (PCA) of  $Y$ .
- In other words, you cannot beat PCA if you only have a linear transfer function at the output (for a single hidden layer autoencoder).

# Autoencoders

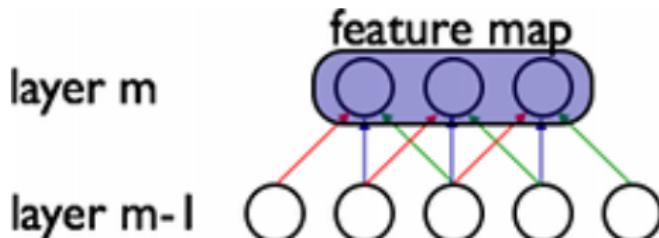
## To go beyond PCA

- We need a non-linear output function
  - If the output function is linear, and we have only a single hidden layer, we cannot beat PCA.
- 

## Caveat Emptor

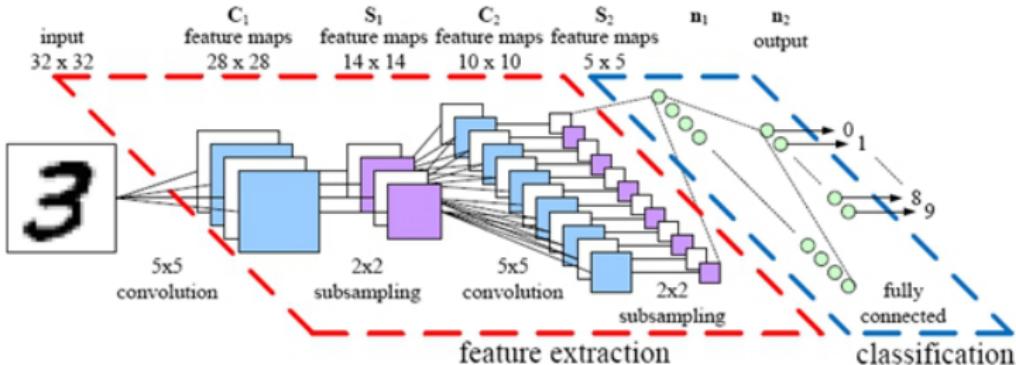
- Consider a situation in which we have  $N$  units in the bottom layer. Unit  $i$  will receive activation  $a_i^n$  for datapoint  $\mathbf{x}^n$ .
- Define the value of the bottom hidden layer to be  $\sigma_{L-1}(x_i) = \mathbb{I}[x_i - a_i^i]$  (this can be effectively achieved using a sigmoid function). This means that when vector  $\mathbf{x}^n$  is inputted, bottom hidden unit  $n$  is 1 and all others are zero.
- We can then perfectly reconstruct  $\mathbf{x}^n$  by using a linear output  $\sigma_L(x) = x$  with weights coming out of hidden unit  $n$  to be equal to  $\mathbf{x}^n$ .
- Hence, even for a bottleneck with dimension 1, we can perfectly reconstruct the input for  $N$  training patterns provided that we have at least  $N$  units in the bottom hidden layer.

# Convolutional NNs



- Consider that the input to the net is an image  $x$  and that we wish to detect if there is a bicycle in the image.
- We want the detector to have translation invariance, so that no matter where the bicycle is in the image, the net will recognise it.
- We can then imagine a small 'bicycle detector' node that will look at a local set of pixels in the image and respond strongly if it sees a bicycle.
- We then replicate this detector across all positions in the image.
- This generates the 'feature' map in the above diagram in which weights with the same colour are shared.
- We can then add an additional 'max pooling' layer on top of this so that if any of the feature map neurons responds strongly, then we know there is a bicycle somewhere in the image. This can be considered a form of 'sub-sampling'.

# Convolutional NNs



- CNNs are particularly popular in image processing
- Often the feature maps correspond (not to macro features such as bicycles) but micro features.
- For example, in handwritten digit recognition they correspond to small constituent parts of the digits.
- These are used then to process the image into a representation that is better for recognition.

# NNs in NLP

## Bag of Words

- We have  $D$  words in a dictionary, aardvark, ..., zorro so that we can relate each word with its dictionary index.
- We can also think of this as a Euclidean embedding  $e$ :

$$\text{aardvark} \rightarrow e_{\text{aardvark}} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \text{zorro} \rightarrow e_{\text{zorro}} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}$$

## Word Embeddings

- Idea is to replace the Euclidean embeddings  $e$  with embeddings (vectors)  $v$  that are *learned*.
- Objective is, for example, next word prediction accuracy.
- These are often called 'neural language models'.

## NNs in NLP

- Each word  $w$  in the dictionary has an associated embedding vector  $\mathbf{v}_w$ .  
Usually around 200 dimensional vectors are used.
- Consider the sentence:

the cat sat on the mat

and that we wish to predict the word on given the two preceding cat sat  
and two succeeding words the mat

- We can use a network that has inputs  $\mathbf{v}_{\text{cat}}, \mathbf{v}_{\text{sat}}, \mathbf{v}_{\text{the}}, \mathbf{v}_{\text{mat}}$
- The output of the network is a probability over all words in the dictionary  $p(w | \{\mathbf{v}_{\text{inputs}}\})$ .
- We want  $p(w = \text{on} | \mathbf{v}_{\text{cat}}, \mathbf{v}_{\text{sat}}, \mathbf{v}_{\text{the}}, \mathbf{v}_{\text{mat}})$  to be high.
- The overall objective is then to learn all the word embeddings and network parameters subject to predicting the word correctly based on the context.
- Note that one can also think about this as  $\mathbf{v}_w = \mathbf{E}\mathbf{e}_w$  where the embedding vectors are the rows of the embedding matrix  $\mathbf{E}$ . Thus this is a NN with Euclidean (one-of- $D$ ) inputs and first layer matrix  $\mathbf{E}$ .

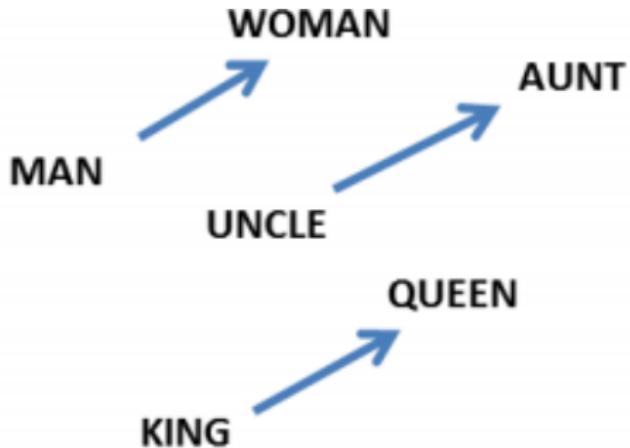
# Word Embeddings

FRANCE	JESUS	XBOX	REDDISH	SCRATCHED	MEGABITS
AUSTRIA	GOD	AMIGA	GREENISH	NAILED	OCTETS
BELGIUM	SATI	PLAYSTATION	BLUISH	SMASHED	MB/S
GERMANY	CHRIST	MSX	PINKISH	PUNCHED	BIT/S
ITALY	SATAN	IPOD	PURPLISH	POPPED	BAUD
GREECE	KALI	SEGA	BROWNISH	CRIMPED	CARATS
SWEDEN	INDRA	PSNUMBER	GREYISH	SCRAPED	KBIT/S
NORWAY	VISHNU	HD	GRAYISH	SCREWED	MEGAHERTZ
EUROPE	ANANDA	DREAMCAST	WHITISH	SECTIONED	MEGAPIXELS
HUNGARY	PARVATI	GEFORCE	SILVERY	SLASHED	GBIT/S
SWITZERLAND	GRACE	CAPCOM	YELLOWISH	RIPPED	AMPERES

Given a word (France, for example) we can find which words  $w$  have embedding vectors closest to  $\mathbf{v}_{\text{France}}$ . From Ronan Collobert (2011).

## Word Embeddings

There appears to be a natural ‘geometry’ to the embeddings. For example, there are directions that correspond to gender.



$$\mathbf{v}_{\text{woman}} - \mathbf{v}_{\text{man}} \approx \mathbf{v}_{\text{aunt}} - \mathbf{v}_{\text{uncle}}$$

$$\mathbf{v}_{\text{woman}} - \mathbf{v}_{\text{man}} \approx \mathbf{v}_{\text{queen}} - \mathbf{v}_{\text{king}}$$

From Mikolov (2013).

## Word Embeddings: Analogies

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Given a relationship, France-Paris, we get the 'relationship' embedding

$$\mathbf{v} = \mathbf{v}_{\text{Paris}} - \mathbf{v}_{\text{France}}$$

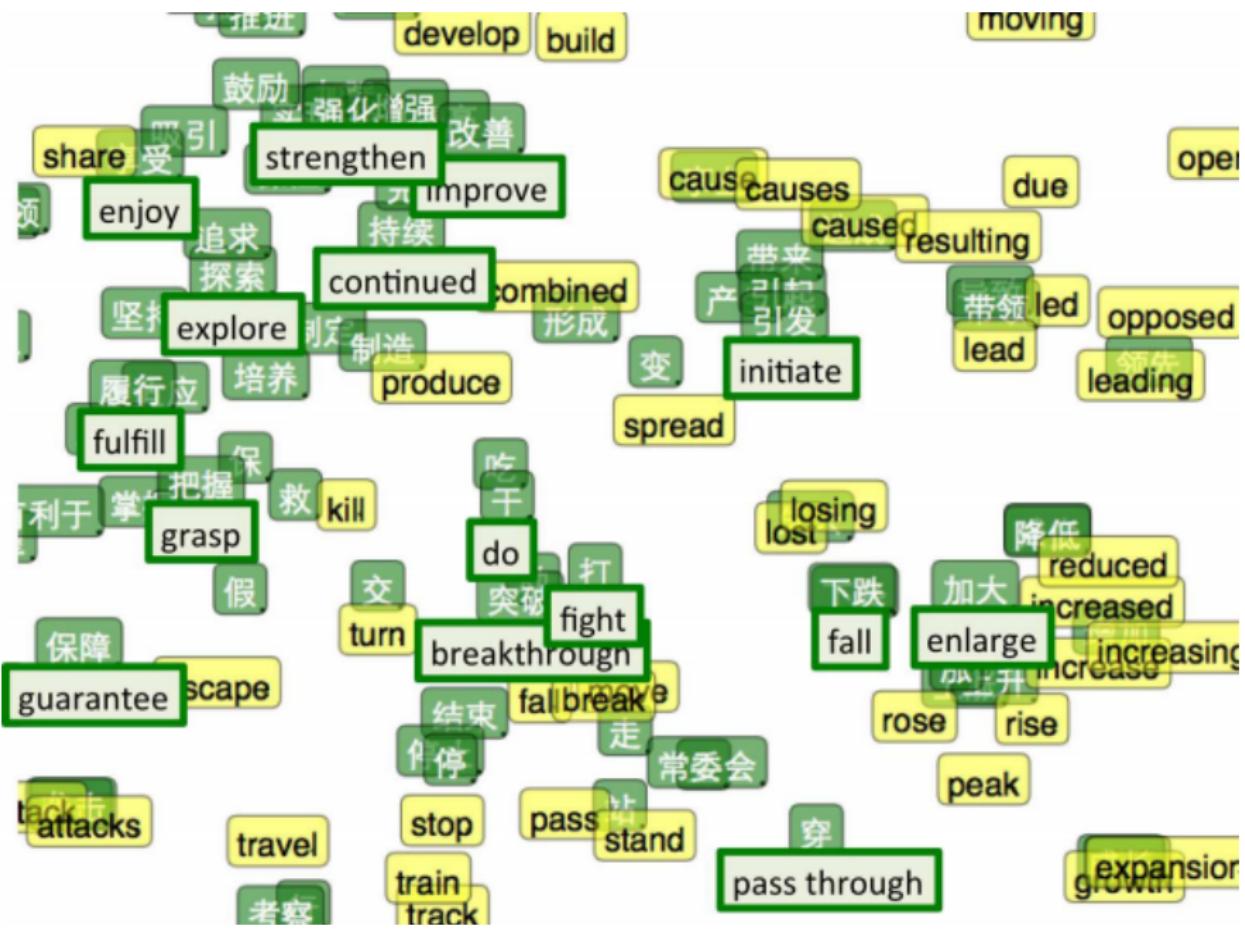
Given Italy we can calculate  $\mathbf{v}_{\text{Italy}} + \mathbf{v}$  and find the word in the dictionary which has closest embedding to this (it turns out to be Rome!). From Mikolov (2013).

# Word Embeddings: Constrained Embeddings

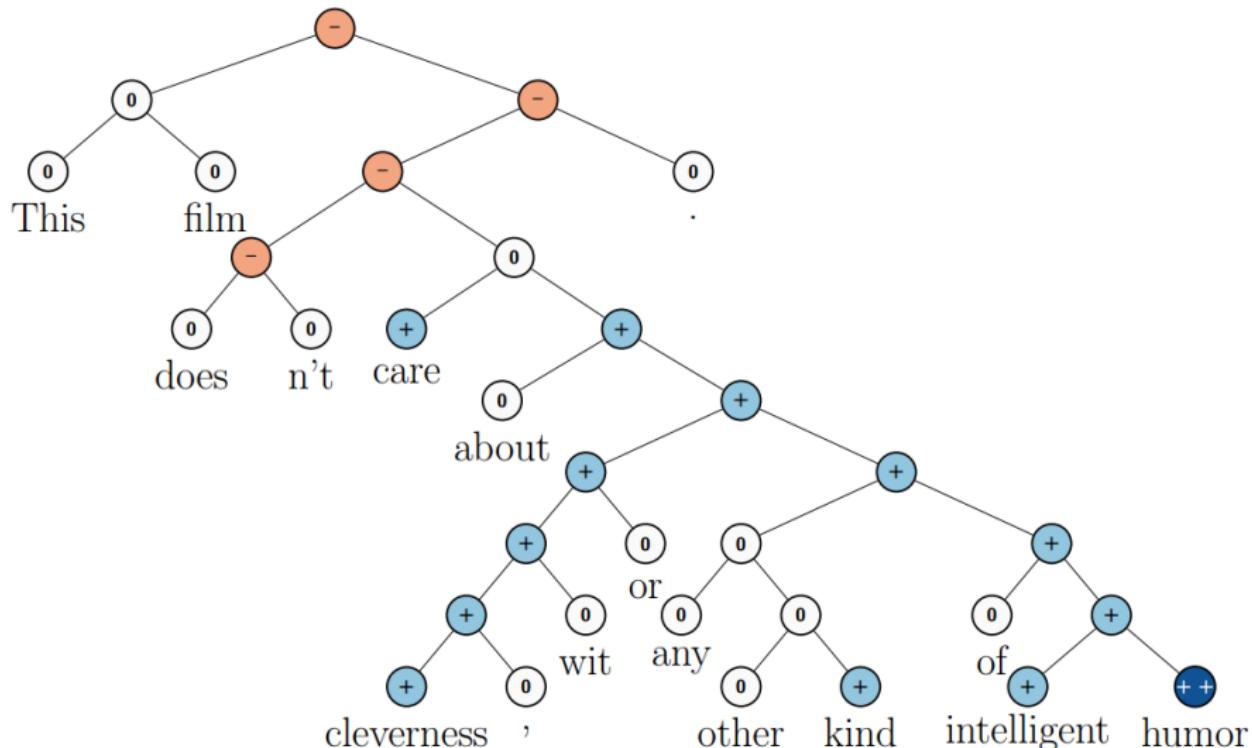


- We can learn embeddings for English words and embeddings for Chinese words.
- However, when we know that a Chinese and English word have a similar meaning, we add a constraint that the word embeddings  $\mathbf{v}_{\text{ChineseWord}}$  and  $\mathbf{v}_{\text{EnglishWord}}$  should be close.
- We have only a small amount of labelled ‘similar’ Chinese–English words (these are the green border boxes in the above; they are standard translations of the corresponding Chinese character).
- We can visualise in 2D (using t-SNE) the embedding vectors. See Socher (2013)

# Word Embeddings: Constrained Embeddings

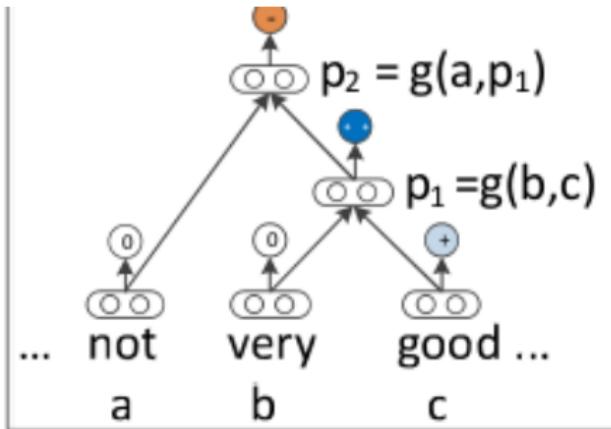


# Recursive Nets and Embeddings



Stanford Sentiment Treebank. Consists of parsed sentences with sentiment labels (--, -, 0, +,++) for each node (phrase) in the tree. 215,000 labelled phrases (obtained from three humans).

## Recursive Nets and Embeddings



- Idea is to recursively combine embeddings such that they accurately predict the sentiment at each node.

# Recursive Nets and Embeddings

## Training

- We have a softmax classifier for each node in the tree, to predict the sentiment of the phrase beneath this node in the tree.
- The weights of this classifier are shared across all nodes.
- At the leaf nodes at the bottom of the tree, the inputs to the classifiers are the word embeddings.
- The embeddings are combined by another network  $g$  with common parameters, which forms the input to the sentiment classifier.
- We then learn all the embeddings, shared classifier parameters and shared combination parameters to maximise the classification accuracy.

---

## Prediction

- For a new movie review, the review is first parsed using a standard grammar tree parser.
- This forms the tree which can be used to recursively form the sentiment class label for the review.
- Currently the best sentiment classifier. Socher (2013)

# Recursive Nets and Embeddings

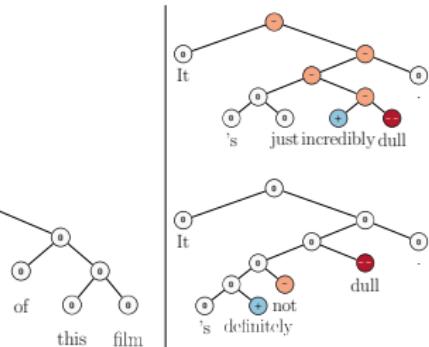
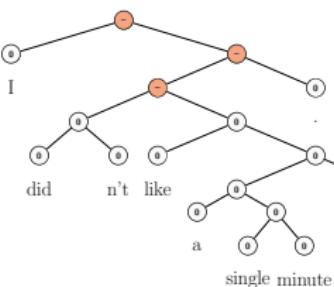
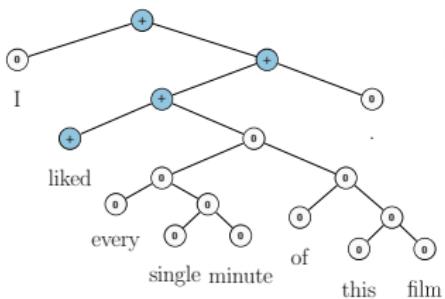
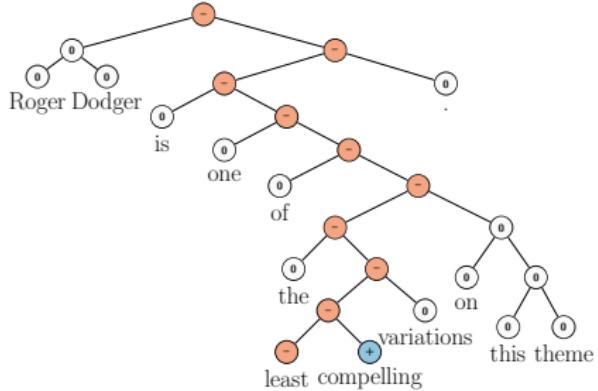
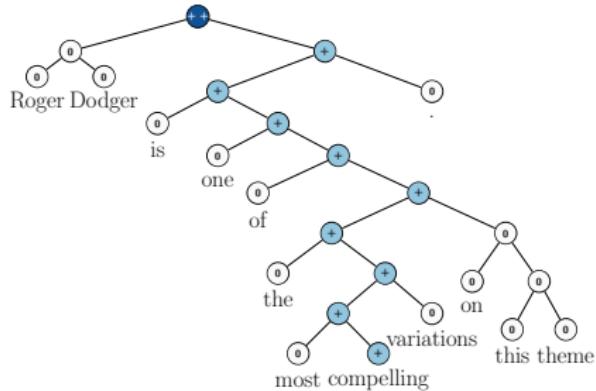
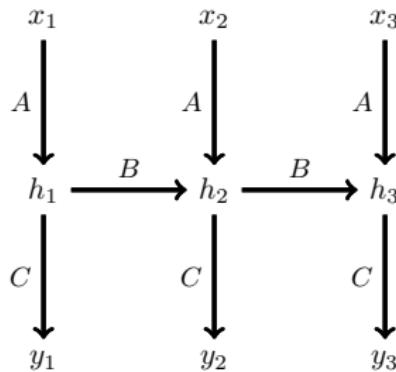


Figure 9: RNTN prediction of positive and negative (bottom right) sentences and their negation.

# Time Series Models

## Recurrent Nets



- RNNs are used in timeseries applications
- The basic idea is that the hidden units at time  $h_t$  (and possibly output  $y_t$ ) depend on the previous state of the network  $h_{t-1}, x_{t-1}, y_{t-1}$  for inputs  $x_t$  and outputs  $y_t$ .
- In the above network, I ‘unrolled the net through time’ to give a standard NN diagram.
- I omitted the potential links from  $x_{t-1}, y_{t-1}$  to  $h_t$ .

## Handwriting Generation using a RNN

would find the bus safe and sound  
As for Clark, unless it were a  
canvass at the ages of fifty-five  
Editorial. Dilemma of  
the the tides in the affairs of men;

Some training examples.

## Handwriting Generation using a RNN

- \* more of national temperament
- more of national temperament

Some generated examples. Top line is real handwriting, for comparison. See Alex Grave's work.

# Computing the Gradient

For a squared loss, we might have an objective of the form

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

- The output of the network at time  $t$  is some function  $f$  (parameterised by a matrix  $C$ ) of the hidden state  $h_t$ .
- The hidden value at time  $t$  is some function  $g$  (parameterised by input to hidden weights  $A$  and hidden to hidden weights  $B$ ) of the input  $x_t$  and previous hidden value  $h_{t-1}$ .
- To train a recurrent network we need to calculate the gradient with respect to  $A, B, C$ . There are several ways to do this with varying storage and time performances, see Williams and Zipser (1995) for a detailed comparison.
- We will discuss two methods below: RTRL (which is a single forward pass in time but has high storage cost) and BPTT (which is a forward and backward pass with more modest storage cost). RTRL is straightforward, but BPTT requires an understanding of parameter tying (or more generally AutoDiff).

## Real Time Recurrent Learning (Williams and Zipser)

Consider a recurrent network with parameters  $\theta$  and (for example) a squared loss:

$$E(\theta) = \frac{1}{2} \sum_t (y_t - f_\theta(h_t))^2, \quad h_t = g_\theta(x_t, h_{t-1})$$

Then

$$\frac{dE}{d\theta} = - \sum_t (y_t - f_\theta(h_t)) \frac{df_\theta(h_t)}{d\theta},$$

where

$$\frac{df_\theta(h_t)}{d\theta} = \frac{\partial f_\theta(h_t)}{\partial \theta} + \frac{\partial f_\theta(h_t)}{\partial h_t} \frac{dh_t}{d\theta}$$

and we can use the recursion:

$$\frac{dh_t}{d\theta} = \frac{\partial g_\theta(x_t, h_{t-1})}{\partial \theta} + \frac{\partial g_\theta(x_t, h_{t-1})}{\partial h_{t-1}} \frac{dh_{t-1}}{d\theta}$$

Hence the gradient of an RNN can be computed using a single forward pass in time. However, this has very high storage cost since for a network with  $H$  units,  $\frac{dh_t}{d\theta}$  would have  $H^3$  elements. For all but special cases, this will be impractical.

## Parameter Tying

- Consider a simple objective such as

$$E(\theta) = [y - f(\theta g(x\theta))]^2$$

- As a network diagram, this would look something like this  $x \xrightarrow{\theta} h \xrightarrow{\theta} y$  in which the parameters from the input to hidden layer and hidden layer to output are tied.
- We can calculate the gradient directly using the usual chain rule of calculus:

$$\frac{dE}{d\theta} = -2 [y - f(\theta g(x\theta))] f'(\theta g(x\theta)) (\theta g'(x\theta)x + g(x\theta))$$

where  $f'$  and  $g'$  denote the derivatives of  $f$  and  $g$  respectively.

## Parameter Tying

Another way to do this is to consider

$$F(\theta_1, \theta_2) = [y - f(\theta_1 g(x\theta_2))]^2$$

which is a Network with unconstrained parameters,  $x \xrightarrow{\theta_1} h \xrightarrow{\theta_2} y$ . Then

$$\frac{dF}{d\theta} = \frac{\partial F}{\partial \theta_1} \frac{\partial \theta_1}{\partial \theta} + \frac{\partial F}{\partial \theta_2} \frac{\partial \theta_2}{\partial \theta}$$

If we now constrain  $\theta_1 = \theta_2 = \theta$ , we obtain

$$\frac{dE}{d\theta} = \frac{dF(\theta, \theta)}{d\theta} = \left. \frac{\partial F(\theta_1, \theta_2)}{\partial \theta_1} \right|_{\theta_1=\theta_2=\theta} + \left. \frac{\partial F(\theta_1, \theta_2)}{\partial \theta_2} \right|_{\theta_1=\theta_2=\theta}$$

where  $|_{\theta_1=\theta_2=\theta}$  means that we evaluate the resulting expression (after calculating the partial derivative) at the constrained values.

## Parameter Tying

We can verify that this works:

$$\frac{\partial F(\theta_1, \theta_2)}{\partial \theta_1} = -2 [y - f(\theta_1 g(x\theta_2))] f'(\theta_1 g(x\theta_2))g(x\theta_2)$$

and

$$\frac{\partial F(\theta_1, \theta_2)}{\partial \theta_2} = -2 [y - f(\theta_1 g(x\theta_2))] f'(\theta_1 g(x\theta_2))\theta_1 x g'(x\theta_2)$$

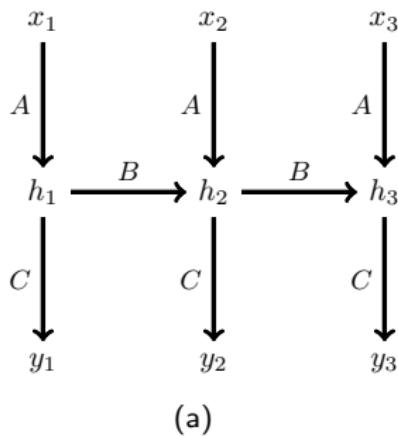
Summing these two and evaluating at  $\theta_1 = \theta_2 = \theta$ , we obtain the correct result.

## Parameter Tying

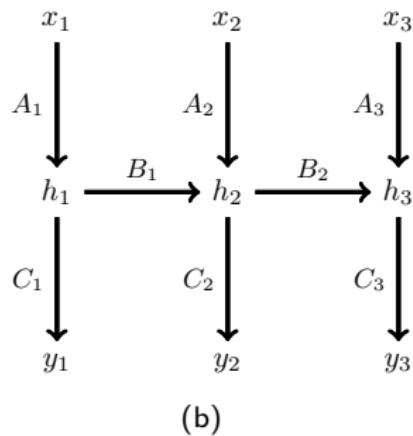
The conclusion is that we can deal with parameter tying in any objective by the following procedure:

1. Treat all parameters as independent and calculate the gradient with respect to each independent parameter.
  2. Sum all the resulting independent gradients together.
  3. Evaluate the expression by setting all the independent parameters to the same value.
- Note that this is a general result and can be used to deal with parameter tying in any objective, not just Deep Learning and Neural Nets.
  - (Of course, this is again just a special case of AutoDiff).

# Backprop Through Time



(a)



(b)

**Figure:** (a): A recurrent Neural Net here written for only 3 timesteps. (b): Unconstrained version of (a) used to derive BPTT.

## Backprop Through Time

- Perhaps the most obvious approach is to directly calculate the gradient by a forward-propagation algorithm (RRTL).
- A more common approach is Backprop Through Time (BPTT).
- Given our understanding about parameter tying, we see that we can calculate the gradient of the recurrent NN objective by first treating all parameters as independent, and then running standard backprop on the resulting architecture.
- Finally, we simply sum all the corresponding gradients (for example with respect to the matrices  $A_1, A_2, A_3$  to calculate the gradient with respect to  $A$ ) and subsequently set the parameters to be equal.
- This is an efficient exact algorithm which, in contrast to RRTL, runs backwards in time.
- Despite having efficient algorithms to compute the gradient, training RNNs is considered particularly challenging and either specialised optimisation procedures, or modifications to the standard architecture (such as LSTM), are usually required.

# Generative Models

## Generative Models

- Generative models parameterise a joint distribution in the form  $p(v, h) = p(v|h)p(h)$  where  $v$  are observable ‘visible’ variables and  $h$  latent ‘hidden’ variables.
- The term  $p(v|h)$  expresses how the observations in the world are generated according to known ‘laws’. Generative models are very common in the natural sciences.
- From a model, we can then form  $p(h|v)$  to understand what latent state likely generated the observation.
- Generative models are also popular in machine learning since they can be used to learn structure in unlabelled data (unsupervised learning).
- Much (most) of the information in the world is unlabelled and unsupervised methods that can learn structure are key to progress in AI.
- Generative models also enable us to construct (‘phantasise/hallucinate’) data, with needing an input. This is done by sampling  $h$  from  $p(h)$  and then  $v$  from  $p(v|h)$ .

## Variational Inference

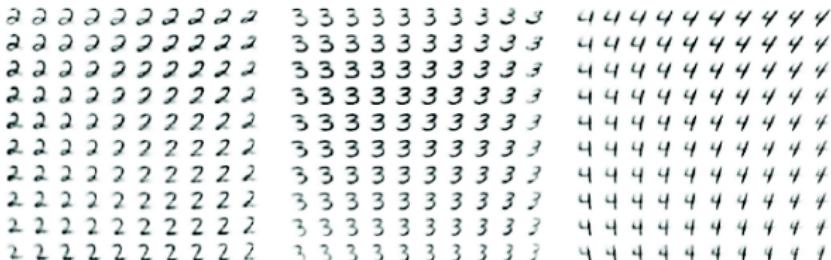
Consider a distribution

$$p(v|\theta) = \int_h p(v|h, \theta)p(h)$$

and that we wish to learn  $\theta$  to maximise the probability this model generates observed data.

$$\log p(v|\theta) \geq - \int q(h|v, \phi) \log q(h|v, \phi) + \int_h q(h|v, \phi)p(v|h, \theta) + \text{const.}$$

- Idea is to choose a ‘variational’ distribution  $q(h|v, \phi)$  such that we can either calculate analytically the bound, or sample it efficiently.
- We then jointly maximise the bound wrt  $\phi$  and  $\theta$ .
- We can parameterise  $p(v|h, \theta)$  using a deep network. Where exact expectation is intractable, sampling can be used.
- Very popular approach – see ‘variational autoencoder’ (a misnomer since this is not a variational form of an autoencoder – it’s a generative model).
- Extension to semi-supervised method using  $p(v) = \int_h \sum_c p(v|h, c)p(c)p(h)$



(a) Handwriting styles for MNIST obtained by fixing the class label and varying the 2D latent variable  $\mathbf{z}$



(b) MNIST analogies



(c) SVHN analogies

Figure 1: (a) Visualisation of handwriting styles learned by the model with 2D  $\mathbf{z}$ -space. (b,c) Analogical reasoning with generative semi-supervised models using a high-dimensional  $\mathbf{z}$ -space. The leftmost columns show images from the test set. The other columns show analogical fantasies of  $\mathbf{x}$  by the generative model, where the latent variable  $\mathbf{z}$  of each row is set to the value inferred from the test-set image on the left by the inference network. Each column corresponds to a class label  $y$ .

## Discussion

## Training Nets

- The surface  $E(\mathcal{W})$  is typically highly complex ●
- Apart from permutation symmetries (which cause local optima) there are typically many non-trivial local optima ●
- There are also typically large parts of the space in which the  $E$  surface is almost flat ●
- There is little real understanding about how to best train NNs ●
- The quality of the solution is often highly dependent on the initialisation of the weights ●
- The field is replete with initialisation and regularisation heuristics ●
- When well trained, these techniques provide state-of-the-art performance ●!

## Summary

Neural nets (deep learning) are complex hierarchical functions that are primarily motivated by analogies to information processing in natural systems.

- They are potentially very powerful function approximators ●
- Once trained, they are very fast to use ●
- They can be scaled up to very large datasets ●
- Training is complex – the objective function contains many local optima ●
- Parameter initialisation is difficult and critical, especially in networks with many layers ●
- Many things that can be adjusted – number of nodes in each layer, number of layers, types of transfer functions, regularisation types, penalty strengths ●
- Currently the state-of-the-art in several areas (object recognition, natural language processing, speech recognition) ●
- Super hot topic at the moment, with intense interest from big companies ●

# Automatic Differentiation

David Barber

## What is AutoDiff?

- AutoDiff takes a function  $f(\mathbf{x})$  and returns an exact value (up to machine accuracy) for the gradient

$$g_i(\mathbf{x}) \equiv \left. \frac{\partial}{\partial x_i} f \right|_{\mathbf{x}}$$

- Note that this is not the same as a numerical approximation (such as central differences) for the gradient.
- One can show that, if done efficiently, one can always calculate the gradient in less than 5 times the time it takes to compute  $f(\mathbf{x})$ .
- This is also *not* the same as symbolic differentiation.

## Symbolic Differentiation

- Given a function  $f(x) = \sin(x)$ , symbolic differentiation returns an algebraic expression for the derivative. This is not necessarily efficient since it may contain a great number of terms.
- As an (overly!) simple example, consider

$$f(x_1, x_2) = (x_1^2 + x_2^2)^2$$

$$\frac{\partial f}{\partial x_1} = 2(x_1^2 + x_2^2) 2x_1, \quad \frac{\partial f}{\partial x_2} = 2(x_1^2 + x_2^2) 2x_2$$

The algebraic expression is not computationally efficient. However, by defining  $y = 4(x_1^2 + x_2^2)$ ,

$$\frac{\partial f}{\partial x_1} = yx_1, \quad \frac{\partial f}{\partial x_2} = yx_2$$

Which is a more efficient *computational* expression.

- Also, more generally, we want to consider computational subroutines that contain loops and conditional if statements; these do not correspond to simple closed algebraic expressions. We want to find a corresponding subroutine that can return the exact derivative efficiently for such subroutines.

# Forward and Reverse Differentiation

## Forward

- This is (usually) easy to implement
  - However, it is not (generally) computationally efficient.
  - It cannot easily handle conditional statements or loops.
- 

## Reverse

- This is exact and computationally efficient.
- It is, however, harder to code and requires a parse tree of the subroutine.
- If possible, one should always attempt to do reverse differentiation.
- As we will discuss, the famous backprop algorithm is just a special case of reverse differentiation.
- Reverse differentiation is also important since, with it, one can understand (for example) how to deal easily with calculating the derivative of a function subject to parameter tying.

# Forward Differentiation

Consider  $f(x) = x^2$ .

---

Complex arithmetic

$$f(x + i\epsilon) = (x + i\epsilon)^2 = x^2 - \epsilon^2 + 2i\epsilon x$$

$$f'(x) = \lim_{\epsilon \rightarrow 0} \frac{1}{\epsilon} \operatorname{Im}(f(x + i\epsilon))$$

- This also holds for any smooth function (one that can be expressed as a Taylor series).
- For finite  $\epsilon$  this gives an *approximation* only.
- More accurate approximation than standard finite differences since we do not subtract two small quantities and divide by a small quantity – the complex arithmetic approach is more numerically stable.
- To implement, we need to overload all functions so that they can deal with complex arithmetic.

# Forward Differentiation

Consider  $f(x) = x^2$ .

---

## Dual arithmetic

Define an idempotent variable,  $\epsilon$  such that  $\epsilon^2 = 0$ .

$$f(x + \epsilon) = (x + \epsilon)^2 = x^2 + 2x\epsilon$$

Hence

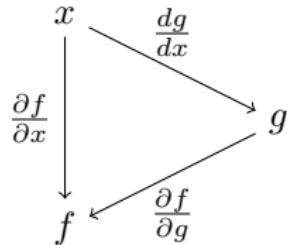
$$f'(x) = \text{DualPart}f(x + \epsilon)$$

- This holds for any smooth function  $f(x)$  and non-zero value of  $\epsilon$ .
- Need to overload every function in the subroutine to work in dual arithmetic.
- Numerically exact.
- Whilst exact, this is not necessarily efficient.

## Reverse Differentiation

A useful graphical representation is that the total derivative of  $f$  with respect to  $x$  is given by the sum over all path values from  $x$  to  $f$ , where each path value is the product of the partial derivatives of the functions on the edges:

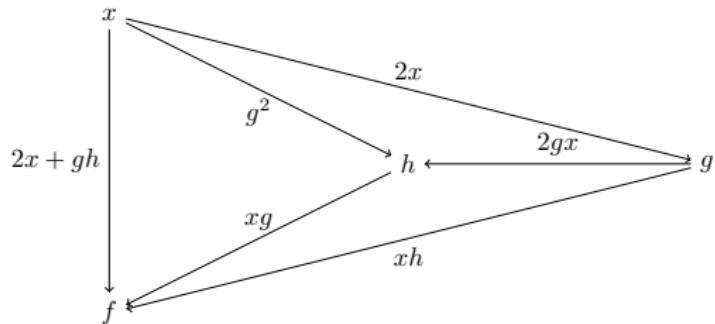
$$\frac{df}{dx} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial g} \frac{dg}{dx}$$



---

### Example

For  $f(x) = x^2 + xgh$ , where  $g = x^2$  and  $h = xg^2$



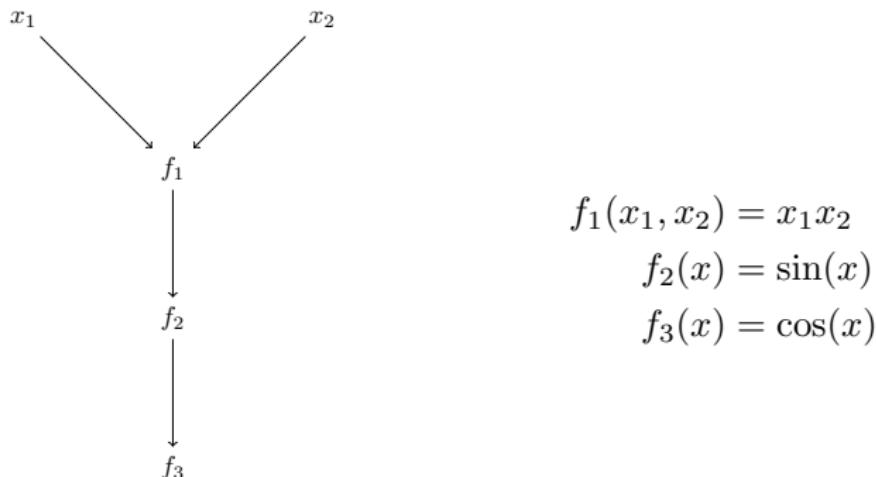
$$f'(x) = (2x + gh) + (g^2 xg) + (2x 2gx xg) + (2xxh) = 2x + 8x^7$$

# Reverse Differentiation

Consider

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

We can represent this computationally using an Abstract Syntax Tree (AST):



Given values for  $x_1, x_2$ , we first run forwards through the tree so that we can associate each node with an actual function value.

## Reverse Differentiation

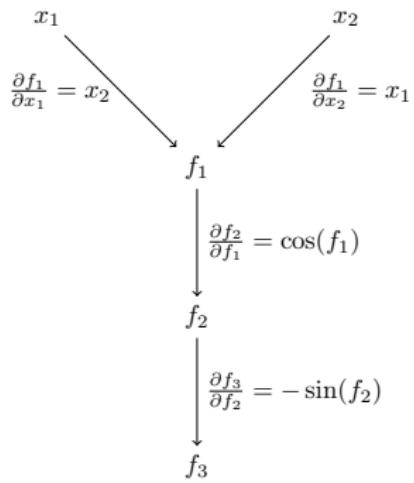
$$\begin{array}{ccc} x_1 & & x_2 \\ \searrow & & \swarrow \\ f_1 & & \\ \downarrow & & \\ f_2 & & \\ \downarrow & & \\ f_3 & & \end{array}$$
$$\frac{df_3}{dx_1} = \frac{\partial f_3}{\partial f_2} \frac{df_2}{dx_1} = \underbrace{\frac{\partial f_3}{\partial f_2} \frac{df_2}{df_1}}_{\frac{df_3}{df_1}} \frac{df_1}{dx_1}$$

Similarly,

$$\frac{df_3}{dx_2} = \underbrace{\frac{\partial f_3}{\partial f_2} \frac{df_2}{df_1}}_{\frac{df_3}{df_1}} \frac{df_1}{dx_2}$$

The two derivatives share the same computation branch and we want to exploit this.

# Reverse Differentiation



1. Find the reverse ancestral (backwards) schedule of nodes  $(f_3, f_2, f_1, x_1, x_2)$ .
2. Start with the first node  $n_1$  in the reverse schedule and define  $t_{n_1} = 1$ .
3. For the next node  $n$  in the reverse schedule, find the child nodes  $\text{ch}(n)$ . Then define
$$t_n = \sum_{c \in \text{ch}(n)} \frac{\partial f_c}{\partial f_n} t_c$$
4. The total derivatives of  $f$  with respect to the root nodes of the tree (here  $x_1$  and  $x_2$ ) are given by the values of  $t$  at those nodes.

This is a general procedure that can be used to automatically define a subroutine to efficiently compute the gradient. It is efficient because information is collected at nodes in the tree and split between parents only when required.

## Hessian-Vector product

- Consider a function  $E(\theta)$  and its Hessian

$$H_{ij} \equiv \frac{\partial^2 E}{\partial \theta_i \partial \theta_j}$$

- In the Newton optimisation method, we update a vector  $\theta$  by the inverse Hessian times the gradient of the objective:

$$\mathbf{x} = \mathbf{H}^{-1} \mathbf{g}$$

- In practice, we find the update  $\mathbf{x}$  by solving the linear system

$$\mathbf{Hx} = \mathbf{g}$$

typically by conjugate gradients.

- For a neural net with  $W$  parameters, just storing the Hessian takes  $O(W^2)$  space and computing the Hessian-vector product  $O(W^2)$  time. This is too expensive.
- Magically, there is a way to compute a Hessian-vector product in  $O(W)$  time and space!

## Hessian-Vector product

- Define the Directional Derivative in the direction  $\mathbf{v}$  as

$$D_{\mathbf{v}}(f) \equiv \lim_{\delta \rightarrow 0} \frac{f(\mathbf{x} + \delta\mathbf{v}) - f(\mathbf{x})}{\delta} = \sum_i v_i \frac{\partial f}{\partial x_i} = \mathbf{v}^T \nabla f$$

This is a scalar value and represents the rate of change of the function along the direction  $\mathbf{v}$ .

- This is a derivative and so all usual rules (chain rule etc) apply as well.
- 

$$\begin{aligned} D_{\mathbf{v}} \left( \frac{\partial E}{\partial \theta_i} \right) &= \lim_{\delta \rightarrow 0} \frac{\frac{\partial E}{\partial \theta_i}(\theta + \delta\mathbf{v}) - \frac{\partial E}{\partial \theta_i}}{\delta} \\ &= \lim_{\delta \rightarrow 0} \frac{\frac{\partial E}{\partial \theta_i} + \delta \sum_j v_j \frac{\partial^2 E}{\partial \theta_i \partial \theta_j} + O(\delta^2) - \frac{\partial E}{\partial \theta_i}}{\delta} \\ &= [\mathbf{H}\mathbf{v}]_i \end{aligned}$$

where  $\mathbf{H}$  is the Hessian.

- Hence, we can find Hessian-vector products by taking the directional derivative of the gradient in the direction of the vector.

## Hessian-Vector product example

- For the function

$$E(\theta) = (x^T \theta)^2$$

the Hessian is

$$\mathbf{H} = 2xx^T$$

Hence,

$$\mathbf{H}\mathbf{v} = 2(x^T \mathbf{v}) \mathbf{x}$$

- We want to see how to calculate this efficiently within the autodiff framework
- According to our approach, we first need to get the gradient which we will do using the usual forward and backward reverse autodiff equations.

## Hessian-Vector product example

For the function

$$f(\theta) = (\mathbf{x}^\top \theta)^2$$

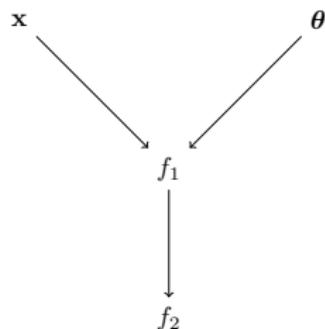
The nodes of the computation graph are

$$\theta$$

$$\mathbf{x}$$

$$f_1 = \theta^\top \mathbf{x}$$

$$f_2 = (f_1)^2$$



## Hessian-Vector product example

- If we run reverse Autodiff, this defines the equations

$$t_2 = 1$$

$$t_1 = \frac{\partial f_2}{\partial f_1} t_2$$

$$t_\theta = \frac{\partial f_1}{\partial \theta} t_1$$

According to our theory, the Hessian-vector product is given by the directional derivative of the gradient, namely  $D_{\mathbf{v}}(t_\theta)$ . We can calculate this as follows:

$$D_{\mathbf{v}}(t_2) = 0$$

$$D_{\mathbf{v}}(t_1) = \frac{\partial f_2}{\partial f_1} D_{\mathbf{v}}(t_2) + t_2 D_{\mathbf{v}}\left(\frac{\partial f_2}{\partial f_1}\right)$$

$$D_{\mathbf{v}}(t_\theta) = \frac{\partial f_1}{\partial \theta} D_{\mathbf{v}}(t_1) + t_1 D_{\mathbf{v}}\left(\frac{\partial f_1}{\partial \theta}\right)$$

## Hessian-Vector product example

- Let's look at some of these terms:

$$\begin{aligned} D_{\mathbf{v}} \left( \frac{\partial f_2}{\partial f_1} \right) &= \sum_i v_i \frac{\partial^2 f_2}{\partial f_1 \partial \theta_i} = \sum_i v_i \frac{\partial^2 f_2}{\partial f_1^2} \frac{\partial f_1}{\partial \theta_i} = \frac{\partial^2 f_2}{\partial f_1^2} \sum_i v_i \frac{\partial f_1}{\partial \theta_i} \\ &= \frac{\partial^2 f_2}{\partial f_1^2} D_{\mathbf{v}}(f_1) \end{aligned}$$

Similarly

$$D_{\mathbf{v}} \left( \frac{\partial f_1}{\partial \theta} \right) = \frac{\partial^2 f_1}{\partial \theta^2} D_{\mathbf{v}}(\theta)$$

- Hence we can relate directional derivatives of gradients to directional derivatives of the nodes.
- We then just need to calculate the node directional derivatives, which we can do by going back to the equations.

## Hessian-Vector product example

- We can find the directional derivative of each node by a forward pass through each of the nodes:
- 

$$D_{\mathbf{v}}(\boldsymbol{\theta}) = \mathbf{v}$$

$$D_{\mathbf{v}}(\mathbf{x}) = \mathbf{0}$$

$$D_{\mathbf{v}}(f_1) = \mathbf{v}^T \mathbf{x}$$

$$D_{\mathbf{v}}(f_2) = 2f_1 D_{\mathbf{v}}(f_1)$$

- We now have everything we need to get the Hessian-vector product. Using the reverse pass, with results computed from the two forward passes and the gradient reverse pass. Using

$$D_{\mathbf{v}}(t_1) = 2\mathbf{v}^T \mathbf{x}, \quad D_{\mathbf{v}}\left(\frac{\partial f_1}{\partial \theta}\right) = 0$$

we get

$$D_{\mathbf{v}}(t_\theta) = \mathbf{x} D_{\mathbf{v}}(t_1) = 2(\mathbf{v}^T \mathbf{x}) \mathbf{x}$$

which is the correct result.

## Hessian-vector product

If each node  $f_i$  is a function of its parent nodes  $\pi_1^i, \dots$ , there are then 4 passes required:

Standard forward pass: With  $i$  indexing a forward ordering calculate the values of the nodes:

$$f_i = f_i(\pi_1^i, \dots, \pi_k^i)$$

Standard Reverse Pass: With  $n$  indexing a reverse ordering:

$$t_n = \sum_{c \in \text{ch}(n)} \frac{\partial f_c}{\partial f_n} t_c$$

Directional Derivative Forward Pass:

$$D_{\mathbf{v}}(f_i) = \sum_k \frac{\partial f_i}{\partial \pi_k^i} D_{\mathbf{v}}(\pi_k^i)$$

Directional Derivative Reverse Pass:

$$D_{\mathbf{v}}(t_n) = \sum_{c \in \text{ch}(n)} \left[ \frac{\partial f_c}{\partial f_n} D_{\mathbf{v}}(t_c) + t_c \frac{\partial^2 f_c}{\partial f_n^2} D_{\mathbf{v}}(f_n) \right]$$

## Gauss-Newton-vector product

We can write the loss as a function of the output  $y = N(\theta)$  of the network

$$E(\theta) = L(N(\theta))$$

Then

$$\frac{\partial E}{\partial \theta_i} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial \theta_i}$$

and

$$\frac{\partial^2 E}{\partial \theta_i \partial \theta_j} = \frac{\partial L}{\partial y} \frac{\partial^2 y}{\partial \theta_i \partial \theta_j} + \frac{\partial y}{\partial \theta_i} \frac{\partial^2 L}{\partial y^2} \frac{\partial y}{\partial \theta_j}$$

Provided the loss  $L$  is a convex function of the output of the network (which is true for squared loss with a linear output and also for log loss with softmax output) then, by ignoring the first term we can definite a positive semidefinite Gauss-Newton matrix

$$G_{ij} \equiv \frac{\partial y}{\partial \theta_i} \frac{\partial^2 L}{\partial y^2} \frac{\partial y}{\partial \theta_j}$$

and

$$\sum_j G_{ij} v_j = \frac{\partial y}{\partial \theta_i} \sum_j \frac{\partial^2 L}{\partial y^2} \frac{\partial y}{\partial \theta_j} v_j = \frac{\partial y}{\partial \theta_i} \sum_j \frac{\partial}{\partial \theta_j} \left( \frac{\partial L}{\partial y} \right) v_j = \frac{\partial y}{\partial \theta_i} D_{\mathbf{v}}(t_y)$$

This means that we can efficiently compute a Gauss-Newton-vector product by only doing the reverse directional derivative pass up to the network output layer.

---

### Net with multiple outputs

- If we have multiple outputs  $y_k$ , then the above becomes

$$G_{ij} \equiv \sum_k \frac{\partial y_k}{\partial \theta_i} \frac{\partial^2 L}{\partial y_k^2} \frac{\partial y_k}{\partial \theta_j}$$

and

$$\sum_j G_{ij} v_j = \sum_k \frac{\partial y_k}{\partial \theta_i} D_{\mathbf{v}}(t_{ky})$$

- This seems like a problem since we need the derivative of each  $y_k$  wrt each input  $\theta_i$ .
- However, we note that  $D_{\mathbf{v}}(t_{ky}) \equiv \mu_k$  can be calculated by doing the reverse direction derivative up to the  $\mathbf{y}$  layer and then stored as known values  $\mu_k$ . To compute

$$\sum_j G_{ij} v_j = \sum_k \frac{\partial y_k}{\partial \theta_i} \mu_k$$

we note that we can consider this as the gradient wrt  $\theta$  of an ‘auxiliary’ loss function

$$\sum_k y_k \mu_k$$

We can calculate the gradient of this auxiliary loss function by simply using the standard reverse autodiff backpass using this loss. No additional forward pass is needed since the auxiliary loss and the original loss share the same layers up to layer  $\mathbf{y}$ .

## Software

- AutoDiff has been around a long time (since the 1960's).
- There are tons of tools out there with varying degrees of sophistication.
- The most efficient tools use special purpose optimisers to first obtain the most compact AST.
- Stan is a popular recent C++ tools from Stanford.
- Theano is a popular tool in python, developed by Montreal Machine Learners.
- TensorFlow is becoming a standard package (though still slower than Theano).

# Deep Learning: Autodiff, Parameter Tying and Backprop Through Time\*

David Barber  
Department of Computer Science  
University College London

February 9, 2015

## **Abstract**

How to do parameter tying and how this relates to Backprop through time.

## 1 Introduction

A common question when learning about Neural Nets (Deep Learning) is how to deal with parameter tying when calculating the gradient of the objective function. In particular, how does this relate to Backprop Through Time [3]? Whilst there is a large literature available on this, the explanations often seen (to my mind) rather over complicated. Below we explain how to deal with parameter tying in general (not just for Neural Nets).

## 2 Parameter Tying

Consider a simple objective such as

$$E(\theta) = [y - f(\theta g(x\theta))]^2 \tag{1}$$

One can think of this as a Neural Net squared loss objective corresponding to a single training input-output  $(x, y)$  with a scalar input  $x$ , single hidden layer  $h = g(x\theta)$ , with weight from input to hidden layer given by  $\theta$  and output  $f(\theta h)$ , with weight  $\theta$  from the hidden layer to the output layer. As a network diagram, this would look something like this  $x \xrightarrow{\theta} h \xrightarrow{\theta} y$  in which the parameters from the input to hidden layer and hidden layer to output are tied.

We can calculate the gradient directly using the usual chain rule of calculus:

$$\frac{\partial E}{\partial \theta} = -2[y - f(\theta g(x\theta))] f'(\theta g(x\theta)) (\theta g'(x\theta)x + g(x\theta)) \tag{2}$$

where  $f'$  and  $g'$  denote the derivatives of  $f$  and  $g$  respectively.

Another way to do this is to consider

$$F(\theta_1, \theta_2) = [y - f(\theta_1 g(x\theta_2))]^2 \tag{3}$$

which is a Network with unconstrained parameters,  $x \xrightarrow{\theta_1} h \xrightarrow{\theta_2} y$ . Then

$$\frac{\partial F}{\partial \theta} = \frac{\partial F}{\partial \theta_1} \frac{\partial \theta_1}{\partial \theta} + \frac{\partial F}{\partial \theta_2} \frac{\partial \theta_2}{\partial \theta} \tag{4}$$

---

\*UCL Department of Computer Science Technical Note

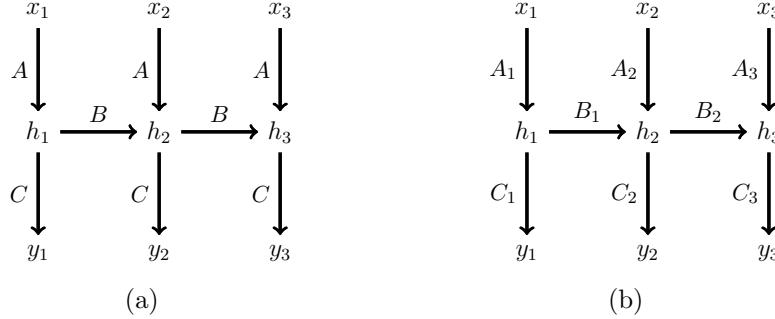


Figure 1: (a): A recurrent Neural Net here written for only 3 timesteps. (b): Unconstrained version of (a) used to derive BPTT.

If we now constrain  $\theta_1 = \theta_2 = \theta$ , we obtain

$$\frac{\partial E}{\partial \theta} = \frac{\partial F(\theta, \theta)}{\partial \theta} = \frac{\partial F(\theta_1, \theta_2)}{\partial \theta_1} \Big|_{\theta_1=\theta_2=\theta} + \frac{\partial F(\theta_1, \theta_2)}{\partial \theta_2} \Big|_{\theta_1=\theta_2=\theta} \quad (5)$$

where  $|_{\theta_1=\theta_2=\theta}$  means that we evaluate the resulting expression (after calculating the partial derivative) at the constrained values.

We can verify that this works:

$$\frac{\partial F(\theta_1, \theta_2)}{\partial \theta_1} = -2 [y - f(\theta_1 g(x\theta_2))] f'(\theta_1 g(x\theta_2))g(x\theta_2) \quad (6)$$

and

$$\frac{\partial F(\theta_1, \theta_2)}{\partial \theta_2} = -2 [y - f(\theta_1 g(x\theta_2))] f'(\theta_1 g(x\theta_2))\theta_1 x g'(x\theta_2) \quad (7)$$

Summing equation (6) and equation (7) and evaluating at  $\theta_1 = \theta_2 = \theta$ , we obtain equation (2).

The conclusion is that we can deal with parameter tying in any objective by the following procedure:

1. Treat all parameters as independent and calculate the gradient with respect to each independent parameter.
  2. Sum all the resulting independent gradients together.
  3. Evaluate the expression by setting all the independent parameters to the same value.
- Note that this is a general result and can be used to deal with parameter tying in any objective, not just Deep Learning and Neural Nets.

### 3 Backprop Through Time

A recurrent network can be thought of as a deterministic temporal model with inputs  $x_t$ , hidden values  $h_t$  and outputs  $y_t$ . All inputs, hidden values and outputs may be vectors. For a squared loss, we would have an objective of the form

$$E(A, B, C) = \sum_t (y_t - f(h_t; C))^2, \quad h_t = g(x_t, h_{t-1}; A, B) \quad (8)$$

which means that the output of the network at time  $t$  is some function  $f$  (parameterised by a matrix  $C$ ) of the hidden state  $h_t$ . Similarly, the hidden value at time  $t$  is some function  $g$  (parameterised by input to hidden weights  $A$  and hidden to hidden weights  $B$ ) of the input  $x_t$  and previous hidden value  $h_{t-1}$ , see fig(1a).

To train a recurrent network we need to calculate the gradient with respect to  $A$ ,  $B$ ,  $C$ . There are several ways to do this with varying storage and time performances, see [4] for a detailed comparison. Perhaps the most obvious approach is to directly calculate the gradient by a forward-propagation algorithm (RTRL) which is also the same technique discussed in [1] and extends recurrent networks to probabilistic models.

A more common approach is Backprop Through Time (BPTT) [3]. We assume that the reader is familiar with the standard backprop algorithm (see for example [2]). Given our understanding about parameter tying, we see that we can calculate the gradient of the recurrent NN objective by first treating all parameters as independent, and then running standard backprop on the resulting architecture, see fig(1b). Finally, we simply sum all the corresponding gradients (for example with respect to the matrices  $A_1$ ,  $A_2$ ,  $A_3$  to calculate the gradient with respect to  $A$ ) and subsequently set the parameters to be equal. This is an efficient exact algorithm which, in contrast to RTRL, runs backwards in time. In practice, to save on storage, backprop is typically run over a fixed window (rather than going back to time 1) which results in an approximation to the true gradient.

### 3.1 Conclusion

One can deal with parameter tying simply by treating parameters as if they are independent, and then summing all the corresponding gradients. This is particularly useful for objectives such as Neural Nets since one can then make use of standard (and efficient) backprop routines to calculate the unconstrained gradients.

## 4 AutoDiff

A more general viewpoint on backprop and parameter tying can be established through Automatic Differentiation. AutoDiff takes a function  $f(\mathbf{x})$  and returns an exact value (up to machine accuracy) for the gradient

$$g_i(\mathbf{x}) \equiv \left. \frac{\partial}{\partial x_i} f \right|_{\mathbf{x}} \quad (9)$$

Note that this is not the same as a numerical approximation (such as central differences) for the gradient. One can show that, if done efficiently, one can always calculate the gradient in less than 5 times the time it takes to compute  $f(\mathbf{x})$ . This is also *not* the same as symbolic differentiation.

In Symbolic Differentiation, given a function  $f(x) = \sin(x)$ , symbolic differentiation returns an algebraic expression for the derivative. This is not necessarily efficient since it may contain a great number of terms. As an (overly!) simple example, consider

$$f(x_1, x_2) = (x_1^2 + x_2^2)^2 \quad (10)$$

$$\frac{\partial f}{\partial x_1} = 2(x_1^2 + x_2^2) 2x_1, \quad \frac{\partial f}{\partial x_2} = 2(x_1^2 + x_2^2) 2x_2 \quad (11)$$

The algebraic expression is not computationally efficient. However, by defining

$$y = 4(x_1^2 + x_2^2) \quad (12)$$

then

$$\frac{\partial f}{\partial x_1} = yx_1, \quad \frac{\partial f}{\partial x_2} = yx_2 \quad (13)$$

Which is a more efficient *computational* expression. Also, more generally, we want to consider computational subroutines that contain loops and conditional *if* statements; these

do not correspond to simple closed algebraic expressions. We want to find a corresponding subroutine that can return the exact derivative efficiently for such subroutines. There are two main flavours of AutoDiff, namely Forward and Reverse mode.

### Forward

- This is (usually) easy to implement
- However, it is not (generally) computationally efficient.
- It cannot easily handle conditional statements or loops.

### Reverse

- This is exact and computationally efficient.
- It is, however, harder to code and requires a parse tree of the subroutine.
- If possible, one should always attempt to do reverse differentiation.
- As we will discuss, the famous backprop algorithm is just a special case of reverse differentiation.
- Reverse differentiation is also important since, with it, one can understand (for example) how to deal easily with calculating the derivative of a function subject to parameter tying.

## 4.1 Forward Differentiation

Consider  $f(x) = x^2$  and that we wish to compute the derivative of this.

### 4.1.1 Central Differences

The most well known approach to *approximating* a derivative is to use

$$f'(x) \approx \frac{f(x + \epsilon) - f(x - \epsilon)}{2\epsilon} = \frac{x^2 + 2\epsilon x + \epsilon^2 - x^2 + 2\epsilon x - \epsilon^2}{2\epsilon} \quad (14)$$

which in this case gives the exact result. More generally, the approximation is accurate up to order  $\epsilon^3$ . Whilst this can be useful, in addition to it only being an approximation, it is also potentially slow since, for vector  $x$ , the calculation has to be repeated for each component of the vector.

### 4.1.2 Complex arithmetic

$$f(x + i\epsilon) = (x + i\epsilon)^2 = x^2 - \epsilon^2 + 2i\epsilon x \quad (15)$$

Hence

$$f'(x) = \lim_{\epsilon \rightarrow 0} \frac{1}{\epsilon} \text{Im}(f(x + i\epsilon)) \quad (16)$$

This also holds for any smooth function (one that can be expressed as a Taylor series). For finite  $\epsilon$  this gives an *approximation* only. More accurate approximation than standard finite differences since we do not subtract two small quantities and divide by a small quantity – the complex arithmetic approach is more numerically stable. To implement, we need to overload all functions so that they can deal with complex arithmetic.

### 4.1.3 Dual arithmetic

Consider  $f(x) = x^2$ . Define an idempotent variable,  $\epsilon$  such that  $\epsilon^2 = 0$ .

$$f(x + \epsilon) = (x + \epsilon)^2 = x^2 + 2x\epsilon \quad (17)$$

Hence

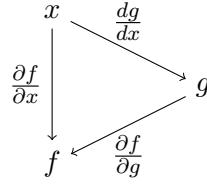
$$f'(x) = \text{DualPart}f(x + \epsilon) \quad (18)$$

This holds for any smooth function  $f(x)$  and non-zero value of  $\epsilon$ . To implement this we need to overload every function in the subroutine to work in dual arithmetic. Also note that this is not an approximation – it is an *exact* numerical computation of the derivative (up to machine accuracy). Whilst exact, this is, however, not necessarily efficient.

## 4.2 Reverse Differentiation

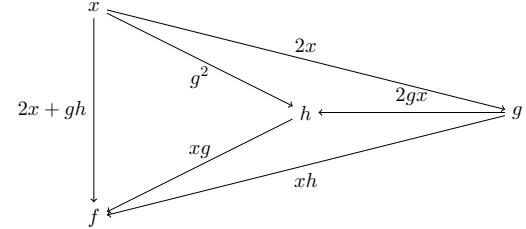
A useful graphical representation is that the total derivative of  $f$  with respect to  $x$  is given by the sum over all path values from  $x$  to  $f$ , where each path value is the product of the partial derivatives of the functions on the edges:

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



### Example 1

For  $f(x) = x^2 + xgh$ , where  $g = x^2$  and  $h = xg^2$



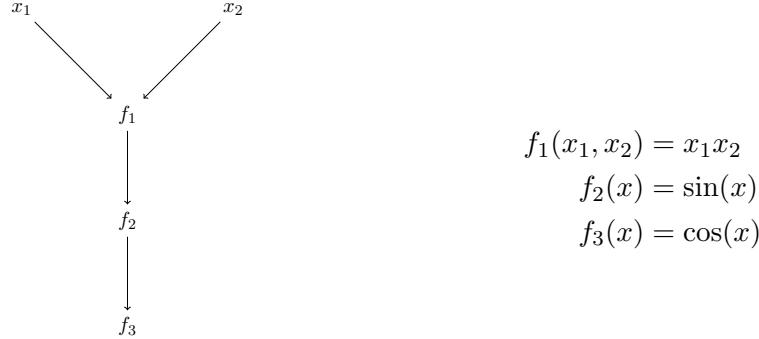
$$f'(x) = (2x + gh) + (g^2 xg) + (2x2gxxg) + (2xxh) = 2x + 8x^7 \quad (19)$$

### Example 2

Consider

$$f(x_1, x_2) = \cos(\sin(x_1 x_2)) \quad (20)$$

We can represent this computationally using an Abstract Syntax Tree (AST), also known as the Computation Tree:



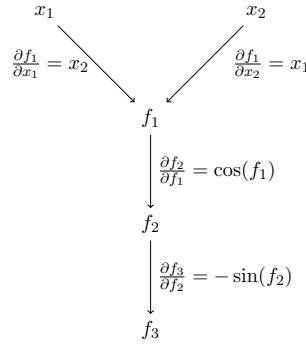
Given values for  $x_1, x_2$ , we first run forwards through the tree so that we can associate each node with an actual function value.

$$\frac{df_3}{dx_1} = \underbrace{\frac{\partial f_3}{\partial f_2} \frac{df_2}{dx_1}}_{\frac{df_3}{df_1}} = \underbrace{\frac{\partial f_3}{\partial f_2} \frac{df_2}{df_1}}_{\frac{df_3}{df_1}} \frac{df_1}{dx_1} \quad (21)$$

Similarly,

$$\frac{df_3}{dx_2} = \underbrace{\frac{\partial f_3}{\partial f_2} \frac{df_2}{df_1}}_{\frac{df_3}{df_1}} \frac{df_1}{dx_2} \quad (22)$$

The two derivatives share the same computation branch and we want to exploit this.



### 4.3 The AutoDiff Algorithm

1. Find the reverse ancestral (backwards) schedule of nodes (In example 2 above, this would be  $f_3, f_2, f_1, x_1, x_2$ ).
2. Start with the first node  $n_1$  in the reverse schedule and define  $t_{n_1} = 1$ .
3. For the next node  $n$  in the reverse schedule, find the child nodes  $\text{ch}(n)$ . Then define

$$t_n = \sum_{c \in \text{ch}(n)} \frac{\partial f_c}{\partial f_n} t_c$$

4. The total derivatives of  $f$  with respect to the root nodes of the tree (here  $x_1$  and  $x_2$ ) are given by the values of  $t$  at those nodes.

This is a general procedure that can be used to automatically define a subroutine to efficiently compute the gradient. It is efficient because information is collected at nodes in the tree and split between parents only when required.

#### 4.3.1 Dealing with loops

```

df=function(x)
f=function(x)
f=0
for i=1:10
    f=f+cos(f*x^i)
end
for i=1:10
    f=f+cos(f*x^i)
    df=df-sin(f*x^i)*f*i*x^{i-1}+df*x^i
end

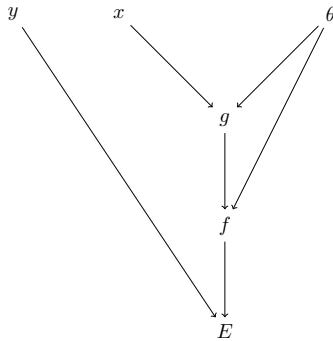
```

In the above example, we expanded the derivative of the cos term symbolically. In Au-

toDiff, unless the derivatives of such standard functions are given, we would need to replace this step with the computations on the AST and include these computations in the AutoDiff procedure.

### 4.4 Notes

Reverse AutoDiff generalises (and predates) Backprop since it holds for any computation tree. Another viewpoint of parameter tying is to write down the computation tree and carry out AutoDiff. In equation (1) the computation tree can be written as:



so that parameter tying can be seen as simply linking the same parameter to different nodes in the tree. One then carries out AutoDiff using the standard algorithm, which will have the same effect as we noted in section(2) – making two copies of  $\theta$ , namely  $\theta_1$  and  $\theta_2$  and summing over them is equivalent to simply summing over the two child paths of  $\theta$  in the computation tree.

## References

- [1] D. Barber. Dynamic Bayesian Networks with Deterministic Tables. In *Advances in Neural Information Processing Systems (NIPS)*, 2003.
- [2] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA, 1995.
- [3] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, Oct 1990.
- [4] R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. In Y. Chauvin and D. E. Rumelhart, editors, *Back-propagation: Theory, Architectures and Applications*, chapter 13, pages 433–486. Hillsdale, NJ: Erlbaum, 1995.

# Neural Nets: Initialisation Strategies

David Barber

## Initialisation

- Particularly for saturating transfer functions that have regions of zero gradient, sensible initialisation of the weights is essential.
- Consider a simple regression net with error

$$\sum_n (y^n - f(\mathbf{w}^\top \mathbf{x}^n))^2$$

- Firstly, it makes sense to scale the error so that it remains roughly order 1, even as the number of datapoints increases. For example

$$E(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^N (y^n - f(\mathbf{w}^\top \mathbf{x}^n))^2$$

- Gradient is

$$E(\mathbf{w}) = -\frac{2}{N} \sum_{n=1}^N (y^n - f(\mathbf{w}^\top \mathbf{x}^n)) f'(\mathbf{w}^\top \mathbf{x}^n) \mathbf{x}^n$$

- For  $f(x) = 1/(1 + e^{-x})$ , for large  $|x|$ , this function saturates to 1 or 0 and the gradient becomes zero. We need therefore to ensure we don't immediately get trapped in these zero gradient regions.

## Initialisation

- Let's assume that we have scaled the inputs  $x_i$  so that they have zero mean and unit variance

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

where

$$\mu_i = \sum_{n=1}^N x_i^n, \quad \sigma_i^2 = \sum_{n=1}^N (x_i^n - \mu_i)^2$$

After this rescaling,

$$\frac{1}{N} \sum_{n=1}^N x_i^n = 0, \quad \frac{1}{N} \sum_{n=1}^N (x_i^n)^2 = 1$$

## Initialisation

- Define the activation (input of the transfer function) as

$$z_n \equiv \mathbf{w}^T \mathbf{x}^n$$

Let's assume that we sample each  $w_i$  identically and independently from a zero mean distribution  $\mathbb{E}(w_i) = 0$ .

$$\mathbb{E}(z_n) = \sum_i \mathbb{E}(w_i) x_i^n = 0$$

$$\begin{aligned}\mathbb{E}(z_n^2) &= \sum_{i \neq j} \mathbb{E}(w_i) \mathbb{E}(w_j) x_i^n x_j^n + \sum_i \mathbb{E}(w_i^2) (x_i^n)^2 \\ &= \mathbb{E}(w^2) \sum_{i=1}^D (x_i^n)^2\end{aligned}$$

- Hence if we sample the  $w_i$  from a distribution with zero mean and variance  $1/D$ , then the activation will be zero mean with variance

$$\mathbb{E}(z_n^2) = \frac{1}{D} \sum_{i=1}^D (x_i^n)^2$$

## Initialisation

- Since each  $x_i^n$  has values roughly in the range -2 to 2 (2 standard deviations from the zero mean), then the activation will have values roughly in the range -2 to 2 as well.
- A reasonable initialisation then of the weights of a network with transfer function  $f(x)$  that has a non-saturating region when  $x$  is between -2 and 2, is: Draw each  $w_i$  from a distribution with zero mean and unit variance and then rescale. For example:

$$w_i \sim \mathcal{N}(w_i|0, 1), \quad w_i \rightarrow \frac{w_i}{\sqrt{D}}$$

$$w_i \sim \text{sign}(\mathcal{N}(w_i|0, 1)), \quad w_i \rightarrow \frac{w_i}{\sqrt{D}}$$

$$w_i \sim \mathcal{U}\left(w_i | -\sqrt{3}, \sqrt{3}\right), \quad w_i \rightarrow \frac{w_i}{\sqrt{D}}$$

- For nets with multiple hidden layers, the previous initialisation strategy is also commonly used. or weights in layer  $l$ , we set (for example)

$$w_i^l \sim \mathcal{N}(w_i^l|0, 1), \quad w_i^l \rightarrow \frac{w_i^l}{\sqrt{D_{l-1}}}$$

where  $D_l$  is the number of units in layer  $l$ .

## Decorrelating Inputs

For an  $\mathbf{x} \rightarrow y$  net the Hessian is

$$\mathbf{H} = -\frac{2}{N} \sum_n \underbrace{\left[ (y^n - f(\mathbf{w}^\top \mathbf{x}_n)) f''(\mathbf{w}^\top \mathbf{x}_n) - (f'(\mathbf{w}^\top \mathbf{x}_n))^2 \right]}_{\equiv \gamma_n} \mathbf{x}_n \mathbf{x}_n^\top$$

As a very crude approximation, we can assume that all the  $\gamma_n$  are roughly equal to a value  $\gamma$ . In that case, the approximate Hessian is

$$\mathbf{H} \approx -\frac{2\gamma}{N} \sum_n \mathbf{x}_n \mathbf{x}_n^\top$$

If we perform SVD of the matrix  $\mathbf{X} = [\mathbf{x}^1, \dots, \mathbf{x}^N]$  then

$$\sum_n \mathbf{x}_n \mathbf{x}_n^\top = \mathbf{X} \mathbf{X}^\top = \mathbf{U} \mathbf{S} \mathbf{V}^\top \mathbf{V} \mathbf{S}^\top \mathbf{U}^\top = \mathbf{U} \mathbf{S}^2 \mathbf{U}^\top$$

Hence applying the ‘whitening’ transformation

$$\mathbf{x}^n \rightarrow \mathbf{S}^{-1} \mathbf{U}^\top \mathbf{x}^n$$

will make the approximate Hessian diagonal and can be a useful initial preprocessing step to make optimisation easier.

## A more complex discussion

- Let's consider what happens for the second hidden layer. This will have inputs

$$f(z_{nj}), \quad z_{nj} \equiv \mathbf{w}_j^T \mathbf{x}^n$$

where  $j$  is the index of the neuron in the first layer.

- And the activation to a unit in the second layer will be

$$\tilde{z}_n \equiv \sum_j u_j f(z_{nj})$$

- If we assume that  $z_{nj} \sim \mathcal{N}(z_{nj} | 0, 1)$  and the  $u_j$  are independently sampled from a zero mean distribution then the activation of the second layer has zero mean and variance

$$\mathbb{E}(u^2) \sum_{j=1}^{D_1} (f(z_{nj}))^2$$

- If we therefore draw each  $u_j^l$  from a distribution with zero mean and variance

$$\frac{1}{D_{l-1} \mathbb{E}(f(z)^2)_{\mathcal{N}(z|0,1)}}$$

the activation of each unit  $j$  in layer  $l$  will be roughly zero mean with unit variance distributed.

# Recurrent Neural Nets

David Barber

## Gradient Decay/Explosion

- Consider a network with linear transfer function  $f(x) = x$ .
- For simplicity, assume each layer has the same width (number of neurons).
- For input  $\mathbf{x}$  the final layer has value

$$\mathbf{W}_L \mathbf{W}_{L-1} \dots \mathbf{W}_2 \mathbf{x}$$

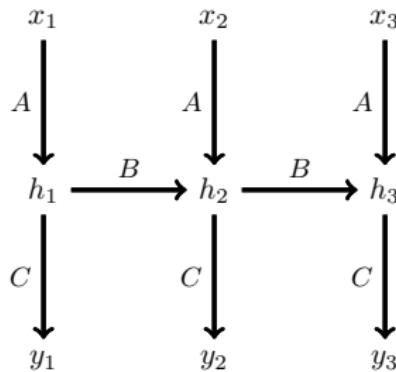
If all the weight matrices are the same  $\mathbf{W}_l = \mathbf{W}$ , we would have final layer

$$\mathbf{W}^{L-1} \mathbf{x} = \mathbf{E} \Lambda^{L-1} \mathbf{E}^{-1} \mathbf{x}$$

where  $\Lambda$  is the diagonal matrix of eigenvalues of  $\mathbf{W}$  and  $\mathbf{E}$  is the matrix of eigenvectors.

- Each element  $\lambda_i^{L-1}$  will be exponentially large (if  $\lambda_i > 1$ ) or small (if  $\lambda_i < 1$ ). Only elements with  $\lambda_i = 1$  will remain well scaled.
- This suggests that, unless initialised very carefully, gradients may become either zero or infinite, leading to significant difficulties in training.
- This is particularly a problem in very deep or recurrent nets.

## Recurrent Nets



- RNNs are used in timeseries applications
- The basic idea is that the hidden units at time  $h_t$  (and possibly output  $y_t$ ) depend on the previous state of the network  $h_{t-1}, x_{t-1}, y_{t-1}$  for inputs  $x_t$  and outputs  $y_t$ .
- In the above network, I ‘unrolled the net through time’ to give a standard NN diagram.
- I omitted the potential links from  $x_{t-1}, y_{t-1}$  to  $h_t$ .

## Mitigating Memory Decay

- Someone shows you a picture of a dog and asks you to commit this to memory.
- They then show you a sequence of other images.
- If a new image is a dog, you should show the previous image of the dog at the output and store the new dog image to memory.
- This would be difficult to do with a traditional recurrent net since typically information is lost from timestep to timestep.
- Here you need to also explicitly store some information and retrieve it at a potentially much later timepoint.
- Thus we have the question of how to commit things to memory and potentially retrieve them later.

---

## Structured RNNs

- It's worth noting that everything we do here will be in the context of RNNs.
- Each model (simple memory model, LSTM) is a *specially constrained* RNN.
- Theoretically, we can do all of this with a conventional unconstrained RNN. In practice, however, training such models to store long term dependencies has proven difficult. Life is much easier if we use specially constrained models that are biased towards solving long-term memory storage.

## Mitigating Memory Decay

- Let's consider how we might solve the previous 'dog' problem.
- One way to solve this is to have units that represent a 'running memory'  $m(t)$ .
- First we need to decide if the current image  $x(t)$  is a 'dog' and should therefore be stored. Using a matrix  $W_{in}$  that is a row vector and scalar bias  $B_{in}$  the logistic sigmoid outputs a value  $\alpha(t) \in [0, 1]$  that represents whether the image  $x(t)$  is a dog and should therefore be committed to memory. We can therefore define a 'gate':

$$\alpha(t) = \sigma(W_{in}x(t) + B_{in})$$

- We can now update the memory vector  $m(t)$  using

$$m(t) = \alpha(t)x(t) + (1 - \alpha(t))m(t - 1)$$

If  $\alpha(t)$  is close to 1, we essentially replace the memory with  $x(t)$ . For  $\alpha(t)$  close to zero, we ignore  $x(t)$  and essentially copy the previous value of the memory  $m(t - 1)$  to the current memory  $m(t)$ .

- We can then simply output

$$y_{out}(t) = \alpha(t)m(t - 1)$$

# Mitigating Memory Decay

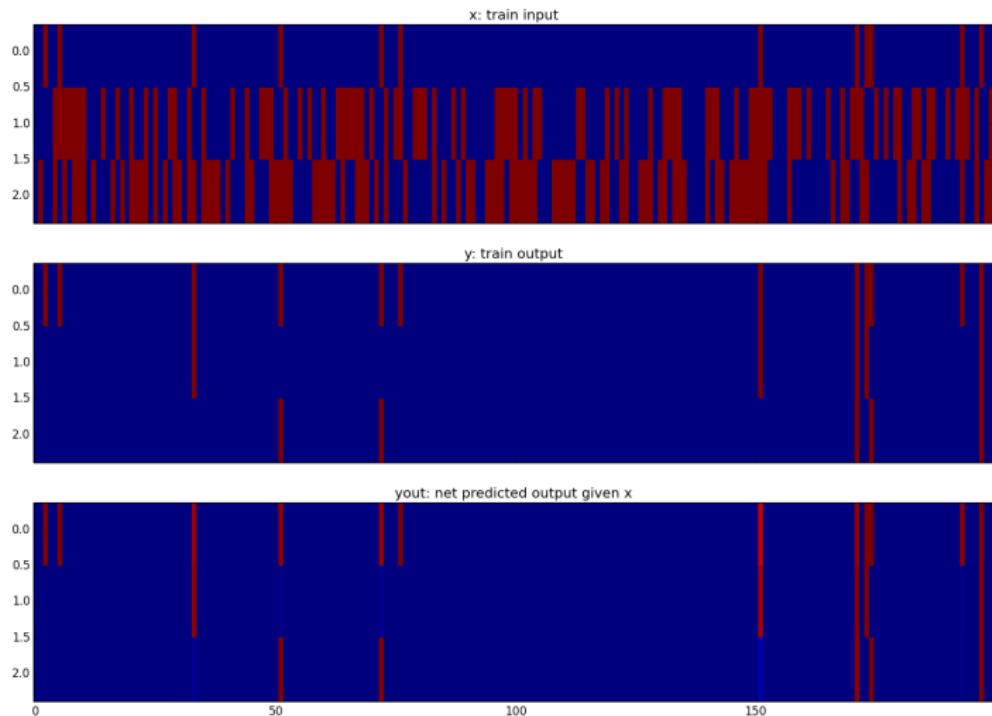
- The key idea is that we have made a structured form of RNN in which units in the network have specific roles. In this case the hidden units at time  $t$  have the role of a gate and a memory store:

$$h(t) = (\alpha(t), m(t))$$

- Critically, the update  $h(t) \rightarrow h(t + 1)$  contains a copying mechanism.
- Provided the  $\alpha(t)$  are very close to 1, then the memory will be retained across many timepoints.
- Note, however, that eventually the memory would still decay (hence it is still 'short-term', just longer than rapid Markovian memory decay. However, this mechanism is a way to construct an architecture that is well suited to storing patterns over longer timescales.
- Technically, the reason this works is that for  $\alpha(t)$  close to 1, then the gradient does not decay significantly over time.

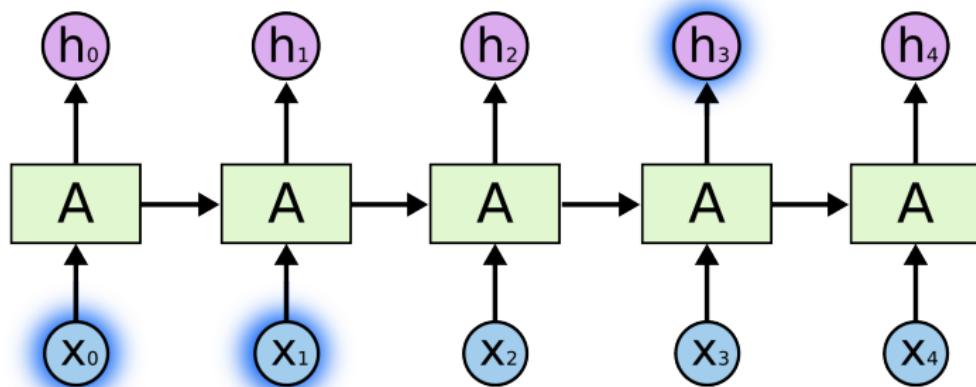
# Simple Memory Storage

The task: If either  $x_1$  or  $x_2$  is 1 (or both), then store the current vector  $x(t)$  to the memory and output the previous memory [DemoSimpleMemory.jl](#):



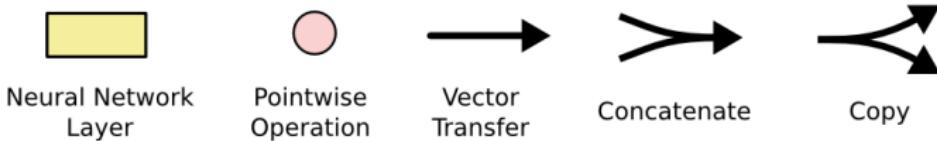
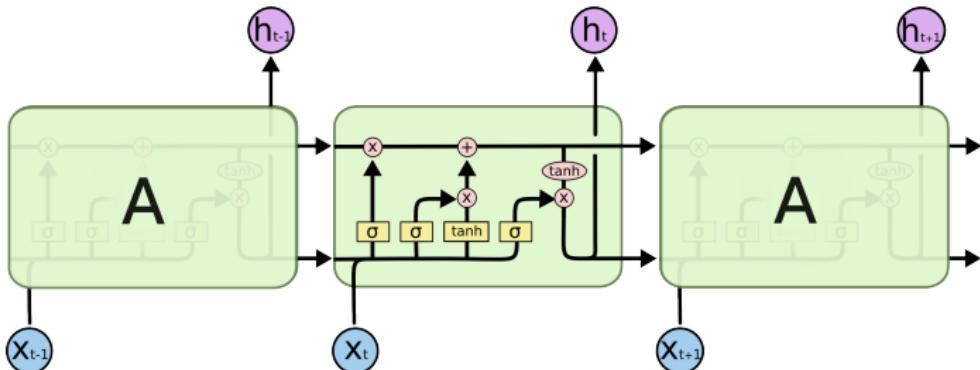
# Long Short Term Memory

- LSTM is an extension of the previous simple memory model
- <http://colah.github.io/posts/2015-08-Understanding-LSTMs/> gives a nice tutorial, from where I've taken some of the images.



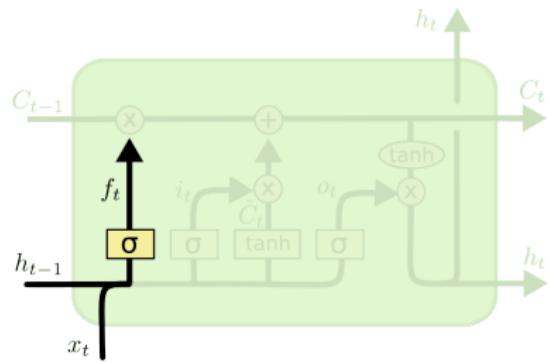
- In this notation,  $x_t$  is an input at time  $t$ ,  $A$  is the state of the hidden nodes at each time and  $h_t$  is the output.
- We can train an RNN to predict the next word in the sequence. For a sentence like "I grew up in France I speak fluent French", predicting that the final word is French requires to store the knowledge "France", encountered many timesteps before.

# Long Short Term Memory



- The forget gate decides whether to keep the old memory, or replace it.
- The input gate decides whether the current input should be added to the memory  $C$  (the upper horizontal line).
- The output gate decides whether the the current memory should be sent to the output.

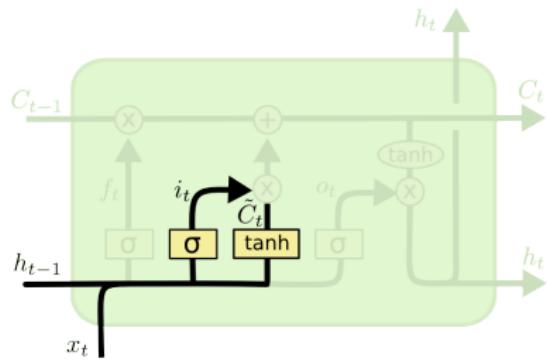
# Long Short Term Memory: Forget Gate



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

- We first concatenate the previous output and current input.
- The value of  $f_t \in [0, 1]$  then decides whether to forget ( $f_t = 1$ ) or keep ( $f_t = 0$ ) the current memory  $C_{t-1}$ .

# Long Short Term Memory: Input Gate

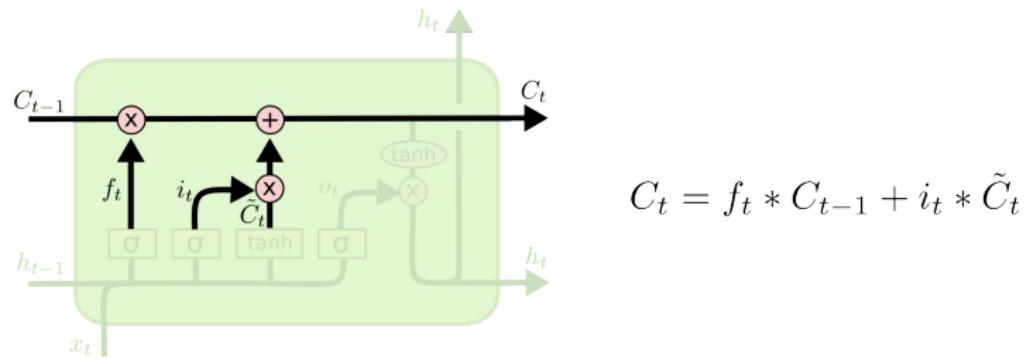


$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

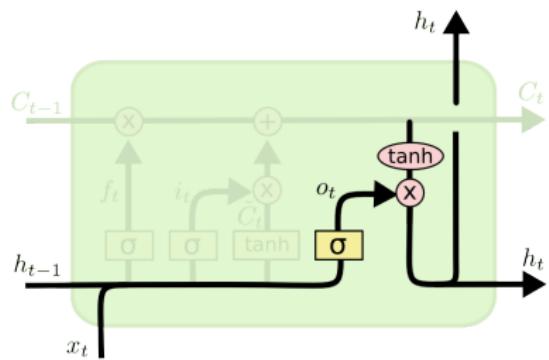
- The input gate decides (on the basis of  $x_t$  and the previous output  $h_{t-1}$  whether something should be stored in the memory at this timestep.
- $\tilde{C}_t$  is what we will store.

# Long Short Term Memory: Memory Cell



- We then update the memory cell.
- We forget previous values and commit new values to the memory.

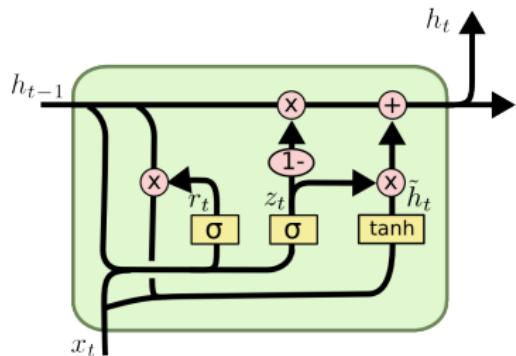
# Long Short Term Memory: Output Gate



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

- The output gate decides whether to output the memory cell at this timestep.
- $h_t$  is what we output. One could easily modify this output if the tanh constraint is not appropriate.

## Gated Recurrent Net



$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- The GRU is a simpler architecture that also works well.
- Here  $h_t$  plays the role of the memory.
- $z_t$  is a factor that determines whether to forget the old and commit a new memory  $\tilde{h}_t$ .
- What you store  $\tilde{h}_t$  depends on the current input and possibly an amount (gated by  $r_t$ ) of the previous memory  $h_{t-1}$ .
- Note that we can also use multiple LSTM or GRU units in a net. Infinitely many architectures for this – eg Grid LSTM, stacked LSTM, etc.

## Handwriting Generation using an LSTM RNN

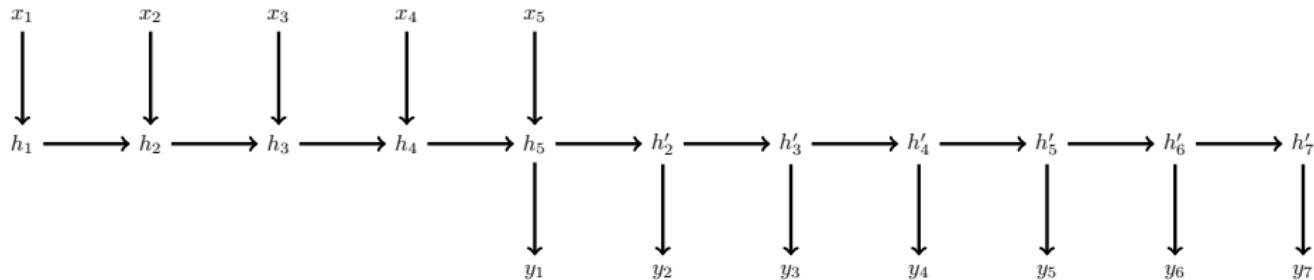
more of national temperament  
more of national temperament

- Top line is real handwriting.
- Some generated examples (from Alex Graves' work).
- The LSTM enables long term consistency in the generated examples.

## Sequence to Sequence models

- Seq2Seq models take an input sequence  $x_1, \dots, x_M$  and output a sequence  $y_1, \dots, y_N$ .
- An important consideration is that the input and output sequences are *potentially of different length*.
- An example is machine translation:  
 $x_{1:4}$ : Stop tickling me!  
 $y_{1:5}$ : Hör auf mich zu kitzeln!
- There are various approaches based on using a RNN to convert the input sequence to a fixed length vector.
- This vector is then used by another RNN to generate the output sequence.  
See Sutskever et al “Sequence to Sequence Learning using Neural Networks”.

# A Sequence to Sequence Model



$x_1 = \text{stop}$ ,  $x_2 = \text{tickling}$ ,  $x_3 = \text{me}$ ,  $x_4 = !$ ,  $x_5 = \text{EndOfSentence}$

$y_1 = \text{Hör}$ ,  $y_2 = \text{auf}$ ,  $y_3 = \text{mich}$ ,  $y_4 = \text{zu}$ ,  $y_5 = \text{kitzeln}$ ,  $y_6 = !$ ,  $y_7 = \text{EndOfSentence}$

- In this model the hidden node  $h_5$  contains all the relevant information in the source sentence.
- We then use a different net to generate the translation sequence.
- One can also add inputs to recursively generate the translation, so that  $y_{t-1}$  is an additional input to  $h'_t$ .
- In practice it is also useful to reverse the input sequence order.
- Sutskever et al used a 4-layer LSTM in which the outputs of each LSTM cell is used as the input of another.

# Translation

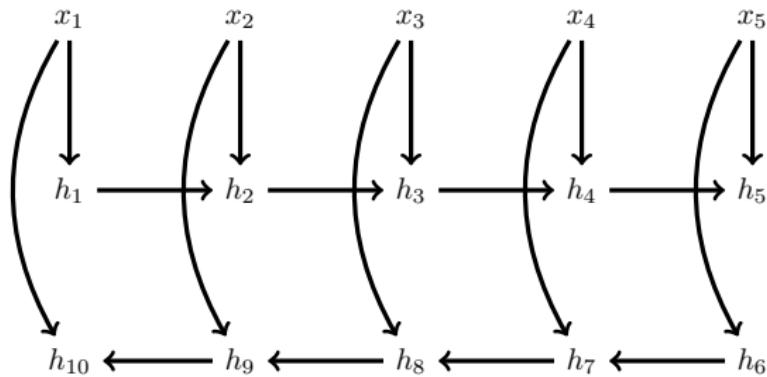
Type	Sentence
<b>Our model</b>	Ulrich UNK , membre du conseil d' administration du constructeur automobile Audi , affirme qu' il s' agit d' une pratique courante depuis des années pour que les téléphones portables puissent être collectés avant les réunions du conseil d' administration afin qu' ils ne soient pas utilisés comme appareils d' écoute à distance .
	Ulrich Hackenberg , membre du conseil d' administration du constructeur automobile Audi , déclare que la collecte des téléphones portables avant les réunions du conseil , afin qu' ils ne puissent pas être utilisés comme appareils d' écoute à distance , est une pratique courante depuis des années .
<b>Our model</b>	“ Les téléphones cellulaires , qui sont vraiment une question , non seulement parce qu' ils pourraient potentiellement causer des interférences avec les appareils de navigation , mais nous savons , selon la FCC , qu' ils pourraient interférer avec les tours de téléphone cellulaire lorsqu' ils sont dans l' air ” , dit UNK .
	“ Les téléphones portables sont véritablement un problème , non seulement parce qu' ils pourraient éventuellement créer des interférences avec les instruments de navigation , mais parce que nous savons , d' après la FCC , qu' ils pourraient perturber les antennes-relais de téléphonie mobile s' ils sont utilisés à bord ” , a déclaré Rosenker .
<b>Our model</b>	Avec la crémation , il y a un “ sentiment de violence contre le corps d' un être cher ” , qui sera “ réduit à une pile de cendres ” en très peu de temps au lieu d' un processus de décomposition “ qui accompagnera les étapes du deuil ” .
	Il y a , avec la crémation , “ une violence faite au corps aimé ” , qui va être “ réduit à un tas de cendres ” en très peu de temps , et non après un processus de décomposition , qui “ accompagnerait les phases du deuil ” .

Table 3: A few examples of long translations produced by the LSTM alongside the ground truth translations. The reader can verify that the translations are sensible using Google translate.

- Some translations (original English sentence not shown).
- These are examples of long sentences to demonstrate the ability of the model.

# Bidirectional RNNs

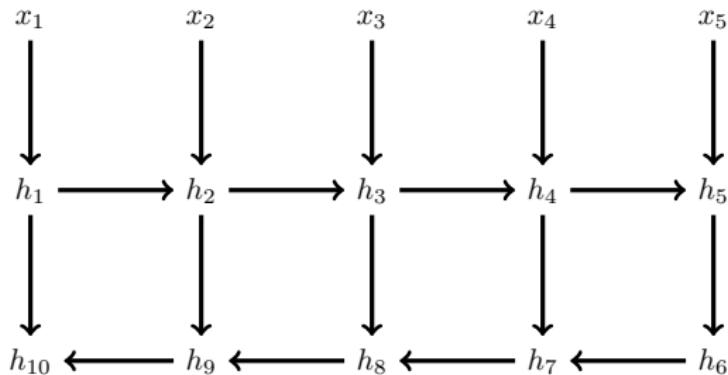
- A potential criticism of the previous Seq2Seq model is that it is biased towards capturing the latter part of the sentence.
- In bidirectional models, the encoder model traverses the input sequence in both directions. One example architecture is:



- The joint state  $(h_5, h_{10})$  in this case could be used to represent the input sequence  $x_{1:5}$  and used as input to a decoder RNN.

# Bidirectional RNNs

- An alternative bidirectional model:



- The state of  $h_{10}$  in this case could be used to represent the input sequence  $x_{1:5}$  and used as input to a decoder RNN.

# Unsupervised Dimension Reduction and Matrix Factorisation

David Barber

# Learning Objectives

## Lectures

- Why we want to find low-dimensional representations of data.
  - How we can do this using Principal Components Analysis.
  - How to go beyond PCA, including Non-Negative matrix factorisation.
  - Example applications in Recommender systems, signal processing, text analysis.
- 

## Practicals

- PCA
- Non-negative Matrix Factorisation

# High-Dimensional Spaces – Low Dimensional Manifolds

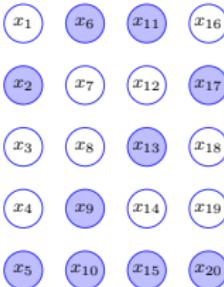


Image represented by a 20 dimensional vector:

$$\mathbf{x} = (0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1)^T.$$

- Data can be high dimensional – images (here a ‘2’).
- As we move through the 20 dimensional space, is each point in the space likely to look like a ‘2’?
- No – only a very limited part of the space correspond to images that look like ‘2s’ – the space is highly constrained. There are only certain directions within the space that correspond to sensible ‘2s’ – we want to find this small number of relevant directions.

`demoLowDManifold.m`

# High-Dimensional Spaces – Low Dimensional Manifolds

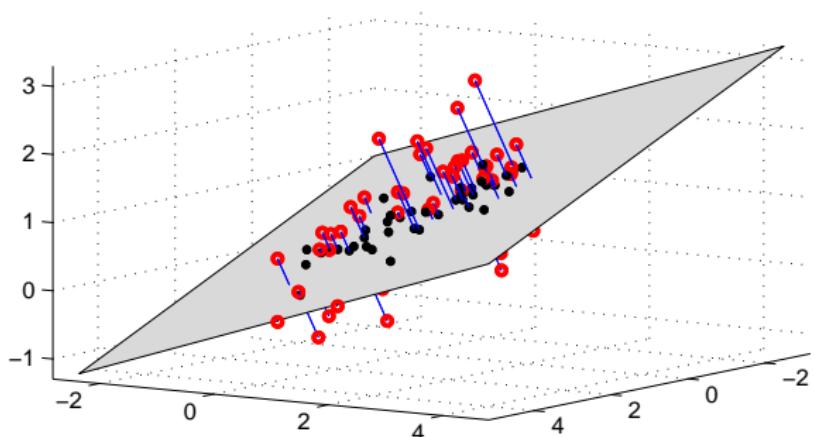
$$\begin{matrix} \textcircled{1} & \textcircled{2} & \textcircled{3} & \textcircled{4} \\ \textcircled{5} & \textcircled{6} & \textcircled{7} & \textcircled{8} \\ \textcircled{9} & \textcircled{10} & \textcircled{11} & \textcircled{12} \\ \textcircled{13} & \textcircled{14} & \textcircled{15} & \textcircled{16} \end{matrix} = \begin{matrix} \textcircled{1} & \textcircled{2} & \textcircled{3} & \textcircled{4} \\ \textcircled{5} & \textcircled{6} & \textcircled{7} & \textcircled{8} \\ \textcircled{9} & \textcircled{10} & \textcircled{11} & \textcircled{12} \\ \textcircled{13} & \textcircled{14} & \textcircled{15} & \textcircled{16} \end{matrix} + \begin{matrix} \textcircled{1} & \textcircled{2} & \textcircled{3} & \textcircled{4} \\ \textcircled{5} & \textcircled{6} & \textcircled{7} & \textcircled{8} \\ \textcircled{9} & \textcircled{10} & \textcircled{11} & \textcircled{12} \\ \textcircled{13} & \textcircled{14} & \textcircled{15} & \textcircled{16} \end{matrix} + \begin{matrix} \textcircled{1} & \textcircled{2} & \textcircled{3} & \textcircled{4} \\ \textcircled{5} & \textcircled{6} & \textcircled{7} & \textcircled{8} \\ \textcircled{9} & \textcircled{10} & \textcircled{11} & \textcircled{12} \\ \textcircled{13} & \textcircled{14} & \textcircled{15} & \textcircled{16} \end{matrix}$$

**x**                    **c**                    **b<sup>1</sup>**                    **b<sup>2</sup>**

- We can describe vectors by first finding the mean **c** of all images of '2s'.
- Then we can add on vectors in directions **b<sup>1</sup>** and **b<sup>2</sup>** to build up other images that look like '2s'.

# High-Dimensional Spaces – Low Dimensional Manifolds

Provided there is some ‘structure’, data will typically lie close to a much lower dimensional ‘manifold’. Here we concentrate on computationally efficient linear dimension reduction techniques.



# Principal Components Analysis (PCA)

## Full rank representation

Any D-dimensional vector  $\mathbf{x}$  can be written as

$$\mathbf{x} = \sum_{j=1}^D y_j \mathbf{b}^j$$

for linearly independent basis vectors  $\mathbf{b}$ .

---

## Reduced rank approximation

$$\mathbf{x} \approx \mathbf{c} + \sum_{j=1}^M y_j \mathbf{b}^j \equiv \tilde{\mathbf{x}}$$

- The vector  $\mathbf{c}$  defines a fixed point in the subspace and  $M < D$ .
- The  $\mathbf{b}^j$  are ‘basis’ vectors that span the subspace. Collectively we can write  $\mathbf{B} = [\mathbf{b}^1, \dots, \mathbf{b}^M]$ .
- The  $y_i$  are the low dimensional co-ordinates of the data.
- We can write

$$\tilde{\mathbf{x}} = \mathbf{c} + \mathbf{B}\mathbf{y}$$

## Minimal square loss approximation

- We have a collection of  $N$   $D$ -dimensional data vectors  $\mathbf{x}^1, \dots, \mathbf{x}^N$ .
- To determine the best low rank approximation we can minimise the square distance error between  $\mathbf{x}^n$  and its approximation  $\tilde{\mathbf{x}}^n$ :

$$E(\mathbf{B}, \mathbf{Y}, \mathbf{c}) = \sum_{n=1}^N \sum_{i=1}^D [x_i^n - \tilde{x}_i^n]^2$$

---

## Redundancy

- Formally, the least squares criterion doesn't uniquely define the directions.
- Trivially, we could use  $-\mathbf{b}^i$  instead of  $\mathbf{b}^i$ , but more generally we can use any vectors which span the same space as  $\mathbf{B}$ .
- These solutions all have the same square loss – they are all ‘optimal’.
- We can make the solution essentially unique by requiring that all the  $\mathbf{b}$  are orthogonal to each other and of unit length.

## Least Squares solution

- Compute the mean of the data:

$$\mathbf{m} = \frac{1}{N} \sum_{n=1}^N \mathbf{x}^n$$

- Compute the matrix:

$$\hat{\mathbf{X}} = [\mathbf{x}^1 - \mathbf{m}, \dots, \mathbf{x}^N - \mathbf{m}]$$

- Find the Singular Value Decomposition

$$\hat{\mathbf{X}} = \mathbf{U} \mathbf{D} \mathbf{V}^\top$$

- The first  $M$  columns of  $\mathbf{U}$  form the matrix  $\mathbf{B}$ .
- The lower dimensional representations are then given by  $\mathbf{Y} = \mathbf{B}^\top \hat{\mathbf{X}}$ .
- The approximate reconstructions are given by  $\tilde{\mathbf{X}} \equiv \mathbf{B}\mathbf{Y} + \mathbf{M}$  where  $\mathbf{M} = [\mathbf{m} \dots \mathbf{m}]$ .

## Eigenvectors and PCA

- One can show that the above SVD approach is equivalent to finding an eigen-decomposition of the sample covariance matrix

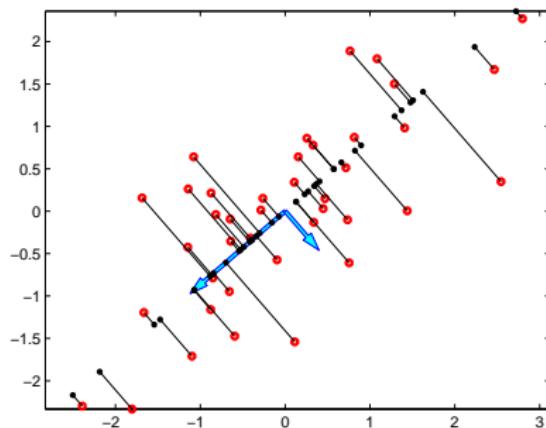
$$\frac{1}{N-1} \sum_{n=1}^N (\mathbf{x}^n - \mathbf{m})(\mathbf{x}^n - \mathbf{m})^\top$$

and taking the leading  $M$  eigenvectors (which form the columns of  $\mathbf{B}$ ) and their corresponding eigenvalues  $\lambda_i$ .

- The singular values and eigenvalues are related by

$$\lambda_i = D_{ii}^2$$

## PCA: Reducing 2D data to a 1D approximation



The original datapoints  $x$  (larger rings) and their reconstructions  $\tilde{x}$  (small dots) using 1 dimensional PCA. The lines represent the orthogonal projection of the original datapoint onto the first eigenvector. The arrows are the two eigenvectors scaled by the square root of their corresponding eigenvalues. For each datapoint  $x$ , the 'low dimensional' representation  $y$  is given by the distance (possibly negative) from the origin along the first eigenvector direction to the corresponding orthogonal projection point.

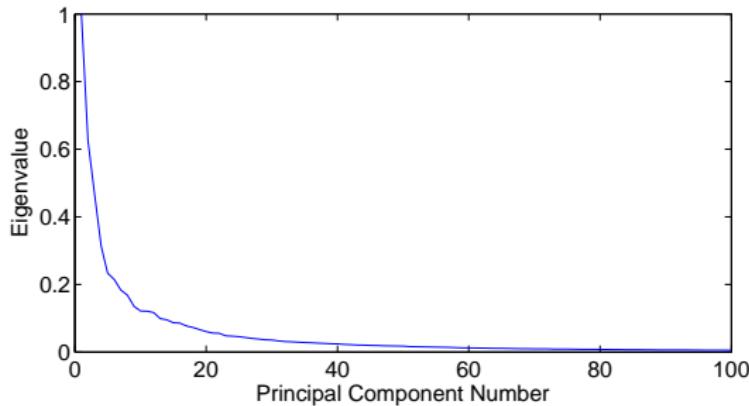
## Reducing the dimension of digits

Each digit image consists of  $28 \times 28 = 784$  pixels, each pixel in the range  $0, \dots, 255$ . For each image, we first form a vector of the image by concatenating the columns of the image matrix into a 784 dimensional vector.



Top row : a selection of '5s' taken from the database of 892 examples. Plotted beneath each digit is the reconstruction using 100, 30 and 5 principal components (from top to bottom). Note how the reconstructions for fewer PCs express less variability from each other, and resemble more a mean 5 digit.

## Eigen-spectrum of the '5' digits



- 100 largest eigenvalues (scaled so that the largest is 1).
- If we order the eigenvalues  $\lambda_1 \geq \lambda_2, \dots$ , the squared error is then given by the sum of the neglected eigenvalues

$$E = (N - 1) \sum_{i=M+1}^D \lambda_i$$

- The 'latent' dimensionality of data can be defined by a 'kink' in the spectrum (where the eigenvalues stop rapidly decreasing)

## PCA via Singular Value Decomposition

- In practice, it can be impractical to first compute the covariance matrix and then the eigenvalues.
- A mathematically equivalent but computationally more convenient approach is to consider the SVD decomposition of the data matrix:

$$\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T$$

where  $\mathbf{U}^T\mathbf{U} = \mathbf{I}_D$  and  $\mathbf{V}^T\mathbf{V} = \mathbf{I}_N$  and  $\mathbf{D}$  is a diagonal matrix of the (positive) singular values.

- We can then approximate

$$\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T \approx \mathbf{U}_M \mathbf{D}_M \mathbf{V}_M^T$$

where  $\mathbf{U}_M$ ,  $\mathbf{D}_M$ ,  $\mathbf{V}_M$  correspond to taking only the first  $M$  singular values of the full matrices.

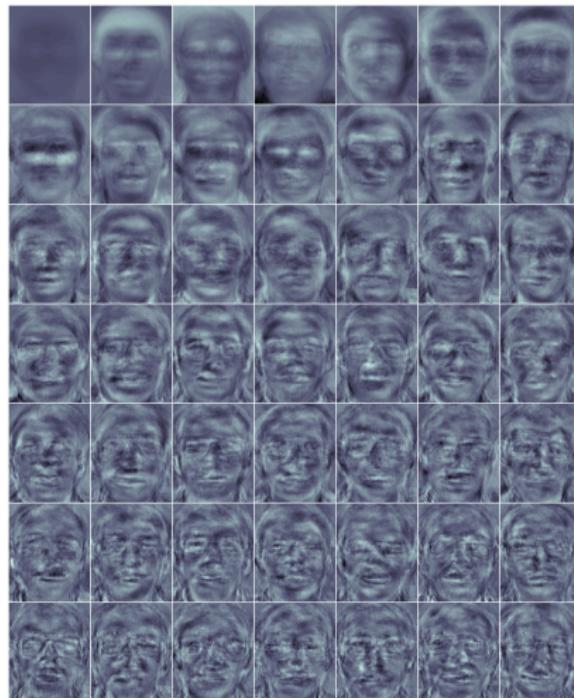
- PCA corresponds to setting  $\mathbf{B} = \mathbf{U}_M$  and the eigenvalues are the diagonal elements of  $\mathbf{D}_M$  squared.

## Finding a low dimensional representation of Faces



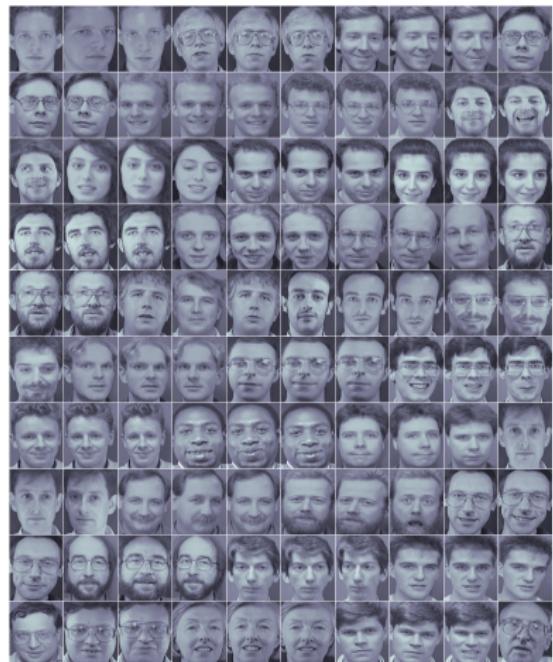
100 of the 120 training images (40 people, with 3 images of each person). Each image consists of  $92 \times 112 = 10304$  non-negative greyscale pixels.

## Eigenfaces: the 49 largest Principal Components

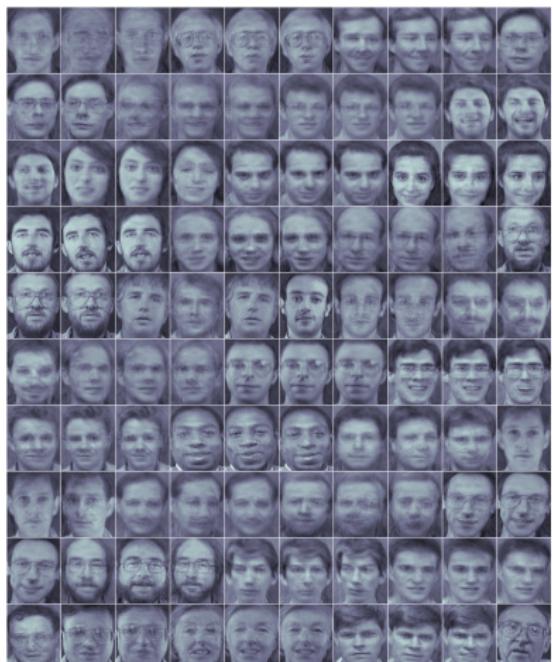


The eigenvectors (each plotted as an image) corresponding to the largest 49 eigenvalues.

## Reconstruction using PCA with 49 components



(a)



(b)

**Figure :** (a): Original Data. (b): PCA reconstruction of the images using a combination of the 49 principal components.

# PCA Mathematics

## Least Squares Objective

To determine the best lower dimensional representation it is convenient to use the square distance error between  $\mathbf{x}$  and its reconstruction  $\tilde{\mathbf{x}}$ :

$$E(\mathbf{B}, \mathbf{Y}, \mathbf{c}) = \sum_{n=1}^N \sum_{i=1}^D [x_i^n - \tilde{x}_i^n]^2$$

---

## Centering

The optimal bias  $\mathbf{c}$  is given by the mean of the data  $\sum_n \mathbf{x}^n / N$ . We therefore assume that the data has been centred (has zero mean  $\sum_n \mathbf{x}^n = \mathbf{0}$ ), so that we can set  $\mathbf{c}$  to zero, and concentrate on finding the optimal basis  $\mathbf{B}$  below.

## PCA Mathematics (2): Rotation Invariance

We wish to minimize the sum of squared differences between each vector  $\mathbf{x}$  and its reconstruction  $\tilde{\mathbf{x}}$ :

$$E(\mathbf{B}, \mathbf{Y}) = \sum_{n=1}^N \sum_{i=1}^D \left[ x_i^n - \sum_{j=1}^M y_j^n b_i^j \right]^2 = \text{trace} \left( (\mathbf{X} - \mathbf{B}\mathbf{Y})^\top (\mathbf{X} - \mathbf{B}\mathbf{Y}) \right)$$

where  $\mathbf{X} = [\mathbf{x}^1, \dots, \mathbf{x}^N]$ .

---

### Orthonormality constraint

Consider an invertible transformation  $\mathbf{Q}$  of the basis  $\mathbf{B}$  so that  $\tilde{\mathbf{B}} \equiv \mathbf{B}\mathbf{Q}$  is an orthonormal matrix,  $\tilde{\mathbf{B}}^\top \tilde{\mathbf{B}} = \mathbf{I}$ . Since  $\mathbf{Q}$  is invertible, we may write

$\mathbf{B}\mathbf{Y} = \tilde{\mathbf{B}}\mathbf{Q}^{-1}\mathbf{Y} \equiv \tilde{\mathbf{B}}\tilde{\mathbf{Y}}$ , which is of then same form as  $\mathbf{B}\mathbf{Y}$ , albeit with an orthonormality constraint on  $\tilde{\mathbf{B}}$ . Hence, without loss of generality, we may impose the orthonormality constraint  $\mathbf{B}^\top \mathbf{B} = \mathbf{I}$ , namely that the basis vectors are mutually orthogonal and of unit length.

## PCA mathematics: Finding the optimal $\mathbf{Y}$

$$-\frac{1}{2} \frac{\partial}{\partial y_k^n} E(\mathbf{B}, \mathbf{Y}) = \sum_i \left[ x_i^n - \sum_j y_j^n b_i^j \right] b_i^k = \sum_i x_i^n b_i^k - \sum_j y_j^n \underbrace{\sum_i b_i^j b_i^k}_{\delta_{jk}}$$

The squared error  $E(\mathbf{B}, \mathbf{Y})$  therefore has zero derivative when

$$y_k^n = \sum_i b_i^k x_i^n, \quad \text{which can be written as } \mathbf{Y} = \mathbf{B}^\top \mathbf{X}$$

The objective  $E(\mathbf{B})$  becomes

$$E(\mathbf{B}) = \text{trace} \left( (\mathbf{X} - \mathbf{B}\mathbf{Y})^\top (\mathbf{X} - \mathbf{B}\mathbf{Y}) \right) = \text{trace} \left( \mathbf{X}^\top (\mathbf{I} - \mathbf{B}\mathbf{B}^\top)^2 \mathbf{X} \right)$$

Since  $(\mathbf{I} - \mathbf{B}\mathbf{B}^\top)^2 = \mathbf{I} - \mathbf{B}\mathbf{B}^\top$ , (using  $\mathbf{B}^\top \mathbf{B} = \mathbf{I}$ )

$$E(\mathbf{B}) = \text{trace} \left( \mathbf{X} \mathbf{X}^\top (\mathbf{I} - \mathbf{B}\mathbf{B}^\top) \right)$$

## PCA mathematics: The role of the sample covariance

Hence the objective becomes

$$E(\mathbf{B}) = (N - 1) [\text{trace}(\mathbf{S}) - \text{trace}(\mathbf{SBB}^T)]$$

where  $\mathbf{S}$  is the sample covariance matrix of the centred data. Since we assumed the data is zero mean, this is

$$\mathbf{S} = \frac{1}{N-1} \sum_{n=1}^N \mathbf{x}^n (\mathbf{x}^n)^T = \frac{1}{N-1} \mathbf{XX}^T$$

---

### General case

If we hadn't centred the data, we would have

$$\mathbf{S} = \frac{1}{N-1} \sum_{n=1}^N (\mathbf{x}^n - \mathbf{m}) (\mathbf{x}^n - \mathbf{m})^T$$

where  $\mathbf{m}$  is the sample mean of the data.

## PCA maths: Enforcing orthonormality

To minimise  $E(\mathbf{B})$  under the constraint  $\mathbf{B}^T \mathbf{B} = \mathbf{I}$  we use a set of Lagrange multipliers  $\mathbf{L}$ , so that the objective is to minimize

$$-\text{trace} (\mathbf{S} \mathbf{B} \mathbf{B}^T) + \text{trace} (\mathbf{L} (\mathbf{B}^T \mathbf{B} - \mathbf{I}))$$

Since the constraint is symmetric, we can assume that  $\mathbf{L}$  is also symmetric. Differentiating with respect to  $\mathbf{B}$  and equating to zero we obtain that at the optimum

$$\mathbf{S} \mathbf{B} = \mathbf{B} \mathbf{L}$$

We need to find matrices  $\mathbf{B}$  and  $\mathbf{L}$  that satisfy this equation. One solution is given when  $\mathbf{L}$  is diagonal in which case this is a form of eigen-equation and the columns of  $\mathbf{B}$  are the corresponding eigenvectors of  $\mathbf{S}$ .

In this case,  $\text{trace}(\mathbf{S}\mathbf{B}\mathbf{B}^T) = \text{trace}(\mathbf{L})$ , which is the sum of the eigenvalues corresponding to the eigenvectors forming  $\mathbf{B}$ .

$$\frac{1}{N-1}E(\mathbf{B}) = -\text{trace}(\mathbf{L}) + \text{trace}(\mathbf{S}) = -\sum_{i=1}^M \lambda_i + \text{const.}$$

Since we wish to minimise  $E(\mathbf{B})$ , we therefore define the basis using the eigenvectors with largest corresponding eigenvalues.

---

### Residual error

If we order the eigenvalues  $\lambda_1 \geq \lambda_2, \dots$ , the squared error is then given by

$$\frac{1}{N-1}E(\mathbf{B}) = \text{trace}(\mathbf{S}) - \text{trace}(\mathbf{L}) = \sum_{i=1}^D \lambda_i - \sum_{i=1}^M \lambda_i = \sum_{i=M+1}^D \lambda_i$$

Hence the sum of the neglected eigenvalues equals the squared reconstruction error.

---

### Uniqueness

Whilst the solution to this eigen-problem is unique, this only serves to define the solution subspace since one may rotate and scale  $\mathbf{B}$  and  $\mathbf{Y}$  such that the value of the squared loss is exactly the same

## PCA: Breaking rotational Invariance

To break the invariance of least squares projection with respect to rotations and rescaling, we need an additional criterion. One such is given by first searching for the single direction  $\mathbf{b}$  such that the variance of the data projected onto this direction is maximal amongst all possible such projections. For a single vector  $\mathbf{b}$  we have

$$y^n = \sum_i b_i x_i^n$$

The projection of a datapoint onto a direction  $\mathbf{b}$  is  $\mathbf{b}^T \mathbf{x}^n$  for a unit length vector  $\mathbf{b}$ . Hence the sum of squared projections is

$$\sum_n (\mathbf{b}^T \mathbf{x}^n)^2 = \mathbf{b}^T \left[ \sum_n \mathbf{x}^n (\mathbf{x}^n)^T \right] \mathbf{b} = (N - 1) \mathbf{b}^T \mathbf{S} \mathbf{b}$$

The optimal single  $\mathbf{b}$  which maximises the projection variance is given by the eigenvector corresponding to the largest eigenvalue of  $\mathbf{S}$ . Under the criterion that the next optimal direction  $\mathbf{b}^{(2)}$  should be orthonormal to the first, one can readily show that  $\mathbf{b}^{(2)}$  is given by the second largest eigenvector, and so on. These maximal variance directions found by PCA are called the principal directions.

## PCA With Missing Data

- Often like to use PCA when there are elements of the data missing.
- There is no ‘quick fix’ PCA solution when some of the  $x_i^n$  are missing.
- One approach is to require the squared reconstruction error to be small only for the existing elements of  $\mathbf{X}$ . That is

$$E(\mathbf{B}, \mathbf{Y}) = \sum_{n=1}^N \sum_{i=1}^D \gamma_i^n \left[ x_i^n - \sum_j y_j^n b_i^j \right]^2$$

where  $\gamma_i^n = 1$  if the  $i^{th}$  entry of the  $n^{th}$  vector is available, and is zero otherwise.

- There are efficient iterative schemes to find  $\mathbf{B}$  and  $\mathbf{Y}$ .

---

## Matrix Completion

- Given a matrix  $\mathbf{X}$  with missing entries, we would like to ‘infer’ the missing values.
- If the elements of the matrix can be well approximated by a low-rank factorisation, we can use PCA to ‘fill in’ the missing values.

# PCA with missing data

Optimize  $\mathbf{Y}$  for fixed  $\mathbf{B}$

$$E(\hat{\mathbf{B}}, \mathbf{Y}) = \sum_{n=1}^N \sum_{i=1}^D \gamma_i^n \left[ x_i^n - \sum_j y_j^n \hat{b}_i^j \right]^2$$

For fixed  $\hat{\mathbf{B}}$  the above  $E(\hat{\mathbf{B}}, \mathbf{Y})$  is a quadratic function of the matrix  $\mathbf{Y}$ , which can be optimised directly. By differentiating and equating to zero, one obtains the fixed point condition

$$\sum_i \gamma_i^n \left( x_i^n - \sum_l y_l^n \hat{b}_i^l \right) \hat{b}_i^k = 0$$

Defining

$$\left[ \mathbf{y}^{(n)} \right]_l = y_l^n, \quad \left[ \mathbf{M}^{(n)} \right]_{kl} = \sum_i \hat{b}_i^l \hat{b}_i^k \gamma_i^n, \quad [\mathbf{c}^n]_k = \sum_i \gamma_i^n x_i^n \hat{b}_i^k,$$

in matrix notation, we then have a set of linear systems:

$$\mathbf{c}^{(n)} = \mathbf{M}^{(n)} \mathbf{y}^{(n)}, \quad n = 1, \dots, N$$

## PCA with missing data

Optimize  $\mathbf{B}$  for fixed  $\hat{\mathbf{Y}}$

One now freezes  $\hat{\mathbf{Y}}$  and considers the function

$$E(\mathbf{B}, \hat{\mathbf{Y}}) = \sum_{n=1}^N \sum_{i=1}^D \gamma_i^n \left[ x_i^n - \sum_j \hat{y}_j^n b_i^j \right]^2$$

For fixed  $\hat{\mathbf{Y}}$  the above expression is quadratic in the matrix  $\mathbf{B}$ , which can again be optimised using linear algebra. This corresponds to solving a set of linear systems for the  $i^{th}$  row of  $\mathbf{B}$ :

$$\mathbf{m}^{(i)} = \mathbf{F}^{(i)} \mathbf{b}^{(i)}$$

where

$$\left[ \mathbf{m}^{(i)} \right]_k = \sum_n \gamma_i^n x_i^n \hat{y}_k^n, \quad \left[ \mathbf{F}^{(i)} \right]_{kj} = \sum_n \gamma_i^n \hat{y}_j^n \hat{y}_k^n$$

Mathematically, this is  $\mathbf{b}^{(i)} = \mathbf{F}^{(i)-1} \mathbf{m}^{(i)}$ . In this manner one is guaranteed to iteratively decrease the value of the squared error loss until a minimum is reached.

## PCA With Missing Data: matrix completion



The training dataset consists of 400 images with randomly removed segments in each image.

## PCA With Missing Data: matrix completion



**Figure :** Left: original complete datapoint (one of the 400 in the original complete training set). Second image: data vector with missing entries. The new training set is composed of such images with randomly selected strips of the image missing. Third image: reconstruction of the missing data vector entries using 50 components. Right: reconstruction of the whole image (from the missing data images).

`demoSVDmissingFace.m`

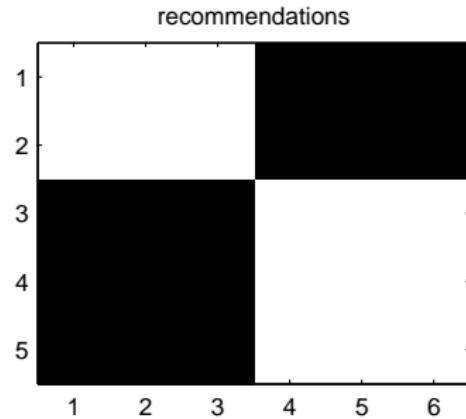
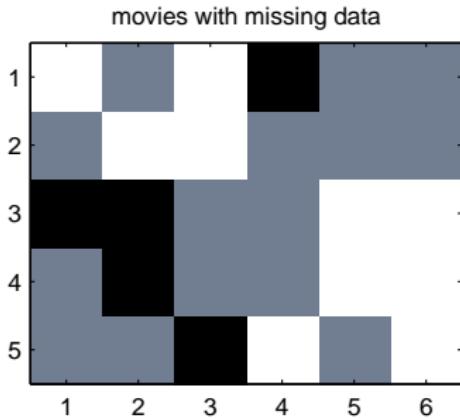
# Collaborative Filtering: Recommending Movies

	User					
	1	2	3	4	5	6
Terminator 1	Green		Green	Red		
Terminator 2		Green	Green			
Toy Story 1	Red	Red			Green	Green
Toy Story 2		Red			Green	Green
Invincibles			Red	Green		Green

This is a matrix completion problem.

# Collaborative Filtering: Recommending Movies

5 movies and 6 users:



Left: gray represents missing data. White is +1 (like), black is -1 (don't like).  
Right: Reconstruction using PCA with 2 components and taking the sign of the reconstructions.

[demoSVDmissingMovie.m](#)

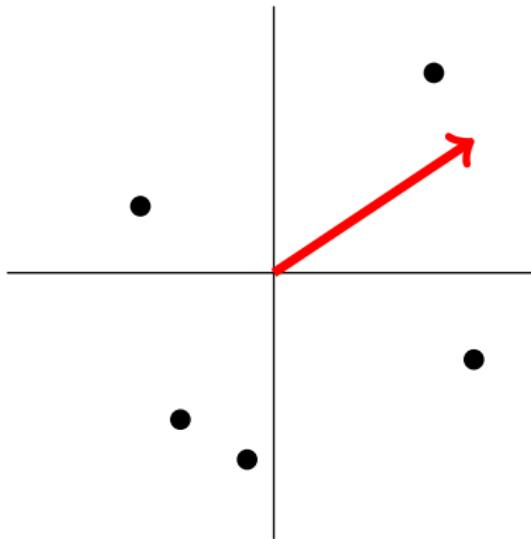
# Matrix Decompositions

Given a data matrix  $\mathbf{X}$  for which each column represents a datapoint, an approximate matrix decomposition is of the form  $\mathbf{X} \approx \mathbf{B}\mathbf{Y}$  into a basis matrix  $\mathbf{B}$  and weight (or coordinate) matrix  $\mathbf{Y}$ . Symbolically, matrix decompositions are of the form

$$\underbrace{\left( \begin{array}{c} X : \text{Data} \end{array} \right)}_{D \times N} \approx \underbrace{\left( \begin{array}{c} B : \text{Basis} \end{array} \right)}_{D \times M} \left( \begin{array}{c} Y : \text{Weights/Components} \end{array} \right) \underbrace{\quad}_{M \times N}$$

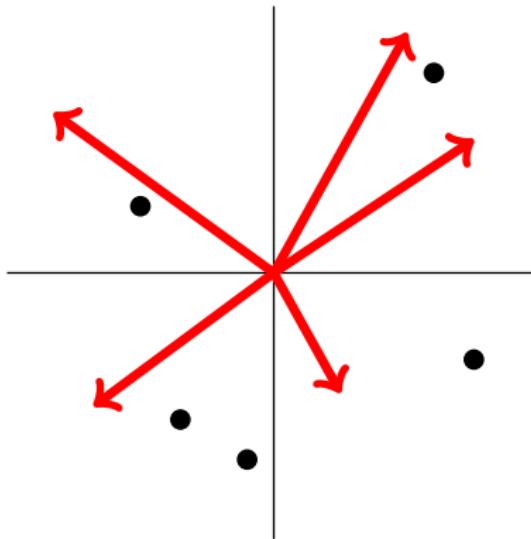
Based on the SVD of the data matrix, we see that PCA is in this class. Many methods can be considered as matrix decompositions under specific constraints.

## Under-complete decompositions



When  $M < D$ , there are fewer basis vectors than dimensions. The matrix  $\mathbf{B}$  is then called 'tall' or 'thin'. In this case the matrix  $\mathbf{Y}$  forms a lower dimensional approximate representation of the data  $\mathbf{X}$ , PCA being a classic example.

## Over-complete decompositions



For  $M > D$  the basis is over-complete, there being more basis vectors than dimensions. In such cases additional constraints are placed on either the basis or components. For example, one might require that only a small number of the large number of available basis vectors is used to form the representation for any given  $x$ . Such sparse-representations are common in theoretical neurobiology where issues of energy efficiency, rapidity of processing and robustness are of interest.

# Non-Negative Matrix Factorisation (NNMF)

$$\begin{matrix} \textcolor{blue}{\circ} & \textcolor{green}{\circ} & \textcolor{green}{\circ} & \textcolor{blue}{\circ} \\ \textcolor{green}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{green}{\circ} \\ \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{green}{\circ} & \textcolor{blue}{\circ} \\ \textcolor{blue}{\circ} & \textcolor{green}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} \\ \textcolor{green}{\circ} & \textcolor{green}{\circ} & \textcolor{green}{\circ} & \textcolor{green}{\circ} \end{matrix} = \begin{matrix} \textcolor{blue}{\circ} & \textcolor{green}{\circ} & \textcolor{green}{\circ} & \textcolor{blue}{\circ} \\ \textcolor{green}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} \\ \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} \\ \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} \\ \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} \end{matrix} + \begin{matrix} \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} \\ \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{green}{\circ} \\ \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{green}{\circ} & \textcolor{blue}{\circ} \\ \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} \\ \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} \end{matrix} + \begin{matrix} \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} & \textcolor{blue}{\circ} \\ \textcolor{green}{\circ} & \textcolor{green}{\circ} & \textcolor{green}{\circ} & \textcolor{green}{\circ} \end{matrix}$$

$\mathbf{x}$                      $\mathbf{b}^1$                      $\mathbf{b}^2$                      $\mathbf{b}^3$

- Can describe a positive data vector by adding up positive parts of a '2'.
- The parts can be more interpretable than those given by PCA.
- If applied to Item-Basket analysis, it would find essentially what kinds of things are bought together. For example a basis vector (or 'component') might be analogous to say 'vegetables' and another might be 'dairy'. We might then expect a customer basket to be able to be described by the amount that it contains vegetables and the amount it contains dairy.

# Non-Negative Matrix Factorisation

- In some cases it is more natural to consider that the data is composed of positive amounts of positive objects.
- We therefore seek an approximate decomposition

$$X_{ij} \approx \sum_k B_{ik} Y_{kj}$$

for non-negative  $B$  and non-negative  $Y$ .

- For example the columns of the data matrix  $\mathbf{X}$  might be images. Then the columns of the matrix  $\mathbf{B}$  are the ‘basis’ vectors that we will use to reconstruct these images. The positive weights of the reconstruction are given by  $\mathbf{Y}$ .

## NNMF: Probabilistic latent semantic analysis

Consider two objects,  $x$  and  $y$ , where  $\text{dom}(x) = \{1, \dots, I\}$  and  $\text{dom}(y) = \{1, \dots, J\}$  and a dataset  $(x^n, y^n, n = 1, \dots, N)$ . We have a count matrix with elements  $C_{ij}$  which describes the number of times the joint state  $x = i, y = j$  was observed in the dataset. We can transform this count matrix into a frequency matrix  $p$  with elements

$$p(x = i, y = j) = \frac{C_{ij}}{\sum_{ij} C_{ij}}$$

Our interest is to find a decomposition of this frequency matrix of the form

$$\underbrace{p(x = i, y = j)}_{X_{ij}} \approx \sum_k \underbrace{\tilde{p}(x = i | z = k)}_{B_{ik}} \underbrace{\tilde{p}(y = j | z = k)}_{Y_{kj}} \underbrace{\tilde{p}(z = k)}_{\tilde{p}(z = k)} \equiv \tilde{p}(x = i, y = j)$$

where all quantities  $\tilde{p}$  are distributions. This is then a form of matrix decomposition into positive basis  $\mathbf{B}$  and positive coordinates  $\mathbf{Y}$ . This has the interpretation of discovering latent topics  $z$  that describe the joint behaviour of  $x$  and  $y$ .

## NNMF: An EM style training algorithm

For probabilities, a useful measure of discrepancy is the Kullback-Leibler divergence

$$\text{KL}(p|\tilde{p}) = \langle \log p \rangle_p - \langle \log \tilde{p} \rangle_p$$

Since  $p$  is fixed, minimising the Kullback-Leibler divergence with respect to the approximation  $\tilde{p}$  is equivalent to maximising the ‘likelihood’ term  $\langle \log \tilde{p} \rangle_p$ . This is

$$L \equiv \sum_{x,y} p(x,y) \log \tilde{p}(x,y)$$

It's convenient to derive an EM style algorithm to learn  $\tilde{p}(x|z)$ ,  $\tilde{p}(y|z)$  and  $\tilde{p}(z)$ .

Consider

$$\begin{aligned} \text{KL}(q(z|x, y) | \tilde{p}(z|x, y)) \\ = \sum_z q(z|x, y) \log q(z|x, y) - \sum_z q(z|x, y) \log \tilde{p}(z|x, y) \geq 0 \end{aligned}$$

where  $\sum_z$  implies summation over all states of the variable  $z$ . Using

$$\tilde{p}(z|x, y) = \frac{\tilde{p}(x, y, z)}{\tilde{p}(x, y)}$$

and rearranging, this gives the bound,

$$\log \tilde{p}(x, y) \geq - \sum_z q(z|x, y) \log q(z|x, y) + \sum_z q(z|x, y) \log \tilde{p}(z|x, y)$$

Plugging this into the ‘likelihood’ term above, we have the bound

$$\begin{aligned} L &\geq - \sum_{x, y} p(x, y) \sum_z q(z|x, y) \log q(z|x, y) \\ &\quad + \sum_{x, y} p(x, y) \sum_z q(z|x, y) [\log \tilde{p}(x|z) + \log \tilde{p}(y|z) + \log \tilde{p}(z)] \end{aligned}$$

## NNMF: M-step

For fixed  $\tilde{p}(x|z), \tilde{p}(y|z)$ , the contribution to the bound from  $\tilde{p}(z)$  is

$$\sum_{x,y} p(x,y) \sum_z q(z|x,y) \log \tilde{p}(z)$$

Up to a constant, this is  $\text{KL}\left(\sum_{x,y} q(z|x,y)p(x,y)|\tilde{p}(z)\right)$  so that, optimally,

$$\tilde{p}(z) = \sum_{x,y} q(z|x,y)p(x,y)$$

Similarly, optimally

$$\tilde{p}(x|z) \propto \sum_y p(x,y)q(z|x,y)$$

and

$$\tilde{p}(y|z) \propto \sum_x p(x,y)q(z|x,y)$$

## NNMF: E-step

The optimal setting for the  $q$  distribution at each iteration is

$$q(z|x, y) = \tilde{p}(z|x, y)$$

which is fixed throughout the M-step.

---

### Convergence

$L$  is guaranteed to increase (and the Kullback-Leibler divergence decrease) under iterating between the E and M-steps, since the method is analogous to an EM procedure.

## NNMF: Conditional PLSA

In some cases it is more natural to consider a conditional frequency matrix

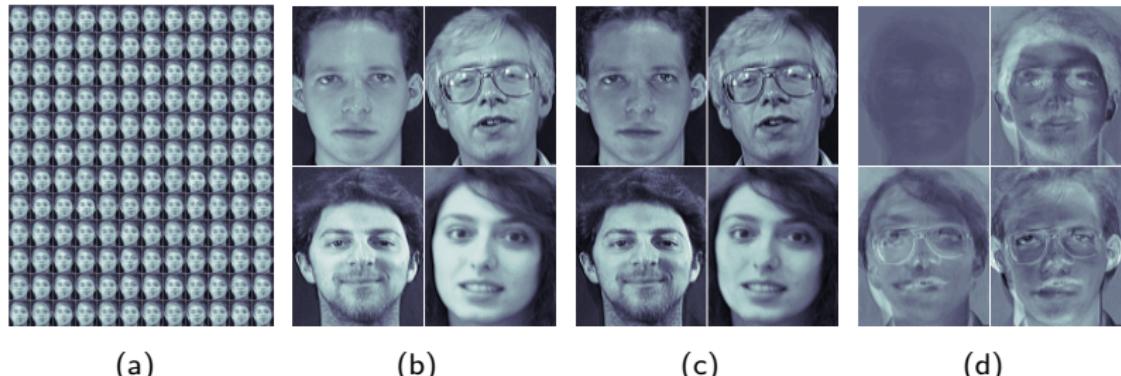
$$p(x = i|y = j)$$

and seek an approximate decomposition

$$\underbrace{p(x = i|y = j)}_{X_{ij}} \approx \sum_k \underbrace{\tilde{p}(x = i|z = k)}_{B_{ik}} \underbrace{\tilde{p}(z = k|y = j)}_{Y_{kj}}$$

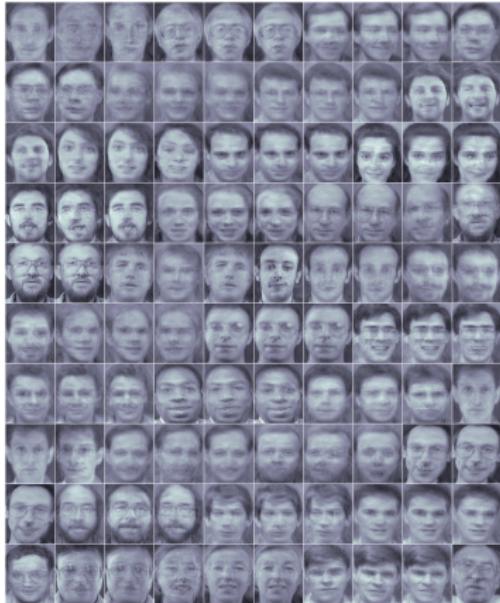
Deriving an EM style algorithm for this is straightforward, being equivalent to the non-negative matrix factorisation algorithm.

# Learning a positive basis

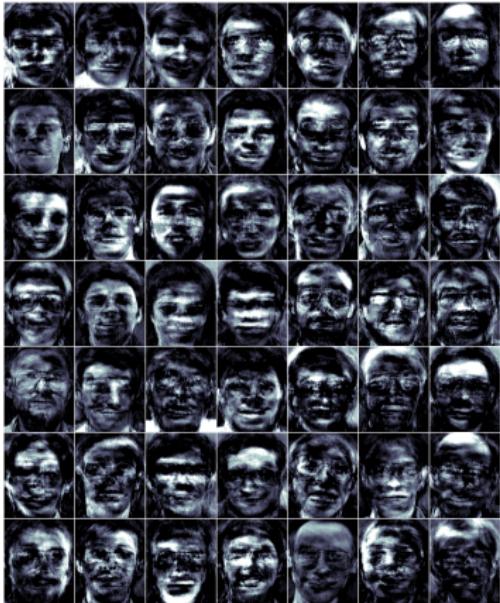


**Figure :** (a): Training data, consisting of a positive (convex) combination of the base images. (b): The chosen base images from which the training data is derived. (c): Basis learned using conditional PLSA on the training data. This is virtually indistinguishable from the true basis. (d): Eigenbasis (sometimes called 'eigenfaces').

## Positive reconstruction



(a)

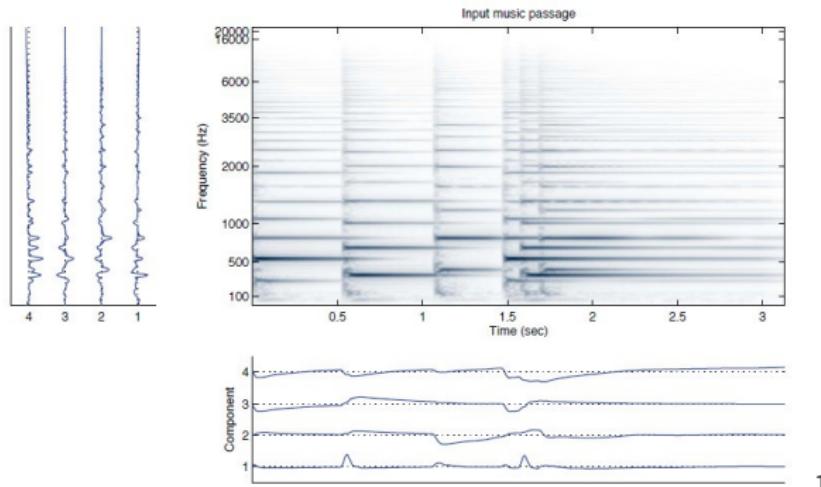


(b)

**Figure :** (a): Conditional PLSA reconstruction of the images in using a positive convex combination of the 49 positive base images in (b). The root mean square reconstruction error is  $1.391 \times 10^{-5}$ . The base images tend to be more 'localised' than the corresponding eigen-images. Here one sees local structure such as foreheads, chins, etc.

# Decomposing Music Signals: PCA

- Compute the spectrogram of a music signal – gives a matrix  $\mathbf{X}$  with elements  $M_{f,t}$ .
- Compute the PCA decomposition using 4 components.
- Doesn't give any nice interpretation.

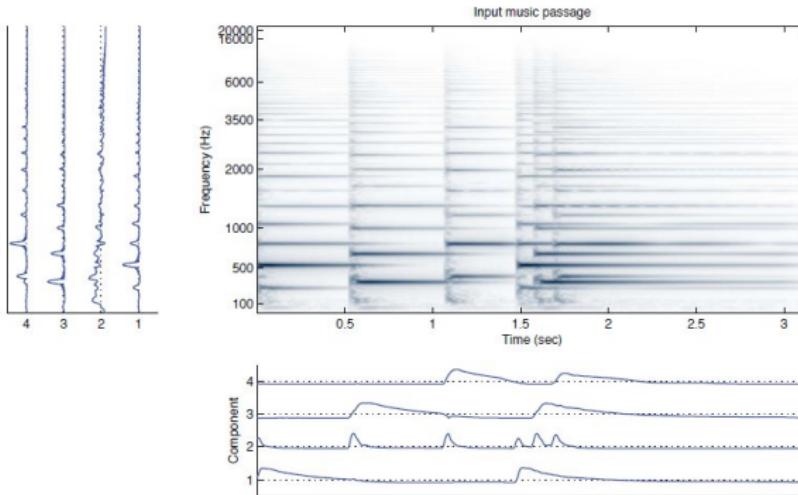


1

<sup>1</sup>Images from Paris Smaragdis WASPAA keynote 2013

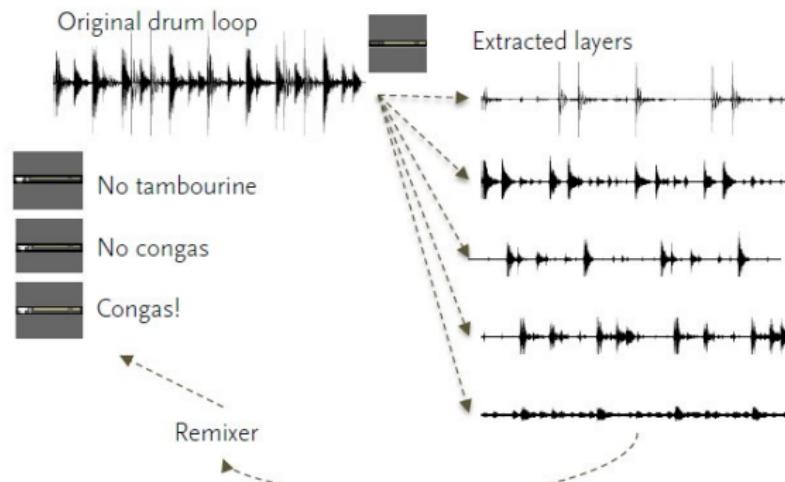
# Decomposing Music Signals: NNMF

- Compute the spectrogram of a music signal – gives a matrix  $\mathbf{X}$  with elements  $M_{f,t}$ .
- Compute the NNMF decomposition using 4 components.
- Gives a very nice interpretation – can see what each note corresponds to and when it is played.



# Decomposing Music Signals: NNMF

- Can use this to isolate the component instruments in music.
- Eg. isolate the individual drums in a piece.



See Paris' demo

# Topic Modelling

- Have a collection of documents.
- Each document may refer to more than one topic
- Eg. A document might discuss the role of climate change in Polynesia. climate change and Polynesia might be two topics that we would like to identify.
- We want to do this in an unsupervised way – no one tells us what the topics are. We just have a collection of unlabelled documents and want to discover automatically what are the ‘latent’ topics.
- Since a single document could contain more than one topic – ‘mixed membership’ model.

---

## Ways to do this

- NMF (also called PLSA in the topic modelling community) is very easy and fast to implement.
- Latent Dirichlet Allocation is an alternative approach which is very similar mathematically but difficult to implement – not clear what the practical advantages are over NMF.

## Frequency Representation

- Define a fixed dictionary of words  $d_1 = \text{aardvark}, \dots, d_{10,000} = \text{zorro}$ .
- For a given document  $n$  represent the document by the 10,000 dimensional frequency vector (number of occurrences of each word)

$$\mathbf{f}^n = (0, 0, 2, 0, 0, 1, \dots, 1, 0) \quad \text{This will be a very sparse vector.}$$

- This is also called a 'bag of words' representation.
- We may also wish to remove so-called 'stop words' (common words such as 'and', 'the', etc. which would otherwise dominate statistics and are anyway common to any topic).
- We may also wish to re-weight the frequencies using the so-called TF-IDF which means that we use instead a value that is high if the word appears many times in a document but gets scaled down if the same word appears in many documents.

# Topic Modelling Example: NNMF

16,333 documents are taken from Associated Press corpus with a dictionary of 23,075 unique terms. Fit a topic model (NNMF) containing 4 topics.

Arts	Budgets	Children	Education
new	million	children	school
film	tax	women	students
show	program	people	schools
music	budget	child	education
movie	billion	years	teachers
play	federal	families	high
musical	year	work	public
best	spending	parents	teacher
actor	new	says	bennett
first	state	family	manigat
york	plan	welfare	namphy
opera	money	men	state
theater	programs	percent	president
actress	government	care	elementary
love	congress	life	haiti

(a)

The William Randolph Hearst Foundation will give \$ 1.25 million to Lincoln Center, Metropolitan Opera Co., New York Philharmonic and Juilliard School. Our board felt that we had a real opportunity to make a mark on the future of the performing arts with these grants an act every bit as important as our traditional areas of support in health, medical research, education and the social services, Hearst Foundation President Randolph A. Hearst said Monday in announcing the grants. Lincoln Centers share will be \$200,000 for its new building, which will house young artists and provide new public facilities. The Metropolitan Opera Co. and New York Philharmonic will receive \$400,000 each. The Juilliard School, where music and the performing arts are taught, will get \$250,000. The Hearst Foundation, a leading supporter of the Lincoln Center Consolidated Corporate Fund, will make its usual annual \$100,000 donation, too.

(b)

# Modelling citations

We have a collection of research documents which cite other documents. For example, document 1 might cite documents 3, 2, 10, etc. Given only the list of citations for each document, can we identify key research papers and the communities that cite them?

---

## A probabilistic formulation

We use the variable  $d \in \{1, \dots, D\}$  to index documents and  $c \in \{1, \dots, D\}$  to index citations (both  $d$  and  $c$  have the same domain, namely the index of a research article). If document  $d = i$  cites article  $c = j$  then we set the entry of the matrix  $C_{ij} = 1$ . If there is no citation,  $C_{ij}$  is set to zero. We can form a 'distribution' over documents and citations using

$$p(d = i, c = j) = \frac{C_{ij}}{\sum_{ij} C_{ij}}$$

and use PLSA to decompose this matrix into citation-topics.

## Modelling citations

The Cora corpus contains an archive of around 30,000 computer science research papers. From this archive the papers in the machine learning category are extracted, consisting of 4220 documents and 38,372 citations.

---

### Using PLSA

The joint PLSA method is fitted to the data using  $z = 7$  topics. From the trained model the expression  $p(c = j|z = k)$  defines how authoritative paper  $j$  is according to community  $z = k$ .

# Modelling citations

factor 1 0.0108 0.0066 0.0065	(Reinforcement Learning) Learning to predict by the methods of temporal differences. Sutton. Neuronlike adaptive elements that can solve difficult learning control problems. Barto et al. Practical Issues in Temporal Difference Learning. Tesauro.
factor 2 0.0038 0.0037 0.0036	(Rule Learning) Explanation-based generalization: a unifying view. Mitchell et al. Learning internal representations by error propagation. Rumelhart et al. Explanation-Based Learning: An Alternative View. DeJong et al.
factor 3 0.0120 0.0061 0.0049	(Neural Networks) Learning internal representations by error propagation. Rumelhart et al. Neural networks and the bias-variance dilemma. Geman et al. The Cascade-Correlation learning architecture. Fahlman et al.
factor 4 0.0093 0.0066 0.0055	(Theory) Classification and Regression Trees. Breiman et al. Learnability and the Vapnik-Chervonenkis dimension. Blumer et al. Learning Quickly when Irrelevant Attributes Abound. Littlestone.
factor 5 0.0118 0.0094 0.0056	(Probabilistic Reasoning) Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Pearl. Maximum likelihood from incomplete data via the em algorithm. Dempster et al. Local computations with probabilities on graphical structures. Lauritzen et al.
factor 6 0.0157 0.0132 0.0096	(Genetic Algorithms) Genetic Algorithms in Search, Optimization, and Machine Learning. Goldberg. Adaptation in Natural and Artificial Systems. Holland. Genetic Programming: On the Programming of Computers by Means of Natural Selection. Koza.
factor 7 0.0063 0.0054 0.0033	(Logic) Efficient induction of logic programs. Muggleton et al. Learning logical definitions from relations. Quinlan. Inductive Logic Programming Techniques and Applications. Lavrac et al.

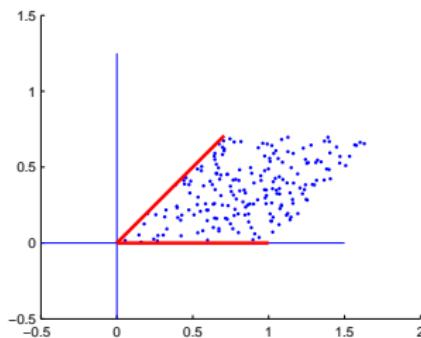
Table : Highest ranked documents according to  $p(c|z)$ . The factor topic labels are manual assignments based on similarity to the Cora topics.

# Independent Components Analysis

- As we saw, PCA finds interesting subspaces, but does not really find interesting directions in the subspace.
- A useful coordinate system would be one in which the data we have could be generated by sampling independently along directions.
- ICA finds ‘basis’ vectors  $\mathbf{b}^i$  and coefficients  $y$  such that the data can be approximately expressed as

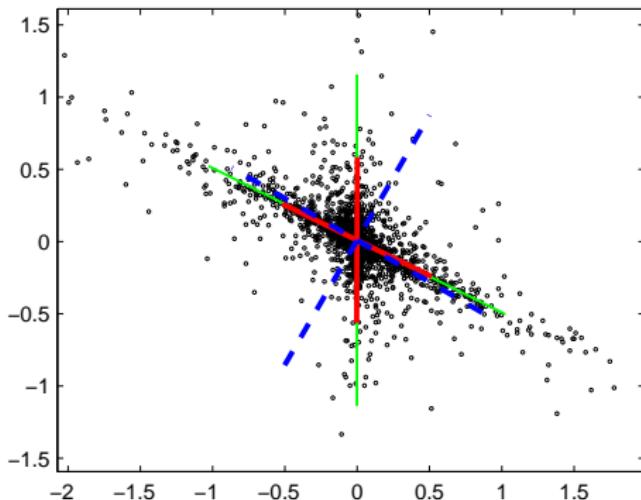
$$\mathbf{x}^n \approx \sum_i y_i^n \mathbf{b}^i$$

where empirically the  $y_i^n$  are uncorrelated.



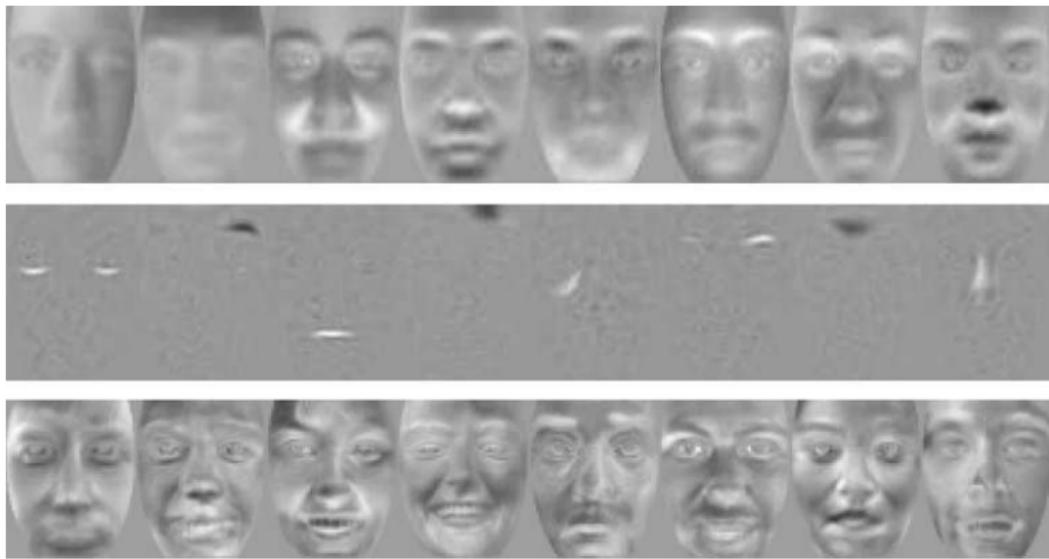
We can generate these datapoints by drawing a random value  $y_1$  from the uniform distribution and independently drawing  $y_2$  from the uniform distribution. A datapoint is then given by taking a linear combination  $y_1 \mathbf{b}^1 + y_2 \mathbf{b}^2$  of the two red basis vectors.

## ICA versus PCA



**Figure :** The red lines are the basis vectors estimated by ICA. For comparison, PCA produces the blue (dashed) components. As expected, PCA finds the orthogonal directions of maximal variation. ICA however, correctly estimates the directions in which the components were independently generated.

## ICA versus PCA for faces



**Figure :** The top row are the 8 largest principal components. The second row is ICA in which the coefficients  $y$  are independent – this identifies things like the eyes, mouth, nose. The bottom row is ICA such that the basis vector elements are independent. From Draper et al, 2003.

# ICA applications

## Training

- ICA is often accomplished using the FastICA package.
- 

## Applications

- Finding independent causes in time series data.
- Often used in audio processing.
- Commonly also used to find the components of natural images.
- Finding independent ‘purchase directions’ in basket-item data.

## ICA audio unmixing

# Summary

Dimension reduction looks for low dimensional structure in data and can be used to compress, interpret and predict missing values. It is often also used a preprocessing step to remove irrelevant ‘noise’ in the data.

---

## PCA

- Most common approach ●
  - Fast to use – efficient scalable algorithms exist for sparse data ●
  - Versions exist for missing data ●
  - Hard to interpret the components ●
- 

## NNMF

- Constrained PCA with the basis vectors and their weights all positive.
- Can be more interpretable than PCA ●
- Interesting applications in text/topic modelling (and elsewhere) ●

# State of the Art

## Deep Learning

- Also called Neural Nets – we will discuss more in another section
  - Can give very good results 
  - Can be hard to train 
- 

## Tensor Factorisation

- Not really tensors (in the differential geometry sense) but multi-dimensional arrays
  - E.g decompose  $X_{ijk} \approx \sum_{mn} A_{im} B_{mjn} C_{nk}$
- 

## Topic Modelling

- Latent Dirichlet Allocation is a probabilistic model that is popular in topic modelling.
- LDA does not really account well for the typical word distribution (Zipf's law) that is common of English language texts. More recent approaches attempt to address this.

# Independent Components Analysis<sup>1</sup>

David Barber

University College London

---

<sup>1</sup> These slides accompany the book *Bayesian Reasoning and Machine Learning*. The book and demos can be downloaded from [www.cs.ucl.ac.uk/staff/D.Barber/bml](http://www.cs.ucl.ac.uk/staff/D.Barber/bml). Feedback and corrections are also available on the site. Feel free to adapt these slides for your own purposes, but please include a link to the above website.

# Independent Components Analysis

We seek a linear coordinate system in which the coordinates are independent. Such independent coordinate systems arguably form a natural representation of the data and can give rise to very different representations than PCA (which assumes the directions are orthogonal).

$$p(\mathbf{v}, \mathbf{h} | \mathbf{A}) = p(\mathbf{v} | \mathbf{h}, \mathbf{A}) \prod_i p(h_i)$$

For technical reasons, the most convenient practical choice is to use

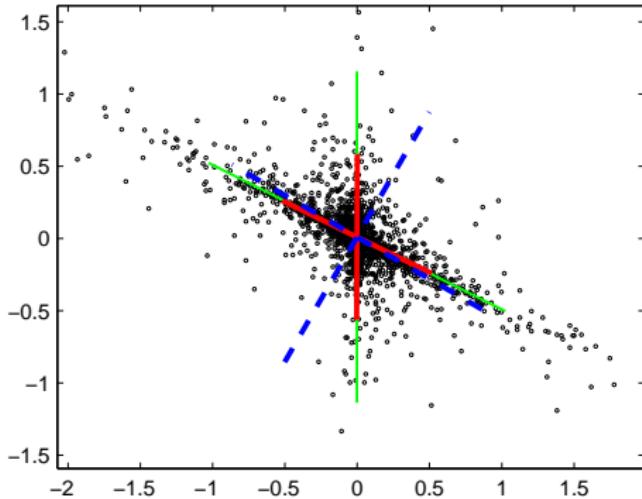
$$\mathbf{v} = \mathbf{A}\mathbf{h}$$

where  $\mathbf{A}$  is a square mixing matrix so that the likelihood of an observation  $\mathbf{v}$  is

$$p(\mathbf{v}) = \int p(\mathbf{v} | \mathbf{h}, \mathbf{A}) \prod_i p(h_i) d\mathbf{h} = \frac{1}{|\det(\mathbf{A})|} \prod_i p([\mathbf{A}^{-1}\mathbf{v}]_i)$$

The underlying independence assumptions are then the same as for PPCA (in the limit of zero output noise). Below, however, we will choose a non-Gaussian prior  $p(h_i)$ .

# ICA versus PCA



**Figure:** Latent data is sampled from the prior  $p(x_i) \propto \exp(-5\sqrt{|x_i|})$  with the mixing matrix  $\mathbf{A}$  shown in green to create the observed two dimensional vectors  $\mathbf{y} = \mathbf{Ax}$ . The red lines are the mixing matrix estimated by `ica.m` based on the observations. For comparison, PCA produces the blue (dashed) components. Note that the components have been scaled to improve visualisation. As expected, PCA finds the orthogonal directions of maximal variation. ICA however, correctly estimates the directions in which the components were independently generated.

# Maximum Likelihood

For a given set of data  $\mathcal{V} = (\mathbf{v}^1, \dots, \mathbf{v}^N)$  and prior  $p(h)$ , our aim is to find  $\mathbf{A}$ . For i.i.d. data, the log likelihood is conveniently written in terms of  $\mathbf{B} = \mathbf{A}^{-1}$ ,

$$L(\mathbf{B}) = N \log \det(\mathbf{B}) + \sum_n \sum_i \log p([\mathbf{B}\mathbf{v}^n]_i)$$

---

## Rotational invariance for a Gaussian prior

Note that for a Gaussian prior

$$p(h) \propto e^{-h^2}$$

the log likelihood becomes

$$L(\mathbf{B}) = N \log \det(\mathbf{B}) - \sum_n (\mathbf{v}^n)^T \mathbf{B}^T \mathbf{B} \mathbf{v}^n + \text{const.}$$

which is invariant with respect to an orthogonal rotation  $\mathbf{B} \rightarrow \mathbf{R}\mathbf{B}$ , with  $\mathbf{R}^T \mathbf{R} = \mathbf{I}$ . This means that for a Gaussian prior  $p(h)$ , we cannot estimate uniquely the mixing matrix. To break this rotational invariance we therefore need to use a non-Gaussian prior.

# Maximum Likelihood

Assuming we have a non-Gaussian prior  $p(h)$ , taking the derivative w.r.t.  $B_{ab}$  we obtain

$$\frac{\partial}{\partial B_{ab}} L(\mathbf{B}) = N A_{ba} + \sum_n \phi([\mathbf{Bv}]_a) v_b^n$$

where

$$\phi(x) \equiv \frac{d}{dx} \log p(x) = \frac{1}{p(x)} \frac{d}{dx} p(x)$$

A simple gradient ascent learning rule for  $\mathbf{B}$  is then

$$\mathbf{B}^{new} = \mathbf{B} + \eta \left( \mathbf{B}^{-T} + \frac{1}{N} \sum_n \phi(\mathbf{Bv}^n) (\mathbf{v}^n)^T \right)$$

# Natural Gradient

An alternative ‘natural gradient’ algorithm is given by multiplying the gradient by  $\mathbf{B}^T \mathbf{B}$  on the right to give the update

$$\mathbf{B}^{new} = \mathbf{B} + \eta \left( \mathbf{I} + \frac{1}{N} \sum_n \phi(\mathbf{Bv}^n) (\mathbf{Bv}^n)^T \right) \mathbf{B}$$

Here  $\eta$  is an empirically set learning rate.

The standard derivation of Natural Gradient for ICA is somewhat opaque.

# Natural Gradient Derivation

Consider a modified objective

$$\tilde{L}(\mathbf{B}) = N \log \det(\mathbf{B})$$

which neglects the second term of the standard objective. Then applying Newton's method to this objective, one can show that the ICA Natural Gradient update is equivalent to

$$\eta \tilde{H}^{-1} \mathbf{g}$$

where  $\tilde{H}$  is the Hessian of the above modified objective and  $\mathbf{g}$  is the gradient of the standard objective.

# ICA in practice

## Choosing the non-linearity

In practice it is common to use the non-linearity

$$\phi(x) = x + c \tanh(x)$$

where  $c = 1$  for a super-Gaussian (fatter tails than a Gaussian) component and  $c = -1$  for a sub-Gaussian (thinner tails than a Gaussian) component. There are automatic ways to determine this.

---

## fast ICA

A popular alternative estimation method is FastICA and can be related to an iterative Maximum Likelihood optimisation procedure, based on an approximate Newton procedure.

# Supervised Dimension Reduction

David Barber

University College London

## Supervised Linear Projections

- In cases where class information is available, and our ultimate interest is to reduce dimensionality for improved classification, it makes sense to use the available class information in forming the projection.
- We consider data from two different classes. For class 1, we have a set of  $N_1$  datapoints

$$\mathcal{X}_1 = \left\{ \mathbf{x}_1^1, \dots, \mathbf{x}_1^{N_1} \right\}$$

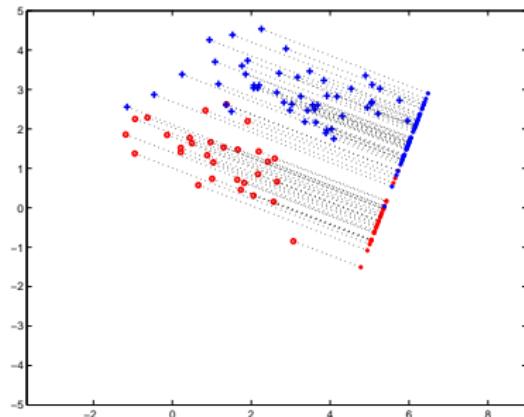
and similarly for class 2, we have a set of  $N_2$  datapoints

$$\mathcal{X}_2 = \left\{ \mathbf{x}_2^1, \dots, \mathbf{x}_2^{N_2} \right\}$$

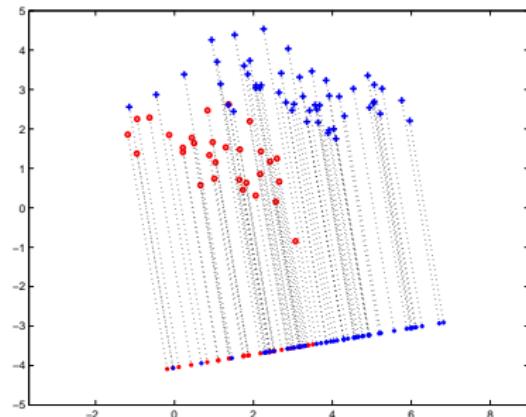
Our interest is then to find a linear projection,

$$\mathbf{y} = \mathbf{W}^T \mathbf{x}$$

where  $\dim \mathbf{W} = D \times L$ ,  $L < D$ , such that for two datapoints  $\mathbf{x}^i$  and  $\mathbf{x}^j$  in the same class, the distance between their projections  $\mathbf{y}^i$  and  $\mathbf{y}^j$  should be small.



(a)



(b)

**Figure :** The large crosses represent data from class 1, and the large circles from class 2. Their projections onto 1 dimension are represented by their small counterparts. **(a):** Fisher's Linear Discriminant Analysis. Here there is little class overlap in the projections. **(b):** Unsupervised dimension reduction using Principal Components Analysis for comparison. There is considerable class overlap in the projection. In both (a) and (b) the one dimensional projection is the distance along the line, measured from an arbitrary chosen fixed point on the line.

## Fisher's Linear Discriminant

We model the data from each class with a Gaussian. That is

$$p(\mathbf{x}_1) = \mathcal{N}(\mathbf{x}_1 | \mathbf{m}_1, \mathbf{S}_1), \quad p(\mathbf{x}_2) = \mathcal{N}(\mathbf{x}_2 | \mathbf{m}_2, \mathbf{S}_2)$$

where  $\mathbf{m}_1$  is the sample mean of class 1 data, and  $\mathbf{S}_1$  the sample covariance; similarly for class 2. The projections of the points from the two classes are then given by

$$y_1^n = \mathbf{w}^T \mathbf{x}_1^n, \quad y_2^n = \mathbf{w}^T \mathbf{x}_2^n$$

Because the projections are linear, the projected distributions are also Gaussian,

$$\begin{aligned} p(y_1) &= \mathcal{N}(y_1 | \mu_1, \sigma_1^2), & \mu_1 &= \mathbf{w}^T \mathbf{m}_1, & \sigma_1^2 &= \mathbf{w}^T \mathbf{S}_1 \mathbf{w} \\ p(y_2) &= \mathcal{N}(y_2 | \mu_2, \sigma_2^2), & \mu_2 &= \mathbf{w}^T \mathbf{m}_2, & \sigma_2^2 &= \mathbf{w}^T \mathbf{S}_2 \mathbf{w} \end{aligned}$$

We search for a projection  $\mathbf{w}$  such that the projected distributions have minimal overlap. A useful objective function therefore is

$$\frac{(\mu_1 - \mu_2)^2}{\pi_1 \sigma_1^2 + \pi_2 \sigma_2^2}$$

where  $\pi_i$  represents the fraction of the dataset in class  $i$ .

In terms of the projection  $\mathbf{w}$ , the objective is

$$F(\mathbf{w}) = \frac{\mathbf{w}^T (\mathbf{m}_1 - \mathbf{m}_2) (\mathbf{m}_1 - \mathbf{m}_2)^T \mathbf{w}}{\mathbf{w}^T (\pi_1 \mathbf{S}_1 + \pi_2 \mathbf{S}_2) \mathbf{w}} = \frac{\mathbf{w}^T \mathbf{A} \mathbf{w}}{\mathbf{w}^T \mathbf{B} \mathbf{w}}$$

where

$$\mathbf{A} = (\mathbf{m}_1 - \mathbf{m}_2) (\mathbf{m}_1 - \mathbf{m}_2)^T, \quad \mathbf{B} = \pi_1 \mathbf{S}_1 + \pi_2 \mathbf{S}_2$$

The optimal  $\mathbf{w}$  can be found by differentiating:

$$\frac{\partial}{\partial \mathbf{w}} \frac{\mathbf{w}^T \mathbf{A} \mathbf{w}}{\mathbf{w}^T \mathbf{B} \mathbf{w}} = \frac{2}{(\mathbf{w}^T \mathbf{B} \mathbf{w})^2} [(\mathbf{w}^T \mathbf{B} \mathbf{w}) \mathbf{A} \mathbf{w} - (\mathbf{w}^T \mathbf{A} \mathbf{w}) \mathbf{B} \mathbf{w}]$$

and therefore the zero derivative requirement is

$$(\mathbf{w}^T \mathbf{B} \mathbf{w}) \mathbf{A} \mathbf{w} = (\mathbf{w}^T \mathbf{A} \mathbf{w}) \mathbf{B} \mathbf{w}$$

Multiplying by the inverse of  $\mathbf{B}$  we have

$$\mathbf{B}^{-1} (\mathbf{m}_1 - \mathbf{m}_2) (\mathbf{m}_1 - \mathbf{m}_2)^T \mathbf{w} = \frac{\mathbf{w}^T \mathbf{A} \mathbf{w}}{\mathbf{w}^T \mathbf{B} \mathbf{w}} \mathbf{w}$$

Since  $(\mathbf{m}_1 - \mathbf{m}_2)^T \mathbf{w}$  is a scalar, the optimal projection is explicitly given by

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

## When the naive method breaks down

- The above derivation relied on the existence of the inverse of  $\mathbf{B}$ .
- A case where  $\mathbf{B}$  is not invertible is when there are fewer datapoints  $N_1 + N_2$  than dimensions  $D$ .
- A related problematic case is when there are elements of the input vectors that never vary. For example, in the hand-written digits case, the pixels at the corner edges are actually always zero. Let's call such a pixel  $z$ . The matrix  $\mathbf{B}$  will then have a zero entry for  $[B]_{z,z}$  (indeed the whole  $z^{th}$  row and column of  $\mathbf{B}$  will be zero) so that for any vector of the form

$$\mathbf{w}^T = (0, 0, \dots, w_z, 0, 0, \dots, 0) \Rightarrow \mathbf{w}^T \mathbf{B} \mathbf{w} = 0$$

This shows that the denominator of Fisher's objective can become zero, and the objective ill defined.

## Canonical Variates

Canonical Variates generalises Fisher's method to projections of more than one dimension and more than two classes. The projection of any point is given by

$$\mathbf{y} = \mathbf{W}^T \mathbf{x}$$

where  $\mathbf{W}$  is a  $D \times L$  matrix. Assuming that the data  $\mathbf{x}$  from class  $c$  is Gaussian distributed,

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x} | \mathbf{m}_c, \mathbf{S}_c)$$

the projections  $\mathbf{y}$  are also Gaussian

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{y} | \mathbf{W}^T \mathbf{m}_c, \mathbf{W}^T \mathbf{S}_c \mathbf{W})$$

Find the mean  $\mathbf{m}$  of the whole dataset and  $\mathbf{m}_c$ , the mean of the each class  $c$ . Form

$$\mathbf{A} \equiv \sum_{c=1}^C N_c (\mathbf{m}_c - \mathbf{m}) (\mathbf{m}_c - \mathbf{m})^T$$

where  $N_c$  is the number of datapoints in class  $c$ .

Compute the covariance matrix  $\mathbf{S}_c$  of the data for each class  $c$ . Define

$$\mathbf{B} \equiv \sum_{c=1}^C N_c \mathbf{S}_c$$

Assuming  $\mathbf{B}$  is invertible we can define the Cholesky factor  $\tilde{\mathbf{B}}$ , with

$$\tilde{\mathbf{B}}^\top \tilde{\mathbf{B}} = \mathbf{B}$$

A natural objective is then to maximise

$$F(\mathbf{W}) \equiv \frac{\text{trace} \left( \mathbf{W}^\top \tilde{\mathbf{B}}^{-\top} \mathbf{A} \tilde{\mathbf{B}}^{-1} \mathbf{W} \right)}{\text{trace} (\mathbf{W}^\top \mathbf{W})}$$

If we assume an orthonormality constraint on  $\mathbf{W}$ , then we equivalently require the maximisation of

$$F(\mathbf{W}) \equiv \text{trace} (\mathbf{W}^\top \mathbf{C} \mathbf{W}), \text{ subject to } \mathbf{W}^\top \mathbf{W} = \mathbf{I}$$

where

$$\mathbf{C} \equiv \frac{1}{D} \tilde{\mathbf{B}}^{-\top} \mathbf{A} \tilde{\mathbf{B}}^{-1}$$

Since  $\mathbf{C}$  is symmetric and positive semidefinite, it has a real eigen-decomposition

$$\mathbf{C} = \mathbf{E}\Lambda\mathbf{E}^T$$

where  $\Lambda = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_D)$  is diagonal with non-negative entries containing the eigenvalues, sorted by decreasing order,  $\lambda_1 \geq \lambda_2 \geq \dots$  and  $\mathbf{E}^T\mathbf{E} = \mathbf{I}$ . Hence

$$F(\mathbf{W}) = \text{trace}(\mathbf{W}^T\mathbf{E}\Lambda\mathbf{E}^T\mathbf{W})$$

- By setting  $\mathbf{W} = [\mathbf{e}_1, \dots, \mathbf{e}_L]$ , where  $\mathbf{e}_l$  is the  $l^{th}$  eigenvector, the objective  $F(\mathbf{W})$  becomes the sum of the first  $L$  eigenvalues.
- This setting maximises the objective function since forming  $\mathbf{W}$  from any other columns of  $\mathbf{E}$  would give a lower sum.
- Note that since  $\mathbf{A}$  has rank  $C$ , there can be no more than  $C - 1$  non-zero eigenvalues and corresponding directions.

# Canonical Variates Algorithm

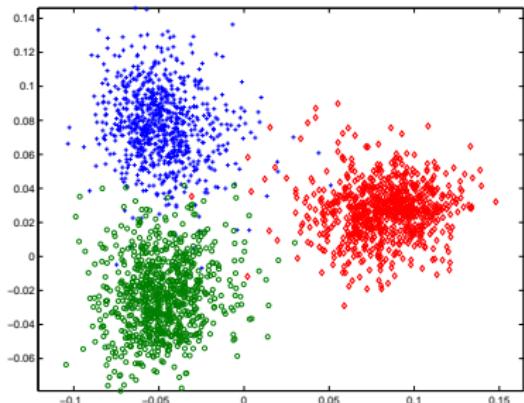
---

## Algorithm 1 Canonical Variates

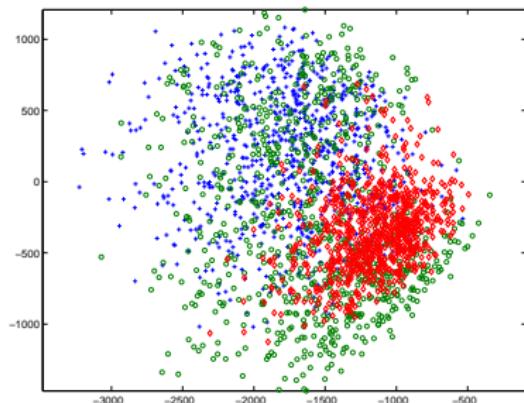
---

- 1: Compute the between and within class scatter matrices  $\mathbf{A}$ , and  $\mathbf{B}$ .
  - 2: Compute the Cholesky factor  $\tilde{\mathbf{B}}$  of  $\mathbf{B}$ .
  - 3: Compute the  $L$  principal eigenvectors  $[\mathbf{e}_1, \dots, \mathbf{e}_L]$  of  $\tilde{\mathbf{B}}^{-T} \mathbf{A} \tilde{\mathbf{B}}^{-1}$ .
  - 4: Return  $\mathbf{W} = [\mathbf{e}_1, \dots, \mathbf{e}_L]$  as the projection matrix.
-

# Using canonical variates on the digit data



(a)



(b)

**Figure :** (a): Canonical Variates projection of examples of handwritten digits 3('+'), 5('o') and 7(diamond). There are 800 examples from each digit class. Plotted are the projections down to 2 dimensions. (b): PCA projections for comparison.

## Dealing with the nullspace

- One may encounter situations where  $\mathbf{B}$  is not invertible.
- A solution is to require that  $\mathbf{W}$  lies only in the subspace spanned by the data (that is there can be no contribution from the nullspace).
- To do this we first concatenate the training data from all classes into one large matrix  $\mathbf{X}$ .
- A basis for  $\mathbf{X}$  can be found using, for example, the thin-SVD technique which returns an orthonormal non-square basis matrix  $\mathbf{Q}$ .
- We then require the solution  $\mathbf{W}$  to be expressed in this basis

$$\mathbf{W} = \mathbf{Q}\mathbf{W}'$$

for some matrix  $\mathbf{W}'$ .

Substituting this in the Canonical Variates objective we obtain

$$F(\mathbf{W}') \equiv \frac{\text{trace}(\mathbf{W}'^T \mathbf{Q}^T \mathbf{A} \mathbf{Q} \mathbf{W}')}{\text{trace}(\mathbf{W}'^T \mathbf{Q}^T \mathbf{B} \mathbf{Q} \mathbf{W}')}$$

This is of the same form as the standard quotient on replacing the between-scatter  $\mathbf{A}$  with

$$\mathbf{A}' \equiv \mathbf{Q}^T \mathbf{A} \mathbf{Q}$$

and the within-scatter  $\mathbf{B}$  with

$$\mathbf{B}' \equiv \mathbf{Q}^T \mathbf{B} \mathbf{Q}$$

In this case  $\mathbf{B}'$  is guaranteed invertible since  $\mathbf{B}$  is projected down to the basis that spans the data. One may then carry out Canonical Variates, as above, which returns the matrix  $\mathbf{W}'$ . Transforming, back,  $\mathbf{W}$  is then given by  $\mathbf{W} = \mathbf{Q} \mathbf{W}'$ .

# Clustering

David Barber

# Learning Objective

## Lectures

- Understanding of Unsupervised Learning and Clustering<sup>1</sup>.
- Familiarity and ability to apply K-means to cluster data. Understanding of the limitations of K-means.
- Understanding of Gaussian Mixture Models
- Understanding of more general mixture models, and how to deal with categorical data.
- How to deal with missing data in clustering.

---

## Practicals

- Clustering product data
- Clustering of questionnaire data, including missing data.

---

<sup>1</sup>Some material inspired by Piyush Rai, CS Utah

# Clustering

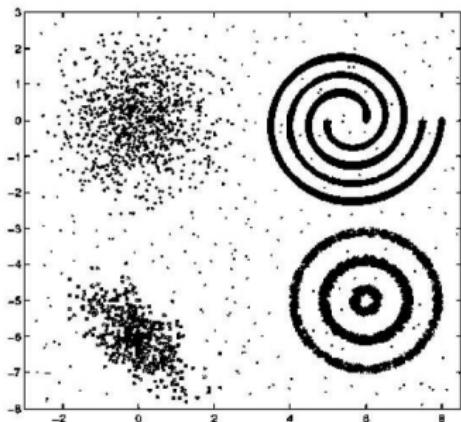
## What is it?

- Data is unlabelled – a form of unsupervised learning.
  - Want to find compact descriptions of data.
- 

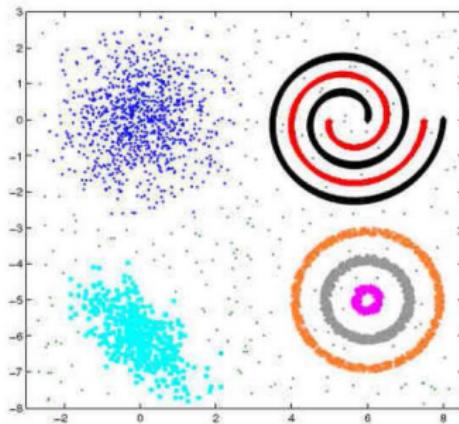
## Applications

- My clients can be considered to belong to 5 different groups, with a typical client from each group looking like this.
- Clustering webpages based on their content.
- Clustering web-search results.
- Clustering people in social networks based on user properties/preferences
- More generally, we want to reduce the apparent complexity to gain insight into the structure of the data.

# Clustering



(a) Input data



(b) Desired clustering

- Want clusters such that datapoints within the same cluster have high similarity<sup>2</sup>
- Want clusters such that datapoints from different clusters have low similarity.
- Different definitions of 'dissimilarity' - most common is the squared distance between points.

<sup>2</sup>Data Clustering: 50 Years Beyond K-Means, A.K. Jain (2008)

## Basket-Item Matrix

		Basket					
		1	2	3	4	5	6
Item	eggs	●		●	●		
	bacon		●	●			
	milk	●	●	●		●	●
	beer	●	●	●		●	
	tea	●		●	●		●

- Data may come in list form (a list of items in each basket).
- Often useful to represent this in matrix form.
- Typically a very sparse matrix.

# A priori algorithm for frequent item sets

		Basket					
		1	2	3	4	5	6
Item	eggs	●		●	●		
	bacon		●	●			
	milk	●	●	●		●	●
	beer	●	●	●		●	
	tea	●		●	●		●

- Start by finding frequent single items, then pairs of frequent items, triplets, etc.
- For a threshold of 3, {Eggs}, {Milk}, {Beer}, {Tea}, {Eggs,Tea}, {Milk, Beer}, {Milk,Tea} are frequent.
- This is a classical topic in Data Mining (finding association rules).
- In ‘clustering’ we want something a bit different – we want to find baskets that look similar and group them together typically into a single ‘cluster basket’. Each basket will belong to only one ‘cluster basket’.

# Clustering

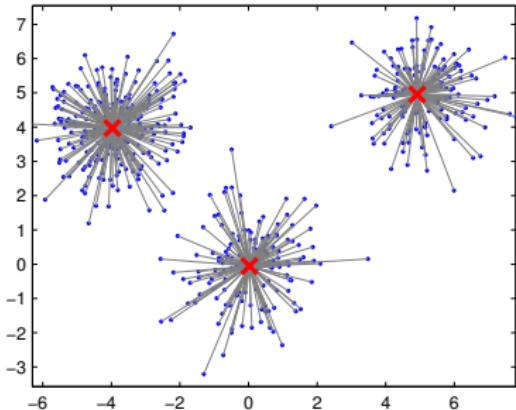
## Example problems

- We have a basket-item matrix in which each basket contains a list of items purchased on a visit. Are there typical groupings of baskets?
  - We have a questionnaire and each customer answers each of the 100 questions in the questionnaire. Are there certain kinds of customers that we can find?
- 

## Solutions

- K-means (classical approach): Fast to run but not always appropriate.
- Gaussian Mixture Models: Popular and most appropriate when the data in each cluster is Gaussian distributed.
- More general mixture models: can naturally handle discrete observations and missing data.
- Mixed Membership style models: useful when it is not appropriate to describe a customer in terms of membership of a single group (customer might be a little bit this, a little bit that).

# K-means



- We have a dataset  $\mathbf{x}^1, \dots, \mathbf{x}^N$  of data vectors ( $N$  customers).
- Want to segment these customers by assigning each datapoint to one of  $K$  groups (or ‘centres’),  $\mathbf{m}^1, \dots, \mathbf{m}^K$ .
- In this example  $K = 3$ ; blue dots are datapoints, red crosses are centres.

- We will assign each datapoint to a cluster,  $\mathbf{x}^n \rightarrow c(n)$  where  $c(n) \in \{1, \dots, K\}$  is the cluster index of datapoint number  $n$ .
- Want to find the assignments and centres that minimise the squared loss

$$\sum_{n=1}^N \left( \mathbf{x}^n - \mathbf{m}^{c(n)} \right)^2$$

- The squared loss is the sum of the squared lengths of the grey lines.

## K-means algorithm

- 1: Initialise the centres  $\mathbf{m}_i$ ,  $i = 1, \dots, K$ .
- 2: **while** not converged **do**
- 3:     For each centre  $i$ , find all the  $\mathbf{x}^n$  for which  $i$  is the nearest centre.
- 4:     Call this set of points  $\mathcal{N}_i$ . Let  $N_i$  be the number of datapoints in set  $\mathcal{N}_i$ .
- 5:     Update the centre by taking the mean of those datapoints assigned to this centre:

$$\mathbf{m}_i^{new} = \frac{1}{N_i} \sum_{n \in \mathcal{N}_i} \mathbf{x}^n$$

- 6: **end while**

---

### Initialisation

- We can get different clusterings depending on the initialisation of the centres.
- Typically we run the algorithm several times with different initialisations.
- The 'best' clustering is that which has the lowest squared loss.

# K-means in action

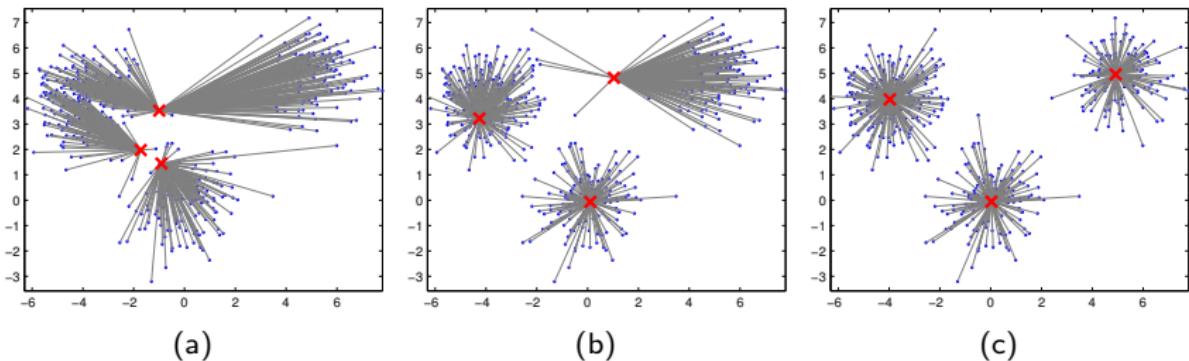
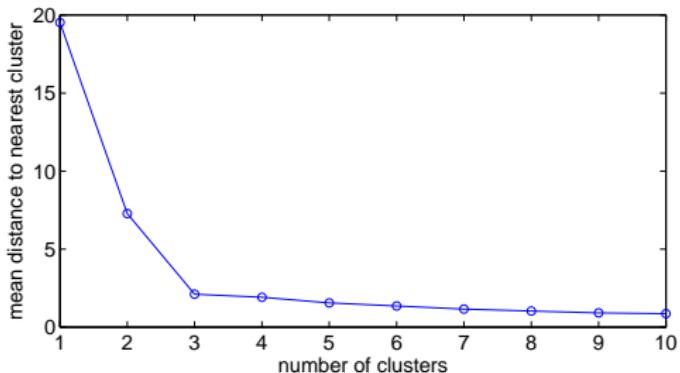


Figure : 550 datapoints clustered using K-means with 3 components. The means are given by the red crosses. (a): First iteration. (b): Second iteration. (c): Third iteration.

How we order the cluster centres is irrelevant – we might call the top right cluster number '2', but we could equally call it number '3'.

`demoKmeansMNIST.m`

## Determining the number of clusters



- As we increase the number of clusters, the mean distance to the nearest cluster centre will decrease.
- Look for a ‘knee’ (or ‘elbow’) – there is one here at  $K = 3$ .
- This is the point in which the distance stops decreasing significantly.

# K-means limitations

## Hard assignment

- A datapoint is assigned to only a single cluster.
  - It might be preferable to make a soft assignment – probably belongs to cluster 1, but could belong to cluster 3 with a certain probability.
  - Mixture models can address this.
- 

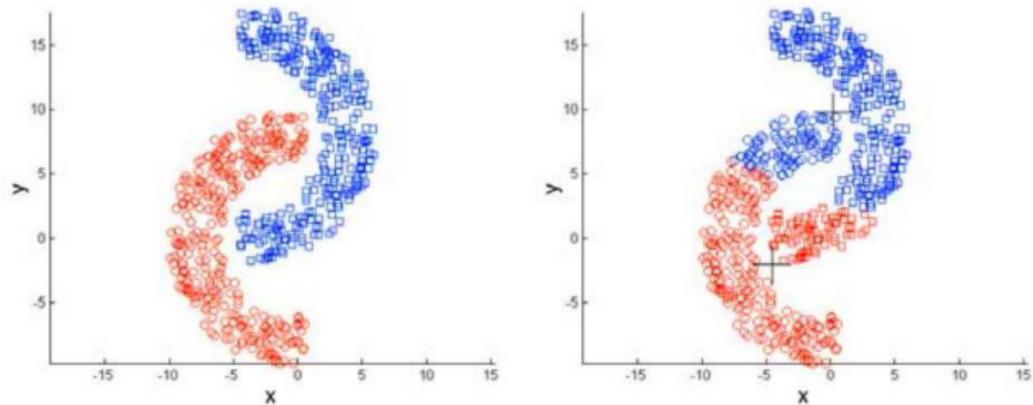
## Outlier sensitivity

- If there is an outlier (datapoint far from the rest) this can throw off the K-means algorithm.
  - The K-medians algorithm can be less sensitive to outliers.
- 

## Shape/density issues

- Works best when the clusters are roughly spherical and of equal number of points.
- Can use spectral clustering in case the data is not spherical.

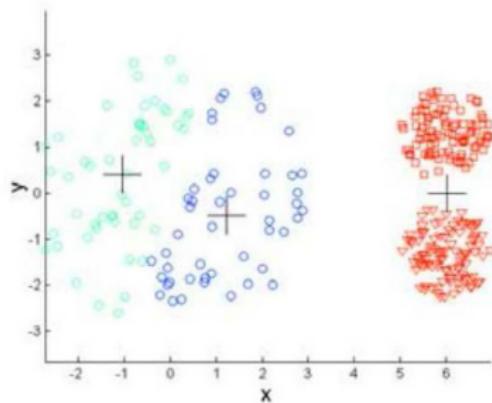
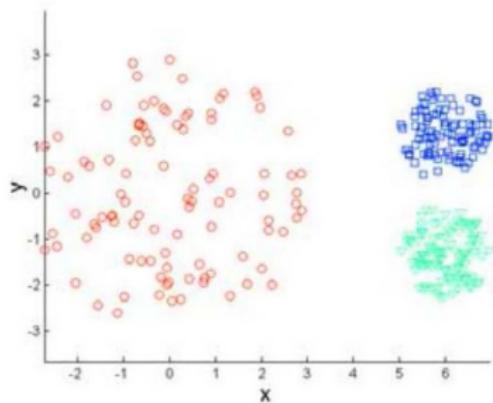
## K-means limitations: non spherical data



- Left: clustering we would like. Right: clustering K-means produces<sup>3</sup>.
- Problem is that the clusters are not spherical and they are too close.
- Can address using spectral clustering (see later).

<sup>3</sup>Images from Christof Monz (Queen Mary, Univ. of London)

## K-means limitations: non equal density



- Left: clustering we would like. Right: clustering K-means produces.
- Hard to address this with K-means.

## K-means: product purchase clustering

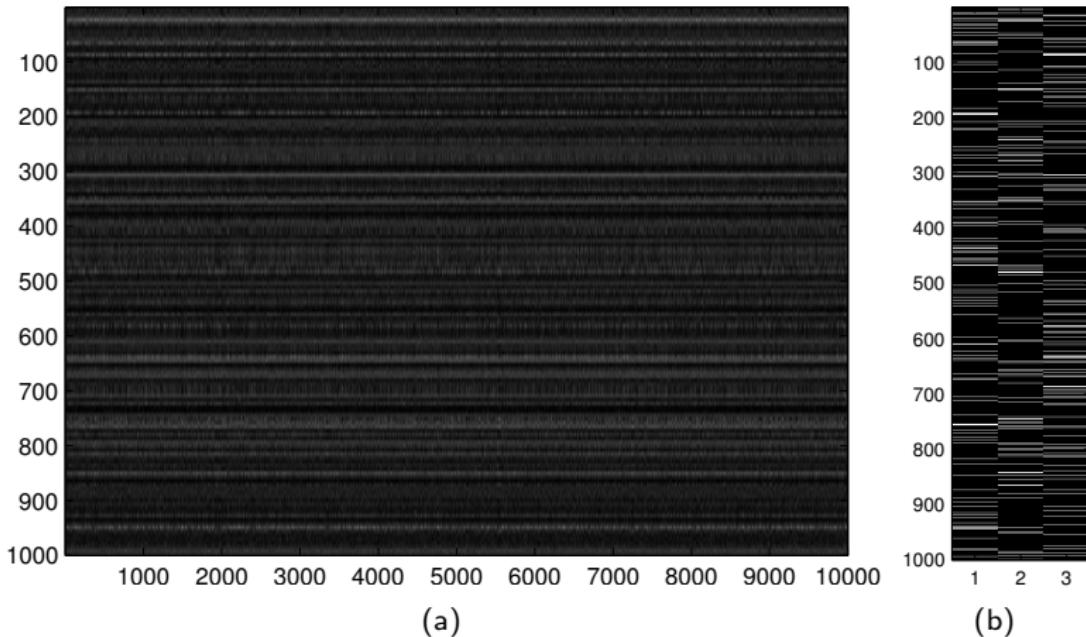


Figure : (a): 10,000 datapoints of 1000 dimensions (90% of entries are zero). (b): 3 product purchase clusters.

[demoKmeansProducts.m](#)

# Practical Issues

## Computational cost

- K-means is a fast method.
  - However, if each datapoint is very high dimensional, finding the nearest centre can be expensive.
  - There are speed-up techniques available (see fastNN material).
- 

## Representation Sensitivity

- Let's say that each data-vector contains two attributes  $x_1$  and  $x_2$ .  $x_1$  represents the temperature in centigrade, and  $x_2$  the distance.
- If we represent the distance in millimetres, then the Euclidean distance will be dominated by the difference in distance.
- If we represent the distance in kilometers, the distance will be most likely dominated by the difference in temperature.
- Rescaling (see data intro) is one way around this.

# Practical Issues

## Categorical Data

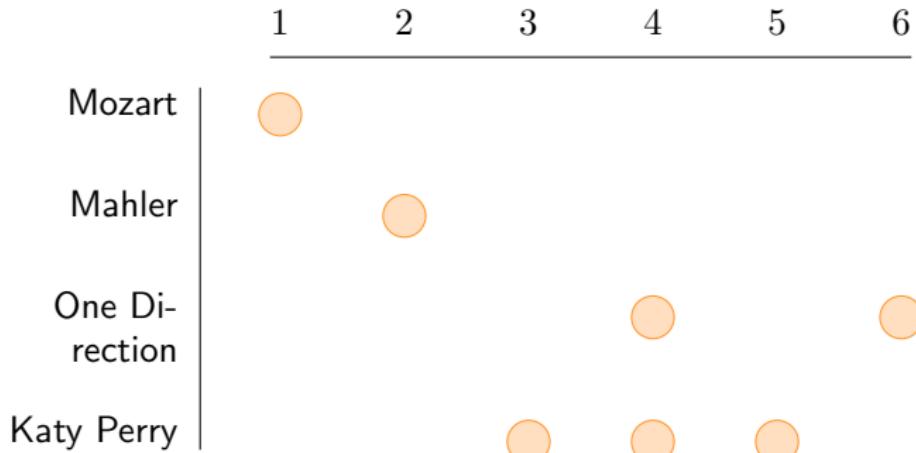
- If the data are for example questionnaire responses, 'a', 'b', 'c', how are we to numerically encode these? We could use 1,2,3, but this would bias the clusters found since 1 is closer to 2 than to 3.
- 1-of- $M$  encoding:  $a \rightarrow \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ ,  $b \rightarrow \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ ,  $c \rightarrow \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ . So, for example a data vector with two attributes might be replaced by  $\begin{pmatrix} a \\ 0.2 \end{pmatrix} \rightarrow \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0.2 \end{pmatrix}$

---

## Missing Data

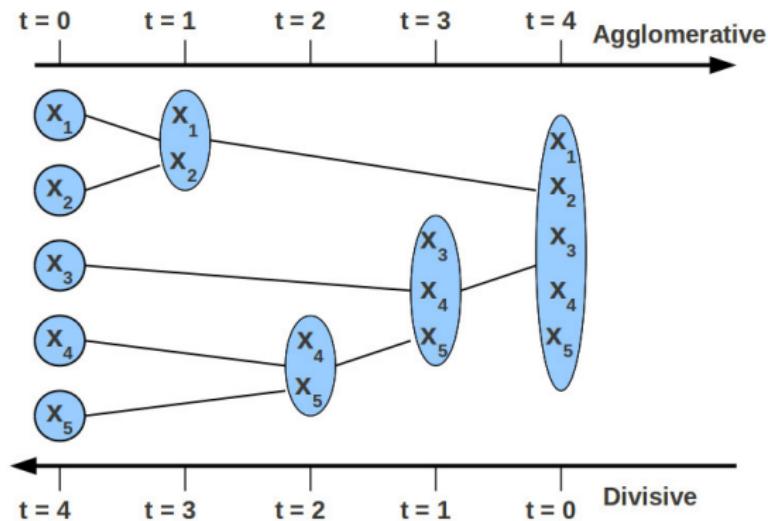
- How can we deal with missing data? Encoding missing data with a '0' will bias the clusters found.
- There is no simple and justifiable way to deal with missing data in K-means.

## Hierarchical K-means



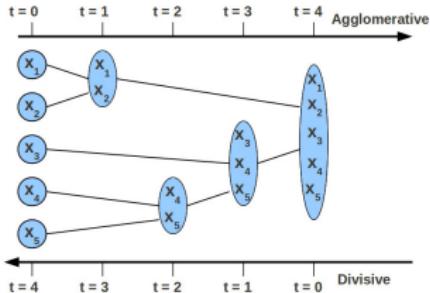
- Here there are two groups – classical music buyers and ‘pop’ music buyers.
- Classical music data can be further split into a Mozart and Mahler group.
- Pop music data splits into One Direction and Katy Perry groups.
- Can do this Top Down (clustering all data into a small number of groups, and then clustering the data in each group) or Bottom Up (recursively merging datapoints or clusters that are close together).

# Hierarchical K-means



- Hierarchical clustering forms a tree in which the children are contained within their parent cluster.
- This is a binary tree example, but could have say ternary tree, etc.

# Hierarchical K-means



## Agglomerative (bottom-up) Clustering

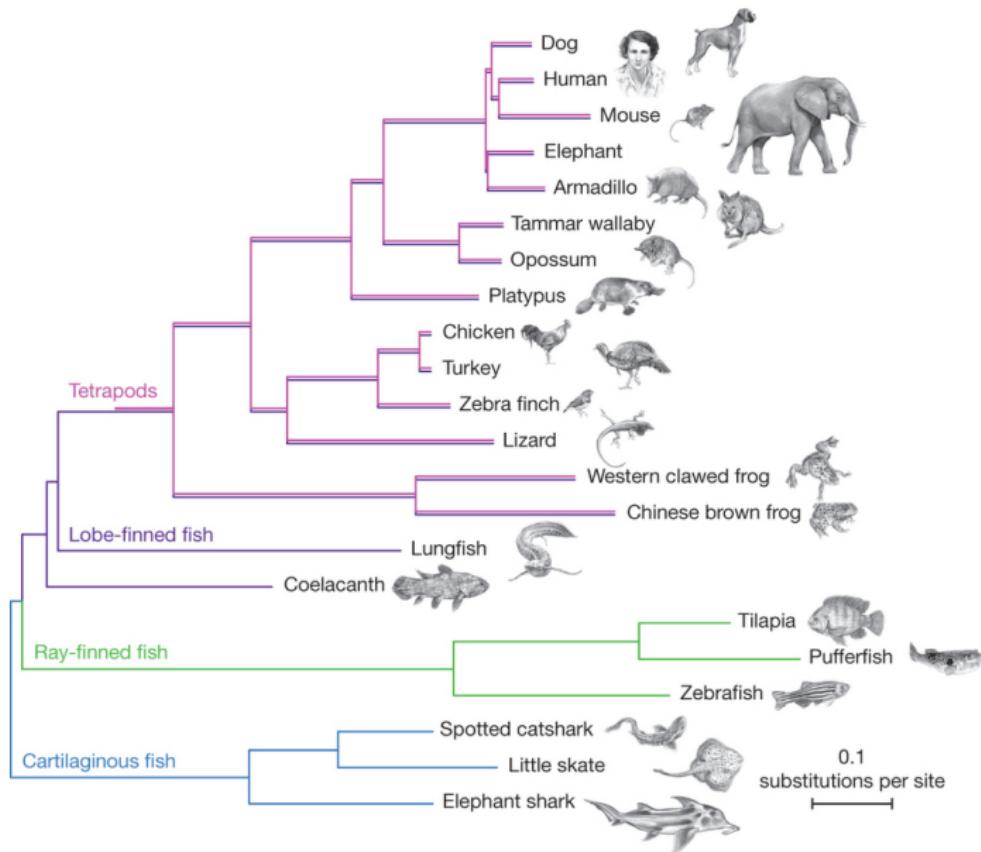
- Start with each example in its own singleton cluster
- At each time-step, greedily merge 2 most similar clusters

---

## Divisive (top-down) Clustering

- Start with all examples in the same cluster
- At each time-step, remove the ‘outsiders’ from the least cohesive cluster

# Hierarchical Clustering using gene similarity<sup>4</sup>



# Spectral Clustering

- Define a symmetric matrix of the similarity of datapoints, for example

$$A_{i,j} = e^{-\lambda \|\mathbf{x}^i - \mathbf{x}^j\|^2}$$

for some chosen  $\lambda$ .

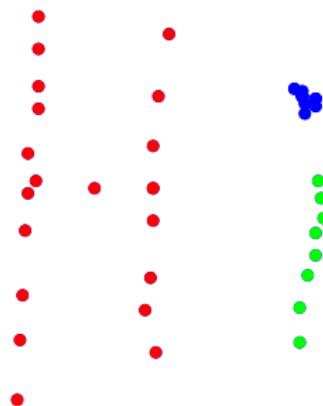
- Define Markov chain transition matrix with elements

$$p(i|j) = \frac{A_{i,j}}{\sum_i A_{i,j}}$$

- This defines a model which specifies the probability that we jump to datapoint  $i$  given that we are currently at datapoint  $j$ .
- The equilibrium distribution of  $p$  is the probability that we visit each datapoint in the long run by transitioning randomly according to  $p$ .
- Key insight is that the closely connected points will form a single high probability cluster.

# Spectral Clustering

- The equilibrium distribution is given by finding the eigenvector  $e$  of  $p$  with largest eigenvalue.
- Datapoints  $i$  with  $e_i$  larger than some threshold are placed then in a single cluster and the remaining datapoints in a separate cluster. Alternatively, compute the largest  $k$  eigenvectors and cluster these using k-means.
- I prefer to take each datapoint and find the equilibrium distribution, starting from that datapoint, and then cluster these distributions using k-means.

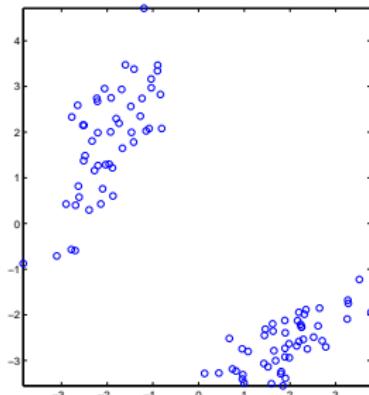


# Spectral Clustering

- Spectral Clustering can be very sensitive to the choice of similarity measure ●
- If we apply K-means to the equilibrium distributions, we still have a potentially difficult optimisation problem, requiring multiple runs to get the best solution ●
- We have to set parameters such as  $\lambda$  (in the  $A$  matrix) by hand ●
- Spectral Clustering is a very powerful method that can deal with data distributions where standard k-means does not produce a satisfactory result ●

`demoSpectralClustering.m`

# Clustering



Samples from a Gaussian mixture model  
 $1/2\mathcal{N}(\mathbf{x}|\mathbf{m}_1, \mathbf{C}_1) + 1/2\mathcal{N}(\mathbf{x}|\mathbf{m}_2, \mathbf{C}_2)$  with means  $\mathbf{m}_1, \mathbf{m}_2$  and covariances  $\mathbf{C}_1, \mathbf{C}_2$ .

- Mixture models have a natural application in clustering data, where  $h$  indexes the cluster.
- This interpretation can be gained from considering how to generate a sample datapoint  $v$  from the model.
- First we sample a cluster  $h$  from  $p(h)$ , and then draw a visible state  $v$  from  $p(v|h)$ .
- Mixture Models are important since they enable the clustering of diverse kinds of data and can readily deal with missing data and hierarchical constraints.

## Clustering as Data generation: an example

- We have four products: apples, broccoli, crisps, doughnut.
- Represent the products a customer bought on a visit by a 4-dimensional vector  $\mathbf{x}$  ( $x_i = 0$  zero represents non-purchase):

$x_1 = 1$  represents that the customer bought apples

$x_2 = 1$  represents that the customer bought broccoli

$x_3 = 1$  represents that the customer bought crisps

$x_4 = 1$  represents that the customer bought doughnuts

### Cluster 1

- These people tend to buy health food products and avoid ‘junk’ food.
- Let’s assume that they will buy any of the products independently.

### Cluster 2

- These people tend to buy ‘junk’ food products and avoid ‘healthy’ food.
- Let’s assume that they will buy any of the products independently.

## Clustering as data generation: an example

- Let's say that 25% of customers are in the healthy group and 75% in the junk food purchasing group.
- Let's try to generate a set of data. To do this we also need to specify the probabilities that each group will buy certain products:

Cluster 1:

$$p(x_1 = 1|\text{cluster 1}) = 0.5,$$

$$p(x_1 = 0|\text{cluster 1}) = 0.5$$

$$p(x_2 = 1|\text{cluster 1}) = 0.7,$$

$$p(x_2 = 0|\text{cluster 1}) = 0.3$$

$$p(x_3 = 1|\text{cluster 1}) = 0.1,$$

$$p(x_3 = 0|\text{cluster 1}) = 0.9$$

$$p(x_4 = 1|\text{cluster 1}) = 0.1,$$

$$p(x_4 = 0|\text{cluster 1}) = 0.9$$

Cluster 2:

$$p(x_1 = 1|\text{cluster 2}) = 0.1,$$

$$p(x_1 = 0|\text{cluster 2}) = 0.9$$

$$p(x_2 = 1|\text{cluster 2}) = 0.1,$$

$$p(x_2 = 0|\text{cluster 2}) = 0.9$$

$$p(x_3 = 1|\text{cluster 2}) = 0.5,$$

$$p(x_3 = 0|\text{cluster 2}) = 0.5$$

$$p(x_4 = 1|\text{cluster 2}) = 0.4,$$

$$p(x_4 = 0|\text{cluster 2}) = 0.6$$

## Clustering as data generation: an example

- For each customer we first decide if they are going to be in the healthy group or the junk food group.
- We do this by drawing a cluster index  $h \in \{1, 2\}$  from the distribution  $p(h = 1) = 0.25, p(h = 2) = 0.75$ .
- Given the cluster we now draw a set of product purchases, independently for each product according to the given probabilities.
- For example, we might have drawn  $h = 1$ , and then drawn a purchase vector

$$\mathbf{x} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

- We repeat this process for 10 people and have the data matrix:

$$\mathbf{X} = \begin{pmatrix} 0, 1, 0, 1, 1, 0, 0, 0, 0, 1 \\ 0, 0, 1, 1, 1, 0, 0, 0, 0, 0 \\ 0, 0, 1, 0, 1, 0, 0, 1, 1, 1 \\ 1, 0, 0, 0, 1, 1, 1, 0, 1, 0 \end{pmatrix}$$

## Clustering as data generation: an example

- Given the data matrix:

$$\mathbf{X} = \begin{pmatrix} 0, 1, 0, 1, 1, 0, 0, 0, 0, 1 \\ 0, 0, 1, 1, 1, 0, 0, 0, 0, 0 \\ 0, 0, 1, 0, 1, 0, 0, 1, 1, 1 \\ 1, 0, 0, 0, 1, 1, 1, 0, 1, 0 \end{pmatrix}$$

- Can we figure out which are the healthy customers and the junk food customers?

---

### Bayes Rule

$$p(h=1|\mathbf{x}) = \frac{p(\mathbf{x}|h=1)p(h=1)}{p(\mathbf{x}|h=1)p(h=1) + p(\mathbf{x}|h=2)p(h=2)}$$

$$p(\mathbf{x}|h=1) = \prod_{i=1}^4 p(x_i|h=1)$$

## Clustering as data generation: an example

Using Bayes rule to calculate cluster assignment

$$p(h = 1 | (0, 0, 0, 1)) = \frac{0.5 \times 0.3 \times 0.9 \times 0.1 \times 0.25}{0.5 \times 0.3 \times 0.9 \times 0.1 \times 0.25 + 0.9 \times 0.9 \times 0.5 \times 0.4 \times 0.75} \\ = 0.027$$

- Hence  $p(h = 2 | (0, 0, 0, 1)) = 1 - 0.027 = 0.973$ .
- This person is very likely to be from the 'junk food' group.
- We do this for each person in the data to get a 'soft' assignment to the clusters.

---

Exercise:

What is the probability that  $\mathbf{x} = (1, 0, 1, 0)$  belongs to the healthy group?

# Clustering as data generation: an example

## Joint distribution

$$p(\mathbf{x}, h) = p(\mathbf{x}|h)p(h) = p(h) \prod_i p(x_i|h)$$

---

## Conditional

Using the joint distribution we can compute the conditional

$$p(h|\mathbf{x}) \propto p(\mathbf{x}, h)$$

---

## Marginal

$$p(\mathbf{x}) = \sum_h p(\mathbf{x}, h) = \sum_h p(h) \prod_i p(x_i|h)$$

---

## Likelihood

Since each customer is generated independently in the same way, the distribution over all customers is

$$p(\mathbf{x}^1, \dots, \mathbf{x}^{10}) = \prod_{n=1}^{10} p(\mathbf{x}^n)$$

## Clustering as data generation: an example

- In this example, we assumed that we know the probabilities  $p(h)$  and  $p(x_i|h)$ .
- However, in general we are interested in the situation that we have an observed dataset  $\mathbf{X}$  and we want to learn these parameters.
- We can do this by setting the parameters to those for which the data we observe is the most likely to have been generated by the model.
- This is called the maximum likelihood approach.

---

### Training philosophy

- Define the model  $p(\mathbf{x}, h|\theta)$ .
- For the given dataset  $\mathbf{X}$ , search over all parameters of the model  $\theta$  such that

$$\theta_{opt} = \arg \max_{\theta} p(\mathbf{X}|\theta) = \arg \max_{\theta} \prod_n p(\mathbf{x}^n|\theta)$$

- Given the optimal parameters, form the soft-clustering

$$p(h^n|\mathbf{x}^n)$$

## Mixture Models

- Mixture models are a powerful technique for unsupervised learning and clustering in particular.
  - Can deal with missing data, categorical data (without the biases of K-means)
- 

A mixture model is one in which a set of component models is combined to produce a richer model:

$$p(v) = \sum_{h=1}^H p(v|h)p(h)$$

The variable  $v$  is ‘visible’ or ‘observable’ and  $h = 1, \dots, H$  indexes each component model  $p(v|h)$ , along with its weight  $p(h)$ .

---

## Clustering

Mixture models have a natural interpretation in terms of clustering with each state of  $h$  corresponding to a cluster model  $p(v|h)$ .

## Learning the Parameters of a Mixture Model

- For a dataset  $\mathbf{v}^1, \dots, \mathbf{v}^N$ , the model has a probability of generating this data

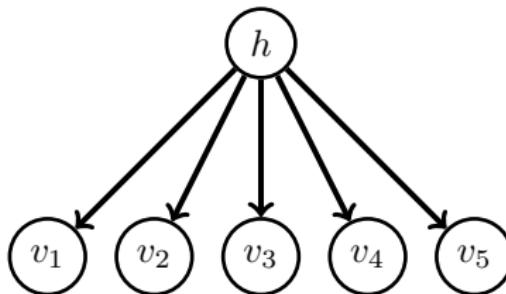
$$L(\theta) \equiv \prod_{n=1}^N p(\mathbf{v}^n | \theta)$$

- The EM (Expectation Maximisation) algorithm is an iterative procedure for maximising  $L(\theta)$  with respect to the model parameters  $\theta$ .
- Typically the EM algorithm (and other parameter learning algorithms) are highly sensitive to the initial guess for the parameters. In practice we therefore run the algorithm several times, using different random initial parameters.
- The best parameters correspond to the highest likelihood value.
- Once trained, we compute the posterior cluster assignment for each datapoint

$$p(h|\mathbf{v}^n) \propto p(\mathbf{v}^n|h)p(h)$$

We can make a hard assignment by finding which  $h$  has the highest posterior probability.

## Mixture of Independent Bernoulli



A model that can be used to cluster a set of binary vectors,  $\mathbf{v}^n = (v_1^n, \dots, v_D^n)^T$ ,  $v_i \in \{0, 1\}$ ,  $n = 1, \dots, N$  is

$$p(\mathbf{v}) = \sum_{h=1}^H p(h) \prod_{i=1}^D p(v_i|h)$$

where each term  $p(v_i|h)$  is a Bernoulli distribution.

---

### Parameters

We need to learn  $p(v_i = 1|h = h)$  and  $p(h = h)$ .

## An example: clustering binary digits

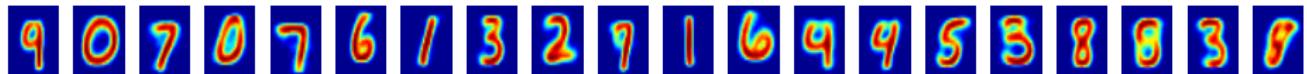
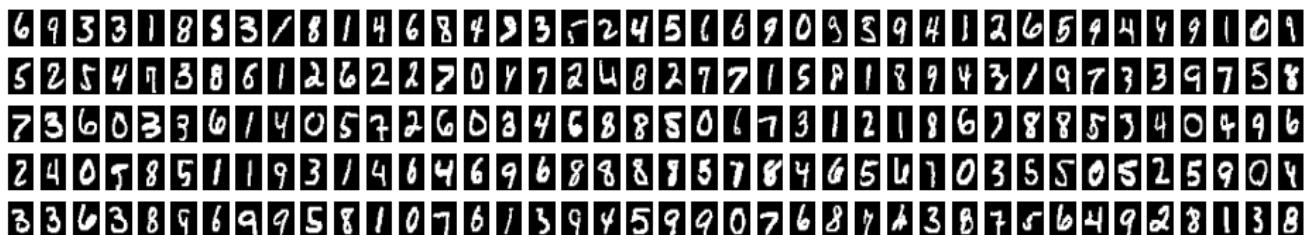


Figure : Top: a selection of 200 of the 5000 handwritten digits in the training set.  
Bottom: the trained cluster outputs  $p(v_i = 1|h)$  for  $h = 1, \dots, 20$  mixtures. See  
`demoMixBernoulliDigits.m`.

## An example: clustering binary digits with missing data

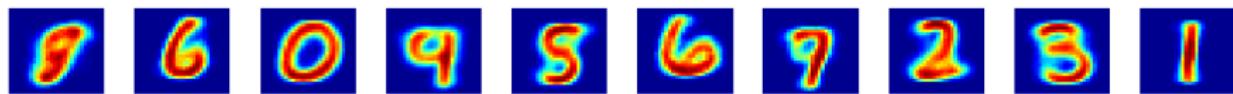
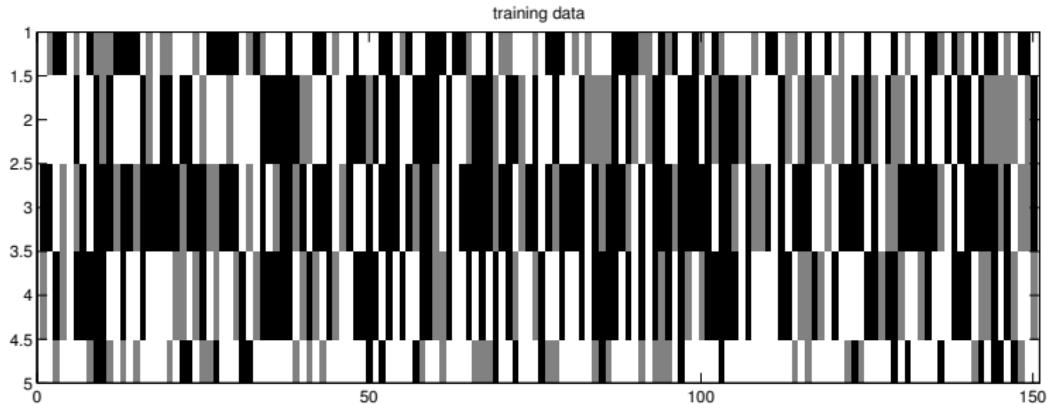


Figure : Top: a selection of the 5000 handwritten digits in the training set with missing data. Bottom: the trained cluster outputs  $p(v_i = 1|h)$  for  $h = 1, \dots, 10$  mixtures. See [demoMixBernoulliDigitsMissing.m](#).

# Clustering Questionnaires using Bernoulli Mixture

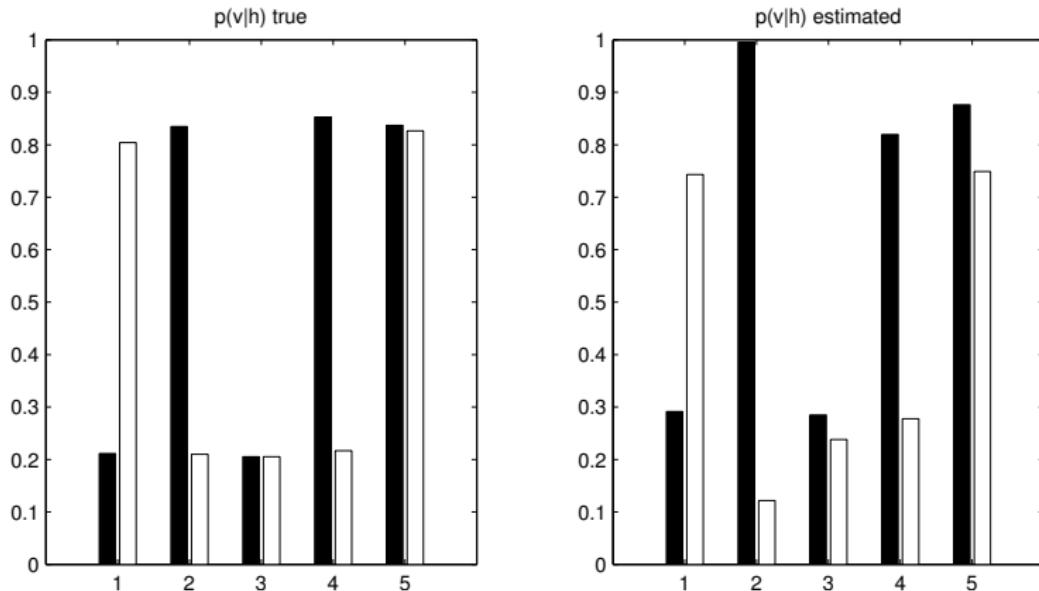
If an attribute  $i$  is missing for this model one simply removes the corresponding factor  $p(v_i^n|h)$  from the algorithm.



**Figure :** Data from questionnaire responses. 150 people were each asked 5 questions, with 'yes' (white) and 'no' (gray) answers. Black denotes the absence of a response (missing data). This training data was generated by two component Binomial mixture. Missing data was simulated by randomly removing values from the dataset. See

`demoMixBernoulli.m`

# Clustering Questionnaires using Bernoulli Mixture



**Figure :** The two components found. On the left is the true data generating probability  $p(v_i|h)$ ,  $i = 1, \dots, 5$  for each of the 5 questions and each of the two clusters (black and white). On the right are the corresponding probabilities learned by maximum likelihood from the dataset alone.

# ABC Questionnaire

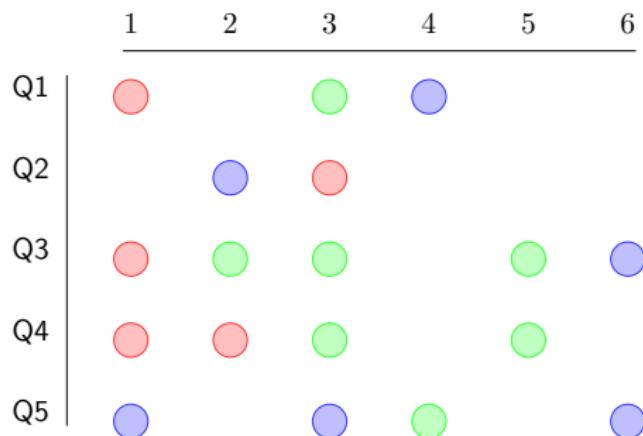
Q1: Is Tesco's expensive? (**yes,no,somewhat**)

Q2: Is the ambience of the supermarket important? (**yes,no,somewhat**)

Q3: Do you like music to be played in the store? (**yes,no,somewhat**)

Q4: Do you use the cafe in the store? (**often,never,sometimes**)

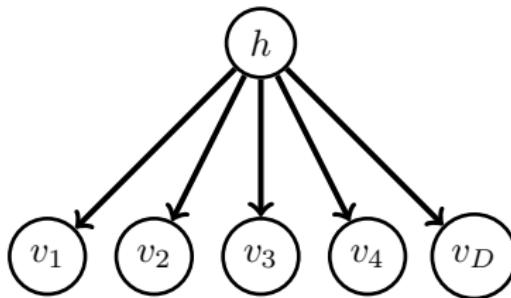
Q5: Do you get annoyed by frequent product reshelfing? (**yes,no,a bit**)



Questionnaire data is often categorical and has missing entries.

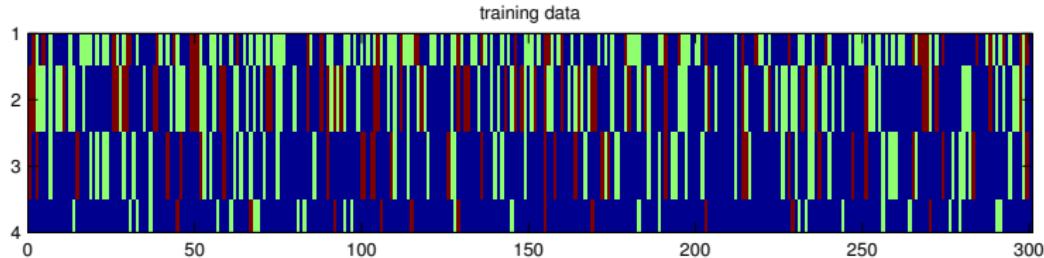
## ABC questionnaire

- We have a questionnaire with  $D$  questions. Each response is 'a', 'b' or 'c' (or missing).
- Want to cluster the responders into  $K$  clusters.



- This time  $p(v_i = c|h)$  is a categorical distribution. For each attribute  $i$  of the datavector  $v$  we have the probability that it belongs to category  $c$ . Each cluster  $h$  has a different set of such distributions.
- We can run the EM algorithm again on this model.

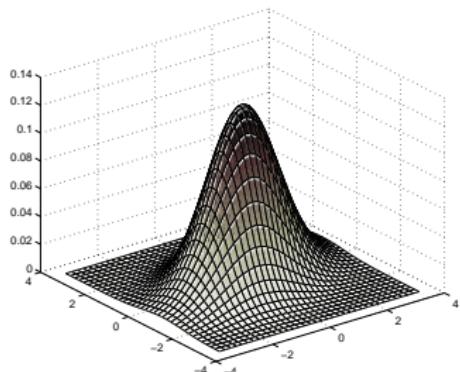
# ABC questionnaire



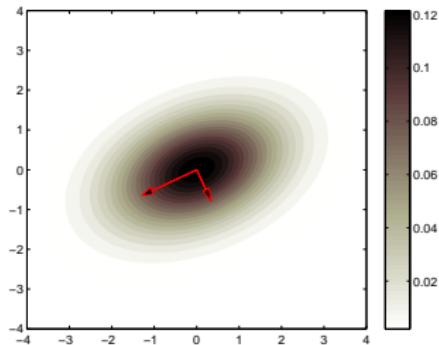
- There are 4 questions, each with possible red, blue, green responses. 300 people returned fully completed questionnaires.
- Let's assume there are two clusters,  $h^n \in \{1, 2\}$ .
- We look at  $p(h^n | \mathbf{v}^n)$  and find the most likely state of  $h^n$ . This is the cluster label we give to datapoint  $n$ .
- Running the demo below, the K-means (1-of-M) and mixture model cluster labels can differ significantly in which datapoints are assigned together into a cluster.
- Straightforward to generalise to any number of states for the categorical variables (or have different numbers of categories for different variables).

`demoMixCategorical.m`

# The multivariate Gaussian distribution



(a)



(b)

Figure : (a): Bivariate Gaussian. (b): Probability density contours.

$$p(\mathbf{x}) = \frac{1}{\sqrt{\det(2\pi\mathbf{S})}} \exp \left\{ -\frac{1}{2} (\mathbf{x} - \mathbf{m})^\top \mathbf{S}^{-1} (\mathbf{x} - \mathbf{m}) \right\}$$

where  $\mathbf{m}$  is the mean and  $\mathbf{S}$  is the covariance matrix.

# The Gaussian Mixture Model

- A mixture of Gaussians is

$$p(\mathbf{x}) = \sum_{i=1}^H p(\mathbf{x}|\mathbf{m}_i, \mathbf{S}_i)p(i)$$

where  $p(i)$  is the mixture weight for component  $i$ .

- This means that we can describe the data density by composing ‘blobs’ of localised Gaussian densities.
- The GMM is very widely used, typically with the simplifying constraint that the covariance matrices are diagonal.

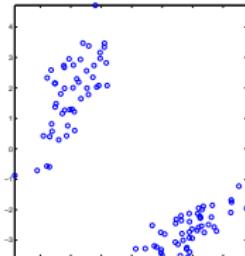


Figure : Two dimensional data which displays clusters. In this case a Gaussian mixture model  $1/2\mathcal{N}(\mathbf{x}|\mathbf{m}_1, \mathbf{C}_1) + 1/2\mathcal{N}(\mathbf{x}|\mathbf{m}_2, \mathbf{C}_2)$  would fit the data well for suitable means  $\mathbf{m}_1, \mathbf{m}_2$  and covariances  $\mathbf{C}_1, \mathbf{C}_2$ .

# Maximum Likelihood

For a set of data  $\mathcal{X} = \{\mathbf{x}^1, \dots, \mathbf{x}^N\}$  the log likelihood is

$$\sum_{n=1}^N \log \sum_{i=1}^H p(i) \frac{1}{\sqrt{\det(2\pi\mathbf{S}_i)}} \exp \left\{ -\frac{1}{2} (\mathbf{x}^n - \mathbf{m}_i)^T \mathbf{S}_i^{-1} (\mathbf{x}^n - \mathbf{m}_i) \right\}$$

---

## Parameter Constraints

- The  $\mathbf{S}_i$  must be symmetric positive definite matrices, in addition to  $0 \leq p(i) \leq 1$ ,  $\sum_i p(i) = 1$ .
- The EM approach which in this case is particularly convenient since it automatically provides parameter updates that ensure these constraints.
- Constraints on the covariance matrices (such as being diagonal) are straightforward to incorporate.

## Infinite troubles

Consider placing a component  $p(\mathbf{x}|\mathbf{m}_i, \mathbf{S}_i)$  with mean  $\mathbf{m}_i$  set to one of the datapoints  $\mathbf{m}_i = \mathbf{x}^n$ . The contribution from that Gaussian will be

$$p(\mathbf{x}^n | \mathbf{m}_i, \mathbf{S}_i) = \frac{1}{\sqrt{\det(2\pi\mathbf{S}_i)}} e^{-\frac{1}{2}(\mathbf{x}^n - \mathbf{x}^n)^T \mathbf{S}_i^{-1} (\mathbf{x}^n - \mathbf{x}^n)} = \frac{1}{\sqrt{\det(2\pi\mathbf{S}_i)}}$$

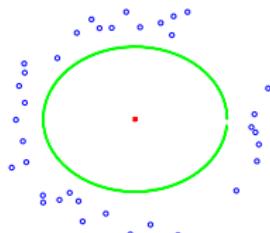
As the covariance goes to zero, this probability density becomes infinite. This means that one can obtain a Maximum Likelihood solution by placing zero-width Gaussians on a selection of the datapoints, resulting in an infinite likelihood.

---

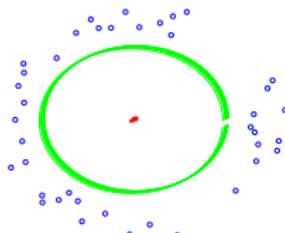
### Remedy

Include an additional constraint on the width of the Gaussians, ensuring that they cannot become too small.

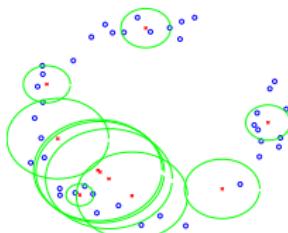
# Symmetry Breaking



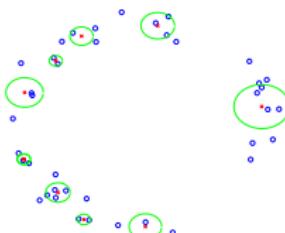
(a) 1 iteration



(b) 50 iterations



(c) 125 iterations



(d) 150 iterations

**Figure :** Training a mixture of 10 isotropic Gaussians  
**(a):** If we start with large variances for the Gaussians, even after one iteration, the Gaussians are centred close to the mean of the data.  
**(b):** The Gaussians begin to separate  
**(c):** One by one, the Gaussians move towards appropriate parts of the data  
**(d):** The final converged solution. The Gaussians are constrained to have variances greater than a set amount.

**demoGMMem.m**

## GMM and K-Means

Consider a mixture of  $K$  Gaussians in which each covariance is constrained to be equal to  $\sigma^2 \mathbf{I}$ ,

$$p(\mathbf{x}) = \sum_{i=1}^K p_i \mathcal{N}(\mathbf{x} | \mathbf{m}_i, \sigma^2 \mathbf{I})$$

- As  $\sigma^2 \rightarrow 0$ , one can show that the parameter updates in the EM algorithm become equivalent to the K-means updates.
- K-means can therefore be seen as a special case of the GMM.
- For non-zero  $\sigma^2$  the GMM gives a 'softer' clustering than K-means.

## Extensions

- We can use any distribution  $p(x_i|h)$  that we wish.
- Could e.g. use a Gaussian for say  $x_1$  and a categorical distribution for  $x_2$ .
- These models are straightforward to train using the EM algorithm.
- The Gaussian Mixture Model is not always very robust to outliers.
- Using a student  $t$  distribution is a common way to make the clustering more robust to outliers.

---

### Finding the number of mixture components

- Most common approach is to use some hold-out data (not part of the training set).
- After training the parameters of the model on the training data, we compute the likelihood of the hold-out data using these parameters.
- That model which has the highest hold-out likelihood is deemed the best model (number of mixture components).
- There are other approaches involving Bayesian statistics (very complex) and simpler techniques (such as the Bayesian Information Criterion).

# Summary

Clustering is a natural way to simplify and give insight into data.

---

## K-means

- Very popular approach and easy to implement ●
- Easy to perform hierarchical clustering ●
- Sensitive to initialisation of cluster centres ●
- Not clear how to cluster data which doesn't have a simple interpretation in terms of a vector ●

## Mixture Models

- Natural way to perform clustering in a probabilistic setting ●
- Significantly generalises K-means ●
- Some models are very straightforward to implement – can be even faster than K-means ●
- Can deal with missing data and data that doesn't have a natural vector representation ●
- Also sensitive to initialisation ●

# Visualisation

David Barber

# What is Data Visualisation?

- Data is often very high dimensional meaning that we can't directly 'see' the data.
- It's difficult to get intuitions about the data since we can't 'see it'.
- In Data Visualisation we try to find a low dimensional representation (2 or 3 dimensions) so that we 'see something'.
- There is no 'correct' or 'perfect' visualisation. Every low dimensional representation will lose some information contained in the original high dimensional data.
- Such visualisations can be useful to see whether there might be clusters of datapoints, or which datapoints are in some sense 'similar' to another.
- Historically, methods such as PCA or Sammon 'mappings' were popular, but they are now less preferred.
- We tend to heuristically prefer representations that better preserve neighbourhood structure.
- Some visualisation methods (autoencoders) can be good but they are very expensive to train.

## Setup

Each data vector  $\mathbf{x}_n$  is in a high dimensional space. Given the set of datapoints

$$\mathcal{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$$

we want to find a corresponding low dimensional (2 or 3) vector representation  $\mathbf{y}_n$  for each  $\mathbf{x}_n$  to give

$$\mathcal{Y} = \{\mathbf{y}_1, \dots, \mathbf{y}_N\}$$

- We would like  $\mathcal{Y}$  to preserve both the local and global structure in  $\mathcal{X}$ .
- Unfortunately, many ‘classical’ visualisation methods (Sammon mapping, Isomap, Locally Linear Embedding) don’t work that well on real-world data sets.
- We will focus on Stochastic Neighbour Embedding (SNE) and its ‘robust’ variant t-SNE which is one of the most popular current approaches.

# Stochastic Neighbour Embedding (SNE)

We define an  $N \times N$  Markov transition matrix:

$$p_{j|i} = \frac{\exp\left(-(\mathbf{x}_i - \mathbf{x}_j)^2 / (2\sigma_i^2)\right)}{\sum_{j \neq i} \exp\left(-(\mathbf{x}_i - \mathbf{x}_j)^2 / (2\sigma_i^2)\right)}, \quad p_{i|i} = 0$$

Similarly we define

$$q_{j|i} = \frac{\exp\left(-(\mathbf{y}_i - \mathbf{y}_j)^2\right)}{\sum_{j \neq i} \exp\left(-(\mathbf{y}_i - \mathbf{y}_j)^2\right)}, \quad q_{i|i} = 0$$

- The transition  $p$  describes the neighbourhood structure – how easily can one jump to other points from a given point.
- If we want to preserve this structure, we need to find  $\mathcal{Y}$  such that  $q$  is approximately the same as  $p$ .
- Note that the  $\mathbf{y}_n$  scale is arbitrary, so we have ‘fixed’ the variance to  $1/\sqrt{2}$  in the  $y$  space.

## SNE

- SNE minimises the KL divergence between each conditional distribution  $p_i \equiv p_{\cdot|i}$ ):

$$E(\mathcal{Y}) = \sum_i \text{KL}(p_i|q_i) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

- The optimisation is performed using gradient descent with parameters  $\mathcal{Y}$ .
- One can show (exercise) that

$$\frac{\partial E}{\partial \mathbf{y}_i} = 2 \sum_j (p_{j|i} - q_{j|i} + p_{i|j} - q_{i|j}) (\mathbf{y}_i - \mathbf{y}_j)$$

- Criticisms about this approach are that the KL divergence is not symmetric.
- For example there is a large cost for using widely separated  $y$  points (small  $q_{j|i}$ ) to represent nearby  $x$  points (large  $p_{j|i}$ ).
- SNE therefore focuses on making sure that the local structure is correct, but loses fidelity in retaining the global structure.
- Another problem is that the Gaussian form of  $p_{j|i}$  means that points which are far away will have negligible impact on the objective. We therefore need to make such points have an influence on the objective.

## t-SNE

There are many potential ways to ‘robustify’ the SNE procedure. The t-SNE approach, which seems to work reasonably well, is:

- t-SNE uses a ‘symmetric’ loss

$$E = \text{KL}(p|q) = \sum_i \sum_j p_{i,j} \log \frac{p_{i,j}}{q_{i,j}}$$

where the ‘joint’ distribution is defined

$$p_{i,j} = \frac{p_{j|i} + p_{i|j}}{2N}, \quad p_{i,i} = 0$$

- Note that this definition ensures that  $p_i = \sum_j p_{i,j} > 1/(2N)$  which encourages each datapoint to have a significant effect on the cost function.
- The gradient is (exercise)

$$\frac{\partial E}{\partial \mathbf{y}_i} = 4 \sum_j (p_{i,j} - q_{i,j}) (\mathbf{y}_i - \mathbf{y}_j)$$

- This works quite well, but there is an additional step that is also useful.

## t-SNE

- If we use a student t-distribution (rather than a Gaussian) this has heavier tails and can therefore assign non-negligible mass to  $y$  points that are quite far apart. Defining a student t-distribution with a single degree of freedom,

$$q_{i,j} = \frac{\left(1 + (\mathbf{y}_i - \mathbf{y}_j)^2\right)^{-1}}{\sum_{i \neq j} \left(1 + (\mathbf{y}_i - \mathbf{y}_j)^2\right)^{-1}}, \quad q_{i,i} = 0$$

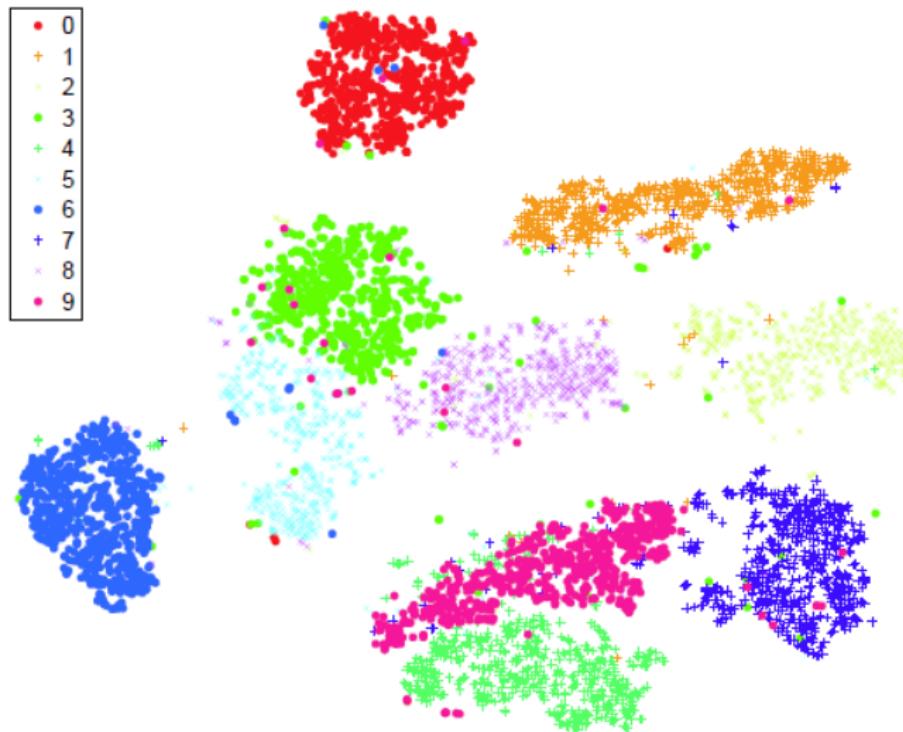
Note that the sum in the denominator is over all pairs of distinct points.

- When  $(\mathbf{y}_i - \mathbf{y}_j)^2$  is large, the '1' term becomes irrelevant and the  $q$  distribution will be essentially invariant with respect to the overall length scale.
- Hence, for all but the finest length scales, pairs of points that are very far apart will have a similar contribution to points that are reasonably far apart.
- The gradient is (exercise)

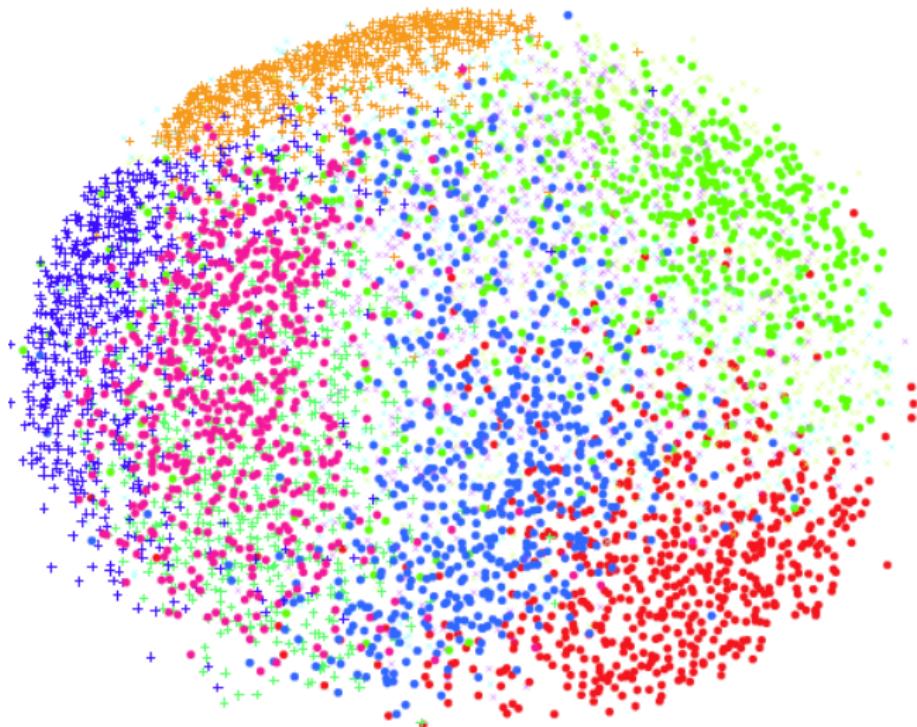
$$\frac{\partial E}{\partial \mathbf{y}_i} = 4 \sum_j \frac{(p_{i,j} - q_{i,j})}{1 + (\mathbf{y}_i - \mathbf{y}_j)^2} (\mathbf{y}_i - \mathbf{y}_j)$$

- Note that it is unclear why the t-SNE authors do not define a student t-distribution on the original  $x$  datapoints as well.

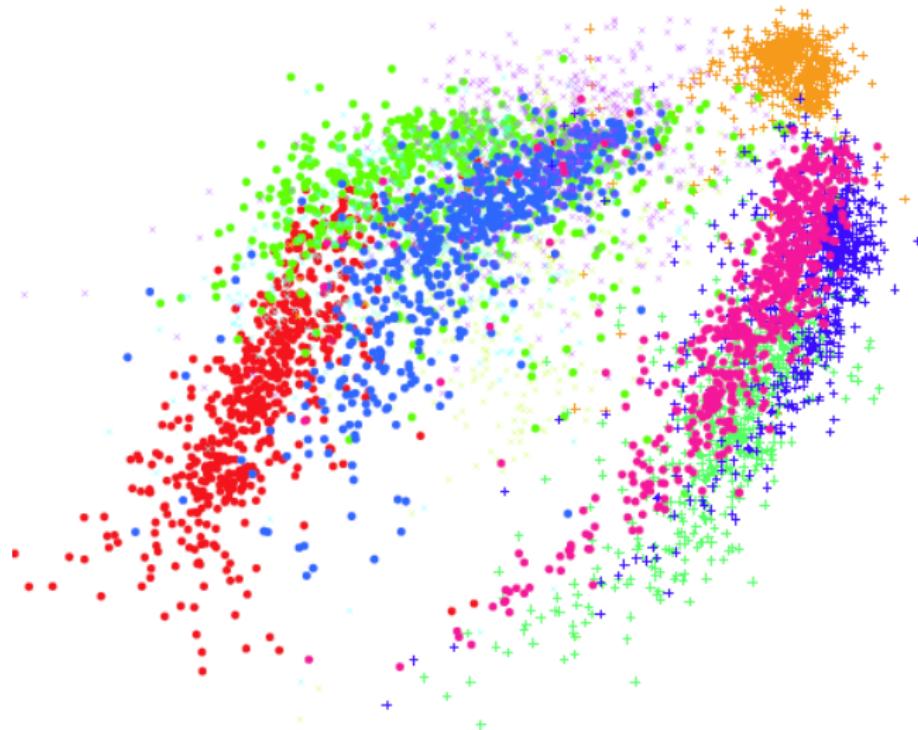
# MNIST 2D visualisation: t-SNE



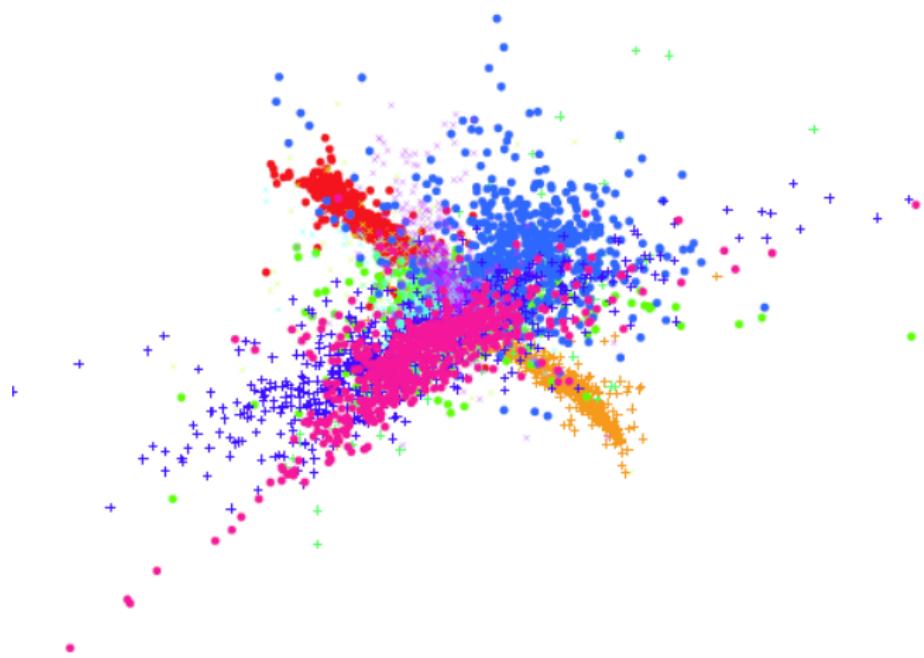
# MNIST 2D visualisation: Sammon Mapping



# MNIST 2D visualisation: Isomap



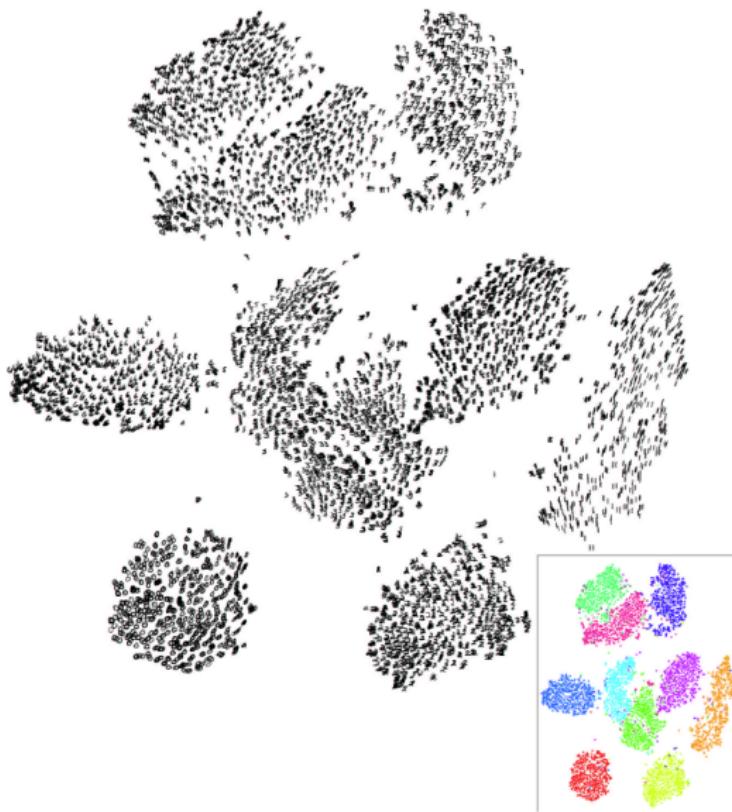
# MNIST 2D visualisation: LLE



## Large Datasets

- Like most visualisation methods, t-SNE has an  $O(N^2)$  cost just to calculate the objective function since the distance between all pairs of points is calculated.
- For large datasets, this means that it is very expensive to train t-SNE and related methods since each iteration of gradient descent requires an  $O(N^2)$  calculation.
- A cheaper alternative is to first define a desired number of neighbours and calculate a graph of which are the nearest neighbours of each node ( $x$  datapoint).
- We then select (randomly) a small set of ‘landmark’ datapoints in  $x$ , indexed by  $i'$ . We can then calculate a new transition matrix for these datapoints as follows. Starting from  $i'$  we randomly sample  $j$  according to  $p(j|i = i')$ . We repeat sampling from this Markov chain until we land on another landmark  $j' \neq i'$ . We then repeat this procedure many times for landmark  $i'$  and then normalise to obtain the transition  $p(j'|i)$ . We then repeat this for each landmark  $i'$ .
- We then use  $p(j'|i')$  in place of the original full  $p(j|i)$  transition to find a visualisation for the chosen landmark points.
- Whilst it is expensive to calculate  $p(j'|i')$ , this only needs to be done once.

## MNIST 2D visualisation: t-SNE



Visualisation of 6000 landmark points (using the random walk transition).

# Using Autoencoders to visualise MNIST

Autoencoders (with a 2D bottleneck) can also be used to visualise data).

**Fig. 3.** (A) The two-dimensional codes for 500 digits of each class produced by taking the first two principal components of all 60,000 training images. (B) The two-dimensional codes found by a 784-1000-500-250-2 autoencoder. For an alternative visualization, see (8).

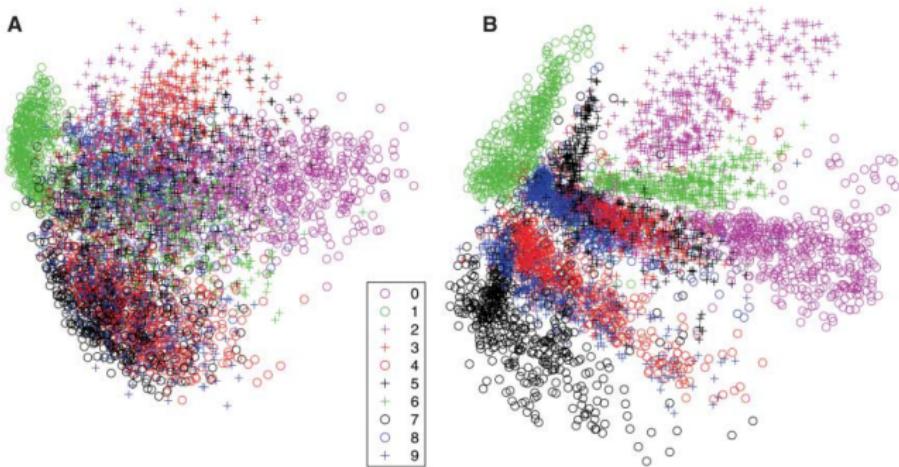


Figure from Hinton and Salakhutdinov Science 2006. The results can also be very good but training is quite slow and not practical for a 'quick' visualisation.

# Fast Nearest Neighbour Computation<sup>1</sup>

David Barber

University College London

---

<sup>1</sup> These slides accompany the book *Bayesian Reasoning and Machine Learning*. The book and demos can be downloaded from [www.cs.ucl.ac.uk/staff/D.Barber/bml](http://www.cs.ucl.ac.uk/staff/D.Barber/bml). Feedback and corrections are also available on the site. Feel free to adapt these slides for your own purposes, but please include a link to the above website.

## Finding your nearest neighbour quickly

- Consider that we have a set of datapoints  $\mathbf{x}^1, \dots, \mathbf{x}^N$  and a new query vector  $\mathbf{q}$ .
- Our task is to find the nearest neighbour to the query  $n^* = \operatorname{argmin}_n d(\mathbf{q}, \mathbf{x}^n)$ ,  
 $n = 1, \dots, N$ .
- For the Euclidean distance

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{(\mathbf{x} - \mathbf{y})^2} = \sqrt{\sum_{i=1}^D (x_i - y_i)^2}$$

and  $D$  dimensional vectors, it takes  $O(D)$  operations to compute this distance.

- For a set of  $N$  vectors, computing the nearest neighbour to  $\mathbf{q}$  would take then  $O(DN)$  operations. For large datasets this can be prohibitively expensive. Is there a way to avoid calculating all the expensive distances?
- Useful for nearest neighbour classification and k-means clustering.

# General Approaches

- Orchard** We'll consider first the situation in which the calculation of a distance  $d(\mathbf{x}, \vec{y})$  is expensive. For example, the distance between very high dimensional vectors. We'll make use of the Triangle Inequality to reduce the number of such calculations at the cost of storing a large interpoint distance matrix.
- AESA** This can be used to eliminate datapoints so that  $d(\vec{q}, \mathbf{x})$  need not be computed for datapoint  $\vec{x}$ . However, this will still require a cost proportional to the number of datapoints. Both AESA and Orchard rely on the distance being a metric.
- KD Tree** An alternative approach is to try to partition the space in such a way that we do not need to use many distance calculations. In this case the number of calculations may be less than  $N$ . The distance does not need to be a metric.

Note that in the worst case, all methods will have complexity equal or worse than simply calculating all distances and finding the smallest.

# Metric Distances

## Using the triangle inequality to speed up search

- For the squared Euclidean distance we have

$$(\mathbf{x} - \mathbf{y})^2 = (\mathbf{x} - \mathbf{z} + \mathbf{z} - \mathbf{y})^2 = (\mathbf{x} - \mathbf{z})^2 + (\mathbf{z} - \mathbf{y})^2 + 2(\mathbf{x} - \mathbf{z})^\top (\mathbf{z} - \mathbf{y})$$

- Since the scalar product between two vectors  $\mathbf{a}$  and  $\mathbf{b}$  relates the lengths of the vectors  $|\mathbf{a}|$ ,  $|\mathbf{b}|$  and the angle  $\theta$  between them by  $\mathbf{a}^\top \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos(\theta)$  we have (using  $|\mathbf{x}| \equiv \sqrt{\mathbf{x}^\top \mathbf{x}}$ )

$$|\mathbf{x} - \mathbf{y}|^2 = |\mathbf{x} - \mathbf{z}|^2 + |\mathbf{z} - \mathbf{y}|^2 + 2 \cos(\theta) |\mathbf{x} - \mathbf{z}| |\mathbf{z} - \mathbf{y}|$$

- Using  $\cos(\theta) \leq 1$  we obtain the triangle inequality

$$|\mathbf{x} - \mathbf{y}| \leq |\mathbf{x} - \mathbf{z}| + |\mathbf{z} - \mathbf{y}|$$

- Geometrically this simply says that for a triangle formed by the points  $\mathbf{x}, \mathbf{y}, \mathbf{z}$ , it is shorter to go from  $\mathbf{x}$  to  $\mathbf{y}$  than to go from  $\mathbf{x}$  to an intermediate point  $\mathbf{z}$  and then from that point to  $\mathbf{y}$ .

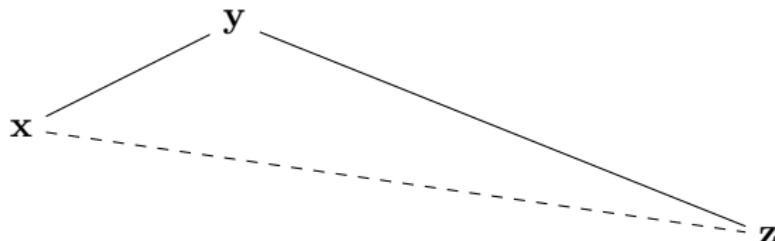
# Metric Distances

- More generally a distance  $d(x, y)$  satisfies the triangle inequality if it is of the form

$$d(x, y) \leq d(x, z) + d(y, z)$$

- Formally the distance is a metric if it is symmetric  $d(x, y) = d(y, x)$ , non negative,  $d(x, y) \geq 0$  and  $d(x, y) = 0 \Leftrightarrow x = y$ .
- Note that the Euclidean distance  $|x - y|$  is a metric – the squared Euclidean distance  $|x - y|^2$  is not a metric (fails triangle inequality).
- The term ‘metric’ is often abused in the literature, referring to any ‘distance’ measure – be aware!

# Exploiting the triangle inequality



- If  $d(x, y) \leq \frac{1}{2}d(z, y)$ , then  $d(x, y) \leq d(x, z)$ , meaning that we do not need to compute  $d(x, z)$ .
- To see this consider

$$d(y, z) \leq d(y, x) + d(x, z)$$

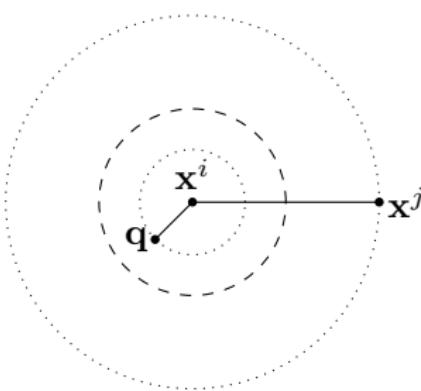
- If we are in the situation that  $d(x, y) \leq \frac{1}{2}d(z, y)$ , then we can write

$$2d(x, y) \leq d(y, x) + d(x, z)$$

and  $d(x, y) \leq d(x, z)$ .

# Exploiting the triangle inequality

In the nearest neighbour context, we can infer that if  $d(\mathbf{q}, \mathbf{x}^i) \leq \frac{1}{2}d(\mathbf{x}^i, \mathbf{x}^j)$  then  $d(\mathbf{q}, \mathbf{x}^i) \leq d(\mathbf{q}, \mathbf{x}^j)$ :



The dashed circle represents points  $\mathbf{q}'$  for which  $d(\mathbf{q}', \mathbf{x}^i) = \frac{1}{2}d(\mathbf{x}^i, \mathbf{x}^j)$ . If we know that  $\mathbf{x}^i$  is close to  $\mathbf{q}$ , but that  $\mathbf{x}^i$  and  $\mathbf{x}^j$  are not close, namely  $d(\mathbf{q}, \mathbf{x}^i) \leq \frac{1}{2}d(\mathbf{x}^i, \mathbf{x}^j)$ , then we can infer that  $\mathbf{x}^j$  will not be closer to  $\mathbf{q}$  than  $\mathbf{x}^i$ , i.e.  $d(\mathbf{q}, \mathbf{x}^i) \leq d(\mathbf{q}, \mathbf{x}^j)$ .

## Orchard's Algorithm

- Precompute all the distance pairs  $d_{ij} \equiv d(\mathbf{x}^i, \mathbf{x}^j)$  in the dataset.
- Given these distances, for each  $i$  we can then compute an ordered list  $\mathcal{L}^i = \{j_1^i, j_2^i, \dots, j_{N-1}^i\}$  of those vectors  $\mathbf{x}^j$  that are closest to  $\mathbf{x}^i$ , with  $d(\mathbf{x}^i, \mathbf{x}^{j_1^i}) \leq d(\mathbf{x}^i, \mathbf{x}^{j_2^i}) \leq d(\mathbf{x}^i, \mathbf{x}^{j_3^i}) \dots$
- We then start with some vector  $\mathbf{x}^i$  as our current best guess for the nearest neighbour to  $\mathbf{q}$  and compute  $d(\mathbf{q}, \mathbf{x}^i)$ . We then examine the first element of the list  $\mathcal{L}^i$  and consider the following cases:
- BOUND CHECK: If  $d(\mathbf{q}, \mathbf{x}^i) \leq \frac{1}{2}d_{i,j_1^i}$  then  $j_1^i$  cannot be closer than  $\mathbf{x}^i$  to  $\mathbf{q}$ ; furthermore, neither can any of the other members of this list since they automatically satisfy this bound as well. In this fortunate situation,  $\mathbf{x}^i$  must be the nearest neighbour to  $\mathbf{q}$ .
- If  $d(\mathbf{q}, \mathbf{x}^i) > \frac{1}{2}d_{i,j_1^i}$  then we compute  $d(\mathbf{q}, \mathbf{x}^{j_1^i})$ . If  $d(\mathbf{q}, \mathbf{x}^{j_1^i}) < d(\mathbf{q}, \mathbf{x}^i)$  we have found a better candidate  $i' \equiv j_1^i$  than our current best guess, and we jump to the start of the new list  $\mathcal{L}^{i'}$ . Otherwise we move down the current list and consider  $j_2^i$  in the BOUND CHECK step above.

# Orchard's Algorithm

## Cost

- Precomputation of distance matrix:  $O(DN^2)$
- Evaluating each member in the list:  $O(D)$
- Need to examine  $M < N$  members in the lists.

---

## Comments

- Orchard's algorithm can work well in low dimensional cases, avoiding the calculation of many distances.
- It requires however a potentially very time consuming one-time calculation of all point to point distances.
- The storage of this inter-point distance matrix can be prohibitive.

# Approximating and Eliminating Search Algorithm (AESAl

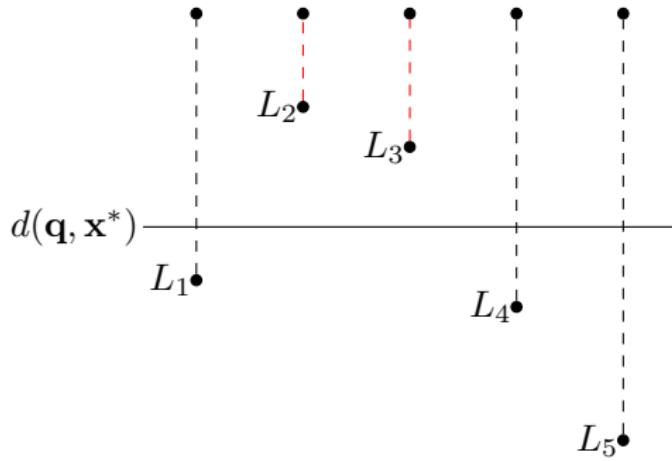
- The triangle inequality can be used to form a lower bound

$$d(\mathbf{q}, \mathbf{x}^j) \geq d(\mathbf{q}, \mathbf{x}^i) - d(\mathbf{x}^i, \mathbf{x}^j)$$

- Define  $\mathcal{I}$  to be the set of datapoints for which  $d(\mathbf{q}, \mathbf{x}^i)$ ,  $i \in \mathcal{I}$  has already been computed. One can then maximise the lower bounds to find the tightest lower bound on all other  $d(\mathbf{q}, \mathbf{x}^j)$

$$d(\mathbf{q}, \mathbf{x}^j) \geq \max_{i \in \mathcal{I}} \{d(\mathbf{q}, \mathbf{x}^i) - d(\mathbf{x}^i, \mathbf{x}^j)\}$$

- All datapoints  $\mathbf{x}^j$  whose lower bound is greater than the current best nearest neighbour distance can then be eliminated. One may then select the next (non-eliminated) candidate datapoint  $\mathbf{x}^j$  corresponding to the lowest bound and continue, updating the bound and eliminating.



We can eliminate datapoints  $x^2$  and  $x^3$  since their distance to the query is greater than the current best candidate distance  $d(q, x^*)$ . After eliminating these points, we use the datapoint with the lowest bound to suggest the next candidate, in this case  $x^5$ .

## AESA Algorithm: Cost

- Precomputation of distance matrix:  $O(DN^2)$
- Evaluating the bound for all  $M$  remaining datapoints:  $O(M(N - M))$
- Need to compute  $M < N$  bounds.

## Using 'buoys'

- Both Orchard's algorithm and AESA can significantly reduce the number of distance calculations required.
- However, we pay an  $O(N^2)$  storage cost. For very large datasets, this storage cost is likely to be prohibitive.
- (More strictly, we can actually compute the distances  $d(\mathbf{x}^i, \mathbf{x}^j)$  'on the fly' when they are required. For a single query this may be effective; however for many different queries, we would end up recomputing these distances – hence we may as well store them.)
- Given the difficulty in storing  $d_{i,j}$ , an alternative is to consider the distances between the training points and a smaller number of strategically placed 'buoys' (also called 'pivots' or 'basis vectors'),  $\mathbf{b}^1, \dots, \mathbf{b}^B$ ,  $B < N$ .
- These buoys can be either a subset of the original datapoints, or new positions.

## Pre-elimination using Buoys

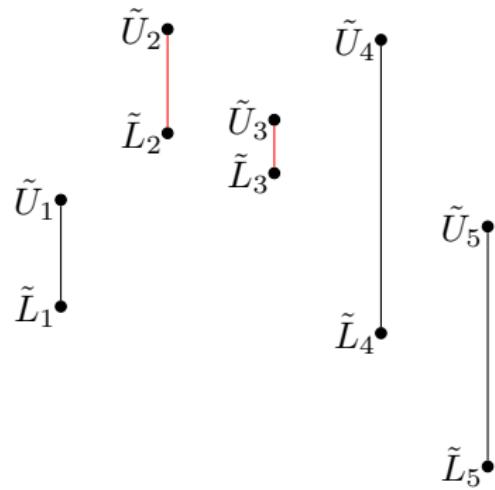
- Given the buoys, the triangle inequality gives the following upper and lower bounds on the distance from the query to each datapoint:

$$d(\mathbf{q}, \mathbf{x}^n) \geq \max_m \{d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{b}^m, \mathbf{x}^n)\} \equiv \tilde{L}_n$$

$$d(\mathbf{q}, \mathbf{x}^n) \leq \min_m \{d(\mathbf{q}, \mathbf{b}^m) + d(\mathbf{b}^m, \mathbf{x}^n)\} \equiv \tilde{U}_n$$

- We can then immediately eliminate any datapoint whose lower bound is greater than the upper bound of some other datapoint.
- This enables one to ‘pre-eliminate’ datapoints, at a cost of  $B$  distance calculations. (Still takes  $O(N)$  operations to do preelimination.)
- The remaining candidates can then be used in the Orchard or AESA algorithms.

## Pre-elimination using bounds

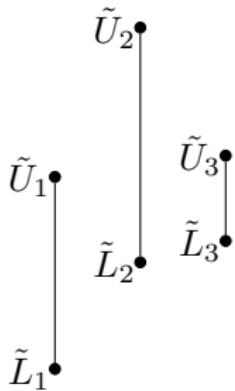


We can eliminate datapoint  $x^2$  since there is another datapoint (either  $x^1$  or  $x^5$ ) that has an upper bound  $U$  that is lower than  $\tilde{L}_2$ . We can similarly eliminate  $x^3$ .

## AESA with buoys

- In place of the exact distances to the datapoints, an alternative is to relabel the datapoints according to  $\tilde{L}_n$ , with lowest distance first  $\tilde{L}_1 \leq \tilde{L}_2, \dots \leq \tilde{L}_N$ .
- We can then compute the distance  $d(\mathbf{q}, \mathbf{x}^1)$  and compare this to  $\tilde{L}_2$ . If  $d(\mathbf{q}, \mathbf{x}^1) \leq \tilde{L}_2$  then  $\mathbf{x}^1$  must be the nearest neighbour, and the algorithm terminates. Otherwise we move on to the next candidate  $\mathbf{x}^2$ .
- If this datapoint has a lower distance than our current best guess, we update our current best guess accordingly. We then move on to the next candidate in the list and continue. If we reach a candidate in the list for which  $d(\mathbf{q}, \mathbf{x}^{best}) \leq \tilde{L}_m$  the algorithm terminates. This algorithm is also called 'linear' AESA.
- The gain here is that the storage costs are reduced to  $O(NB)$  since we only now need to pre-compute the distances between the buoys and the dataset vectors. By choosing  $B \ll N$ , this can be a significant saving. The loss is that, because we are now not using the true distance, but a bound, that we may need more distance calculations  $d(\mathbf{q}, \mathbf{x}^i)$ .

# Linear AESA

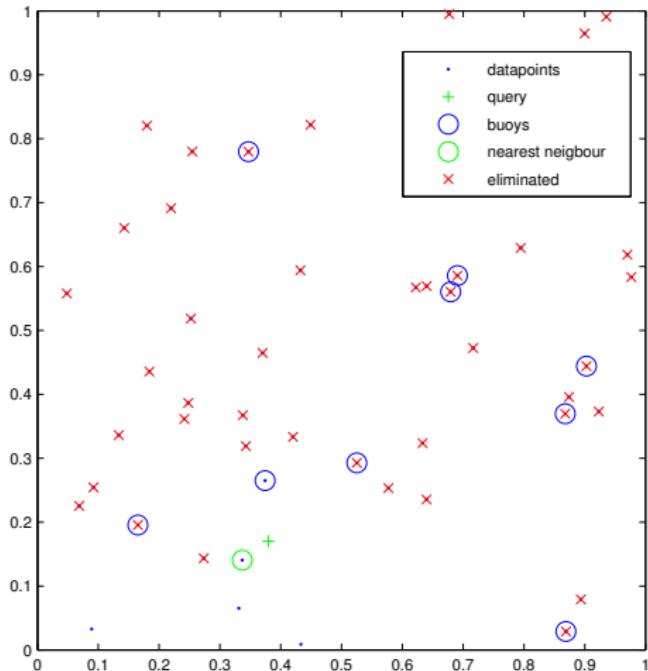


- The lower bounds of non-eliminated datapoints relabelled such that  $\tilde{L}_1 \leq \tilde{L}_2 \leq \dots$
- In ‘linear’ AESA we use these lower bounds to order the search for the nearest neighbour, starting with  $x^1$ .
- If we get to a bound where  $\tilde{L}_m$  is greater than our current best distance, then all remaining distances must be greater than our current best distance, and the algorithm terminates.

## Orchard with buoys

- One can also use buoys to replace the exact distance  $d(\mathbf{x}^i, \mathbf{x}^j)$  in the Orchard algorithm with an upper bound  $d(\mathbf{x}^i, \mathbf{b}) - d(\mathbf{x}^j, \mathbf{b})$ .
- However, one can show that AESA-buoys (linear AESA) dominates Orchard-buoys in terms of the number of distance calculations  $d(\mathbf{q}, \mathbf{x}^i)$  required (see main text).
- No obvious advantage of this approach since computing the upper bounds on the distance requires an  $O(N)$  computation.

# Elimination using buoys



- All points except for the query are datapoints.
- Using buoys, we can pre-eliminate (crossed datapoints) a large number of the datapoints from further consideration.

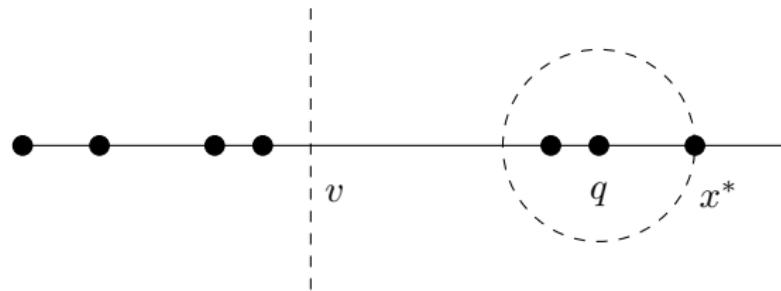
# KD Tree

# KD trees

- K-dimensional trees are a way to form a hierarchical partition of the space that can be used to help speed up search.
- This is a form of ‘spatial data structure’
- An advantage over the Orchard and AESA approaches is that the storage cost of the tree is only  $O(N)$ .
- Also, only in the worst case are  $O(N)$  operations required for a new query.
- Before introducing the tree, we’ll discuss the basic idea on which the potential speed-up is based.

## Basic idea in one-dimension

- If we consider first one-dimensional data  $x^n, n = 1, \dots, N$  we can partition the data into points that have value less than a chosen value  $v$ , and those with a value greater than this.
- If the distance of the current best candidate  $x^*$  to the query point  $q$  is smaller than the distance of the query to  $v$ , then points to the left of  $v$  cannot be the nearest neighbour.



## One dimension case

Consider a query point that is in the right space,  $q > v$  and a candidate  $x$  that is in the left space,  $x < v$ , then

$$(x - q)^2 = (x - v + v - q)^2 = (x - v)^2 + 2\underbrace{(x - v)}_{\leq 0} \underbrace{(v - q)}_{\leq 0} + (v - q)^2 \geq (v - q)^2$$

- Let the distance of the current best candidate to the query be  $\delta^2 \equiv (x^* - q)^2$ .
- Then if  $(v - q)^2 \geq \delta^2$  it follows that all points in the left space are further from  $q$  than  $x^*$ .

## $K$ dimensional case

- In the more general  $K$  dimensional case, consider a query vector  $\mathbf{q}$ .
- Let's imagine that we have partitioned the datapoints into those with first dimension  $x_1$  less than a defined value  $v$  (to its 'left'), and those with a value greater or equal to  $v$  (to its 'right'):

$$\mathcal{L} = \{\mathbf{x}^n : x_1^n < v\}, \quad \mathcal{R} = \{\mathbf{x}^n : x_1^n \geq v\}$$

- Let's also say that our current best nearest neighbour candidate is  $\mathbf{x}^i \in \mathcal{R}$  and that this point has squared Euclidean distance  $\delta^2 = (\mathbf{q} - \mathbf{x}^i)^2$  from  $\mathbf{q}$ .
- The squared Euclidean distance of any datapoint  $\mathbf{x} \in \mathcal{L}$  to the query is

$$(\mathbf{x} - \mathbf{q})^2 = \sum_k (x_k - q_k)^2 \geq (x_1 - q_1)^2 \geq (v - q_1)^2$$

- If  $(v - q_1)^2 \geq \delta^2$ , then  $(\mathbf{x} - \mathbf{q})^2 > \delta^2$ .
- That is, all points in  $\mathcal{L}$  must be further from  $\mathbf{q}$  than the current best point  $\mathbf{x}^i$ .
- On the other hand, if  $(v - q_1)^2 < \delta^2$ , then it is possible that some point in  $\mathcal{L}$  might be closer to  $\mathbf{q}$  than our current best nearest neighbour candidate, and we need to check these points.

## Constructing the tree

- For  $N$  datapoints, the tree consists of  $N$  nodes. Each node contains a datapoint, along with the axis along which the data is split.
- We first need to define a routine

```
[middata leftdata rightdata]=splitdata(x, axis)
```

that splits data along a specified dimension (axis).

**middata** We first form the set of scalar values that correspond to the axis components of  $x$ . These are sorted and `middata` is the datapoint closest to the median of the data.

**leftdata** These are the datapoints ‘to the left’ of the `middata` datapoint.

**rightdata** These are the datapoints ‘to the right’ of the `middata` datapoint.

## Splitting example

If the datapoints are  $(2, 3), (5, 4), (9, 6), (4, 7), (8, 1), (7, 2)$  and we split along dimension 1, then we would have:

```
>>x =
```

2	5	9	4	8	7
3	4	6	7	1	2

```
>> [middata leftdata rightdata]=splitdata(x,1)
```

```
middata =
```

7
2

```
leftdata =
```

2	4	5
3	7	4

```
rightdata =
```

8	9
1	6

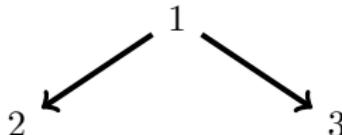
## Tree construction (top layer)

- We start with node(1) and create a temporary storage node(1).data that contains the complete dataset.
- We then call

```
[middata leftdata rightdata]=splitdata(node(1).data,1)
```

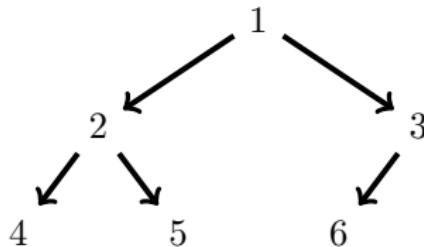
forming node(1).x=middata.

- We now form two child nodes and populate them with data  
node(2).data=leftdata; node(3).data=rightdata.
- We store also in node(1).axis which axis was used to split the data.
- The temporary data node(1).data can now be removed.



## Tree construction (next layer)

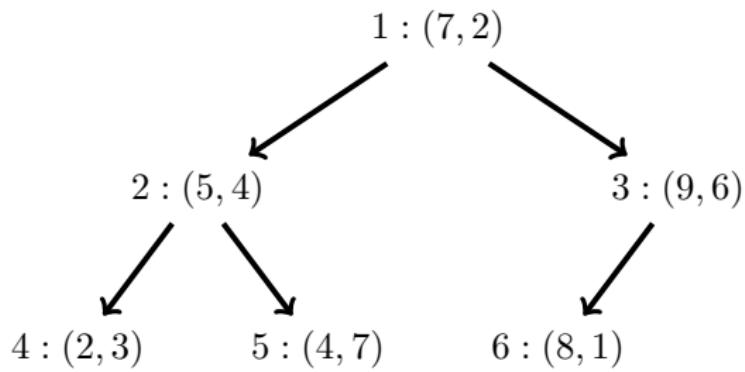
- We now move down to the second layer, and split along the next dimension for nodes in this layer.
- We then go through each of the nodes and split the corresponding data, forming a layer beneath.
- If `leftdata` is empty, then the corresponding child node is not formed, and similarly if `rightdata` is empty.



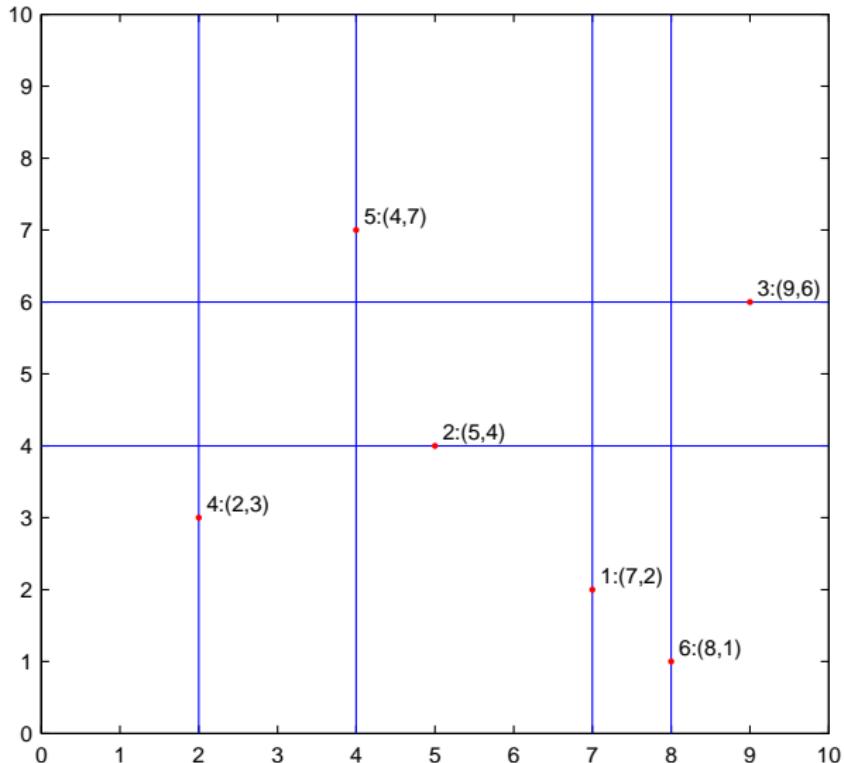
In this way, the top layer splits the data along axis 1, and then the nodes 2 and 3 in the second layer split the data along dimension 2.

- The nodes in the third layer don't require any more splitting since they contain single datapoints.

# Final KD Tree



# Visualisation



The KD tree partitions the space into hyperrectangles.

## Nearest Neighbour search $q = (9, 2)$

- For a query point  $q$  we first find the leaf node of the tree by traversing the tree from the root, and seeing if the corresponding components of  $q$  is ‘to the left’ or ‘to the right’ of the current tree node.
- For the above tree, for the query  $q = (9, 2)$ , we would first consider the value 9 (since the first layer splits along dimension 1).
- Since 9 is ‘to the right’ of 7 (the first dimension of node 1), we go now to node 3.
- In this case, there is a unique child of node 3, so we continue to the leaf, node 6.

## Nearest Neighbour search $q = (9, 2)$

- Node 6 now represents our first guess for the nearest neighbour.
- This has distance  $(9 - 8)^2 + (2 - 1)^2 = 2 \equiv \delta^2$  from the query. We set this to our current best guess of the nearest neighbour and corresponding distance.
- We then go up the tree, to the parent, node 3.
- We check if this is a better neighbour. It has distance  $(9 - 9)^2 + (2 - 6)^2 = 16$ , so this is worse.
- Since there are no other children to consider, we move up another level, this time to node 1. This has distance  $(9 - 7)^2 + (2 - 2)^2 = 4$ , which is also worse than our current best.
- We now need to consider if there are any nodes in the other branch of the tree containing nodes 2,4,5, that could be better than our current best.
- We know that for all nodes 2,4,5 they have first dimensions with value less than 7.
- We can then check if the corresponding datapoints are necessarily further than our current best guess by checking if  $(v - q_1)^2 > \delta^2$ , namely if  $(7 - 9)^2 > 2$ . Since this is true, it must be that all the points in nodes 2,4,5 are further away from the query than our current best guess.
- At this point the algorithm terminates, having examined only 3 datapoints in which to find the nearest neighbour, rather than all 6.

$$q = (6, 5)$$

- With this query, we go down the tree to node  $(4, 7)$
- This gives our initial  $\delta^2 = (6 - 4)^2 + (5 - 7)^2 = 8$
- Go up to  $(5, 4)$ . This has distance from the query  $(6 - 5)^2 + (5 - 4)^2 = 2$ . This is therefore a better neighbour and forms our new best guess for the nearest neighbour.
- What about  $(2, 3)$ ? Here we have  $v = 4$  and check  $v - q_2)^2 = (4 - 5)^2 = 1$ . This is less than  $\delta^2$  so we have to check  $(2, 3)$ . This has distance  $(6 - 2)^2 + (5 - 3)^2 = 20$ . This is bigger than our current  $\delta^2$  (2) so we reject this and move up the tree to  $(7, 2)$ .
- This has distance  $(7 - 6)^2 + (2 - 5)^2 = 10$  which is larger than our current  $\delta^2$  (2) so we reject this.
- Do we need to check the right branch of the tree? Here  $v = 7$  and  $(v - q_1)^2 = (7 - 6)^2 = 1$ . This is less than  $\delta^2$  so we have to check the right branch.
- Continuing in this way, we actually end up having to explicitly check all nodes.
- This is an example of a worst-case situation in which the computation is as bad (even slightly worse) than just calculating all the distances.

# A note on quickly finding the nearest neighbour

David Barber

Department of Computer Science  
University College London

May 17, 2016

## 1 Finding your nearest neighbour quickly

Consider that we have a set of datapoints  $\mathbf{x}^1, \dots, \mathbf{x}^N$  and a new query vector  $\mathbf{q}$ . Our task is to find the nearest neighbour to the query  $n^* = \operatorname{argmin}_n d(\mathbf{q}, \mathbf{x}^n)$ ,  $n = 1, \dots, N$ . For the Euclidean distance

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{(\mathbf{x} - \mathbf{y})^2} = \sqrt{\sum_{i=1}^D (x_i - y_i)^2} \quad (1)$$

and  $D$  dimensional vectors, it takes  $O(D)$  operations to compute this distance. For a set of  $N$  vectors, computing the nearest neighbour to  $\mathbf{q}$  would take then  $O(DN)$  operations. For large datasets this can be prohibitively expensive. Is there a way to avoid calculating all the distances? This is a large research area (see [2] for a review) and we will focus here on first methods that make use of the triangle inequality for metric distances and secondly a KD-trees which form a spatial data structure.

## 2 Using the triangle inequality to speed up search

### 2.1 The triangle inequality

For the Euclidean distance we have

$$(\mathbf{x} - \mathbf{y})^2 = (\mathbf{x} - \mathbf{z} + \mathbf{z} - \mathbf{y})^2 = (\mathbf{x} - \mathbf{z})^2 + (\mathbf{z} - \mathbf{y})^2 + 2(\mathbf{x} - \mathbf{z})(\mathbf{z} - \mathbf{y}) \quad (2)$$

Since the scalar product between two vectors  $\mathbf{a}$  and  $\mathbf{b}$  relates the lengths of the vectors  $|\mathbf{a}|$ ,  $|\mathbf{b}|$  and the angle  $\theta$  between them by  $\mathbf{a}^\top \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos(\theta)$  we have (using  $|\mathbf{x}| \equiv \sqrt{\mathbf{x}^\top \mathbf{x}}$ )

$$|\mathbf{x} - \mathbf{y}|^2 = |\mathbf{x} - \mathbf{z}|^2 + |\mathbf{z} - \mathbf{y}|^2 + 2 \cos(\theta) |\mathbf{x} - \mathbf{z}| |\mathbf{z} - \mathbf{y}| \quad (3)$$

Using  $\cos(\theta) \leq 1$  we obtain the triangle inequality

$$|\mathbf{x} - \mathbf{y}| \leq |\mathbf{x} - \mathbf{z}| + |\mathbf{z} - \mathbf{y}| \quad (4)$$

Geometrically this simply says that for a triangle formed by the points  $\mathbf{x}, \mathbf{y}, \mathbf{z}$ , it is shorter to go from  $\mathbf{x}$  to  $\mathbf{y}$  than to go from  $\mathbf{x}$  to an intermediate point  $\mathbf{z}$  and then from that point to  $\mathbf{y}$ .

More generally a distance  $d(\mathbf{x}, \mathbf{y})$  satisfies the triangle inequality if it is of the form<sup>1</sup>

$$d(\mathbf{x}, \mathbf{y}) \leq d(\mathbf{x}, \mathbf{z}) + d(\mathbf{y}, \mathbf{z}) \quad (5)$$

Formally the distance is a metric if it is symmetric  $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$ , non negative,  $d(\mathbf{x}, \mathbf{y}) \geq 0$  and  $d(\mathbf{x}, \mathbf{y}) = 0 \Leftrightarrow \mathbf{x} = \mathbf{y}$ .

---

<sup>1</sup>Note here that  $d(\cdot, \cdot)$  is defined as the Euclidean distance, not the squared Euclidean distance.

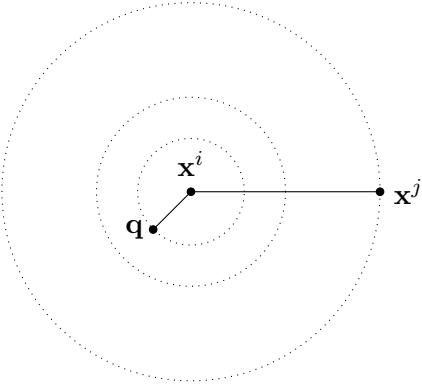


Figure 1: If we know that  $\mathbf{x}^i$  is close to  $\mathbf{q}$ , but that  $\mathbf{x}^i$  and  $\mathbf{x}^j$  are not close, namely  $d(\mathbf{q}, \mathbf{x}^i) \leq \frac{1}{2}d(\mathbf{x}^i, \mathbf{x}^j)$ , then we can infer that  $\mathbf{x}^j$  will not be closer to  $\mathbf{q}$  than  $\mathbf{x}^i$ , i.e.  $d(\mathbf{q}, \mathbf{x}^i) \leq d(\mathbf{q}, \mathbf{x}^j)$ .

A useful basic fact that we can deduce for such metric distances is the following: If  $d(\mathbf{x}, \mathbf{y}) \leq \frac{1}{2}d(\mathbf{z}, \mathbf{y})$ , then  $d(\mathbf{x}, \mathbf{y}) \leq d(\mathbf{x}, \mathbf{z})$ , meaning that we do not need to compute  $d(\mathbf{x}, \mathbf{z})$ . To see this consider

$$d(\mathbf{y}, \mathbf{z}) \leq d(\mathbf{y}, \mathbf{x}) + d(\mathbf{x}, \mathbf{z}) \quad (6)$$

If we are in the situation that  $d(\mathbf{x}, \mathbf{y}) \leq \frac{1}{2}d(\mathbf{z}, \mathbf{y})$ , then we can write

$$2d(\mathbf{x}, \mathbf{y}) \leq d(\mathbf{y}, \mathbf{x}) + d(\mathbf{x}, \mathbf{z}) \quad (7)$$

and hence  $d(\mathbf{x}, \mathbf{y}) \leq d(\mathbf{x}, \mathbf{z})$ . In the nearest neighbour context, we can infer that if  $d(\mathbf{q}, \mathbf{x}^i) \leq \frac{1}{2}d(\mathbf{x}^i, \mathbf{x}^j)$  then  $d(\mathbf{q}, \mathbf{x}^i) \leq d(\mathbf{q}, \mathbf{x}^j)$ , see fig(1).

## 2.2 Using all datapoint to datapoint distances

One way to use the above result is to first precompute all the distance pairs  $d_{ij} \equiv d(\mathbf{x}^i, \mathbf{x}^j)$  in the dataset.

### Orchard's algorithm

Given these distances, for each  $i$  we can then compute an ordered list  $\mathcal{L}^i = \{j_1^i, j_2^i, \dots, j^{iN-1}\}$  of those vectors  $\mathbf{x}^j$  that are closest to  $\mathbf{x}^i$ , with  $d(\mathbf{x}^i, \mathbf{x}^{j_1^i}) \leq d(\mathbf{x}^i, \mathbf{x}^{j_2^i}) \leq d(\mathbf{x}^i, \mathbf{x}^{j_3^i}) \dots$

- We then start with some vector  $\mathbf{x}^i$  as our current best guess for the nearest neighbour to  $\mathbf{q}$  and compute  $d(\mathbf{q}, \mathbf{x}^i)$ . We then examine the first element of the list  $\mathcal{L}^i$  and consider the following cases:
- BOUND CHECK: If  $d(\mathbf{q}, \mathbf{x}^i) \leq \frac{1}{2}d_{i,j_1^i}$  then  $j_1^i$  cannot be closer than  $\mathbf{x}^i$  to  $\mathbf{q}$ ; furthermore, neither can any of the other members of this list since they automatically satisfy this bound as well. In this fortunate situation,  $\mathbf{x}^i$  must be the nearest neighbour to  $\mathbf{q}$ .
- If  $d(\mathbf{q}, \mathbf{x}^i) \not\leq \frac{1}{2}d_{i,j_1^i}$  then we compute  $d(\mathbf{q}, \mathbf{x}^{j_1^i})$ . If  $d(\mathbf{q}, \mathbf{x}^{j_1^i}) < d(\mathbf{q}, \mathbf{x}^i)$  we have found a better candidate  $i' \equiv j_1^i$  than our current best guess, and we jump to the start of the new list  $\mathcal{L}^{i'}$ . Otherwise we continue to traverse the current list and consider  $j_2^i$  in the BOUND CHECK step above, checking if  $d(\mathbf{q}, \mathbf{x}^i) \leq \frac{1}{2}d_{i,j_2^i}$ , etc. [6].

In summary, this process of traversing a list for the candidate continues until we either: (i) find a better candidate (and jump to the top of its list) and restart traversing the new candidate list (ii) find that the bound criterion is met and declare that the current candidate is then optimal (iii) get to the end of a list, in which case the current candidate is optimal.

See algorithm(1) and `fastnnOrchard.m` for a formal description of the algorithm.

### Complexity

Orchard's algorithm requires  $O(D^2)$  storage to precompute the distance sorted lists. In the worst case, it can take  $O(N)$  distance calculations to find the nearest neighbour. To see this, consider a simple one dimensional dataset:

$$x^{n+1} = x^n + 1, \quad n = 1, \dots, N \quad x^1 = 0 \quad (8)$$

**Algorithm 1** Orchard's Nearest Neighbour Search

---

```

1: Compute all pairwise distances metric(data{m},data{n})
2: For each datapoint  $n$ , compute the list{n} that stores the distance list{n}.distance and index list{n}.index
   of each other datapoint, sorted by increasing distance, list{n}.distance(1);list{n}.distance(2)...
3: for all Query points do
4:   cand.index=randi( $N$ )            $\triangleright$  assign first candidate index randomly as one of the  $N$  datapoint indices
5:   cand.distance=metric(query,data{cand.index})
6:   Assign all nodes to state not tested
7:    $i = 1$                           $\triangleright$  start at the beginning of the list
8:   while  $i \leq N$  and list{cand.index}.distance( $i$ )<2*cand.distance do
9:     node=list{cand.index}.index( $i$ )            $\triangleright$  get the index of the  $i^{th}$  member of the current list
10:    if tested(node)=false then            $\triangleright$  just to avoid computing this distance again
11:      tested(node)=true
12:      querydistance=metric(query,data{node})
13:      if querydistance<cand.distance then            $\triangleright$  found a better candidate
14:        cand.index=node;
15:        cand.distance=querydistance;
16:         $i = 1$                           $\triangleright$  go to start of next list
17:      else
18:         $i = i + 1$                     $\triangleright$  continue to traverse the current list
19:      end if
20:       $i = i + 1$ 
21:    end if
22:  end while
23:  cand.index and cand.distance contain the nearest neighbour and distance thereto
24: end for

```

---

If the query point is for example  $q = x^N + 1$  and our initial candidate for the nearest neighbour is  $x^1$ , then at iteration  $k$ , the nearest non-visited datapoint will be  $x^{k+1}$ , meaning that we will simply walk through all the data, see fig(2).

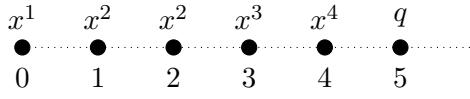


Figure 2: A worst-case scenario for Orchard's algorithm. If the initial candidate is  $x^1$ , then  $x^2$  becomes the next best candidate, and subsequently  $x^3$  etc.

**Approximating and Eliminating Search Algorithm (AESa)**

The triangle inequality can be used to form a lower bound

$$d(\mathbf{q}, \mathbf{x}^j) \geq d(\mathbf{q}, \mathbf{x}^i) - d(\mathbf{x}^i, \mathbf{x}^j) \quad (9)$$

For datapoints  $\mathbf{x}^i$ ,  $i \in \mathcal{I}$  for which  $d(\mathbf{q}, \mathbf{x}^i)$  has already been computed, one can then maximise the lower bounds to find the tightest lower bound on all other  $d(\mathbf{q}, \mathbf{x}^j)$ <sup>2</sup>:

$$d(\mathbf{q}, \mathbf{x}^j) \geq \max_{i \in \mathcal{I}} d(\mathbf{q}, \mathbf{x}^i) - d(\mathbf{x}^i, \mathbf{x}^j) \equiv L_j \quad (10)$$

All datapoints  $\mathbf{x}^j$  whose lower bound is greater than the current best nearest neighbour distance can then be eliminated, see fig(3). One may then select the next (non-eliminated) candidate datapoint  $\mathbf{x}^j$  corresponding to the lowest bound and continue, updating the bound and eliminating [7], see algorithm(2) and `fastnnAESa.m`.

---

<sup>2</sup>In the classic AESA one only retains the best current nearest neighbour  $\mathcal{I} = best$ .

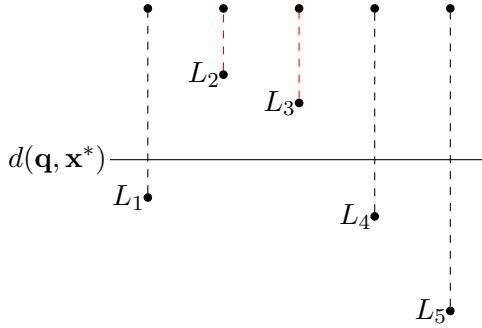


Figure 3: We can eliminate datapoints  $\mathbf{x}^2$  and  $\mathbf{x}^3$  since their distance to the query is greater than the current best candidate distance  $d(\mathbf{q}, \mathbf{x}^*)$ . After eliminating these points, we use the datapoint with the lowest bound to suggest the next candidate, in this case  $\mathbf{x}^5$ .

---

**Algorithm 2** AESA Nearest Neighbour Search

---

```

1: For all datapoints compute and store the distances  $d(\mathbf{x}^i, \mathbf{x}^j)$ 
2: for all Queries do
3:   best.dist= $\infty$ 
4:    $\mathcal{I} = \emptyset$                                  $\triangleright$  Datapoints examined
5:    $\mathcal{J} = \{1, \dots, N\}$                        $\triangleright$  Datapoints not examined
6:    $L(n) = \infty, \quad n = 1, \dots, N$ 
7:   while  $\mathcal{J}$  is not empty do
8:     cand.ind= $\arg \min_{j \in \mathcal{J}} \text{bound}(j)$            $\triangleright$  select candidate based on lowest bound
9:     cand.dist=metric(query,data{cand.ind})
10:    distQueryData(cand.ind)=cand.dist            $\triangleright$  store computed distances
11:     $\mathcal{I} = \mathcal{I} \cup \text{cand.ind}$              $\triangleright$  Add candidate to examined list
12:    if cand.dist<best.dist then            $\triangleright$  If candidate is nearer than current best
13:      best.dist=cand.dist
14:      best.ind=cand.ind
15:    end if
16:     $L(j) = \max_{i \in \mathcal{I}} \text{distQueryData}(i) - d(\mathbf{x}^i, \mathbf{x}^j), \quad j \in \mathcal{J} \setminus \text{cand.ind}$        $\triangleright$  lower bound
17:     $\mathcal{J} = \{j \text{ such that } L(j) < \text{best.dist}\}$            $\triangleright$  eliminate
18:  end while
19: end for

```

---

## Complexity

As for Orchard’s algorithm, AESE requires  $O(D^2)$  storage to precompute the distance matrix  $d(\mathbf{x}^i, \mathbf{x}^j)$ . During the first lower bound computation, when no datapoints have yet been eliminated, AESE needs to compute all the  $N - 1$  lower bounds for datapoints other than the first candidate examined. Whilst each lower bound is fast to compute, this still requires an  $O(N)$  computation. A similar computation is required to update the bounds at later stages, with the worst case being that there are  $N/2$  datapoints left to examine, meaning that computing the optimal lower bound scales  $O(N^2)$  in this case. One can limit this complexity by restricting the elements of  $\mathcal{I}$  in the max operation to compute the bound; however this may result in more iterations being required since the bound is then potentially inferior.

The experiments in `demofastnn.m` also suggest that the AESA method typically requires less distance calculations than Orchard’s approach. However, there remains at least an  $O(N)$  calculation required to compute the bounds in AESA which may be prohibitive, depending on the application.

## 2.3 Using the datapoints to ‘buoys’ distances

Both Orchard’s algorithm and AESA can significantly reduce the number of distance calculations required. However, we pay an  $O(N^2)$  storage cost. For very large datasets, this storage cost is likely to be prohibitive. Given the difficulty in storing  $d_{i,j}$ , an alternative is to consider the distances between the training points and a smaller number of strategically placed ‘buoys’<sup>3</sup>,  $\mathbf{b}^1, \dots, \mathbf{b}^B$ ,  $B < N$ . These buoys can be either a subset of the original datapoints, or new positions.

<sup>3</sup>Also called ‘pivots’ or ‘basis’ vectors by other authors.

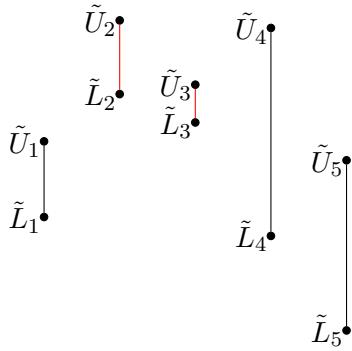


Figure 4: We can eliminate datapoint  $\mathbf{x}^2$  since there is another datapoint (either  $\mathbf{x}^1$  or  $\mathbf{x}^5$ ) that has an upper bound  $\tilde{U}$  that is lower than  $\tilde{L}_2$ . We can similarly eliminate  $\mathbf{x}^3$ .

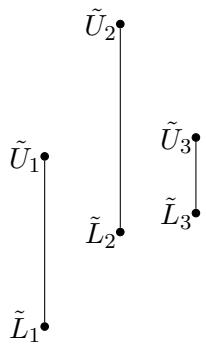


Figure 5: The lower bounds of non-eliminated datapoints from fig(4) relabelled such that  $L_1 \leq L_2 \leq \dots$ . In ‘linear’ AESA we use these lower bounds to order the search for the nearest neighbour, starting with  $\mathbf{x}^1$ . If we get to a bound where  $L_m$  is greater than our current best distance, then all remaining distances must be greater than our current best distance, and the algorithm terminates.

## Pre-elimination

Given the buoys, the triangle inequality gives the following upper and lower bounds on the distance from the query to each datapoint:

$$d(\mathbf{q}, \mathbf{x}^n) \geq \max_m d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{b}^m, \mathbf{x}^n) \equiv \tilde{L}_n \quad (11)$$

$$d(\mathbf{q}, \mathbf{x}^n) \leq \min_m d(\mathbf{q}, \mathbf{b}^m) + d(\mathbf{b}^m, \mathbf{x}^n) \equiv \tilde{U}_n \quad (12)$$

We can then immediately eliminate any  $m$  for which there is some  $n \neq m$  with  $\tilde{L}(m) \geq \tilde{U}(n)$ , see fig(4). This enables one to ‘pre-eliminate’ datapoints, at a cost of  $B$  distance calculations, see `fastnnBuoyElim.m`. The remaining candidates can then be used in the Orchard or AESA algorithms (either the standard ones described above or the buoy variants described below).

## AESA with buoys

In place of the exact distances to the datapoints, an alternative is to relabel the datapoints according to  $\tilde{L}_n$ , with lowest distance first  $\tilde{L}_1 \leq \tilde{L}_2, \dots \leq \tilde{L}_n$ . We can then compute the distance  $d(\mathbf{q}, \mathbf{x}^1)$  and compare this to  $\tilde{L}_2$ . If  $d(\mathbf{q}, \mathbf{x}^1) \leq \tilde{L}_2$  then  $\mathbf{x}^1$  must be the nearest neighbour, and the algorithm terminates. Otherwise we move on to the next candidate  $\mathbf{x}^2$ . If this datapoint has a lower distance than our current best guess, we update our current best guess accordingly. We then move on to the next candidate in the list and continue. If we reach a candidate in the list for which  $d(\mathbf{q}, \mathbf{x}^{best}) \leq \tilde{L}_m$  the algorithm terminates, see fig(5). This algorithm is also called ‘linear’ AESA [5], see `fastnnLAESA.m`.

The gain here is that the storage costs are reduced to  $O(NB)$  since we only now need to pre-compute the distances between the buoys and the dataset vectors. By choosing  $B \ll N$ , this can be a significant saving. The loss is that, since we are now not using the true distance but a bound, we may need more distance calculations  $d(\mathbf{q}, \mathbf{x}^i)$ .

## Orchard with buoys

For Orchard’s algorithm we can also use buoys to construct bounds on  $d_{i,j}$  on the fly. Consider an arbitrary vector  $\mathbf{b}$ , then,

$$d(\mathbf{x}^i, \mathbf{b}) - d(\mathbf{x}^j, \mathbf{b}) \leq d(\mathbf{x}^i, \mathbf{x}^j) \quad (13)$$

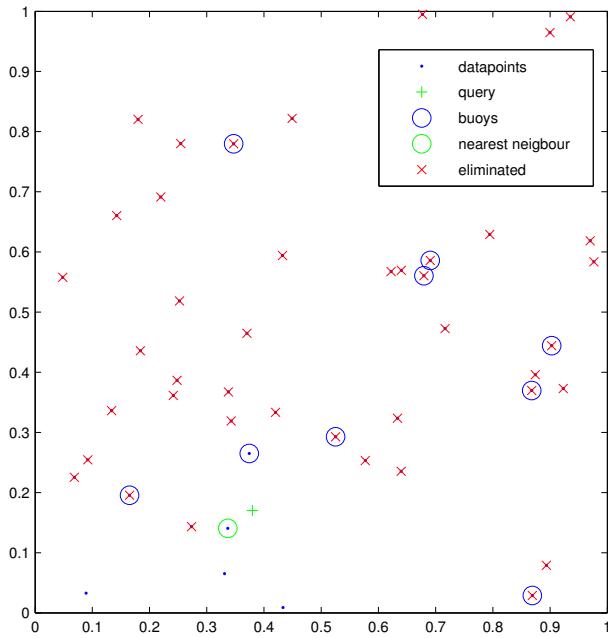


Figure 6: Example of elimination using buoys. All points except for the query are datapoints. Using buoys, we can pre-eliminate (crossed datapoints) a large number of the datapoints from further consideration. See `demofastnn.m`.

We can use this in Orchard's approach since if there is a vector  $\mathbf{b}$  such that

$$2d(\mathbf{q}, \mathbf{x}^i) \leq d(\mathbf{x}^i, \mathbf{b}) - d(\mathbf{x}^j, \mathbf{b}) \quad (14)$$

then it follows that  $d(\mathbf{q}, \mathbf{x}^i) \leq d(\mathbf{q}, \mathbf{x}^j)$ . This suggests that we can replace using the exact distance  $d(\mathbf{x}^i, \mathbf{x}^j)$  with an upper bound  $d(\mathbf{x}^i, \mathbf{b}) - d(\mathbf{x}^j, \mathbf{b})$ . If we have a set of such buoys  $\mathbf{b}^1, \dots, \mathbf{b}^B$ , we want to use the highest lower bound approximation to the true distance  $d(\mathbf{x}^i, \mathbf{x}^j)$ :

$$2d(\mathbf{q}, \mathbf{x}^i) \leq \max_m d(\mathbf{x}^i, \mathbf{b}^m) - d(\mathbf{x}^j, \mathbf{b}^m) \equiv \tilde{d}_{i,j} \quad (15)$$

These surrogate distances  $\tilde{d}_{i,j}$  can then be used as in the standard Orchard algorithm to form (on the fly) a list  $\mathcal{L}^i$  of closest vectors  $\mathbf{x}^j$  to the current candidate  $\mathbf{x}^i$ , sorted according to this surrogate distance. The algorithm then proceeds as before, see `fastnnOrchardBuoys.m`.

Whilst this may seem useful, one can show that AESA-buoys dominates Orchard-buoys. In AESA-buoys the lower bound on  $d(\mathbf{q}, \mathbf{x}^i)$  is given by

$$\tilde{L}_i = \max_m d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{x}^i, \mathbf{b}^m) \quad (16)$$

Let  $m$  be the optimal buoy index for  $i$ , namely

$$\tilde{L}_i = d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{x}^i, \mathbf{b}^m) \quad (17)$$

Furthermore

$$\tilde{L}_j = \max_m d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{x}^j, \mathbf{b}^m) \geq d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{x}^j, \mathbf{b}^m) \quad (18)$$

If we have reached the Orchard-buoys termination criterion

$$-d(\mathbf{x}^j, \mathbf{b}^m) \geq 2d(\mathbf{q}, \mathbf{x}^i) - d(\mathbf{x}^i, \mathbf{b}^m) \quad \text{for all } j > i \quad (19)$$

then datapoint  $\mathbf{x}^i$  is the nearest neighbour. Hence, if Orchard-buoys terminates for  $\mathbf{x}^i$ , we must have, for  $j > i$ :

$$\tilde{L}_j \geq d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{x}^i, \mathbf{b}^m) + 2d(\mathbf{q}, \mathbf{x}^i) \geq d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{x}^i, \mathbf{b}^m) = \tilde{L}_i \quad \text{for all } j > i \quad (20)$$

Furthermore,

$$\begin{aligned} \tilde{L}_j &\geq d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{x}^j, \mathbf{b}^m) \\ &\geq d(\mathbf{q}, \mathbf{b}^m) + d(\mathbf{q}, \mathbf{x}^i) - d(\mathbf{x}^i, \mathbf{b}^m) + d(\mathbf{q}, \mathbf{x}^i) \\ &\geq \underbrace{d(\mathbf{b}^m, \mathbf{x}^i)}_0 - d(\mathbf{x}^i, \mathbf{b}^m) + d(\mathbf{q}, \mathbf{x}^i) \quad \text{for all } j > i \end{aligned} \quad (21)$$

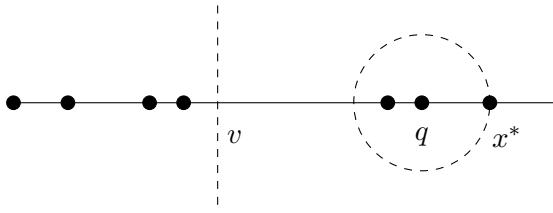


Figure 7: Consider one dimensional data in which the datapoints are partitioned into those that lie to the left of  $v$  and those to the right. If the current best candidate  $x^*$  has a distance to the query  $q$  that is less than the distance of the query  $q$  to  $v$ , then none of the points to the left of  $v$  can be the nearest neighbour.

which is precisely the AESA-buoys termination criterion. Hence Orchard-buoys terminating implies that AESA-buoys terminates. Since AESA-buoys has a different termination criterion to Orchard-buoys, AESA-buoys has the opportunity to terminate before Orchard-buoys. This explains why Orchard-buoys cannot outperform AESA-buoys in terms of the number of distance calculations  $d(\mathbf{q}, \mathbf{x}^i)$ . This observation is also borne out in `demofastnn.m`, see also fig(6) for a  $D = 2$  demonstration.

## 2.4 Other uses of nearest neighbours

Note also that nearest neighbour calculations are required in other applications, for example k-means clustering – see [4] for a fast algorithm based on similar applications of the triangle inequality.

# 3 KD trees

K-dimensional trees [1] are a way to form a partition of the space that can be used to help speed up search. Before introducing the tree, we'll discuss the basic idea on which the potential speed-up is based.

## 3.1 Basic idea in one-dimension

If we consider first one-dimensional data  $x^n, n = 1, \dots, N$  we can partition the data into points that have value less than a chosen value  $v$ , and those with a value greater than this, see fig(7). If the distance of the current best candidate  $x^*$  to the query point  $q$  is smaller than the distance of the query to  $v$ , then points to the left of  $v$  cannot be the nearest neighbour. To see this, consider a query point that is in the right space,  $q > v$  and a candidate  $x$  that is in the left space,  $x < v$ , then

$$(x - q)^2 = (x - v + v - q)^2 = (x - v)^2 + 2\underbrace{(x - v)}_{\leq 0} \underbrace{(v - q)}_{\leq 0} + (v - q)^2 \geq (v - q)^2 \quad (22)$$

Let the distance of the current best candidate to the query be  $\delta^2 \equiv (x^* - q)^2$ . Then if  $(v - q)^2 \geq \delta^2$  it follows that all points in the left space are further from  $q$  than  $x^*$ .

In the more general  $K$  dimensional case, consider a query vector  $\mathbf{q}$ . Let's imagine that we have partitioned the datapoints into those with first dimension  $x_1$  less than a defined value  $v$  (to its ‘left’), and those with a value greater or equal to  $v$  (to its ‘right’):

$$\mathcal{L} = \{\mathbf{x}^n : x_1^n < v\}, \quad \mathcal{R} = \{\mathbf{x}^n : x_1^n \geq v\} \quad (23)$$

Let's also say that our current best nearest neighbour candidate has squared Euclidean distance  $\delta^2 = (\mathbf{q} - \mathbf{x}^i)^2$  from  $\mathbf{q}$  and that  $q_1 \geq v$ . The squared Euclidean distance of any datapoint  $\mathbf{x} \in \mathcal{L}$  to the query is

$$(\mathbf{x} - \mathbf{q})^2 = \sum_k (x_k - q_k)^2 \geq (x_1 - q_1)^2 \geq (v - q_1)^2 \quad (24)$$

If  $(v - q_1)^2 > \delta^2$ , then  $(\mathbf{x} - \mathbf{q})^2 > \delta^2$ . That is, all points in  $\mathcal{L}$  must be further from  $\mathbf{q}$  than the current best point  $\mathbf{x}^i$ . On the other hand, if  $(v - q_1)^2 \leq \delta^2$ , then it is possible that some point in  $\mathcal{L}$  might be closer to  $\mathbf{q}$  than our current best nearest neighbour candidate, and we need to check these points.

The KD-tree approach essentially is a recursive application of the above intuition.

### 3.2 Constructing the tree

For  $N$  datapoints, the tree consists of  $N$  nodes. Each node contains a datapoint, along with the axis along which the data is split.

We first need to define a routine `[middata leftdata rightdata]=splitdata(x, axis)` that splits data along a specified dimension of the data. Whilst not necessary, it is customary to use the median value along the split dimension to partition. The routine should return :

**middata** We first form the set of scalar values that correspond to the `axis` components of `x`. These are sorted and the `middata` is the datapoint close to the median of the data.

**leftdata** These are the datapoints ‘to the left’ of the `middata` datapoint.

**rightdata** These are the datapoints ‘to the right’ of the `middata` datapoint.

For example, if the datapoints are  $(2, 3), (5, 4), (9, 6), (4, 7), (8, 1), (7, 2)$  and we split along dimension 1, then we would have:

```
>>x =
```

2	5	9	4	8	7
3	4	6	7	1	2

```
>> [middata leftdata rightdata]=splitdata(x,1)
```

```
middata =
```

7
2

```
leftdata =
```

2	4	5
3	7	4

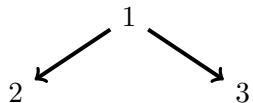
```
rightdata =
```

8	9
1	6

The tree can be constructed recursively as follows. We start with `node(1)` and create a temporary storage `node(1).data` that contains the complete dataset. We then call

```
[middata leftdata rightdata]=splitdata(node(1).data,1)
```

forming `node(1).x=middata`. We now form two child nodes and populate them with data `node(2).data=leftdata`; `node(3).data=rightdata`. We store also in `node(1).axis` which axis was used to split the data.



The temporary data `node(1).data` can now be removed. We now move down to the second layer, and split along the next dimension for nodes in this layer. We then go through each of the nodes and split the corresponding data, forming a layer beneath. If `leftdata` is empty, then the corresponding child node is not formed, and similarly if `rightdata` is empty:

**Algorithm 3** KD tree construction

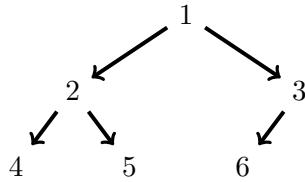
---

```

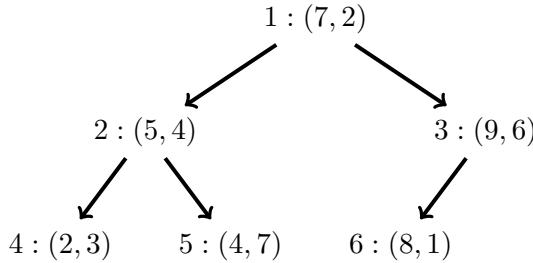
function [node A]=KDTreeMake(x)
[D N]=size(x);
A=sparse(N,N);
node(1).data=x;
for level=1:1+floor(log2(N)) % generate the binary tree:
    split_dimension=rem(level-1,D)+1; % cycle over split dimensions
    if level>1
        pa=layer{level-1}; % parents
        ch=children(A,pa); % children
    else
        ch=1;
    end
    idx=max(ch);
    layer{level}=ch;
    for i=ch
        [node(i).x leftdata rightdata]=splitdata(node(i).data,split_dimension);
        node(i).split_dimension=split_dimension;
        if ~isempty(leftdata); idx=idx+1; node(idx).data=leftdata; A(i,idx)=1; end
        if ~isempty(rightdata); idx=idx+1; node(idx).data=rightdata; A(i,idx)=1; end
        node(i).data=[]; % remove to save storage
    end
end

```

---



In this way, the top layer split the data along axis 1, and then the nodes 2 and 3 in the second layer split the data along dimension 2. The nodes in the third layer don't require any more splitting since they contain single datapoints. Recursive programming is a natural way to construct the tree. Alternatively, one can avoid this by explicitly constructing the tree layer by layer. See algorithm(3) and `KDTreeMake.m` for the full algorithm details in MATLAB. We can also depict the datapoints that each node represents:



The hierarchical partitioning of the space that this tree represents can also be visualised, see fig(8). There are many extensions of the KD tree, for example to search for the nearest  $K$  neighbours, or search for datapoints in a particular range. Different hierarchical partitioning strategies, including non-axis aligned partitions can also be considered. See [3] for further discussion.

## Complexity

Building a KD tree has  $O(N \log N)$  time complexity and  $O(KN)$  space complexity.

### 3.3 Nearest Neighbour search

To search we can make recursive use of our simple observation in section(3.1). For a query point  $\mathbf{q}$  we first find the leaf node of the tree by traversing the tree from the root, and seeing if the corresponding components of  $\mathbf{q}$  is ‘to the left’ or ‘to the right’ of the current tree node. For the above tree, for the query  $\mathbf{q} = (9, 2)$ , we would first consider the value 9 (since the first layer splits along dimension 1). Since 9 is ‘to

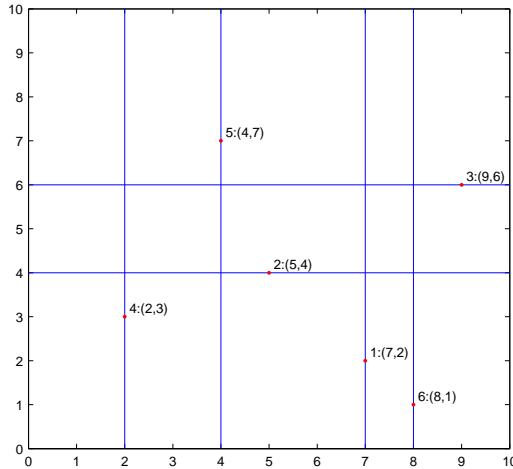


Figure 8: KD tree. The data is plotted along with its node index, and corresponding split dimension (either horizontal or vertical). The KD tree partitions the space into hyperrectangles.

the right' of 7 (the first dimension of node 1), we go now to node 3. In this case, there is a unique child of node 3, so we continue to the leaf, node 6.

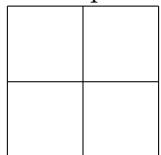
Node 6 now represents our first guess for the nearest neighbour. This has distance  $(9-8)^2 + (2-1)^2 = 2 \equiv \delta^2$  from the query. We set this to our current best guess of the nearest neighbour and corresponding distance. We then go up the tree, to the parent, node 3. We check if this is a better neighbour. It has distance  $(9-9)^2 + (2-6)^2 = 16$ , so this is worse. Since there are no other children to consider, we move up another level, this time to node 1. This has distance  $(9-7)^2 + (2-2)^2 = 4$ , which is also worse than our current best. We now need to consider if there are any nodes in the other branch of the tree containing nodes 2,4,5, that could be better than our current best. We know that for all nodes 2,4,5 they have first dimensions with value less than 7. We can then check if the corresponding datapoints are necessarily further than our current best guess by checking if  $(v - q_1)^2 > \delta^2$ , namely if  $(7-9)^2 > 2$ . Since this is true, it must be that all the points in nodes 2,4,5 are further away from the query than our current best guess. At this point the algorithm terminates, having examined only 3 datapoints in which to find the nearest neighbour, rather than all 6. The full procedure is given in MATLAB in algorithm(4) `KDTreeNN.m` (again using non-recursive programming). See also `demoKDTree.m` for a demonstration.

### Complexity

Searching a KD tree has in the worst case  $O(N)$  time complexity. For ‘dense’ and non-pathological data, typically the algorithm is much more efficient and  $O(\log N)$  time complexity can be achieved.

## 4 Curse of Dimensionality

Whilst KD-trees and Orchard’s approach can work well when the data is reasonably uniformly distributed over the space, their efficacy can reduce dramatically when the space is not well covered by the data. If we split each of the  $D$  dimensions into 2 parts, for example in 2 dimensions:



we will have  $2^D$  partitions of the data. For data to be uniformly distributed, we need a dataset to occupy each of these partitions. This means that we need at least  $2^D$  datapoints in order to begin to see reasonable coverage of the space. Without this exponentially large number of datapoints, most of these partitions will be empty, with the effectively small number of datapoints very sparsely scattered throughout the space. In this case there is little speed up that can be expected based on either triangle or KD tree methods.

It is also instructive to understand that in high dimensions, two datapoints will typically be far apart. To

**Algorithm 4** KD tree nearest neighbour search

---

```

function [bestx bestdist]=KDTreeNN(q,x)
[node A]=KDTreeMake(x);
bestdist=realmax; N=size(x,2); tested=false(1,N);
treenode=1; % start at the top of the tree
for loop=1:N % maximum possible number of loops
    % assign query point to a leaf node (in the remaining tree):
    for level=1:1+floor(log2(N))
        ch=children(A,treenode);
        if isempty(ch); break; end % hit a leaf
        if length(ch)==1
            treenode=ch(1);
        else
            if q(node(treenode).split_dimension)<node(treenode).x(node(treenode).split_dimension);
                treenode=ch(1);
            else
                treenode=ch(2);
            end
        end
    end
end

% check if leaf is closer than current best:
if ~tested(treenode)
    testx=node(treenode).x; dist=sum((q-testx).^2);
    tested(treenode)=true;
    if dist<bestdist; bestdist=dist; bestx=testx; end
end

parentnode=parents(A,treenode);
if isempty(parentnode); break; end % finished searching all nodes
A(parentnode,treenode)=0; % remove child from tree to stop searching

% first check if this is a better node:
if ~tested(parentnode)
    testx=node(parentnode).x; dist=sum((q-testx).^2);
    if dist<bestdist; bestdist=dist; bestx=testx; end
    tested(parentnode)=true;
end
% see if could be points closer on the other branch:
if (node(parentnode).x(node(parentnode).split_dimension) ...
- q(node(parentnode).split_dimension))^2>bestdist
    A(parentnode,children(A,parentnode))=0; % if not then remove this branch
end
treenode=parentnode; % move up to parent on tree
end

```

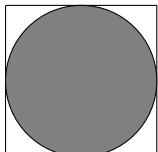
---

see this, consider the volume of a hypersphere of radius  $r$  in  $D$  dimensions. This is given by the expression

$$V = \frac{\pi^{D/2} r^D}{(D/2)!} \quad (25)$$

The fraction of volume that a unit hypersphere occupies when inscribed by a unit hypercube is

$$\frac{\pi^{D/2}}{2^D (D/2)!} \quad (26)$$



This value drops exponentially quickly with the dimension of the space  $D$ , meaning that nearly all points lie in the ‘corners’ of the hypercube. Hence two randomly chosen points will typically be in different corners and far apart – nearest neighbours are likely to be a long distance away. The triangle and KD tree approaches are effective in cases where data is locally concentrated, a situation highly unlikely to occur in high dimensional data.

## References

- [1] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 9(18):509–517, 1975.
- [2] K. L. Clarkson. Nearest-neighbor searching and metric space dimensions. In *In Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*. MIT Press, 2006.
- [3] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry (Algorithms and Applications)*. Springer, 1998.
- [4] C. Elkan. Using the Triangle Inequality to Accelerate k-Means. In T. Fawcett and N. Mishra, editors, *International Conference on Machine Learning*, pages 147–153, Menlo Park, CA, 2003. AAAI Press.
- [5] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (AES) with linear preprocessing time and memory requirements. 15(1):9–17, 1994.
- [6] M. T. Orchard. A fast nearest-neighbor search algorithm. *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 4:2297—3000, 1991.
- [7] E. Vidal. An algorithm for finnding nearest neighbours in (approximately) constant average time. *Pattern Recognition Letters*, 4(3):145–157, 1986.