Introduction to ML

Machine learning is the field of study that gives computers the ability to learn without being explicitly programmed (Arthur Samuel, 1959).

Features are easily observable properties of data on which the model will try to make predictions.

Supervised Learning

Supervised learning: program is 'trained' on a given set of examples (with labels). It learns how to reach accurate conclusion when given new data.

For supervised learning to be successful, we need:

1. Representative training data
2. Sophisticated feature extraction
3. Sophisticated choice of machine learning algorithm

Supervised learning starts by representing:

- x_i: attributes / features of some data, usually D dimensional vectors of numbers (can be complex objects)
- y_i: class which some data belongs to

Then we take a training set that contains the right answers y_i:

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$$

With which we must learn a function f that maps from features x_i to class y_i:

$$f(x_i) = y_i$$

Classification is a supervised learning problem where we organise data in classes and determine the class of new data points.

$$y \in \{1, ..., C\}$$

Where C is the number of classes.

Classification is often not very clear at boundaries, so we should instead interpret the problem such that it returns a probability of being in each class given some input and set of training data.

$$p(y_i \mid x_i, \mathcal{D})$$

The **Maximum a Posteriori** (MAP) estimate is the most likely class label/mode of the distribution.

$$\hat{y} = \arg\max_{c=1}^{C} p(y = c, x_i, \mathcal{D})$$

Regression is a supervised learning problem where we fit functions to data and determine values of new data points, it is similar to classification but the output is continuous.

In **linear regression**, we assume output y is a linear combination of input x.

$$y(x) = \sum_{j=1}^{D} w_j x_j + \varepsilon$$

Where w_j make up a weight vector $\boldsymbol{w}$ and ε is the residual error.

A one-dimensional output, $\boldsymbol{x} = (x_0, x_1), x_0 = 1, x_1 = x$, produces $\overline{y}(\boldsymbol{x}) = w_0 + w_1 x_1$.

Unsupervised Learning

Unsupervised learning is where the program is given a bunch of unlabelled data and must discover patterns and relationships in them, common techniques include clustering and association rules.

Starting from a set of examples

$$\mathcal{D} = \{\boldsymbol{x}_i\}_{i=1}^{N}$$

We look for patterns/interesting structure in the data, producing modules of the form:

$$p(\boldsymbol{x}_j \mid \mathcal{D})$$

Reinforcement Learning

Reinforcement learning is where the program learns from consequences of its actions (reward/punishment) rather than being explicitly taught and selects actions on basis of past experience (exploitation) or by new choices (exploration).

Parametric Models

An ML technique is **parametric** if it makes assumptions about the structure of the data, they are generally computationally simpler but assumptions can lead to inaccurracy.

k-nearest neighbour (kNN) is a simple non-parametric classifier that looks at k points in the training set that are nearest to test input x.

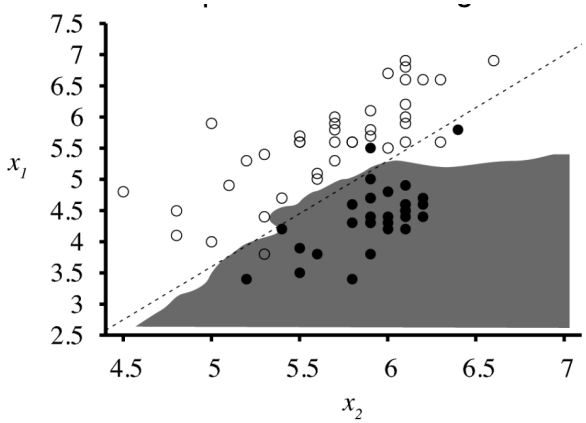


Figure 1: kNN Classifier
Filled area represents sample inference on the data.

We can be a bit more sophisticated and use k nearest points to estimate probability of class membership:

$$p(y = c|\boldsymbol{x}, \mathcal{D}, K) = \frac{1}{K} \sum_{i \in N_K(\boldsymbol{x}, \mathcal{D})} \mathcal{I}(y_i = c)$$

Where $N_K(\boldsymbol{x}, \mathcal{D})$ are indices of k nearest points to x in $\mathcal{D}$.
Where $\mathcal{I}(e) = \begin{cases} 1 \text{ if } e \text{ is true} \\ 0 \text{ if } e \text{ is false} \end{cases}$ is the indicator function.

Performance measurement

In supervised learning, we look at "can we predict y_i well given x_i?"

To compute the **misclassification rate** on the data we have examples to learn from:

$$\text{err}(F, \mathcal{D}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{I}(F(x_i) \neq y)$$

This rate gives us some idea on whether we have learnt well from training data but does not say anything about new data (generalisation error).

The **learning curve** is the % correct on a test set as a function of training set size.

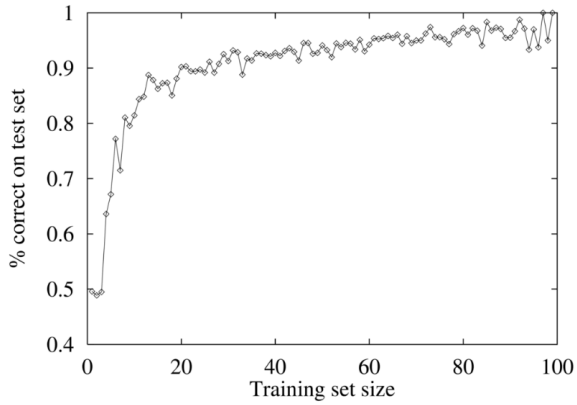


Figure 2: Learning Curve

Overfitting is when the model fits too much to the training data and fails to generalise unseen data.

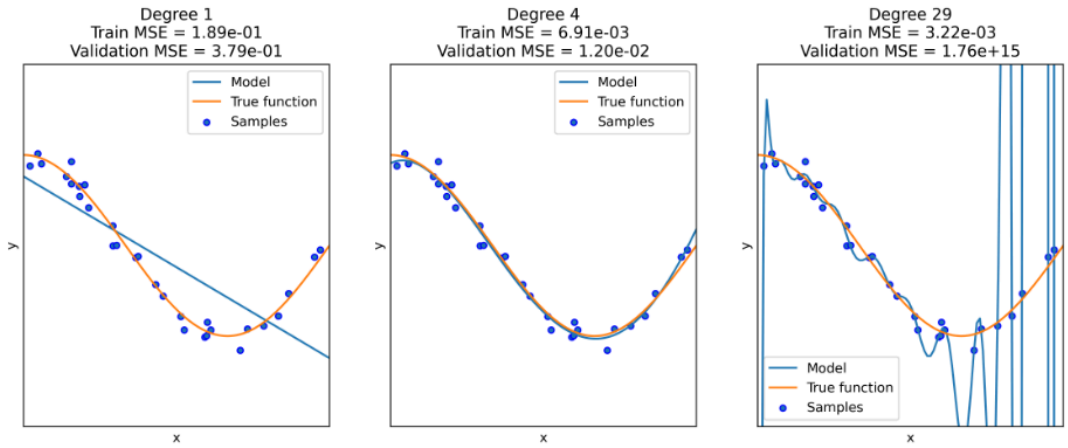


Figure 3: Evaluation samples for trained polynomial of varying degrees.

Accuracy is the proportion of correct predictions done by a classification algorithm over all predictions in the test set.

$$\text{accuracy} = \frac{\text{correct}}{\text{total}}$$

A **confusion matrix** is used to understand what is going on with our model by giving us an overview of the true/false positive/negative spread.

Predicted		**Actual**	
		Yes	No
Yes	true positive	false positive	
No	false negative	true negative	

Table 1: Sample confusion matrix

True Label		**Predicted Label**	
		0	1
0	12	1	
1	2	10	

Table 2: Sample confusion matrix

Precision is the proportion of true positives over true and false positives.

$$\frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

Sensitivity (recall) is the proportion of true positives over true positives and false negatives.

$$\frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

F1 Score is the harmonic mean between sensitivity and precision.

$$F_1 = \frac{2}{\mathrm{recall}^{-1} + \mathrm{precision}^{-1}}$$

$$F_\beta = \frac{1 + \beta^2}{\beta^2(\mathrm{recall}^{-1} + \mathrm{precision}^{-1})}$$

By plotting the **ROC Curve** (TP against FP) we find:

- area under curve is related to probability that classifier will correctly classify a random example
- dotted line is performance of random classifier (on average)
- closer we are to top left, the better the model is

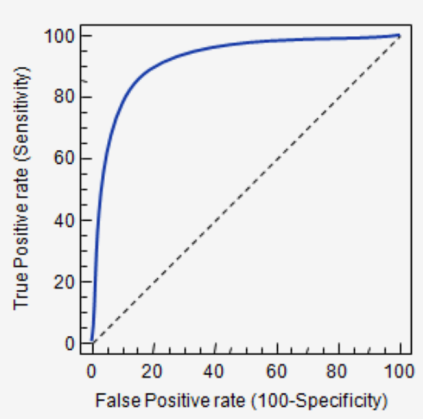


Figure 4: ROC Curve

Evaluating clusters

If we know what clusters should be, we can use classification methods specified before, otherwise:

- *external evaluation*: how well number of clusters serves a downstream task
- *internal evaluation*: establish how coherent the clusters are/how well separated they are

Splitting Datasets

Typically, we partition the training set into three subsets:

- training set (~80%)
- validation set (~10%)
  used for model selection/parameter tuning
- test set (~10%)
  always remains unseen, used to evaluate generalisation performance

K-fold cross-validation (CV) is used when we don't have enough training examples:

- Split the data into k equal subsets/folds to test on
- Train on $k - 1$ sets and test on remainder fold only
- Repeat k times (test on each fold only once)
- Computer misclassification error averaged over all k test folds

The final performance is the average of the performances for each fold. Average tests set score is a better estimate of error rate than a single score. Common values of k are 5 and 10, both giving likely accurate error estimates.

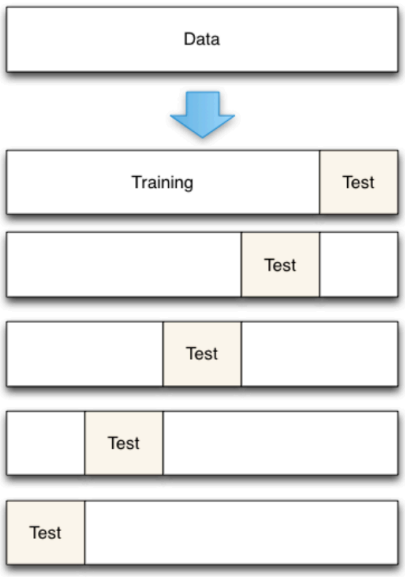


Figure 5: K-fold cross-validation (CV)

Inductive Learning

Inductive learning is learning from examples.

The **inductive learning hypothesis** says any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.

Decision Trees

A **decision tree** is a disjunction of conjunctions of constraints on feature values of instances/data points, used to approximate discrete-valued target functions. Learnt trees are effictively sets of if-then rules.

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T

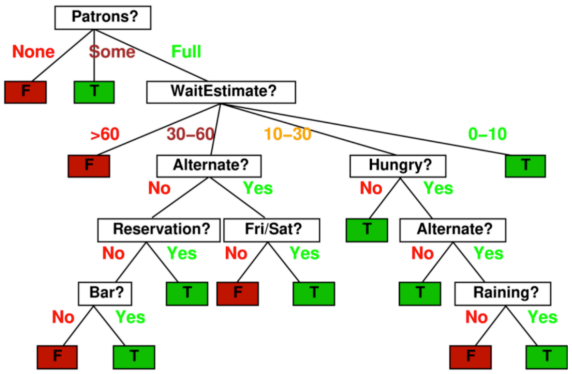


Figure 6: Example Decision Tree

Decision tree learning works as follows:

- Recursively choose "most significant" feature as root of (sub)tree

1. If all remaining examples are positive, we are done. Result is $\top$.
2. If all remaining examples are negative, we are done. Result is $\bot$.
3. If there are some positive and negative, choose best feature to split them.
4. If there are no examples left, there are no examples that fit the case.

   Return a default value, take a best guess based on the parent node, (**plurality classification**):
   - Pick majority, $\top$ if most examples at parent node are $\top$.
   - Pick randomly, weighted by ratio of examples at parent node.
   - Provide probability, based on ratio of examples at parent node.
5. If there are no features left, but there are still positive/negative examples, we can use plurality classification again. We may end up here due to noise or unobservability.

The DTL algorithm grows the tree to perfectly classify all examples however it may lead to overfitting if the set is too small or contains noise.

Given two decision trees d and d', d is said to overfit the training data if:
1. d has smaller error than d' on training data
2. d' has smaller error than d in all other cases

We can try to prevent overfitting by:
1. Stopping tree growth early
2. (Potentially) overfit the tree then **prune** it after learning

inputs: *examples*, *attributes*, *parent_examples*

if *examples* **is empty then**
  return pluralityClassification(*parent_examples*) ▷ use plurality classification to pick a default value from parent
else if [all *examples* have same *classification*] **then**
  return *classification* ▷ use the common classification
else if *attributes* **is empty then**
  return pluralityClassification(*examples*) ▷ use plurality classification on the remaining examples
else
  best ← mostImportantAttribute(*attributes*, *examples*) ▷ select the attribute
  tree ← [new decision tree rooted at *best*]
  for value v_i ← *best* **do**
    $examples_i \leftarrow \{e \mid e \in examples, e[best] = v_i\}$ ▷ select all examples where the attribute *best* is v_i
    remain ← *attributes* $-\{best\}$ ▷ remove the attribute from the list
    subtree ← **DTL** ($examples_i$, *remain*, *examples*) ▷ recursively build the tree
    [add branch to *tree* with label v_i and subtree *subtree*]
  end
  return *tree*
end

Algorithm 1: Decision Tree Learning Algorithm

Pruning strategies

In **reduced error pruning**:

- consider every branch in tree as candidate for pruning: removing it and making the root of the branch into a leaf
- use pluraity classification for examples at newly created leaf
- test new tree and keep branch pruned if new tree performs better on validation set
- repeat while possible to prune and improve performance

In **rule post-pruning**: (pruning with limited data)

1. Build decision tree
2. Create a rule set (one rule for each path from root to leaf)
3. Remove preconditions (feature tests) from rules if it improves their accuracy
4. Sort rules by accuracy and apply them in order / sequence when classifying examples

Broadening decision tree approach

To add additional support to our decision trees:

- Multi-valued features: when features have many values, information gain gives inappropriate estimate so we must convert these to boolean tests
- Continuous / integer input features: infinite sets of possible values, modify our approach to identify split points which give highest information gain (e.g. weight > 160)
- Continuous output values: when trying to predict continuous output values we need to create a regression tree that ends with a linear function

Entropy

Entropy is a measure of uncertainty.
Entropy in a probability distribution $\langle P_1, ..., P_n \rangle$ is

$$H(\langle P_1, ..., P_n \rangle) = \sum_{i=1}^{n} -P_i \log_2 P_i$$

Suppose we have a set S of p positive and n negative examples at the root.
Given that the chance of Class $= 1$ would be $p(\text{Class} = 1) = \frac{p}{p+n}$.
So we have a probability distribution over classes, and we can compute entropy of S:

$$H(S) = H\left(\left\langle \frac{p}{p+n}, \frac{n}{p+n} \right\rangle\right)$$

Information Gain

A feature A splits examples S into subsets S_i, each of these subsets is a new branch in the decision tree which have their own entropy. The **information gain** is a measure of how 'good' A is by the reduction in entropy of S and entropy of all of the S_i.

$$\text{Gain}(S, A) = H(S) - \sum_i \frac{|S_i|}{|S|} H(S_i)$$

Assuming S_i has p_i/n_i positive/negative examples, the entropy is $H(S_i) = H\left(\left\langle \frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i} \right\rangle\right)$.

We choose feature A with the biggest $\text{Gain}(S, \boldsymbol{A})$ given by:

$$\text{Gain}(S, A) = H(S) - \sum_i \frac{p_i + n_i}{p + n} H\left(\left\langle \frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i} \right\rangle\right)$$

Gain Ratio is a metric that incorporates another metric *split information* that penalises features with lots of values (it allows us to ask what the value of the feature is).

$$\text{SplitInformation}(S, A) = -\sum_i \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

$$\text{GainRatio}(S, A) = \frac{\text{Gain}(S, A)}{\text{SplitInformation}(S, A)}$$

The **Gini Ratio** (Gini Impurity) is a measure of statistical dispersion (measure of inequality/ probability of this classification mislabelling a randomly selected example).

$$G(S) = 1 - \sum_{i=1}^{c} p_i^2$$

Where c is the number of classes.
Where $p(\text{Class} = \top) = \frac{p}{p+n}$ is probability based on number of p positive, n negative examples.

Consider we have just two classes, we find $G(S) = 1 - \left(\left(\frac{p}{p+n}\right)^2 + \left(\frac{n}{p+n}\right)^2\right)$.
Suppose we have a feature A that splits S into subsets S_i, the Gini impurity of these sets is:

$$G(S, A) = \sum_i \frac{|S_i|}{|S_i|} G(S_i)$$

Linear Regression

Linear regression is learning a linear function of continuous inputs.

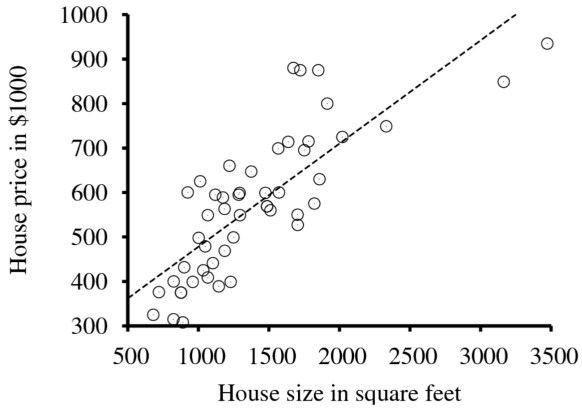


Figure 7: Linear Regression Line

The (univariate) equation is of the form $h_{\boldsymbol{w}}(x) = w_1 x + w_0$.

The **loss function** is defined using the squared loss function L_2:

$$\begin{aligned} J(h_w) = \text{Loss}(h_w) &= \sum_{j=1}^{N} L_2\big(y_j, h_w(x_j)\big) \\ &= \sum_{j=1}^{N} \big(y_j - h_w(x_j)\big)^2 \end{aligned}$$

Gradient Descent

To choose the best $\boldsymbol{w}$, we use the **gradient descent algorithm**:

- Start with some initial values for $w_0, w_1, \ldots$
- Keep changing values until cost / loss function $J(h_w)$ is reduced
  For each w_i, update with:

  $$w_i \leftarrow w_i \alpha \frac{\partial}{\partial w_i} \text{Loss}(w)$$

  where α is the learning rate and controls how much of a step we take downhill.
  - α too small may mean long convergence
  - α too large may mean we miss the minimum and not converge
- Continue until we reach a minimum

The slope at the minimum is 0 which means $\frac{\partial}{\partial w_i} \text{Loss}(w) = 0$

In **batch gradient descent**, we consider all training examples simulatenously, and at each step we update using:

$$w_0 \leftarrow w_0 + \alpha \sum_j \big(y_j - h_w(x_j)\big)$$

$$w_1 \leftarrow w_1 + \alpha \sum_j \big(y_j - h_w(x_j)\big) x_j$$

Batch gradient descent is guaranteed to converge but can be slow since we need to compute for all N examples at each step of the process.

In **stochastic gradient descent**, we update for each N examples in turn:

$$w_0 \leftarrow w_0 + \alpha(y - h_w(x))$$
$$w_1 \leftarrow w_1 + \alpha(y - h_w(x))x$$

Stochastic gradient descent is often much quicker than batch gradient descent but if the learning rate is constant, may not converge.

Multivariate linear regression

Suppose we have more variables $x_{j,1}, ..., x_{j,i}, ..., x_{j,n}$ and we are interested in a vector of weights w_i. First, we specify a dummy attribute to pair with w_0: $x_{j,0} = 1$.

Then $h_w(x_j)$ is the weighted sum of the variable values:

$$h_w(x_j) = \sum_{i=0}^{i=n} w_i x_{j,i}$$

Gradient descent for multivariate problems works as follows:

- Batch gradient descent

$$w_i \leftarrow w_i + \alpha \sum_j (y_j - h_w(x_j))x_{j,i}$$

- Stochastic gradient descent

$$w_i \leftarrow w_i + \alpha(y_j - h_w(x_j))x_{j,i}$$

To avoid overfitting, we add a **regularisation term** to the loss function:

$$\text{Loss}'(h) = \text{Loss}(h) + \lambda \cdot \text{Complexity}(h)$$

Where $\text{Complexity}(h_w) = \sum_i w_i^2$.
Where λ is the **regularisation parameter**: trade-off between fitting data and simple function.

Linear classifiers

We can turn a linear function into a classifier, the function defines the decision boundary that separates two classes.

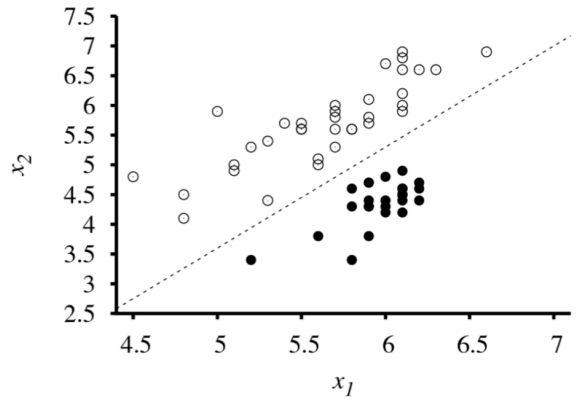


Figure 8: Seismic data due to earthquakes and nuclear explosions

$$h_w(x_j) = 1 \text{ if } \left(\sum_{i=0}^{i=n} w_i x_{j,i} > 0 \right) \text{ else } 0$$

Logistic Regression

The linear classifier always predicts 0 or 1, even for examples close to the boundary.

We can soften the threshold function (approximate hard threshold with continuous function),by using the **logistic function** as a threshold:

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

We also change the update function:

$$w_i \leftarrow w_i + \alpha(y - h_w(x)) \cdot h_w(x)(1 - h_w(x)) \cdot x_1$$

Logistic regression convergence is slower but behaves more predictably.

Ensemble Methods

An **ensemble** improves a classifier's error rate by taking N classifiers and using them on the same example by having them vote on the classification.

Boosting is an extension of ensembles where higher weighted examples are counted as more important during training (i.e. we put more copies into the training set).

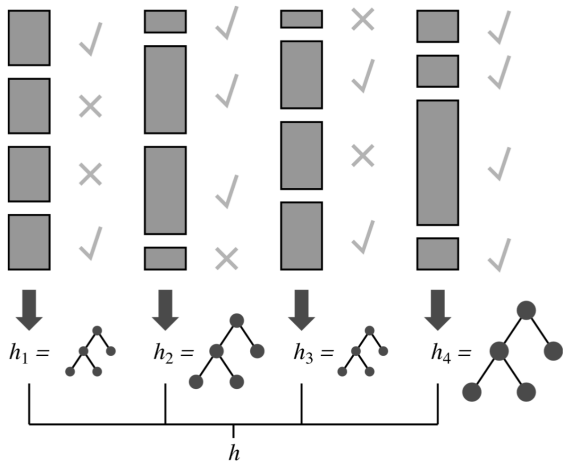


Figure 9: Ensemble Example

The **AdaBoost** algorithm is commonly used, it can take an initial classifier that is slighter better than random, and generate an ensemble that will perfectly classify the training set.

Bagging: given a training set $\mathcal{D}$, we create multiple training sets $\mathcal{D}_1, \ldots, \mathcal{D}_n$ which from each we learn a classifier. To classify an example, we combine the results of the ensemble. $\mathcal{D}_i$ are sampled from $\mathcal{D}$ with a uniform distribution (with replacement).

Random forests is bagging applied to decision trees so set of decision trees are learnt from bagged training sets and we also use the *random subspace method*.

The **random subspace method** for decision trees is the random selection of features for each tree. We select features at random with replacement (between trees). This is bagging at the feature level.

Probabilistic Models

We have already established that we can think of:

- classification: $p(y_i|\boldsymbol{x}_i, \mathcal{D})$ - probability example x_i is in class y_i given training data $\mathcal{D}$
- clustering: $p(\boldsymbol{x}_i|\mathcal{D})$ - probability of example x_i given data $\mathcal{D}$

There is no hard boundary between 'yes' and 'no':

- Incremental effect from each training example on estimated probability.
- Prior knowledge combined with observed data.
- Need initial knowledge of many probabilities.
- Computational cost can be high.

Bayes' Theorem

We are interested in some set of **hypotheses**, e.g. which class an example is in. Write H for all possible ones. Write h for a specific one. To find the most probable hypothesis, we calculate

$$p(h|\mathcal{D}) \qquad \forall h \in H$$

Bayes' Theorem states

$$p(h|\mathcal{D}) = \frac{p(\mathcal{D}|h)p(h)}{p(\mathcal{D})}$$

Where $p(h)$ indicates the probability of hypothesis h before we have any data. **(prior probability)**
Where $p(\mathcal{D})$ denotes the prior probability that the data will be observed. i.e. how probable we think the data is without any knowledge about which hypothesis holds.
Where $p(\mathcal{D}|h)$ is probability of observing data, given hypothesis is true.
Where $p(h|\mathcal{D})$ is probability of observing the hypothesis, given the data. **(posterior probability)**

Bayesian Learning

Bayes' Theorem is the basis for a learning algorithm that calculates the posterior of different h and outputs the one with highest probability.

Maximum a posteriori (h_{MAP}) hypothesis:

$$\begin{aligned} h_{\text{MAP}} &= \arg\max_{h \in H} p(h|\mathcal{D}) \\ &= \arg\max_{h \in H} \frac{p(\mathcal{D}|h)p(h)}{p(\mathcal{D})} \\ &= \arg\max_{h \in H} p(\mathcal{D}|h)p(h) \end{aligned}$$

We can remove $p(\mathcal{D})$ as it does not affect our result, we will still get the same hypothesis.

In some cases, we assume every hypothesis $h \in H$ is equally likely a priori.
$\therefore$ only consider the likelihood $p(\mathcal{D}|h)$, so we want the **maximum likelihood hypothesis** (h_{ML}):

$$h_{\text{ML}} = \arg\max_{h \in H} p(\mathcal{D}|h)$$

Naive Bayes Classifier

The **naive bayes classifier** is a practical approach to Bayesian learning which is simple (as such, particularly resistant to overfitting) but effective. It works by fitting the mould of $p(y_i|\boldsymbol{x}_i, \mathcal{D})$ by learning an approximation of the mapping from $\boldsymbol{x}$ to y: $y = f(\boldsymbol{x})$ where $f(\boldsymbol{x})$ takes on values from a finite set V (target value) and $\boldsymbol{x}_i$ is a tuple/vector of feature values, $\boldsymbol{x}_i = \langle a_1, a_2, ..., a_n \rangle$.

One approach to fitting the naive bayes classifier is to classify new instances by looking for:

$$V_{\text{MAP}} = \arg\max_{v_i \in V} p(v_i \mid a_1, a_2, ..., a_n)$$

rewrite using Bayes' Theorem:

$$= \arg\max_{v_i \in V} \frac{p(a_1, a_2, ..., a_n \mid v_i) p(v_i)}{p(a_1, a_2, ..., a_n)}$$

$$= \arg\max_{v_i \in V} p(a_1, a_2, ..., a_n \mid v_i) p(v_i)$$

We can estimate these values based on training data:
- $p(v_i)$ is based on number of times v_i appears in training data.
- $p(a_1, a_2, ..., a_n \mid v_i)$ is proportion of cases that have v_i, which also have $a_1, a_2, ..., a_n$.

However, the number of combinations is very large, so we would need to see them many times to get good estimates, and hence need a lot of data to train.

The **Naive Bayes Assumption** takes the assumptions that:
- position doesn't matter (if applicable)
- features are conditionally independent given the target v (strong independence assumption)

$$p(a_1, a_2, ..., a_n \mid v_i) = \prod_j p(a_j, v_i)$$

After calculating $p(v_i)$ and $p(a_j, v_i)$ (using Naive Bayes Assumption), we can make predictions about the v_i for a given $\boldsymbol{x}_k = \langle a_1, a_2, ..., a_n \rangle$:

$$V_{\text{NB}} = \arg\max_{v_i \in V} p(v_i) \prod_j p(a_j \mid v_i)$$

For small probabilities (e.g. 0.0053, 0.0206), we may find we run into underflow errors, we can avoid this by taking logs of values:

$$V_{\text{NB}} = \arg\max_{v_i \in V} \log P(v_i) + \sum_j \log P(a_j \mid v_i)$$

So far probabilities have been estimated by counting, e.g.

$$p(A = a \mid B = b) = \frac{n_c \text{ - total number of cases where A = a}}{n \text{ - total number of cases where B = b}}$$

This can lead to poor performance when n_c is close to zero which can happen for few samples, and hence lead to overfitting.

Instead we find the m-estimate:

$$\frac{n_c + m \cdot p}{n + m}$$

Where p is a prior estimate (usually $\frac{1}{k}$ if feature has k values).
Where m is a constant, equivalent sample size (weight given to prior).

[omitted slides 59-61: additional notes on Naive Bayes]

Gaussian Mixture Models

Given a dataset D of non-anomalous instances, is a new $\boldsymbol{x}'$ anomalous? We model $p(\boldsymbol{x})$ to detect anomalous cases: $p(\boldsymbol{x}') < \text{some threshold}$.

We could model each feature x as a Gaussian distribution with mean μ and variance σ^2, but often the data is much more complex, so we model the data in terms of a mixture of several components.

A **mixture model** is a combination of probability models.

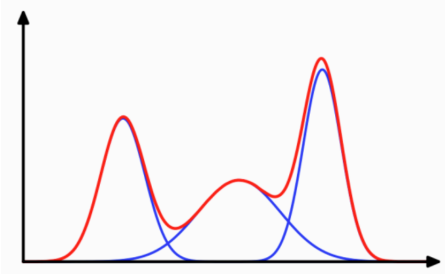


Figure 10: A univariate Gaussian mixture

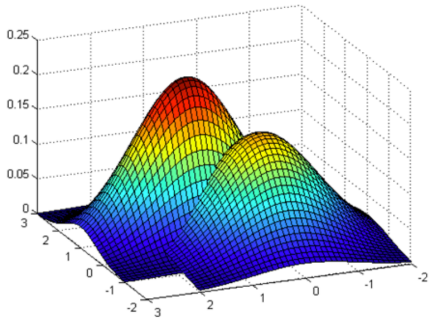


Figure 11: A multivariate Gaussian mixture

For the general case of mixture models, we mix K distributions (of any type) such that:

$$p(\boldsymbol{x}_i) = \sum_{k=1}^{K} \pi_k p_k(\boldsymbol{x}_i)$$

Where π_k are the mixing weights.

The most widely used mixture model is the **mixture of Gaussians**, each model is a multivariate Gaussian with mean μ_k and covariance Σ_k (matrix $n \times n$):

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

When all Gaussians are univariate, we instead have:

$$p(\boldsymbol{x}_i) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x}_i | \mu_k, \sigma_k^2)$$

Where σ_k^2 is the variance.

EM Algorithm

Assume $\mathcal{D}$ is generated by a mixture model, we would like to recover the Gaussians' parameters through estimation. Assume that each Gaussian has the same variance, so then we want to learn a hypothesis that describes the means of each distribution $h = \langle \mu_1, ..., \mu_k \rangle$ such that $p(\mathcal{D}|h)$ is maximum.

If we just had one Gaussian, the solution is just the sample mean: $\mu_{\text{ML}} = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{x}_i$.

The **Expectation Maximisation** (EM) algorithm is used to estimate means of k Gaussians, it can be used in any situation where we want to estimate parameters θ that describe a distribution, given only a portion of the data generated by the distribution

The EM algorithm works by repeating the following until convergence:

- **Expectation (E) Step**
  Given a current hypothesis $h_j = \langle \mu_1, ..., \mu_k \rangle$, estimate expected values of hidden variables z_{ij}.
- **Maximisation (M) Step**
  Then re-calculate maximum likelihood hypothesis given those values.

EM algorithm explained more generally:

- We have $\mathcal{D} = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_m\}$ as m independently drawn observations and $Z = \{\boldsymbol{z}_1, ..., \boldsymbol{z}_m\}$ as the corresponding unknown data which identifies which distribution $\boldsymbol{x}_n$ comes from. So then $Y = \mathcal{D} \cup Z$ is the full data.
- h is our current hypothesis about the value(s) of θ
  (θ is the set of parameters for model we are trying to learn)
  - For a set of k Gaussians with same variance:
    $$\theta = \langle \mu_1, ..., \mu_k \rangle$$
  - For a set of k Gaussians with different variance:
    $$\theta = \langle \mu_1, \sigma_1^2, ..., \mu_k, \sigma_k^2 \rangle$$
- EM searches for ML hypothesis h' by looking for h' that maximises $E[\ln p(Y|h')]$.
  Where $p(Y|h')$ is the likelihood of all the data given h'.
  If we maximise its log, we will maximise it.

  We do this by repeating until convergence:
  1. Expectation step: calculate $Q(h'|h)$ using current hypothesis h and observed data $\mathcal{D}$ to estimate probability over Y:
     $$Q(h'|h) \leftarrow E[\ln p(Y|h')|h, \mathcal{D}]$$
  2. Maximisation step: replace h by h' which maximises Q:
     $$h \leftarrow \arg\max_{h'} Q(h'|h)$$

Example: Two Gaussian case

Initialise EM by picking $h = \langle \mu_1, \mu_2 \rangle$ for arbitrary initial values.

Then repeat until convergence (change in parameters is small):

1. Calculate expected value $E\left[z_{ij}\right]$ of each hidden variable h_{ij} assuming $h = \langle \mu_1, \mu_2 \rangle$ is true.

   $E\left[z_{ij}\right]$ is probability that x_i was generated by j-th normal distribution:

   $$E\left[z_{ij}\right] = \frac{p(x = x_i | \mu = \mu_i)}{\sum_{n=1}^{2} p(x = x_i | \mu = \mu_n)} = \frac{e^{-\frac{1}{2\sigma^2}(x_i - \mu_i)^2}}{\sum_{n=1}^{2} e^{-\frac{1}{2\sigma^2}(x_i - \mu_n)^2}}$$

   Relative ratio of j-th Gaussian in i-th example.
   We just have to plug values of μ_1, μ_2, and x_i into above equation.

2. Calculate a new maximum likelihood hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$ assuming each z_{ij} takes its expected value $E\left[z_{ij}\right]$.

   Adjust values of the means to maximise likelihood:

   $$\mu_j \leftarrow \frac{\sum_{i=1}^{m} E\left[z_{ij}\right] x_i}{\sum_{i=1}^{m} E\left[z_{ij}\right]}$$

   The sample mean for u_j, weighted by expectation $E\left[z_{ij}\right]$.
   If expectation is small (the probaility that x_i belongs to the j-th distribution), then x_i contributes less to u_j (mean of j-th distribution).

K-means Clustering

Given n points $x_1, x_2, ..., x_n$, we want to partition them into k sets $S_1, S_2, ..., S_k$ such that the cost of the partition is minimised:

$$c(S_1, S_2, ..., S_k) = \sum_{i=1}^{n} \min_{j \in [k]} [\![|x_i - \mu_j\|_2^2]\!]$$

Where μ_i is the mid-point of each cluster, i.e. $\mu_i = \frac{1}{|S_i|} \sum_{j \in S_i} x_j$.
Where $\| \cdot \|_2^2$ is the squared L2 norm (squared Euclidian distance).

The K-means algorithm works as follows:

1. Pick random values for μ_k.
2. Repeat until convergence:
   1. Assign $\boldsymbol{x}_i$ to closest cluster centre:

$$z_i = \arg\min_z |\boldsymbol{x}_i - \boldsymbol{\mu}_k|$$

   2. Update each cluster centre as mean of points assigned:

$$\boldsymbol{\mu}_k = \frac{1}{|\{j : z_j = k\}|} \sum_{i \in \{j : z_j = k\}} \boldsymbol{x}_i$$

There are two types of K-means clustering:

- hard clustering: reports which cluster x_i is part of
- soft clustering: probability that x_i is in the cluster

Markov Chains

The first-order **Markov Assumption** is that transitions depend only on the current state.

$$P(w_t|w_{t-1}, w_{t-2}, ..., w_1) \approx P(w_t|w_{t-1})$$

The joint probability of a sequence of observations / events is:

$$P(w_1, w_2, ..., w_t) = \prod_{t=1}^{n} P(w_t|w_{t-1})$$

A **Markov Chain** is a stochastic process that embodies the Markov Assumption, can be seen as a probabilistic finite-state automaton. It models sequential problems.

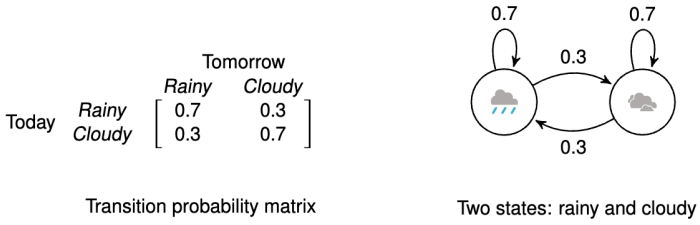


Figure 12: Example Markov Chain Definition

A **Hidden Markov Model** is a Markov Chain over hidden states where we only have access to observations at each time step and there is no 1:1 mapping between observations and hidden states.

$S_e = \{s_1, ..., s_N\}$ set of N emitting hidden states
$K = \{k_1, ..., k_M\}$ output alphabet of M observations
$O = O_1, ..., O_T$ sequence of T observations $\in K$
$X = X_1, ..., X_T$ sequence of T states $\in S_e$

s_0, s_1 special start/end states
k_0, k_f special start/end symbols

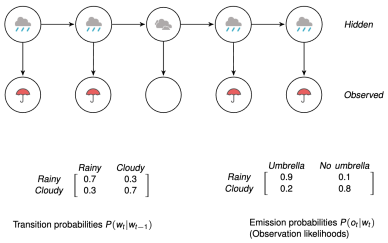
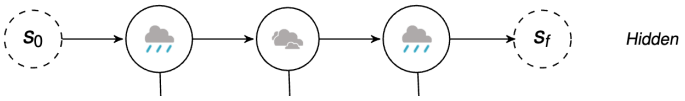


Figure 13: Example Hidden Markov Model

In practice, we also add special start/end states not associated with 'real' observations, s_0, s_f.



We also define matrices:

- A: state transition probability matrix of size $(N+2) \times (N+2)$
  a_{ij} is the probability of moving from state s_i to s_j

$$a_{ij} = P(x_t = s_j | X_{t-1} = s_i) \qquad \forall_i \sum_{j=0}^{N+1} a_{ij} = 1$$

- B: emission probability matrix of size $(M+2) \times (N+2)$
  $b_i(k_j)$ is the probability of emitting vocabulary item k_j from state s_j

$$b_i(k_j) = P(O_t = k_j | X_t = s_i)$$

As such, our HMM is defined by its parameters: $\boldsymbol{\mu} = (\boldsymbol{A}, \boldsymbol{B})$.

The **Output Independence** of a HMM is that the probability of an output observation only depends on the current state:

$$P(O_t|X_1, ..., X_t, ..., X_T, O_1, ..., O_t, ..., O_T) \approx P(O_t \mid X_t)$$

Fundamental tasks with HMMs:

- **Labelled Learning**: given parallel observation and state sequence O and X, learn the HMM parameters A and B
- *Unlabelled Learning*: given an observation sequence O (and only set of emitting states S_e), learn HMM parameters A and B
- *Likelihood*: given HMM $\mu = (A, B)$ and observation sequence O, determine likelihood $P(O|\mu)$.
- **Decoding**: given an observation sequence O and an HMM $\mu = (A, B)$, discover the best hidden state sequence X.

[omitted dice HMM example (slides 19-22)]

Viterbi Algorithm

Decoding a HMM involes finding the most likely hidden state sequence X that explains the observation O given the HMM parameters μ:

$$\begin{aligned}\hat{X} &= \arg\max_X P(X, O|\mu) \\ &= \arg\max_X P(O|X, \mu)P(X, \mu) \\ &= \arg\max_{X_1,\ldots,X_T} \prod_{t=1}^{T} P(O_t|X_t)P(X_t|X_{t-1})\end{aligned}$$

The search space of possible state sequences X is $O(N^T)$ which is too large for brute force. The Viterbi algorithm allows us to search the space of $O(T \times N^2)$ sequences instead!

The **Viterbi algorithm** is used to estimate the most likely underlying hidden series of states, given a series of observations and the parameters of the HMM, $\mu = (A, B)$, which are the state transition and emission probabilities.

The Viterbi algorithm works as follows:

- **Initialisation:** For each observation, add a node for each possible emission.
  For each emission, connect to the entire next set of emissions (from the next observation).
  Add a start/end nodes that link to the first/last set of emissions before/after respectively.
  Set the start node to have a value of 1.
- **Calculation:** Perform the following steps throughout the graph (for each hidden state):
  - Calculate the weight of the edges from current set of emissions to next by multiplying the *transition probability from current emission to the next* by the *emission proability of the next state*.
  - Calculate the value of the next emission nodes by taking the most contributing edge, i.e. look at all the edges, multiply the weight by the source node, and take the largest value. After selecting, keep track of the winning edge.
- **Backtracing:** Once the best edges have been selected, generate the sequence of hidden states (the emission nodes the edges pass-through).

[slides 60-61 omitted here since they seem to randomly repeat info about precision/recall]

Viterbi in action

Suppose we have the HMM parameters[1]:

$$\mu = (A, B), \quad A = \begin{array}{c|cccc} & \text{DT} & \text{NN} & \text{VB} & \text{(end)} \\ \hline \text{(start)} & 0.8 & 0.2 & 0 & - \\ \text{DT} & 0 & 0.9 & 0.1 & 0 \\ \text{NN} & 0 & 0.45 & 0.45 & 0.1 \\ \text{VB} & 0.45 & 0.45 & 0 & 0.1 \end{array}, \quad B = \begin{array}{c|cccc} & \text{THE} & \text{FANS} & \text{WATCH} & \text{RACE} \\ \hline \text{DT} & 0.2 & 0 & 0 & 0 \\ \text{NN} & 0 & 0.1 & 0.3 & 0.1 \\ \text{VB} & 0 & 0.2 & 0.15 & 0.3 \end{array}$$

We initialise a node for all possible emissions and the edges between them:

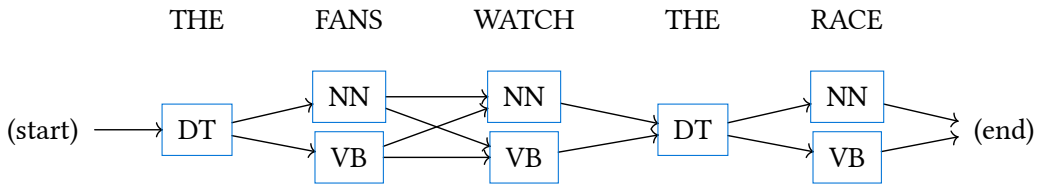


Figure 14: Our initial representation

Calculate all weights and find the best path through:

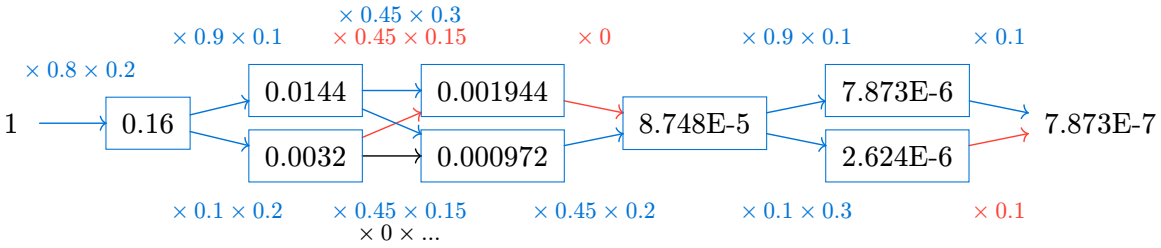


Figure 15: Final values

And now, we backtrace the most probable path:

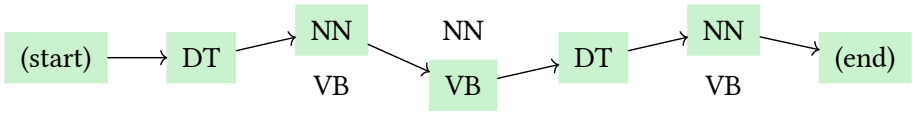


Figure 16: The backtraced path

Therefore our final result is:

$$[\text{DT}, \text{NN}, \text{VB}, \text{DT}, \text{NN}]$$

[1]Example adapted from ritvikmath's excellent lecture: **https://youtu.be/IqXdjdOgXPM**

A bunch of bullshit in slides that I don't think you actually need to know.

Note! The ψ helper variable, δ Viterbi probability and other shit is omitted.
In effect, it just overcomplicates the actual algorithm and is wholely unnecessary.

We can use Dynamic Programming to solve HMMs if:

- **Optimal substructure property**: optimal state sequence $X_1, ..., X_j, X_T$ contains inside it the sequence $X_1, ..., X_j$ which is also optimal.
- **Overlapping subsolutions property**: if both X_t and X_u are on the optimal path, with $u > t$, then the calculation of probability for being in state X_t is part of each of many calculations for being in the state X_u.

By intuition, we can save a lot of time because of the limited horizon the HMM.
For the first-order HMM, we only need to record one previous step, so we calculate the probability of reaching each state once for each time step then memoise this property in a table.
This reduces our effort to $O(N^2T)$.

Memoisation is done using a **trellis** which is equivalent to a Dynamic Programming table, which is $(N+2) \times (T+2)$ in size with states j as rows and time steps t as columns.

Each cell j, t in the trellis records the Viterbi probability $\delta_{j(t)}$: proability of the most liokely patyh that ends in state s_j at the time t.

$$\delta_j(t) = \max_{1 \leq i \leq N} \left[\delta_i(t-1) a_{ij} b_j(O_t)\right]$$

This is calculated by maximising over the best ways of going to s_j from each s_i.
Where a_{ij} is transition probability from s_i to s_j.
Where $b_j(O_t)$ is probability of emitting observation O_t from destination state s_j.

Baselines

Suppose you implemented Viterbi and measured its performance. How 'good' an algorithm's performance is depends on what alternatives are available.

A **baseline** is defined as a simple alternative that serves as a comparison.

A good baseline should:
- Represent considerably *less effort* than the proposed solution.
- Be *understandable* to other researchers.
- The baseline should still be *strong*: shouldn't be too easy.
  And have fair access to information.

For example for a HMM, we could use "random guess of hidden states" as the baseline.

To measure **observed system improvement**, we use a statistical test, e.g. higher F measure.

There may be variation in the data, some examples may make it easier for your system to score well, or allow other systems to score well. In this case, we perform *statistical significance testing*.

Statistical Significance

A procedure for **statistical significance testing**:

- Specify **null hypothesis**: two result sets come from the same distribution
  i.e. system 1 is (really) equally as good as system 2
- Choose a **significance level**: e.g. $\alpha = 0.01$ or 0.05
- Try to **reject** the null hypothesis with confidence $1 - \alpha$.
  By doing so, we show observed result is very unlikely to have occurred by chance.
  We can then report, e.g. "difference between system 1 and 2 is statistically significant at $\alpha = 0.05$".

The **sign test** is a non-parametric test (no assumptions about distribution of data) which uses a **binary event model** (we can use it to compare a new and baseline system). It is a binomial event model, where events can have binary outcomes.

If you run two models on the same dataset, you can either have:

- *Positive outcome*: System 1 beats system 2 on this example.
- *Negative outcome*: System 2 beats system 1 on this example.
- *Tie*: System 1 and 2 perform equally well.

Our null hypothesis in this case is that baseline and new system are equally good, i.e. probability of positive outcome equals probability of negative outcome, so the counts obey a binomial distribution with our mean equal to 0.5.

Assume probability of a negative outcome is q (here $q = 0.5$). The probability of observing $X = k$ negative events out of N is therefore:

$$P_q(X = k|N) = \binom{N}{k} q^k (1-q)^{N-k}$$

Or, at most k negative events:

$$P_q(X \leq k|N) = \sum_{i=0}^{k} \binom{N}{i} q^i (1-q)^{N-i}$$

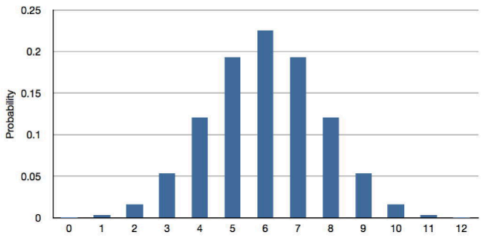


Figure 17: Binomial Distribution

If the probability of observing our events under the Null hypothesis is very small (smaller than significance level, α), we can safely reject the Null hypothesis.

The probability $P(X \leq k)$ gives us the probability p we are after.

A **one-tailed test** intends to test in one particular direction, e.g. system 1 is better than 2.

A **two-tailed test** intends to test for any statistical signifiance regardless of the direction.
This is given by $2P(X \leq k)$ because $B(N, 0.5)$ is symmetric.

When comparing two systems in classification tasks, ties are common. Disregarding ties can affect a study's statistical power. Instead, we can treat ties by adding 0.5 events to positive/negative sides .

(then round up at the end)

Kernel Machines

Primer

Suppose we have a binary classifier with three different solutions.

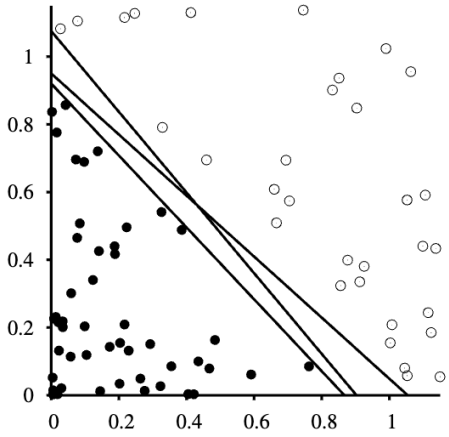


Figure 18: Three binary classifiers solutions

Typical classifiers minimise expected empirical loss on training data, while kernel machines attempt to minimise **generalisation loss**.

Support Vector Machines try to classify the data by finding the decision boundary that has the largest margin (*max margin separator*).

The **margin** is the width of the area bounded by the dashed lines.

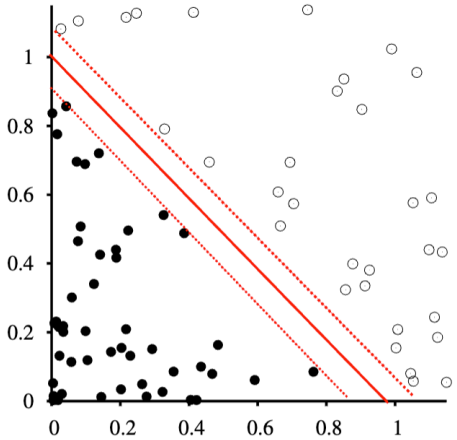


Figure 19: Example separator chosen by the SVM

Linear classifier has the form $y = \boldsymbol{w}^T \boldsymbol{x} + b$, we can change $\boldsymbol{w}$ to change the decision boundary/ separator. Finding the optimal decision boundary is a matter of finding the biggest margin.

The decision function is specified by a small subset of training samples, the support vectors (datapoints that lie on the two extremes of the boundary).

What we want is that:

$$\boldsymbol{w}^T \boldsymbol{x}_i + b \geq 0 \text{ when } y_i = +1 \text{ i.e. positive if element belongs to positive class}$$
$$\boldsymbol{w}^T \boldsymbol{x}_i + b < 0 \text{ when } y_i = -1 \text{ i.e. negative if element belongs to negative class}$$

In practice, SVMs include a *safety margin factor*:

$$\boldsymbol{w}^T \boldsymbol{x}_i + b \geq +1 \text{ when } y_i = +1$$
$$\boldsymbol{w}^T \boldsymbol{x}_i + b \leq -1 \text{ when } y_i = -1$$

Hence, we now define our margin using:

$$h1 : \boldsymbol{w}^T \boldsymbol{x} + b = +1$$
$$h1 : \boldsymbol{w}^T \boldsymbol{x} + b = -1$$

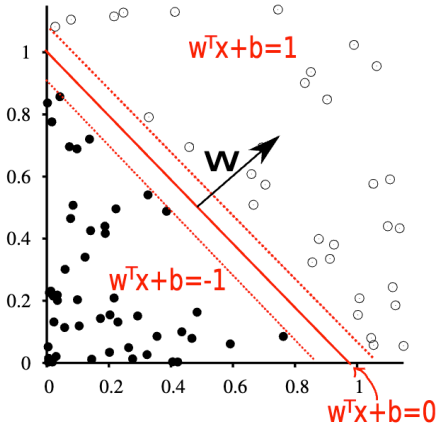


Figure 20: Example separator chosen by the SVM

The distance between h_1/h_2 and our decision boundary is $\frac{1}{\|w\|}$, where $\|w\|$ is the norm of vector $\boldsymbol{w}$:

$$\|w\| = \sqrt{w_1^2 + w_2^2}$$

The total distance between h_1 and h_2 is therefore $\frac{2}{\|w\|}$.

We can use gradient descent to search the space of $\boldsymbol{w}$ and look for the maximum margin separator that classifies examples correctly. We can define our optimisation problem as:

$$\max_{\boldsymbol{w},b} \frac{2}{\|w\|}$$
$$\text{subject to}$$
$$y_{i(\boldsymbol{w}^T \boldsymbol{x}_i + b)} \geq 1, \quad i = 1, ..., m$$

This constraint means that if a data point belongs to class $+1$ and our function returns something equal to or greater than $+1$, then our inequality is satisifed. Similarly, if we have a data point belonging to class -1 and our function returns something equal to or less than -1, then our inequality is satisfied.

However, this optimisation problem is difficult to solve.

Minimisation problem for SVMs

Instead, we define the minimisation problem:

$$\min_{\boldsymbol{w},b} \frac{1}{2} \|w\|^2$$

$$\text{subject to}$$

$$y_{i(\boldsymbol{w}^T \boldsymbol{x}_i + b)} \geq 1, \quad i = 1, ..., m$$

We now have a single global minimum, which is easy to solve!

Using Lagrangian to solve optimisation problems

We use Lagrange multipliers $\alpha_i \geq 0$ and a **Lagrangian**:

$$L(\boldsymbol{w}, b, \alpha) = \frac{1}{2} \|w\|^2 - \sum_{i=1}^{m} \alpha_i (y_i (\boldsymbol{w}^T \boldsymbol{x}_i + b) - 1)$$

Minimum is found by taking partial derivatives set to zero:

$$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_i \alpha_i y_i x_i$$

$$\frac{\partial L}{\partial b} = -\sum_i \alpha_i y_i$$

Substituting these and simplifying we find the **dual representation**, which allows us to find the optimal separator by solving:

$$\arg\max_{\alpha} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j (\boldsymbol{x}_i \cdot \boldsymbol{x}_j)$$

Where m is the number of training examples, $\alpha_i \geq 0$, and $\sum_{i=1}^{m} \alpha_i y_i = 0$.

The dot production $\boldsymbol{x}_i \cdot \boldsymbol{x}_j$ is defined:

$$\boldsymbol{x}_i \cdot \boldsymbol{x}_j = \sum_{k=1}^{n} x_{ik} \cdot x_{jk}$$

Where $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are feature vectors:

$$\boldsymbol{x}_i = \{x_{i1}, ..., x_{in}\}$$

$$\boldsymbol{x}_j = \{x_{j1}, ..., x_{jn}\}$$

For example, $[1, 2, 3] \cdot [4, 2, 3] = 1 \times 4 + 2 \times 2 + 3 \times 3 = 17$.

Finally, we can convert back to $\boldsymbol{w}$ by using:

$$\boldsymbol{w} = \sum_i \alpha_i y_i \boldsymbol{x}_i$$

For a specific $\boldsymbol{x}$, we want to know if it is $+1$ or -1, so our decision function can be written as:

$$h(\boldsymbol{w}) = \text{sign}\left(\sum_i \alpha_i y_i (\boldsymbol{x} \cdot \boldsymbol{x}_i) + b \right)$$

The Kernel Trick

Let's suppose we have some data we want to classify:

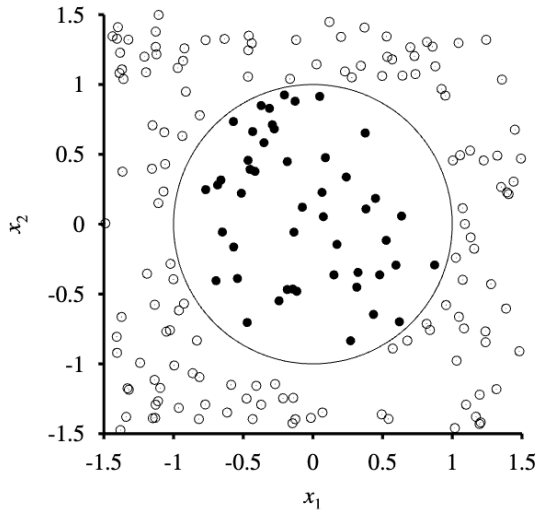


Figure 21: 2-dimensional data
Cannot fit linear boundary!

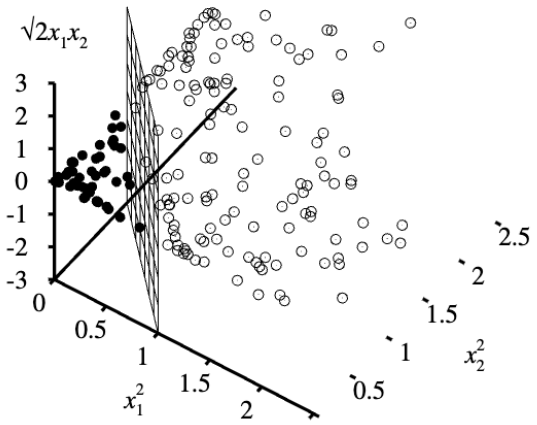


Figure 22: 3-dimensional data
Can fit linear boundary!

Data will always be linearly separable if we map into a space with enough dimensions.
In general, N data points will always be separable in $N - 1$ dimensions.

In a higher dimensional space, the separator will be a **hyperplane**.

Given some initial space:

$$\boldsymbol{x} = (x_1, x_2)$$

Map $\boldsymbol{x}$ to $F(\boldsymbol{x})$ such that:

$$f_1 = x_1^2$$
$$f_2 = x_2^2$$
$$f_3 = \left(\sqrt{2}\right)x_1x_2$$

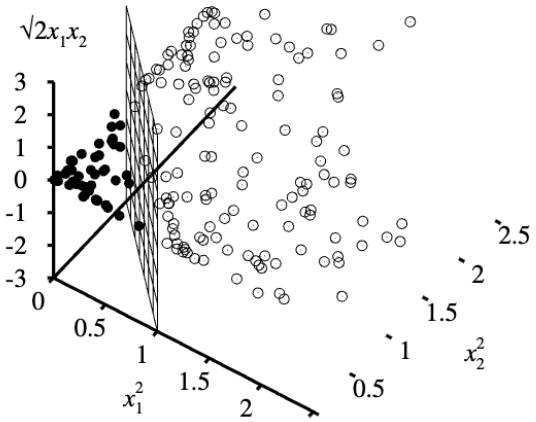


Figure 23: Projecting into x_1^2, x_2^2 space gives us the linear classifier seen in Figure 22

Kernels

A **kernel** is a mapping from input space $\boldsymbol{x}$ to the higher dimensional space.

Instead of

$$\arg\max_{\alpha} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (\boldsymbol{x}_i \cdot \boldsymbol{x}_j)$$

We write

$$\arg\max_{\alpha} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (F(\boldsymbol{x}_i) \cdot F(\boldsymbol{x}_j))$$

We can replace $\boldsymbol{x}$ by $F(\boldsymbol{x})$ in any ML algorithm.

The **dot-product of polynomials**:

$$F(\boldsymbol{u}) \cdot F(\boldsymbol{v}) = (\boldsymbol{u} \cdot \boldsymbol{v})^d$$

Where d is the mapping into high dimensional space by F.

Kernel Trick Example

Given input data $\boldsymbol{x} \in \mathbb{R}^2$, we project into 3D space with $F : \mathbb{R}^2 \to \mathbb{R}^3$

$$F(\boldsymbol{x}) = \left[\boldsymbol{x}_1^2, \boldsymbol{x}_2^2, \left(\sqrt{2}\right)\boldsymbol{x_1 x_2}\right]$$

If we want to compute inner product between the two, we could transform first:

$$\begin{aligned} F(\boldsymbol{x}) \cdot F(\boldsymbol{z}) &= \left[\boldsymbol{x}_1^2, \boldsymbol{x}_2^2, \left(\sqrt{2}\right)\boldsymbol{x_1 x_2}\right] \cdot \left[\boldsymbol{z}_1^2, \boldsymbol{z}_2^2, \left(\sqrt{2}\right)\boldsymbol{z_1 z_2}\right] \\ &= \ldots \end{aligned}$$

This is expensive as it is a computation in $\mathbb{R}^3$ space!

Instead, we use the kernel function $K(\boldsymbol{x}, \boldsymbol{z})$.

$$K(\boldsymbol{x}, \boldsymbol{z}) = F(\boldsymbol{x}) \cdot F(\boldsymbol{z})$$

e.g. for quadratic, $K(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x} \cdot \boldsymbol{z})^2$

The **kernel function** is the expression:

$$(\boldsymbol{x}_i \cdot \boldsymbol{x}_j)^2$$

usually written as:

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

For any $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, we can find the linear separator in higher dimension feature space by solving:

$$\arg\max_{\alpha} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

The resulting linear separators, when mapped to original input space, can correspond to arbitrary/ wiggly, non-linear decision boundaries.

Other kernel functions

Other kernels functions correspond to other feature spaces:

- **Polynomial kernel:**

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i \cdot \boldsymbol{x}_j)^d$$

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (1 + \boldsymbol{x}_i \cdot \boldsymbol{x}_j)^d$$

- **Gaussian kernel:**

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(-\frac{\| \boldsymbol{x}_i - \boldsymbol{x}_j \|_2^2}{2\sigma^2}\right)$$

- **Sigmoid:**

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \tanh(\mathcal{K}(\boldsymbol{x}_i \cdot \boldsymbol{x}_j) + \Theta)$$

  Where d is an integer and $\sigma, \mathcal{K}, \Theta \in \mathcal{R}$

- **Radial basis function kernel**: defines a spherical kernel with centre $\boldsymbol{x}_i$ and radius s.

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(-\frac{\| \boldsymbol{x}_i - \boldsymbol{x}_j \|^2}{2s^2}\right)$$

All of the given kernel functions have been found to give SV classifiers with similar accuracies and a similar set of support vectors.

Reasonable kernel functions

A kernel function $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is reasonable if it is **positive definite**.

That is, for some feature space $\boldsymbol{X}$:

$$\sum_{i,j=1}^{n} c_i, c_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) \geq 0$$

For any n, and any scalars c_i, c_j, and any $\boldsymbol{x}_i, \boldsymbol{x}_j \in \boldsymbol{X}$.

Other uses of kernels

There are two parts to learning, a method for finding optimal linear separators and the kernel trick.

We don't have to use them together:

- We can use the optimal linear classifier piece to learn linear classifiers. (linear kernel)
- We kernelise any learning algorithm where we use dot product of pairs of data points. (k-nearest neighbour, perceptron learning)

Dot product primer

The **dot product** is a measure of similarity.

$$a \cdot b = |a| \ |b| \cos\theta$$

In k-nearest neighbour, the dot product of one feature with another is a measure of 'nearness' between them. We classify by finding the k nearest examples and going with majority classification. So we could instead:

- Calculate similarity instead of distance between points.

Soft Margin Classifiers

For some problems, even in the high dimensional space of the kernel, there is no hyperplane that cleanly separates the training examples, this may be due to high noise. We can relax the constraints under which we look for a hyperplane.

A **soft margin classifier** allows the use of slack variables:

$$\xi_i \geq 0 \text{ for all } i = 1, ..., m$$

Now instead of looking for $y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \geq 1, \forall i$ in the original weight/feature space, we look for:

$$y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \geq 1 - \xi_i$$

Finding a soft margin classifier that generalises well is found by controlling the 'capacity' of the classifier, measured by $\|w\|$ and the amount of slack, measured by $\sum_{i=1}^{m} \xi_i$. This later will provide an upper bound on the number of training errors .

The *minimisation problem* for soft margin classifiers:

$$\tau(w, \xi) = \frac{1}{2} \|\boldsymbol{w}\|^2 + C \sum_{i=1}^{m} \xi_i$$

$$\text{subject to}$$

$$\begin{aligned} \xi_i &\geq 0 && \text{for all } i = 1, ..., m \\ y_i(\boldsymbol{w} \cdot x + b) &\geq 1 - \xi_i && \text{for all } i \end{aligned}$$

The constant $C > 0$ determines the amount of slack that is allowed and controls the trade-off between margin maximisation and minimising training errors. We typically have to search for the optimal C for a given problem.

After incorporating a kernel and rewriting in terms of Lagrange multipliers, we solve:

$$\arg\max_{\alpha} \sum_{i} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

$$\text{subject to}$$

$$\begin{aligned} 0 \leq \alpha_i &\leq C && \text{for all } i = 1, ..., m \\ \sum_{i=1}^{m} \alpha_i y_i &= 0 \end{aligned}$$

Only difference with separable case is upper bound on C on α_i, this limits the influence of individual data points. By intuition, this prevents outliers from having big effect on separator.

We end up with a hyperplane separator such that the support vectors ($\boldsymbol{x}_i$ which lie on margins) are the ones with slack variables $\xi_i = 0$.

Support Vector Regression

Regression is a generalisation of classification, rather than looking for output $y \in \{1, -1\}$, we are interested in output $y \in \mathcal{R}$, we do this by learning function $f(\boldsymbol{x})$ that approximates y.

Vapnik's ε-insensitive loss function is a way of quantifying the loss incurred by predicting $f(\boldsymbol{x})$ rather than y.

$$\begin{aligned} c(\boldsymbol{x}, y, f(\boldsymbol{x})) &= |y - f(\boldsymbol{x})|_\varepsilon \\ &= \max\{0, |y - f(\boldsymbol{x})| - \varepsilon\} \end{aligned}$$

In effect, we fit a 'tube' with radius ε to the data:

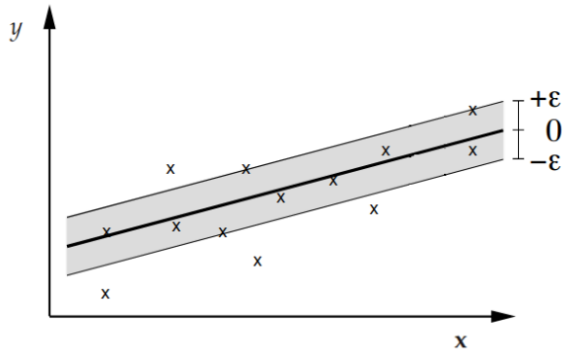


Figure 24: Support Vector Regression Example

To estimate a linear regression

$$f(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x} + b$$

we minimise:

$$\frac{1}{2} \|\boldsymbol{w}\|^2 + C \sum_{i=1}^{m} |y_i - f(\boldsymbol{x}_i)|_\varepsilon$$

We can then transform this into a constrained optimisation problem by intoducing slack variables (like with soft margin). We introduce ξ for when $f(\boldsymbol{x}_i) - y_i > \varepsilon$ and ξ^* for when $y_i - f(\boldsymbol{x}_i) > \varepsilon$. We write both ε as $\varepsilon^{(*)}$.

We can now minimise:

$$\tau\left(w, \xi^{(*)}\right) = \frac{1}{2} \|\boldsymbol{w}\|^2 + C \sum_{i=1}^{m} (\xi_i + \xi_i^*)$$

$$\text{subject to}$$

$$\begin{aligned} f(\boldsymbol{x}_i) - y_i &\leq \varepsilon + \xi_i \\ y_i - f(\boldsymbol{x}_i) &\leq \varepsilon + \xi_i^* \\ \xi_i, x_i^* &\geq 0 \text{ for all } i \end{aligned}$$

Constraints mean errors smaller than ε require non-zero ξ_i and x_i^* and so do not affect minimisation.

Using Lagrange multipliers, we end up with a regression estimate of:

$$f(\boldsymbol{x}) = \sum_{i=1}^{m} (\alpha_i^* - \alpha_i) K(\boldsymbol{x}_i, \boldsymbol{x}) + b$$

Gaussian Process Models

Gaussian process models combine SVM with probabilistic models:

1. Take basic likelihood calculation:

$$p(h|\mathcal{D}) = \frac{p(\mathcal{D}|h)p(h)}{p(\mathcal{D})}$$

2. Model $p(\mathcal{D}|h)$ as a Gaussian. (the linear model)
3. Now kernelise it.

Non-parametric Methods

Methods are **non-parametric** if they cannot be characterised by a bounded set of parameters. Instance-based methods like k-nearest neighbour are non-parametric. Grows without bound.

The **Minkowski distance** is a generalisation of Euclidean and Manhattan distances to p dimensions.

$$L^p(\boldsymbol{x}_j, \boldsymbol{x}_q) = \left(\sum_i |\boldsymbol{x}_{ji} - \boldsymbol{x}_{qi}|\right)^{\frac{1}{p}}$$

[omitted slides 70-80, seems to be yap about kNN and then vague application to kernels]

Neural Networks

A **neuron** is the building block of an artificial neural network, it is made up of:

- input edges (with weights)
- output edges (with weights)
- activiation level (function of inputs)

Weights can be positive or negative and change over time (learning).

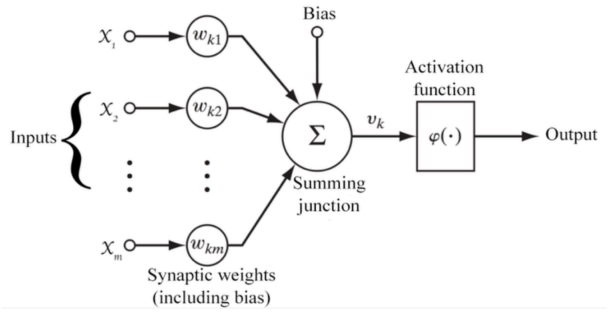


Figure 25: Neuron

The **input function** is the weighted sum of the activation levels of inputs.

The activation level is a non-linear **transfer function** g/φ of the *input function*:

$$\begin{aligned}\text{activation}_j &= g(s_j) \\ &= g\left(\sum_{i=1}^{N} w_{i,j}x_i\right)\end{aligned}$$

An **Artificial Neural Network** is a mathematical function that can produce approximations of real values, discrete values or vectors of values. It is built out of a densely interconnected set of units where each unit takes a number of real-valued inputs and produces a single real-valued output.

A **perceptron** is the simplest form of an ANN, consisting of a single neuron.
The input of the neuron is the weighted sum of the inputs plus the bias term and the output is some function of the input. A perceptron can compute **linearly-separable** functions.

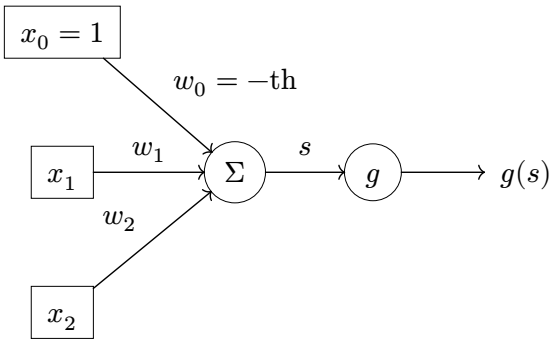


Figure 26: A Perceptron with inputs x_1, x_2, bias weight w_0, synaptic weights w_1, w_2, threshold th, and a transfer function g.

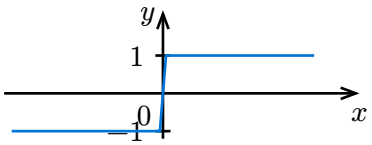
s is the dot product of the input and weight vectors plus the bias:

$$\begin{aligned}s &= w_0x_0 + w_1x_1 + w_2x_2 \\ &= -\text{th} + w_1x_1 + w_2x_2\end{aligned}$$

Transfer/threshold functions

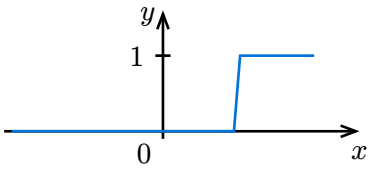
Sign function

$$g(s) = \begin{cases} 1, & s > 0 \\ -1, & s < 0 \end{cases}$$



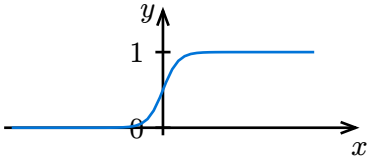
Step function

$$g(s) = \begin{cases} 1 & s \geq t \\ 0, & s < t \end{cases}$$



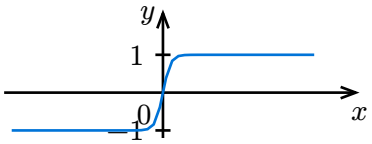
Sigmoid function

$$g(s) = \frac{1}{1 + e^{-s}}$$



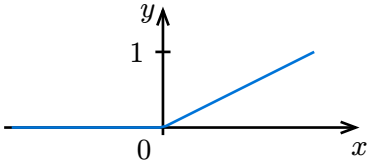
Hyperbolic tangent function

$$g(s) = \tanh(s)$$



ReLU

$$g(s) = \begin{cases} x \text{ if } x > 0 \\ 0 \text{ otherwise} \end{cases}$$



Error correction method

If the data is linearly separable, we can use the error correction method to find the linear boundary between classes.

For n inputs, the boundary is a hyperplane:

$$\begin{aligned} w_0 x_0 + w_1 x_1 + ... + w_n x_n &= 0 \\ \sum_{i=0}^{n} w_i x_i &= 0 \\ \boldsymbol{w} \cdot \boldsymbol{x} &= 0 \text{ where } x_0 = 1 \end{aligned}$$

We want to find the weights $w_0, ..., w_n$ for which members of one class have value 1 and members of the other class all have value -1.

We can train a perceptron by adjusting its weights:
- begin with random weights
- repeat until stopping condition:
  for each training example, update each weight by:

$$\begin{aligned} w_i &\leftarrow w_i + \Delta w_i \\ \Delta w_i &= \alpha(t - \boldsymbol{g}(s))x_i \end{aligned}$$

Where α is the learning rate, t is the target output for network, and $\boldsymbol{g}(s)$ is prediction produced.

By intuition:
- if $t = \boldsymbol{g}(s)$: no update is needed
- if $t > \boldsymbol{g}(s)$: perceptron outputs -1 when 1 is needed
  weights are adjusted to increase value of $\boldsymbol{w} \cdot \boldsymbol{x}$
- if $t < \boldsymbol{g}(s)$: perceptron outputs 1 when -1 is needed
  decrease weights aassociated with positive x_i

Theorem 0.1 (Convergence theorem): If the boundary between classes is linear, and α is small enough, error correction method will find it. The method will converge within a finite number of steps and classify all training examples correctly.

Theorem 1: Convergence theorem

If the boundary between classes is non-linear: the error correction method can fail to find the solution because no linear hyperplane can separately the data perfectly.

Delta rule

Consider an unthresholded perceptron (i.e. linear unit), we use an error function E to show how far the output is from our desired output:

$$\begin{aligned} \text{error} &= E(\boldsymbol{w}) \\ &= \frac{1}{2} \sum_{j=1}^{M} \left(t_j - s_j\right)^2 \end{aligned}$$

Where M is the number of instances in our dataset, t_j is the target evaluated for the j-th element, and s_j is the prediction evaluated for the j-th instance. *We want to minimise this error.*

The error is not defined in terms of $g(s)$ instead we opt to use s which means we are effectively training the unit without the transfer function by looking at the output of the weighted sum.

To minimise the error function, we use gradient descent, we move own the landscape into the valley where the error is small. The gradient points in the direction of greatest increase of function.

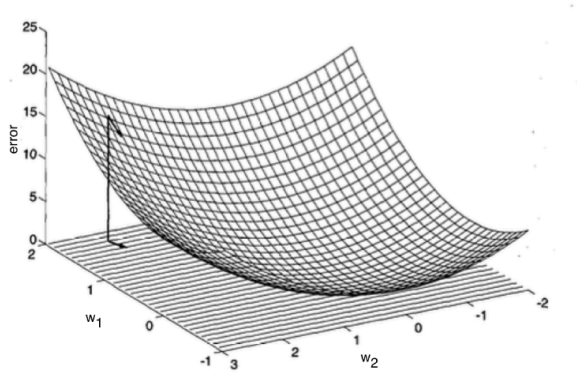


Figure 27: Solution space/landscape for a simple perceptron with two weights $\{w_1, w_2\}$

The gradient is computed by taking derivative of E with respect to each of the n elements in the weights vector $\boldsymbol{w}$:

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{j=1}^{M}\left(t_j - s_j\right)^2 \\ &= \dots \\ &= \sum_{j=1}^{M}(t_j - s_j)(-x_{i,j})\end{aligned}$$

The rule for updating each of the weights is:

$$w_i \leftarrow w_i - \alpha\frac{\partial E}{\partial w_i}$$

Substituting the derived gradient gives us the update rule:

$$w_i \leftarrow w_i + \alpha\sum_{j=1}^{M}(t_j - s_j)x_{i,j}$$

This is batch gradient descent (and equivalent to linear regression).
Presenting all training examples once is called an **epoch**.

We can also train by stochastic gradient descent:

$$w_i \leftarrow w_i + \alpha(t - s)x_i \qquad \forall w_i$$

Where (x, t) is each datapoint with output s.

With the delta rule, we learn with no transfer function so we need to add the transfer function back afterwards, moving from a regression to a classification. Bias w_0 sets midpoint of transfer function.

Generalised delta rule

The delta rule updates weights based on the error in the unthresholded linear combination of inputs. The **generalised delta** rule keeps the transfer function (but it must be differentiable).

Say we are using the sigmoid transfer function:

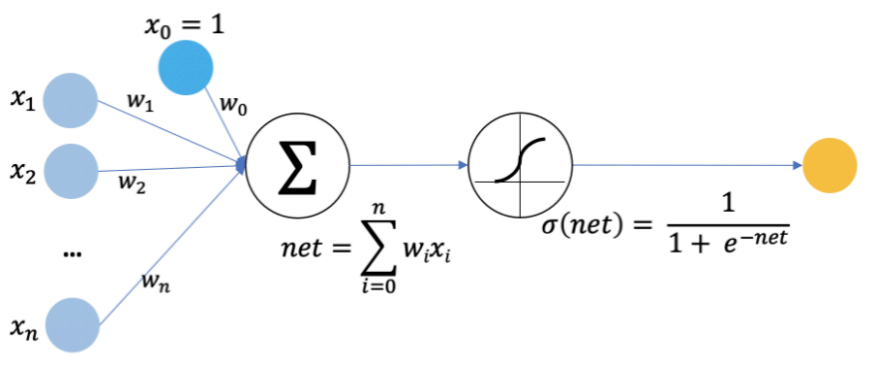


Figure 28: Perceptron using Sigmoid transfer function

We find the derivative of error with respect to each of n elements in the weights vector $\boldsymbol{w}$: (unlike the delta rule, the generalised derivation involves using the chain rule)

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\delta E}{\delta s}\frac{\delta s}{\delta w_i} \\
&= \frac{\delta E}{\delta s}\frac{\delta}{\delta w_i}\left(\sum_{i=0}^{n} w_i x_i\right) \\
&= \frac{\delta E}{\delta s} x_i \qquad\qquad \frac{\delta E}{\delta s} = \frac{\partial E}{\partial g(s)}\frac{\partial g(s)}{\partial s} \\
&= -(t - g(s))\frac{\partial g(s)}{\partial s} x_i
\end{aligned}$$

The sigmoid function is $g(s) = \frac{1}{1+e^{-s}}$ and $\frac{\partial g(s)}{\partial s} = g(s)(1 - g(s))$.

Therefore we find:

$$\frac{\partial E}{\partial w_i} = -(t - g(s))g(s)(1 - g(s))x_i$$

This gives us another rule for changing weights:

$$w_i \leftarrow w_i + \alpha(t - g(s))g(s)(1 - g(s))x_i$$

With the sigmoid, $g(s)(1 - g(s))$ varies in value from 0 to 1.
Recall: $w_i \leftarrow w_i - \alpha\frac{\partial E}{\partial w_i}$

Multi-layer networks

A **feedforward network** has a layered structure where each layer consists of units which receive their input from units from a layer directly below and send their output to units in a layer directly above. There are no connections within a layer.

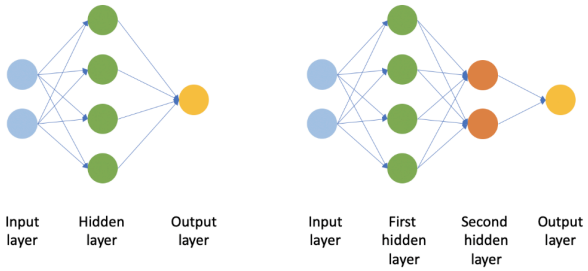


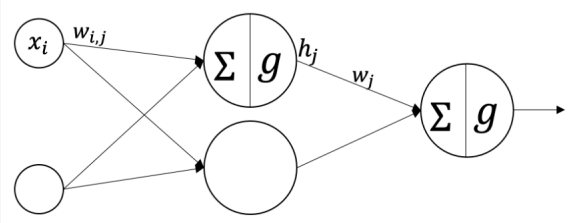
Figure 29: Example Feedforward Networks

A **multi-layer network** is a non-linear function between the inputs and outputs.
The hidden units produce hidden values and use non-linear threshold functions. Typically, we also use a non-linear threshold function at the end. Weights are applied at each layer which are updated during learning so that the output matches the input.

Theorem 0.2 (Universal function approximator): A feed-forward neural network with at least one hidden layer (and sufficient number of hidden units) can approximate any continous function.

Theorem 2: Universal function approximator

In a multi-layer network, each layer is evaluated in sequence.



1. N input nodes $\rightarrow$ H hidden nodes

$$\forall j \in \{1...h\} : h_j = g\left(b_i + \sum_{i=1}^{N} x_i w_{i,j}\right)$$

2. H hidden nodes $\rightarrow$ 1 output node

$$\text{out} = g\left(b_j + \sum_{v=1}^{H} h_j w_j\right)$$

where $g(\cdot)$ is the threshold function.

We may also include a bias at each layer.

Backpropagation

In **backpropagation**, we adjust weights while moving backwards through the network.

In essence, it works as follows:

- We know the error at any given output is $y - g(o)$.
- The total error for k outputs is therefore:

$$E = \frac{1}{2} \sum_k (y_k - o_k)^2$$

- We have a learning rate α.
- We want to find the derivative of E with respect to $w_{i,j}$ (the weights).
- Using this derivative, we use the update rule:

$$w_{i,j} \leftarrow w_{i,j} - \alpha \left(\frac{\partial E}{\partial w_{i,j}} \right) \text{ for all } w_{i,j}$$

- For any questions given, we assume $g(\cdot)$ (activation) is sigmoid function $\sigma(\cdot)$.
  Remember that: $\frac{\partial}{\partial x}\sigma(x) = x(1-x)$.
- **Note!** Commit this derivative to memory, it will come in useful:

$$\frac{\partial}{\partial o_k} \left(\frac{1}{2} \sum_k (y_k - o_k)^2 \right) = -(y_k - o_k)$$

$$\text{e.g. } \frac{\partial}{\partial o_1} \left(\frac{1}{2} \sum_k (y_k - o_k)^2 \right) = -(y_1 - o_1)$$

Given a simple feedforward neural network, with N inputs, one hidden layer with H nodes, and O outputs, we find that the update equation is:

$$w_{j,k} \leftarrow w_{j,k} + \alpha \delta_k h_j$$

Where h_j is the output of the node j that connects to node k.

Where δ_k where $k \in O$ is defined as:

$$\begin{aligned} \delta_{k \in O} &= (y_k - o_k) z_k (1 - z_k) \\ \text{where } y_k &\text{ is the target value} \\ o_k &\text{ is the predicted value} \\ z_k &\text{ is the input to } g(\cdot) \text{ for node } k \end{aligned}$$

Where δ_k where $k \in H$ is defined as:

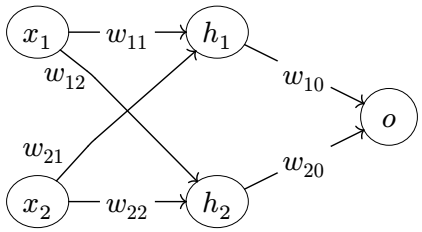
$$\delta_{k \in H} = z_k (1 - z_k) \sum_o^O w_{k,o} \delta_o$$

In the case that we are calcuating the edge from a bias j, we omit h_j:

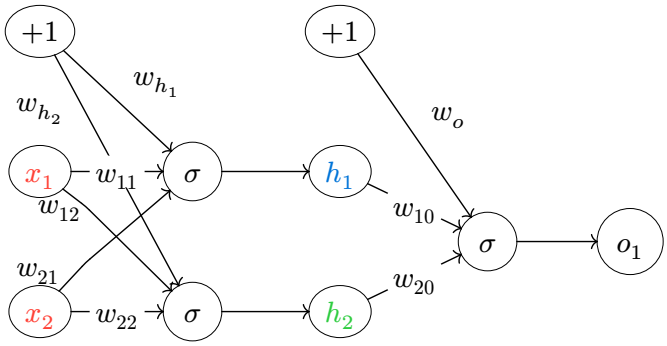
$$w_{j,k} \leftarrow w_{j,k} + \alpha \delta_k$$

Backpropagation Worked Derivations

1. Assume we have a network like so:



2. Normalise by adding activation func. and biases:



3. Derive all the formulae for generating a prediction from the network.

$$E = \frac{1}{2}\sum_k (y_k - o_k)^2 \qquad \text{... from earlier}$$

$$o_k = g(z_k) \qquad \text{output of } o_k \text{ (here } o_1\text{)}$$

$$z_o = w_o + w_{10}h_1 + w_{20}h_2 \qquad \text{input to activation function for } o_1$$

$$h_1 = g\big(z_{h_1}\big), \quad h_2 = g\big(z_{h_2}\big) \qquad \text{outputs of } h_1, h_1$$

$$z_{h_1} = w_{h_1} + w_{11}x_1 + w_{21}x_2 \qquad \text{input to activation function for } h_1$$

$$z_{h_2} = w_{h_2} + w_{12}x_1 + w_{22}x_2 \qquad \text{input to activation function for } h_2$$

4. Find the derivative of the error with respect to all the weights:

- Weight w_{10}

$$\frac{\partial E}{\partial w_{10}} = \frac{\partial E}{\partial o_1}\frac{\partial o_1}{\partial z_0}\frac{\partial z_0}{\partial w_{10}} \quad \text{notice how we follow the eqns. back to } w_{10} \text{ from } E!$$

$$= \frac{\partial}{\partial o_1}\left(\frac{1}{2}\sum_k (y_k - o_k)^2\right)\frac{\partial}{\partial z_0}(g(z_0))\frac{\partial}{\partial w_{10}}(w_0 + w_{10}h_1 + w_{20}h_2)$$

$$= -(y_1 - o_1)z_0(1 - z_0)h_1$$

$$= -\delta_o h_1 \quad \text{where } \delta_o = (y_1 - o_1)z_0(1 - z_0)$$

So we can now write our update rule as:

$$w_{10} \leftarrow w_{10} - \alpha\frac{\delta E}{\delta w_{10}}$$

$$\boldsymbol{w_{10} \leftarrow w_{10} + \alpha\delta_o h_1} \qquad w_{10} \leftarrow w_{10} + \alpha(y_1 - o_1)z_0(1 - z_0)h_1$$

- Weight w_{20} is derived in the same way.
- Weight w_o (the bias case)

$$\begin{aligned}\frac{\partial E}{\partial w_o} &= \frac{\partial E}{\partial o_1}\frac{\partial o_1}{\partial z_0}\frac{\partial z_0}{\partial w_0} \\ &= \frac{\partial}{\partial o_1}\left(\frac{1}{2}\sum_k (y_k - o_k)^2\right)\frac{\partial}{\partial z_0}(g(z_0))\frac{\partial}{\partial w_{10}}(w_0 + ...) \\ &= (y_1 - o_1)z_0(1 - z_0)\cdot 1 \\ &= -\delta_o\end{aligned}$$

  Therefore, for any biases, we simply omit the output (as it is just 1).

- Weight w_{11}

$$\frac{\partial E}{\partial w_{11}} = \frac{\partial E}{\partial o_1}\frac{\partial o_1}{\partial z_0}\frac{\partial z_0}{\partial h_1}\frac{\partial h_1}{\partial z_{h_1}}\frac{\partial z_{h_1}}{\partial w_{11}} \text{ employ the same strategy}$$

$$= \frac{\partial}{\partial o_1}\left(\frac{1}{2}\sum_k (y_k - o_k)^2\right)\frac{\partial}{\partial z_0}(g(z_0))\frac{\partial}{\partial h_1}(w_{10}h_1 + ...)\frac{\partial}{\partial z_{h_1}}\left(g\left(z_{h_1}\right)\right)\frac{\partial}{\partial w_{11}}(w_{11}x_1 + ...)$$

$$= -(y_1 - o_1)z_0(1 - z_0)w_{10}z_{h_1}\left(1 - z_{h_1}\right)x_1$$

$$= -\boldsymbol{\delta_o} w_{10}z_{h_1}\left(1 - z_{h_1}\right)x_1$$

$$= -\delta_{h_1}x_1 \text{ where } \delta_{h_1} = z_{h_1}\left(1 - z_{h_1}\right)w_{10}\delta_o$$

  So we can now write our update rule as:

$$w_{11} \leftarrow w_{11} - \alpha\frac{\partial E}{\partial w_{11}}$$

$$\boldsymbol{w_{11}} \leftarrow \boldsymbol{w_{11}} + \boldsymbol{\alpha\delta_{h_1}x_1} \qquad w_{11} \leftarrow w_{11} + \alpha\left(z_{h_1}\left(1 - z_{h_1}\right)w_{10}\delta_0\right)x_1$$

- Weights w_{12}, w_{21}, and w_{22} follow the same steps.

A (poor) implementation of this is available here[2].

[2] https://git.is.horse/insert/university/algorithms/-/blob/master/ml1/5-neural-networks.ipynb

Momentum

A common variation on the backpropagation algorithm is to introduce a **momentum term**, where we keep track of the weight updates in the previous iteration and make the weight update in the n-th iteration depend on the update in the $(n-1)$-th iteration.

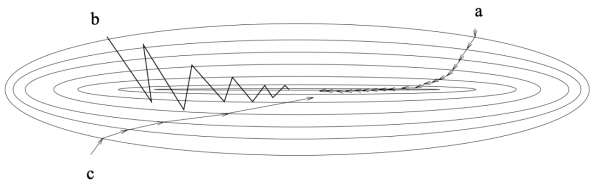
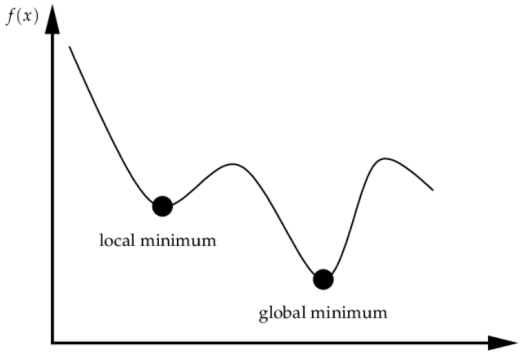


Figure 30: Descent in weight space.
a, b represent small/large learning rates
c represents large learning rate w/ momentum

This also allows us to 'roll over' local minima:



Instead of updating according to:

$$w_j \leftarrow w_j + \Delta_{w_j}$$
$$\Delta_{w_j} = \alpha h_j \Delta$$

We create a term for momentum, μ, and apply it as such:

$$\Delta_{w_j}(n) = \alpha h_j \Delta + \mu \Delta_{w_j}(n-1)$$
$$w_j \leftarrow w_j + \Delta_{w_j}(n)$$

Function approximation

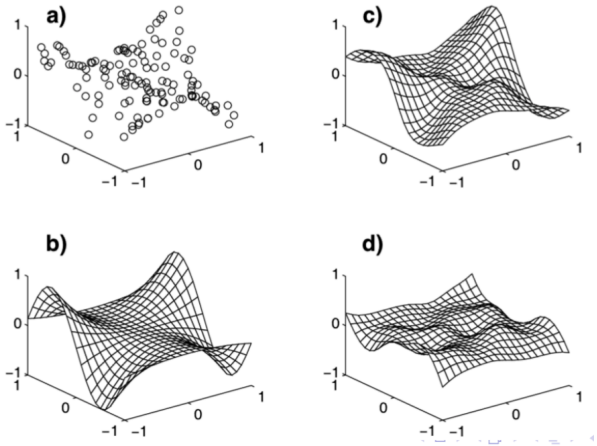


Figure 31: We can use feedforward networks to approximate functions.
a are samples of function b
c is approximation in network with error d

Generalisation

The **generalisation accuracy** is a measure of how well the network generalises to datasets outside of the training set. We can try to improve it by:

- Tweaking weights by small amounts each iteration.
- Introducing a validation dataset, keeping set of weights that performs best on it.

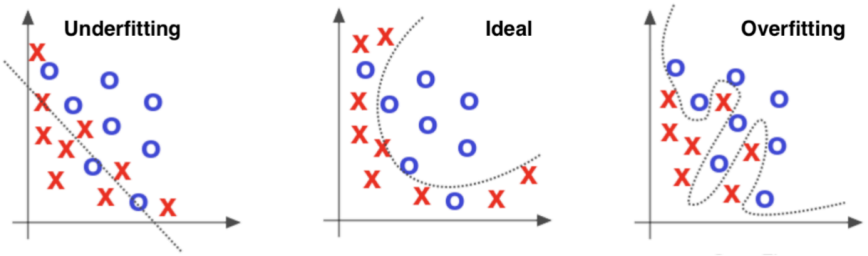
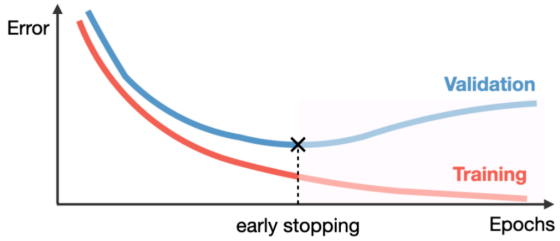


Figure 32: We can under or over fit a network.

We can try to prevent overfitting by:

- limiting number of hidden nodes/connections
- limiting training time, using validation set (early stop)
- decaying weights over time (decrease by some factor during each iteration)

Early stopping: evaluate on validation data after each epoch and stop training when performance of model has not improved for several epochs.



Deep Neural Networks

Deep neural networks are neural networks with many layers.

Often with very deep NNs, the gradient vanishes (or explodes) as it propagates backward. This means that: if gradients become very small, updates to weights become negligible. This can happen as the number of layers increase as the number of products (derived from chain rule; derivatives of each layer are multiplied together) required to update the weights increases.

We can combat this issue by using ReLU!

Convolutional Neural Networks

A **convolutional neural network** can capture spatial features from an image.

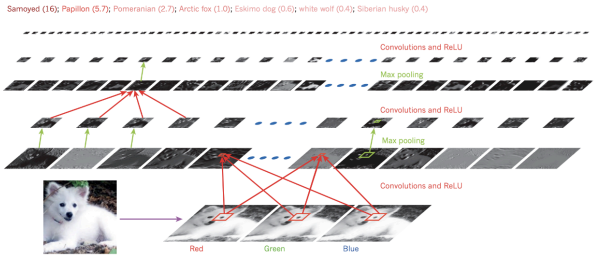


Figure 33: Example CNN for image classification

Recurrent Neural Networks

A **recurrent neural network** has recurrent connections on hidden states which ensures that sequential information is captured in the input data.

An RNN can be seen as a feedforward neural network by unrolling according to no. of time steps:

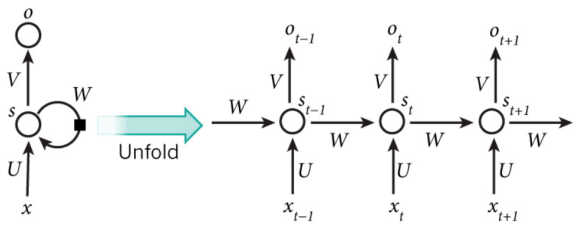


Figure 34: Unrolled RNN

Reinforcement Learning

Reinforcement learning is where the program learns from consequences of its actions (reward/punishment) rather than being explicitly taught and selects actions on basis of past experience (exploitation) or by new choices (exploration).

RL works with **evaluative feedback** which says how good the action was and not whether it was the best choice.

Associative learning: map inputs to outputs and learn best output for each input

n-armed bandits is a situation where we have n actions to choose from, reward probabilities are not known, and we want to maximise total reward in long term.

We take the approach:
- Choose repeatedly from one of n actions, this is a **play**.
- **Action values**: after each play a_t, we get reward r_t, where $E\langle r_t \mid a_t \rangle = Q^*(a)$.
  Distribution of r_t only depends on a_t.
- Explore a variety of options and exploit the best of them.

Assume you have *action value estimates*, $Q_t(a)$ which estimates $Q^*(a)$.
The **greedy** action at t is then:

$$a_t^* = \arg\max_a Q_t(a)$$

Picking a_t^* is **exploitation**, picking $a_t \neq a_t^*$ is **exploration**.

Action-value methods

In the **'sample-average' method**, we maintain a list of all rewards received and average them for each action:

- If by t-th play, action a has been chosen k_a times prior to t yielding $r_1, r_2, ..., r_{k_a}$:

$$Q_{t(a)} = \frac{r_1 + r_2 + ... + r_{k_a}}{k_a}$$

- If $k_a = 0$, let $Q_{t(a)} = 0$.
- Updating often enough, the estimate of the value of each action will converge to actual value.

$$\lim_{k_a \to \infty} Q_{t(a)} = Q^*(a)$$

This method requires a lot of memory!

In the **incremental implementation**: if Q_k is the average of the first k rewards and r_{k+1} is the $k+1$ th reward, then

$$\begin{aligned} Q_{k+1} &= \frac{1}{k+1} \sum_{i=1}^{k+1} r_i \\ &= ... \\ &= Q_k + \frac{1}{k+1}(r_{k+1} - Q_k) \end{aligned}$$

ε-greedy gives us a way to balance exploration-exploitation, we select some ε then select actions:

$$a_t = \begin{cases} a_t^* & \text{with probability } 1 - \varepsilon \\ \text{random action} & \text{with probability } \varepsilon \end{cases}$$

Softmax picks non-optimal actions based on their reward, commonly using Gibbs distribution.

Choose action a on the r-th play with probability:

$$\frac{e^{\frac{Q_{t(a)}}{\tau}}}{\sum_{b=1}^{n} e^{\frac{Q_{t(b)}}{\tau}}}$$

Where τ is the temperature.
As temperature tends to 0, action selection tends to greedy selection. Can vary over time.

Markov Decision Process

A **Markov decision process** (MDP) describes a problem with:

- a set of states $s \in S$ with initial state s_0
- a set of actions $A(s)$ in each state
- a transition model $P(s'|s, a)$
- a reward function $R(s)$

A **policy** π is a solution to the MDP, it represents mapping of states to actions to take.

The **optimal policy** π^* is the policy with the highest expected value, agent should prefer $\pi^*(s)$.

The **value iteration** algorithm allows us to compute the utility of each state:

1. Assign an arbitrary utility $U_0(s)$ to every state.
2. For every state, carry out the following update

   $$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s, a) U_i(s')$$

   Where $0 < \gamma \leq 1$ is the discount rate.
3. Repeat until convergence (values do not change).

Passive Learning

Suppose we do not know the transition model $P(s'|s,a)$ and reward function $R(s)$:
We need to learn the transition model and reward.

In **passive reinforcement learning**, the agent's policy is fixed.
The agent learns utility $U^\pi(s)$ by carrying out runs through environment, following some policy π.
That is, the agent does not make any choices about how to act while learning.

After learning the agent decides what to do by computing each step using one-step lookahead on expected value of actions, i.e. picking action a with greatest expected utility.

A **run** is a sequence of states and actions that continues until the agent reaches a terminal state.

Direct utility estimation is when the utility of a state is a function of all the states that are visited later. Each run gives us one or more samples for the reward from a state.

Suppose we have the run $(1,1)_{-0.04} \overset{\text{up}}{\rightarrow} (1,2)_{-0.04} \overset{\text{up}}{\rightarrow} (1,3)_{-0.04} \overset{\text{right}}{\rightarrow} (1,2)_{-0.04} \overset{\text{up}}{\rightarrow} (1,3)_{-0.04} \overset{\text{right}}{\rightarrow} (2,3)_{-0.04} \overset{\text{right}}{\rightarrow} (3,3)_{-0.04} \overset{\text{right}}{\rightarrow} (4,3)_{+1}$.

A sample reward for $(1,1)$ from the run is the sum of all the rewards to the terminal state, in this case the value is 0.72. The discount factor γ is set to 0.

As the agent moves, we can calculate a sample estimate of $P(s'|s,\pi(s))$:

$$P((1,2)|(1,1),\text{up}) = 1$$
$$P((1,2)|(1,3),\text{right}) = 0.5$$
$$P((2,3)|(1,3),\text{right}) = 0.5$$

Adaptive Dynamic Programming

In passive learning, we have π so we know what action we will carry out.
As such, the fixed policy version of the Bellman equation is:

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s))U^{\pi}(s')$$

This is a set of simultaneous equations that we can solve with an LP solver.
We update all utilities of states where we have experienced transitions.
Updated values are estimates, no better estimate but quicker convergence.

With passive learning, we can also use value iteration to update utilities for each state:

$$U_{i+1}(s) \leftarrow R(s) + \gamma\sigma_{s'} P(s'|s, \pi(s))U_i(s')$$
$$\text{until convergence}$$

The transition model is a maximum likelihood estimate (sample average) which we know are prone to overfitting. It may be dangerous as we might not have yet experienced the bad effects of an action, there is no way to make sure the action the reinforcement learner is picking does not have possible bad outcomes.

To combat this, we could:
- use better priors
- learn probability that a particular model is true
- ensure learner explores widely

Temporal Difference Learning

In **temporal difference learning**, we use observed transitions to adjust utilities of states, it is therefore **model-free**: there is no transition model which makes it easier to apply.

The temporal difference update for a transition from s to s' is:

$$U^\pi(s) \leftarrow U^\pi(S) + \alpha(R(s) + \gamma U^\pi(s') - U^\pi(s))$$

Where α is a learning rate. It is "temporal difference" as the update occurs between successive states.

Active Reinforcement Learning

In **active reinforcement learning**, learning agents must decide what to do while learning.
As opposed to a fixed policy in passive reinforcement learning.

With active learning, we can use value iteration to update utilities for each state:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$$

Usually, after value iteration we choose the action w/ highest expected utility, greedy agent. But this will not learn optimal policy. Could use ε-greedy to force the agent to learn but this can be slow.

What we can do instead is manipulate values to force the learner to explore:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} f\left(\sum_{s'} P(s'|s,a) U_{i(s')}, N(s,a)\right)$$

$$f(u,n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

Where $N(s,a)$ counts how many times we have done a in s. Where R^+ is the optimistic reward. Where N_e is the number of times we want the agent to be forced to pick an action in every state.

Q-learning

Q-learning is a model-free approach to active reinforcement learning.
learn Q-value of state/action pair as opposed to just an action.

$Q(s,a)$ denotes the value of doing a in s, so our utility becomes:

$$U(s) = \max_a Q(s,a)$$

The update rule is performed every time a is executed in s and takes the agent to s':

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

Where a' are all the actions we know about in s'.

SARSA

State-Action-Reward-State-Action (SARSA) is a variant of Q-learning where the update rule is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma Q(s',a') - Q(s,a))$$

Where a' is the actual action taken in s', that is, after every s, a, r, s', a' cycle.

Q-learning is off-policy, it does not care about policy being followed while SARSA is on-policy, it will only update with $Q(s',a')$ if a' is taken.

Function Approximation

So far, we have explicitly recorded utility functions/Q-values in a lookup table. This is fine for small numbers of states, but time to convergence slows rapidly as number of states grows. This creates a hard limit on the kind of problems we can solve.
(for ADP, time per iteration also increases)

Instead, we can use **function approximation**: have some function on state that provides a value, this is an approximation because we do not know what the real utility is.

Given a state s, one reasonable function is a **weighted linear function** of a set of features:

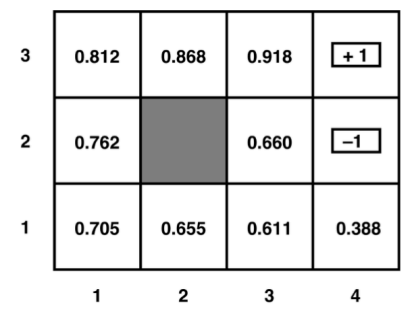
$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + ... + \theta_n f_n(s)$$

Then, we try to learn θ_i. Features are also called **basis functions**.
We can then learn, for example: 20 parameters rather than 10^{40} states.

This allows us to represent utilities efficiently, and allows us to generalise by allowing us to get utilities for states that we have not yet visited.

There is a trade-off between a function which is likely to represent the true utility function and how long it takes to learn.

Let's suppose we have our grid example again:



We could assume utility is related to x and y:

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

If $\theta = (0.5, 0.2, 0.1)$, then $\hat{U}_\theta(1, 1) = 0.8$.

To learn this, we can run passive learning trials and collect direct utility estimates of some states. Then learning θ becomes a supervised learning problem (it is just linaer regression).

Offline learning is where we do some learning and then we do some acting.

Online learning is where we learn while acting: as we learn new estimates of states, we adjust weights to reduce error for that state.

Evolutionary Algorithms

An **evolutionary algorithm** is a method of doing search.
We have a "population" of individuals where each "individual" is a representation of a solution to a problem. We build a representation and decide how to combine representations by doing selection and reproduction. A population of candidate solutions evolves towards better solutions by repeatedly selecting the fit individuals from the current population and modifying them to form the next generation population.

Genetic Algorithms

In general, genetic algorithms work by:

1. **Hypothesise a population** of candidate solutions to a particular problem.
2. **Evaluate** each member of population to decide how good that candiate is at solving the problem.
3. **Select** those members of the population that are the best.
4. **Reproduce** to obtain new members of the population.
5. **Repeat**, evaluate with new set of candidate solutions.

Genetic algorithms prove useful as:

- Good for situations where it is difficult to ascertain impact of particular partial solution.
- Easy to parallelise, each candidate solution can be d independently.
- Can adapt easily in dynamic environments.

1. Representation

In a Genetic Algorithm, the classic representation is a binary/bit string, with each element representing the presence/absence of a particular 'trait'.

The bit string represents the **genotype**.

The set of traits encoded is the **phenotype**.

It is possible to generate genotypes that have *multiple*, *uncertain*, or *invalid* phenotypes. We can employ different strategies to deal with invalid phenotypes such as preventing their generation or reporting the lowest fitness for invalid phenotypes.

2. Fitness

Fitness is a metric of success, it is a way of measuring how well a rule (or algorithm) performs. It is measured for each candidates solution in a population. Fitness is domain dependent.

3. Selection

Fitness proportionate selection is a domain-independent way to select members of the population:

$$\Pr(h_i) = \frac{\text{Fitness}(h_i)}{\sum_{j=1}^{N} \text{Fitness}(h_j)}$$

Where N is the size of the population, each h is the representation of a candidate solution, and $\text{Fitness}(h)$ is the fitness/score of that candidate. Also referred to as **roulette wheel selection**.

Other (domain-independent) methods of selection:

- **Tournament selection**: randomly select two candidates from population and keep the candidate with the higher fitness, with probability p proportional to each candidate's fitness (i.e. more fit, more likely to survive)
- **Rank selection**: sort candidates and rank them according to their fitness, keep candidates with rank higher than (or equal to) the proportion p

In the classic algorithm, a fixed **fitness threshold** is defined (domain-independent).
We segment candidates into 2 groups:

- P_{keep} - candidates whose fitness is above or equal to threshold
- P_{replcae} - candidates whose fitness is smaller than threshold

We then apply probabilistic selection to P_{keep} as described before.
The proportion between P_{keep} and P_{replace} is the exploitation-exploration ratio.

4. Reproduction

Genetic algorithms perform two reproduction operations (for population size N):

- **Crossover**: create two new offspring from two parent strings, by replaing some genes in one parent with genes from another and vice-versa

$$\text{Crossover rate} = r \text{ where } 0 \leq r \leq 1$$

$$\text{step 1: } \boxed{\begin{array}{c}(r \cdot N)\\ \text{crossed over}\end{array}} + \boxed{\begin{array}{c}((1-r) \cdot N)\\ \text{selected as is}\end{array}}$$

> **1-point crossover** is randomly selecting one bit in two strings, swapping bits from that point.
>
> **n-point crossover** is selecting n points ($\leq$ length of string)
> We find higher values of n are not useful, too much variation or not enough as $n \to$ length.
>
> A **crossover mask** may be used to indicate which bits should come from which parent.
>
> In **uniform crossover**, we create a mask by randomly selecting bits from each parent, thus making it equally likely to draw any bit from either parent.

- **Mutation**: produce small random changes to bitstring

$$\text{Mutation rate} = m \text{ where } 0 \leq m \leq 1$$

$$\text{step 2: } \boxed{\begin{array}{c}(m \cdot N)\\ \text{mutated}\end{array}} + \boxed{\begin{array}{c}((1-m) \cdot N)\\ \text{selected as is}\end{array}}$$

> **1-point mutation** is randomly selecting one bit in a string and flipping it.
>
> **n-point mutation** is selecting n points ($\leq$ length of string) and flipping them.

Trade-offs

Classic trade-off for genetic algorithms is between population size and number of generations: A smaller population generally implies more generations are required but they are evaluated more quickly. A larger population can mean less generations required but can take longer to evaluate.

We can choose a variety of exploration-exploitation ratios: A higher exploitation means set of candidates moves around the solution space more slowly/smoothly, we consider solutions in a much narrower search space/finer resolution. A higher exploration means set of candidates jumps around the solution space and solutions at more disparate points in search space are considered.

Variations

We do not necessarily need to do binary encoding, we could create a real-valued GA where we handle real numbers. Mutations could just randomly change value. Everything else works as with bit-string representations.

inputs: function assigning evaluation score from hypothesis **Fitness**, threshold specifying termination criterion **Fitness_threshold**, number of hypotheses to be included in population p, fraction of population to be replaced by Crossover at each step r, mutation rate m

outputs: winning hypothesis w

$P \leftarrow p$ [random hypothesis] ▷ initialise

for $h \leftarrow P$ **do**

  $\text{eval}_h \leftarrow$ **Fitness**(h)

end

while $\max_h \text{eval}_h <$ **Fitness_threshold do**

  $P_s \leftarrow \emptyset$

  [Probabilistically select $(1-r)p$ members of P to add to P_s] ▷
  1. Select

$$\Pr(h_i) = \frac{\text{Fitness}(h_i)}{\sum_{j=1}^{p} \text{Fitness}(h_j)}$$

  [Probabilistically select $\frac{r \cdot p}{2}$ pairs of hypotheses from P.
   For each pair, $\langle h_1, h_2 \rangle$, produce two offspring by applying Crossover operator.
   Add all offspring to P_s.] ▷
  2. Crossover

  [Choose m percent of members of P_s, with uniform probability.
   For each, invert one randomly selected bit in representation.] ▷
  3. Mutate

  $P \leftarrow P_s$ ▷
  4. Update

  for $h \leftarrow P$ **do** ▷
  5. Evaluate

    $\text{eval}_h \leftarrow$ **Fitness**(h)

  end

end

$w \leftarrow \text{argmax}_h \text{eval}_h$

Algorithm 2: Basic Genetic Algorithm

Genetic Programming

In **genetic programming**, instead of binary strings of fixed size, we use programs represented as trees for our genotype.

We can determine fitness by running the program:

$$\text{fitness} = \sin x + \sqrt{x^2 + y}$$

Then, we can use the value returned directly with the threshold to determine if the candidate program belongs to P_{keep} or P_{replace}.

1. Selection

Selection is performed the same way as in genetic algorithms, we can use:

- fitness proportionate selection
- rank selection
- tournament selection

2. Reproduction

We do not perform any mutation.

We can do crossover of trees where one (or more) nodes in the tree are selected from two parents, and then the sub-trees starting with those nodes are swapped.

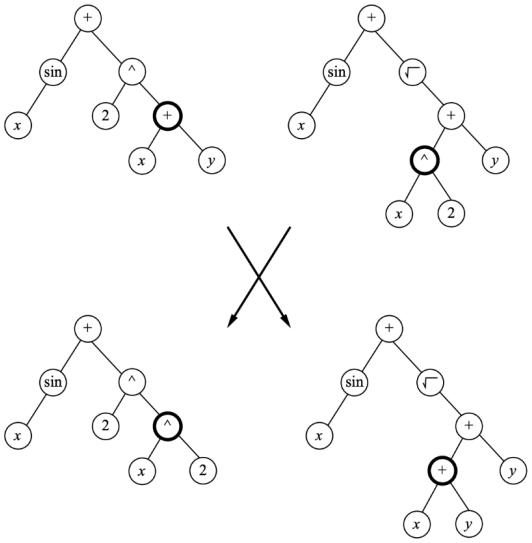


Figure 35: Example Tree Reproduction

$\sin x + 2^{x+y}$ $\sin x + \sqrt{x^2 + y}$

$\sin x + 2^{x^2}$ $\sin x + \sqrt{(x + y) + y}$

One significant issue with GP is **code bloat** where the size of evolving programs grows extremely large. We could try to solve this by limiting the maximum length of an s-expression that results from a crossover operation.

Co-evolutionary Algorithms

Traditionally, we have one evolving population. Success of candidate solutions in population is measured against a fixed fitness function and threshold.

With **co-evolution**, we have two evolving populations.
The fitness of a candidate solution is measured by comparing members of opposing populations.
This is, in effect, an arm's race for one population (as a whole) to surpass the other.

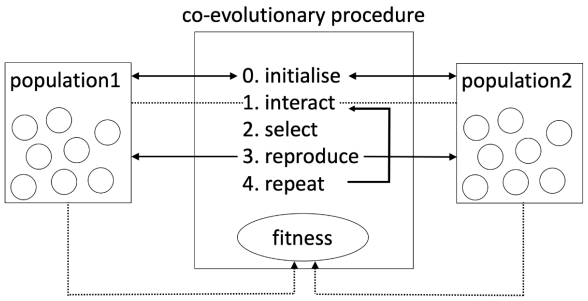


Figure 36: Co-evolutionary Learning

The advantage to co-evolution is when we have no easy way to define a fixed fitness function or one simply does not exist, in such a case we still have a way to compare populations.

Co-evolution presents some issues:

- **Collusion**: nobody tries to win, each keeps getting better but nobody gets the best
- **Mediocre Stable State** (MSS) / suboptimal equilibria: populations settle into portion of the landscape and do not change (coverge) but this might not be the most optimal.

Learning from Demonstration[3]

In **Learning from Demonstration**, we learn a mapping from state to actions (a policy) much like in reinforcement learning, but instead we supervise the learning from given examples (model-free learning). Policy is derived only from states encountered during learning, so the policy is *partial*.

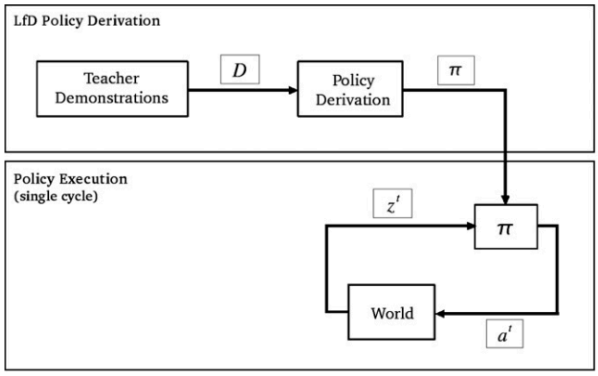


Figure 37: Control policy derivation and execution. Fig. 1 [1]

An **example** is a sequence of state-action pairs.

Formal definition of a problem in LfD:

- We have a set of states S, aset of actions A, and a transition function:

$$T(s'|s,a) : S \times A \times S \to [0,1]$$

  This is a probability distribution, but we can also think of it as a function.

- The learner has access to observed state Z.
- Policy selects actions based on observations of the world state:

$$\pi : Z \to A$$

  Similar to an MDP, but we have an indirection to Z, rather than $\pi : S \to A$.

- We also have a mapping between the actual and observed state:

$$M : S \to Z$$

  Given a fully observable state, then $M = I$ (identity function).

In LfD, we have an explicit teacher who provides demonstrations that the learner learns on. Formally, this is defined as:

Demonstration $d_j \in D$ such that

$$d_j = \left\{ \left(z_j^i, a_j^i \right) \right\}$$

Where $z_j^i \in Z$, $a_j^i \in A$, and $i = 0, \ldots, k_j$. This is the sequence of state-action pairs.

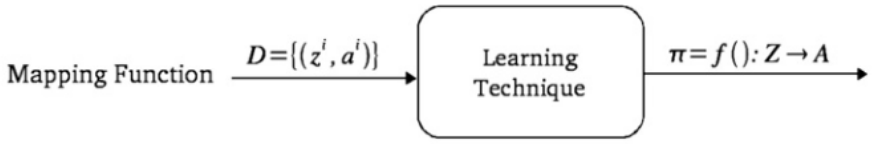
Design choices in LfD:

- **Choice of demonstrator**: who demonstrates, usually a human performing the task or human teleoperating the robot to perform the task.
- **Demonstration technique**: how are demonstrations gathered; in batch learning, policy is learnt once after all data is gathered; in interactive learning, policy is learnt incrementally.
- **Problem space representation**: should states/actions be continuous or discrete?
- **Policy derivation**: how should we derive the policy?

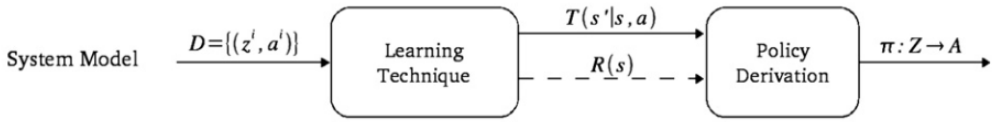
[3] This content is heavily derived from Argall [1].

Policy derivation strategies in LfD:

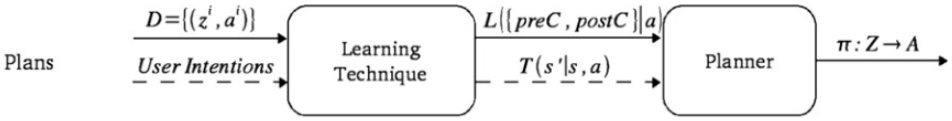
- **Mapping Function**: Demonstration data D is directly used to approximate the function mapping from observed state to actions.



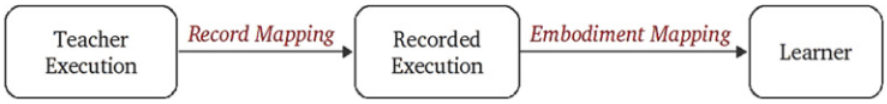
  This is in effect a classification problem.
- **System Model**: Demonstration data D is used to determine a transition model $T(s'|s,a)$ and possibly a reward function $R(s)$, from which a policy π is derived.



- **Plans**: Demonstration data D along with user intention is used to learn rules that associate a set of pre- and post- conditions with each action $L(\{\text{preC}, \text{postC}\}|a)$, then we can use a planner.



There are two parts to mapping a teacher execution to the learner.



Record Mapping
Teacher exec → recorded exec

Are the states/actions experienced by teacher recorded in the dataset?

- **Yes**: $I(z,a)$ (identity)
- **No**: $g_R(z,a)$ (record mapping)

Embodiment Mapping
Recorded exec → learner

Are the states/actions recorded in the dataset those that learner will observe/execute?

- **Yes**: $I(z,a)$ (identity)
- **No**: $g_E(z,a)$ (embodiment ...)

Strategies for gathering data:

- **Teleoperation**: human teacher operates the robot and robot's sensors record data.

$$g_R(Z,a) \equiv I(Z,a)$$

- **Shadowing**: robot operates itself but attempts to mimic human motions.

$$g_R(Z,a) \not\equiv I(Z,a)$$

- **Sensors on teacher**: direct data from demonstrator, but not the robot.

$$g_R(Z,a) \equiv I(Z,a)$$

- **External observation**: sensor data is external to demonstrator and robot.

$$g_R(Z,a) \not\equiv I(Z,a)$$

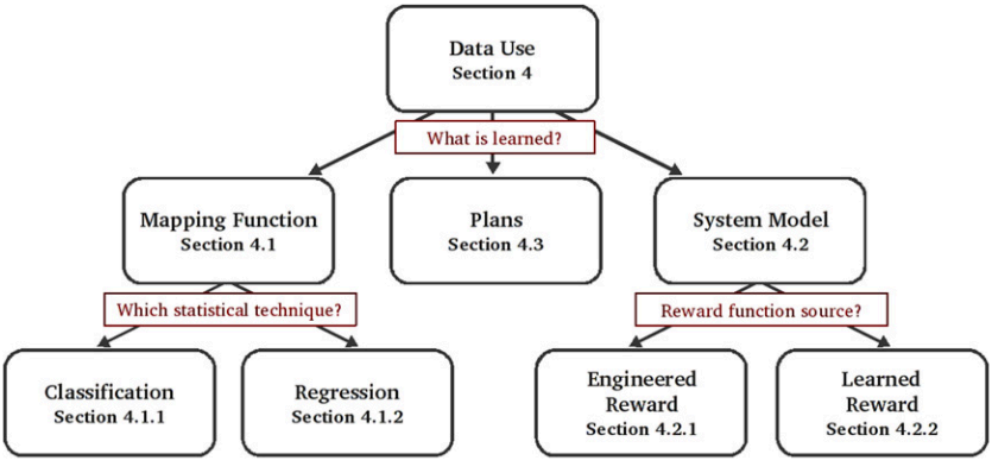


Figure 38: Categorization of approaches to learning a policy from demonstration data. Fig. 5 [1]

Approach to learning		**Policy strategy**
Mapping Function	Classification	Used for discrete actions. We need to pick a low-level action, can use: Gaussian Mixture Models, kNN, decision trees, or Bayes-net. Low-level actions can be put together in sequence using something like a Hidden Markov Model (HMM) or DBN. High-level actions are sequences, so we want to classify between different possible sequences.
	Regression	Used for continuous actions, can use any model that maps to continuous values such as kNN, locally weighted regression, or neural networks.
Plans		We represent desired behaviour as a plan which is a sequence of actions that lead from initial to goal state. The plan has pre-/post-conditions, world state $T(s'\|s,a)$, and action state. Does not handle non-determinism well!
System Model		Derive a policy $\pi : Z \to A$ from the model of the world $T(s'\|s,a)$ and reward $R(s)$. Maximise cumulative reward over time.
	Engineered Reward Function	"usual case": sparse rewards for particular states LfD concentrates on helping to locate rewards, reducing blind exploration. Demonstration can highlight rewards. Can use "ask for help" request when all actions roughly equal in value.
	Learned Reward Function	Try to learn a reward function that generalises from individual rewards. Function approximation in RL.

Confidence-Based Autonomy[4]

Confidence-Based Autonomy is an interactive algorithm for LfD policy learning, where initially the learner has no knowledge and acquires policy through observing demonstrations and practicing the task. If the learner encounters unfamiliar state, the learner can ask for more demonstrations.

The agent has a set of actions $a \in \mathcal{A}$, the underlying model used by the agent is a Markov Decision Process. We also have a sequence of demonstrations, (s_i, a_i), where $a_i \in A$ selected by teacher.

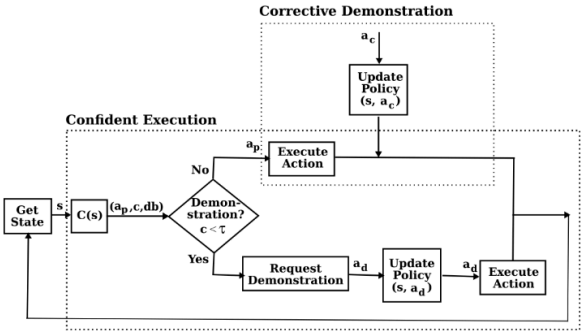


Figure 39: Confidence-Based Autonomy learning process. Fig. 1 [2]

The robot's behaviour policy is represented by a classifier:

$$C : s \rightarrow (a, c, db)$$

Where s is the state, $a \in \mathcal{A}$ is an action, c is an action selection confidence, and db is the decision boundary. This policy is trained on state vectors s_i (inputs) and actions a_i (labels).

The algorithm consists of two components:

- **Confident execution**: demonstrations triggered by learner's confidence
  If $c > \tau$, learner executes a_p, otherwise the learner receives a demonstration, a_d.

  The state space is divided into two regions: high confidence $\rightarrow$ autonomous execution and low confidence $\rightarrow$ demonstration. Confidence may be low if there is unfamiliar data (outliers) or it is ambiguous (overlapping regions).

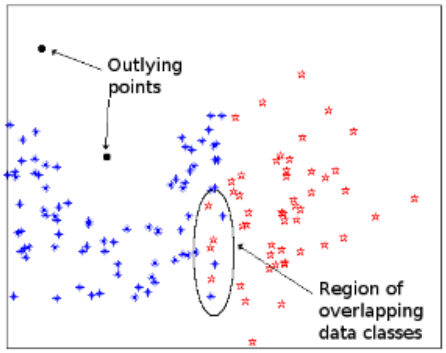


Figure 40: Chernova Fig. 2 [2]

- **Corrective demonstration**: demonstrations triggered by teacher correcting errors
  Teacher may decide to provide corrective demonstration, a_c.

Our goal is for the learner to generalise from demonstrations and learn a policy.

[4] This content is heavily derived from Chernova [2].

There are two evaluation criteria used to decide between autonomous execution and demonstration:

$$\text{if } (c > \tau_{\text{conf}}) \land (d < \tau_{\text{dist}}) \ \{\text{autonomous}\} \text{ else } \{\text{demonstration}\}$$

Initially, we set $\tau_{\text{dist}} = 0$ and $\tau_{\text{conf}} = \infty$.

- τ_{dist} is set to $3 \times$ average nearest-neighbour distance across dataset of demonstrations. Euclidean distance from current state to most similar training data point.
- τ_{conf} is more complicated to compute and a different threshold is computed for each decision boundary.

  A **classified point** is (o, a, a_m, c) where o is the original observation, a is the demonstrated action, a_m is the model-selected action, and c is the model action confidence.

  Hence, the set of all points mistakenly classified by decision boundary i is given by:

  $$M_i = \{(o, a_i, a_m, c) \mid a_m \neq a_i\}$$

  Finally, the **confidence threshold** is the average confidence of misclassified points:

  $$\tau_{\text{conf}_i} = \frac{\sum_{j=1}^{M_i} c}{|M_i|}$$

  If $M_i = 0$, then there were no classification mistakes.

Ethics of Machine Learning

Weapons of Math Destruction: concept that ML models have the capacity to create great damage to society through various avenues

- **Opacity**: black box models cannot be easily interrogated
- **Scale**: effects of model can extend beyond decision it makes
- **Damage**: model can cause harm to people, especially members of vulnerable groups

[two case studies omitted]

Bibliography

[1] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and Autonomous Systems*, vol. 57, no. 5, pp. 469–483, 2009, doi: https://doi.org/10.1016/j.robot.2008.10.024.

[2] S. Chernova and M. Veloso, "Interactive Policy Learning through Confidence-Based Autonomy," *Journal of Artificial Intelligence Research*, vol. 34, pp. 1–25, Jan. 2009, doi: 10.1613/jair.2584.