Machine Learning

Andrea Sanchietti

2022/2023

Contents

Chapter 1

Classification Evaluation

1.1 Classification Evaluation

1.1.1 Performance evaluation

Performance evaluation in classification is based on *accuracy* and *error rate*.

Consider a typical classification problem:

$$f : X \rightarrow Y \tag{1.1}$$

$\mathcal{D}$: probability distribution over X
$\mathcal{S}$: sample of n instances drawn from X
Consider a hypothesis h, solution of a learning algorithm obtained from S. What is the best estimate of the accuracy of h over future instances drawn from the same distribution? What is the probable error in this accuracy estimate?

1.1.1.1 true error and sample error

true error of hypothesis h with respect to target function f and distribution D is the probability that h will misclassify an instance drawn at random according to D

$$error_{\mathcal{D}}(h) \equiv \prod_{x \in D} [\sigma(f(x) \neq h(x))] \tag{1.2}$$

The sample error of h with respect to target function f and data sample S is the proportion of examples h misclassifies

$$error_{\mathcal{S}}(h) \equiv \frac{1}{n} \sum_{x \in S} \sigma(f(x) \neq h(x)) \tag{1.3}$$

where $\sigma(f(x) \neq h(x))$ is equal to 1 if $f(x) \neq h(x)$ and 0 otherwise.

Note: $accuracy(h) = 1 - error(h)$

The true error cannot be computed, the sample error is computed only on a small data sample.

If $accuracyS(h)$ is very high, but $accuracyD(h)$ is poor, then our system would not be very useful.

1.1.1.2 Estimating Bias

$$bias = E[error_S(h)] - error_D(h) \tag{1.4}$$

1.1.1.3 Confidence interval

with approximate N% probabiliry, $error : D(h)$ lies in interval

$$error_S(h) + -z_N \sqrt{\frac{error_S(h)(1 - error_S(h))}{n}} \tag{1.5}$$

where

$N\%$:	50%	68%	80%	90%	95%	98%	99%
z_N:	0.67	1.00	1.28	1.64	1.96	2.33	2.58

Figure 1.1: Value of z_N based on the $N\%$ chosen

1.1.1.4 K-Fold Cross Validation

Partition data set D into k disjoint sets $S_1, S_2, ..., S_k(|S_i| > 30)$ and then use one set as test set while the remaining sets as training set

The error is

$$error_{L,D} \equiv \frac{1}{k} \sum_{i=1}^{k} \sigma_i \tag{1.6}$$

note: $accuracy_{L,D} = 1 - error_{L,D}$

1.1.1.5 Comparing learning algorithms

If we have two hypothesis h_1, h_2, the true comparison is

$$d \equiv error_D(h_1) - error_D(h_2) \tag{1.7}$$

and its estimator is

$$\hat{d} \equiv error_{S_1}(h_1) - error_{S_2}(h_2) \tag{1.8}$$

$\hat{d}$ is an unbiased estimator for d, iff $h_1, h_2, S_1 and S_2$ are independent from each other

$$E[\hat{d}] = d \tag{1.9}$$

to estimate which algorithm is better we would like to estimate:

$$E_{S \subset D}[error_D(L_A(S)) - error_D(L_B(S))] \tag{1.10}$$

where L(S) is the hypothesis output by learner L using training set S
this measure can be approximated by K-Fold cross validation

1.1.2 Performance Metrics

	Predicted class	
True Class	Yes	No
Yes	TP: True Positive	FN: False Negative
No	FP: False Positive	TN: True Negative

Error rate = | errors | / | instances | = (FN + FP) / (TP + TN + FP + FN)
Accuracy = 1 - Error rate = (TP + TN) / (TP + TN + FP + FN)

Figure 1.2

Sometimes accuracy is not enough. Imagine a dataset of 90 cats and 10 dogs. An algorithm that always returns "cat" will have 90% accuracy while a classification algorithm may have 85%, but actually be good.

Unbalanced data sets are very common in problems related to anomaly detection.

Other Performance metrics are:

Recall = | true positives | / | real positives | = TP / (TP + FN)
ability to avoid false negatives (1 if FN = 0)

Precision = | true positives | / | predicted positives | = TP / (TP + FP)
ability to avoid false positives (1 if FP = 0)

Impact of false negatives and false positives depend on the application.

F1-score = $2(Precision \cdot Recall)/(Precision + Recall)$

Figure 1.3

- Recall, Sensitivity, True Positive Rate
  $TPR = TP/P = TP/(TP + FN)$
- Specificity, True Negative Rate
  $TNR = TN/N = TN/(TN + FP)$
- False Positive Rate
  $FPR = FP/N = TP/(TN + FP)$
- False Negative Rate
  $FNR = FN/P = FN/(TP + FN)$
- ROC curve: plot TPR vs FPR varying classification threshold
- AUC (Area Under the Curve)

Figure 1.4

Chapter 2

Decision Trees

2.1 Decision Trees

2.1.1 The problem

Problem: Given a training set D for a target function c, compute the ”best” consistent hypothesis wrt D.

Consider a discrete input space described with m attributes $X = A_1 \times \cdots \times A_m$ with A_i finite, and a classification problem for f: $X \rightarrow C$

The hypothesis space H: set of decision trees.

2.1.2 The decision tree

A decision tree has the following characteristics:

- each internal note tests an attribute A_i
- each branch denotes a value of an attribute $a_{i,j} \in A_i$
- each leaf node assigns a classification value $c \in C$

Instances $X = Outlook \times Temperature \times Humidity \times Wind$

$$Outlook = \{Sunny, Overcast, Rain\}$$

$$Temperature = \{Hot, Mild, Cold\}$$

$$Humidity = \{Normal, High\}$$

$$Wind = \{Weak, Strong\}$$

Classification values:

$$PlayTennis = \{Yes, No\}$$

Learning problem:

$$f : Outlook \times Temperature \times Humidity \times Wind \rightarrow PlayTennis$$

Figure 2.1

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Figure 2.2

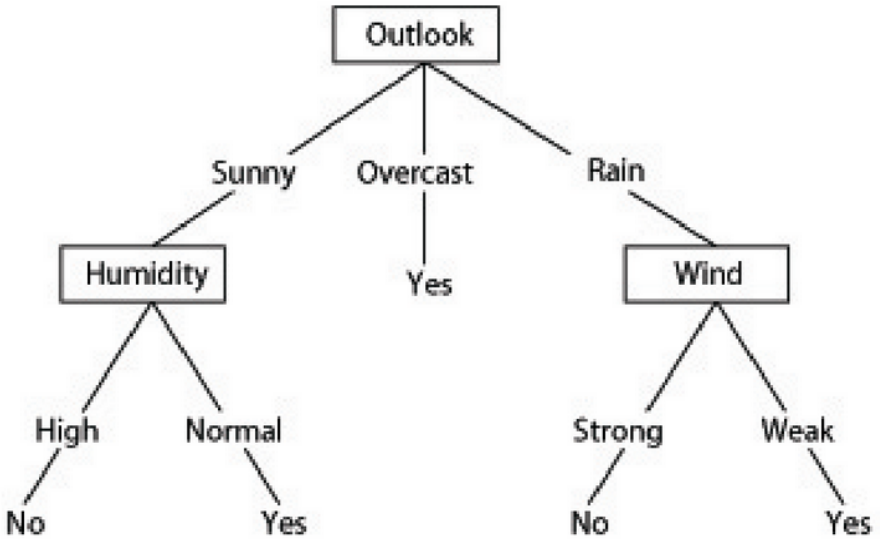


Figure 2.3

Decision trees represent a disjunction of conjunctions of constraints on the attribute values of instances.

$$(Outlook = Sunny \wedge Humidity = Normal)$$
$$\vee$$
$$(Outlook = Overcast)$$
$$\vee$$
$$(Outlook = Rain \wedge Wind = Weak)$$

2.1.3 ID3 Algorithm

Input: Examples, Target attribute, Attributes
Output: Decision Tree

1. Create a Root node for the tree
2. if all Examples are positive, then return the node Root with label +
3. if all Examples are negative, then return the node Root with label -
4. if Attributes is empty, then return the node Root with label = most common value of Target attribute in Examples
5. Otherwise
    - A ← the "best" decision attribute for Examples

- Assign A as decision attribute for Root
- For each value v_i of A
  - add a new branch from Root corresponding to the test A = v_i
  - Examples$_{v_i}$ = subset of Examples that have value v_i for A
  - if Examples$_{v_i}$ is empty then add a leaf node with label = most common value of Target_attribute in Examples
  - else add the tree ID3(Examples$_{v_i}$, Target_attribute, Attributes-A)

Remember: Output tree depends on attribute order.
We have to measure which attribute gives us the best information

2.1.3.1 Information Gain

Information gain measures how well a given attribute separates the training examples according to their target classification. ID3 selects the attribute that induces highest information gain.

The Information Gain is measured as a reduction in entropy.

$$Entropy(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus} \tag{2.1}$$

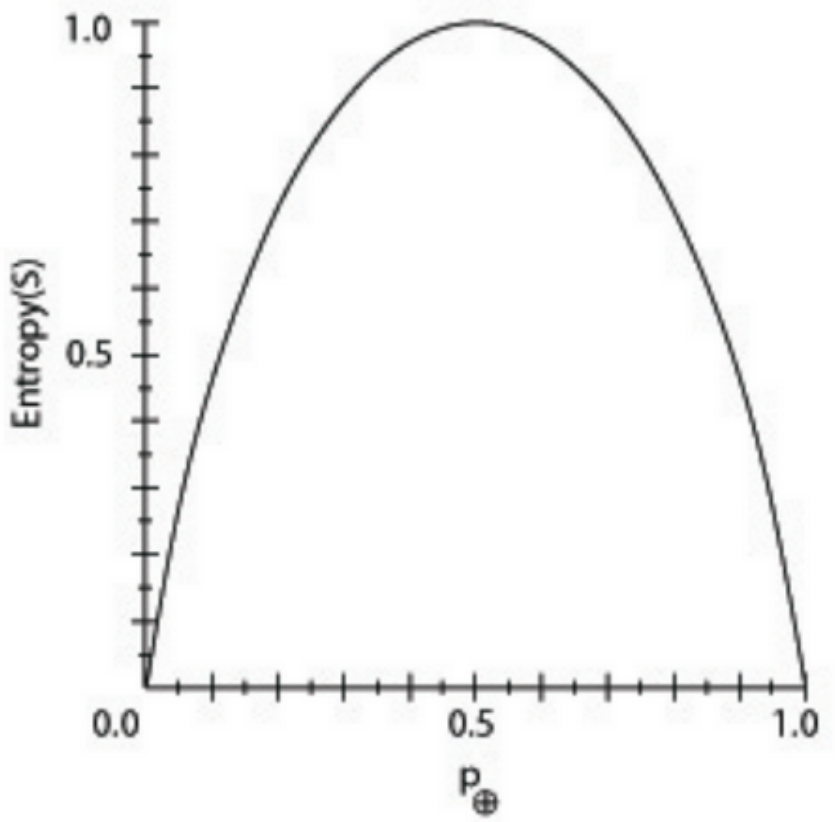


Figure 2.4

Consider the set S = [9+, 5-] (9 positive examples, 5 negative examples)

$$Entropy(s) = -(9/14)\log_2(9/14) - (5/14)\log_2(5/14) = 0.940$$

$$Entropy(S) \equiv \sum_{i=1}^{c} -p_i \log_2 p_i \tag{2.2}$$

Now we can calculate the *Gain*, which is by definition the expected reduction in entropy fo S caused by knowing the value of an attribute A.

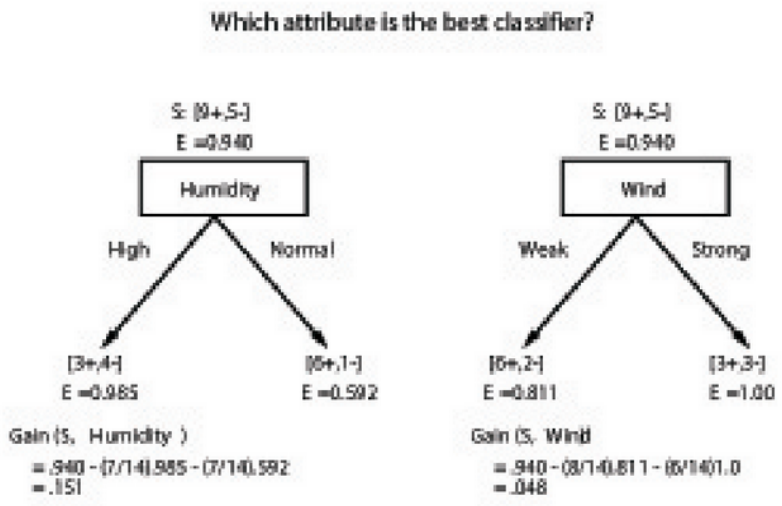
$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v) \tag{2.3}$$

where $Values(A)$ is the set of all possible values of A
$S_v = s \in S | A(s) = v$

$$\begin{aligned} Values(Wind) &= \{Weak, Strong\} \\ S &= [9+, 5-] \\ S_{Weak} &= [6+, 2-] \\ S_{Strong} &= [3+, 3-] \end{aligned}$$

$$\begin{aligned} Gain(S, Wind) &= Entropy(S) - \frac{8}{14} Entropy(S_{Weak}) - \frac{6}{14} Entropy(S_{Strong}) \\ &= 0.940 - \frac{8}{14} 0.811 - \frac{6}{14} 1.00 \\ &= 0.048 \end{aligned}$$

Figure 2.5



$$\begin{aligned} \text{Gain(S,Outlook)} &= 0.246 \\ \text{Gain(S,Humidity)} &= 0.151 \\ \text{Gain(S,Wind)} &= 0.048 \\ \text{Gain(S,Temperature)} &= 0.029 \end{aligned}$$

Figure 2.6

2.1.4 Issues in Decision Tree Learning

Determining how deeply to grow the DT
Handling continuous attributes
Choosing appropriate attribute selection measures
Handling training data with missing attribute values
Handling attributes with different costs

How can we avoid overfitting?

- stop growing when data split not statistically significant
- grow full tree, then post-prune

To determine the correct tree size:

- use a separate set of examples (distinct from the training examples) to evaluate the utility of post-pruning
- apply a statistical test to estimate accuracy of a tree on the entire data distribution
- using an explicit measure of the complexity for encoding the examples and the decision trees.

2.1.4.1 Reduced-Error Pruning

Split data into *training* and *validation* set
Do until further pruning is harmful (decreases accuracy):

- Evaluate impact on validation set of pruning each possible node (remove all the subtree and assign the most common classification)
- Greedily remove the one that most improves validation set accuracy

By pruning the tree we are generating smaller and more accurate versions of the tree. Obviously, if the dataset is limited, pruning can led to bad result.

2.1.4.2 Continuous variables

We can use discrete threshold and set rules based on them. As instance, if we have to measure the temperature, we can st a check "IF temperature > 30C°"

2.1.4.3 Attributes with many values

We use the $GainRation$ instead

$$GainRation(S, A) \equiv \frac{Gain(S, A)}{SplitInformation(S, A} \tag{2.4}$$

$$SplitInformation(S, A) \equiv -\sum_{i=1}^{c} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|} \tag{2.5}$$

where S_i is subset of S for which A has value v_i

2.1.4.4 Attributes with Cost

What if Attributes have costs related to them?
we can replace the gain with:

- Tan and Schlimmer (1990)
$$\frac{Gain^2(S, A)}{Cost(A)} \tag{2.6}$$
- Nunez (1988) (w $\in$ [0, 1] determines importance of cost)
$$\frac{2^{Gain(S,A)} - 1}{(Cost(A) + 1)^w} \tag{2.7}$$

2.1.4.5 Unknown Attribute values

We can use training example anyway, sort through tree

- If node n tests A, assign most common value of A among other examples sorted to node n
- assign most common value of A among other examples with same target value
- assign probability p_i to each possible value v_i of A
  - assign fraction p_i of example to each descendant in tree

Classify new examples in same fashion

2.1.5 Other algorithms based on Decision Trees

Random Forests: this algorithm generates a collection of decision trees with some random criteria and integrates their values into a final result.

The Random criteria can be: 1) random subsets of data (bagging), 2) random subset of attributes (feature selection) etc.

The Integration of results is made by looking at the majority vote

Random forests are less sensitive to overfitting

Chapter 3

Probabilities

3.1 Probability

3.1.1

sectionUncertainty We want to predict and approximate Uncertainty.

A pure logical approach may not be fully realistic. It can lead to conclusions that are too weak for decision making. It also leads to non-optimal decisions.

We want to have a representation of uncertainty with probabilities.

3.1.2 Probability

Sample space: Ω is the sample space (set of possibilities). $\omega \in \Omega$ is a sample point/possible world/atomic event/outcome of a random process...

Probability space (or model): is a function $P : \Omega \rightarrow R$ such that:

- $0 \leq P(\omega) \leq 1$
- $\sum_{\omega \in \Omega} P(\omega) = 1$

3.1.3 Event

An event A is any subset of Ω.
Probability of an event A is a function assigning to A a value [0, 1]

$$P(A) = \sum_{\omega \in A} P(\omega) \tag{3.1}$$

3.1.4 Random Variables

A **random variable** is a function from the sample space Ω to some range $X : \Omega \rightarrow B$
Example: $Odd : \Omega \rightarrow Boolean$
P introduces a probability distribution for a random variable X:

$$P(X = x_i) = \sum_{\{\omega \in \Omega | X(\omega) = x_i\}} P(\omega) \tag{3.2}$$

3.1.5 Proposition

A **proposition** is the event where an assignment to a random variable holds. Propositions can be combined using standard logical operators. We can map functions to propositional logic.

3.1.6 Syntax for proportions

Propositional or Boolean random variables, Discrete random variables, Continuous random variables, Arbitrary Boolean combinations of basic proposition.

3.1.7 Prior

Prior or **unconditional** probabilities of propositions correspond to belief prior to arrival of any (new) evidence

3.1.8 Probability Distribution

A **probability distribution** is a function that assigns to each possible value of the random variable an apriori probability. The sum of all the values must be 1. Concepts of Probability distribution can be extended to continuous variables.

3.1.9 Joint probability distribution

Joint probability distribution for a set of random variables gives the probability of every atomic joint event of those random variables.

Joint probability distribution of two random variables is a grid where each cell is the sum of the probabilities of the event that correspond to the column and row.

3.1.10 Conditional/Posterior probability

Belief after the arrival of some evidence. I know the outcome of a random variable, how does this affect probability of other random variables?
$P(X = true|W = true) =?$
$P(X = true|W = true) \neq P(X = true, W = true) \neq P(X = true)$

3.1.10.1 Definition of conditional probability

Conditional probability:

$$P(a|b) \equiv \frac{P(a \wedge b)}{P(b)} \; if \; P(b) \neq 0 \tag{3.3}$$

Product rule:

$$P(a \vee b) = P(a|b)P(b) = P(b|a)P(a) \tag{3.4}$$

A general version holds for whole distributions.

3.1.11 Total probabilities

$$P(a) = P(a|b)P(b) + P(a|\neg b)P(\neg b) \tag{3.5}$$

in general, for a random variable Y accepting mutual exclusive values y:

$$P(X) = \sum_{y_i \in D(Y)} P(X|Y = y_i)P(Y = y_i) \tag{3.6}$$

D(Y): set of values for variable Y.

3.1.12 Chain Rule

Chain rule of derived by successive application of product rule:

$$P(X_1, X_2 = P(X_1)P(X_2|X_1)P(X_1 \dots X_n) = \prod_{i=1}^{n} P(X_i|X_1 \dots X_{i-1}) \tag{3.7}$$

3.1.13 Inference by enumeration

Start with the joint distribution:

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	**.108**	**.012**	**.072**	**.008**
¬ *cavity*	**.016**	**.064**	**.144**	**.576**

For any proposition ϕ, sum the atomic events where it is true:
$P(\phi) = \sum_{\omega:\omega \models \phi} P(\omega)$

Figure 3.1

3.1.14 Independence

A and B are independent iff $P(A|B) = P(A)$ or $P(B|A) = P(B)$ or $P(A, B) = P(A)P(B)$

Absolute independence is impossible.

3.1.15 Conditional independence

X is conditional independent from Y given Z iff $P(X|Y, Z) = P(X|Z)$
$P(X, Y|Z) = P(X|Y, Z)P(Y|Z) = P(X|Z)P(Y|Z)$

Y_i conditional independent from Y_j given Z:

$$P(Y_1 \dots Y_n|Z) = P(Y_1|Z) \dots P(Y_j|Z)$$

Chain rule + conditional independence:

$$P(X, Y, Z) = P(X|Y, Z)P(Y, Z) = P(X|Y, Z)P(Y|Z)P(Z) = P(X|Z)P(Y|Z)P(Z)$$

3.1.16 Bayes' Rule

Product rule:

$$P(a \wedge b) = P(a|b)P(b) = P(b|a)P(b) = P(b|a)P(a) =$$

$$= Bayes'rule P(a|b) = \frac{P(b|a)P(a)}{P(b)}$$

3.1.17 Bayesian networks

Graphical notation for conditional independence assertions and hence for compact specification of full joint distributions.
Syntax:

1. a set of nodes, one per variable
2. a direct, acyclic graph
3. a conditional distribution for each node given its parents

represented as a conditional probability table (CPT) giving the distribution over X_i for each combination of parent values

Chapter 4

Bayesian Learning

4.1 Bayesian Learning

4.1.1 the problem

We want to learn a function given a dataset D. We want the function to approximate a new given instance x' as close as possible to the real value.

$$v* = \underset{h \in H}{argmax} P(v|x', D)$$

Given a dataset D and hypothesis H, compute a probability distribution over H given D.

$$P(H|D)$$

we can apply Bayes rule

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

4.1.2 MAP Hypothesis

$$P(h|D) = \frac{P(H|h)P(h)}{P(D)}$$

Generally we want the most probable hypothesis h given D.
Maximum A Posteriori hypothesis h_{MAP}:

$$h_{MAP} = \underset{h \in H}{argmax} P(h|D) = \underset{h \in H}{argmax} \frac{P(D|h)P(h)}{P(D)} = \underset{h \in H}{argmax} P(D|h)P(h)$$

We removed P(D) because it is constant and does not influence the argmax.
We don't always have information about the prior $P(h)$. We need a new definition.

4.1.2.1 ML Hypothesis

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

If i assume $P(h_i) = P(h_j)$, we can further simplify, and choose the Maximum Likelihood hypothesis:

$$h_{ML} = \underset{h \in H}{argmax} P(D|h)$$

And consider all the posterior with the same probability.

4.1.3 Bruteforce MAP Hypothesis Learner

Calculate the posterior probability for each hypothesis and return the highest one.

4.1.4 Bayes Optimal Classifier

consider the target function $f : X \rightarrow V, V = \{v_1 \dots v_k\}$, data set D and a new instance x $\notin$ D:

$$P(v_j|x, D) = \sum_{h_j \in H} P(v_j|x, h_i)P(h_i|D)$$

total probability over H.
h_j does not depend on $x \notin D \leftarrow P(h_i|x, D) = P(h_i, D)$

4.1.4.1 Bayes optimal classifier

If we use MAP or ML, we only get the most probable hypothesis and it is not true that the most probable hypothesis is also the correct class given a sample x. We can make a step further and use Bayes Optimal Classifier to get the most probable class. Given a Bayes Optimal Classifier, we can predict the class v_j of a new instance x as:

$$v_{OB} = \underset{v_j \in V}{argmax} \sum_{h_i \in H} P(v_j|x, h_i)P(h_i|D)$$

As we can see, the classifier needs the conditional probability of the class given a sample and an hypothesis i, and the conditional probability of the hypothesis given the Dataset.

4.1.5 General Approach

Given dataset D with $d_j = \{0, 1\}$ assuming a probability distribution $P(d_j|\Theta$
Maximum likelihood estimation

$$\Theta_{ML} = \underset{\Theta}{argmax} \log P(d_i|\Theta).$$

We can apply this method to different distributions.

4.1.5.1 Bayes Naive classifier

We could consider each attribute of x = $\{sx_1, \ldots, x_n\}$ as independent and use a Naive Bayesian Classifier to learn the probability of a given class:

$$C_{NB} = \underset{argmax}{c_j} P(c_j|D) \prod_{i=0}^{n} P(x_i|c_j, D) \tag{4.1}$$

The assumption is fundamental in Naive Bayes. However it is not true in the vast majority of real life situations. Still it behaves well. It is good for spam detection and document classification.

Chapter 5

Models For Classification

5.1 Probabilistic Models

Assuming $P(C_1) = \pi$, $P(x|C_i)\nu(x; \mathbb{N}_i, \Sigma)$. Given the dataset $D = (x_n, t_n)_{n=1}^N, t_n = 1$ if x_n belongs to class C_1, $T_n = 0$ otherwise.

N_1 is the number of samples in D belonging to C_1 and N_2 be the number of samples from C_2 ($N = N_1 + N_2$).

The likelihood function is:

$$P(t|\pi, \mu_1, \mu_2, \Sigma, D) = \prod_{n=1}^{N} [\pi \mathbb{N}(x_n; \mu_1, \Sigma)]^{t_n} [(1-\pi)\mathbb{N}(x_n; \mu_2, \Sigma)]^{(1-t_n)} \tag{5.1}$$

Determine $\pi, \mu_!, \mu_2, \Sigma$

By maximizing the log likelihood function we obtain:

$$\pi = \frac{N_1}{N}, \mu_1 = \frac{1}{N_1}\sum_{n=1}^{N} t_n x_n, \mu_2 = \frac{1}{N_2}\sum_{n=1}^{N} t_n x_n, \Sigma = \frac{N_1}{N}S_1 + \frac{N_2}{N}S_2. \tag{5.2}$$

with $S_i = \frac{1}{N_i}\sum_{n \in C_i}(x_n - \mu_i)(x_n - \mu_i)^T, i = 1, 2.$

If we want to predict a new sample x':

$$P(C_1|x') = \sigma(w^T x' + W_0) \tag{5.3}$$

5.1.1 Multiple classes

$$P(C_k|x) = \frac{P(x|C_k)P(C_k)}{\sum_j P(x|C_j)P(C_j)} = \frac{exp(a_k)}{\sum_j exp(a_j)} \tag{5.4}$$

with $a_k = \ln P(x|C_k)P(C_k)$

5.1.2 Gaussian Naive Bayes

$P(C_k) = \pi_k, P(x|C_k) = \mathbb{N}(x : \mu_k, \Sigma)$ Data set $D = \{(x_n, t_n)_{n=1}^N\}$, with t_n 1-of-K encoding

$$\pi = \frac{N_k}{N}, \mu_k = \frac{1}{N_k}\sum_{n=1}^{N} t_{nk}x_n, \Sigma = \sum_{k=1}^{K}\frac{N_k}{N}S_k, S_k = \frac{1}{N_k}\sum_{n=1}^{N} t_{nk}(x_n - \mu_k)(x_n - \mu_k)^T \tag{5.5}$$

Prediction of new sample $x \notin D$

$$\underset{C_k \in C}{argmax} P(C_k|x) = \underset{k}{argmax}\frac{exp(a_k)}{\sum_j exp(a_j)} \tag{5.6}$$

with $a_k = \ln P(x|C_k)P(C_k)$

5.1.3 Compact Notation

Model:

$$w^T x + w_0 = (w_0 w)(\begin{bmatrix}1\\x\end{bmatrix}) \tag{5.7}$$

5.2 Discriminative models

In the discriminative models we only want to compute the posterior probabilities without estimating the distribution. We can not sample new elements from the distribution

The likelihood for a parametric model $M_\Theta : P(t|\Theta, D), D =$ and the maximum likelihood solution is:

$$\Theta^* = \underset{\Theta}{argmax} \ln P(t|\Theta, X) \tag{5.8}$$

when M_Θ belongs to the exponential family, likelihood $P(t|\Theta, X)$ can be expressed in the form $P(t|\tilde{w}, X)$, with maximum likelihood

$$\tilde{w}^* = \underset{\tilde{w}}{argmax} lnP(t|\tilde{w}, X) \tag{5.9}$$

Without estimating the model parameters. Simplified notation: $P(t|\tilde{w})$

5.2.1 Logistic regression

5.2.1.1 Two classes

given the dataset D with each sample of the dataset $t_i \in \{0, 1\}$

Likelihood function:

$$p(t|\tilde{w}) = \prod_{n=1}^{n} y_n^{t_n}(1 - y_n)^{1-t_n} \tag{5.10}$$

with

$$t_n = p(C_1|\tilde{x}_n) = \sigma(\tilde{w}^T\tilde{x}_n) = \frac{1}{1 + e^{-\tilde{w}^T\tilde{x}_n}}$$

If we use the logarithm of the previous formula as loss function, we get the cross entropy error function (negative log likelihood)

$$E(\tilde{w}) \equiv -\ln P(t|\tilde{w}) = -\sum_{n=1}^{N}[t_n \ln y_n + (1 - t_n)\ln(1 - y_n)] \tag{5.11}$$

We have to solve $\tilde{w}^* = \underset{\tilde{w}}{argmin} E(\tilde{w})$. Many solvers are available for this problem.

Iterative reweight least squares Neqton-Raphson iterative optimization for minimizing the Gradient of the error wrt $\tilde{w}$

$$\nabla E(\tilde{w}) = \sum_{n=1}^{N}(y_n - t_n)\tilde{x}_n \tag{5.12}$$

Gradient descent step

$$\tilde{w} \rightarrow \tilde{w} - H(\tilde{w})^{-1}\nabla E(\tilde{w}) \tag{5.13}$$

where $H(\tilde{w}) = \nabla\nabla E(\tilde{w})$ is the Heissan matrix of $E(\tilde{w})$

Classify a new sample $\tilde{x}'$ as C_k where $k^* = \underset{k=1\ldots K}{argmax} P(C_k|\tilde{x}', \tilde{w}^*)$

Generalization if we have a target function $f : X \leftarrow C$ and a dataset D we define an error function $E(\Theta)$ and solve the optimization problem

$$\Theta^* = \underset{\Theta}{argmin} E(\Theta) \tag{5.14}$$

and classify a new sample as $y(x'; \Theta^*)$

5.2.1.2 Learning in feature space

we can replace x_n with $\Phi(x_n)$ to transform the dataset in an input space to get Φ' (we transform the dataset).

Compact notion two classes:

$$y(x) = w^T x + w_0 = \tilde{w}^T\tilde{x} \tag{5.15}$$

with

$$\tilde{w} = \begin{bmatrix} w_0 \\ w \end{bmatrix}, \tilde{x} = \begin{bmatrix} 1 \\ x \end{bmatrix} \tag{5.16}$$

with k classes there are more rows

Multiple classes Cannot use combination of binary linear models!!! One-versus-the-rest classifiers and one-versus-one-classifiers. We can instead use K-class discriminant comprising K linear functions. Then classify x as C_k if $y_k(x) > y_j(x)$ for all $j \neq k$. Decision boundary between C_k and C_j:

$$(\tilde{w}_k - \tilde{w}_j)^T \tilde{x} = 0 \tag{5.17}$$

5.3 Learning linear discriminant

methods to solve the linear discriminant problem

5.3.1 Least Squares

Given the dataset D, use 1 of K coding scheme, where each row is $t_n = (0, \ldots, 1, \ldots, 0)^T$
Minimize

$$E(\tilde{W}) = \frac{1}{2} \mathrm{Tr}\{(\tilde{X}\tilde{W}\ T)^T (\tilde{X}\tilde{W}\ T)\} \tag{5.18}$$

where:

$$\tilde{W} = (\tilde{X}^T \tilde{X})^{-1} (\tilde{X}^T\ T) \tag{5.19}$$

the $(\tilde{X}^T \tilde{X})^{-1}$ term is usually called pseudo inverse and is denoted as $X^\dagger$. Note: Least square is not robust to outliers.

5.3.2 Perceptron

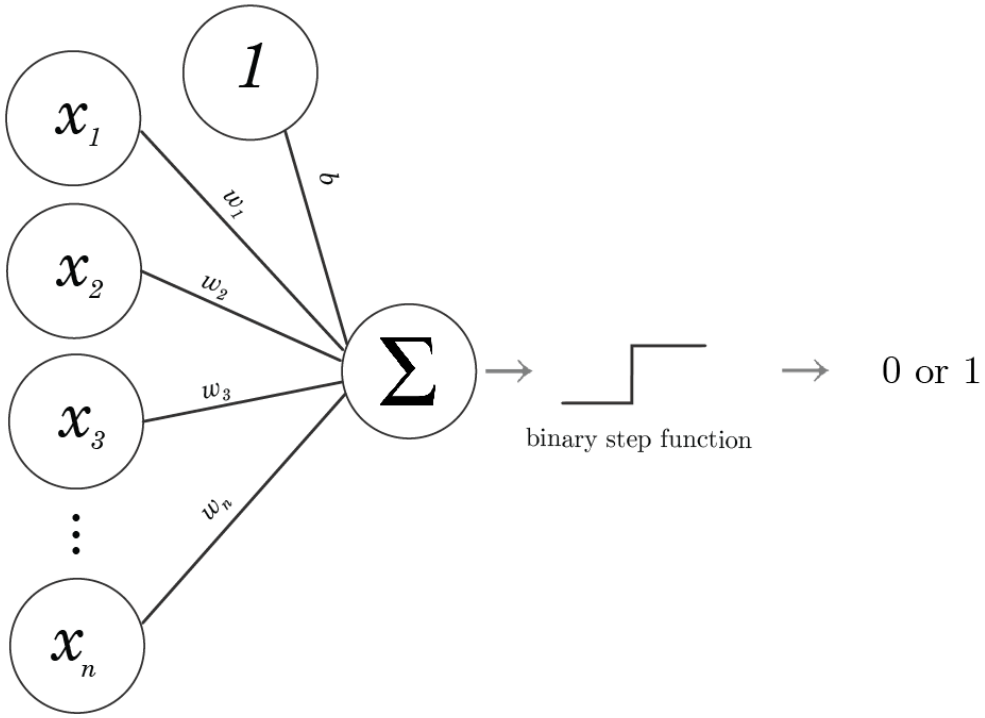


Figure 5.1

$o(x_1, \ldots, x_d)$ is 1 if $w_0 + w_1 x_1 + \cdots + w_0 + w_n x_n > 0$, -1 otherwise.

The perceptron training rule is to minimize the squared error (loss function):

$$E(w) \equiv \frac{1}{2} \sum_{n=1}^{N} (t_n - o_n)^2 = \frac{1}{2} \sum_{n=1}^{N} (t_n - w^T x_n)^2 \tag{5.20}$$

if we calculate the derivative of the error function we get:

$$\frac{\partial E}{\partial w_i} = \cdots = \sum_{n=1}^{N} (t_n - w^T x_n)(-x_{i,n}) \tag{5.21}$$

and then we use this derivative to move weights towards the the minimum.

$$w_i \leftarrow w_i + \eta \Delta w_i \tag{5.22}$$

where Δw_i is the derivative of the loss.

Perceptron converge if training data is clearly separable and the learning rate η is sufficiently small. It can terminate if the error reaches a threshold or there is a max number of iterations.

5.3.3 Ficher's linear discriminant

adjust w to find a direction that maximizes class separation based on the projection on the line. Consider a data set with N_1 points in C_1 and N_2 points in C_2

$$m1 = \frac{1}{N_1} \sum_{n \in C_1} x_n \tag{5.23}$$

$$m2 = \frac{1}{N_2} \sum_{n \in C_2} x_n \tag{5.24}$$

choose w that maximizes $J(w) = w^t(m_2 - m_1)$, subject to $||w|| = 1$

$$w \propto S_W^{-1}(m_2 - m_1) \tag{5.25}$$

we have to maximize

$$J(w) = \frac{w^T S_B w}{w^T S_W w} \tag{5.26}$$

by solving $\frac{d}{dw} J(w) = 0$. In this formula, S_B is the between covariance matrix while S_w is the within covariance matrix.

5.3.4 Support Vector Machines

Support Vector Machines (SVM) for classification aims at finding a maximizing the margin providing better accuracy.

Assume that the dataset is linearly separable,

let x_k be the closest point of the data set D to the hyperplane $\overline{h} : \overline{w}^T x + \overline{w_0}$. The **margin** (smallest distance between x_k and $\overline{h}$) is $\frac{y(x_k}{||w||}$ So, given a dataset, the margin is computed as:

$$\min_{n=1,\dots,N} \frac{y(x_k)}{||w||} = \cdots = \frac{1}{||w||} \min_{n=1,\dots,N} [t_n(\overline{w}^T x_n + \overline{w_0})] \tag{5.27}$$

using the property $|y(x_n)| = t_n y(x_n)$

Now we can calculate the hyperplane with the maximum margin by maximizing:

$$w^*, w_0^* = \underset{w, w_0}{argmax} \frac{1}{||w||} \min_{n=1,\dots,N} [t_n(w^T x_n + w_0)] \tag{5.28}$$

We can rescale the dataset such that the closest point x_k has:

$$t_k(w^T x_k + w_0) = 1 \tag{5.29}$$

and each other point has margin ≥ 1

When we find the maximum margin hyperplane w^*, w_0^*, there will be at least 2 closest points $x_k^{\oplus}$ and $x_k^{\ominus}$ (one for each class). The margin for $x_k^{\oplus}$ will be +1 while the margin for $x_k^{\ominus}$ will be -1.

In order to find a solution to he problem:

$$w^*, w_0^* = argmax \frac{1}{||w||} = argmin \frac{1}{2}||w||^2 \tag{5.30}$$

subject to:

$$t_n(w^T x_n + w_0) \geq 1 \forall n = 1, \ldots, N \tag{5.31}$$

we can use the quadratic programming solution with the Lagrangian method. The solution will be:

$$w^* = \sum_{n=1}^{N} a_n^* t_n x_n \tag{5.32}$$

a_i^* (Lagrange multipliers): result of the Lagrangian optimization problem:

$$\tilde{L}(a) = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m t_n t_m x_n^T x_m \tag{5.33}$$

subject to

$$\begin{aligned} a_n \geq 0 \; \forall n = 1, \ldots, N \\ \sum_{n=1}^{N} a_n t_n = 0 \end{aligned} \tag{5.34}$$

Notes samples x_n for which $a_n^* = 0$ will not count to the solution.
Karush-Kuhn-Tucker (KKT) condition: for each $x_n \in D$, either $a_n^* = 0$ or $t_n y(x_n) = 1$
thus $t_n y(x_n) > 1$ implies $a_n^* = 0$
Support vectors are those points such that $t_k y(x_k) = 1$ and $a_k^* > 0$

$$SV \equiv \{x_k \in D | t_k y(x_k) = 1\} \tag{5.35}$$

If we want to express the hyperplane with his support vectors we can use:

$$y(x) = \sum_{x_j \in SV} a_j^* t_j x^T x_j \text{ è } w_0^* = 0 \tag{5.36}$$

To compute w_0^*, we have to find support vectors that satisfies $t_k y(x_k) = 1$:

$$t_k(\sum_{x_j \in SV} a_j^* t_j x_k^T x_j) = 1 \tag{5.37}$$

multiplying by t_k and using $t_k^2 = 1$

$$w_0^* = t_k - \sum_{x_j \in SV} a_j^* t_j x_k^T x_j \tag{5.38}$$

Instead of using one particular support vector x_k to determine w_0, a more stable solution can be obtained by averaging over all the support vectors:

$$w_0^* = \frac{1}{|SV|} \sum_{x_k \in SV} \left(t_k - \sum_{x_j \in S} a_j^* t_j x_k^T x_j \right) \tag{5.39}$$

In order to classify a new instance, we can look at the sign:

$$y(x') = sign \left(\sum_{x_k \in SV} a_j^* t_j x'^T x_j \right) \tag{5.40}$$

Soft margin If data is almost linearly separable, we can use a soft margin $\xi_n \geq 0,\ n = 1, \ldots, N$ (called slack variable). A soft margin lets some of the points to lay between the support vectors, adding a border.

$$\begin{gathered} t_n y(x_n) \geq 1 - \xi_n,\ n = [1, \ldots, N] \\ w^*, w_0^* = \text{argmin} \frac{1}{2} ||w||^2 + C \sum_{n=1}^{N} \xi_n \\ subject\ to \\ t_n y(x_n) \geq 1 - \xi_n,\ n = 1, \ldots, N \\ \xi_n \geq 0, n = 1, \ldots, N \end{gathered} \tag{5.41}$$

where C is a constant.

5.3.5 Basis Function

Until now, we considered only models that work directly on the dataset D, but what if we first perform a non-linear transformation of the inputs $\Phi(x)$ (basis function)?.
By transforming the data, we can then find a linear hyperplane for the feature space $\Phi(D)$ that is not linear in the original space of D, actually separating points that were not linearly separable before the transformation.

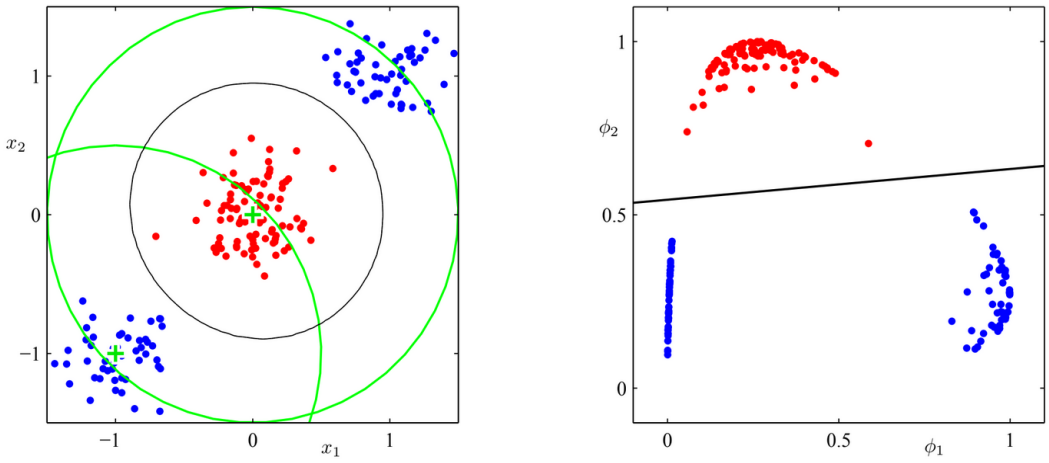


Figure 5.2

What we actually do is to train a linear model on a non-linear transformation:

$$y(x) = w^T \phi(x) + w_0 \text{ (two classes)}$$
$$y_k(x) = w_k^T \phi(x) + w_{k0} \text{ (multiple classes)} \tag{5.42}$$

Chapter 6

Linear Regression

6.1 Linear Models for Regression

6.1.1 What we want to do?

We have a dataset $d = \{(x_n, t_n)_{n=1}^N\}$ and we want to learn a function $f : X \rightarrow Y$ with $X \subseteq \mathcal{R}^d$ and $Y = \mathcal{R}$
The model is such that $y(x; W) = W^T x$ approximates the target function (W are weights of the function).

Linear model for linear functions

$$y(x; w) = w_0 + w_1 x_1 + \cdots + w_d x_d = w^T x \tag{6.1}$$

with $x = \begin{bmatrix} 1 \\ x_1 \\ \dots \\ x_d \end{bmatrix}$ and $w = \begin{bmatrix} w_0 \\ w_1 \\ \dots \\ w_d \end{bmatrix}$

Using non-linear functions of input variables means having:

$$y(x; w) = \sum_{j=0}^{M} w_j \phi_j(x) = w^T \phi(x) \tag{6.2}$$

with $w = \begin{bmatrix} w_0 \\ w_1 \\ \dots \\ w_d \end{bmatrix}$, $x = \begin{bmatrix} \phi_0(x) \\ \phi_1(x) \\ \dots \\ \phi_M(x) \end{bmatrix}$ and $\phi_0(x) = 1$ (still linear in the parameter w).

6.1.2 Example of basis function

We can define a set of functions:

$$\Phi(x) = \begin{bmatrix} \phi_0(x) \\ \dots \\ \phi_M(x) \end{bmatrix} \tag{6.3}$$

such that $y(x; W) = W^T \Phi(x)$ is linear in W and not linear in x.

Well known transformations are for example:

$$\Phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \\ \dots \\ x^M \end{bmatrix} \tag{6.4}$$

Other used transformations are Radial, Sigmoid and Tanh

6.1.3 maximum Likelihood and least square

Target value t is given by the model $y(x; w)$ affected by additive noise ϵ: $t = y(x; w) + \epsilon$
Assume Gaussian noise $P(\epsilon|\beta) = \mathbb{N}(\epsilon|0, \beta^{-1})$, with precision β. We have:

$$P(t|x, w, \beta) = \mathbb{N}(t|y(x; w), \beta^{-1}) \tag{6.5}$$

If we get all the labels t_n, regroup them in a single vector t and assume they are all independent, we can seek the maximum likelihood function:

$$p(t|x_1, \dots, x_N, w, \beta) = \prod_{n=1}^{N} \mathbb{N}(t_n|w^T \phi(x_n), \beta^{-1}) \tag{6.6}$$

or equivalently, we can consider the natural logarithm:

$$\begin{aligned} \ln p(t|x_1, \dots, x_N, w, \beta) = \sum_{n=1}^{N} \ln \mathbb{N}(t_n|w^T \phi(x_n), \beta^{-1}) = \\ \frac{N}{2} \ln(\beta) - \frac{N}{2} \ln(2\pi) - \beta E_D(w) \end{aligned} \tag{6.7}$$

where we use the sum-of-squares error function.

$$argmin\ E_D(w) = argmin\ \frac{1}{2} \sum_{n=1}^{N} [t_n - w^T \phi(x_n)]^2 \tag{6.8}$$

The gradient of the function is then:

$$\nabla \ln p(t|w, \beta) = \beta \sum_{n=1}^{N} \{t_n - w^T \phi(x_n)\} \phi(x_n)^T \tag{6.9}$$

We can solve the problem in two ways:

1. **Close Form Solution** if we set the gradient of $E_D(w)$to zero and solve for w we obtain:
$$w_{ML} = (\Phi^T\Phi)^{-1}\Phi^T t \tag{6.10}$$
where t is the array $t = \begin{bmatrix} t_0 \\ w_1 \\ \dots \\ t_{dN} \end{bmatrix}$ and Φ is $\Phi = \begin{bmatrix} \phi_0(x_1) & \dots & \phi_M(x_1) \\ \phi_0(x_2) & \dots & \phi_M(x_2) \\ \dots & \dots & \dots \\ \phi_0(x_N) & \dots & \phi_M(x_N)) \end{bmatrix}$.
Notice: $(\Phi^T\Phi)^{-1}\Phi^T$ is the *pseudo-inverse* $\Phi^\dagger$

2. **Iterative Solution**: this method does not deliver the optimal solution, but a good enough one (Close Form Solution might be too much costly in term of computation).
Stochastic Gradient Descent algorithm:
$$\hat{w} \leftarrow \hat{w} - \eta\nabla E_n \tag{6.11}$$
therefore:
$$\hat{w} \leftarrow \hat{w} - \eta[t_n - \hat{w}^T\phi(x_n)]\phi(x_n) \tag{6.12}$$

None of these methods can actually solve the method.

6.1.3.1 Regularization

For Stochastic gradient descent, regularization is crucial. We have to choose the right η in order to avoid overfitting. One of the simplest regularizer is given by the sum-of-squares of the weight vector elements.

If we consider:
$$E_D(w) + \lambda E_W(w) \tag{6.13}$$
where λ is the regularizer that control the relative importance of the data-dependent error $E_D(w)$ and the regularization term $E_W(w)$. The sum-of-squares is:
$$E_W(w) = \frac{1}{2}w^T w \tag{6.14}$$

Another common function is:
$$E_W(w) = \sum_{j=0}^{M} |w_j|^q \tag{6.15}$$

This technique is also known as *weight decay* since it penalizes the weights and tents to lower them towards zero, unless supported by the data.

6.1.3.2 Multiple Outputs

We can introduce a different set of basis functions for each component of t, leading to multiple, independent regression problems. another interesting method is using the same set of basis functions to model all of the components of the target vectors so that

$$t(x, w) = W^T \phi(x) \tag{6.16}$$

where y is a K-dimensional column vector, W is a $M \times K$ matrix of parameters, and $\phi(x)$ is a M-dimensional column vector with elements $\phi_j(x)$ and $\phi_0(x) = 1$.

We can combine a set of observations $t_1, \ldots, t_N$ into a matrix T of size $N \times K$ such that the n^{th} row is given by t_n^T. Similarly, we can combine the input vectors $x_1, \ldots, x_N$ into a matrix X. The log likelihood is then:

$$\ln p(T|X, W, \beta) = \sum_{n=1}^{N} \ln \mathbb{N}(t_n|W^T \phi(x_n), \beta^{-1} I) = \frac{NK}{2} \ln\left(\frac{\beta}{2\pi}\right) - \frac{\beta}{2} \sum_{n=1}^{N} ||t_n - W^T \phi(x_n)||^2 \tag{6.17}$$

As before, we can maximize the function wrt W:

$$W_{ML} = (\Phi^T \Phi)^{-1} \Phi^T T \tag{6.18}$$

where, for each target variable t_k, we have:

$$w_k = (\Phi^T \Phi)^{-1} \Phi_T t_k = \Phi^{\dagger} t_k \tag{6.19}$$

Chapter 7

Kernel Methods

7.1 Kernel Methods

Some of the linear parametric models can be re-cast into an equivalent 'dual representation' in which the predictions are also based on linear combination of a kernel function evaluated at training time on training points.

7.1.1 Approach

We use a similarity measure $k(x, x') \geq 0$ between the instances x, x'. $k(x, x')$ is called kernel function.
If we have $\phi(x)$, a possible choice is $k(x, x') = \phi(x)^T \phi(x')$

7.1.2 Kernel

A kernel is a real-valued function $k(x, x') \in \mathcal{R}$ for $x, x' \in \mathcal{X}$, where $\mathcal{X}$ is some abstract space. A kernel typically satisfies the following conditions:

1. symmetric: $k(x, x') = k(x', x)$
2. non-negative: $k(x, x') \geq 0$

(not strictly required)

7.1.2.1 Input normalization

Input dataset D must be normalized in order for the kernel to be a good similarity measure in practice. If we don't do that, the solution may be affected.
Types of normalization:

- min-max: $\overline{x} = \frac{x - min}{max - min}$
- normalizaiton: $\overline{x} = \frac{x - \mu}{\sigma}$
- unit vector: $\overline{x} = \frac{x}{||x||}$

Kernel families:

- **Linear**: $k(x, x') = x^T x'$
- **Polynomial**: $k(x, x') = (\beta x^T x' + \gamma)^d,\ d \in \{2, 3, \dots\}$
- **Radial Basis Function (RBF)**: $k(x, x') = exp(-\beta|x - x'|^2)$
- **Sigmoid**: $k(x, x') = tanh(\beta x^T x' + \gamma)$

7.1.3 How do we use the kernels?

Consider a linear model $y(x; w) = w^T x$ with a dataset D.
Minimize $J(w) = (t - Xw)^T (t - Xw) + \lambda||w||^2$
$X = \begin{bmatrix} x_1^T \\ \dots \\ x_N^T \end{bmatrix}$ design matrix, $t = \begin{bmatrix} t_1 \\ \dots \\ t_N \end{bmatrix}$ output vector.
The optimal solution is:

$$\hat{w} = (X^T X + \lambda I_N)^{-1} X^T t = X^T (XX^T + \lambda I_N)^{-1} t \tag{7.1}$$

with I_n an $N \times N$ identity matrix.
We can call $\lambda = (XX^T + \lambda I_N)^{-1} t$ and express our solution as:

$$\hat{w} = X^T \alpha = \sum_{n=1}^{N} \alpha_n x_n \tag{7.2}$$

which is a linear combination of coefficient α_n
We can then write: $y(x; \hat{w} = \hat{w}^T x = \sum_{n=1}^{N} \alpha_n x_n^T x$. If we then consider a linear kernel $k(x, x') = x^T x'$, we can rewrite the model as

$$y(x; \hat{w}) = \sum_{n=1}^{N} \alpha_n k(x_n, x) \tag{7.3}$$

with $\alpha = (K + \lambda I_N)^{-1} t$, and $K = XX^T$ is the **Gram Matrix**.

7.1.3.1 Recap

We can use the kernel $k(x, x') = x^T x'$ and express a model $y(x; \alpha)$ as the linear combination of α and $x_n^T X$, where α is the gram matrix summed to λI_N, inverted and multiplied by t.
The **Gram matrix** is the dot product of all the possible points in the dataset.

7.1.3.2 Trick of kernel

We can use the same problem formulation and solution for every kernel, even if it is not linear.

Kernel trick or kernel substitution: If input vector x appears in an algorithm only in the form of an inner product $x^T x'$, replace the inner product with some kernel $k(x, x')$.

7.1.3.3 SVM with kernel method

In SVM, the solution has the form:

$$w^* = \sum_{n=1}^{N} \alpha_n x_n \tag{7.4}$$

the linear model with a linear kernel is:

$$y(x;\alpha) = sign(w_0 + \sum_{n=1}^{N} \alpha_n x_n^T x) \tag{7.5}$$

and the kernel trick is:

$$y(x;\alpha) = sign(w_0 + \sum_{n=1}^{N} \alpha_n k(x_n, x)) \tag{7.6}$$

Lagrangian problem for kernelized SVM classification

$$\tilde{L}(a) = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m t_n t_m k(x_n, x_m) \tag{7.7}$$

solution:

$$a_n = \dots$$
$$w_0 = \frac{1}{|SV|} \sum_{x_i \in SV} (t_i - \sum_{x_j \in S} a_j t_j k(x_i, x_j) \tag{7.8}$$

The main problem for kernelized methods is that we have to deal with a matrix that exponentially grows with the size of the dataset. For most of the elements in SVM, the Lagrangian multiplier will be zero and so, only a subset of the values in the Gramm matrix are used for the solution, making the approach good for this case.

7.1.3.4 Linear regression and kernels

For linear regression the problem is that the computation of K requires $> N^2$ operations and K is not sparse, making the approach too costly for usage on big datasets.

What we can do for linear regression is instead use a different error function:

$$E_\epsilon(y,t) = \begin{cases} 0, & \text{if } |y-t| < \epsilon \\ |y-t| - \epsilon, & \text{otherwise} \end{cases} \tag{7.9}$$

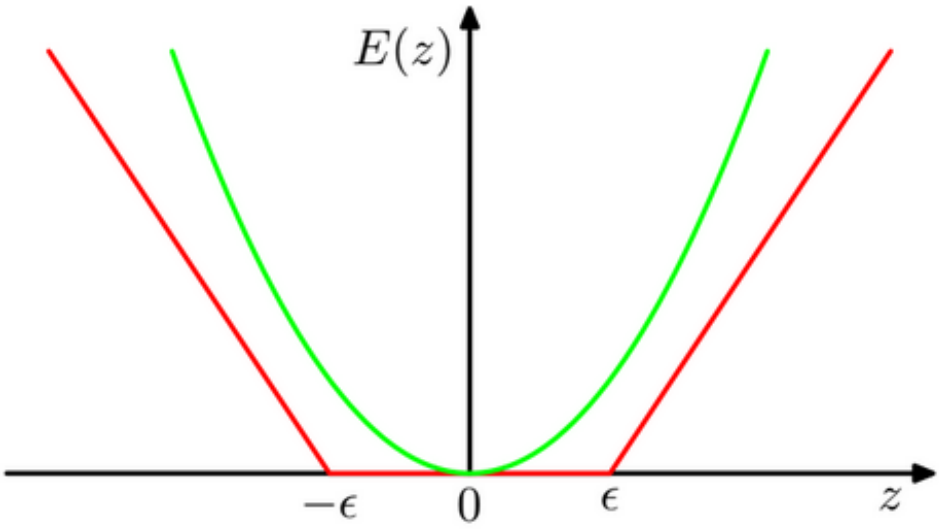


Figure 7.1: error function for linear regression that makes the kernel sparse.

Problem: it is not differentiable, so it is difficult to solve.
We can the introduce *slack variables* $\xi_n^+, \xi_n^- \geq 0$:

$$\begin{aligned} t_n &\leq y_n + \epsilon + \xi_n^+ \\ t_n &\leq y_n - \epsilon - \xi_n^- \end{aligned} \tag{7.10}$$

Points that are into the ϵ-tube $y_n - \epsilon \leq t_n \leq t_n + \epsilon \Rightarrow \xi_n = 0$

$$\begin{aligned} \xi_n^+ > 0 &\Rightarrow y_n + \epsilon \\ \xi_n^- > 0 &\Rightarrow y_n - \epsilon \end{aligned} \tag{7.11}$$

with $y_n = t(x_n, w)$.
The loss function can be rewritten as:

$$J(w) = C \sum_{n=0}^{N} (\xi_n^+ + \xi_n^-) + \frac{1}{2}||w||^2 \tag{7.12}$$

subject to 7.10 and 7.11.
This is a standard quadratic program (QP), can be "easily" solved. The Lagrangian problem is:

$$\tilde{L}(a, a') = \cdots \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m \ldots k(x_{\tilde{n}}, x_m) \ldots \tag{7.13}$$

from which we compute $\hat{a}_n$, $\hat{a}'_m$ (sparse values, most of them are zero) and

$$\hat{w}_0 = t_n - \epsilon - \sum_{m=1}^{N} (\hat{a}_m - \hat{a}'_m) k(x_n, x_m) \tag{7.14}$$

for some data point n such that $0 < a_n < C$.
The **prediction** is:

$$y(x) = \sum_{n=1}^{N} (\hat{a}_{\tilde{n}} - \hat{a}_n)' k(x, x_n) + \hat{w}_0 \tag{7.15}$$

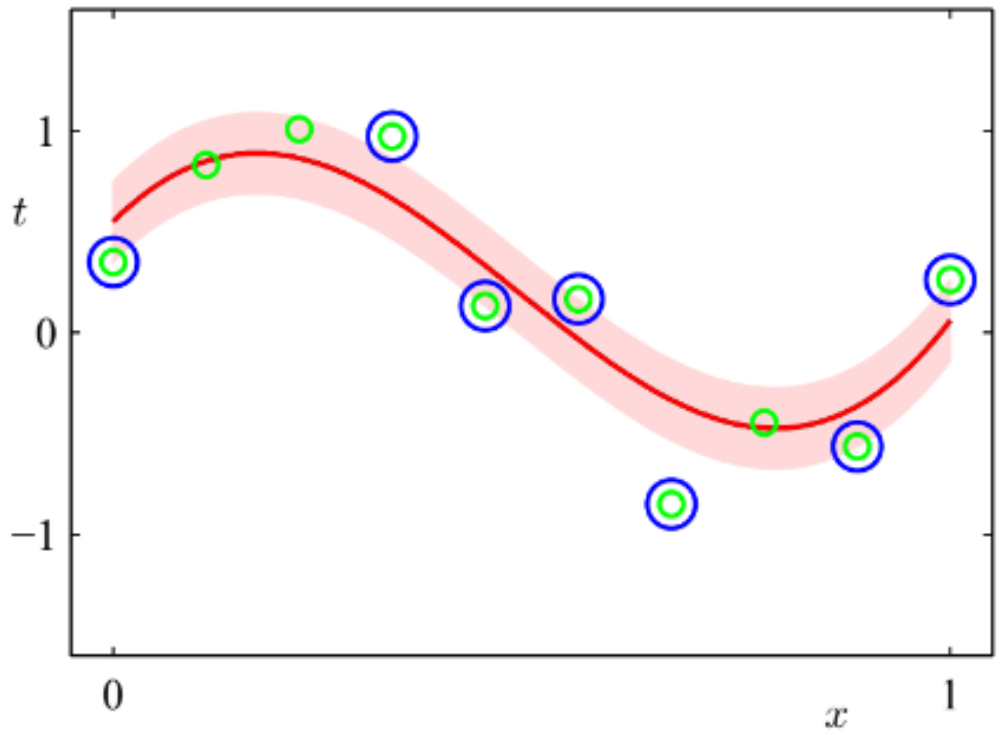
From Karush-Kuhn-Tucker (KKT) condition, **support vectors** contribute to predictions

$$\hat{a}_n > 0 \Rightarrow \epsilon + \xi_n + y_n - t_n = 0$$

data point lies on or above upper boundary of the $\epsilon - tube$, while

$$\hat{a}'_n > 0 \Rightarrow \epsilon + \xi_n - y_n + t_n = 0$$

data point lies on or below lower boundary of the ϵ-tube. All other datapoints inside the ϵ-tube have $\hat{a}_n = 0$ and $\hat{a}'_n = 0$ and thus do not contribute to prediction.



Chapter 8

Instance Based Learners

8.1 Instance Based Learners

8.1.1 Parametric vs Non-Parametric Models

Parametric models have a fixed number of parameters. Examples:

- Linear regression
- Logistic regression
- Perceptron
- ...

Non-parametric models: Number of parameters grows with amount of data.
A Simple non-parametric model is the instance-based learning.

8.1.2 K-nearest neighbors

It tries to solve the classification problem.
The classification with K-NN is as follows:

1. Find K nearest neighbors of new instance x
2. Assign to x the most common label among the majority of neighbors

The likelihood of the class c for a new instance is given by:

$$p(c|x, D, K) = \frac{1}{K} \sum_{x_n \in N_k(x_n, D)} I(t_n = c) \tag{8.1}$$

with $N_K(x_n, D)$ the K nearest points to x_n and $I(e) = \begin{cases} 1 & \text{if e is true} \\ 0, & \text{if e is false} \end{cases}$

We can change K to fit better the dataset and avoid overfitting.

8.1.3 Kernelized nearest neighbours

we can use a distance function in computing $N_K(x, D)$

$$||x - x_n||^2 = x^T x + x_n^T - 2x^T x_n \tag{8.2}$$

can be kernelized by using a kernel $k(x, x_n)$

8.1.4 Locally weighted regression

Regression problem $f : X \rightarrow \mathcal{R}$ with data set $D = \{(x_n, t_n)_{n=1}^N\}$. We can fit a local regression around a certain datapoint x_q.
We can firstly compute the K nearest neighbours of x_q, fit a regression on those neighbours and finally return $y(x_q; w)$. We can also use a kernelized regression.

Chapter 9

Multiple Learners

9.1 Multiple Learners

Also called Ensemble Learning, is based on training many different learner/model and combine their result.

The general idea is that, if we use a baseline (single model) performances may be good, if we instead use bagging or voting (parallel) performances increases and if we use boosting (sequential) performance are even better.

9.1.1 Voting

Given a dataset D

1. use D to train a set of models $y_m(x)$, for $m = 1, \dots, M$
2. make prediction with

$$\begin{aligned} y_{voting}(x) &= \sum w_m y_m(x) && \text{(regression)} \\ y_{voting}(x) &= \underset{c}{argmax} \sum w_m l(y_m(x) = c) && \text{weighted majority (classification)} \end{aligned} \tag{9.1}$$

with the sum of $w_m = 1$, $l(e) = 1$ if e is true, 0 otherwise.
If we add a Non linear gating function f depending on input, it is called **Mixture of experts**.
If we also try to learn the combination function f it is called **Stakcing**.
Cascading learners are instead based on confidence threshold.

check coding scheme.

9.1.2 Bagging

Given a dataset D:

1. generate M bootstrap data sets $D_1 \dots D_m$, with $D_i \in D$.
2. use each bootstrap data set D_m to train a model $y_m(x)$, for each m.
3. make predictions with a voting scheme

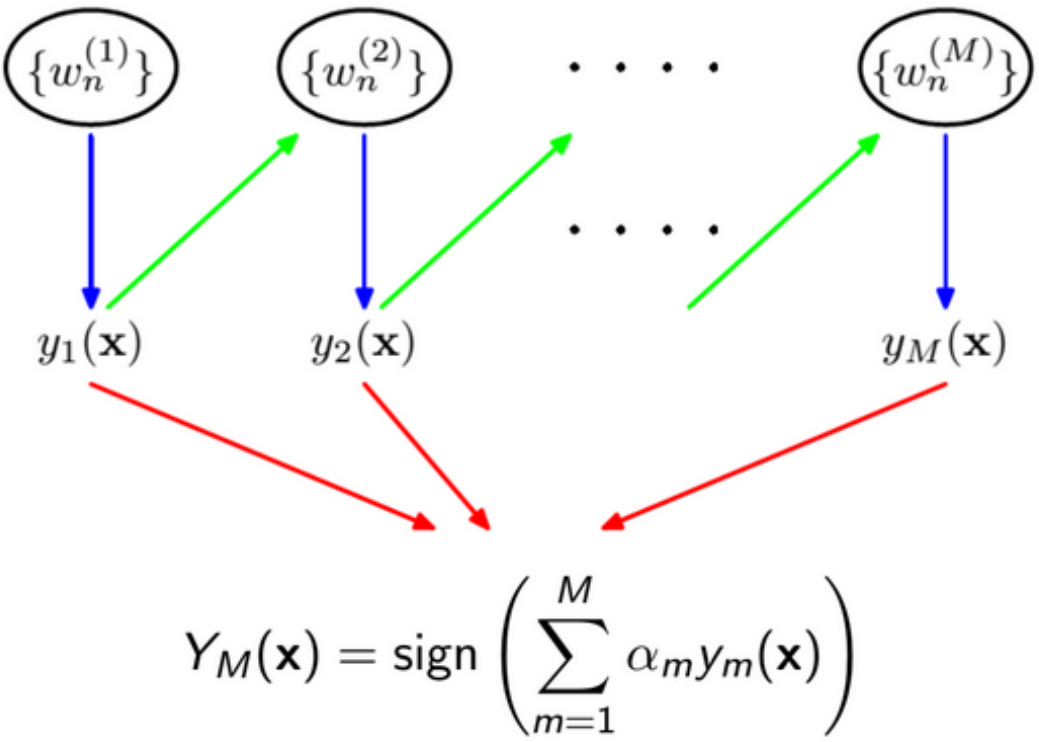
$$y_{bagging}(x) = \frac{1}{M} \sum y_m(x) \tag{9.2}$$

9.1.3 Boosting

9.1.3.1 general approach

main points are:

- Base classifiers (**weak learners**) trained **sequentially**
- Each classifier trained on weighted data
- Weights depend on performance of previous classifiers
- Points misclassified by previous classifiers are given greater weight. So that next models will classify better those points
- Predictions based on weighted majority of votes



9.1.4 AdaBoost

Given $D = \{(x_1, t_1), \ldots, (x_n, t_n)\}$, where $x_n \in X, t_n \in \{-1, +1\}$

1. Initialize $w_n^{(m)} = 1/N,\ n = \ldots, N$
2. For $m = 1, \ldots, M$:
   - Train a weak learner $y_n(x)$ my minimizing the weighted error function:

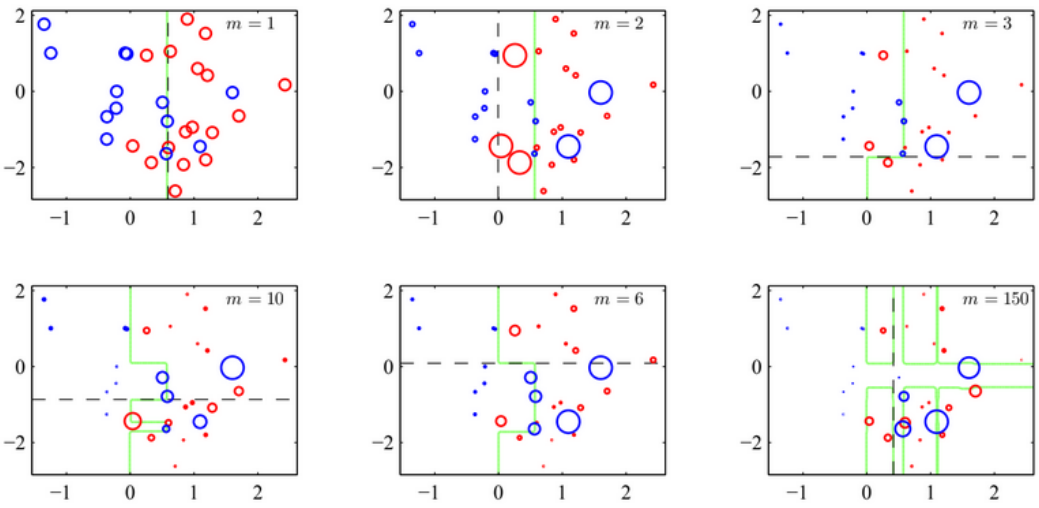
$$J_m = \sum_{n=1} * N w_n^{(m)} l(y_m(x_n) \neq t_n) \tag{9.3}$$

   - Evaluate: $\epsilon_m = \frac{\sum w_n^{(m)} l(y_m(x_n) \neq t_n)}{\sum w_n^{(m)}}$ and $\alpha_m = \ln[\frac{1-\epsilon_m}{\epsilon_m}]$
   - Update the data weighting coefficients:

$$w_n^{(m+1)} = w_n^{(m)} exp[\alpha_m l(y_m(x_n \neq t_n)] \tag{9.4}$$

3. Output the final classifier:

$$Y_M(x) = sign(\sum_{m=1}^{M} \alpha_m y_m(x)) \tag{9.5}$$



9.1.4.1 Exponential Error Minimization

AdaBoosting can be explained as the sequential minimization of an exponential error function.
Consider the error function

$$E = \sum_{n=1}^{N} exp[-t_n f_M(x_n], \tag{9.6}$$

where f_M is the final classifier

$$f_M(x) = \frac{1}{2} \sum_{m=1}^{M} \alpha_m y_m(x), \quad t_n \in \{-1, +1\} \tag{9.7}$$

the goal is to minimize the error function E w.r.t. $\alpha_m, y_m(x), m = 1, \ldots, M$

Sequential minimization, instead of minimizing E *globally*

- assume $y_1(x), \ldots, y_{M-1}(x)$ and $\alpha_1, \ldots, \alpha_{M-1}$ fixed
- minimize w.r.t. $y_M(x)$ and α_M.

Making $y_M(x)$ and α_M explicit we have:

$$E = \sum_{n=1}^{N} exp[-t_n f_{M-1}(x_n) - \frac{1}{2} t_n \alpha_M y_M(x_n] = \sum w_n^{(M)} exp[-\frac{1}{2} t_n \alpha_M y_M(x_n)] \tag{9.8}$$

with $w_n^{(M)} = exp[-t_n f_{M-1}(x_n)]$ constant as we are optimizing w.r.t. α_M and $y_M(x)$

From sequential minimization we obtain $w_n^{(m+1)}$. Then we can predict with:

$$sign(f_M(x)) = sign(\frac{1}{2}\sum \alpha_m y_m(x)) \tag{9.9}$$

which is equal to

$$Y_M(x) = sign(\sum \alpha_m y_m(x)) \tag{9.10}$$

thus providing that AdaBoost minimizes such error function

9.1.4.2 AdaBoost: remarks

Pros:

- Fast and simple to implement
- no prior knowledge about base learners is required
- no parameters to tune
- can be combined with any method for finding best learners
- theoretical guarantees given sufficient data and base learners with moderate accuracy

Cons:

- Performance depends on data and the base learners
- Sensitive to noise

Chapter 10

Artificial Neural Networks

10.1 Artificial Neural Networks

With this chapter, the second part of the course begins.

10.1.1 Problem

We have $f : X \rightarrow Y$, a dataset $D = \{(x, t)\}$, a model $\hat{f}(x; \Theta)$, an error function $E(\Theta)$, we want to minimize $\Theta^* = argminE(\Theta)$ and predict the value of $x' \notin D$, $\hat{f}(x', \Theta)$. The minimization will be tone through Iterative Linear Descent.

What is different from the previous models? $\hat{f}$ is non-linear both in the parameters and the dataset. So, NN provide non linear models in the set of parameters.

The second important aspect is that NN has the advantage that you don't have to specify a kernel. This is because kernels are automatically learned.

Alternative names are **Feedforward neural network** or **Multilayer perceptrons**. NN are function estimators and learn parameters so that the function **f** that they are learning approximates well the unknown function f.

10.1.1.1 Inspiration

NN were inspired by what we know of our brain. Note that they are not a model of the brain, but only comprehends some insights. Outputs can be seen as an array of units (neurons) activation based on the connections with the previous units. Feedforward NN are organized in layers, where each layer contains a set of neurons and each one is connected to every unit of the layer that follows. Information flows without any loop from the input to the output, passing through the hidden units.

$$f(x; \Theta) = f^3(f^2(f^1(x; \Theta^1); \Theta^2); \Theta^3) \tag{10.1}$$

where f^i is the i-th layer and Θ^i is the corresponding parameters.

The power of FNN is based on the fact that we don't have to handcraft kernels or to use a known one.

10.1.2 XOR problem

Consider the problem where we have a dataset: $D = \{((0,0)^T, 0), ((0,1)^T, 1), ((1,0)^T, 1), ((1,1)^T, 0), \}$ and we want to use the linear regression using the Mean Square Error (MSE). We can instantly notice that the datast is not linearly separable. If we use a linear model with a linear kernel it will not work. We could try a kernel, but with FNN we don't need to handcraft it.

We can build a two-layer network with two units as input layer, two units as hidden layer and one unit as output layer. We also have to define **activation function** These functions are present in any hidden layers as well as in the output layer. We can chose a different activation function for each layer. For this example we can use a **ReLU (rectified linear unit)** for the hidden layer and a linear activation function for the output layer.
ReLU: The ReLU units is a function that returns: $f(z) = max(0, z)$.
The i-th neuron of the hidden layer will have the form of:

$$h_i = ReLU\left((w_{i,1}, w_{i,2})^T \begin{pmatrix} x_{i,1} \\ x_{i,2} \end{pmatrix} + w_{i,0}\right) \tag{10.2}$$

we can then organize as a column vector each unit h_i of the layer and rewrite as:

$$h = \begin{pmatrix} h_1 \\ h_2 \end{pmatrix} = ReLU\left(\begin{pmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} w_{1,0} \\ x_{2,0} \end{pmatrix}\right) = ReLU\left(W^T x + W_0\right) \tag{10.3}$$

and so the ourpur of the model y is

$$y = w^T h + w_0 = w^t ReLU\left(W^T x + W_0\right) + w_0 \tag{10.4}$$

We can now choose an error function. For instance, can use the MSE and minimize.

10.1.3 Cost Function

A model implicitly defines a conditional distribution $p(t|x, \Theta)$. This cost function is the Maximum likelihood principle (**cross-entropy**).

$$J(\Theta) = E_{x,t\sim D}\left[-ln(p(t|x;\Theta))\right] \tag{10.5}$$

For example we can assume the function to be a Gaussian distribution and hence use it to estimate and minimize the loglikelihood.

10.1.4 Output units

The choice of the loss function is related to the problem that we are modeling and to the output units of the network.

1. Regression
2. Binary Classification
3. Multi-classes classification

Regression
linear units: identity activation function.

$$y = W^T h + b \tag{10.6}$$

use a Gaussian distribution noise model

$$p(t|x) = N(t|y, \beta^{-1}) \tag{10.7}$$

Cost function = maximum likelihood that is equal to the mean squared error.

Binary classification
Sigmoid units: sigmoid activation function.
The likelihood corresponds to a Bernoulli distribution.

$$\begin{aligned} &J(\Theta) = E_{x,t\sim D}\left[-ln(p(t|x))\right] \\ &\textbf{where } -ln(p(t|x)) = softplus((1-2t)\alpha) \\ &\qquad\qquad\qquad\qquad \textbf{with } \alpha = w^T h + b. \end{aligned} \tag{10.8}$$

Multi-class Classification Uses Softmax units as activation function.

$$y_i = softmax(\alpha_i) = \frac{exp(\alpha_i)}{\sum exp(\alpha_i)} \tag{10.9}$$

And corresponds to a multinomial distribution:

$$J_i(\Theta) = E_{x,t\sim D}\left[-\ln(softmax(\alpha_i))\right] \tag{10.10}$$

with $\alpha_i = w_i^T h + x$

10.1.4.1 Activation Functions for Hidden Units

There is not a real rule. The choice is driven by intuition.

- **ReLU**: rectified linear units

$$g(\alpha) = max(0, \alpha) \tag{10.11}$$

  easy to optimize but not differentiable on zero (does not cause problems in practice).

- **Sigmoid**:
$$g(\alpha) = \sigma(\alpha) \tag{10.12}$$
- **Hyperbolic tangent**:
$$g(\alpha) = tanh(\alpha) \tag{10.13}$$
It is closely related to the sigmoid: $tanh(\alpha) = 2\sigma(2\alpha) - 1$

Do not use logaraithms since units saturate easily otherwise.Gradient based learning is very slow. $tanh$'s gradient is larger than sigmoid's one.

10.1.5 Gradient computation

Information flows forward through the network while computing network output y from input x.

To train the parameters we need to compute gradients wrt the network parameters Θ.

The **backpropagation** algorithm propagates the information back from the output to the whole network.

Backpropoagation is not specific to FNNs and is not a training algorithm. It only used for computing the gradients.

10.1.5.1 Chain rule

The chain rule is the core of backpropagation:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{10.14}$$

and for vector functions the rule is:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} \tag{10.15}$$

The equivalent in vector notation is:

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^T \nabla_y z \tag{10.16}$$

with $\frac{\partial y}{\partial x}$ the $n \times m$ Jacobian matrix of g.

10.1.5.2 Backpropagation algorithm

Forward step

Require: Network depth l
Require: $W^{(i)}, i \in \{1, \ldots, l\}$ weight matrices
Require: $\mathbf{b}^{(i)}, i \in \{1, \ldots, l\}$ bias parameters
Require: $\mathbf{x}$ input value
Require: $\mathbf{t}$ target value

$h^{(0)} = \mathbf{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{\alpha}^{(k)} = \mathbf{b}^{(k)} + W^{(k)}\mathbf{h}^{(k-1)}$
  $\mathbf{h}^{(k)} = f(\boldsymbol{\alpha}^{(k)})$
end for
$y = \mathbf{h}^{(l)}$
$J = L(\mathbf{t}, \mathbf{y})$

Figure 10.1: Forward step

Backward step

$\mathbf{g} \leftarrow \nabla_{\mathbf{y}} J = \nabla_{\mathbf{y}} L(\mathbf{t}, \mathbf{y})$

for $k = l, l-1, \ldots, 1$ **do**

Propagate gradients to the pre-nonlinearity activations:

$\mathbf{g} \leftarrow \nabla_{\boldsymbol{\alpha}^{(k)}} J = \mathbf{g} \odot f'(\boldsymbol{\alpha}^{(k)})$ {$\odot$ denotes elementwise product}

$\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g}$

$\nabla_{W^{(k)}} J = \mathbf{g}(\mathbf{h}^{(k-1)})^T$

Propagate gradients to the next lower-level hidden layer:

$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = (W^{(k)})^T \mathbf{g}$

end for

Figure 10.2: Backward step

Note: this algorithm is specific for MLPs. This algorithm can also be optimized through caching.

10.1.6 Learning Algorithms

10.1.6.1 SGD

Require: Learning rate $\eta \geq 0$

Require: Initial values of $\boldsymbol{\theta}^{(1)}$

$k \leftarrow 1$

while stopping criterion not met **do**

Sample a subset (minibatch) $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ of m examples from the dataset $\mathcal{D}$

Compute gradient estimate: $\mathbf{g} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}^{(k)}), \mathbf{t}^{(i)})$

Apply update: $\boldsymbol{\theta}^{(k+1)} \leftarrow \boldsymbol{\theta}^{(k)} - \eta \mathbf{g}$

$k \leftarrow k + 1$

end while

Figure 10.3: SGD algorithm. The gradient is calculated through the backpropagation.

We change η according to some rules through iterations.

Until iteration τ:

$$\eta^{(k)} = \left(1 - \frac{k}{\tau}\right)\eta^{(k)} + \frac{k}{\tau}\eta^{(\tau)} \tag{10.17}$$

after iteration τ we have $\eta^{(k)} = \eta^{(\tau)}$

10.1.6.2 SGD with momentum

Momentum can accelerate learning
Motivation: Stochastic gradient can largely vary through the iterations

Require: Learning rate $\eta \geq 0$
Require: Momentum $\mu \geq 0$
Require: Initial values of $\boldsymbol{\theta}^{(1)}$
  $k \leftarrow 1$
  $\mathbf{v}^{(1)} \leftarrow 0$
  while stopping criterion not met **do**
    Sample a subset (minibatch) $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ of m examples from the dataset D
    Compute gradient estimate: $\mathbf{g} = \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}^{(k)}), \mathbf{t}^{(i)})$
    Compute velocity: $\mathbf{v}^{(k+1)} \leftarrow \mu\mathbf{v}^{(k)} - \eta\mathbf{g}$, with $\mu \in [0, 1)$
    Apply update: $\boldsymbol{\theta}^{(k+1)} \leftarrow \boldsymbol{\theta}^{(k)} + \mathbf{v}^{(k+1)}$
    $k \leftarrow k + 1$
  end while

Figure 10.4: SGD algorithm with momentum. The gradient is calculated through the backpropagation.

Momentum helps overcome local minimums. It starts an oscillation movement when he reaches a minimum.

SGD with **Nestov** momentum is another technique that consists in applying the momentum before computing the gradient. This technique sometimes improves the convergence rate.

10.1.6.3 Adaptive learnings

Based on th eanalisys of the gradient of the loss function it's possible to determine, at any step algorithm, whether the learning rate should be increased or decreased.

Some examples of adaptive learning rates are:

- AdaGrad

- RMSProp
- Adam

10.1.7 Architectural Design

What can we chose of a FNN?

- Depth
- Width of a layer
- Activation functions
- Loss function

The **Universal Approximation Theorem** does not say how to set these parameters.

For the output of the neural network we can use the softmax if we have to classify the datapoint. The softmax will be translated into a one-hot encoding (1, 0, 0, 0, 1, 0, ...) or in a Categorical encoding (1, 2, 3).
for the loss function we have two formats of the output: Categorical Cross Entropy vs Sparse categorical cross entropy.

How do we calculate the number of parameters for a Fully connected layer? if we have a layers as input and the next layer has b units we can calculate the number of parameters as: $a \times b + b = (a + 1) \times b$. We can calculate the number of parameters for each Fully connected layer and sum the values to get the total number of parameters.

10.1.8 Regularization

Regularizatoin is an important step for FNN too if we want to reduce overfitting.
We have different options for FNN:

- Parameter norm penalties
- Dataset argumentation
- Early stopping
- Parameter sharing
- Dropout

If the network improves the accuracy on Training set but not on Test set then the network is overfitting.

10.1.8.1 Parameter norm penalties

We can penalize high absolute values of the parameters:

$$E_{reg}(\Theta) = \sum_j |\Theta_j|^q \tag{10.18}$$

resulting in the cost function:

$$\overline{J}(\Theta) = J(\Theta) + \lambda E_{reg}(\Theta) \tag{10.19}$$

Programming Libraries usually let you choose a λ for each layer. If the value is not set, il will be set to 0 and the parameters will not be penalized.

10.1.8.2 Dataset Argumentation

We can generate new datapoint and include them in the dataset.

- Data Transformation
- Adding noise

10.1.8.3 Early Stopping

We can stop iterations early to avoid overfitting. In order to know when to stop we can use cross-validation to determine the best values. We can also send different training run with random initialization values, where each run will evolve differently an maybe one of them will have a better accuracy.

10.1.8.4 Parameter sharing

Parameter sharing consists in setting constraints on having subset of model parameters to be equal.

In CNNs this method allows for invariance to translation.

10.1.8.5 Dropout

Randomly ignore network units with some probability α. So I will apply the algorithm only on a subset of connection (for a single step). I can apply dropout to every step an randomly ignore a different random subset of parameters at each iteration.

10.1.9 Convolutional Neural Networks

10.1.10 Motivation

Up to now we treated inputs as general feature vectors. In some cases they have a special structure:

- Audio

- Images
- Videos

Signals are a numerical representation of physical quantities. Deep learning can be directly applied on signals by using suitable operations.

We cannot simply flatten the input because images, audio and videos have a semantics that cannot be ignored. By flatting the input we loose information and consequently the network will not behave well.

10.1.10.1 Convolution

Given an image I and a kernel K, the 2-D convolution is:

$$(I * K)(i, j) \equiv \sum_{m \in S_1} \sum_{n \in S_2} I(m, n) K(i - m, j - n) \tag{10.20}$$

where S_i are finite sets. for a 3-D convolution the formula is:

$$(I * K)(i, j, k) \equiv \sum_{m \in S_1} \sum_{n \in S_2} \sum_{u \in S_3} I(m, n, u) K(i - m, j - n, k - u) \tag{10.21}$$

Properties:

- convolution is commutative: $(I * K)(i, j) = (K * I)(i, j)$
- Cross-correlation: we can flip the kernel and modify the convoution

$$(I * K)(i, j) \equiv \sum_{m \in S_1} \sum_{n \in S_2} I(i + m, j + n) K(m, n) \tag{10.22}$$

10.1.10.2 Different types of Convolution

based on the direction of the movement of the kernel, the convolution can be 1D, 2D, 3D...
So it is based on the degree of freedom of the kernel.

Convolution changes the shape of the tensor.

10.1.10.3 Padding

We can add a padding in order to convolve and keep the size of the image. Padding can be all zeros

10.1.11 Terminology

- Input size
- Kernel size

- Feature map or Depth slice: output of convolution between an input and one kernel
- Depth: number of kernels
- Padding
- Stride: step of sliding kernel
- Receptive field: region in the input space that a particular feature is looking at.

10.1.11.1 Convolutional Layers

Are usually made up by three layers:

- Convolution between input and kernel
- non-linear activation function
- pooling

For the detector stage we use one of the same that we use for FNNs.

10.1.11.2 Properties of CNNs

Sparse interaction/connectivity: brings in the concept of locality. Parameter sharing: constraint on values of the kernels.

Pooling stage implements invariance to local translations. Some frequently used pooling layers are **maxpooling** and **average pooling**. When applied with stride it reduces the size of the output layer (**subsampling**).

10.1.11.3 Calculate the feature map size

Consider the input size $w_{in} \times h_{in} \times d_{in}$, d_{out} kernels of size $w_k \times h_k \times d_{in}$, stride s and padding p, the dimension of the feature map can be calculated as:

$$w_{out} = \frac{w_{in} - w_k + 2p}{s} + 1 \tag{10.23}$$

$$h_{out} = \frac{h_{in} - k_k + 2p}{s} + 1 \tag{10.24}$$

while the number of trainable parameters is:

$$|\Theta| = w_k \cdot h_k \cdot d_{in} \cdot d_{out} + d_{out} \tag{10.25}$$

10.1.12 Networks for Images

LeNet is a simple but yet good neural network composed of 7 layers. The input is a 32×32 image and the output are 10 neurons (10 classes).

Other famous neural networks (not deep) are:

- ResNet
- VGG
- GoogLeNet
- AlexNet

10.1.13 Transfer Learning

The idea is to understand how to exploit learning on a different task for learning on a new task.
D is a domain defined by data points $x_i \in X$, distributed according to $D(X)$. T is a learning task defined by labels $y \in Y$, a target function $f : X \rightarrow Y$, and distribution $P_D(y|x)$.
Given:

- D_T and T_S a source domain learning task
- D_T and T_T a target domain learning task
- in general $D_S \neq D_T$ and $T_S \neq T_T$

Our goal is to improve learning of $f_T : X_T \rightarrow Y_T$ using knowledge in D_S and DT_S.

10.1.13.1 Solutions

Fine-tuning: Use the same architecture (or part of it) and initialize it with the pretrained models. Then you can:

- train of all network parameters
- freeze parameters of some layers.
  - **Pros**: Full advantage of CNNs.
  - **Cons**: 'Heavy' training.

CNN as feature extractor: We use a CNN as a feature extractor network, usually by cutting the last layers and by then using a flatten/dense layer. Then we train a new classifier based on those extracted features.

- **Pros**: No need to train CNNs.
- **Cons**: Cannot modify features

Chapter 11

Unsupervised Learning

11.1 Unsupervised Learning

We want to learn a distribution without knowing the labels. This is considered a more complex and difficult problem than supervised learning.

$$\begin{aligned} f &: X \rightarrow Y \\ D &= \{(x_n)\} \end{aligned} \tag{11.1}$$

11.1.1 Gaussian Mixture Model

We assume that the dataset is generated by a probability distribution which is the sum of different Gaussian functions:

$$P(x) = \sum_{k=1}^{k} \pi_k N\left(x; \mu_k, \Sigma_k\right) \tag{11.2}$$

Where π_k is the prior probability, μ_k is the mean and Σ_k is the covariance matrix. When we have these information we can easily generate datapoints by sampling the distribution. We firstly select one of the Gaussians based on the probability π and then we sample it.

11.1.2 K-means

Now that we know how to sample from a GMM, we can try to find the means of the distributions.

The K-means algorithm tries to find the K centroids of the K Gaussians.
The iterative process of the algorithm is the following:

1. Begin with a decision on the number of clusters (k)
2. Initialize the positions of the means. You may assign the training samples randomly, or systematically as follows

- Take the first k training samples as single element cluster
- Assign each of the remaining (N - k) training samples to the cluster with the nearest centroid. After each assignment, recompute the centroid of the new cluster.

The algorithm stops when there are no changes in the assignment.
The algorithm converges because:

- At each switch in step 2, the distance between the points and the centroids is decreased.
- There are only a finite number of partitions that assign data points to k clusters.

Cons of K-means:

- The number of K must be decided before hand. There are algorithms that tries to find the best K.
- Sensitive to initial condition (local optimum) when a few data available.
- Not robust to outliers
- the result is a circular cluster shape because it is based on distance.

Some improvements to K-means:

- Use it only if there are many data available
- use median instead of mean
- define better distance functions.

11.1.3 Predict GMM

A different way of predicting the GMM is by introducing a latent variable $z_k \in \{0, 1\}$., with $z = (z_1, \ldots, z_k)$ and 1-out-of-K encoding.
Let's define

$$P(z_k = 1) = \pi_k \tag{11.3}$$

$$P(z_k) = \sum_{k=1}^{K} \pi_k^{z_k} \tag{11.4}$$

For a given value of z:

$$P(x|z_k = 1) = N(x; \mu_k, \Sigma_k) \tag{11.5}$$

thus

$$P(x|z) = \prod_{k=1}^{K} N(x; \mu_k, \Sigma_k)^{Z_k} \tag{11.6}$$

Joint distribution: $P(x, z) = P(x|z)P(z)$ (chain rule).
When z are variables with 1-out-of-K encoding and $P(z_k = 1) = \pi_k$,

$$P(x) = \sum_z P(z)P(x|z) = \sum_{k=1}^{K} \pi_k N(x; \mu_k, \Sigma_k) \tag{11.7}$$

GMM distribution can be seen as the maximization of $P(x, z)$ over variables z. By reversing the equation 11.7 we can learn the GMM.

Let's define the **posterior**

$$\gamma(z_k) \equiv P(z_k = 1|x) = \frac{P(z_k = 1)P(x|z_k = 1)}{P(x)} = \frac{\pi_k N(x; \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j N(x; \mu_j, \Sigma_j)} \tag{11.8}$$

11.1.4 Expectation Maximization (EM)

It is a general iterative algorithm based on two steps: **Expectation** and **Maximization**. The algorithm computes the maximum likelihood

$$\underset{params}{argmax} P(X|params) \tag{11.9}$$

Expectation maximization consists of:

- **E step**: Given π_k, μ_k, and Σ_k compute $\gamma(Z_{nk})$
- **M step**: Given $\gamma(Z_{nk})$, compute π_k, μ_k, and Σ_k

11.1.4.1 EM for GMM

$$\begin{aligned} \mu_k &= \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) x_n \\ \Sigma_k &= \frac{1}{N_n} \sum_{n=1}^{K} \gamma(z_{nk})(x_n - \mu_k)(x_n - \mu_k)^T \\ \pi_k &= \frac{N_k}{N}, \text{ with } N_k = \sum_{n=1}^{N} \gamma(z_{nk}) \end{aligned} \tag{11.10}$$

The algorithm for GMM:

- Initialize $\pi_k^{(0)}, \mu_k^{(0)}$, and $\Sigma_k^{(0)}$
- repeat until termination condition:
  - **E step**

$$\gamma(z_{nk})^{(t+1)} = \frac{\pi_k^{(t)} N(x; \mu_k^{(t)}, \Sigma_k^{(t)})}{\sum_{j=1}^{K} \pi_j^{(t)} N(x; \mu_j^{(t)}, \Sigma_j^{(t)})} \tag{11.11}$$

– **M step**

$$\mu_k^{(t+1)} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})^{(t+1)} x_n$$

$$\Sigma_k^{(t+1)} = \frac{1}{N_n} \sum_{n=1}^{K} \gamma(z_{nk})^{(t+1)} (x_n - \mu_k^{(t+1)})(x_n - \mu_k^{(t+1)})^T \tag{11.12}$$

$$\pi_k^{(t+1)} = \frac{N_k}{N}, \text{ with } N_k = \sum_{n=1}^{N} \gamma(z_{nk})^{(t+1)}$$

Notice that if we consider only μ, this is the same as K-means.

11.1.4.2 General EM problem

Given observed data, an unobserved latent variable and a parameterized probability distribution $P(Y|\Theta)$, where Θ are the parameters

Em in general converges to a local optimum. It provides an estimation for the latent variables.

EM has many uses in Unsupervised clustering, Bayesian Networks and Hidden Markov Models.

The method is the following: Given a likelihood function $Q(\Theta'|\Theta)$ which calculates $Y = X \cup Z$, the algorithm is:

1. **Estimation step**: Calculate $Q(\Theta'|\Theta)$ using current hypothesis Θ and observed data X to estimate probability distribution over Y

$$Q(\Theta'|\Theta) \leftarrow E[\ln P(Y|\Theta')|\Theta, X] \tag{11.13}$$

2. **Maximization step**: Replace hypothesis Θ by the hypothesis Θ' that maximizes the Q function

$$\Theta \leftarrow \underset{\Theta'}{argmax} Q(\Theta'|\Theta) \tag{11.14}$$

Chapter 12

Bayesian Network

12.1 Bayesian Network

The syntax of a Bayesian network is the following:

- a set of nodes, one per variable
- a direct, **acyclic** graph
- a conditional distribution for each node given its parents.

It is a Probability Graphical Model (PBG). We use a Condition Probability Table (CPT) to represent causes and effects. Given the parents of the node X_i, he is independent from all the other variables. I only have to express $P(X_i|Parents(X_i)$.

Consider a set of rules: $A \rightarrow C$, $B \rightarrow C$, $B \rightarrow E$, $C \rightarrow E$, $C \rightarrow D$. Instead of having $P(A, B, C, D, E)$, I can use: $P(A)$, $P(B)$, $P(C|A, B)$, $P(E|B, C)$ and $P(D|C)$. We can use the chain rule to calculate probabilities:

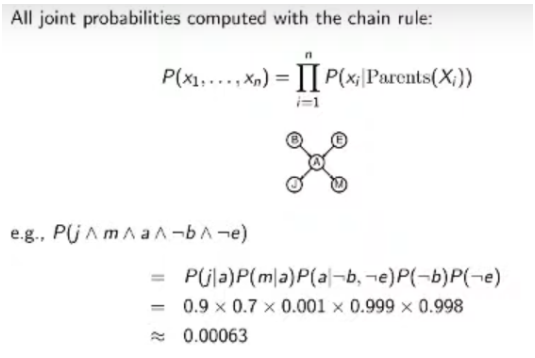


Figure 12.1: Caption

We can use the parameters $\Theta =< \Theta_1, \ldots, \Theta_n >$ and calculate the inference. In machine learning we are interested in learning these parameters given a dataset.

If we can observe every variable it is easy since we can count the number of times that each variable is true/false and infer probabilities:

$$P(A = 1|B = 0) \approx \frac{|\{d_k = \langle a_k, x_k, \dots \rangle\}|a_k = 1 \ \ \& \ \ x_k = 0|}{|\{d_k|x_k = 0|\}} \tag{12.1}$$

If we have an unknown variable X we can define $\Theta_1 \dots \Theta_n$ as $P(X = 0) = \Theta_0$, $P(A = a_1|X = 0) = \Theta_1$, $P(A = a_1|X = 1) = \Theta_2$, $P(B = b_1|X = 0) = \Theta_3$, $P(B = b_1|X = 1) = \Theta_4$, $\Theta = \langle \Theta_0, \Theta_1, \Theta_2, \Theta_3, \Theta_4 \rangle$ and use EM algorithm to maximize Θ from $D = \{(a_1, b_1), \dots, (a_k, b_k)\}$.

Estimation:

$$\begin{aligned} P(X = x_j) &= \frac{1}{N} E[\hat{N}(X = x_j)] \\ P(A = a_j|X = x_j) &= \frac{E[\hat{N}(A = a_j, X = x_j)]}{E[\hat{N}(X = x_j]} \end{aligned} \tag{12.2}$$

Maximization:

$$E[\hat{N}(\cdot)] = E\left[\sum_k I(\cdot|d_k)\right] = \sum_k P(\cdot|d_k) \tag{12.3}$$

Chapter 13

Dimensionality

13.1 Dimensionality Reduction

13.1.1 Latent Variables

Sometimes not every information that we have from the input is important. Data may lie in a different lower dimension.

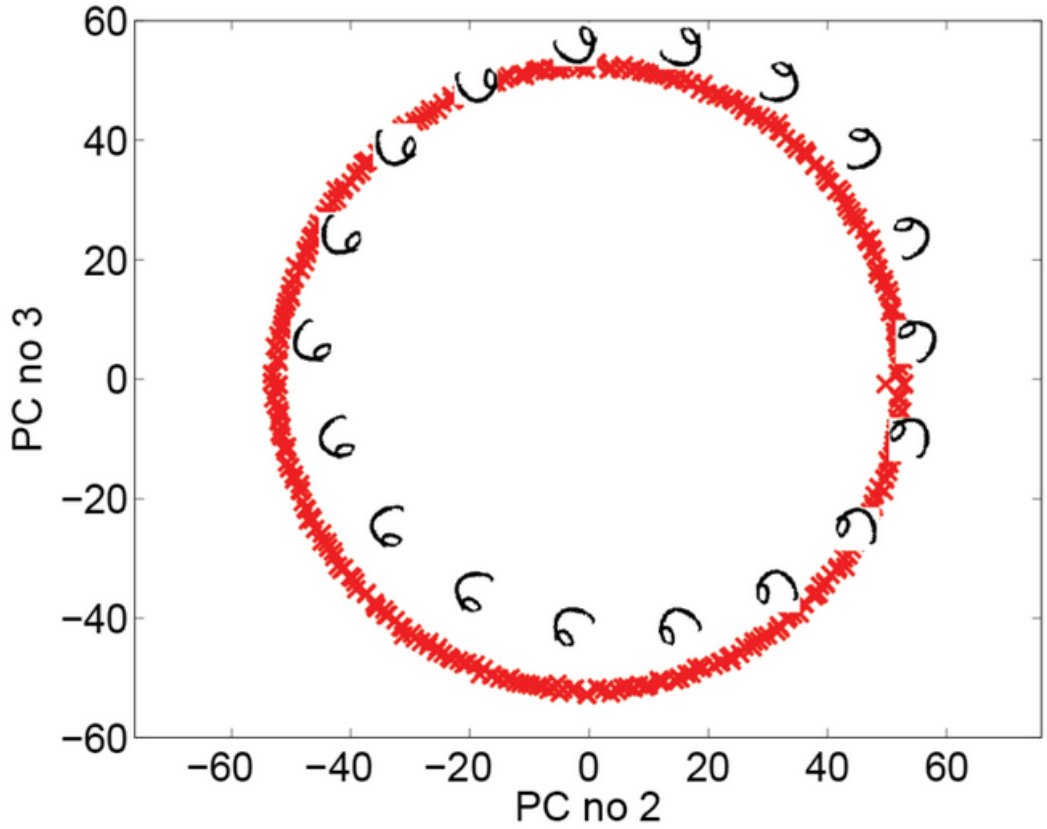


Figure 13.1: Manifold that represents the rotation of a digit

This is usually possible for images since they have a structure and a semantic segmentation.

For data with structure we expect fewer distortions than dimensions and data usually live on a lower dimensional manifold. As a conclusion, we can deal with high dimensional data by looking for lower dimensional embeddings.

13.1.1.1 PCA

PCA is a method for dimension reduction that aims to maximize data variance after projection to some direction u_1. Projection points are:

$$u_1^T x_n \tag{13.1}$$

note that $u_1^T u_1 = 1$

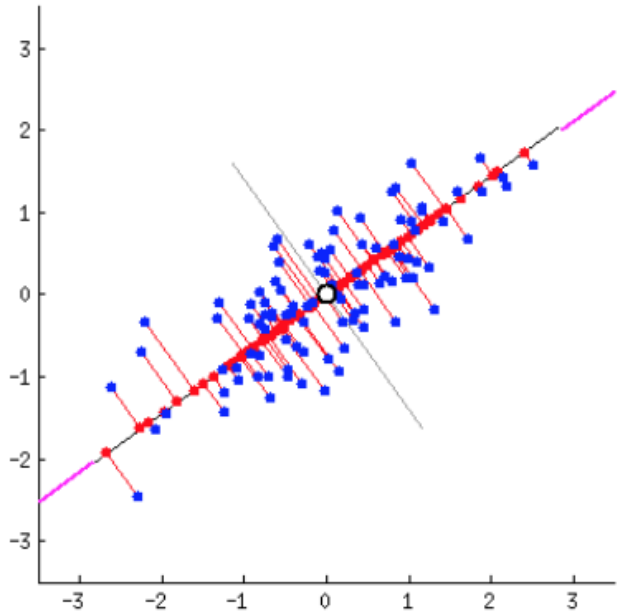


Figure 13.2: PCA variance maximization

The first thing that we do is find the mean value of datapoints

$$\overline{x} = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{13.2}$$

and the data-centered matrix is:

$$X = \begin{bmatrix} (x_1 - \overline{x})^T \\ \cdots \\ (x_n - \overline{x})^T \\ \cdots \\ (x_N - \overline{x})^T \end{bmatrix} \tag{13.3}$$

PCA is able to reduce **ONLY** to linear transformations. There are some techniques that are able to find non linear projections. Now that data is normalized, we can find the linear projection that maximize the data. The projection point can be calculated as shown in 13.1. If we apply the projection on every point, we have

$$D = \langle u_1^T x_n, \ldots, u_1^T x_n, \ldots, u_D^T x_n \rangle \tag{13.4}$$

now that we know how to apply the projection of points on a direction, we have to find which are the best directions. the mean of the projected points is $u_1^T \overline{x}$ and the variance of the projection points is:

$$\frac{1}{N}\sum_{n=1}^{N} [u_1^T x_n - u_1^T \overline{x}]^2 = u_1^T S u_1 \tag{13.5}$$

with

$$S = \frac{1}{N}\sum_{n=1}^{N}(x_n - \overline{x})(x_n - \overline{x})^T = \frac{1}{N}X^T X \tag{13.6}$$

note: S (13.6) is equal to the covariance normalized matrix, while 13.5 is just a matrix multiplication on left and right of S by u_1 (the left one is transposed).

If we try to maximize this value we get:

$$\max_{u_1} u_1^T S u_1 \tag{13.7}$$

subject to $u_1^T u_1 = 1$. This is equivalent to unconstrained maximization with a Lagrange multiplier

$$\max_{u_1} u_1^T S u_1 + \lambda_1(1 - u_1^T u_1) \tag{13.8}$$

and the solution is reached by setting the derivative w.r.t. u_1 to zero:

$$Su_1 = \lambda_1 u_1 \tag{13.9}$$

where u_1 must be an eigenvector if S.

Left-multiplying by u_1^T and using $u_1^T u_1 = 1$, we have

$$u_1^T S y_1 = \lambda_1 \tag{13.10}$$

which is the variance after the projection. In general, S will have different eigenvalues and eigenvectors. The eigenvalues that maximize the projections are the ones with an higher value. so, the first vector that will maximize the projection is the one associated to the highest eigenvalue, the second one will be the one associated to the second highest value, and so on.

PCA Error minimization: Consider a complete orthonormal D-dimensional basis such that

$$u_i^T u_j = \delta_{ij} \tag{13.11}$$

with

$$\delta_{ij} = \begin{cases} 1 \text{ if } i = j \\ 0 \text{ otherwise} \end{cases} \tag{13.12}$$

Each datapoint can be written as

$$x_n = \sum_{i=1}^{d} \alpha_{ni} u_i \tag{13.13}$$

using the orthonormal property we have $\alpha_{ni} = x_n^T u_i$, hence

$$x_n = \sum_{i=1}^{d} (x_n^T u_i) u_i \tag{13.14}$$

the goal is to approximate x_n using a lower-dimensional representation. We can write

$$\tilde{x}_n = \sum_{i=1}^{m} z_{ni} u_i + \sum_{i=m+1}^{d} b_i u_i \tag{13.15}$$

Note: b_i terms do not depend on sample x_n.
Evaluate approximation error (MSE) as

$$J = \frac{1}{N} \sum_{n=1}^{N} ||x_n - \tilde{x}_n||^2 \tag{13.16}$$

Minimizing wrt z_{nj} and b_i, we get:

$$\begin{aligned} z_{ni} &= x_n^T u_i, i = 1, \ldots, m \\ b_i &= \overline{x}^T u_i, i = M+1, \ldots, d \end{aligned} \tag{13.17}$$

Using these expression we get

$$x_n - \tilde{x}_n = \sum_{i=m+1}^{d} [(x_n - \overline{x})^T u_i] u_i \tag{13.18}$$

The overall approximation error becomes

$$J = \frac{1}{N} \sum_{n=1}^{N} \sum_{i=m+1}^{d} (x_n^T u_i - \overline{x}^T u_i)^2 = \sum_{i=m+1}^{d} u_i^T S u_i \tag{13.19}$$

Minimize the approximation error subject to constraint $u_i^T u_i = 1$:

$$\tilde{J} = \sum_{i=m+1}^{d} u_i^T S u_i + \lambda_i (1 - u_i^T u_i) \tag{13.20}$$

If we set the derivative of a u_i to zero, we get the exact same mathematical problem that we have found before:

$$S u_i = \lambda_i u_i \tag{13.21}$$

hence u_i is an eigenvector of S with eigenvalue λ_i. The approximation error is then given by:

$$J = \sum_{i=m+1}^{d} \lambda_i \tag{13.22}$$

and this is minimized by selecting u_i as the eigenvectors corresponding to the $d - m$ smallest eigenvalues. In other words, the error is the sum of those last eigenvectors that we do not consider.

If we have less samples than dimensions, PCA is inefficient. a trick that we can use in this case is letting X be the $N \times d$ centered data matrix (i.e., n-th row is $(x_n - \overline{x})^T$) and the corresponding covarinace matrix:

$$S = \frac{1}{N} X^T X \tag{13.23}$$

the corresponding eigenvector equation is

$$\frac{1}{N} X^T X u_i = \lambda_i u_i \tag{13.24}$$

By left multiplying by X we obtain:

$$\frac{1}{N} X X^T (X u_i) = \lambda_i (X u_i) \tag{13.25}$$

and if we define $v_i = X u_i$ we have

$$\frac{1}{N} X X^T v_i = \lambda_i v_i \tag{13.26}$$

so we have XX^T which has the same $N - 1$ eigenvalues of $X^T X$ and is a $N \times N$ matrix whose eigenvalues can be computed efficiently.

given the eigenvalues λ_i of XX^T, to find the eigenvectors we left-multiply by X^T

$$\left(\frac{1}{N} X X^T\right) \left(X^T v_i\right) = \lambda_i \left(X^T v_i\right) \tag{13.27}$$

This makes clear that $\left(X^T v_i\right)$ is an eigenvector of S with eigenvalue λ_i. To find the direction u_i we have to normalize the eigenvectors such that $u_i^T u_i = 1$:

$$u_i = \frac{1}{\sqrt{N\lambda_i}} X^T v_i \tag{13.28}$$

Linear latent variable model:

- Represent data x with lower dimensional latent variables z
- Assume linear relationship $x = Wx + \mu$
- Assume Gaussian distribution of latent variables z $P(z) = N(z; 0, I)$
- Assume Linear-Gaussian relationship between latent variables and data $P(x|z) = N(x; Wz + \mu, \sigma^2 I)$

So, **Probabilistic PCA** is a model that, given the dataset, estimates the parameters of the models (W, z and σ). we can setup a Maximum Likelihood that, given data X, sets the derivative of

$$\underset{W,\mu,\sigma^2}{argmax} \ln P(X|W, \mu, \sigma^2) = \sum_{n=1}^{N} \ln P(x_n|W, \mu, \sigma^2) \tag{13.29}$$

to zero and finds the parameters of the model that, given z will generate a sample x.

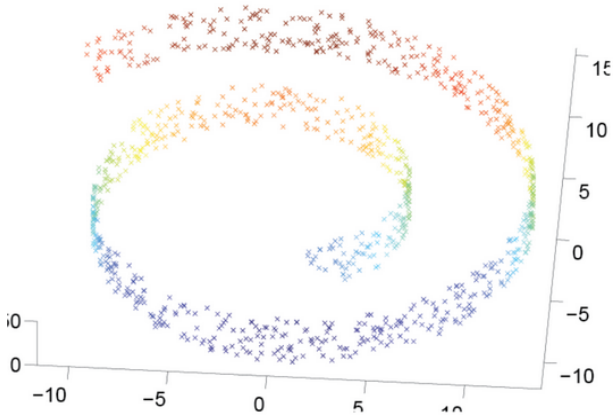


Figure 13.3: The 'Swiss Roll' dataset. 2D manifold embedded in 3D space.

13.1.2 non-Linear Latent Variable Models

Sometimes, linear representations are not sufficient for complex data

13.1.2.1 Autoassociative Neural Networks (Autoencoders)

An autoencoder is a combination of two Neural Networks: an encoder and a decoder. The train is based on reconstruction loss and provides a low-dimensional representation.

The structure is a neural network with a reduced size of hidden layers which learn to reconstruct their input by minimizing a loss function.

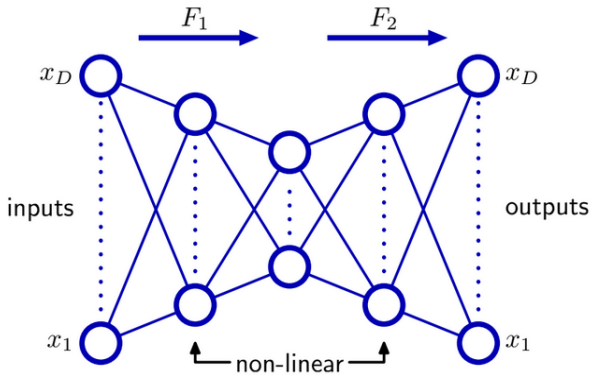


Figure 13.4: A basic autoencoder example.

It is important to have the same number of neurons in input and output.

We can also use autoencoders for image based application with Convolutional and Convolutional Transposed layers

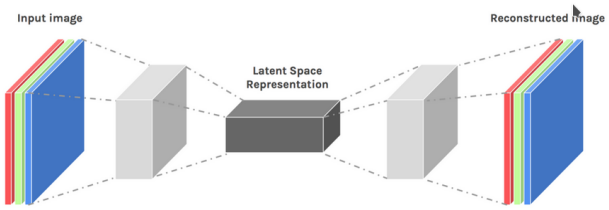


Figure 13.5: Autoencoder for images.

Given a dataset x_n, Autoencoders are trained with the same sample x_n both in input and in output. Autoencoders learn how to encode/decode the samples in a dataset in a low-dimensional space. Autoencoders can be seen as a method for non-linear principal component analysis.

Autoencoders can be used for anomaly detection. We can teach an Autoencoder to reconstruct "good" samples and then calculate a threshold based on the final train loss found during training. We can then test samples, see if the reconstruction error is higher than the threshold and, if so, consider that sample an anomaly.

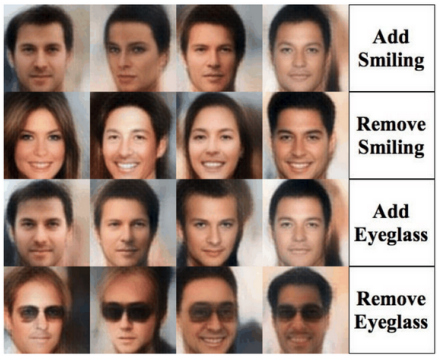
13.1.3 Generative Models

For these models, the aim is not only to reduce dimensionality, but also to use the latent space to reconstruct the input and modify it.

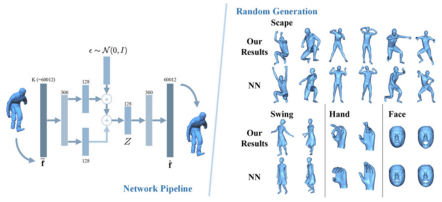
Variational Auto-Encoders (VAEs) focus on learning latent space structure
Generative Adversal Networks (GANs) focus on learning a distribution (no latent space in general)

13.1.3.1 VAEs

The goal is to modify the data in specific directions and identify meaningful directions in latent space. Some examples are face distortion, digit production and 3D mesh distortion.



(a) Example of application of VAEs on faces.



(b) Example of application of VAEs on 3D mesh fold.

Main idea: Encoder produces a distribution instead of a vector. Decoder operates on samples from this distribution

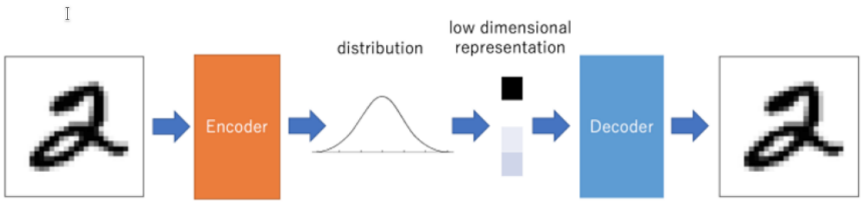


Figure 13.7: VAE architecture.

But how do we produce a distribution? How do we prevent degeneration? We can consider a parametric distribution (typically Gaussian that produces a mean and a variance). Then we add a loss term based on the KL divergence (Evidence Lower Bound) that gives how much the distribution of the latent space differs from a standard Gaussian distribution with 0 mean and an identity covariance matrix. We also have another factor ϵ that works as a weight and regulates the strictness of the re-parametrization of the Gaussian that the VAE is learning (an high ϵ brings better looking Gaussians but worsen the reconstruction quality).

In practice, since we have a distribution, we could try to sample one value of the distribution. For instance, if we have an image and we give it to the decoder of a VAE, we get a value as output. Then, the value is processed by the Gaussian distribution and this generates a sample. This sample is then passed to the decoder who generates the output.

The sampling operation is easy to perform forward given the parameters of the Gaussian. However, if we have to compute the gradient for the backpropagation, and the sampling operation is not invertible. We then solve this problem by introducing the concept of generating a sample during backpropagation and sum this value to the output of the encoder. Since this can be treated just like a constant value, during backpropagation we know how to derive it and so we can proceed.

13.1.3.2 GANs

The main goal is to sample from the input data distribution X. The idea is based on inverting a CNN and use adversal training.

A GAN firstly has an encoder that uses a CNN to process an image and transform it into a vector. Following, a Decoder receives a code and produces an image. This second element uses a "deconvolutional" operator.

What we then want to do is being able to use only the decoder so that, if we give a vector as input, it is able to reconstruct an image. The meaning of dimensionality reduction is loosing importance and what really matters is the generation of an image from a vector.

now the problem becomes: How to train the decoder to produce meaningful data?

Adversal training is when two parts of the network are competing each other in order to find an optimal solution. In GANs, we use adversal training to improve reconstruction. A GAN is then a combination of two networks: the generator network (or decoder) and the discriminator network (or critic).

The Discriminator can take as input both real images from the dataset and images sampled from the generator. He then has to decide if the input is a real or a fake image.

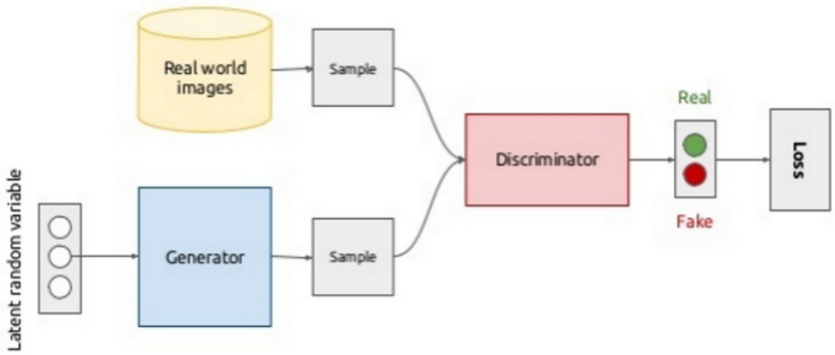


Figure 13.8: GAN architecture.

So, there are two roles in a GAN:

- Generator produces samples of the distribution $P(X)$
- Discriminator identifies if a sample actually comes from the (unknown) $P(X)$ or not

during training procedure, we have the networks competing with each other:

- generator tries to fool the discriminator in believing that the sample is 'real'
- discriminator tries to discriminate as good as possible 'real' from 'fake' samples

and the training steps are:

1. Train the discriminator by keeping the generator fixed. During this phase, images from the generator are labeled as fake because the discriminator has to learn the difference between fake and real images.

   AKA. Train the discriminator with a batch of data $\{(x_n, \text{Real})\} \cup (x'_m, \text{Fake})$, where x_n comes from the data set, while x'_m are images generated from the generator with random values of the latent variable.

2. Train the generator by keeping the discriminator fixed. During this phase, images from the generator are labeled as real because the generator wants to fool the discriminator.

   AKA. Train the generator by using the entire model (generator + discriminator) with discriminator layers fixed (i.e., not trainable) with a bacth of data (r_k, Real), where r_k are random values of the latent variable

What about the loss? We can notice that when the loss increases for the discriminator, it decreases for the generator and viceversa. Very often, if you plot the losses during time it will have a 2d-DNA shape that slowly decreases until they will converge. We don't want to have that one of the two losses goes to zero while the other diverges. A model is trained when the two losses converge and the accuracy of the discriminator is around 50%. Now we can generate images from the generators that are similar to the ones from the dataset.

Additional topic: GANs can be exploited to attack ML models !!! A generative model can be defined in order to make an anomaly detection model fail.

Chapter 14

Reinforcement Learning

14.1 Dynamic Systems

A dynamic system is a system that evolves during time. Until now, we didn't consider time. Now we will consider a system that also considers time and, in particular, discrete time.

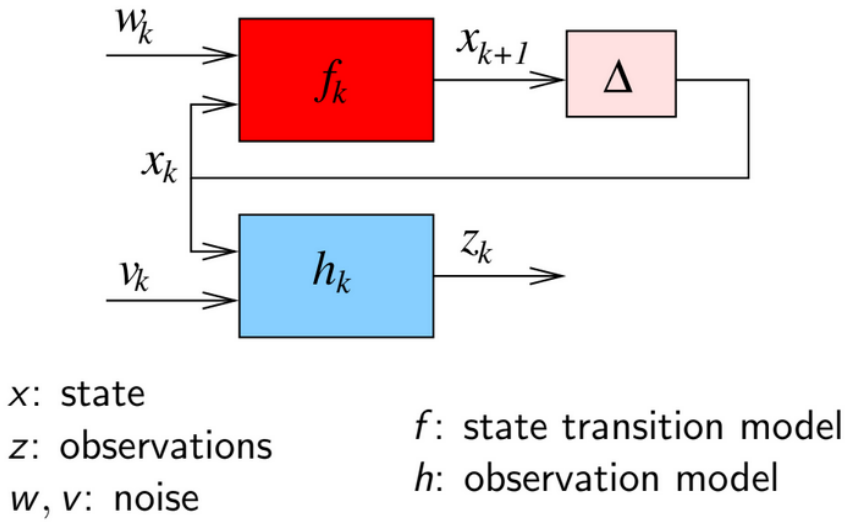


Figure 14.1: Taxonomy ao a dynamic system

An important step in the dynamic system 14.1 is X_k which is the snapshot f how is the situation at this moment, while X_{k+1} is the snapshot at the next timestep. The evolution can be represented by a list of snapshots: $X_1, X_2, \ldots, X_t$. The state transmission model f_k is the function that evolves the system. This function models how the state changes between steps. In this picture, the input variables that influences this function are not shown. However they are implicitly represented inside the noise W_k. We have another important model which is the observation model h_k. z_k are the

observations created by the observation model. Δ is an offset in time that gives to the model consideration of time.

14.1.1 Reasoning vs learning in Dynamic Systems

For **reasoning**, we have the model (f, h) (Input) and the current state x_k, so we can predict the future (x_{k+T}, z_{k+T}). In **learning** we have the past experience $(z_{0:k})$ (Input) and we want to determine the model (f, h).

14.1.2 State of a Dynamic System

The staet x encodes:

- all the past knowledge needed to predict the future
- the knowledge gathered through operation
- the knowledge needed to pursue the goal

When the state is fully observable, the decision making problem for an agent is to decide which action must be executed in a given state. The agent has to compute the function:

$$\pi : X \rightarrow A \tag{14.1}$$

The goal of the agent is to learn a function π that maps states to actions.

If we want to compare Reinforcement Learning and Supervised learning, we can point out that:

- Supervised learning tries to learn a function $f : X \rightarrow Y$, given $D = \{x_i, y_i\}$
- Reinforcement learning tries to learn a function $\pi : X \rightarrow A$, given $D = \{\langle (x_1, a_1, r_1), \ldots, (x_n, a_n, r_n) \rangle\}^{(i)}$

where x_i is the i-th state, a_i is the i-th action and r_i is the i-th reward. Notice that in a RL dataset we have any combination of states, actions and rewards.

14.1.3 Dynamic System Representation

X: set of states

- **explicit discrete and finite representation** $X = \{x_1, \ldots, x_n\}$
- continuous representation $X = F(\ldots)$ (state function)
- probabilistic representation $P(X)$ (probabilistic state function)

A: set of actions

- **explicit discrete and finite representation** $A = \{xa_1, \ldots, a_n\}$

- continuous representation $A = U(\dots)$ (control function)

δ: transition function

- deterministic / non-deterministic / **probabilistic**

Z: set of observations

- explicit discrete and finite representation $Z = \{xz_1, \dots, z_n\}$
- continuous representation $Z = \zeta(\dots)$ (observation function)
- probabilistic representation $P(Z)$ (probabilistic observation function)

14.1.4 Markov property

The Markov property states that:

- Once the **current state is known**, the evolution of the dynamic system does **not depend** on the **history** of states, actions and observations.
- The **current state contains all the information needed** to predict the future.
- **Future** states are **conditionally independent** of past states and past observations given the current state.
- The knowledge about the current state makes past, present and future observations statistically independent.

A Markov process is a process that has the Markov property

14.1.5 Markov Decision Process (MDP)

Markov processes for decision making.
States are fully observable, no need of observations.
Graphical model:

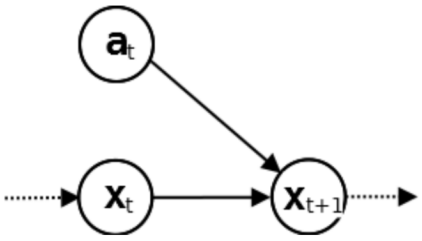


Figure 14.2: MDP graphical model

14.2 Markov Decision Process (MDP)

It is a decision process based on subsequent actions that makes the model evolve during time.

14.2.1 Deterministic transaction

$$MDP = \langle X, A, \delta, r \rangle \tag{14.2}$$

where X is a finite set of states, A is a finite set of actions, $\delta : X \times A \rightarrow A$ is a transition function and $r : X \times A \rightarrow \mathcal{R}$ is a reward function.

The Markov property states that $x_{t+1} = \delta(x_t, a_t)$ and $r_t = r(x_t, a_t)$. Sometimes, the reward function is defied as $r : X \rightarrow \mathcal{R}$.

14.2.2 Non-deterministic transaction

$$MDP = \langle X, A, \delta, r \rangle \tag{14.3}$$

where X is a finite set of states, A is a finite set of actions, $\delta : X \times A \rightarrow 2^X$ is a transition function and $r : X \times A \times X \rightarrow \mathcal{R}$ is a reward function.

We have that the transition function has a set of possible alternatives:

$$\delta(x_t, Q_t) = \begin{cases} 1 \\ x_{t+1} \\ x_{t+4} \\ \dots \\ x_{t+n} \end{cases}$$

An example is a game of tick-tack-toe, where player has a non-deterministic view of the game as it cannot chose for his opponent.

14.2.3 Stochastic transitions

$$MDP = \langle X, A, \delta, r \rangle \tag{14.4}$$

where X is a finite set of states, A is a finite set of actions, δ is modeled as $P(x'|x, a)$ which is a probability distribution over transitions and $r : X \times A \times X \rightarrow \mathcal{R}$ is a reward function.

14.2.4 Fully observability in MDP

States are fully observable. In presence of non-deterministic or stochastic actions, the state resulting from the execution of an action is not known before the execution of the action, but it can be fully observed after its execution.

So, non determinism means that the future is uncertain, not that the state cannot be known.

14.2.5 MDP Solution Concept

Given an MDP, we want to find an optimal policy function $\pi : X \rightarrow A$. For each state $x \in X, \pi(x) \in A$ is the optimal action to be executed in such state.

optimality = maximize the cumulative reward

Optimality is defined with respect to maximizing the (expected value of the) cumulative discounted reward. Discounted because future rewards are less valuable than current rewards (better some money today than some money tomorrow).

$$V^{\pi}(x_1) \equiv E[\overline{r_1} + \gamma\overline{r_2} + \gamma^2\overline{r_3} + \dots] \tag{14.5}$$

where $\overline{r_t} = r(x_t, a_t, x_{t+1})$, $a_t = \pi(x_t)$, and $\gamma \in [0, 1]$ is the discount factor for future rewards.

Optimal policy: $\pi^* \equiv \underset{\pi}{argmax}\, V^{\pi}(x),\ \forall x \in X$. It is the policy that guarantees the maximum value of the value function overall the possible policies for every initial state.

V is called value function. We can distinguish a value function for Deterministic case and Non-deterministic / Stochastic case:

$$\text{Deterministic: } V^{\pi}(x_1) \equiv r_1 + \gamma r_2 + \gamma^2 r_3 + \dots \tag{14.6}$$

$$\text{Non-deterministic / Stochastic: } V^{\pi}(x_1) \equiv E[r_1 + \gamma r_2 + \gamma^2 r_3 + \dots] \tag{14.7}$$

14.2.5.1 Optimal Policy

π^* is an **optimal policy** if and only if for any other policy π:

$$V^{\pi^*}(x) >= V^{\pi}(x), \forall x \tag{14.8}$$

for infinite horizon problems, a stationary MDP always has an optimal stationary policy.

14.3 Reasoning and Learning in MDP

Let's re define the difference between Reasoning and Learning by considering the MDP model. The Probelm is: MDP$\langle X, A, \delta, r\rangle$ and the solution is a policy $pi : X \rightarrow A$. If the MDP$\langle X, A, \delta, r\rangle$ is completely known, we have reasoning or planning, while if it not completely known we have learning. Simple examples of reasoning in MDP can be modeled as a search problem and solved using standard search algorithm (e.g., A*).

14.4 One-state Markov Decision Process

$$MDP = \langle\{x_0\}, A, \delta, r\rangle \tag{14.9}$$

where x_0 is an unique state, A is a finite set of actions, $\delta(x_0, a_i) = x_0,\ \forall a_i \in A$ is the transition function, $r(x_0, a_i, x_0) = r(a_i)$ is the reward function. The optimal policy is $\pi^*(x_0) = a_i$.

We can have different configurations, based on the environment and the actions:

- If $r(a_i)$ is **deterministic** and reward function **known**, then the optimal policy is $\pi^*(x_0) = \underset{a_i \in A}{argmax} r(a_i)$
- If $r(a_i)$ is **deterministic** and reward function **unknown**, then what we can do is:
  1. for each $a_i \in A$, execute the action a_i and collect the reward r_i
  2. the optimal policy is then: $\pi^*(x_0) = a_i$, with $i = \underset{i=1,\ldots,|A|}{argmax}\ r(i)$

  Note that we need $|A|$ iterations.
- If $r(a_i)$ is **non-deterministic** and reward function **known**, then the optimal policy is $\pi^*(x_0) = \underset{a_i \in A}{argmax}\ E[r(a_i)]$.

  For instance, if we have that $r(a_i) = Gauss(\mu_i, \sigma_i)$, then $\pi^*(x_0) = a_i$, with $i = \underset{i=1,\ldots,|A|}{argmax}\ \mu_i$
- If $r(a_i)$ is **non-deterministic** and reward function **unknown**, then what we can do is:
  1. Initialize a data structure Θ
  2. for each time t=1, . . . , T (until termination condition):
     a) choose an action $a_{(t)} \in A$
     b) execute $a_{(t)}$ and collect the reward $r_{(t)}$
     c) update the data structureΘ
  3. the optimal policy is then: $\pi^*(x_0) = \ldots$, according to the data structure Θ

  For example, if we only know that $r(a_i) = N(_i, \sigma_i)$, we can use a vector of N components initialized to zero for the data structure Θ and anther data structure $c[i] \leftarrow 0$, $i = 1, \ldots, |A|$, then at each loop, when we have to update the data structure, we increment $c[\hat{i}]+ = 1$ and update $\Theta_{(t)}[\hat{i}] \leftarrow \frac{1}{c[\hat{i}]}\left(r_{(t)} + \left(r_{(t)} + \left(c[\hat{i}] - 1\right)\right)\Theta_{(t-1)}[\hat{i}]\right)$.

  The optimal policy can then be computed as $\pi^*(x_0) = a_i$, with $i = \underset{i=1,\ldots,|A|}{argmax}\ \Theta_{(T)}[i]$

14.4.1 Experimentation strategies

How can we chose actions? We have to distinguish **exploration** (select a random action) and **exploitation** (select the best action).

Two strategies are explained next.

14.4.1.1 σ-greedy

σ-greedy picks a random action with probability σ and the best action with probability $1 - \sigma$. σ can vary during time and be adaptive, or simply increase during time (first exploration, then exploitation)..

14.4.2 soft-max strategy

action with higher $\hat{Q}$ values are assigned higher probabilities, but every action is assigned a non-zero probability.

$$P(a_i|x) = \frac{k^{\hat{Q}(x,a_i)}}{\sum_j k^{\hat{Q}(x,a_j)}} \tag{14.10}$$

$k > 0$ determines how strongly the selection favors actions with high $\hat{Q}$ values. k may increase over time (first exploration, then exploitation)..

14.4.3 Learning with Markov decision processes

Given an agent accomplishing a task according to an $MDP\langle X, A, \delta, r\rangle$ for which functions δ and r are unknown to the agent, determine the optimal policy π^*. Note that this is not a supervised learning approach.

The target function is $\pi : X \rightarrow A$ but we don't have training examples $\{(x_{(i)}, \pi(x_{(i)})\}$.

Since δ and r are not known, the agent cannot predict the effect of its actions. But it can execute them and then observe the outcome.
The learning task is thus performed by repeating these steps:

- choose an action
- execute the chosen action
- observe the results
- collect the reward

14.4.4 Approaches to Learning with MDP

we have two possible approaches: **value iteration** (estimate the Value function and then compute π) and **Policy iteration** (estimate directly π)

14.4.4.1 value iteration

The agent could learn the value function $V^{\pi^*}(x)$ (written as $V^*(x)$ and determine the optimal policy from it.

$$\pi^*(x) = \underset{a \in A}{argmax}\,[r(x,a) + \gamma V^*(\delta(x,a))] \tag{14.11}$$

However, this policy cannot be computed in this way because δ and r are not known.

14.4.4.2 Q Function

when δ and r are not known, we need something else. What we can do is use $Q^{\pi}(x, a)$, which is the expected value when executing a in the state x and then act according to π. So it says how good is to execute a from x.

$$Q^{\pi}(x, a) \equiv r(x, a) + \gamma V^{\pi}(\delta(x, a)) \tag{14.12}$$

$$Q^{*}(x, a) \equiv r(x, a) + \gamma V^{*}(\delta(x, a)) \tag{14.13}$$

If the agent learns Q, then it can determine the optimal policy without knowing δ and r.

$$\pi^{*}(x) = \underset{a \in A}{argmax}\ Q(x, a) \tag{14.14}$$

let's consider the **deterministic transition**. Observe that

$$V^{*}(x) = \underset{a \in A}{max}\{r(x, a) + \gamma V^{*}(\delta(x, a))\} = \underset{a \in A}{max} Q(x, a) \tag{14.15}$$

thus we can rewrite:

$$Q(x, a) \equiv r(x, a) + \gamma V^{*}(\delta(x, a)) \tag{14.16}$$

as

$$Q(x, a) \equiv r(x, a) + \gamma \underset{a \in A}{max} Q(\sigma(x, a), a') \tag{14.17}$$

The **training rule** is:

$$\hat{Q}(x, a) \leftarrow \overline{r} + \gamma \underset{a'}{max} \hat{Q}(x', a') \tag{14.18}$$

both the reward and the next state are given by the environment.

The **algorithm** is then:

- initialize the Q table with all 0.
- observe the current state
- foreach $t = 1, \ldots, T$ until termination condition:
  - chosse an action a
  - execute the action a
  - observe the new state x'
  - collect the immediate reward $\overline{r}$
  - update the qtable:

$$\hat{Q}_t(x, a) \leftarrow \overline{r} + \gamma \underset{a'}{max} \hat{Q}_{t-1}(x', a') \tag{14.19}$$

- optimal policy:

$$\pi^{*}(x) = \underset{a \in A}{argmax}\ \hat{Q}_T(x, a) \tag{14.20}$$

Property: $\hat{Q}_n(x,a)$ underestimates $Q_n(x,a)$ and we have that $0 \leq \hat{Q}_n(x,a) \leq \hat{Q}_{n+1}(x,a) \leq Q(x,a)$. Convergence is guaranteed if all state-action pairs visited infinitely often. This means that every state must always have a probability of being visited (exploration). A way of doing this is by choosing action a with an uniform distribution. We can also use σ-greedy or softmax probability.

Lets now consider the **non deterministic** case: the reward and transition funcions are non deterministic. This means that the same action may give different results. What we do is to use the expected values in V and Q:

$$V^{\pi}(x) \equiv E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots] = E[\sum_{i=0}^{\inf} \gamma^i r_{t+i}] \tag{14.21}$$

optimal policy:

$$\pi^* \equiv \underset{\pi}{argmax}\, V^{\pi}(x), (\forall x) \tag{14.22}$$

Q is defined as:

$$Q(x,a) \equiv E[r(x,a)+\gamma V^*(\gamma(x,a))] = \dots = E[r(x,a)]+\gamma \sum_{x'} P(x'|x,a) \underset{a'}{max}\, Q(x',a') \tag{14.23}$$

properties: **Deterministic Q-learning does not converge in non-deterministic worlds**. But Non-deterministic Q-learning also converges when every pair state, action is visited infinitely often

14.4.5 Evaluating RL Agents

Cumulative reward plot may be very noisy. A better approach could be:

Repeat until termination condition:

- Execute k steps of learning
- Evaluate the current policy π_k (average and stddev of cumulative reward obtained in d runs with no exploration)

14.4.6 Different RL algorithms

- **Temporal Difference (TD)** learning
- **SARSA**

14.4.7 k-armed bandit

we have a single state and k actions. Each action returns a reward s.t. $r(a_{\cdot}i) = N(\mu_i, \sigma_i)$ Gaussian distriubution. train rule (same for non-deterministic Q learning):

$$Q_n(a_i) \leftarrow Q_{n-1}(a_i) + \alpha[\overline{r} - Q_{n-1}(a_i)] \tag{14.24}$$

$$\alpha = \frac{1}{1 + v_{n-1}(a_i)} \tag{14.25}$$

14.5 HMM and POMDP

14.5.1 Markov chain

Dynamic system evolving according to the Markov property.

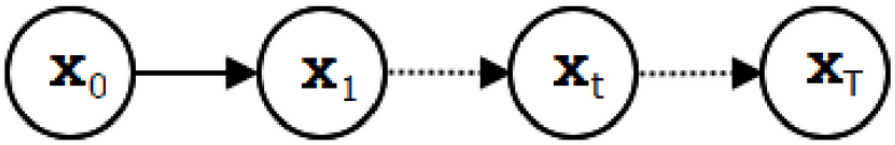


Figure 14.3: A Markov chain

Future evolution depends only on the current state x_t

14.5.2 Hidden Markov Models (HMM)

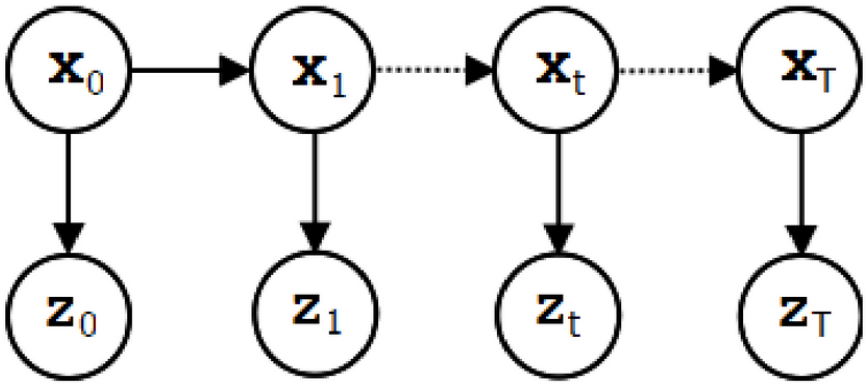


Figure 14.4: An Hidden Markov Model draw

states x_t are discrete and non-observable, observations (emissions) z_t can be either discrete or continuous. Controls u_t are not present (i.e., evolution is not controlled by our system).

In general, I cannot observe the state variables x, but I can observe the observations z which depends on the observations.

It is a Markov model because is a Markov chain and is hidden because we cannot observe the real state.

14.5.2.1 Formal definition

HMM = $\langle X, Z, \pi_0 \rangle$ where

- transition model: $P(x_t|x_{t-1})$

- observation model: $P(z_t|x_t)$
- initial distribution: π_0

the state transition matrix $A = \{A_{ij}\}$

$$A_{ij} \equiv P(x_t = j|x_{t-1} = i) \tag{14.26}$$

Observation model (discrete or continuous):

$$b_k(z_t) \equiv P(z_t|x_t = k) \tag{14.27}$$

Initial probabilities:

$$\pi_0 = P(x_0) \tag{14.28}$$

14.5.2.2 HMM properties and formulae

We apply the chain rule on HMM:

$$P(x_{0:T}, z_{0:T}) = P(x_0)P(x_1|x_0)P(z_1|x_1)P(x_2|x_1)P(z_2|x_2)\dots \tag{14.29}$$

Given a series of observations, we want to determine the distribution over states at some time stamp. Concretely, we want to determine $P(X_x|z_1, z_2, \dots, z_n)$. The task is called filtering if $t = n$, smoothing if $t < n$ Given HMM = $\langle X, Z, \pi_0 \rangle$,

- **Filtering**: $P(x_T = k|z_{1:T}) = \frac{\alpha_T^k}{\sum_j \alpha_T^j}$
- **Smoothing**: $P(x_t = k|z_{1:T}) = \frac{\alpha_t^k \beta_t^k}{\sum_j \alpha_t^j \beta_t^j}$

where:

- forward iterative steps to compute

$$\alpha_t^k \equiv P(x_t = k, z_{1:t}) \tag{14.30}$$

  - for each state k do: $\alpha_0^k = \pi_0 b_k(z_0)$
  - for each time $t = 1, \dots, T$ do:
    - for each state k do:

$$\alpha_t^k = b_k(z_t) \sum_j \alpha_{t-1}^j A_{jk} \tag{14.31}$$

- Backward iterative steps to compute

$$\beta_t^k \equiv P(z_{t+1:T}|x_t = k) \tag{14.32}$$

  - for each state k do: $\beta_T^k = 1$
  - for each time $t = T - 1, \dots, 1$ do:
    - for each state k do:

$$\alpha_t^k = \sum_j \beta_{t+1}^j A_{jk} b(z_{t+1}) \tag{14.33}$$

14.5.2.3 learning in HMM

Given output sequences, determine maximum likelihood estimate of the parameters of the HMM (transition and emission probabilities).

- **Case 1**: states can be observed at training time. Note, we are assuming that during training we can see the model. In this case transition and observation models can be estimated with statistical analysis.

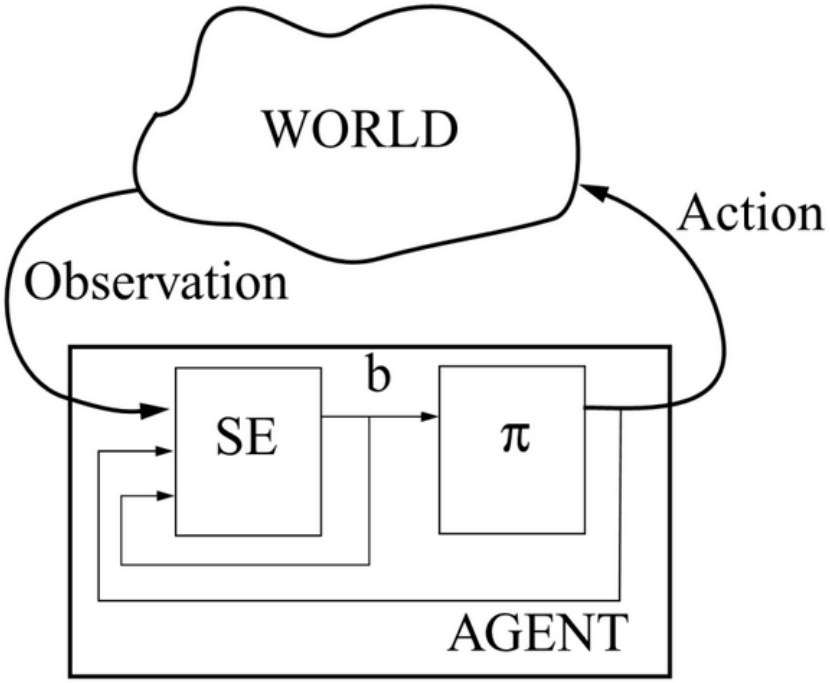
$$A_{ij} = \frac{|\{i \rightarrow j \ transitions\}|}{|\{i \rightarrow * \ transitions\}|}$$

$$b_k(v) = \frac{|\{observe \ v \wedge state \ k\}|}{|\{observe \ * \wedge state \ k\}|}$$

- **Case 2**: states cannot be observed at training time. in this case compute a local maximum likelihood with an Expectation-Maximization (EM) method.

14.5.3 POMDP agent

Combines decision making of MDP and non-observability of HMM. We are introducing actions in HMM and we cannot observe states like in MDP.



the POMDP representation is $POMDP = \langle X, A, Z, \delta, r, o \rangle$:

- X is a set of states
- A is a set of actions
- Z is a set of observations
- $P(x_0)$ is a probability distribution of the initial state
- $\delta(x, a, x') = P(x'|x, a)$ is a probability distribution over transitions
- r(x, a) is a reward function
- $o(x', a, z') = P(z'|x', a)$ is a probability distribution over observations.

the solution of the model is a policy (which is a functions from states to actions like in MDP), but we do not know the states. The solution can then be a function that maps a set of observations to an action. however is difficult to learn a function that maps observations to actions as observations are potentially infinite. What we can add is the concept of **belief** of observation.

14.5.4 Belief MDP

we can add concept of belief and learn a function from belief space to actions space. Belief $b(x)$ = probability distribution over the states. POMDP can be described as an MDP in the belief states, but belief states are infinite. The set of possible belief is **exponential** wrt the number of states.

It is possible to transform the POMDP into an MDP, but it is not practical as it moves the model in a very complex space. However there are methods that approximates an MDP. We can use parametric models or piecewise linears to approximate the real transformation and make the POMDP algorithm more efficient.