Contents

Chapter 1

Probability

www.probabilitycourse.com

1.1 Special Distributions

1.1.1 Binomial Distribution

Binomial random variable as a sum of Bernoulli random variables

Note that a $Binomial(n,p)$ random variable can be obtained by n independent coin tosses. If we think of each coin toss as a $Bernoulli(p)$ random variable, the $Binomial(n,p)$ random variable is a sum of n independent $Bernoulli(p)$ random variables. This is stated more precisely in the following **lemma**:

If $X_1, X_2, ..., X_n$ are independent $Bernoulli(p)$ random variables, then the random variable X is defined by $X = X_1 + X_2 + ... + X_n$ has a $Binomial(n,p)$ distribution.

That is, to generate a random variable X $Binomial(n,p)$, we can toss a coin n times and count the number of heads. Counting the number of heads is exactly the same as finding $X_1 + X_2 + ... + X_n$, where each X_i is equal to one if the corresponding coin toss results in heads and zero otherwise.

Example: Let X $Binomial(n,p)$ and Y $Binomial(m,p)$ be two independent random variables. Define a new random variable as $Z = X + Y$. Find the PMF of Z.

$$\begin{aligned} Z &= X + Y \\ &= X_1 + X_2 + ... + X_n + Y_1 + Y_2 + ... + Y_m, \text{where the } X_i's \text{ and } Y_j's \text{are independent } Bernoulli(p) \text{ random variables} \end{aligned}$$

So Z is a binomial random variable with parameters $m+n$ and p, i.e., $Binomial(m+n,p)$. Therefore, the PMF of z is

$$P_z(k) = \begin{cases} \binom{m+n}{k} p^k (1-p)^{m+n-k} & \text{for } k = 0, 1, 2, 3, ..., m+n \\ 0 & \text{otherwise} \end{cases}$$

So what if we want to obtain PMF of Z using probability rules??
First, $R_z = 0, 1, 2, ..., m+n$
For $k \in R_z$, we can write

$$P_z(k) = P(Z = k) = P(X + Y = k)$$

Therefore,

$$\begin{aligned}
P_z(k) &= P(X+Y=k) \\
&= \sum_{i=0}^{n} P(X+Y=k|X=i)P(X=i) \quad \text{(law of total probability)} \\
&= \sum_{i=0}^{n} P(Y=k-i|X=i)P(X=i) \\
&= \sum_{i=0}^{n} P(Y=k-i)P(X=i) \quad \text{(since } X \text{ and } Y \text{ are independent)} \\
&= \sum_{i=0}^{n} \binom{m}{k-i} p^{k-i}(1-p)^{m-k+i} \binom{n}{i} = p^i(1-p)^{n-i} \quad \text{(since } X \text{ and } Y \text{ are independent)} \\
&= \sum_{i=0}^{n} \binom{m}{k-i} \binom{n}{i} p^k(1-p)^{m+n-k} \\
&= p^k(1-p)^{m+n-k} \sum_{i=0}^{n} \binom{m}{k-i} \binom{n}{i} p^k(1-p)^{m+n-k} \\
&= \binom{m+n}{k} p^k(1-p)^{m+n-k}
\end{aligned}$$

So, we have proved Z $Binomial(m+n, p)$ by directly finding PMF of Z.

Negative Binomial (Pascal) Distribution

- The negative binomial or Pascal distribution is a generalization of the geometric distribution.
- It relates to the random experiment of repeated independent trials until observing m successes.
- Suppose that I have a coin with P(H)=p. I toss the coin until I observe m heads, where $m \in \mathbb{N}$. We define X as the total number of coin tosses in this experiment. Then X is said to have Pascal with parameter m and p.
- We write X $Pascal(m, p)$.
- Note that $Pascal(1, p) = Geometric(p)$.
- Note that by our definition the range of X is given by $R_X = \{m, m+1, m+2, ...\}$

Let us derive the PMF of a $Pascal(m, p)$ random variable X. Suppose that I toss the coin until I observe m heads, and X is defined as the total number of coin tosses in the experiment. To find the probability of the event $A = \{X = k\}$, we argue as follows. By definition, event A can be written as $A = B \cap C$, where

- B is the event that we observe $m-1$ heads (successes) in the first $k-1$ trials, and
- C is the event that we observe a heads in the kth trial

Note that B and C are independent events because they are related to different independent trials (coin tosses).
Thus we can write

$$P(A) = P(B \cap C) = P(B)P(C)$$

Chapter 2

Vector Spaces

2.1 Your heading goes here...

Your text goes here... R

2.2 R^n and C^n

2.2.1 Complex Numbers

Definition:

- A ***complex number*** is an order pair (a, b), where $a, b \in \mathbb{R}$, but we write this as $a + bi$.
- The set of all complex number is denoted by $\mathbb{C}$:
$$\mathbb{C} = \{a + bi : a, b \in \mathbb{R}\}$$
- ***Addition and multiplication*** on $\mathbb{C}$ are defined by:
$$(a + bi) + (c + di) = (a + c) + (b + d)i \quad ,$$
$$(a + bi)(c + di) = (ac - bd) + (ad + bc)i \quad ;$$
here $a, b, c, d \in \mathbb{R}$

If $a \in \mathbb{R}$, we identify $a + 0i$ with the real number a. Thus we can think of $\mathbb{R}$ as a subset of $\mathbb{C}$. We also usually write $0 + bi$ as just bi, and we usually write $0 + 1i$ as just i.

Properties of complex arithmetic

- **commutativity**
$\alpha + \beta = \beta + \alpha$ and $\alpha\beta = \beta\alpha$ for all $\alpha, \beta \in \mathbb{C}$
- **associativity**
$(\alpha + \beta) + \lambda = \alpha + (\beta + \lambda)$ and $(\alpha\beta)\lambda = \alpha(\beta\lambda)$ for all $\alpha, \beta, \lambda \in \mathbb{C}$
- **identities**
$\lambda + 0 = \lambda$ and $\lambda 1 = \lambda$ for all $\lambda \in \mathbb{C}$
- **additive inverse**
for every $\alpha \in \mathbb{C}$, there exists a unique $\beta \in \mathbb{C}$ such that $\alpha + \beta = 0$

- **multiplicative inverse**
  for every $\alpha \in \mathbb{C}$ with $\alpha \neq 0$, there exists a unique $\beta \in \mathbb{C}$ such that $\alpha\beta = 1$
- **distributive property**
  $\lambda(\alpha + \beta) = \lambda\alpha + \lambda\beta$ for all $\lambda, \alpha, \beta \in \mathbb{C}$

Definition: $-\alpha$***, subtraction,*** $1/\alpha$***, division***

Let $\alpha, \beta \in \mathbb{C}$,

- Let $-\alpha$ denote the additive inverse of α. Thus $-\alpha$ is the unique complex number such that
$$\alpha + (-\alpha) = 0$$
- ***Subtraction*** on $\mathbb{C}$ is defined by
$$\beta - \alpha = \beta + (-\alpha)$$
- For $\alpha \neq 0$, let $1/\alpha$ denote the multiplicative inverse of α. Thus $1/\alpha$ is the unique complex number such that
$$\alpha(1/\alpha) = 1$$
- ***Division*** on $\mathbb{C}$ is defined by
$$\beta/\alpha = \beta(1/\alpha)$$

Notation: $\mathbb{F}$

So we can conveniently make definitions and prove theorems that apply to both real and complex numbers, we adopt the following notation:
Throughout this book, $\mathbb{F}$ stands for either $\mathbb{R}$ or $\mathbb{C}$

The letter $\mathbb{F}$ is used because $\mathbb{R}$ and $\mathbb{C}$ are examples of what are called ***fields***.

Elements of $\mathbb{F}$ are called ***scalars***. The word "scalar", a fancy word for "number", is often used when we want to emphasize that an object is a number, as opposed to a vector (vectors will be defined soon).

For $\alpha \in \mathbb{F}$ and m a positive integer, we define α^m to denote the product of α with itself m times:
$$\alpha^m = \underbrace{\alpha...\alpha}_{m\text{times}}$$

Clearly $(\alpha^m)^n = \alpha^{mn}$ and $(\alpha\beta)^m = \alpha^m\beta^m$ for all $\alpha\beta \in \mathbb{F}$ and all positive integers m, n.

Definition: $\mathbb{F}^n$

$\mathbb{F}^n$ is the set of all lists of length n of elements of $\mathbb{F}$:
$$\mathbb{F}^n = \{(x_1, ..., x_n) : x_j \in \mathbb{F}\text{for}j = 1, ..., n\}$$
For $(x_1, ...x_n) \in \mathbb{F}^n$ and $j \in \{1, ..., n\}$, we say that x_j is the j^{th} ***coordinate*** of $(x_1, ..., x_n)$

If $\mathbb{F} = \mathbb{R}$ and n equals 2 or 3, then this definition of $\mathbb{F}^n$ agrees with $\mathbb{R}^2$ and $\mathbb{R}^3$.

Example: $\mathbb{C}^4$ is the set of all lists of four complex numbers:

$$\mathbb{C}^4 = \{(z_1, z_2, z_3, z_4) : z_1, z_2, z_3, z_4 \in \mathbb{C}\}$$

2.3 Definition of Vector Space

We will define a vector space to be a set V with an addition and a scalar multiplication on V that satisfy the properties:

- Addition is commutative, associative, and has an identity
- Every element has an additive inverse.
- Scalar multiplication is associative. Scalar multiplication by 1 acts as expected
- Addition and scalar multiplication are connected by distributive properties

2.3.1 Definition: *addition, scalar multiplication*

- An ***addition*** on a set V is a function that assigns an element $u + v \in V$ to each pair of elements $u, v \in V$.
- A ***scalar multiplication*** on a set V is a function that assigns an element $\lambda v \in V$ to each $\lambda \in \mathbb{F}$ and each $v \in V$.

2.3.2 Definition: *vector space*

A ***vector space*** is a set V along with an addition on V and a scalar multiplication on V such that the following properties hold:

- **commutativity**
  $u + v = v + u$ for all $u, v \in V$
- **associativity**
  $(u + v) + w = u + (v + w)$ and $(ab)v = a(bv)$ for all $u, v, w \in V$ and all $a, b \in \mathbb{F}$
- **additive identity**
  there exists an element $0 \in V$ such that $v + 0 = v$ for all $v \in V$;
- **additive inverse**
  for every $v \in V$, there exists $w \in V$ such that $v + w = 0$
- **multiplicative identity**
  $1v = v$ for all $v \in V$;
- **distributive properties**
  $a(u + v) = au + av$ and $(a + b)v = av + bv$ for all $a, b \in \mathbb{F}$ and all $u, v \in V$

2.3.3 Definition: *vector,point*

The following geometric language sometimes aids out intuition:
Elements of a vector space are called ***vectors*** or ***points***.

2.3.4 Definition: *real vector space, complex vector space*

- A vector space over $\mathbb{R}$ is called a ***real vector space***
- A vector space over $\mathbb{C}$ is called a ***complex vector space***

The simplest vector space contains only one point. In other words, $\{0\}$ is a vector space.

With the usual operations of addition and scalar multiplication, $\mathbb{F}^n$ is a vector space over $\mathbb{F}$. The example of $\mathbb{F}^n$ motivated our definition of vector space.

Example

F^∞ is defined to be the set of all sequences of elements of $\mathbb{F}$:

$$F^\infty = \{(x_1, x_2, ...) : x_j \in \mathbb{F} \text{ for } j = 1, 2...\}$$

Addition and scalar multiplication on F^∞ are defined as expected:

$$(x_1, x_2, ...) + (y_1, y_2, ...) = (x_1 + y_1, x_2 + y_2, ...),$$
$$\lambda(x_1, x_2, ...) = (\lambda x_1, \lambda x_2, ...)$$

.

With these definitions, F^∞ becomes a vector space over $\mathbb{F}$. The additive identity in this vector space is the sequence of all 0's.

2.3.5 $\mathbb{F}^S$

Our next example of a vector space involves a set of functions.

- If S is a set, then $\mathbb{F}^S$ denotes the set of functions from S to $\mathbb{F}$.
- For $f, g \in \mathbb{F}^S$, the ***sum*** $f + g \in \mathbb{F}^S$ is the function defined by
  $$\text{(f+g)(x)=f(x)+g(x)}$$
  for all $x \in S$
- For $\lambda \in \mathbb{F}$ and $f \in \mathbb{F}^S$, the ***product*** $\lambda f \in \mathbb{F}^S$ is the function defined by
  $$(\lambda f)(x) = \lambda f(x)$$
  for all $x \in S$.

$\mathbb{F}^S$ is a vector space

- If S is a nonempty set, then $\mathbb{F}^S$ (with the operations of addition and scalar multiplication as defined above) is a vector space over $\mathbb{F}$.
- The additive identity of $\mathbb{F}^S$ is the function $0 : S \to \mathbb{F}$ defined by
  $$\text{0(x)=0}$$
  for all $x \in S$.
- For $f \in \mathbb{F}^S$, the additive inverse of f is the function $-f : S \to \mathbb{F}$ defined by
  $$\text{(-f)(x)=-f(x)}$$
  for all $x \in S$.

The elements of the vector space $\mathbb{R}^{[0,1]}$ are real-valued functions on [0,1], not lists. In general, a vector space is abstract entity whose elements might be lists, functions, or weird objects.

Our previous examples of vector spaces, $\mathbb{F}^n$ and F^∞ are special cases of the vector space F^S because a list of length n of numbers in $\mathbb{F}$ can be thought of as a function from $\{1, 2, ..., n\}$ to $\mathbb{F}$ and a sequence of numbers in $\mathbb{F}$ can be thought of as a function from the set of positive integers to $\mathbb{F}$. In other words, we can think of $\mathbb{F}^n$ as $\mathbb{F}^{\{1,2,...,n\}}$ and $\mathbb{F}^\infty$ as $\mathbb{F}^{\{1,2,...\}}$.

2.3.6 Unique additive identity

Unique additive identity: A vector space has a unique additive identity.

Proof:
Suppose 0 and 0' are both additive identities for some vector space V.
Then

$$0' = 0' + 0 = 0 + 0' = 0$$

- where the first equality holds because 0 is an additive identity
- the second equality comes from commutativity
- the third equality holds because 0' is an additive identity

Thus 0' = 0, proving that $\mathbb{V}$ has only one additive identity.

2.3.7 Unique additive inverse

Unique additive inverse: Every element in a vector space has a unique additive inverse.

Proof:
Suppose V is a vector space. Let $v \in V$. Suppose w and w' are both additive inverses of v.
Then

$$w=w+0=w+(v+w')=(w+v)+w'=0+w'=w'$$

Thus $w = w'$, as desired.
Because additive inverses are unique, the following notation now makes sense:

Let $v, w \in V$. Then

- $-v$ denotes the additive inverse of v
- $w - v$ is defined to be $w + (-v)$

And from now on, V denotes a vector space over $\mathbb{F}$.

2.3.8 The number 0 times a vector

The number 0 times a vector: $0v = 0$ for every $v \in V$.

Proof:
For $v \in V$, we have

$$0v = (0 + 0)v = 0v + 0v$$

Adding the additive inverse of $0v$ to both sides of the equation above gives $0 = 0v$, as desired.

2.3.9 A number times the vector 0

A number times the vector 0: $a0 = 0$ for every $a \in \mathbb{F}$.

Proof:
For $a \in \mathbb{F}$, we have

$$a0 = a(0 + 0) = a0 + a0$$

Adding the additive inverse of $a0$ to both sides of the equation above gives $0 = a0$, as desired.

Difference between the above two sections:
The first section states that the product of the scalar 0 and any vector equals the vector 0, whereas the next section states that the product of an scalar and the vector 0 equals the vector 0.

2.3.10 The number -1 times a vector

The number -1 times a vector: $(-1)v = -v$ for every $v \in V$.

Proof:
For $v \in V$, we have

$$v + (-1)v = 1v + (-1)v = (1 + (-1))v = 0v = 0$$

This equation says that $(-1)v$, when added to v, gives 0. Thus $(-1)v$ is the additive inverse of v, as desired.

2.4 Subspaces

2.4.1 Definition: *subspace*

A subset U of V is called a ***subspace*** of V if U is also a vector space (using the same addition and scalar multiplication ass on V).

Example:

$(x_1, x_2, 0) : x_1, x_2 \in \mathbb{F}$ is a subspace of $\mathbb{F}^3$

Conditions for a subspace

A subset U of V is a subspace of V if and only if U satisfies the following three conditions:

- **additive identity**
$$0 \in U$$
- **closed under addition**
$$u, w \in U \text{ implies } u + w \in U$$
- **closed under scalar multiplication**
$$a \in \mathbb{F} \text{ and } u \in U \text{ implies } au \in U$$

Proof:
($\Rightarrow$)
If U is subspace of V, then U satisfies the three conditions above by the definition of vector space.

($\Leftarrow$) Suppose U satisfies the three conditions above:

- The first condition above ensures that the additive identity of V is in U.
- The second condition above ensures that addition makes sense on U.
- The third condition ensures that scalar multiplication makes sense on U.

Note that the additive identity condition above could be replaced with the condition that U is nonempty (then taking $u \in U$, multiplying it by 0, and using the condition that U is closed under scalar multiplication would imply that $0 \in U$). However, if U is indeed a subspace of V, then the easiest way to show that U is nonempty is to show that $0 \in U$.

- If $u \in U$, then $-u$ (which equals $(-1)u$) is also in U by the third condition above. Hence every element of U has an additive inverse in U.
- The other parts of the definition of a vector space, such as associativity and commutativity, are automatically satisfied for U beause they hold on the larger space V. Thus U is a vector space and hence is a subspace of V.

Example:

- If $b \in \mathbb{F}$, then
$$(x_1, x_2, x_3, x_4) \in \mathbb{F}^4 : x_3 = 5x_4 + b$$
is a subspace of $\mathbb{F}^4$ if and only if $b = 0$.
- The set of continuous real-valued functions on the interval $[0, 1]$ is a subspace of $\mathbb{R}^{[0,1]}$
- The set of differentiable real-valued functions on $\mathbb{R}$ is a subspace of $\mathbb{R}^{\mathbb{R}}$.
- The set of differentiable real-valued functions f on the interval $(0, 3)$ such that $f'(2) = b$ is a subspace of $\mathbb{R}^{(0,3)}$ if and only if $b = 0$.
- The set of all sequences of complex numbers with limit 0 is a subspace of $\mathbb{C}^{\infty}$

Sums of Subspaces

Suppose $U_1, ..., U_m$ are subsets of V. The **sum** of $U_1, ..., U_m$, denoted $U_1 + ... + U_m$ is the set of all possible sums of elements of $U_1, ..., U_m$. More precisely,

$$U_1 + ... + U_m = \{u_1 + ... + u_m \; u_1 \in U_1, ..., u_m \in U_m\}$$

Note that the union of subspaces is rarely a subspace, which is why we usually work with sums rather than unions.

Example 1:
Suppose U is the set of all elements of $\mathbb{F}^3$ whose second and third coordinates equal 0, and W is the set of all elements of $\mathbb{F}^3$ whose first and third coordinates equal 0:

$$U = \{(x, 0, 0) \in \mathbb{F}^3 : x \in \mathbb{F}\} \text{ and } W = \{(0, y, 0) \in \mathbb{F}^3 : y \in \mathbb{F}\}$$

Then

$$U + W = \{(x, y, 0) : x, y \in \mathbb{F}\}$$

as you should verify.

Example 2:
Suppose that $U = \{(x, x, y, y) \in \mathbb{F}^4 : x, y \in \mathbb{F}\}$ and $W = \{(x, x, x, y) \in \mathbb{F}^4 : x, y \in \mathbb{F}\}$. Then

$$U + W = \{(x, x, y, z) \in \mathbb{F}^4 : x, y, z \in \mathbb{F}\}$$

as you should verify.

Sum of subspaces is the smallest containing subspace

Suppose $U_1, ..., U_m$ are subspaces of V. The $U_1 + ... + U_m$ is the smallest subspace of V containing $U_1, ..., U_m$.

Proof:

- It is easy to see that $0 \in U_1 + ... + U_m$
- $U_1 + ... + U_m$ is closed under addition and scalar multiplication
- **So $U_1 + ... + U_m$ is a subspace of V.**

And then,

- $U_1, ..., U_m$ are all contained in $U_1 + ... + U_m$ (to see this, consider sums $u_1 + ... + u_m$ where all except one of the u's are 0).
- Conversely, every subspace of V containing $U_1, ..., U_m$ contains $U_1 + ... + U_m$ (because subspaces must contain all finite sums of their elements).
- **Thus $U_1 + ... + U_m$ is the smallest subspace of V containing $U_1, ..., U_m$.**

(not finished)

Chapter 3

Finite-Dimensional Vector Spaces

3.1 Span and Linear Independence

3.1.1 Linear Combination

A linear combination of a list $v_1, ..., v_m$ of vectors in V is a vector of the form
$$a_1v_1 + ... + a_mv_m,$$
where $a_1, ..., a_m \in \mathbb{F}$

3.1.2 span

The set of all linear combinations of a list of vectors $v_1, ..., v_m$ in V is called the **span** of $v_1, ..., v_m$, denoted $span(v_1, ..., v_m)$. In other words,
$$span(v_1, ..., v_m) = \{a_1v_1 + ... + a_mv_m : a_1, ..., a_m \in \mathbb{F}\}$$
The span of the empty list () is defined to be 0.

3.1.3 Span is the smallest containing subspace

The span of a list of vectors in V is the smallest subspace of V containing all the vectors in the list.

Proof:
Suppose $v_1, ...v_m$ is a list of vectors in V.

- The additive identity is in $span(v_1, ...v_m)$, because
$$0 = 0v_1 + ... + 0v_m$$
- Also, $span(v_1, ..., v_m)$ is closed under addition, because
$$(a_1v_1 + ... + a_mv_m) + (c_1v_1 + ... + c_mv_m) = (a_1 + c_1)v_1 + ... + (a_m + c_m)v_m$$
- Furthermore, $span(v_1, ...v_m)$ is closed under scalar multiplication, because
$$\lambda(a_1v_1 + ... + a_mv_m) = \lambda a_1v_1 + ... + \lambda a_mv_m$$

Thus $span(v_1, ..., v_m)$ is a subspace of V.

- Each v_j is a linear combination of $v_1, ..., v_m$. Thus $span(v_1, ..., v_m)$ contains each v_j.

- Conversely, because subspaces are closed under scalar multiplication and addition, every subspace of V containing each v_j contains $span(v_1, ..., v_m)$.

Thus $span(v_1, ..., v_m)$ is the smallest subspace of V containing all the vectors $v_1, ..., v_m$.

3.1.4 spans

If $span(v_1, ..., v_m)$ equals V, we say that $v_1, .., v_m$ **spans** V.

3.1.5 finite-dimensional vector space

A vector space is called ***finite-dimensional*** of some list of vectors in it spans the space.

3.1.6 polynomial, $P(\mathbb{F})$

- A function $p : \mathbb{F} \to \mathbb{F}$ is called a ***polynomial*** with coefficients in $\mathbb{F}$ if there exists $a_0, ..., a_m \in \mathbb{F}$ such that
$$p(z) = a_0 + a_1 z + a_2 z^2 + ... + a_m z^m$$
for all $z \in \mathbb{F}$
- $P(\mathbb{F})$ is the set of all polynomials with coefficients in $\mathbb{F}$

With the usual operations of addition and scalar multiplication, $P(\mathbb{F})$ is a vector space over $\mathbb{F}$, as you should verify. In other words, $P(\mathbb{F})$ is a subspace of $\mathbb{F}^{\mathbb{F}}$, the vector space of functions from $\mathbb{F}$ to $\mathbb{F}$.

If a polynomial (thought of as a function from $\mathbb{F}$ to $\mathbb{F}$) is represented by two sets of coefficients, then subtracting one representation of the polynomial from the other produces a polynomial that is identically zero as a function on $\mathbb{F}$ and hence has all zero coefficients (will prove it later).

Conclusion: the coefficients of a polynomial are uniquely determined by the polynomial. Thus the next definition uniquely defines the degree of a polynomial.

3.1.7 degree of a polynomial, deg p

- A polynomial $p \in P(\mathbb{F})$ is said to have ***degree*** m if there exists scalars $a_0, a_1, ..., a_m \in \mathbb{F}$ with $a_m \neq 0$ such that
$$p(z) = a_0 + a_1 z + ... + a_m z^m$$
for all $z \in \mathbb{F}$. If p has degree m, we write deg $p = m$.
- The polynomial that is identically 0 is said to have degree $-\infty$

3.1.8 $P_m(\mathbb{F})$

For m a nonnegative integer, $P_m(\mathbb{F})$ denotes the set of all polynomials with coefficients in $\mathbb{F}$ and degree at most m.

3.1.9 infinite-dimensional vector space

A vector space is called ***infinite-dimensional*** if it is not finite-dimensional.

Example: $P(\mathbb{F})$ is infinite-dimensional.
Proof:
Consider any list of elements of $P(\mathbb{F})$.
Let m denote the highest degree of polynomials in this list. Then every polynomial in the span of this list has degree at most m.
Thus z^{m+1} is not in the span of out list.
Hence no list spans $P(\mathbb{F})$.
Thus $P(\mathbb{F})$ is infinite-dimensional.

3.1.10 Linear Independence

Suppose $v_1, ..., v_m \in V$ and $v \in span(v_1, ..., v_m)$.
By the definition of span, there exist $a_1, ..., a_m \in \mathbb{F}$ such that

$$v = a_1v_1 + ... + a_mv_m$$

Consider the question of whether the choice of scalars in the equation above is unique:
Suppose $c_1, ..., c_m$ is another set of scalars such that

$$v = c_1v_1 + ... + c_mv_m$$

Subtracting the last two equations, we have

$$0 = (a_1 - c_1)v_1 + ... + (a_m - c_m)v_m$$

Thus we have written 0 as a linear combination of $(v_1, ..., v_m)$.
If the only way to do this is the obvious way (i.e., using 0 for all scalars),
then each $a_j - c_j$ equals 0, which means that each a_j equals c_j, and thus the choice of scalars was indeed unique.

This situation is so important that we give it a special name - linear independence.

linearly independent

- A list $v_1, ..., v_m$ of vectors in V is called ***linearly independent*** if the only choice of $a_1, ..., a_m \in \mathbb{F}$ that makes $a_1v_1 + ... + a_mv_m$ equals 0 is $a_1 = ... = a_m = 0$.
- The empty list () is also declared ro be linearly independent.

The reasoing above shows that $v_1, ..., v_m$ is linearly independent if and only if each vector in $span(v_1, ..., v_m)$ has only one representation as a linear combination of $v_1, ..., v_m$.

Examples:

- A list v of one vector $v \in V$ is linearly independent if and only if $v \neq 0$.
- A list of two vectors in V is linearly independent if and only if neither vecor is a scalar multiple of the other.

- $(1,0,0,0), (0,1,0,0), (0,0,1,0)$ is linearly independent in F
  $mathbbF^4$
- The list $1, z, ..., z^m$, is linearly independent in $P(\mathbb{F})$ for each nonnegative integer m.

linearly dependent

- A list $v_1, ..., v_m$ of vectors in V is called ***linearly dependent*** if it is not linearly independent.
- In other words, a list $v_1, ..., v_m$ of vectors in V is linearly dependent if there exists $a_1, ..., a_m \in \mathbb{F}$, not all 0, such that $a_1v_1 + ... + a_mv_m = 0$.

Example:
$(2,3,1), (1,-1,2), (7,3,8)$ is linearly dependent in $\mathbb{F}^3$ because
2(2,3,1)+3(1,-1,2)+(-1)(7,3,8)=(0,0,0)

Linear Dependence Lemma

Suppose $v_1, ..., v_m$ is a linearly dependent list in V.
Then there exists $j \in \{1, 2, ..., m\}$ such that the following hold:
- $v_j \in span(v_1, ..., v_{j-1})$
- if the j^{th} term is removed from $v_1, ..., v_m$, the span of the remaining list equals $span(v_1, ..., v_m)$.

Proof:

- Since the list $v_1, ..., v_m$ is linearly dependent, there exist numbers $a_1, ..., a_m \in \mathbb{F}$, not all 0, such that
$$a_1v_1 + ... + a_mv_m = 0$$
Let j be the largest element of $\{1, ..., m\}$ such that $a_j \neq 0$, Then
$$v_j = -\frac{a_1}{a_j}v_1 - ... - \frac{a_{j-1}}{a_j}v_{j-1}$$
proving the first item.
- Suppose $u \in span(v_1, ..., v_m)$. Then there exist numbers $c_1, ..., c_m \in \mathbb{F}$ such that
$$u = c_1v_1 + ... + c_mv_m$$
In this equationm we can replace v_j with the right side of the previous equation, which shows that u is in the span of the list obtained by removing j^{th} term from $v_1, ..., v_m$. Thus the second item hold.

Length of linearly independent $\leq$ list length of spanning

In a finite-dimensional vector space, the length of every linearly independent list of vector is less than or equal to the length of every spanning list of vectors.

Proof:
Suppose $u_1, ..., u_m$ is linearly independent in V.
Suppose also that $w_1, ..., w_n$ spans V.
We need to prove that $m \leq n$.

We do so through a multi-step process where we'll add one of the u's and remove one of the w's.

Step 1:

- Let B be the list $w_1, ..., w_n$ which spans V. Thus adjoining any vector in V to this list produces a linearly dependent list (because the newly adjoined vector can be written as a linear combination of the other vectors).
- In particular,
$$u_1, w_1, ..., w_n$$
is linearly dependent.
- Using the lemma we mentioned, we can remove one of the w's so that the new list B (which will still have length n) consists of u_1 and the remaining w's and still spans V.
- The reason is that u_1 can be written as
$$u_1 = a_1 w_1 + ...a_n w_n =$$
and the vector-to-be-removed w_j can then be written as
$$w_j = u_1 - a_1 w_1 - ... - a_n w_n$$
so we can remove w_j and add u_1

Step m:

- By repeating the above step m times, we've added all of the u's, and the process stops.
- At each step as we add a u to B, the lemma implies that there s some w that can be removed. So there are at least as many ws as u's.

This result can be used to show that certain lists are not linearly independent and that certain lists do not span a given vector space, without any computation.

Example:
List $(1, 2, 3), (4, 5, 8), (9, 6, 7), (-3, 2, 8)$ is not linearly independent in $\mathbb{R}^3$.
Proof: The list $(1, 0, 0), (0, 1, 0), (0, 0, 1)$ spans $\mathbb{R}^3$ and has length 3. Thus no list of length larger than 3 is linearly independent in $\mathbb{R}^3$.

Our intuition suggests that every subspace of a finite-dimensional vector space should also be finite-dimensional.

Finite-dimensional subspaces

Every subspace of a finite-dimensional vector space is finite-dimensional.

Proof:
Suppose V is finite-dimensional and U is a subspace of V. We need to prove that U is finite-dimensional.

Step 1:
If $U = \{0\}$, then U is finite-dimensional and we are done. If $U \neq \{0\}$, then choose a nonzero vector $v_1 \in U$.

Step j:
If $U = span(v_1, ..., v_{j-1})$, then U is finite-dimensional and we are done. If $U \neq span(v_1, ..., v_{j-1})$, then choose a vector $v_j \in U$ such that
$$v_j \neq span(v_1, ..., v_{j-1})$$

After each step, as long as the process continues, we have considered a list of vectors such that no vector in this list is in the span of the previous vectors.

Thus, after each step we have constructed a linearly independent list cannot be longer than any spanning list of V.

Thus, the process eventually terminates, which means that U is finite-dimensional.

3.2 Bases

3.2.1 basis

A ***basis*** of V is a list of vectors in V that is linearly independent and spans V.

Example:

- The list $(1, 0, ..., 0), (0, 1, 0, ..., 0), ..., (0, ..., 0, 1)$ is a basis of $\mathbb{F}^n$, called the **standard basis of** $\mathbb{F}^n$.
- The list $(1, 2), (3, 5)$ is a basis of $\mathbb{F}^2$
- The list $(1, 1, 0), (0, 0, 1)$ is a basic of $\{(x, x, y) \in \mathbb{F}^3 : x, y \in \mathbb{F}\}$
- The list $1, z, ..., z^m$ is a basis of $P_m(\mathbb{F})$

3.2.2 Criterion for basis

A list $v_1, ..., v_n$ of vectors in V is a basis of V if and only if every $v \in V$ can be written uniquely in the form

$$v = a_1v_1 + ...a_nv_n,$$

where $a_1, ..., a_n \in \mathbb{F}$

Proof:

$\Rightarrow$

- Suppose that $v_1, ...v_n$ is a basis of V. Let $v \in V$.
- Since $v_1, ..., v_n$ spans V, there exist $a_1, ..., a_n \in \mathbb{F}$ such that $v = a_1v_1 + ...a_nv_n$
- To show that $v = a_1v_1 + ...a_nv_n$ is unique, suppose $c_1, ..., c_n$ are scalars such that we also have
$$v = c_1v_1 + ...c_nv_n$$
- Therefore, we have $0 = (a_1 - c_1)v_1 + ... + (a_n - c_n)v_n$
- Hence, $a_1 = c_1, ..., a_n = c_n$, we have the desired uniqueness

$\Leftarrow$

- Suppose $v \in V$ can be written uniquely in $v = a_1v_1 + ...a_nv_n$
- This implies $v_1, ..., v_n$ spans V.
- To show $v_1, ..., v_n$ are linearly independent, suppose $a_1, ..., a_n \in \mathbb{F}$ are such that
$$0 = a_1v_1 + ...a_nv_n$$
- The uniqueness of $v = a_1v_1 + ...a_nv_n$ implies that $a_1 = ... = a_n = 0$
- Thus, $v_1, ..., v_n$ is linearly independent and hence is a basis of V.

3.2.3 Spanning list contains a basis

Every spanning list in a vector space can be reduced to a basis of the vector space.

Proof:
Suppose $v_1, ..., v_n$ spans V. We want to remove some of the vectors from $v-1, ..., v_n$ so that the remaining vectors form a basis of V.

Start with B equal to the list $v_1, ..., v_n$.
Step 1:

- If $v_1 = 0$, delete v_1 from B. If $v_1 \neq 0$, leave B unchanged.

Step j:

- If v_j is in $span(v_1, ..., v_{j-1})$, delete v_j from B. If v_j is not in $span()v_1, ..., v_{j-1}$, leave B unchanged.

Stop the process after step n, getting a list B. This list B spans$V$ because our original list spanned V and we have discarded only vectors that were already in the span of the previous vectors.
This process ensures that no vector in B is in the span of the previous ones.
Thus B is linearly independent by the Linear Independence Lemma Hence B is a basis of V.

3.2.4 Basis of finite-dimensional vector space

Every finite-dimensional vector space has a basis.

Proof: By definition, a finite-dimensional vector space has a spanning list. The previous result tells us that each spanning list can be reduced to a basis.

3.2.5 Linearly independent list extends to a basis

Every linearly independent list of vectors in a finite-dimensional vector space can be extended to a basis of the vector space.

Proof:

- Suppose $u_1, ...u_m$ is linearly independent in a finite-dimensional vector space V.
- Let $w_1, ..., w_n$ be a basis of V.
- Thus, the list
$$u_1, ..., u_m, w_1, ..., w_n$$
spans V.
- Therefore, we can apply the method we used in proving "Spanning list contains a basis" to reduce this lit to a basis of V.
- And this list is a basis consisting of the vectors $u_1, ...u_m$ (none of the u's get deleted in this procedure because $u_1, ...u_m$ is linearly independent) and some of the w's.

3.2.6 Every subspace of V is a part of a direct sum equal to V

Suppose V is finite-dimensional and U is a subspace of V. Then there is a subspace W of V sch that $V = U \bigoplus W$

Proof:

-

Chapter 4

Linear Maps

Chapter 5

Polynomials

Chapter 6

Eigenvalues, Eigenvectors, and Invariant Subspaces

Chapter 7

Inner Product Spaces

Chapter 8

Operators on Inner Product Spaces

Chapter 9

Operators on Complex Vector Spaces

Chapter 10

Operators on Real Vector Spaces

Chapter 11

Trace and Determinant

Chapter 12

Lecture 2:

12.1 The dot product

$http://mathinsight.org/dot_product$

The dot product between two vectors is based on the projection of one vector onto another. Let's imagine we have two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, and we want to calculate how much of a is pointing in the same direction as the vector $\boldsymbol{b}$.

Idea: We want a quantity that would be

- **positive** if the two vectors are pointing in similar directions, zero if t
- **zero** if they are perpendicular
- **negative** if the two vectors are pointing in nearly opposite directions

We will define the dot product between the vectors to capture these quantities.

First, notice that the question "how much of $\boldsymbol{a}$" is pointing in the same direction as the vector $\boldsymbol{b}$" does not have anything to do with the magnitude (or length) of $\boldsymbol{b}$
$\Rightarrow$ It is only based on $\boldsymbol{b}$'s direction.
$\Rightarrow$ The answer to this question should not depend on the magnitude of $\boldsymbol{b}$, but only its direction.
$\Rightarrow$ Let's replace $\boldsymbol{b}$ with the unit vector that points in the same direction as $\boldsymbol{b}$. We'll call this vector $\boldsymbol{u}$, which is defined by

$$\boldsymbol{u} = \frac{\boldsymbol{b}}{\|\boldsymbol{b}\|}$$

The dot product of $\boldsymbol{a}$ with unit vector $\boldsymbol{u}$, denoted $\boldsymbol{a} \cdot \boldsymbol{u}$, is defined to be the projection of $\boldsymbol{a}$ in the direction of $\boldsymbol{u}$, or the amount that $\boldsymbol{a}$ is pointing in the same direction as unit vector $\boldsymbol{u}$.

Let's assume for a moment that $\boldsymbol{a}$ and $\boldsymbol{u}$ are pointing in similar directions. Then, you can image $\boldsymbol{a} \cdot \boldsymbol{u}$ as the length of the shadow of $\boldsymbol{a}$ onto $\boldsymbol{u}$ if their tails were together and the sun was shining from a

direction perpendicular to $\boldsymbol{u}$.

By forming a right triangle with $\boldsymbol{a}$ and this shadow, you can use geometry to calculate that

$$\boldsymbol{a} \cdot \boldsymbol{u} = \|\boldsymbol{a}\| cos\theta$$

where θ is the angle between $\boldsymbol{a}$ and $\boldsymbol{u}$.

Insert diagram here

- If $\boldsymbol{a}$ and $\boldsymbol{u}$ were perpendicular, there would be no shadow. That corresponds to the case when $cos\theta = cos(\pi/2) = 0$ and $\boldsymbol{a} \cdot \boldsymbol{u} = 0$
- If the angle θ between $\boldsymbol{a}$ and $\boldsymbol{u}$ were larger than $\pi/2$, then the shadow wouldn't hit $\boldsymbol{u}$. Since in this case $cos\theta < 0$, the dot product $\boldsymbol{a} \cdot \boldsymbol{u}$ is also negative. You could think of $-\boldsymbol{a} \cdot \boldsymbol{u}$ (which is positive in this case) as being the length of the shadow of $\boldsymbol{a}$ on the vector $-\boldsymbol{u}$, which points in the opposite direction of $\boldsymbol{u}$.

Now we go back to the dot product $\boldsymbol{a} \cdot \boldsymbol{b}$, where $\boldsymbol{b}$ may have a magnitude different than one. $\Rightarrow$ This dot product $\boldsymbol{a} \cdot \boldsymbol{b}$ should depend on the magnitude of both vectors, $\|\boldsymbol{a}\|$ and $\|\boldsymbol{b}\|$, and be symmetric in those vectors.
$\Rightarrow$ Hence, we don't want to define $\boldsymbol{a} \cdot \boldsymbol{b}$ to be exactly the projection of $\boldsymbol{a}$ onto $\boldsymbol{b}$;
$\Rightarrow$ We want it to reduce to this project for the case when $\boldsymbol{b}$ is a unit vector.
$\Rightarrow$ We can accomplish this very easily: just plug the definition $\boldsymbol{u} = \frac{\boldsymbol{b}}{\|\boldsymbol{b}\|}$ into our previous product definition.
$\Rightarrow$ This leads to the definition that the dot product $\boldsymbol{a} \cdot \boldsymbol{b}$, divided by the magnitude $\|\boldsymbol{b}\|$ of $\boldsymbol{b}$, is the projection of $\boldsymbol{a}$ onto $\boldsymbol{b}$.

$$\frac{\boldsymbol{a} \cdot \boldsymbol{b}}{\|\boldsymbol{b}\|} = \|a\| cos\theta$$

Then, if we multiply it by $\|\boldsymbol{b}\|$, we get a nice symmetric definition for the dot product $\boldsymbol{a} \cdot \boldsymbol{b}$.

$$\boldsymbol{a} \cdot \boldsymbol{b} = \|a\| \|b\| cos\theta$$

The picture of the geometric interpretation of

12.2 The Inner Product as a Decision Rule

https://jeremykun.com/2017/05/22/the-inner-product-as-a-decision-rule/

In the Euclidean plane, if we have a line L passing through the origin, with w being a unit vector perpendicular to L.

Insert diagram here

If you take any vector x, the the dot product $x \cdot w$ is positive if x is on the same side of L as w, and negative otherwise. The dot product is zero iff x is exactly on the line L, including when x is the zero vector.

Insert diagram here

The core fact that makes it work is that the dot product tells you how one vector *projects* onto another. When I say "projecting" a vector x onto another vector w, I mean you take only the components of x that point in the direction of w. The demo shows what the result looks like using the red (or green) vector.

Insert diagram here

Recall $w \cdot x = |w||x|cos\theta$

Theorem: Let x, y be 2 points that lie on H. Then $w \cdot (y - x) = 0$.

Proof: $w \cdot (y - x) = -\alpha - (-\alpha) = 0$.

w is called the normal vector of H.

because (as the theorem shows) w is normal (perpendicular) o H. Or simply speaking, w is perpendicular to every pair of points lying on H.

Insert diagram here.

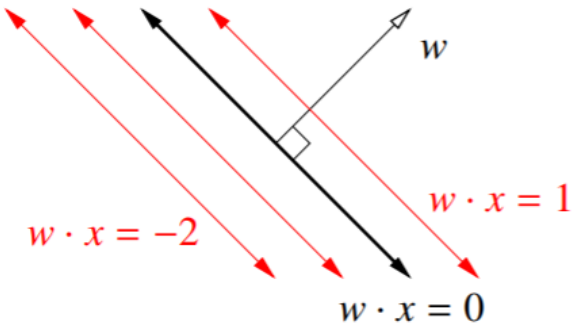

Figure 12.1: A picture of a gull.

If w is a unit vector, then $w \cdot x + \alpha$ is the signed distance from x to H. Why is that??

First, we'll consider the case $w \cdot x = 0$. Consider any point X_i, the dot product $X_i \cdot w$ (the same as $w \cdot X_i$), indicates how much of X_i is pointing to the same direction of w, and since w is a unit vector,

$X_i \cdot w = \|X_i\| cos\theta$, which, as the diagram below shows, is the **signed** distance from the hyperplane $H : x : w \cdot x = 0$.

You can picture $w \cdot x = c$ as $w \cdot x + \alpha = 0$, where $\alpha = -c$. And as you can see from the red lines in the diagram, the lines move with respect to what value α is. So $w \cdot x + \alpha$ is really the signed distance from x to H. Moreover, the distance from H to the origin is α.

Chapter 13

Lecture 3: Perceptron Learning; Maximum Margin Classifiers

13.1 Maximum Margin Classifier

The margin of a linear classifier is the distance from the decision boundary to the nearest sample.

Insert diagram here.

And a "Maximum Margin Classifier" means we are making the margin as big as possible, by enforcing the constraints

$$y_i(\boldsymbol{w} \cdot X_i + \alpha) \geq 1 \quad \forall i \in [1, n]$$

Note that here we are still assigning $y_i = -1, 1$.

And by changing the right hand side to be 1, all the sample points can only at most touch the two dotted lines.

Notice that the right-hand side is a 1, rather than a

Why w now is impossible to be set to 0 is because if $w = 0$, X_i can be at any position, thus violates the constraint of setting a maximum margin.

Recall from Lecture 2: If $\|w\| = 1$, i.e., w is a unit vector, signed distance from hyperplane to X_i is $w \cdot x + \alpha$.

Otherwise, we normalize the expression to get a unit weight vector

$$w \cdot x + \alpha = 0$$

$$\frac{w}{\|w\|} \cdot x + \frac{\alpha}{\|w\|} = 0$$

And by using this, the constraint now becomes:

$$y_i(\tfrac{w}{\|w\|} \cdot x + \tfrac{\alpha}{\|w\|}) \geq \tfrac{1}{\|w\|}$$

So There is a slab of width $\frac{2}{|w|}$ containing no sample points [with the hyperplane running along its middle]

So to maximize the margin, we have to minimize $\|w\|$. Hence the optimization problem becomes:

$\boxed{\text{Find } w \text{ and } \alpha \text{ that minimizes } \|w\|^2 \text{subject to } y_i(X_i \cdot w + \alpha) \geq 1 \quad \text{for all } i \in [1, n]}$

Chapter 14

DLB - Chapter 3: Probability and Information Theory

14.1 Marginal Probability

Sometimes we know the probability distribution over a set of variables and we want t know the probability distribution over just a subset of them. And the probability distribution over the subset is known as the *marginal probability distribution.*

For instance, suppose we have discrete random variables X and Y, and we know $P(X, Y)$.
We can find $P(X)$ with the *sum rule*:

$$\forall x \in X, P(X = x) = \sum_y P(X = x, Y = y)$$

For continuous variables, we need to use integration instead of summation:

$$p(x) = \int p(x, y)dy$$

14.2 Conditional Probability

The probability of some event, given that some other event has happened, is called *conditional probability.* We denote the conditional probability that $Y = y$ given $X = x$ as $P(Y = y|X = x)$. which can be computed with the formula:

$$P(Y = y|X = x) = \frac{P(Y=y,X=x)}{P(X=x)}$$

14.3 The Chain Rule of Conditional Probabilities

Any joint probability distribution over many random variables may be decomposed into conditional distributions over only one variable:

$$P(X_1, ..., X_n) = P(X_1) \prod_{i=2}^{n} P(X_i|X_1, ..., X_{i-1})$$

$$
\begin{aligned}
P(X_1, ..., X_n) &= P(X_n|X_1, ..., X_{n-1})P(X_1, ..., X_{n-1}) \\
&= P(X_n|X_1, ..., X_{n-1})P(X_{n-1}|X_1, ..., X_{n-2})P(X_1, ..., X_{n-2}) \\
&= P(X_n|X_1, ..., X_{n-1})P(X_{n-1}|X_1, ..., X_{n-2})P(X_{n-2}|X_1, ..., X_{n-3})P(X_1, ..., X_{n-3}) \\
&= ... \\
&= P(X_1)\prod_{i=2}^{n} P(X_i|X_1, ..., X_{i-1})
\end{aligned}
$$

For example,

- $P(a, b, c) = P(a|b, c)P(b, c)$
- $P(b, c) = P(b|c)P(c)$
- $P(a, b, c) = P(a|b, c)P(b|c)P(c)$

14.4 Independence and Conditional Independence

Two random variables are ***independent*** if their probability distribution can be expressed as a product of two factors, one involving only X and one involving only Y:

$$\forall x \in X, y \in Y, p(X = x, Y = y) = p(X = x)p(Y = y)$$

Two random variables X and Y are conditionally independent given a random variable Z if the conditional probability distribution over X and Y factorizes in this way for every value of Z:

$$\forall X \in x, Y \in y, z \in Z, p(X = x, Y = y|Z = z) = p(X = x|Z = z)p(Y = y|Z = z)$$

We can denote independence and conditional independence with compact notation: $X \perp Y$, means that X and Y are independent, while $X \perp Y|Z$ means that X and Y are conditionally independent given Z.

14.5 Expectation, Variance and Covariance

The ***expectation*** or ***expected value*** of some function $f(x)$ with respect to a probability distribution $P(x)$ is the average or mean value that f takes on when x is drawn from P. Simply speaking, it means we are finding out if we're choosing different x from P, then calculate $f(x)$, what's the average of $f(x)$.

For discrete variables this can be computed with a summation:

$$\mathbb{E}_{X\ P}[f(x)] = \sum_x P(x)f(x)$$

while for continuous variables, it is computed with an integral:

$$\mathbb{E}_{X\ P}[f(x)] = \int p(x)f(x)dx$$

We can also simply write the name of the random variable that the expectation is over, as in $\mathbb{E}_X[f(x)]$. We may also omit the subscript entirely, as in $\mathbb{E}[f(x)]$.

Expectations are linear, for example,

$$\mathbb{E}_X[\alpha f(x) + \beta g(x)] = \alpha \mathbb{E}_X[f(x)] + \beta \mathbb{E}_X[g(x)]$$

when α and β are not independent on x.

The ***variance*** gives a measure of how much the values of a function of a random variable X vary as we sample different values of x from its probability distribution:

$$\text{Var}(f(x)) = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])^2]$$

As you can see, what $\text{Var}(f(x))$ is doing is it compares every ($f(x)$ with $\mathbb{E}[f(x)]$), but since that value can be positive or negative, so if calculate the average, we may get near to zero variance even though the values of the function varies a lot, so we square it.

When the variance is low, the values of $f(x)$ cluster near their expected value. The square root of the variance is known as ***standard deviation***.

The covariance gives some sense of how much two values are linearly related to each other, as well as the scale of these variables:

$$\text{Cov}[f(x), g(y)] = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])(g(y) - \mathbb{E}[g(y)])]$$

High absolute

Chapter 15

Machine Learning Basics

15.1 Learning Algorithms

A machine learning algorithm is an algorithm that is able to learn from data. But what do we mean by learning?

> Mitchell (1997): *"A computer is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."*

15.1.1 The Task, T

If we want a robot to be able to walk, then walking is the task. We could program the robot to learn to walk, or we could attempt to directly write a program that specifies how to walk manually.

Machine learning tasks are usually described in terms of how the machine learning system should process an ***example***. An ***example*** is a collection of ***features*** that have been quantitatively measured from some object or event that we want the machine learning system to process.

We typically represent an ***example*** as a vector $x \in \mathbb{R}^n$ where each entry x_i of the vector is another ***feature***. For example, the features of an image are usually the values of the pixels in the image.

Some of the most common machine learning tasks include the following:

- Refer to Deep Learning Book's section 5.1.1, page 100-102

15.1.2 The Performance Measure, P

For tasks such as classification, classification with missing inputs, and transcription, we often measure the ***accuracy*** of the model. Accuracy is just the proportion of examples for which the model produces the correct output. We can also obtain equivalent information by measuring the ***error rate***, the proportion of examples for which the model produce an incorrect output. We often refer to the error rate as the expected 0-1 loss. The 0-1 loss on a particular example is 0 if it is correctly classified and 1 if it is not.

For tasks such as density estimation, it does not make sense to measure accuracy, error rate, or any other kind of 0-1 loss. Instead, we must use a different performance metric that gives the model a continuous-valued score for each example. The most common approach is to report the average log-probability the model assigns to some examples.

Usually we are interested in how well the machine learning algorithm performs on data that it has not seen before, since this determines how well it will work when deployed in the real world. We therefore evaluate these performance measures using a ***test set*** of data that is separate from the data used for training the machine learning system.

In some cases, this is because it is difficult to decide what should be measured. For example, when performing a transcription task, should we measure the accuracy of the system at transcribing entire sequences, or should we use a more fine-grained performance measure that gives partial credit for getting some elements of the sequence correct? When performing a regression task, should we penalize the system more if it frequently medium-sized mistakes or if it rarely makes very large mistakes? These kinds of design choices depend on the application.

In other cases, we know what quantity we would ideally like to measure, but measuring it is impractical. For example, this arises frequently in the context of density estimation. Many of the best probabilistic value assigned to a specific point in space in many such models is intractable. In these cases, one must design an alternative criterion that still corresponds to the design objectives, or design a good approximation to the desired criterion.

15.1.3 The Experience, E

Machine learning algorithms can be broadly categorized as ***unsupervised*** or ***supervised*** by what kind of experience they are allowed to have during the learning process.

Most of the learning algorithms in this book can be understood as being allowed to experience an entire ***dataset***. A dataset is a collection of many examples, as defined in Sec. 15.1.1. Sometimes we will also call examples ***data points***.

One of the oldest datasets studied by statisticians and machine learning researchers is the Iris dataset (Fisher, 1936). It is a collection of measurements of different parts of 150 iris plants. Each individual plant corresponds to one example. The features within each example are the measurements of each of the parts of the plant: the sepal length, sepal width, petal length and petal width. The dataset also records which species each plant belonged to. Three different species are represented in the dataset.

Unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of the structure of this dataset. In the context of deep learning, we usually want to learn the entire probability distribution that generated a dataset, whether explicitly as in density estimation or implicitly for tasks like synthesis or denoising. Some other unsupervised learning algorithms perform other roles, like clustering, which consists of dividing the dataset into clusters of similar examples.

Supervised learning algorithms experience a dataset containing features, but each example is also associated with a label or target. For example, the Iris dataset is annotated with the species of each iris plant. A supervised learning algorithm can study the Iris dataset and learn to classify iris plants

into three different species based on their measurements.

Roughly speaking, unsupervised learning involves observing several examples of a random vector x, and attempting to implicitly or explicitly learn the probability distribution $p(x)$, or some interesting properties of that distribution, while supervised learning involves observing several examples of a random vector x and an associated value or vector y, and learning to predict y from x, usually by estimating $p(y|x)$. The term ***supervised learning*** originates from the view of the target y being provided by an instructor or teacher who shows the machine learning system what to do. In ***unsupervised learning***, there is no instructor or teacher, and the algorithm must learn to make sense of the daata without this guide.

Unsupervised learning and ***supervised learning*** are not formally defined terms. The lines between them are often blurred. Many machine learning technologies can be used to perform both tasks. For example, the chain rule of probability states that for a vector $x \in \mathbb{R}^n$, the joint distribution can be decomposed as

$$p(x) = \prod_{i=1}^{n} p(x_i|x_1, ..., x_{i-1})$$

This decomposition means that we can solve the ostensibly unsupervised problem of modeling $p(x)$ by splitting it into n supervised learning problems. Alternatively, we can solve the supervised learning problem of learning $p(y|x)$ by using traditional unsupervised learning technologies to learn the joint distribution $p(x, y)$ and inferring

$$p(y|x) = \frac{p(x,y)}{\sum_{y'} p(x,y')}$$

Though unsupervised learning and supervised learning are not completely formal or distinct concepts, they do help to roughly categorize some of the things we do with machine learning algorithms. Traditionally, people refer to regression, classification and structured output problems as supervised learning. Density estimation in support of other tasks is usually considered unsupervised learning.

Other variants of the learning paradigm are possible. For example, in semisupervised learning, some examples include a supervision target but others do not. In multi-instance learning, an entire collection of examples is labeled as containing or not containing an example of a class, but the individual members of the collection are not labeled. For a recent example of multi-instance learning with deep models, see Kotzias et al. (2015).

Some machine learning algorithms do not just experience a fixed dataset. For example, reinforcement learning algorithms interact with an environment, so there is a feedback loop between the learning system and its experiences. Such algorithms are beyond the scope of this book. Please see Sutton and Barto (1998) or Bertsekas and Tsitsiklis (1996) for information about reinforcement learning, and Mnih et al. (2013) for the deep learning approach to reinforcement learning.

Most machine learning algorithms simply experience a dataset. A dataset can be described in many ways. In all cases, a dataset is a collection of examples, which are in turn collections of features.

One common way of describing a dataset is with a ***design matrix***. A design matrix is a matrix containing a different example in each row. Each column of the matrix corresponds to a different feature.

For instance, the Iris dataset contains 150 examples with four features for each example. This means we can represent the dataset with a design matrix $\mathbb{X} \in \mathbb{R}^{150x4}$, where $\mathbb{X}_{i,1}$ is the sepal length of plant i, $\mathbb{X}_{i,2}$ is the sepal width of plant i, etc. We will describe most of the learning algorithms in this book in terms of how they operate on design matrix datasets.

Of course, to describe a dataset as a design matrix, it must be possible to describe

15.2 Example: Linear Regression

Linear regression solves a regression problem. In other words, the goal is to build a system that can take a vector $x \in \mathbb{R}^n$ as input and predict the value of a scalar $y \in \mathbb{R}$ as its output. In the case of linear regression, the output is a linear function of the input. Let $\hat{y}$ be the value that our model predicts y should take on. We define the output to be

$$\hat{y} = w^T x$$

where $w \in \mathbb{R}^n$ is a vector of *parameters*.

We can think of w as a set of *weights* that determine how each feature affects the prediction.

We thus have a definition of out task T: to predict y from $\mathbf{x}$ by outputting $\hat{y} = \mathbf{w}^T\mathbf{x}$.

Now we need a definition of our performance measure, P. Suppose that we have a design matrix of m example inputs that we will not use for training, only for evaluating how well the model performs. We also have a vector of regression targets providing the correct value of y for each of these examples.

Chapter 16

Deep Feedforward Networks

The goal of a feedforward network is to approximate some function f^*. For example, for a classifier, $y = f^*(x)$ maps an input x to a category y. A feedforward network defines a mapping $y = f(x; \theta)$ and learns the value of θ that result in the best function approximation.

These models are called *feedforward* because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y. There are no *feedback* connections in which outputs of the model are fed back into itself. When feedforward neural networks are extended to include feedback connections, they are called *recurrent neural networks*.

Feedforward neural networks are called *networks* because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together. For example, we might have three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain, to form $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the *first layer of the network*, $f^{(2)}$ is called the *second layer*, and so on. The overall length of the chain gives the *depth* of the model. The final layer of a feedforward network is called *output layer*.

During neural network training, we drive $f(x)$ to match $f^*(x)$. The training data provides us with noisy, approximate examples of $f^*(x)$ evaluated at different training points. Each example x is accompanied by a label $y \approx f^*(x)$. The training examples specify directly what the output layer must do at each point x; it must produce a value that is close to y. The behavior of other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data does not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of f^*. Because the training data does not show the desired output for each of these layers, these layers are called *hidden layers*.

Each hidden layer of the network is typically vector-valued. The dimensionality of these hidden layers determines the *width* of the model. Each element of the vector may in interpreted as playing a role analogous to a neuron. Rather than thinking of the layer as consisting of many *units* that act in parallel, each representing a vector-to-scalar function. Each unit resembles a neuron in the sense that it receives input from other units and computes its own activation value. The idea of using many layers of vector-valued representation is drawn from neuroscience. The choice of the functions $f^{(i)}(x)$ used to compute these representations is also loosely guided by neuroscientific observations about the

functions that biological neurons compute. However, it is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function.

One way to understand feedforward networks is to begin with linear models and consider how to overcome their limitations. Linear models, such as logistic regression and linear regression, are appealing because they may be fit efficiently and reliably, either in closed form or with convex optimization. Linear models also have the obvious defect that the model capacity is limited to linear functions, so the model cannot understand the interaction between any two input variables.

To extend linear models to represent nonlinear functions of x, we can apply the linear model not to x itself but to a transformed input $\phi(x)$, where ϕ is a nonlinear transformation. Equivalently, we can apply the kernel trick described in (Deep Learning Book's Section 5.7.2), to obtain a nonlinear learning algorithm based on implicitly applying the ϕ mapping. We can think of ϕ as providing a set of features describing x, or as providing a new representation for x.

The question is then how to choose the mapping ϕ:

1. Use a very generic ϕ, such as the infinite-dimensional ϕ that is implicitly used by kernel machines based on the RBF kernel.

    - If $\phi(x)$ is of high enough dimension, we can always have enough capacity to fit the training set, but generalization to the test set often remains poor.
    - Very generic feature mappings are usually based only on the principle of local smoothness and do not encode enough prior information to solve advanced problems.

2. Manually engineer ϕ

    - Until the advent of deep learning, this was the dominant approach.
    - This approach requires decades of human effort for each separate task, with peactitioners specializing in different domains such as speech recognition or computer vision, and with little transfer between domains.

3. Learn ϕ.

    - In this approach, we have a model $y = f(x; \theta, w) = \phi(x; \theta)^T w$

Chapter 17

Convolutional Networks

Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

17.1 The Convolution Operation

In its most general form, convolution is an operation on two functions of a real-valued argument.

Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output $x(t)$, the position of the spaceship at time t. Both x and t are real-valued, i.e., we can get a different reading from the laser sensor at any constant in time.

Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average together several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted function $w(a)$, where a is the age of a measurement. We can do this a weighting average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the spaceship:

$$s(t) = \int [x(a)][w(t-a)]da$$

This operation is called *convolution*. The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * w)(t)$$

In our example, w needs to be a valid probability density function, or the output is not a weighted average. Also, w needs to be 0 for all negative arguments, or it will look into the future, which is presumably beyond our capabilities. These limitations are particular to our example though. In general, convolution is defined for any functions for which the above integral is defined, and may be used for other purposes besides taking weighted averages.

In convolutional network terminology, the first argument (in this example, the function x) to the convolution is often referred to as the *input* and the second argument (in this example, the function w) as

the *kernel*. The output is sometimes referred to as the *feature map*.

In our example, the idea of a laser sensor that can provide measurements at every instant in time is not realistic. Usually, when we work with data on a computer, time will be discretized, and our sensor will provide data at regular intervals. In our example, it might be more realistic to assume that our laser provides a measurement once per second. The time index t can then take on only integer values. If we now assume that x and w are defined only on integer t, we can define the discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} [x(a)][w(t-a)]$$

In machine learning applications, the input is usually a multidimensional array of parameters that are adapted by the learning algorithm. We will refer to these dimensional arrays as tensors. Because each element of the input and kernel must explicitly stored separately, we usually assume that these functions are zero everywhere but the finite set of points for which we store the values. This means that in practice we can implement the infinite summation as a summation over a finite number of array elements.

Finally, we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image I as our input, we probably also want to use a two-dimensional kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

Here, the input is I, the kernel is K.

Convolution is commutative, meaning we can equivalently write:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n K(m,n)I(i-m, j-n) = I(i-m, j-n)K(m,n)$$

Usually the latter formula is more straightforward to implement in a machine learning library, because there is less variation in the range of valid values of m and n.

The commutative property of convolution arises because we have *flipped* the kernel relative to the input, in the sense that as m increases, the index into the input increases, but the index into the kernel decreases. The only reason to flip the kernel is to obtain the commutative property. While the commutative property is useful for writing proofs, it is not usually an important property of a neural network implementation.

Instead, many neural network libraries implement a related function called the ***cross-correlation***, which is the same as convolution but without flipping the kernel:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

Many machine learning libraries implement cross-correlation but call it convolution.

blah blah blah

17.1.1 Convolution and Correlation

http://www.cs.umd.edu/ djacobs/CMSC426/Convolution.pdf

Convolution and correlation have two key features:

- *shift-invariant*: we perform the same operation at every point in the image
- *linear*: the operation is linear, that is, we replace every pixel with a linear combination of its neighbors.

These two properties make these operations very simple; it's simpler if we do the same thing everywhere, and linear operations are always the simplest ones.

Example:

One of the simplest operations that we can perform with correlation is local averaging, which we replace every pixel in a 1D image by the average of that pixel and its two neighbors. Suppose we have an image I equal to:

5	4	2	3	7	4	6	5	3	6

Table 17.1: image I

Averaging is an operation that takes an image as input, and produces a new image as output. When we average the fourth pixel, for instance, we replace the value 3 with the average of 2,3 and 7. That is, if we call the new image that we produce J we can write:

$$J(4) = [I(3) + I(4) + I(5)]/3 = 4$$

Notice that every pixel in the new image depends on the pixels in the old image. Averaging like this is shift-invariant, because we perform the same operation at every pixel. Every new pixel is the average of itself and its two neighbors. Averaging is linear because every new pixel is a linear combination of the old pixels. This example illustrates another property of all correlation and convolution that we will consider: The output image at a pixel is based on only a small neighborhood of pixels around it in the input image.

Boundaries: We still haven't fully described correlation, because we haven't said what to do at the boundaries of the image. What is $J(1)$? There is no pixel on its left to include in the average, i.e., $I(0)$ is not defined. There are four common ways of dealing with this issue:

- The first way is to image that I is part of an infinitely long image which is zero everywhere except where we have specified. So we can say:

$$J(1) = [I(0) + I(1) + I(2)]/3 = (0 + 5 + 4)/3 = 3$$

.	.	.	5	4	2	3	7	4	6	5	3	6	.	.	.

Table 17.2: image I

- The second way is to also imagine that I is part of an infinite image, but to extent it using the first and last pixels in the image. So we can say:

$$J(1) = [I(0) + I(1) + I(2)]/3 = (5 + 5 + 4)/3 = 4.66666$$

.	.	.	5	5	5	4	2	3	7	4	6	5	3	6	6	6	.	.	.

Table 17.3: image I

- The third way is to imagine the image as being like a circle, so that the pixel values repeat over and over again. So we would have

$$J(1) = [I(0) + I(1) + I(2)]/3 = (6 + 5 + 4)/3 = 5$$

- Finally, we can simply say that the image is undefined beyond the values that we have been given. In that case, we cannot compute any average that uses these undefined values, so $J(1)$ and $J(10)$ will be undefined, and J will be smaller that I.

These four methods have different advantages and disadvantages. If we imagine that the image we are using is just a small window on the world, and we want to use values outside the boundary that are most similar to the values that we would have obtained if we'd taken a bigger picture, then the second approach probably makes the most sense. That is, if we had to guess at the value of $I(0)$, even though we can't see it, the value we can see in $I(1)$ is probably a pretty good guess.

Correlation as a Sliding, Windowed Operation

We're now going to look at the same averaging operation in a slightly different way which is more graphical, and perhaps more intuitive to generalize. In averaging, for a specific pixel we multiply it and its neighbors by 1/3 each, and then add up the three resulting numbers. The numbers we multiply, (1/3, 1/3, 1/3) form a *filter*. This particular filter is called a *box* filter. We can think of it as a 1x3 structure that we slide along the image. At each position, we multiply each number of the filter by the image number that lies underneath it, and add these all up. The result is a new number corresponding to the pixel that is underneath the center of the filter.

We continue to do the sliding until we have produced every pixel in J. With this view of correlation, we can define a new averaging procedure by just defining a new filter. For example, suppose instead of averaging a pixel with its immediate neighbors, we want to average each pixel with immediate neighbors and their immediate neighbors. We can define a filter as (1/5,1/5,1/5,1/5,1/5). Then we perform the same operation as above, but using a filter that is five pixels wide. The first pixel in the resulting image will then be: $J(1) = I(-1)/5 + I(0)/5 + I(1)/5 + I(2)/5 + I(3)/5 = 1 + 1 + 1 + 4/5 + 2/5 = 4.2$

A Mathematical Definition for Correlation

Suppose F is a correlation filter, it will be convenient notationally to suppose that F has an odd number of elements, so we can suppose that as it shifts, its center is right on top of an element of I. So we say that F has $2N + 1$ elements, and that these are indexed from $-N$ to N, so that the center element of F is $F(0)$. Then we can write:

$$F \circ I(x) = \sum_{i=-N}^{N} F(i)I(x + i)$$

where the circle denotes correlation. With this notation, we can define a simple box filter as:

$$F(i) = \begin{cases} 1/3 & for \quad i = -1, 0, 1 \\ 0 & for \quad i \neq -1, 0, 1 \end{cases}$$

Constructing an Filter from a Continuous Function

It is pretty intuitive what a reasonable averaging filter should look like. Now we want to start to consider more general strategies for constructing filters. It commonly occurs that we have in mind a continuous function that would make a good filter, and we want to come up with a discrete filter that approximates this continuous function. Some reasons for thinking of filters first as continuous functions will be given when we talk about the Fourier transform. But in the mean time we'll give an example of an important continuous function used for image smoothing, the Gaussian.

A one-dimensional Gaussian is:

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

This is also known as a Normal distribution. Here μ is the mean value, and σ is the variance.
The mean, μ, gives the location of the peak of the function. The parameter σ controls how wide the peak is. As σ gets smaller the peak becomes narrower.

The Gaussian has a lot of very useful properties, some of which we'll mention later.

- When we use it as a filter, we can think of it as a nice way to average. The averaging filter that we introduced above replaces each pixel with the average of its neighbors. This means that nearby pixels all play an equal role in the average, and more distant pixels play no role.
- It is more appealing to use the Gaussian to replace each pixel with a weighted average of its neighbors. In this way, the nearest pixels influence the average more, and more distant pixels play a smaller and smaller role. This is more elegant, because we have a smooth and continuous change. We will show this more rigorously when we discuss the Fourier transform.

When we use a Gaussian for smoothing, we will set $\mu = 0$, because we want each pixel to be the one that has the biggest effect on its new, smoothed value. So our Gaussian has the form:

$$G_0(x) = \frac{1}{\sigma\sqrt{2\pi}} exp\left(\frac{-x^2}{2\sigma^2}\right)$$

σ will serve as a parameter that allows us to control how much we smooth the image (that is, how big a neighborhood we use for averaging). The bigger σ is, the more we smooth the image.

We now want to build a discrete filter that looks like a Gaussian. We can do this by just evaluating the Gaussian at discrete locations. That is, although G is defined for any continuous value of x, we will just use its values at discrete locations (..., -3, -2, -2, 0, 1, 2, 3, ...). One way to think about this is that we are approximating our original, continuous function with a piecewise constant function, where the pieces are each one pixel wide.

However, we cannot evaluate G at every integer location, because this would give s a filter that is infinitely long, which is impractical. Fortunately, as x gets away from 0, the value of $G(x)$ will approach 0 pretty quickly, so it will be safe to ignore some values of $G(x)$, but a reasonable rule of thumb is to make sure we capture 99% of the function. What we mean by this is that the sum of values in our filter will contain at least 0% of what we would get in an infinitely long filter, that is, we choose N so that:

$$\frac{\sum_{-N}^{N} G(x)}{\sum_{-\infty}^{\infty} G(x)} > .99$$

There is one final point. An averaging filter should have all its elements add up to 1. This is what averaging means: if the filter elements don't add up to one, then we are not only smoothing the iamge, we are making it dimmer or brighter. The continuous Gaussian integrates to one (after all, it's a probability density function, and probabilities must add up to one), but when we sample it and truncate it, there is no guarantee that the values we get will still add up to one. So we must normalize our filter, meaning the elements of our filter will be calculate as:

$$F(i) = \frac{G(i)}{\sum_{-N}^{N} G(x)} \text{ for } i = -N...N$$

Taking Derivatives with Correlation

Averaging, or smoothing, is one of the most important things we can do with a filter. However, another very useful thing is taking a derivative. This allows us to measure how quickly an image is changing. Taking derivatives is something we usually think of doing with a continuous function. Technically, derivatives aren't even defined on a discrete image, because a derivative measures how quickly the image is changing over an interval, in the limit, as the interval becomes infinitely small. However, just as we were able to produce a discrete filter that was an approximation to a continuous Gaussian, we can discretely approximate a continuous derivative operator.

Intuitively, a derivative is found by taking the difference as we go from one part of a function to another. That is, it's the change in y divided by the change in x. So it's natural that a derivative filter will look something like: (-1/2 0 1/2). When we apply this filter, we get $J(i) = [I(i+1) - I(i-1)]/2$. This is taking the change in y, that is, the image intensity, and dividing it by the change in x, the image position.

Let's consider an example. Suppose image intensity grows quadratically with position, so that $I(x) = x^2$. Then if we look at the intensities at positions 1,2,3,... they will look like:

1	4	9	16	25	36	.	.	.

Table 17.4: image I

If we filter this with a filter of the form (-1/2 0 1/2) we will get:

1.5	4	6	8	10	12	.	.	.

Table 17.5: image I

We know that the derivative of $I(x), dI/dx = 2x$. And sure enough, we have, for example, that $J(2) = 4, J(3) = 6, ...$ Notice that $J(1)$ is not equal to 2 because of the way we handle the boundary, by setting $I(0) = 1$ instead of $I(0) = 0$. So at the boundary, our image doesn't reflect $I(x) = x^2$.

We could just as easily have used a filter like (0 -1 1), which computes the expression: $J(i) = [I(i+1) - I(i)]/1$, or a filter (-1 1 0), which computes $J(i) = [I(i) - I(i-1)]/1$. These are all reasonable approximations to a derivative. One advantage of the filter (-1/2 0 1/2) is that it treats the neighbors of a pixel symmetrically.

Matching with Correlation

Part of the power of correlation is that we can use it, and related methods, to find locations in an image that are similar to a template. To do this, think of the filter as a template; we are sliding it around the image looking for a location where the template overlaps the images so that values in the template are aligned with similar values in the image.

First, we need to decide how to measure the similarity between the template and the region of the image with with which it is aligned. A simple and natural way to do this to measure the sum of the square of the differences between values in the template and in the image. This increases as the difference between the two increases. For the difference between the filter and the portion of the image centered at x, we can write this as:

$$\sum_{i=-N}^{N} [F(i) - I(x+i)]^2 = \sum_{i=-N}^{N} [F^2(i) + I^2(x+i) - 2F(i)I(x+i)]$$
$$= \sum_{i=-N}^{N} [F^2(i)] + \sum_{i=-N}^{N} [I^2(x+i)] + 2 \sum_{i=-N}^{N} [F(i)I(x+i)]$$

As shown, we can break the Euclidean distance into three parts:

- The first part depends only on the filter. This will be the same for every pixel in the image.
- The second part is the sum of squares of pixel values that overlap the filter.
- And the third part is twice the negative value of the correlation between F and I.

We can see that, all things being equal, as the correlation between the filter and the image increases, the Euclidean distance between them decreases. This provides an intuition for using correlation to match a template with an image. Places where the correlation between the two is high tend to be locations where the filter and image match well.

This also shows one of the weaknesses of using correlation to compare a template and image. Correlation can also be high in locations where the image intensity is high, even if it doesn't match the template well. Here's an example. Suppose we correlate the filter

3	7	5

with the image:

3	2	4	1	3	8	4	0	3	8	0	7	7	7	1	2

We get the following the result:

40	43	39	34	64	85	52	27	61	65	59	84	105	75	38	27

Notice that we get a high result (85) when we center the filter on the ***sixth*** pixel, because (3,7,5) matches very well with (3,8,4). But, the highest correlation occurs at the thirteenth pixel, where the filter is matched to (7,7,7). Even though this is not as similar to (3,7,5), its magnitude is greater.

One way to overcome this is by just using the sum of square differences between the signals, as given above. This produces the result:

25	26	26	41	29	2	59	54	34	26	78	13	20	32	61	38

We can see that using Euclidean distance, (3,7,5) best matches the part of the image with pixel values (3,8,4). The next best match has values (0,7,7), but this is significantly worse.

Another option is to perform ***normalized correlation*** by computing:

$$\frac{\sum_{i=-N}^{N}[F(i)I(x+i)]}{\sqrt{\sum_{i=-N}^{N}[I(x+i)]^2}\sqrt{\sum_{i=-N}^{N}[F(i)]^2}}$$

This measure of similarity is similar to correlation, but it is invariant to scaling the template or a corresponding region in the image. If we scale $I(x-N)...I(x+N)$, by a single, constant factor, this will scale the numerator and denominator by the same amount. Consequently, for example, (3,7,5) will have the same normalized correlation with (7,7,7) that it would have with (1,1,1). When we perform normalized correlation between (3,7,5) and the above image we get:

.946	.877	.934	.73	.81	.989	.64	.59	.78	.835	.61	.931	.95	.83	.57	.988

(not finished)

17.1.2 Motivation

Convolution leverages three important ideas that can help improve a machine learning system:

- **sparse interaction**
- **parameter sharing**
- **equivariant representations**

Moreover, convolution provides a means for working wit inputs of variable size.

Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means that every output unit interacts with every input unit.

Convolutional networks, however, typically have **sparse interactions**. This is accomplished by making the kernel smaller that the input. For example, when processing an image, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency.It also means that computing the output requires fewer operations.

These improvements in efficiency are usually quite large. If there are m inputs and n outputs, then

- matrix multiplication requires mxn parameters
- algorithms used in practice have $O(m \times n)$ runtime (per sample)

If we limit the number of connections each output may have to k, then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime. For graphical demonstration of sparse connectivity, check out the following figure:

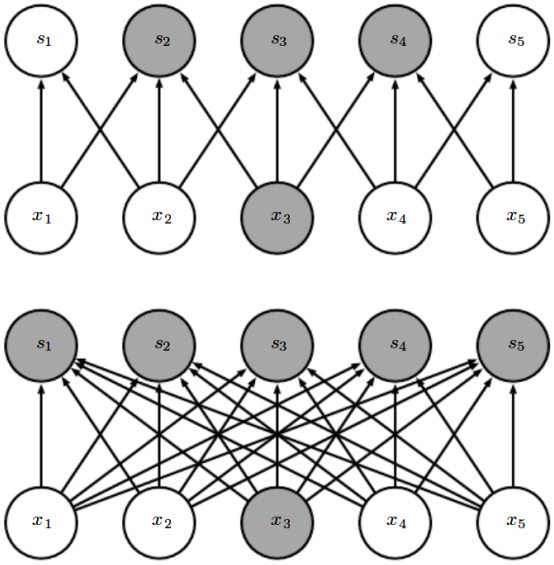

Figure 17.1: **Sparse Connectivity, view from below**: We highlight one input unit, x_3, and highlight the output units in s that are affected by this unit. At the ***top***, when s is formed by convolution with a kernel of width 3, only three inputs are affected by x. At the ***bottom***, when s is formed by matrix multiplication, connectivity is no longer sparse, so all the outputs are affected by x_3.

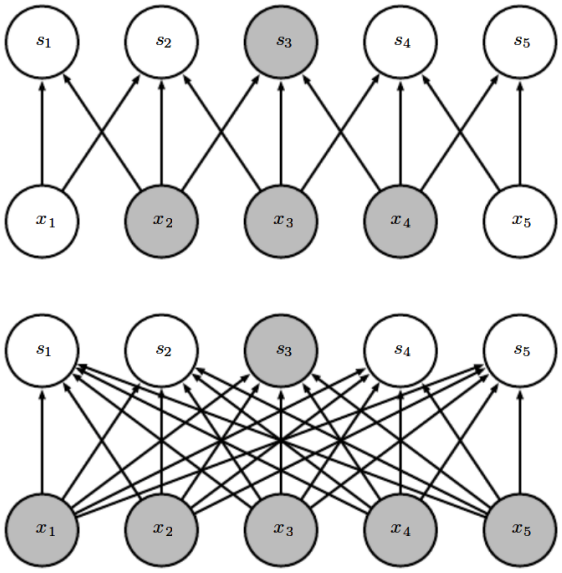

Figure 17.2: **Sparse Connectivity, view from above**: We highlight one output unit, s_3, and highlight the input units in x that affect this unit. These units are known as the **receptive field** of s_3. At the ***top***, when s is formed by convolution with a kernel of width 3, only three inputs affect s_3. At the ***bottom***, when s is formed by matrix multiplication, connectivity is no longer sparse, so all the inputs affect s_3.

In a deeper convolutional network, units in the deeper layers may *indirectly* interact with a larger portion of the input, as shown in the following figure. This allows the network efficiently describe complicated interactions between many variables by constructing such interaction from simple building blocks that each describe only sparse interactions.

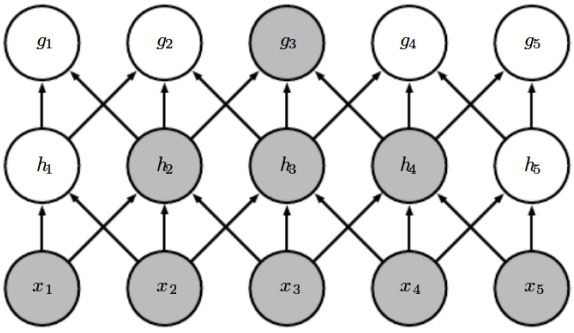

Figure 17.3: The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers. This effect increases if the network includes architectural features like *strided* convolution or *pooling*. This means that even though ***direct*** connections in a convolutional net are very sparse, units in the deeper layers can be ***indirectly*** connected to all or most of the input image.

Parameter sharing refers to using the same parameter for more than one function in a model. In a **traditional neural net**, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. As a synonym for parameter sharing, one can say that a network has **tied weights**, because the value of the weight

applied to one input is tied to the value of a weight applied elsewhere. In a **convolutional neural net**, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). Te parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set.

This does not affect the runtime of forward propagation - it is still $O(k \times n)$ - but it does further reduce the storage requirements of the model to k parameters. Recall than k is usually several orders of magnitude smaller than m. Since m and n are usually roughly the same size, k is practically insignificant compared to $m \times n$. Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency. For a graphical depiction of how parameter sharing works, check out the following figure:

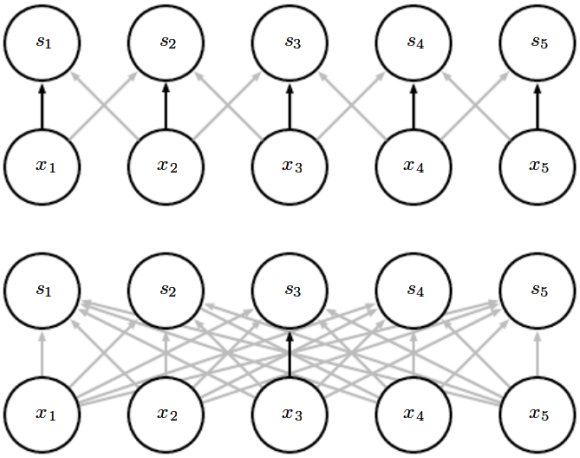

Figure 17.4: **Parameter sharing**: Black arrows indicate the connections that use a particular parameter in two different models. At the ***top***, the black arrows indicate uses of the central element of a 3-element kernel in a convolutional model.