Note on Machine Learning
Haocheng Dai

1 Linear Algebra

<table>
<thead>
<tr>
<th>Finite dimension domain</th>
<th>Infinite dimension domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix</td>
<td>operator</td>
</tr>
<tr>
<td>covariance matrix</td>
<td>covariance operator (kernel)</td>
</tr>
<tr>
<td>vector</td>
<td>function</td>
</tr>
</tbody>
</table>

1.1 Understand a matrix

**Example 1.** We have a matrix as below and multiply it with two base vectors:

\[
\begin{bmatrix}
3 & 1 \\
1 & 2 \\
3 & 1 \\
1 & 2
\end{bmatrix} \times \begin{bmatrix}
1 \\
0 \\
1 \\
1
\end{bmatrix} = \begin{bmatrix}
3 \\
1 \\
1 \\
2
\end{bmatrix}
\]

Therefore we can have an intuition of a matrix by viewing it column by column

\[
\begin{bmatrix}
1 \\
0
\end{bmatrix} \rightarrow \begin{bmatrix}
3 \\
1
\end{bmatrix}, \begin{bmatrix}
0 \\
1
\end{bmatrix} \rightarrow \begin{bmatrix}
1 \\
2
\end{bmatrix}
\]

which tells us after applying this matrix, the base vectors would rotate and be scaled at what degree.

1.2 Multiplication

**Definition 1.1.** Matrix multiplication of \( \mathbf{A}, \mathbf{B} \) can be presented by two ways

\[
\mathbf{A}\mathbf{B} = \left[ \langle \mathbf{A}_{1,1}, \mathbf{B}_{1,1} \rangle \ldots \langle \mathbf{A}_{1,i}, \mathbf{B}_{1,n} \rangle \langle \mathbf{A}_{2,1}, \mathbf{B}_{2,1} \rangle \ldots \langle \mathbf{A}_{2,i}, \mathbf{B}_{2,n} \rangle \ldots \ldots \ldots \langle \mathbf{A}_{n,1}, \mathbf{B}_{n,1} \rangle \ldots \langle \mathbf{A}_{n,i}, \mathbf{B}_{n,n} \rangle \right]
\]

\( \because \) inner product

\[
= \sum_{i=1}^{n} \mathbf{A}_{i,1} \otimes \mathbf{B}_{i,1}
\]

\( \because \) outer product
1.3 Conjugate transpose

**Definition 1.2.** Conjugate transpose $A^H$ of a matrix $A$ is defined as

$$A = \begin{bmatrix} 1 & -2 - i & 5 \\ 1 + i & i & 4 - 2i \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 + i \\ -2 - i & i \\ 5 & 4 - 2i \end{bmatrix}$$

$$A^H = \begin{bmatrix} 1 & 1 - i \\ -2 + i & -i \\ 5 & 4 + 2i \end{bmatrix}$$

1.4 Orthogonality

**Definition 1.3.** If $A^T A = I$ namely $A^T = A^{-1}$, then $A$ is an orthogonal matrix. The rows(columns) in the orthogonal matrix are mutually orthogonal.

**Example 2.** The rotation matrix is an orthogonal matrix:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

**Remark 1.1.** If $A$ is a orthogonal matrix, then $\text{Det}(A)$ is either 1 or -1.

**Remark 1.2.** $A$ is orthogonal means rows (columns) of $A$ are orthonormal (orthogonal and magnitude is 1) in $\mathbb{R}^{n \times n}$, namely $|\text{Det}(A)| = 1, ||Ax|| = ||x||$. When $A$ is orthogonal and $\text{Det}(A) = 1$, it's called rotation matrix.

**Remark 1.3.** The orthogonal matrices are of interest because their inverse is very cheap to compute.

1.5 Eigen

**Definition 1.4.** If $x, \lambda$ satisfy $Ax = \lambda x$, then $x, \lambda$ are eigenvalue and eigenvector of $A$.

$$Ax = \lambda x$$

$$Ax - \lambda x = 0$$

$$(A - \lambda I)x = 0$$

$$|A - \lambda I| = 0$$
Figure 1: In this shear mapping the red arrow changes direction, but the blue arrow does not. The blue arrow is an eigenvector of this shear mapping because it does not change direction, and since its length is unchanged, its eigenvalue is 1.

**Remark 1.4.** The effect of the transformation brought by $A$ is that the space is scaled by $\lambda(i)$ in direction of $x^{(i)}$.

**Remark 1.5.**

$$\prod_i \lambda^{(i)} = \text{Det}(A)$$

$$\sum_i \lambda^{(i)} = \text{Tr}(A)$$

**Remark 1.6.** Let $A$ be a square $n \times n$ matrix with $n$ linearly independent eigenvectors $x^{(i)}$. Then $A$ can be factorized as

$$A = QAQ^{-1},$$

where $Q$ is the square $n \times n$ matrix whose $i$-th column is the eigenvector $x^{(i)}$ of $A$, and $\Lambda$ is the diagonal matrix whose diagonal elements are the corresponding eigenvalues, $\Lambda_{i,i} = \lambda^{(i)}$. Note that only diagonalizable matrices can be factorized in this way. For example, the defective matrix cannot be diagonalized.

1.6 Inverse

- $(AB)^{-1} = B^{-1}A^{-1}$
- $(ABC\ldots)^{-1} = \ldots C^{-1}B^{-1}A^{-1}$
- $(A^{-1})^T = (A^T)^{-1}$
- $(rA)^{-1} = r^{-1}A^{-1}$

1.7 Transpose

- $(AB)^T = B^T A^T$
- $(ABc\ldots)^T = \ldots c^T B^T A^T$
- $(A + B)^T = A^T + B^T$
- $(A^{-1})^T = (A^T)^{-1}$
- $(rA)^T = rA^T$
1.8 Trace

- $\text{Tr}(AB) = \text{Tr}(BA)$
- $\text{Tr}(ABC) = \text{Tr}(BCA) = \text{Tr}(cAB)$
- $\text{Tr}(A + B) = \text{Tr}(A) + \text{Tr}(B)$
- $\text{Tr}(A^T) = \text{Tr}(A)$
- $\text{Tr}(rA) = r \cdot \text{Tr}(A)$
- $\text{Tr}(P^{-1}AP) = \text{Tr}(A)$

1.9 Determinant

- $\text{Det}(AB) = \text{Det}(A) \cdot \text{Det}(B)$
- $\text{Det}(A^{-1}) = \text{Det}(A)^{-1}$, if $A$ is invertible
- $\text{Det}(A^T) = \text{Det}(A)$
- $\text{Det}(rA) = r^n \cdot \text{Det}(A)$
- $\text{Det}(A) = \prod \lambda^{(i)}$
- $\text{Det}(A) = \prod A_{i,i}$, and $A_{i,i}$ are the eigenvalues of $A$, if $A$ is triangular or diagonal
- $\text{Det}(A)A^{-1} = A^*$

Definition 1.5. **Singular matrix** is the matrix non-invertible $\iff$ Det$(A) = 0 \iff$ rank deficient $\iff$ singular $\iff$ each column not linearly independent

Definition 1.6. **Nonsingular matrix** is the matrix invertible $\iff$ Det$(A) \neq 0 \iff$ full rank $\iff$ nonsingular $\iff$ each column linearly independent

1.10 Logarithm

- $\log(AB) = \log A + \log B$

1.11 Matrix exponential and logarithm

Definition 1.7. **Real exponential** $\exp : \mathbb{R} \to \mathbb{R}$ is defined by

$$\exp(a) = \sum_{i=0}^{\infty} \frac{a^i}{i!} = \frac{a^0}{0!} + \frac{a^1}{1!} + \frac{a^2}{2!} + \frac{a^3}{3!} + \cdots.$$

Definition 1.8. **Matrix exponential** $\exp : \mathbb{R}^{n \times n} \to \mathbb{R}^{n \times n}$ is likewise defined by

$$\exp(A) = \sum_{i=0}^{\infty} \frac{A^i}{i!} = \frac{A^0}{0!} + \frac{A^1}{1!} + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots.$$

where $A^i$ is the matrix multiplication of $i \cdot A$. 
Proof. Given the following ODE
\[ \frac{d}{dt} f(t) = xf(t), \]
where \( t, x \in \mathbb{R}, f(\cdot) : \mathbb{R} \to \mathbb{R} \), we can easily know that
\[ f(t) = c \cdot e^{xt}. \]

When it comes to the matrix \( A \in \mathbb{R}^{n \times n} \) and \( f(\cdot) : \mathbb{R} \to \mathbb{R}^n \), we still expect the matrix to satisfy the form above, namely
\[ \frac{d}{dt} f(t) = Af(t) \]
\[ f(t) = e^{At} \cdot c. \]

To prove it, we substitute the definition of the matrix exponential into the equation above, we have
\[
\frac{d}{dt} f(t) = \frac{d}{dt} \sum_{i=0}^{\infty} \frac{t^i}{i!} A^i \cdot c \\
= \sum_{i=0}^{\infty} \frac{t^i}{i!} A^i \cdot c \\
= \sum_{i=0}^{\infty} \frac{t^{i-1}}{(i-1)!} (A-I)^i \cdot c \\
= A \cdot \sum_{i=0}^{\infty} \frac{t^{i-1}}{(i-1)!} (A-I)^i \cdot c \\
= A \cdot e^{At} \cdot c \\
= Af(t)
\]
which proves the definition of matrix exponential is true and consistent with the original exponential. \( \square \)

Definition 1.9. **Real logarithm** \( \ln : \mathbb{R} \to \mathbb{R} \) is defined by
\[
\ln(a) = \sum_{i=1}^{\infty} (-1)^{i+1} \frac{(a-1)^i}{i} \\
= (a-1) - \frac{(a-1)^2}{2} + \frac{(a-1)^3}{3} - \frac{(a-1)^4}{4} + \cdots.
\]

Definition 1.10. **Matrix logarithm** \( \ln : \mathbb{R}^{n \times n} \to \mathbb{R}^{n \times n} \) is likewise defined by
\[
\ln(A) = \sum_{i=1}^{\infty} (-1)^{i+1} \frac{(A-I)^i}{i} \\
= (A-I) - \frac{(A-I)^2}{2} + \frac{(A-I)^3}{3} - \frac{(A-I)^4}{4} + \cdots,
\]
where \( (A-I)^i \) is the matrix multiplication of \( i \cdot (A-I) \).

1.12 Positive Definite

If \( A, B \) are two positive definite matrices, \( \lambda^{(i)} \) is a eigenvalue of \( A \), then
- \( A + B \) is still positive definite.
\[
\begin{align*}
\forall \lambda^{(i)} > 0. \\
\text{Tr}(A) &= \prod_i \lambda^{(i)} > 0. \\
\text{Det}(A) &= \sum_i \lambda^{(i)} > 0. \\
\text{Any principal submatrix of a positive definite matrix is positive definite.}
\end{align*}
\]

1.13 Decomposition

Definition 1.11. **Cholesky decomposition.** Let \( A \in \mathbb{R}^{n \times n} \) be symmetric and positive definite, then there exists a unique lower triangular matrix \( L \in \mathbb{R}^{n \times n} \) with strictly positive diagonal entries s.t.

\[
A = LL^T
\]

Remark 1.7. Using the Cholesky decomposition, we solve \( LL^T x = b \) instead of \( Ax = b \) by

1. Solving \( Lz = b \) for \( z \) using forward substitution
2. Solving \( L^T x = z \) for \( x \) using backward substitution

Definition 1.12. **LU decomposition.** Let \( A \in \mathbb{R}^{n \times n} \) be invertable and let \( A \) be diagonally dominant, namely \( \forall i, |A_{i,i}| \geq \sum_{j \neq i} |A_{i,j}| \), then there exist a lower triangular matrix \( L \in \mathbb{R}^{n \times n} \) and a upper triangular matrix \( U \in \mathbb{R}^{n \times n} \) s.t.

\[
A = LU
\]

Remark 1.8. Using the LU decomposition, we solve \( LUx = b \) instead of \( Ax = b \) by

1. Solving \( Lz = b \) for \( z \) using forward substitution
2. Solving \( Ux = z \) for \( x \) using backward substitution

Definition 1.13. **Eigendecomposition.** Let \( A \) be a square \( n \times n \) matrix with \( n \) linearly independent eigenvectors \( x^{(i)} \). Then \( A \) can be factorized as

\[
A = Q\Lambda Q^{-1},
\]

where \( Q \) is the square \( n \times n \) matrix whose \( i \)-th column is the eigenvector \( x^{(i)} \) of \( A \), and \( \Lambda \) is the diagonal matrix whose diagonal elements are the corresponding eigenvalues, namely \( \Lambda_{i,i} = \lambda^{(i)} \).

Remark 1.9. When \( A \) is a \( n \times n \) real symmetric matrix can be decomposed as

\[
A = Q\Lambda Q^T,
\]

since the eigenvalues are real and the eigenvectors are orthonormal, therefore \( Q \) is an orthogonal matrix.

Remark 1.10. A square \( n \times n \) matrix \( A \) is called diagonalizable or non-defective if there exists an invertible matrix \( P \) s.t. \( P^{-1}AP \) is a diagonal matrix. Only diagonalizable matrices can be eigendecomposed.

Definition 1.14. **Singular value decomposition (SVD).** Let \( A \in \mathbb{R}^{n \times n} \), then there exist orthogonal matrices \( U, V \in \mathbb{R}^{n \times n},UU^T = I, VV^T = I \) and a diagonal matrix \( \Sigma = \text{diag}(\sigma^{(1)}, \cdots, \sigma^{(n)}) \) \( \forall \sigma^{(i)} \geq 0 \) s.t.

\[
A = U\Sigma V^T,
\]

where \( \sigma^{(i)} \) are called singular values.
Figure 2: Illustration of the singular value decomposition $U\Sigma V^T$ of a real $2 \times 2$ matrix $M$. Down left: The action of $V^T$, is a rotation. Down right: The action of $\Sigma$, a scaling by the singular values $\sigma^{(1)}$ horizontally and $\sigma^{(2)}$ vertically. Top right: The action of $U$ is also a rotation.

Remark 1.11. The SVD can be thought of as decomposing a matrix into a weighted, ordered sum of separable matrices. By separable, we mean that a matrix $A$ can be written as the weighted sum of outer products of vectors. Specifically, the matrix $A$ can be decomposed as

$$A = \sum_{i=1}^{n} \sigma^{(i)} U_{:,i} V_{:,i}^T,$$

where $\sigma^{(i)}$ are the ordered singular values. As a result, SVD is widely used in PCA.

Remark 1.12. For real positive definite symmetric matrix

- The columns of $U$ are eigenvectors of $AA^T$.
- The columns of $V$ are eigenvectors of $A^TA$.
- The non-zero singular values from $\Sigma$ are the square roots of the non-zero eigenvalues of $AA^T$ or $A^TA$.

Remark 1.13. Using the SVD decomposition, we solve $Ax = b$ by

$$x = V\Sigma^{-1}U^Tb$$

Example 3. SVD in image compression The figure below is of the size $450 \times 333$, so we can store this image as a matrix $A \in \mathbb{R}^{450 \times 333}$. 
Now we can decompose $A$ by SVD as below

$$A = \sigma^{(1)}u_{i:1}v_i^T + \sigma^{(2)}u_{i:2}v_i^T + \cdots + \sigma^{(n)}u_{i:n}v_i^T,$$

where $n = \min\{450, 333\}$, $u_i v_i^T$ are matrices of rank 1. And assuming $\sigma^{(1)} \geq \sigma^{(2)} \geq \cdots \geq \sigma^{(n)} \geq 0$. And we can see the influence that adding more terms to the image brings in the figure below.

Figure 3: Original image

Figure 4: From left to right: only keep the first term; keep the first five terms; keep the first twenty terms; keep the first fifty terms

Without any compression techniques, we need to store $450 \times 333 = 149,850$ pixels’ value to present an image. With SVD, if we keep the first fifty terms, only $(1 + 450 + 333) \times 50 = 39,200$ pixels’ value needs to be saved, which is 26% of the original size.

Definition 1.15. **Diagonalization.** Let $A \in \mathbb{R}^{n \times n}$, if there exist invertible matrix $P \in \mathbb{R}^{n \times n}$ s.t.

$$P^{-1}AP = \text{Diag},$$

then $A$ is diagonalizable.

2 Probability and Statistics

**Definition 2.1.** **Likelihood** is the probability of certain observation given the reason, namely

$$\Pr(\text{observation}|\text{reason}).$$

**Definition 2.2.** **Prior probability** is the probability of the reason without any observation, namely

$$\Pr(\text{reason}).$$
Definition 2.3. **Posterior probability** is the probability of the reason with certain observations, namely

\[
Pr(\text{reason}|\text{observation}) = \frac{Pr(\text{observation}|\text{reason})}{Pr(\text{observation})} \times Pr(\text{reason})
\]

\[\propto \text{Likelihood} \times \text{Prior probability}\]

Definition 2.4. **Probability mass function (PMF)** is a discrete function \(f(x)\) that gives the probability that a discrete random variable is exactly equal to some value. The probability is acquired through

\[
Pr(X = a) = f(a).
\]

Definition 2.5. **Probability density function (PDF)** is a continuous function \(f(x)\) that gives the probability of a continuous random variable is located among a certain interval. The probability is acquired through

\[
Pr(a \leq X \leq b) = \int_a^b f(x) \, dx.
\]

Example 4. Suppose bacteria of a certain species typically live 4 to 6 hours. The probability that a bacterium lives exactly 5 hours is equal to zero. A lot of bacteria live for approximately 5 hours, but there is no chance that any given bacterium dies at exactly 5.0000000000... hours. However, the probability that the bacterium dies between 5 hours and 5.01 hours is quantifiable.

Remark 2.1. A PDF must be integrated over an interval to yield a probability.

Definition 2.6. **Marginal probability** can be yield by sum rule

\[
Pr(X = x) = \sum_Y Pr(X = x, Y = y) \quad \text{▷ Discrete}
\]

\[
Pr(x) = \int Pr(x, y) \, dy \quad \text{▷ Continuous}
\]

Definition 2.7. **Conditional probability** can be yield by

\[
Pr(X = x|Y = y) = \frac{Pr(X = x, Y = y)}{Pr(Y = y)}
\]

Example 5. Below is an illustration of marginal probability and conditional probability. The marginal distribution for \(Y\) is akin to integrating along the \(X\) axis. The conditional distribution for \(Y\) at \(X = 1.2\) is cut through the original distribution. Both marginal and conditional distributions of a Gaussian distribution are still Gaussian distributions.
Definition 2.8. **Chain rule of conditional probability** is in form of

\[ \Pr(X^{(1)}, \ldots, X^{(n)}) = \Pr(X^{(1)}) \prod_{i=2}^{n} \Pr(X^{(i)}|X^{(1)}), \ldots, X^{(i-1)}). \]

Definition 2.9. **Cumulative density function (CDF)** is a continuous function \( F(x) \) that gives the probability of a continuous random variable is less than or equal to \( x \).

\[ F(x) = \int_{-\infty}^{x} f(t) \, dt. \]

\[ P(X \leq b) = F(b) \]

Remark 2.2. **PDF is the derivative of the CDF.**

Definition 2.10. **Maximum likelihood estimation (MLE)** is a method of estimating the parameters of a probability distribution by maximizing a likelihood function, so that under the assumed statistical model the observed data is most probable.

Example 6. Suppose that there was only one coin and its probability of tossing a ‘head’ is \( p \), which could have been any value \( 0 \leq p \leq 1 \). The likelihood function to be maximized is

\[ L(p) = \binom{80}{49} p^{49}(1-p)^{31} \]

and the maximization is over all possible values \( 0 \leq p \leq 1 \).

By differentiating \( L(p) \) with respect to \( p \) and setting it to zero, we yield that the maximum likelihood estimator for \( p \) is \( \frac{49}{89} \).

Definition 2.11. **Monte Carlo methods** are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. Monte Carlo methods vary but tend to follow a particular pattern:

1. Define a domain of possible inputs;
2. Generate inputs randomly from a probability distribution over the domain;
3. Perform a deterministic computation on the inputs;
4. Aggregate the results.

Example 7. Consider a quadrant (circular sector) inscribed in a unit square. Given that the ratio of their areas is \( \frac{\pi}{4} \), the value of \( \pi \) can be approximated using a Monte Carlo method:

1. Draw a square, then inscribe a quadrant within it;
2. Uniformly scatter a given number of points over the square;
3. Count the number of points inside the quadrant, i.e. having a distance from the origin of less than 1. The ratio of the inside count and the total sample count is an estimate of the ratio of the two areas, namely the approximation of \( \frac{\pi}{4} \);
4. Multiply the result by 4 to estimate \( \pi \).
Definition 2.12. **Normal distribution** $X \sim \mathcal{N}(\mu, \sigma^2)$ is defined as

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma^2} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, F(x) = \int_0^x \frac{1}{\sqrt{2\pi}\sigma^2} e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt,$$

where $\mu$ is the mean value and $\sigma$ is the standard deviation.

**Remark 2.3.** Some properties of the normal distribution:

- If $X \sim \mathcal{N}(\mu, \sigma^2)$, $a, b$ are real, then $aX + b \sim \mathcal{N}(a\mu + b, (a\sigma)^2)$.

- If $X \sim \mathcal{N}(\mu_X, \sigma_X^2)$ and $Y \sim \mathcal{N}(\mu_Y, \sigma_Y^2)$ are two independent random variables, then $X + Y \sim \mathcal{N}(\mu_X + \mu_Y, \sigma_X^2 + \sigma_Y^2), X - Y \sim \mathcal{N}(\mu_X - \mu_Y, \sigma_X^2 + \sigma_Y^2)$.

**Theorem 2.1. Law of large numbers (LLN).** When $\{X^{(1)}, \ldots, X^{(n)}\}$ is an infinite sequence of independent identically distributed random variables with expectation $E(X^{(1)}) = \cdots = E(X^{(n)}) = \mu$, we can have

$$\bar{X}_n \rightarrow \mu, \text{ when } n \rightarrow \infty,$$

where $\bar{X}_n = \frac{1}{n} \sum_{i=1}^n X_i$.

**Remark 2.4.** We can approximate the expectation of an unknown distribution by performing a sufficient number of trials.

![Figure 5: Law of large numbers](image-url)
**Theorem 2.2. Central limit theorem (CLT).** In many situations, when independent random variables are added, their properly normalized sum tends toward a normal distribution even if the original variables themselves are not normally distributed.

**Remark 2.5.** The theorem is a key concept in probability theory because this theorem allows us to leverage the normal distribution to solve other unknown distributions.

**Example 8.** Given a sequence of i.i.d. random variables \( \{X^{(1)}, \ldots, X^{(n)}\} \), we randomly pick \( m \) (relatively large) samples from the population to calculate the mean value and repeat it \( k \) times. And the \( k \) mean values would be normally distributed.

**Definition 2.13. Student’s t-distribution** is defined as

\[
f(t) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi} \Gamma\left(\frac{\nu}{2}\right)} \left(1 + \frac{t^2}{\nu}\right)^{-\frac{\nu+1}{2}},
\]

where \( \nu \) is the number of degrees of freedom and \( \Gamma \) is the gamma function.

**Definition 2.14. Moment** of a function are quantitative measures related to the shape of the function’s graph.

- 1st order moment: expectation \( E(x) = \int_{-\infty}^{\infty} x \Pr(x) \, dx \)
- 2nd order moment: variance \( \text{Var}(x) = \int_{-\infty}^{\infty} (x - E(x))^2 \Pr(x) \, dx \)
- 3rd order moment: skewness \( S(x) = \int_{-\infty}^{\infty} [x - E(x)]^3 \Pr(x) \, dx \)

**Definition 2.15. Expectation** is a generalization of the weighted average, and is intuitively the arithmetic mean of a large number of independent realizations of \( X \).

\[
E(X) = \int_{\mathbb{R}} x \Pr(x) \, dx
\]

**Definition 2.16. Variance** is the expectation of the squared deviation of a random variable from its population mean or sample mean.

\[
\text{Var}(X) = \int \Pr(x) (x - \mu)^2 \, dx
\]

\[
= E((X - \mu)^2)
\]

\[
= E((X - E(X))^2)
\]

\[
= E(X^2 - 2XE(X) + E(X)^2)
\]

\[
= E(X^2) - 2E(X)E(X) + E(X)^2
\]

\[
= 2E(X^2) - 2E(X)^2
\]

**Definition 2.17. Confidence interval** is a range of values that you can be some percentage (e.g. 95%) certain contains the true mean of the population.
Example 10. Suppose \{s^{(1)}, \ldots, s^{(n)}\} is a sequence of independent samples from a normally distributed population with unknown parameters mean \(\mu\) and variance \(\sigma^2\). Let

\[
\bar{s} = \frac{s^{(1)} + \cdots + s^{(n)}}{n},
\]

\[
\text{Var}(s) = \frac{1}{n-1} \sum_{i=1}^{n} (s^{(i)} - \bar{s})^2,
\]

where \(\bar{s}\) is the sample mean, and \(S^2\) is the sample variance. Then

\[
T = \frac{\bar{s} - \mu}{\sqrt{\text{Var}(s)}/\sqrt{n}}
\]

has a Student’s \(t\) distribution with \(n - 1\) degrees of freedom. Note that the distribution of \(T\) does not depend on the values of the unobservable parameters \(\mu\) and \(\sigma^2\); i.e., it is a pivotal quantity. Suppose we wanted to calculate a 95% confidence interval for \(\mu\). Then, denoting \(c^T\) as the 97.5th percentile of this distribution,

\[
\Pr(-c \leq T \leq c) = 0.95
\]

Note that “97.5th” and “0.95” are correct in the preceding expressions. There is a 2.5% chance that \(T\) will be less than \(-c\) and a 2.5% chance that it will be larger than \(+c\). Thus, the probability that \(T\) will be between \(-c\) and \(+c\) is 95%.

Consequently,

\[
\Pr\left(\bar{s} - \frac{c\sqrt{\text{Var}(s)}}{\sqrt{n}} \leq \mu \leq \bar{s} + \frac{c\sqrt{\text{Var}(s)}}{\sqrt{n}}\right) = 0.95
\]

and we have a theoretical (stochastic) 95% confidence interval for \(\mu\).

After observing the sample we find values \(\bar{s}\) and \(\text{Var}(s)\), from which we compute the confidence interval

\[
\left(\bar{s} - \frac{c\sqrt{\text{Var}(s)}}{\sqrt{n}}, \bar{s} + \frac{c\sqrt{\text{Var}(s)}}{\sqrt{n}}\right).
\]

Definition 2.18. **Covariance** for two random variables \(X\) and \(Y\), each with sample size \(n\) is defined by

\[
\text{Cov}(X, Y) = E((X - E(X))(Y - E(Y)))
\]

\[
= \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})
\]

Definition 2.19. **Covariance matrix** for a set of random variables \(\{X^{(1)}, \ldots, X^{(n)}\}\), each with sample size \(m\) is defined by

\[
\Sigma = \begin{bmatrix}
\text{Cov}(X^{(1)}, X^{(1)}) & \cdots & \text{Cov}(X^{(1)}, X^{(n)}) \\
\vdots & \ddots & \vdots \\
\text{Cov}(X^{(n)}, X^{(1)}) & \cdots & \text{Cov}(X^{(n)}, X^{(n)})
\end{bmatrix}
\]

Remark 2.6. The kernel function in the Gaussian process can be regarded as an infinite-dimensional covariance matrix.

Definition 2.20. **Gaussian random field (GRF)** is a random field involving Gaussian probability density functions of the variables.

\[\text{In practice, we find the } c \text{ in a } t\text{-table}\]
Definition 2.21. A **Gaussian process** is a (potentially infinite) collection of random variables s.t. the joint distribution of every finite subset of random variables is multivariate Gaussian:

\[ f \sim GP(\mu, k), \]

where \( \mu(x) \) and \( k(x, x') \) are the mean and covariance function.

**Remark 2.7.** A Gaussian process is an infinite-variate Gaussian distribution.

**Remark 2.8.** The Gaussian process is a one-dimensional Gaussian random field.

**Remark 2.9.** How to visualize samples from a Gaussian process:

1. Define the mean function and kernel for the GP \( \mu = 0, k(x, x') = \exp\left(-\frac{(x-x')^2}{2\gamma^2}\right) \);
2. Sample inputs \( x, x^{(i)} = \epsilon \times i, i = \{0, 1, \ldots, 1/\epsilon\} \);
3. Compute the kernel matrix \( \Sigma \). For the example with \( \epsilon = 0.1 \), we would have an 11 \( \times \) 11 matrix;
4. Sample from the multivariate Gaussian distribution \( \mathcal{N}(0, \Sigma) \).

Figure 6: Gaussian process sampling

**Remark 2.10.** Just like multi-dimensional Gaussian distribution is described by a mean vector and a covariance matrix, the Gaussian process can also be depicted by a mean vector and a covariance function, since Gaussian processes can be seen as an infinite-dimensional generalization of multivariate normal distributions. Below are some common covariance functions of the Gaussian process:

1. **RBF:** \( k(x, x') = \exp\left(-\frac{(x-x')^2}{2\gamma^2}\right) \)
2. **Linear:** \( k(x, x') = x^T x' \)
3. **Laplace:** \( k(x, x') = \exp\left(-\frac{|x-x'|}{\gamma}\right) \)

Figure 7: Darker color indicates higher value. (a) At first, no training points have been observed. Accordingly, the mean prediction remains at 0 and the standard deviation is the same for each test point. (b) By hovering over the covariance matrix you can see the influence of each point on the current test point. As long as no training points have been observed, the influence of neighboring points is limited locally. As we would expect, the uncertainty of the prediction is small in regions close to the training data and grows as we move further away from those points. In the constrained covariance matrix, we can see that the correlation of neighboring points is affected by the training data. If a predicted point lies in the training data, there is no correlation with other points. Therefore, the function must pass directly through it. Predicted values further away are also affected by the training data—proportional to their distance.

**Definition 2.22.** Kernel $k(\cdot, \cdot) : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a function, where $\mathcal{H}$ is a Hilbert space and $\mathcal{X}$ is a non-empty set, if there exists a function $\phi(\cdot) : \mathcal{X} \to \mathcal{H}$ s.t. for any $x, y \in \mathcal{X}$, we have

$$k(x, y) = \langle \phi(x), \phi(y) \rangle_{\mathcal{H}}$$

**Remark 2.11.** The kernel can be regarded as a distance function, which tells you the similarity of the two samples. In the Gaussian process, the kernel is a covariance function.

**Remark 2.12.** We imposed almost no conditions on $\mathcal{X}$: we don’t even require there to be an inner product defined on the elements of $\mathcal{X}$. Defining the inner product on $\mathcal{H}$ is enough. For example, let $x, y$ represent two different books, we can’t take an inner product between books, but we can take an inner product between the feature maps $\phi(x), \phi(y)$ corresponding to $x, y$.

**Remark 2.13.** The kernel gives a way to compute inner products in some feature space without even knowing what this space is and what is $\phi$. In most cases, we care more about the inner product than the feature mapping itself. We never need the coordinates of the data in the feature space. One example is the Gaussian kernel $k(x, y) = \exp(-\gamma\|x - y\|^2)$. If we Taylor-expand this function, we’ll see that it corresponds to an infinite-dimensional codomain of $\phi$. 

Figure 8: $\phi(x) = [x_1, x_2, x_1x_2]^T$ example of the kernel: on the left, the points are plotted in the original space; on the right, the points are plotted into a higher dimensional feature space by $\phi$.

Remark 2.14. • Stationary kernels, such as the RBF kernel or the periodic kernel, are functions invariant to translations, and the covariance of two points is only dependent on their relative position.

• Non-stationary kernels, such as the linear kernel, do not have this constraint and depend on an absolute location.

Definition 2.23. [Kernel methods] are a type of non-parametric, instance-based machine learning algorithms. Assuming we have known all the labels of training samples $(x^{(i)}, y^{(i)})$, the label for a new input $x$ is predicted by a weighted sum $\sum_i k(x^{(i)}, x) \cdot y^{(i)}$.

Definition 2.24. [Principal components analysis (PCA)] seeks a sequence of linear subspaces that maximize the variance of the data, the intent of which is to find an orthonormal basis $\{e^{(1)}, \ldots, e^{(k)}\}$ of a set of points $\{s^{(1)}, \ldots, s^{(n)}\} \in \mathbb{R}^d$, and the new basis is of the same dimension $d$ as the original space, which satisfies the recursive relationship

$$e^{(1)} = \arg \max_{||e||=1} \sum_{i=1}^{n} (e, s^{(i)})^2$$

$$e^{(2)} = \arg \max_{||e||=1} \sum_{i=1}^{n} [(e^{(1)}, s^{(i)})^2 + (e, s^{(i)})^2]$$

$$\vdots$$

$$e^{(k)} = \arg \max_{||e||=1} \sum_{i=1}^{n} [(e^{(1)}, s^{(i)})^2 + \ldots + (e^{(k-1)}, s^{(i)})^2 + (e, s^{(i)})^2]$$

In other words, the subspace $V_k = \text{span}(\{e^{(1)}, \ldots, e^{(k)}\})$ is the $k$-dimensional subspace that maximizes the variance of the data projected to the subspace. The basis $\{e^{(i)}\}$ is computed as the set of ordered eigenvectors of the sample covariance matrix of the data.

Remark 2.15. Concatenating all the feature vectors together as a matrix, and using SVD to decompose the matrix is a common way of performing PCA.

Example 11. Given a dataset $\{s^{(1)}, \ldots, s^{(n)}\}$ of $n$ samples, where $s^{(i)} \in \mathbb{R}^d$ and $\bar{s}$ is the mean of the dataset

$$s^{(i)} = \begin{bmatrix} s_1^{(i)} \\ \vdots \\ s_d^{(i)} \end{bmatrix}, \bar{s} = \begin{bmatrix} \bar{s}_1 \\ \vdots \\ \bar{s}_d \end{bmatrix}.$$
Then we can have a $d \times d$ covariance matrix $\Sigma$

$$\Sigma = \begin{bmatrix} \text{Cov}(s_1, s_1) & \cdots & \text{Cov}(s_1, s_d) \\ \vdots & \ddots & \vdots \\ \text{Cov}(s_d, s_1) & \cdots & \text{Cov}(s_d, s_d) \end{bmatrix} = \begin{bmatrix} \text{cov of 1}^{\text{st}} \text{ and } 1^{\text{st}} \text{ dim} & \cdots & \text{cov of 1}^{\text{st}} \text{ and } d^{\text{th}} \text{ dim} \\ \vdots & \ddots & \vdots \\ \text{cov of } d^{\text{th}} \text{ and } 1^{\text{st}} \text{ dim} & \cdots & \text{cov of } d^{\text{th}} \text{ and } d^{\text{th}} \text{ dim} \end{bmatrix},$$

where

$$\text{Cov}(s_j, s_k) = \frac{\sum_{i=1}^{n} (s_j^{(i)} - \bar{s}_j)(s_k^{(i)} - \bar{s}_k)}{n}.$$ 

To form a reduced dimension space $S'$, simply pick the largest $k$ eigenvalues of the covariance matrix above and the corresponding eigenvectors, and the new dimension-reduced $s'$ is formed as below

$$\begin{bmatrix} s^{(1)} & \cdots & s^{(n)} \end{bmatrix}_{k \times n} = \begin{bmatrix} e^{(1)T} \\ \vdots \\ e^{(k)T} \end{bmatrix}_{k \times d} \begin{bmatrix} s^{(1)} & \cdots & s^{(n)} \end{bmatrix}_{d \times n}.$$

**Example 12.** Given a dataset $\{s^{(1)}, \ldots, s^{(n)}\}$ of $n$ samples, where $s^{(i)} \in \mathbb{R}^d$. Concatenating all the $s^{(i)}$ together, we can have a matrix $S$ of size $n \times d$.

- If we use the SVD to solve PCA, we simply decompose $S$ to $U\Sigma V^T$, where the $V$ is the eigenvector matrix regarding the dimension covariance, and $U$ is the eigenvector matrix regarding the sample size covariance.

- For PCA, we first yield the covariance matrix of $S$ by $S^T S \in \mathbb{R}^{d \times d}$, then we perform the eigen-decomposition on $S^T S$, as $S^T S = (U \Sigma V^T)^T U \Sigma V^T = V \Sigma U^T U \Sigma V^T = V \Sigma^2 V^T = QAQ^T$ to acquire the largest $k$ eigenvalues and their corresponding eigenvectors.

SVD is widely adopted in PCA because the formation of $S^T S$ can cause the loss of precision, whereas the eigendecomposition is also slower than SVD calculation.

**Definition 2.25.** **Linear Discriminant Analysis (LDA)** projects the input data to a linear subspace consisting of the directions which maximize the separation between classes.

**Remark 2.16.** Differences between PCA and LDA:

- PCA focuses on capturing the direction of maximum variation in the data set;
- LDA focuses on finding a feature subspace that maximizes the separability between the groups.

**Definition 2.26.** **Mahalanobis distance** is a measure of the distance between a point $s^{(i)}$ and a distribution, can be written as

$$d_M(s^{(i)}) = \sqrt{(s^{(i)} - \mu)^T \Sigma^{-1}(s^{(i)} - \mu)},$$

where $\mu$ is the mean vector and $\Sigma$ is the covariance matrix of the distribution.

It can also be defined as a dissimilarity measure between two random vectors $S^i$ and $S^j$ of the same distribution with the covariance matrix $\Sigma$:

$$d_M(s^{(i)}, s^{(j)}) = \sqrt{(s^{(i)} - \mu)^T \Sigma^{-1}(s^{(j)} - \mu)}.$$
Remark 2.17. This intuitive approach can be made quantitative by defining the normalized distance between the test point and the set to be \( \frac{|x - \mu|_2}{\sigma} \). By plugging this into the normal distribution we can derive the probability of the test point belonging to the set. The drawback of the above approach was that we assumed that the sample points are distributed about the center of mass in a spherical manner. Were the distribution to be decidedly non-spherical, for instance ellipsoidal, then we would expect the probability of the test point belonging to the set to depend not only on the distance from the center of mass but also on the direction. In those directions where the ellipsoid has a short axis the test point must be closer, while in those where the axis is long, the test point can be further away from the center. To understand this intuitive approach can be made quantitative by defining the normalized distance between the test point and the set to be \( \frac{|x - \mu|_2}{\sigma} \), Mahalanobis distance \( d_M(s^i) = \sqrt{\sum_{ij} v_i \Sigma_{i,j}^{-1} v_j} \), and set \( p_{ij} = 0 \). Note that \( \sum_j p_{ij} = 1 \) for all \( i \). As Van der Maaten and Hinton explained: “The similarity of datapoint \( S^i \) to datapoint \( S^j \) is the conditional probability, \( p_{ji} \), that \( S^i \) would pick \( S^j \) as its neighbor if neighbors were picked in proportion to their probability density under a Gaussian centered at \( S^j \).” Now define \( p_{ij} = \frac{p_{ji} + p_{ij}}{2n} \) and note that \( p_{ij} = p_{ji}, p_{ii} = 0, \) and \( \sum_{i,j} p_{ij} = 1 \).

The bandwidth of the Gaussian kernels \( \sigma_i \) is set in such a way that the entropy of the conditional distribution equals a predefined entropy using the bisection method. As a result, the bandwidth is adapted to the density of the data: smaller values of \( \sigma_i \) are used in denser parts of the data space.

Since the Gaussian kernel uses the Euclidean distance \( \|S^i - S^j\| \), it is affected by the curse of dimensionality\(^3\) and in high dimensional data when distances lose the ability to discriminate, the \( p_{ij} \) become too similar (asymptotically, they would converge to a constant). It has been proposed to adjust the distances with a power transform, based on the intrinsic dimension of each point, to alleviate this.

\( t \)-SNE aims to learn a \( d \)-dimensional map \( S'_1, \ldots, S'_n \) (with \( S'_i \in \mathbb{R}^d \) and \( d \) typically chosen as 2 or 3) that reflects the similarities \( p_{ij} \) as well as possible. To this end, it measures similarities \( q_{ij} \) between two points in the map \( S'_i \) and \( S'_j \), using a very similar approach. Specifically, for \( i \neq j \), define \( q_{ij} \) as

\[
q_{ij} = \frac{(1 + \|S'_i - S'_j\|^2)^{-1}}{\sum_k \sum_{l \neq k}(1 + \|S'_k - S'_l\|^2)^{-1}}
\]

and set \( q_{ii} = 0 \). Herein a heavy-tailed Student t-distribution (with one degree of freedom, which is the same as a Cauchy distribution) is used to measure similarities between low-dimensional points in order to allow dissimilar objects to be modeled far apart in the map.

\(^3\)This expression was coined by Bellman in 1961 to refer to the fact that many algorithms that work fine in low dimensions become intractable when the input is high-dimensional.
The locations of the points $S_i'$ in the map are determined by minimizing the (non-symmetric) Kullback–Leibler divergence of the distribution $P$ from the distribution $Q$, that is:

$$D_{KL}(P \parallel Q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

The minimization of the Kullback–Leibler divergence with respect to the points $S_i'$ is performed using gradient descent. The result of this optimization is a map that reflects the similarities between the high-dimensional inputs.
3 Inside of a Network

3.1 Techniques

Definition 3.1. **Transfer learning** aims to help improve the learning of the target predictive function \(f_T(\cdot)\) in \(D_T\) using the knowledge in \(D_S\) and \(T_S\), where a source domain \(D_S\) and learning task \(T_S\), a target domain \(D_T\) and learning task \(T_T\), where \(D_S \neq D_T\), or \(T_S \neq T_T\).

Definition 3.2. **Domain adaptation** is the ability to apply an algorithm trained in one or more “source domains” to a different (but related) “target domain”. Domain adaptation is a subcategory of transfer learning. In domain adaptation, the source and target domains all have the same feature space (but different distributions); in contrast, transfer learning includes cases where the target domain’s feature space is different from the source feature space or spaces.

Definition 3.3. **Cross-validation** is a resampling procedure used to evaluate machine learning models on a limited data sample.

Example 14. The general procedure is as follows: Shuffle the dataset randomly. Split the dataset into \(k\) groups. For each unique group:

1. Take the group as a hold-out or test data set;
2. Take the remaining groups as a training data set;
3. Fit a model on the training set and evaluate it on the test set;
4. Retain the evaluation score and discard the model;
5. Summarize the skill of the model using the sample of model evaluation scores.

Remark 3.1. Cross-validation is a method to evaluate the performance of a type of model, e.g. DenseNet, rather than a specific parameterized model, and that is the reason why we will discard the model once we acquired the evaluation score. The architecture (hyper-parameters, which are initialized at the beginning of the training process) stays the same, but the parameters inside of the model vary.

3.2 Common terms

Definition 3.4. One **batch** of samples is a group of samples concatenated together to go through the network before updating the model weights.

<table>
<thead>
<tr>
<th>Types</th>
<th>Batch size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stochastic gradient descent</td>
<td>1</td>
</tr>
<tr>
<td>Mini-Batch gradient descent</td>
<td>(1, size of training dataset)</td>
</tr>
<tr>
<td>Batch gradient descent</td>
<td>size of training dataset</td>
</tr>
</tbody>
</table>

Example 15. In PyTorch’s implementation, the mini-batch concatenates the designated samples together and puts them into the network. For instance, if the image is of size \(128 \times 128 \times 128\), 4 channels, and mini-batch size is 2, then the shape of the input tensor during training is \(2 \times 4 \times 128 \times 128 \times 128\).

You may wonder that when it comes to testing, the input is only one sample at a time, but the model stays the same, how can that still work? The thing is, a model only regulates the shape of the convolution...
kernel, pooling parameters, activation function and how they are organized, the size of the input does not matter, it can either \(2 \times 4 \times 128 \times 128 \times 128\) or \(1 \times 4 \times 128 \times 128 \times 128\). The final loss is actually the summation of all samples’ loss. As long as the output size matches the ground-truth size, the workflow can be performed uninterrupted.

**Definition 3.5.** One **iteration** is a time span when a batch of training data is passed forward and backward through the neural network.

**Definition 3.6.** One **epoch** is a time span when an entire training dataset is passed forward and backward through the neural network.

**Definition 3.7.** In **convolutional layer**, multiple channels of feature maps were extracted by sliding the trainable convolutional kernels across the input feature maps.

**Example 16.** Observation of convolution operation:

1. Kernel tensor features the same channel as the input;
2. Output is the summation across the channel;
3. One kernel tensor is always associated with only one channel in the output tensor, hence the total number of kernel tensors of one convolutional layer is equivalent to the output channel number, so as the bias.

![Convolution Example](https://distill.pub/2016/deconv-checkerboard/?utm_source=researcher_app&utm_medium=referral&utm_campaign=RESR_MRKT_Researcher_inbound)

**Definition 3.8.** In **transposed convolutional layer**, multiple channels of feature maps were extracted by sliding the trainable convolutional kernels across the input feature maps, but expand the output from input, with the special setting of stride and padding.

---

*In the machine learning community, we use the term “transpose convolution” and “deconvolution” interchangeably, while deconvolution is actually a mathematical operation that reverses the effect of convolution. Transposed convolution is basically still convolution.*

---

[21]
Figure 10: Upsampling uses techniques such as padding, strides, and dilation to control the size of the output.

Figure 11: Input tensor provides coefficients for a convolutional filter.

**Definition 3.9.** **Max pooling layer** is a way to reduce the image sizes, to provide an abstracted form of the representation, to reduce the computational cost and to promote spatial invariance of the network.

**Remark 3.2.** When processing multi-channel input data, the pooling layer pools each input channel separately, rather than summing the inputs up over channels as in a convolutional layer. This means that the number of output channels for the pooling layer is the same as the number of input channels.

**Remark 3.3.** The pooling layer can also have configurations of padding and stride.

**Definition 3.10.** **Fully-connected input layer** takes the output of the previous layers, “flattens” them, and turns them into a single vector that can be an input for the next stage.

**Remark 3.4.** It’s easy to understand the fully-connected layer in the multilayer perceptron, as it is neurons connecting neurons. For the convolutional neural network, the fully-connected layer would stretch the last output as a 1D vector, while the final output is also a 1D vector, then it’s actually the same as the FCN in the MLP.

**Remark 3.5.** **Why CNN is a neural network.** Intuitively, the CNN seems quite different from the typical multilayer perceptron (neural network) as the key part of it is sliding the convolution kernel over the whole image. But if we think of it this way, excepting assigning weights to the current sample point’s every dimension, just as MLP does, CNN also assigns weights to the neighbor sample vectors, which incorporates the relationship information into the whole optimization process.
Remark 3.6. Always remember the CNN or MLP are the model doing the calculation point-wisely, forget about the shape of the input tensor.

Figure 12: Simple illustration of a multilayer perceptron model. The first column of perceptrons in the figure stand for the four channels of one sample vector, rather than a flattened input tensor, which may consist of many sample points. The input tensor of four channels of any shape would fits this network, as the framework would do the calculation point-wisely. When it comes to a $3 \times 3$ convolutional layer, there would be $3 \times 3 = 9$ sample vectors concatenated together, making the input a 36-dimension vector. Each column (except the output layer) should have a bias perceptron, which is omitted here.

Remark 3.7. FCN vs. CNN. The fully convolutional networks (FCNs) are extensions of CNNs for pixel-wise prediction, e.g. semantic segmentation. FCNs replace the fully connected layer in CNNs with convolutional layers, and add upsampling layers in the end to recover the input spatial resolution. To put it in another way, FCNs’ outputs are image-like, while CNNs’ outputs are typically vector-like, depending on which kind of task they perform.

Example 17. An MLP implementation for images, by leveraging Conv layer and processing the pixels in an MLP manner all at once.

```python
class MLP(nn.Module):
    def __init__(self, in_channels, out_channels=None, hidden_channels=None, n_layers=2, n_dim=2, non_linearity=F.gelu, dropout=0., **kwargs):
        super().__init__()
        self.n_layers = n_layers
        self.in_channels = in_channels
        self.out_channels = in_channels if out_channels is None else out_channels
        self.hidden_channels = in_channels if hidden_channels is None else hidden_channels
        self.non_linearity = non_linearity
        self.dropout = nn.ModuleList([nn.Dropout(dropout) for _ in range(n_layers)]) if dropout > 0.
        Conv = getattr(nn, f'Conv{n_dim}d')
        self.fcs = nn.ModuleList()
        for i in range(n_layers):
            if i == 0:
                self.fcs.append(Conv(self.in_channels, self.hidden_channels, 1))
            elif i == (n_layers - 1):
                self.fcs.append(Conv(self.out_channels, self.out_channels, 3, padding=1, bias=True))
            else:
                self.fcs.append(Conv(self.hidden_channels, self.hidden_channels, 3, padding=1, bias=True))
```
self.fcs.append(Conv(self.hidden_channels, self.out_channels, 1))
else:
    self.fcs.append(Conv(self.hidden_channels, self.hidden_channels, 1))

def forward(self, x):
    for i, fc in enumerate(self.fcs):
        x = fc(x)
        if i < self.n_layers:
            x = self.non_linearity(x)
        if self.dropout is not None:
            x = self.dropout(x)

    return x

Remark 3.8. **torch.nn.DataParallel** This container parallelizes the application of the given module by splitting the input across the specified devices by chunking in the batch dimension (other objects will be copied once per device). In the forward pass, the module is replicated on each device, and each replica handles a portion of the input. During the backward pass, gradients from each replica are summed into the original module.

The batch size should be larger than the number of GPUs used.

Definition 3.11. **Dropout layer** is a technique to randomly drop units (along with their connections) from the neural network during training, which alleviates data overfitting.

Remark 3.9. Training a neural network with dropout can be seen as training a collection of $2^n$ (n stands for the number of nodes in the networks) thinned networks with extensive weight sharing, where each thinned network gets trained very rarely, if at all.

Remark 3.10. Dropout has a tunable hyperparameter $p$ (the probability of retaining a unit in the network). For any layer $l$, $r^{(l)}$ is a vector of independent Bernoulli random variables each of which has probability $p$ of being 1. Namely

$$r^{(l)}_i \sim \text{Bernoulli}(p)$$

Figure 13: Dropout Neural Net Model. Left: A standard neural net with 2 hidden layers. Right: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped. Courtesy of [16]
Figure 14: Left: A unit at training time that is present with probability $p$ and is connected to units in the next layer with weights $w$. Right: At test time, the unit is always present and the weights are multiplied by $p$. The output at test time is the same as the expected output at training time. Courtesy of [16].

Definition 3.12. **Unsupervised learning algorithms** experience a dataset containing many features, then learn useful properties of the structure of this dataset.

Definition 3.13. **Supervised learning algorithms** experience a dataset containing features, but each example is also associated with a label or target.

Definition 3.14. **Semi-supervised learning algorithms** experience a dataset containing features, but only a portion of the examples are associated with a label or target, and the rest of them are in absence of a label.

Remark 3.11. First, we use the images with correct labels to train a classification model. We then use this classification model to label the unlabelled images. Images with labels of high confidence from the model will then be added to the model with their predicted labels as pseudo-labels for continued training. We iterate this process until all the data are utilized for the best classification model.

Definition 3.15. **Pseudo labeling** is a method that uses a small set of labeled data along with a large amount of unlabeled data to improve a model’s performance.

Example 18. The technique itself is incredibly simple and follows just 4 basic steps:

1. Train model on a batch of labeled data;
2. Use the trained model to predict labels on a batch of unlabeled data;
3. Use the predicted labels to calculate the loss on unlabeled data;
4. Combine labeled loss with unlabeled loss and backpropagate.

Now instead of simply adding the unlabeled loss with the labeled loss, Lee[5] proposes using weights:

$$\text{loss} = \text{loss of labeled data} + \alpha(t) \times \text{loss of unlabeled data}$$

Definition 3.16. **Underfitting** is the phenomenon when the training error is too large. Underfitting occurs when the model cannot capture the underlying trend of data, namely does not fit the data well enough. It usually happens in high-bias but low-variance models, e.g. decision trees.

Definition 3.17. **Overfitting** is the phenomenon when the gap between the training error and testing error is too large. Overfitting occurs when the model captures the noise of the data, namely fits the data too well. It usually happens in high-variance but low-bias models, e.g. random forest.
Example 19. Methods to avoid overfitting: regularization, dropout, pruning, and data augmentation.

Definition 3.18. **Bias** is a learner’s tendency to consistently learn the same wrong thing \cite{1}. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting), e.g. decision tree.

Definition 3.19. **Variance** is the tendency to learn random things irrespective of the real signal \cite{1}. The high variance may result from an algorithm modeling the random noise in the training data (overfitting), e.g. random forest.

![Figure 15: Variance and bias](image)

Definition 3.20. **Capacity** of a model is its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set, whereas models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the testing set.

Definition 3.21. **Regularization** is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error. It’s a way of expressing the preferences for different solutions, both implicitly and explicitly.

Definition 3.22. **$L^1$ regularization** adds $L^1$ penalty to the loss function:

\[
\mathcal{L} = \text{error}(v_{\text{true}} - v_{\text{pred}}) + \lambda \sum \|\theta\| 
\]

Note that it is only differentiable except $\theta = 0$.

Definition 3.23. **$L^2$ regularization** adds $L^2$ penalty to the loss function:

\[
\mathcal{L} = \text{error}(v_{\text{true}} - v_{\text{pred}}) + \lambda \sum \|\theta\|^2
\]

Remark 3.12. **Difference between the $L^1$ and $L^2$ regularization.** $L^1$ shrinks the less important feature’s coefficient to zero, thus removing some features altogether. So, this works well for feature selection in case we have a huge number of features. While it’s impossible for $L^2$ regularization to cut down small coefficients to zero the shrinking speed lower drastically when coefficients near zero.

Example 20. Weight decay is one kind of regularizer in linear regression, which expresses a preference for the weights to have smaller squared norm:

\[
L(\theta) = \text{MSE}_{\text{train}} + \lambda \theta^T \theta
\]
Definition 3.24. **Generative model** is a class of statistical models that can generate new data instances. These models are used in unsupervised machine learning.

Example 21. Naïve Bayes, Bayesian networks, Markov random fields, hidden Markov models (HMMs), latent Dirichlet allocation (LDA), and generative adversarial networks (GANs).

Definition 3.25. **Discriminative model** is a class of models used in statistical classification, mainly used for supervised machine learning. These types of models are also known as conditional models since they learn the boundaries between classes or labels in a dataset.

Example 22. Logistic regression, SVM, neural networks, k-NN, CRF, decision trees, and random forest.

Definition 3.26. **Gradient vanishing** occurs when the derivative or slope will get smaller and smaller as we go backward with every layer during backpropagation.

Remark 3.13. A vanishing Gradient problem occurs with the sigmoid and tanh activation function because the derivatives of the sigmoid and tanh activation functions are between 0 to 0.25 and 0-1.

Definition 3.27. **Gradient explosion** occurs when the derivatives or slope will get larger and larger as we go backward with every layer during backpropagation.

Remark 3.14. Gradient explosion happens because of weights, not because of the activation function. Due to high weight values, the derivatives will also be higher so that the new weight varies a lot from the older weight, and the gradient will never converge.

Table 1: Parameter number calculation for different types of layer

<table>
<thead>
<tr>
<th>Layer</th>
<th>Parameters number</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Transposed-)Convolutional Layer</td>
<td>(input channel # × kernel size + 1(bias)) × output channel #</td>
</tr>
<tr>
<td>Pooling Layer</td>
<td>0</td>
</tr>
<tr>
<td>Dropout Layer</td>
<td>0</td>
</tr>
<tr>
<td>Activation Layer</td>
<td>0(others) or 1(α in ELU)</td>
</tr>
<tr>
<td>Linear Layer</td>
<td>(input channel # × 1 + 1(bias)) × output channel #</td>
</tr>
<tr>
<td>Batch Normalization Layer</td>
<td>input channel # × 4</td>
</tr>
</tbody>
</table>

Definition 3.28. **Curse of dimensionality** refers to the fact that many algorithms that work fine in low dimensions become intractable when the input is high dimensional. In another word, our intuitions, which come from a three-dimensional world, often do not apply in high-dimensional ones.

Remark 3.15. Generalizing correctly becomes exponentially harder as the dimensionality of the examples grows because a fixed-size training set covers a dwindling fraction of the input space. Even with a moderate dimension of 100 and a huge training set of a trillion examples, the latter covers only a fraction of about 10–18 of the input space. This is what makes machine learning both necessary and hard.

Remark 3.16. Naively, one might think that gathering more features never hurts, since at worst they provide no new information about the class. But in fact, their benefits may be outweighed by the curse of dimensionality.
Example 23. In high dimensions, all examples look alike. Suppose, for instance, that examples are laid out on a regular grid, and consider a test example \( s(t) \). If the grid is \( d \)-dimensional, \( s(t) \)'s 2d nearest examples are all at the same distance from it. So as the dimensionality increases, more and more examples become nearest neighbors of \( s(t) \), until the choice of nearest neighbor (and therefore of class) is effectively random.

3.3 Loss Functions

Definition 3.29. **Confusion matrix** is a \( 2 \times 2 \) table that contains 4 outputs provided by the binary classifier.

<table>
<thead>
<tr>
<th>Ground truth</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td>True positive(TP)</td>
</tr>
<tr>
<td>Negative</td>
<td>False positive(FP)</td>
</tr>
</tbody>
</table>

\[
\text{precision} = \frac{TP}{TP + FP}
\]

\[
\text{recall} = \frac{TP}{TP + FN}
\]

Definition 3.30. **Precision** is the percentage of positive identifications that were actually correct.

\[
\text{precision} = \frac{TP}{TP + FP}
\]

Definition 3.31. **Recall** is the percentage of actual positive that was identified correctly.

\[
\text{recall} = \frac{TP}{TP + FN}
\]

Remark 3.17. Positive predictive value (PPV) vs. negative predictive value (NPV):

- **PPV** is the precision on the positive side. In medical studies, it reads as the percentage of the positively identified patients are truly positive.

- **NPV** is the precision on the negative side. In medical study, it reads as the percentage of the negatively identified patients are truly negative.

Remark 3.18. Sensitivity vs. specificity:

- **Sensitivity** is the recall on positive side. In medical study, it reads as the percentage of the real patients are identified as patients.
• **Specificity** is the recall on negative side. In medical study, it reads as the percentage of the real non-patients are identified as non-patients.

**Definition 3.32.** **Accuracy** is the fraction of predictions identified correctly.

\[
\text{accuracy} = \frac{TP + TN}{TP + FN + TN + FP}
\]

**Definition 3.33.** **F-score** is the harmonic mean of precision and recall:

\[
F_\beta = \frac{(1 + \beta^2) \cdot TP}{(1 + \beta^2) \cdot TP + \beta^2 \cdot FN + FP}
\]

where \(\beta\) is chosen s.t. recall is considered \(\beta\) times as important as precision.

**Example 24.** \(F_1\) score is a special case of F-score

\[
F_1 = \frac{2 \times TP}{2 \times TP + FN + FP}
\]

**Definition 3.34.** **ROC** curve plots true positive rate (TPR) vs. false positive rate (FPR) at different classification thresholds, where

\[
\text{TPR} = \frac{TP}{TP + FN}, \quad \text{FPR} = \frac{FP}{FP + TN}
\]

**Definition 3.35.** **Area under the ROC Curve (AUC)** measures the entire two-dimensional area underneath the entire ROC curve from \((0, 0)\) to \((1, 1)\).

**Remark 3.19.** When the AUC is high but the accuracy is low, it means that the classifier performs well on the positive class while not that good on identifying negative samples.

![Figure 16: ROC and AUC.](image)

**Definition 3.36.** **Intersection of Union (Jaccard index, IoU)** is defined as

\[
\text{IoU} = \frac{|A \cap B|}{|A \cup B|} = \frac{TP}{TP + FP + FN}
\]
Definition 3.37. **Dice loss function** is defined as

\[ L_{\text{dice}} = -\frac{2|A \cap B|}{|A| + |B|} = -\frac{TP}{2 \times TP + FP + FN} = \frac{2 \times \langle v_{\text{true}}, v_{\text{pred}} \rangle}{\|v_{\text{true}}\|^2 + \|v_{\text{pred}}\|^2 + \epsilon} \in (-1, 0], \]

where \(v_{\text{true}}, v_{\text{pred}} \in \mathbb{R}^{h \times w \times d}\) are **one-hot vectors**, and \(\epsilon\) is a small constant to avoid zero division.

**Remark 3.20.** The difference between the IoU and Dice scores:

\[
\text{IoU} = \frac{TP}{TP + FP + FN}, \\
\text{Dice} = \frac{TP}{2 \times TP + FP + FN}
\]

Definition 3.38. **\(L^1\) loss function** is defined as

\[ L_{L^1} = \|v_{\text{true}} - v_{\text{pred}}\|_1, \]

\[ = \sum_{i=1}^{n}|v_{i}^{\text{pred}} - v_{i}^{\text{true}}|, \]

where \(v_{\text{true}}, v_{\text{pred}}\) are vectors.

Definition 3.39. **\(L^2\) loss function** is defined as

\[ L_{L^2} = \|v_{\text{true}} - v_{\text{pred}}\|_2^2, \]

\[ = \sum_{i=1}^{n}|v_{i}^{\text{pred}} - v_{i}^{\text{true}}|^2, \]

where \(v_{\text{true}}, v_{\text{pred}}\) are vectors.

**Remark 3.21.** \(L^2\) loss function is essentially the maximum likelihood estimation of the normal distribution. For the loss function, we need its value to be as small as possible, while for MLE, we want it as large as possible. Maximization of MLE is equivalent to minimization of MSE if the observation follows the normal distribution.

**Proof.** Assuming the estimation \(X^{(i)}\) follows normal distribution \(\text{Pr}_{\text{model}}(X^{(i)} = x^{(i)}; \theta) = \mathcal{N}(x^{(i)}, \sigma^2)\), the mean of which is the true value \(\hat{x}^{(i)}\). Then we can have

\[
\theta_{\text{ML}} = \arg \max_{\theta} \prod_{i=1}^{n} P_{\text{model}}(X = x^{(i)}; \theta) \\
= \arg \max_{\theta} \sum_{i=1}^{n} \log(P_{\text{model}}(X = \hat{x}^{(i)}; \theta)) \\
= \arg \max_{\theta} \sum_{i=1}^{n} \log \left( \frac{1}{\sqrt{2\pi\sigma}} \exp \left( -\frac{(x^{(i)} - \hat{x}^{(i)})^2}{2\sigma^2} \right) \right) \\
= \arg \max_{\theta} \sum_{i=1}^{n} \left( -\log(\sqrt{2\pi\sigma}) - \frac{(x^{(i)} - \hat{x}^{(i)})^2}{2\sigma^2} \right) \\
= \arg \min_{\theta} \left( \sum_{i=1}^{n} ((x^{(i)} - \hat{x}^{(i)})^2) \right) \quad \triangleright \text{\(L^2\) loss}
\]
Definition 3.40. **Mean squared error (MSE) loss function** is defined as

\[ \mathcal{L}_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^{n} (v_{i}^{\text{pred}} - v_{i}^{\text{true}})^2. \]

Remark 3.22.

\( \mathcal{L}^2 \text{ norm} \xrightarrow{\text{squared}} \mathcal{L}^2 \text{ loss} \xrightarrow{\text{averaged}} \text{MSE loss} \)

Definition 3.41. The **least-squares method** finds the optimal parameter values by minimizing the sum of squared residuals

\[ \mathcal{L} = \sum_{i=1}^{n} r(i)^2, \]

where \( r(i) = y(i) - f(x(i), \theta). \)

Remark 3.23. The least squares method is a method that builds a model, whereas mean squared error is a metric that evaluates your model’s performance.

Definition 3.42. **Hausdorff distance** is defined as

\[ d_H(X, Y) = \max \left\{ \sup_{x \in X} \inf_{y \in Y} d(x, y), \sup_{y \in Y} \inf_{x \in X} d(x, y) \right\} \]

where \( X \) and \( Y \) be two non-empty subsets of a metric space \((M, d)\).

Remark 3.24. If entropy is closer to 1, means the distribution is of a high level of disorder (low level of purity, each class evenly distributed). If entropy is closer to 0, means the distribution is of a low level of disorder (high level of purity, only one class).
**Definition 3.44.** Cross-entropy loss function of the discrete probability distribution $p$ and $q$ over a given set is defined as

$$H(p, q) = -\sum_{x^{(i)} \in X} p(X = x^{(i)}) \log(q(X = x^{(i)})) = -\langle p, \log(q) \rangle \in [0, \infty),$$

which measures the distance between the distributions.

**Remark 3.25.** Cross-entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label.

**Remark 3.26.** If there are only two categories, then it is called binary cross-entropy (BCE).

**Remark 3.27.** Cross-entropy is essentially the maximum likelihood estimation of the data distribution.

**Proof.** Assuming the estimated probability of category $c_i$ from our model to be $q(X = c_i; \theta)$, and denote the real frequency of sample belonging to category $c_i$ by $p(X = c_i)$. After sampling $N = \sum p(X = c_i)$ conditionally independent samples, which construct the training set, the likelihood of the parameters $\theta$ of the model $q(X = c_i; \theta)$ on the training set is

$$P_{\text{model}}(x; \theta) = q(X = c_1) \times \cdots \times q(X = c_1) \times q(X = c_2) \times \cdots \times q(X = c_2) \times \cdots$$

We want to find parameter $\theta$ s.t. to maximize the likelihood, then we have

$$\theta_{\text{ML}} = \arg \max_{\theta} \left( \frac{q(X = c_1) \times \cdots \times q(X = c_1) \times q(X = c_2) \times \cdots \times q(X = c_2) \times \cdots}{\frac{p(X = c_1)}{p(X = c_2)} \times \cdots \times \frac{p(X = c_1)}{p(X = c_2)} \times \cdots} \right)$$

$$= \arg \max_{\theta} \left( \prod_{i} q(X = c_i)^{p(X = c_i)} \right) \quad \triangleright C: \text{category number}$$

$$= \arg \max_{\theta} \left( \sum_{i} \log \left( q(X = c_i)^{p(X = c_i)} \right) \right)$$

$$= \arg \max_{\theta} \left( \sum_{i} (p(X = c_i) \log(q(X = c_i)) \right) \quad \triangleright \text{Opposite of cross-entropy}$$

In the case of one-hot vector distance measurement, the frequency of the right category $p(\hat{x})$ would always be one, while the frequencies of the other category are always zero. In other words, cross-entropy only cares about the prediction of the position of the right category.
Example 25. As for the two discrete distributions below:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Class A</th>
<th>Class B</th>
<th>Class C</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>q</td>
<td>0.1</td>
<td>0.2</td>
<td>0.7</td>
</tr>
</tbody>
</table>

we can have the cross entropy

\[ H(p, q) = -(0 \times \log(0.1) + 0 \times \log(0.2) + 1 \times \log(0.7)) = 0.15 \]

Remark 3.28. According to the example above, the only thing that contributes to the value of the loss is the predicted possibility of the ground-truth class, since the possibilities of other classes are all wiped out by 0. The closer the predicted possibility of the ground-truth class to 1, the lower loss will be.

Figure 19: \( y = -\log(x) \)

Definition 3.45. **Triplet loss function** is defined as

\[ \mathcal{L}(x^A, x^P, x^N) = \max(\|f(x^A) - f(x^P)\|^2 - \|f(x^A) - f(x^N)\|^2 + \alpha, 0) \]

where \( x^A \) is an anchor input, \( x^P \) is a positive input of the same class as \( x^A \), \( x^N \) is a negative input of a different class from \( x^A \), \( \alpha \) is a margin between positive and negative pairs, and \( f \) is an embedding function.

Definition 3.46. **Hinge loss function** is defined as

\[ \mathcal{L} = \max(0, 1 - y_{\text{true}} \times y_{\text{pred}}). \]

When \( y_{\text{true}} \) and \( y_{\text{pred}} \) have the same sign (meaning \( y_{\text{pred}} \) predicts the right class) and \( y_{\text{true}} \times y_{\text{pred}} \geq 1 \), the hinge loss \( \mathcal{L} = 0 \). When they have opposite signs, \( \mathcal{L} \) increases linearly with \( y_{\text{pred}} \), and similarly if \( y_{\text{true}} \times y_{\text{pred}} < 1 \), even if it has the same sign (correct prediction, but not by enough margin).
**Definition 3.47.** [**Kullback–Leibler (KL) divergence**] is defined as

\[ D_{KL}(p, q) = \sum_{x \in \mathcal{X}} p(x) \log \left( \frac{p(x)}{q(x)} \right) = -\sum_{x \in \mathcal{X}} p(x) \log q(x) + \sum_{x \in \mathcal{X}} p(x) \log p(x) = H(p, q) - H(p) \in [0, \infty) = \text{cross entropy} - \text{entropy}, \]

**Remark 3.29.** KL divergence describes how different \( q \) is from \( p \) from the perspective of \( p \). When \( p \) is fixed, just as the ground-truth data in training, minimizing KL divergence and cross-entropy is equivalent, as \( H(p) \) is a constant.

**Example 26.** As for the two discrete distributions below:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Class A</th>
<th>Class B</th>
<th>Class C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>0.1</td>
<td>0.2</td>
<td>0.7</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>1 – 1( -\infty )</td>
<td>0</td>
<td>1( -\infty )</td>
</tr>
</tbody>
</table>

we can have the KL divergence

\[ D_{KL}(p, q_1) = (0 \times \log(0) + 0 \times \log(0) + 1 \times \log(1)) - (0 \times \log(0.1) + 0 \times \log(0.2) + 1 \times \log(0.7)) = 0.15 \]

When the two distribution are very far away just like \( p, q_2 \), \( D_{KL}(p, q_2) = \infty \), which is meaningless.

**Definition 3.48.** [**Jensen–Shannon (JS) divergence**] is defined as

\[ D_{JS}(p, q) = \frac{1}{2}D_{KL}(p, \frac{1}{2}(p + q)) + \frac{1}{2}D_{KL}(q, \frac{1}{2}(p + q)) \]

**Definition 3.49.** [**Wasserstein distance**] is defined as

\[ W_p(\mu, \nu) = \left( \inf_{\gamma \in \Gamma(\mu, \nu)} \int_{M \times M} \text{dist}(x, y)^p d\gamma(x, y) \right)^{1/p}, \]

\[ = \left( \inf_{\gamma \in \Gamma(\mu, \nu)} \int_{M \times M} \text{dist}(x, y)^p \gamma(x, y)dx dy \right)^{1/p}, \]

\[ W_2(\mu, \nu) = \left( \inf_{\gamma \in \Gamma(\mu, \nu)} \int_{M \times M} \text{dist}(x, y)^2 \gamma(x, y) dx dy \right)^{1/2}, \]

where \( \Gamma(\mu, \nu) \) denotes the set of all coupling of marginal distribution \( \mu \) and \( \nu \), and \( x, y \) are actually indicating the position in the respective distribution.

In the discrete case, the distance reads as

\[ W_p(\mu, \nu) = \min_{T \in \Gamma(\mu, \nu)} \langle T, M_{\mu\nu} \rangle = \min_{T \in \Gamma(\mu, \nu)} \text{Tr}(T^T M_{\mu\nu}), \]

where \( M_{\mu\nu} = [\text{dist}(x^{(i)}, y^{(j)})]^p_{ij} \in \mathbb{R}^{m \times n} \) and \( \Gamma(\mu, \nu) = \{ T \in \mathbb{R}^{m \times n}_+ | T \mathbb{1}_m = a, T^T \mathbb{1}_n = b \} \), namely the transport map matrix \( T \) is under the constraint that the sum of each row and column equals to \( a \) and \( b \), respectively. The two matrices correspond to \( \text{dist}(x, y) \) and \( \gamma(x, y) \) in continuous form.
Remark 3.30. Wasserstein distance is closely related to the optimal transport problem. That is, for two distributions of mass $\mu(x), \nu(y)$ in the space $S$, where $x, y \in S$, we wish to transport the mass at the lowest cost. The problem only makes sense when the sums of two distributions are identical, fortunately, $\mu(x), \nu(y)$ are two probability distributions, namely the sum of mass equals 1, and this premise will be satisfied.

Assuming there is also a cost function $c(x, y) \to [0, +\infty)$ which indicates the cost of transporting unit mass from point $x$ to point $y$. Function $\gamma(x, y)$ depicts a transport plan which gives the amount of mass moved from point $x$ to point $y$. Therefore, the cost of the whole transport plan equal to

$$\int \int c(x, y)\gamma(x, y)dxdy,$$

and the Wasserstein distance is exactly the cost of the optimal transport plan.

Lemma 3.1. The $p$-Wasserstein distance between the two probability measures $\mu$ and $\nu$ on $\mathbb{R}^1$ has the following closed-form expression:

$$W_p(\mu, \nu) = \left(\int_{-\infty}^{+\infty} |U(s) - V(s)|^pds\right)^{1/p} \overset{\triangleright}{=} \int \text{quantity} \times \text{unit distance}$$

$$= \left(\int_{0}^{1} |U^{-1}(t) - V^{-1}(t)|^pdt\right)^{1/p}, \overset{\triangleright}{=} \int \text{distance} \times \text{unit quantity}$$

where $U$ and $V$ are the CDFs of $\mu, \nu$ respectively.

Proof. Assuming $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)})\} \subset \text{supp}(\gamma^*)$, $x^{(1)} < x^{(2)}$, where $\gamma^*$ denotes the optimal transport plan. Given the previous assumption, we claim $y^{(1)} \leq y^{(2)}$.

Supposing that $y^{(1)} \leq y^{(2)}$ is not the case, namely $y^{(1)} > y^{(2)}$, which yields

$$|x^{(1)} - y^{(2)}|^p + |x^{(2)} - y^{(1)}|^p < |x^{(1)} - y^{(1)}|^p + |x^{(2)} - y^{(2)}|^p$$

However this inequality suggests that $\{(x^{(1)}, y^{(2)}), (x^{(2)}, y^{(1)})\} \subset \text{supp}(\gamma^*)$, rather than $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)})\} \subset \text{supp}(\gamma^*)$, which contradicts the initial assumption, namely the optimality of $\gamma^*$, as it indicates that $\gamma^*$ is no cyclically monotone.

---

8The support of a probability distribution can be loosely thought of as the closure of the set of possible values of a random variable having that distribution.

9For a more detailed deviation of this inequality in the case of $p > 1$, refers to Appendix A in https://arxiv.org/pdf/1509.02237.pdf
Now, for \( x \in \text{supp}(\mu), y \in \text{supp}(\nu) \), we claim that \((x, y) \in \text{supp}(\gamma^*)\) if and only if \(U(x) = V(y)\). To see this, note that form the monotonicity property we just built, we deduce that \((x, y) \in \text{supp}(\gamma^*)\) if and only if

\[
\gamma^*([\cdot, y]) = \gamma^*([\cdot, x], [\cdot, y]) = \gamma^*([\cdot, x], \mathbb{R})
\]

In turn, the fact that \(\gamma^* \in \Gamma(\mu, \nu)\) implies that \(\gamma^*([\cdot, x], \mathbb{R}) = F(x)\) and \(\gamma^*([\cdot, y]) = G(y)\). From previous relation, we conclude that

\[
W_p(\mu, \nu) = \left( \inf_{\gamma \in \Gamma(\mu, \nu)} \int_{M \times M} \text{dist}(x, y)^p \gamma(x, y) \, dx \, dy \right)^{1/p} = \left( \int_0^1 |F^{-1}(t) - G^{-1}(t)|^p \, dt \right)^{1/p}
\]

**Example 27.** For one dimensional discrete case, to transport \(\nu\) to \(\mu\),

1. 4 extra squares would be moved from 0 to 1;
2. 3 extra squares would be moved from 1 to 2;
3. 2 extra squares would be moved from 2 to 3;
4. 1 extra squares would be moved from 3 to 4.

The “earth” need to be moved is exactly the difference between the two CDFs at each location. Therefore the \(p\)-Wasserstein distance equals to \((\sum |U(s) - V(s)|^p \, ds)^{1/p} = (\sum 4^p \times 1 + 3^p \times 1 + 2^p \times 1 + 1^p \times 1)^{1/p}\)
Remark 3.31. Relationship between KL divergence, JS divergence and Wasserstein distance:

- Intuitively, KL divergence looks like a distance between two distributions, however, $D_{KL}(p,q) \neq D_{KL}(q,p)$, namely it is asymmetric. So comes the JS divergence.

- When the two distributions are far apart, the KL divergence cannot reflect the distance between the distributions while JS divergence is constant, which is deadly for backpropagation in neural network. Nevertheless, the Wasserstein distance can tackle this drawback of KL/JS divergence, as the optimal transport plan of two distant distributions would always make sense and variable.

Remark 3.32. Advantages of Wasserstein distance:

- By leveraging Wasserstein distance, we can get a better average/summary image of two distribution.

Figure 23: Top: Some random circles. Bottom left: Euclidean average of the circles. Bottom right: Wasserstein barycenter.

- When we are creating a geodesic between two distributions $P_0$, $P_1$, and $P_t$ interpolates between them, Wasserstein distance can preserve the basic structure of the distribution.

Figure 24: Top row: Geodesic path from $P_0$ to $P_1$. Bottom row: Euclidean path ($P_t = tP_0 + (1-t)P_1$) from $P_0$ to $P_1$.

- Wasserstein distance is insensitive to small wiggles.

3.4 Normalization

Normalization is used to reduce internal covariate shift among the training samples.

\footnotesize{Figures in this remark credit to https://www.stat.cmu.edu/~larry/=sml/Opt.pdf}
Definition 3.50. **Batch normalization** is defined as

\[
\mu_c = \frac{1}{HWN} \sum_n \sum_w \sum_h x_{cnhw}
\]

\[
\sigma^2_c = \frac{1}{HWN} \sum_n \sum_w \sum_h (x_{cnhw} - \mu_c)^2
\]

\[
\hat{x}_c = \frac{x_c - \mu_c}{\sqrt{\sigma^2_c + \epsilon}}
\]

where \( x \) are the values of input over a mini-batch, \( C, N, H, W \) are the channel, batch, height, and width size, respectively. Batch normalization should be performed by each channel.

Remark 3.33. The inventors of batch normalization postulated informally that this drift in the distribution of such variables could hamper the convergence of the network. Intuitively, we might conjecture that if one layer has variable values that are 100 times that of another layer, this might necessitate compensatory adjustments in the learning rates.

Remark 3.34. **Batch size matters.** When applying batch normalization, the choice of batch size may be even more significant than without batch normalization. In some preliminary research, [17] and [11] relate the properties of batch normalization to Bayesian priors and penalties respectively. This sheds some light on the puzzle of why batch normalization works best for moderate minibatch sizes in the [50, 100] range.

Remark 3.35. **BN in training and testing differentiates slightly.** Typically, after training, we use the entire dataset to compute stable estimates of the variable statistics and then fix them at prediction time. Consequently, batch normalization behaves differently during training and at test time. Recall that dropout also exhibits this characteristic.

Remark 3.36. **BN is typically implemented after the convolutional or fully-connected layer and before the activation function.**

Definition 3.51. **Layer normalization** is defined as

\[
\mu_n = \frac{1}{HWC} \sum_c \sum_w \sum_h x_{cnhw}
\]

\[
\sigma^2_n = \frac{1}{HWC} \sum_c \sum_w \sum_h (x_{cnhw} - \mu_n)^2
\]

\[
\hat{x}_n = \frac{x_n - \mu_n}{\sqrt{\sigma^2_n + \epsilon}}
\]

where \( x \) are the values of input over a mini-batch, \( C, N, H, W \) are the channel, batch, height, and width size, respectively. Layer normalization should be performed by each sample in a mini-batch.

Definition 3.52. **Instance normalization** is defined as

\[
\mu_{cn} = \frac{1}{HW} \sum_w \sum_h x_{cnhw}
\]

\[
\sigma^2_{cn} = \frac{1}{HW} \sum_w \sum_h (x_{cnhw} - \mu_{cn})^2
\]

\[
\hat{x}_{cn} = \frac{x_{cn} - \mu_{cn}}{\sqrt{\sigma^2_{cn} + \epsilon}}
\]
where $x$ are the values of input over a mini-batch, $C, N, H, W$ are the channel, batch, height, width size, respectively. Instance normalization should be performed by each channel and sample in a batch.

**Definition 3.53.** Group normalization \cite{Wu2016} is a middle ground between layer and instance normalization, which is defined as

$$
\mu_{gn} = \frac{1}{|G_i|} \sum_{c \in G_i} \sum_{w} \sum_{h} x_{cnhw}
$$

$$
\sigma^2_{gn} = \frac{1}{|G_i|} \sum_{c \in G_i} \sum_{w} \sum_{h} (x_{cnhw} - \mu_n)^2
$$

$$
\hat{x}_{gn} = \frac{x_n - \mu_n}{\sqrt{\sigma_n^2 + \epsilon}}
$$

where $x$ are the values of input over a mini-batch, $C, N, H, W, |G|$ are the channel, batch, height, width, group size, respectively. Instance normalization should be performed by each group among the channel set.

**Figure 25:** Normalization methods. Each subplot shows a feature map tensor, with $N$ as the batch axis, $C$ as the channel axis, and $(H, W)$ as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

### 3.5 Activation Function

<table>
<thead>
<tr>
<th>Types</th>
<th>Functions</th>
<th>Range</th>
<th>Order of continuity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sigmoid(logistic)</td>
<td>$\sigma(x) = \frac{1}{1 + e^{-x}}$</td>
<td>$(0, 1)$</td>
<td>$C^\infty$</td>
</tr>
<tr>
<td>Softplus</td>
<td>$\zeta(x) = \log(1 + e^x)$</td>
<td>$(0, \infty)$</td>
<td>$C^\infty$</td>
</tr>
<tr>
<td>Tanh(hyperbolic tangent)</td>
<td>$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$</td>
<td>$(-1, 1)$</td>
<td>$C^\infty$</td>
</tr>
<tr>
<td>ReLU(rectified linear unit)</td>
<td>$\text{ReLU}(x) = \max{0, x}$</td>
<td>$[0, +\infty)$</td>
<td>$C^0$</td>
</tr>
<tr>
<td>Leaky ReLU</td>
<td>$\text{LeakyReLU}(x) = \max{0.01x, x}$</td>
<td>$(-\infty, +\infty)$</td>
<td>$C^0$</td>
</tr>
</tbody>
</table>
| ELU(exponential linear unit) | $\begin{cases} 
\alpha (e^x - 1) & \text{if } x \leq 0 \\
        x & \text{if } x > 0
\end{cases}$ | $(-\alpha, +\infty)$ | $\begin{cases} 
C^1 & \text{if } \alpha = 1 \\
C^0 & \text{otherwise}
\end{cases}$ |
| GELU(gaussian error linear unit) | $\frac{1}{2}x \left(1 + \text{erf} \left(\frac{x}{\sqrt{2}}\right)\right)$ | $(-0.17, +\infty)$ | $C^\infty$         |
| SiLU(Sigmoid linear unit) | $x \cdot \sigma(x)$                      | $(-0.28, +\infty)$ | $C^\infty$         |
| Smooth ReLU(Softplus)  | $\ln(1 + e^x)$                               | $(0, +\infty)$  | $C^\infty$         |
The following table lists activation functions that are not functions of a single fold $x$ from the previous layer or layers:

<table>
<thead>
<tr>
<th>Types</th>
<th>Functions</th>
<th>Range</th>
<th>Order of continuity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Softmax</td>
<td>$\frac{e^{x(i)}}{\sum_{j=1}^{J} e^{x(j)}}$</td>
<td>$(0, 1)$</td>
<td>$C^\infty$</td>
</tr>
<tr>
<td>Maxout</td>
<td>$\max_i x^{(i)}$</td>
<td>$(-\infty, \infty)$</td>
<td>$C^0$</td>
</tr>
</tbody>
</table>

Remark 3.37. Activation functions have different mathematical properties:

- **Nonlinear.** When the activation function is non-linear, then a two-layer neural network can be proven to be a universal function approximator. This is known as the Universal Approximation Theorem. The identity activation function does not satisfy this property. When multiple layers use the identity activation function, the entire network is equivalent to a single-layer model.

- **Range.** When the range of the activation function is finite, gradient-based training methods tend to be more stable because pattern presentations significantly affect only limited weights. When the range is infinite, training is generally more efficient because pattern presentations significantly affect most of the weights. In the latter case, smaller learning rates are typically necessary.

- **Continuously differentiable.** This property is desirable (ReLU is not continuously differentiable and has some issues with gradient-based optimization, but it is still possible) for enabling gradient-based optimization methods. The binary step activation function is not differentiable at 0, and it differentiates to 0 for all other values, so gradient-based methods can make no progress with it.

- **Monotonic.** When the activation function is monotonic, the error surface associated with a single-layer model is guaranteed to be convex.

Remark 3.38. Unfolding ReLU:

- **Why ReLU works?** ReLU has the advantage of bending the linear function at a certain point, to a certain degree. Combined with the biases and weights from the previous layer, the ReLU can take the form of a bend at any location at any degree.

- **When ReLU works?** The strength of the ReLU function lies not in itself, but in an entire army of ReLUs. This is why using a few ReLUs in a neural network does not yield satisfactory results; instead, there must be an abundance of ReLU activations to allow the network to construct an entire map of points. In multi-dimensional space, rectified linear units combine to form complex polyhedra along the class boundaries.
Figure 27: Whereas tanh, a smooth, curved function, draws a clean envelope around the circle (and linear fails completely), ReLU draws a hexagon, with several pointed corners.

3.6 Optimizer for gradient descent

Remark 3.39. In PyTorch practices, the learning rate is fixed during the training process once the optimizer is initialized. The only way to adjust the learning rate during the training is through a scheduler.

Definition 3.54. **Momentum** is a method that speeds up convergence by preserving the influence of the previous update direction on the next iteration to a certain degree.

\[ v_t = \gamma v_{t-1} + \eta \nabla \theta L(\theta), \]
\[ \theta = \theta - v_t, \]

where \( \gamma \) is called the momentum term and is typically set to 0.9.

Remark 3.40. If the current gradient is parallel to the previous speed \( v_{t-1} \), the previous speed can speed up this search. If the current gradient is opposite to the previous speed \( v_{t-1} \), the value of \( v_t \) will have a deceleration effect on this search.

Definition 3.55. **NAG (Nesterov Accelerated Gradient)** is a method similar to the momentum, but calculates the gradient w.r.t. the approximate future position of parameter \( \theta \), rather than the current parameter.

\[ v_t = \gamma v_{t-1} + \eta \nabla \theta (\theta - \gamma v_{t-1}) \]
\[ \theta = \theta - v_t \]

where \( \theta - \gamma v_{t-1} \) is the approximation of the next position of the parameters.
Figure 28: Nesterov momentum. Instead of calculating the gradient at the red circle, NAG looks ahead to the tip of the green arrow, which is slightly different from the original momentum method. This anticipatory update prevents us from going too fast and results in increased responsiveness.

**Definition 3.56. AdaGrad(Adaptive subgradient)** is a method that adjusts the learning rate for every parameter $\theta$ (not a vector by a scalar) at every time step $t$ dynamically based on the historical gradient in previous iterations:

$$
\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\sum_{i=1}^{t} \|g(i)\|^2} + \epsilon} \cdot g^{(t)}
$$

where $g^{(t)}$ is the gradient w.r.t. $\theta$ at step $t$, while $\epsilon$ is a smoothing term that avoids division by zero (usually on the order of $1e^{-8}$).

**Remark 3.41.** Without the square root operation, the algorithm performs much worse.

**Remark 3.42.** As the training time increases, the accumulated gradient will become larger and larger, making the learning rate tend to zero, resulting in ineffective parameter updates.

**Definition 3.57. AdaDelta or RMSprop** is a method based on AdaGrad but solves radically diminishing learning rate by focusing only on the gradients in a window over a period (taking a decaying average of all past squared gradients in practice, the running average $E$):

$$
\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E(\|g\|^2) + \epsilon}} \cdot g^{(t)}
$$

$$
E^{(t)}(\|g\|^2) = \gamma E^{(t-1)}(\|g^{(t)}\|^2) + (1 - \gamma)\|g^{(t)}\|^2
$$

where $g^{(t)}$ is the gradient w.r.t. $\theta$ at step $t$, while $\epsilon$ is a smoothing term that avoids division by zero (usually on the order of $1e^{-8}$).

**Definition 3.58. Adam(Adaptive moment)** is a method that combines adaptive learning rate and momentum methods:

$$
m^{(t)} = \beta_1 m^{(t-1)} + (1 - \beta_1) g^{(t)}, \quad m^{(t)}_{\tilde{}} = \frac{m^{(t)}}{1 - \beta_1^t}
$$

$$
v^{(t)} = \beta_2 v^{(t-1)} + (1 - \beta_2) \|g^{(t)}\|^2, \quad v^{(t)}_{\tilde{}} = \frac{v^{(t)}}{1 - \beta_2^t}
$$

$$
\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v^{(t)}_{\tilde{}}} + \epsilon} m^{(t)}_{\tilde{}}
$$

where $m^{(t)}$ and $v^{(t)}$ are exponentially decaying averages of the past gradient (the mean) and the average of past squared gradients (the uncentered variance) respectively. The default values are 0.9 for $\beta_1$, 0.999 for $\beta_2$ and $10^{-8}$ for $\epsilon$. 

42


4 Practices

4.1 Classification and clustering

**Definition 4.1.** Logistic regression is a statistical model that uses a logistic function to model a binary dependent variable. Given a dataset of \( \{(x^{(i)}, y^{(i)})\} \), where \( x^{(i)} \in \mathbb{R}^n \) is the representation of sample \( i \) and \( y^{(i)} \in \{0, 1\} \) is the label of sample \( i \), we want to build a model s.t.

\[
h_{\theta}(x) = g(\theta^T x), g(z) = \frac{1}{1 + e^{-z}}
\]

to estimate the latent distribution of this group of data. The value of \( h_{\theta}(x) \) stands for the probability of \( y = 1 \) given \( x, \theta \).

**Remark 4.1.** The conditional distribution \( y \mid x \) is a Bernoulli distribution rather than a Gaussian distribution.

**Remark 4.2.** Logistic regression is indeed non-linear in terms of Odds and Probability, however, it is linear in terms of Log Odds.

\[
\begin{align*}
\text{• Probability of } (Y = 1): & \quad p = \frac{1}{1 + \exp(-\theta_0 - \theta_1 x^{(1)} - \theta_2 x^{(2)})} \\
\text{• Odds of } (Y = 1): & \quad \frac{p}{1-p} = \exp(-\theta_0 - \theta_1 x^{(1)} - \theta_2 x^{(2)}) \\
\text{• Log Odds of } (Y = 1): & \quad \log \left(\frac{p}{1-p}\right) = -\theta_0 - \theta_1 x^{(1)} - \theta_2 x^{(2)}
\end{align*}
\]

**Remark 4.3.** Typically, logistic regression adopts cross entropy as the loss function, below is the rationale.

**Proof.** Assuming the estimated probability of category \( c_i \) from our model to be \( q(X = c_i; \theta) \), and denote the frequency of sample belonging to category \( c_i \) by \( p(X = c_i) \). After sampling \( N = \sum p(X = c_i) \) conditionally independent samples, which construct the training set\(^{11}\) the likelihood of the parameters \( \theta \) of the model \( q(X = c_i; \theta) \) on the training set is

\[
P_{\text{model}}(x; \theta) = q(X = c_1)
\]

We want to find parameter \( \theta \) s.t. to maximize the likelihood, then we have (assuming binary classification)

\[
\begin{align*}
\theta_{\text{ML}} &= \arg \max_{\theta} \left( \frac{\prod_i^2 q(X = c_1) \times \cdots \times q(X = c_1) \times q(X = c_2) \times \cdots \times q(X = c_2)}{p(X = c_1) \text{ times} \quad p(X = c_2) \text{ times}} \right) \\
&= \arg \max_{\theta} \left( \prod_i^2 q(X = c_i)p(X = c_i) \right) \\
&= \arg \max_{\theta} \left( \sum_i^2 \log(q(X = c_i)p(X = c_i)) \right) \\
&= \arg \max_{\theta} \left( \sum_i^2 (p(X = c_i) \log(q(X = c_i))) \right) \quad \triangleright \text{Opposite of cross-entropy}
\end{align*}
\]

In the case of one-hot vector distance measurement, the frequency of the right category \( p(\hat{x}) \) would always be one, while the frequencies of the other category are always zero. In other words, cross-entropy only cares about the prediction of the position of the right category. \( H(S) \) and \( H(S_v) \) are the entropy of \( S \) and \( S_v \) w.r.t. final class\(^{12}\).

\(^{11}\)The training set we have is actually derived from real-world sampling.

\(^{12}\)For real example, please refer to this [https://towardsdatascience.com/decision-trees-for-classification-id3-algorithm-explained-89df7b7ce72df1](https://towardsdatascience.com/decision-trees-for-classification-id3-algorithm-explained-89df7b7ce72df1)
Definition 4.2. **Hyperplane** $\mathbf{w}^T \mathbf{x} - b = 0$ of an $n$-dimensional space $V$ is a subspace of dimension $n-1$.

![Hyperplane of an 2D space. Right: distance between two hyperplanes.](image)

Remark 4.4. The distance between the origin and the hyperplane is $\left(\frac{\mathbf{w}}{\|\mathbf{w}\|}\right)^T \mathbf{x} = \frac{\mathbf{w}^T \mathbf{x}}{\|\mathbf{w}\|} = \frac{b}{\|\mathbf{w}\|}$, namely the projection of $\mathbf{x}$ on $\mathbf{w}/\|\mathbf{w}\|$.

Remark 4.5. The distance $|a|$ between $\mathbf{w}^T \mathbf{x} - b = c_1$ and $\mathbf{w}^T \mathbf{x} - b = c_2$ can be derived as below: Assuming $\mathbf{w}$ is a unit vector and $a\mathbf{w}$ is distance vector between $\mathbf{w}^T \mathbf{x} - b = c_1$ and $\mathbf{w}^T \mathbf{x} - b = c_2$, $\mathbf{x}'$ is a point on $\mathbf{w}^T \mathbf{x} - b = c_1$, hence $\mathbf{x}' + a\mathbf{w}$ is located on $\mathbf{w}^T \mathbf{x} - b = c_2$, namely

\[
\begin{align*}
\mathbf{w}^T (\mathbf{x}' + a\mathbf{w}) - b &= c_2 \\
\mathbf{w}^T \mathbf{x}' + a\mathbf{w}^T \mathbf{w} - b &= c_2 \\
c_1 + a\mathbf{w}^T \mathbf{w} &= c_2 \\
a\mathbf{w}^T \mathbf{w} &= c_2 - c_1 \\
a &= \frac{c_2 - c_1}{\|\mathbf{w}\|}.
\end{align*}
\]

Definition 4.3. **Support vector machine (SVM)** tries to find a hyperplane to classify data samples.

![Support vector machine.](image)

Example 28. Suppose we are given a training dataset of $n$ points of the form $\{(\mathbf{x}^{(i)}, y^{(i)})\}$ where the $y^{(i)}$ are either 1 or $-1$, each indicating the class to which the point $\mathbf{x}^{(i)}$ belongs. Each $\mathbf{x}^{(i)}$ is a
\( n \)-dimensional real vector. We want to find the “maximum-margin hyperplane” that divides the group of points \( x^{(i)} \) for which \( y^{(i)} = 1 \) from the group of points for which \( y_i = -1 \), which is defined so that the distance between the hyperplane and the nearest point \( x^{(i)} \) from either group is maximized. Any hyperplane can be written as the set of points \( x \) satisfying \( w^T x - b = 0 \), where \( w \) is the (not necessarily normalized) normal vector to the hyperplane.

**Hard-margin** If the training data is linearly separable, we can select two parallel hyperplanes that separate the two classes of data, so that the distance between them is as large as possible. The region bounded by these two hyperplanes is called the “margin”, and the maximum-margin hyperplane is the hyperplane that lies halfway between them. With a normalized or standardized dataset, these hyperplanes can be described by the equations

\[
\begin{align*}
  w^T x - b &= 1 & \text{anything on or above this boundary is of one class, with label } 1 \\
  w^T x - b &= -1 & \text{anything on or below this boundary is of the other class, with label } -1
\end{align*}
\]

Geometrically, the distance between these two hyperplanes is \( \frac{2}{\|w\|} \), so to maximize the distance between the planes we want to minimize \( \|w\| \). The distance is computed using the distance from a point to a plane equation. We also have to prevent data points from falling into the margin, we add the following constraint: for each

\[
\begin{align*}
  w^T x^{(i)} - b &\geq 1, & \text{if } y^{(i)} = 1, \\
  w^T x^{(i)} - b &\leq -1, & \text{if } y^{(i)} = -1.
\end{align*}
\]

These constraints state that each data point must lie on the correct side of the margin. This can be rewritten as

\[
y^{(i)}(w^T x^{(i)} - b) \geq 1, \quad \forall 1 \leq i \leq n.
\]

We can put this together to get the optimization problem:

\[
\begin{align*}
\text{minimize} & \quad \|w\|^2_2 \\
\text{subject to} & \quad y^{(i)}(w^T x^{(i)} - b) \geq 1, \forall i \in \{1, \ldots, n\}
\end{align*}
\]

The \( w \) and \( b \) that solve this problem determine our classifier. An important consequence of this geometric description is that the max-margin hyperplane is completely determined by those \( x^{(i)} \) that lie nearest to it. These \( x^{(i)} \) are called support vectors.

**Soft-margin** To extend SVM to cases in which the data are not linearly separable, it can be formulated as the following optimization problem:

\[
\begin{align*}
\text{minimize} & \quad \lambda \|w\|^2 + \frac{1}{n} \sum_{i=1}^{n} \max \left(0, 1 - y^{(i)}(w^T x^{(i)} - b)\right) \\
\text{subject to} & \quad y^{(i)}(w^T x^{(i)} - b) \geq 1, \forall i \in \{1, \ldots, n\}
\end{align*}
\]

where the parameter \( \max (0, 1 - y^{(i)}(w^T x^{(i)} - b)) \) is the hinge loss (Def. 3.46), which will remove the wrong side sample out of consideration. \( \lambda > 0 \) determines the trade-off between increasing the margin size.

**Primal-Dual problem** We can write the Lagrangian as follows:

\[
\mathcal{L}(w, b, \lambda) = \frac{1}{2} \|w\|^2 + \sum_i \lambda_i \left(1 - y^{(i)}(w^T x^{(i)} - b)\right)
\]
Remark 4.6. Why are we solving the dual problem instead of the primal one? \[ \text{Primal: minimize } \|w\|^2, \text{ subject to } y(i)(wx(i)-b) \geq 1, \forall i \in \{1, \ldots, n\} \]

\[ \text{Dual: } \max_{\lambda} \min_w L(w, b, \lambda) = \frac{1}{2} \|w\|^2 + \sum_i \lambda_i \left( 1 - y(i)(wx(i) - b) \right) \]

We first take the derivative with respect to \( w \) and set to zero:

\[ \frac{\partial L}{\partial w} = w - \sum_i y(i)x(i) = 0 \]

\[ w = \sum_i \lambda_i y(i)x(i) \]

Substituting \( w \) back into the Lagrangian we get:

\[ L(b, \lambda) = \frac{1}{2} \left( \sum \lambda_i y(i)x(i) \right)^T \left( \sum \lambda_j y(j)x(j) \right) + \sum \lambda_i \left( 1 - y(i) \left( \left( \sum \lambda_j y(j)x(j) \right)^T x(i) - b \right) \right) \]

\[ = \frac{1}{2} \sum_i \sum_j \lambda_i \lambda_j y(i)y(j)x(i)^T x(j) + \sum \lambda_i \left( 1 - \sum_j \lambda_j y(j)x(j)^T x(i) \right) - b \]

\[ = \frac{1}{2} \sum_i \sum_j \lambda_i \lambda_j y(i)y(j)x(i)^T x(j) + \sum \lambda_i - \sum \lambda_j \sum_i \lambda_i y(i)x(j)^T x(i) + \sum \lambda_i y(i)b \]

\[ L(b, \lambda) = -\frac{1}{2} \sum_i \sum_j \lambda_i \lambda_j y(i)y(j)x(i)^T x(j) + \sum \lambda_i + \sum_i \lambda_i y(i)b \]

\[ L(\lambda) = -\frac{1}{2} \sum_i \sum_j \lambda_i \lambda_j y(i)y(j)x(i)^T x(j) + \sum \lambda_i \]

On the other hand, for \( \sum_i \lambda_i y(i)b \neq 0 \) such that \( \theta(\lambda) = -\infty \) (by taking \( b \) to infinity).

**Remark 4.6.** Why are we solving the dual problem instead of the primal one?\[ ^{13} \]

- There are some quadratic programming algorithms that can solve the dual faster than the primal, (especially when high dimensions \( d \gg n \))
- The kernel trick can be used in dual problems but not in the primal one:

  **Without kernel trick:** \( \max_{\lambda} \mathcal{L}(\lambda) = -\frac{1}{2} \sum_i \sum_j \lambda_i \lambda_j y(i)y(j)x(i)^T x(j) + \sum \lambda_i \)

  and \( w = \sum \lambda_i y(i)x(i) \)

  **With kernel trick:** \( \max_{\lambda} \mathcal{L}(\lambda) = -\frac{1}{2} \sum_i \sum_j \lambda_i \lambda_j y(i)y(j)k(x(i), x(j)) + \sum \lambda_i \)

  where \( k(x(i), x(j)) = \langle \phi(x(i)), \phi(x(j)) \rangle \)

  and \( w = \sum \lambda_i y(i)\phi(x(i)), b = w^T x(i) - y(i) \)

\[ ^{13} \text{max dual } = \min \text{ primal} \]

\[ ^{14} \text{https://www.cs.cmu.edu/~aarti/Class/10315_Fall20/lecs/svm_dual_kernel.pdf} \]
Remark 4.7. **Difference between the SVM and LR.** In the linearly separable case, LR returns any solution that separates the two classes, while hard SVM finds the solution among all possible ones that has the maximum margin. In the case of linearly inseparable, LR finds a hyperplane that corresponds to the minimum of some error, while soft SVM tries to minimize another error and at the same time trades off that error with the margin via a regularization parameter. SVM is a hard classifier but LR is a probabilistic one.

Definition 4.4. **Naive Bayes** is an algorithm that assigns a class to a new sample by the votes from \( k \) nearest neighbors.

Definition 4.5. **\( k \)-nearest neighbors algorithm** is an algorithm that assigns a class to a new sample by the votes from \( k \) nearest neighbors.

Definition 4.6. **\( k \)-means clustering** is a method that aims to partition \( n \) observations into \( k \) clusters in which each observation belongs to the cluster with the nearest mean.

Remark 4.8. **Difference between kNN and k-means.**

- \( k \)NN is supervised while \( k \)-means is unsupervised;
- \( k \)NN is a classification or regression model while \( k \)-means is a clustering model;
- \( k \)NN performs much better if all of the data have the same scale, whereas this does not apply to \( k \)-means.

Example 29. Lloyd’s algorithm starts with an initial placement of some number \( k \) of point sites in the input domain. It then repeatedly executes the following relaxation step:

1. Calculate the closest point site for every sample;
2. Recalculate the mean of \( k \) clusters based on current partition;
3. Go to the first step.

Remark 4.9. The algorithm converges slowly or, due to limitations in numerical precision, may not converge. Therefore, real-world applications of Lloyd’s algorithm typically stop once the distribution is “good enough”. One common termination criterion is to stop when the maximum distance moved by any site in an iteration falls below a preset threshold.

4.2 Ensembles

Definition 4.7. **Information gain** calculates the reduction in the entropy and measures how well a given feature separates or classifies the target classes. The feature with the highest Information Gain is selected as the best one.

Example 30. Information gain for a feature column \( F \) is calculated as:

\[
IG(S, F) = H(S) - \sum_v \left( \frac{|S_v|}{|S|} \times H(S_v) \right)
\]

where \( S_v \) is the set of sample (row) in \( S \) for which the feature (column) \( F \) has value \( v \), \( |\cdot| \) stands for the number of set element.
Definition 4.8. [Decision tree] is a decision support tool that uses a tree-like model of decisions and their possible consequences. Decision tree learning is basically an algorithm picking the feature which brings the most information gain before making the decision.

Example 31. ID3 tries to pick the feature with maximum information gain (resulting entropy after splitting is minimized):

1. Calculate the entropy of every attribute $F$ of the dataset $S$.
2. Split the set $S$ into subsets using the attribute of which the information gain is maximum.
3. Make a decision tree node containing that attribute.
4. Recurse on subsets using the remaining attributes.

Example 32. C4.5 builds decision trees from a set of training data in the same way as ID3, using the concept of information entropy. C4.5 has a few base cases:

- All the samples in the list belong to the same class. When this happens, it simply creates a leaf node for the decision tree saying to choose that class.
- None of the features provide any information gain. In this case, C4.5 creates a decision node higher up the tree using the expected value of the class.
- Instance of previously-unseen class encountered. Again, C4.5 creates a decision node higher up the tree using the expected value.

And the workflow is:

1. Check for the above base cases.
2. For each attribute $a$, find the normalized information gain ratio from splitting on $a$.
3. Let $F_{best}$ be the attribute with the highest normalized information gain.
4. Create a decision node that splits on $F_{best}$.
5. Recurse on the sublists obtained by splitting on $F_{best}$, and add those nodes as children of the node.

Remark 4.10. C4.5 made a number of improvements to ID3. Some of these are:

- Handling both continuous and discrete attributes - In order to handle continuous attributes, C4.5 creates a threshold and then splits the list into those whose attribute value is above the threshold and those that are less than or equal to it.
- Handling training data with missing attribute values - C4.5 allows attribute values to be marked as ? for missing. Missing attribute values are simply not used in gain and entropy calculations.
- Handling attributes with differing costs.
- Pruning trees after creation - C4.5 goes back through the tree once it’s been created and attempts to remove branches that do not help by replacing them with leaf nodes.
**Definition 4.9.** [Bootstrap aggregating](bagging), is a machine learning ensemble meta-algorithm (weak learner, e.g. shallow decision tree) designed to improve the stability and accuracy of machine learning algorithms used in statistical classification and regression. It also reduces variance and helps to avoid overfitting.

**Example 33.** Given a standard training set $D$ of size $n$, bagging generates $m$ new training sets $D_i$, each of size $n'$, by sampling from $D$ uniformly and with replacement. By sampling with replacement, some observations may be repeated in each $D_i$. This kind of sample is known as a bootstrap sample. Sampling with replacement ensures each bootstrap is independent from its peers, as it does not depend on previous chosen samples when sampling. Then, $m$ models are learned using the above $m$ bootstrap samples and combined by averaging the output (for regression) or voting (for classification).

![Diagram of bagging process]

**Definition 4.10.** [Random forests] is an extension over bagging. In addition to taking the random subset of data, it also takes the random selection of features\(^{15}\) rather than using all features to grow trees.

**Remark 4.11.** Pros of the random forest:

- Consisting of multiple decision trees, forests are able to more accurately make predictions;
- Works well with non-linear data;
- There is a lower risk of overfitting and runs efficiently on even large data sets, which is the result of the random forest’s use of bagging in conjunction with random feature selection;
- The random forest classifier operates with a high speed because of using a smaller dataset;
- Can be performed in parallel (boosting is sequential).

**Cons of the random forest:**

- Random forests are incredibly dependent on their data sets, changing these can drastically change the individual trees’ structures;
- Requires much more time to train the data compared to decision trees, make decisions, and vote within the random forest classifier.

\(^{15}\)Typically, for a classification problem with $p$ features, $\sqrt{p}$ (rounded down) features are used in each split. For regression problems, the inventors recommend $p/3$ (rounded down) with a minimum node size of 5 as the default.
Remark 4.12. The difference between boosting and random forest comes from error = bias + variance:

- Boosting reduces error mainly by reducing bias (and also to some extent variance, by aggregating the output from many models);
- Random Forest tackles the error reduction task in the opposite way: by reducing variance, which is often existing in a complex system.

Definition 4.11. Gradient boosting gives a prediction model in the form of an ensemble of weak prediction models, which are typically decision trees. When a decision tree is a weak learner, the resulting algorithm is called a gradient-boosted tree; it usually outperforms random forest.

Example 34. Input: training set \( \{(x^{(i)}, y^{(i)})\}_{i=1}^{n} \), a differentiable loss function \( L(y, F(x)) = \frac{1}{2}(y - F(x))^2 \), where \( F \) is the tree model, \( M \) is the number of iterations. Algorithm:

1. Initialize model with a constant value:
   \[
   F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y^{(i)}, \gamma).
   \]

2. For \( m = 1 \) to \( M \), compute so-called pseudo-residuals:
   
   (a) \[
   r_{im} = -\left( \frac{\partial L(y^{(i)}, F(x^{(i)}))}{\partial F(x^{(i)})} \right)_{F(x)=F_{m-1}(x)} = y_i - F(x_i) \quad \text{for} \quad i = 1, \ldots, n.
   \]

   (b) Fit a base learner (making a new tree) closed under scaling \( h_m(x) \) to pseudo-residuals, i.e. train it using the training set \( \{(x^{(i)}, r_{im})\}_{i=1}^{n} \).

   (c) Compute multiplier \( \gamma_m \) by solving the following one-dimensional optimization problem:
   
   \[
   \gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L \left( y^{(i)}, F_{m-1}(x^{(i)}) + \gamma h_m(x^{(i)}) \right).
   \]

   (d) Update the model:
   
   \[
   F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).
   \]

3. Output \( F_M(x) \).

---

\( F(x) \) is the scaling factor, which prevents overfitting.
4.3 Regression

Definition 4.12. **Linear regression** is a linear approach for modeling the relationship between a scalar response and one or more explanatory variables. The case of one explanatory variable is called simple linear regression; for more than one, the process is called multiple linear regression. Given a dataset of \{ (x^{(i)}, y^{(i)}) \}, where \( x^{(i)} \in \mathbb{R}^n \) is the representation of sample \( i \) and \( y^{(i)} \in \mathbb{R} \) is the value of sample \( i \), we want to build a model s.t.
\[
y = X\theta + \varepsilon
\]
where
\[
y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}, \quad X = \begin{bmatrix} x^{(1)}_1 & \cdots & x^{(1)}_p \\ x^{(2)}_1 & \cdots & x^{(2)}_p \\ \vdots & \vdots & \vdots \\ x^{(n)}_1 & \cdots & x^{(n)}_p \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{bmatrix}, \quad \varepsilon = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}.
\]

Definition 4.13. **Newton’s method** is an iterative method for finding the roots of a differentiable function \( f \), which are solutions to the equation \( f(x) = 0 \). If the tangent line to the curve \( f(x) \) at \( x^{(n)} \) (the initial guess) intercepts the \( x \)-axis at \( x^{(n+1)} \) then the slope is
\[
f’(x^{(n)}) = \frac{f(x^{(n)}) - 0}{x^n - x^{(n+1)}}
\]
Solving for \( x^{(n+1)} \) gives
\[
x^{(n+1)} = x^{(n)} - \frac{f(x^{(n)})}{f’(x^{(n)})}
\]

Definition 4.14. **Newton’s method in optimization** is an iterative method for finding the minima of a twice-differentiable function \( f \), which are solutions to the equation \( f’(x) = 0 \). There are two ways to derive the updating rule:

- By leveraging the conclusion in Newton’s method, we can treat \( f’ \) as the function for which we are trying to find roots, hence we have the following updating rule:
\[
x^{(n+1)} = x^{(n)} - \frac{f’(x^{(n)})}{f’’(x^{(n)})}
\]
- By Taylor expansion, we can have \( f(x^{(n)} + t) \) decomposed as following:
\[
f(x^{(n)} + t) \approx f(x^{(n)}) + f’(x^{(n)}) t + \frac{1}{2} f’’(x^{(n)}) t^2.
\]
By taking the derivative of \( f(x^{(n)} + t) \) w.r.t. \( t \), namely find the \( t \) can make \( f(x^{(n)} + t) \) as small as possible, we have
\[
\frac{d}{dt} f(x^{(n)} + t) \approx \frac{d}{dt} \left( f(x^{(n)}) + f’(x^{(n)}) t + \frac{1}{2} f’’(x^{(n)}) t^2 \right)
\]
\[
= f’(x^{(n)}) + f’’(x^{(n)}) t.
\]
Making the derivative equal to 0 to find the stationary point
\[ 0 = f'(x^{(n)}) + f''(x^{(n)}) t \]
\[ t = -\frac{f'(x^{(n)})}{f''(x^{(n)})} \]

Hence we can have the following update rule:
\[ x^{(n+1)} = x^{(n)} + t = x^{(n)} - \frac{f'(x^{(n)})}{f''(x^{(n)})} \]

which is rather similar to the gradient descent but has step size configured as \( \frac{1}{f''(x^{(n)})} \).

**Remark 4.13.** For more than one dimension case, the update rule remains similar but replacing \( f' \) with \( \nabla f \) and \( 1/f'' \) with \( (\nabla^2 f)^{-1} \), namely Hessian matrix inverse \( H_f^{-1} \).

\[ x^{(n+1)} = x^{(n)} - (\nabla^2 f)^{-1} \left( x^{(n)} \right) \nabla f \left( x^{(n)} \right) \]

**Remark 4.14.** We now consider the implications of an indefinite Hessian at a critical point. Suppose that \( f(x) \) has continuous second partial derivatives on a set \( D \subseteq \mathbb{R}^n \). Furthermore, let \( x^* \) be an interior point of \( D \) that is a critical point (could be either a local maximum, a local minimum, or a saddle point) of \( f(x) \). If \( H_f(x^*) \) is indefinite, then there exist vectors \( u, v \) such that
\[ u^T H_f(x^*) u > 0, v^T H_f(x^*) v < 0 \]

By continuity of the second partial derivatives, there exists an \( \epsilon > 0 \) such that
\[ u^T H_f(x^* + tu) u > 0, v^T H_f(x^* + tv) v < 0 \]

for \( |t| < \epsilon \). If we define
\[ U(t) = f(x^* + tu), \]
\[ U'(t) = \frac{\partial f(x^* + tu)}{\partial t} = \frac{\partial f(x^* + tu)}{\partial x^*} \cdot \frac{\partial (x^* + tu)}{\partial t} = \nabla f(x^* + tu)^T u, \]
\[ U''(t) = \frac{\partial^2 f(x^* + tu)}{\partial t^2} = \frac{\partial \nabla f(x^* + tu)}{\partial (x^* + tu)} \cdot \frac{\partial (x^* + tu)}{\partial t} = u^T \nabla^2 f(x^* + tu) u \]
\[ V(t) = f(x^* + tv), \]
\[ V'(t) = \frac{\partial f(x^* + tv)}{\partial t} = \frac{\partial f(x^* + tv)}{\partial x^*} \cdot \frac{\partial (x^* + tv)}{\partial t} = \nabla f(x^* + tv)^T v, \]
\[ V''(t) = \frac{\partial^2 f(x^* + tv)}{\partial t^2} = \frac{\partial \nabla f(x^* + tv)}{\partial (x^* + tv)} \cdot \frac{\partial (x^* + tv)}{\partial t} = v^T \nabla^2 f(x^* + tv) v \]

then we have \( U'(0) = \nabla f(x^*)^T u = 0, V'(0) = \nabla f(x^*)^T v = 0 \) (as \( x^* \) is a critical point of \( f(x) \)), while \( U''(0) = u^T \nabla^2 f(x^* + tu) u > 0 \) and \( V''(0) = v^T \nabla^2 f(x^* + tv) v < 0 \) (by continuity of the second partial derivatives). Therefore, \( t = 0 \) is a strict local minimizer of \( U(t) \) and a strict local maximizer of \( V(t) \), namely \( x^* \) is a saddle point of \( f \).

**Remark 4.15.** Newton’s method, in its original version, has several caveats:

- It does not work if the Hessian is not invertible. This is clear from the very definition of Newton’s method, which requires taking the inverse of the Hessian.

\[ \text{https://www.math.usm.edu/lambers/mat419/lecture3.pdf} \]
• It can converge to a saddle point instead of to a local minimum if the Hessian is not a positive definite matrix.

Figure 31: (a) Quadratic form for a positive-definite matrix. (b) For a negative-definite matrix. (c) For a singular (and positive-indefinite) matrix. A line that runs through the bottom of the valley is the set of solutions. (d) For an indefinite matrix. Because the solution is a saddle point, the steepest descent and conjugate gradient will not work. In three dimensions or higher, a singular matrix can also have a saddle.

**Definition 4.15.** Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm is an iterative method for solving unconstrained nonlinear optimization problems. From an initial guess $x^{(0)}$ and an approximate Hessian matrix $H^{(0)}$ the following steps are repeated as $x^{(k)}$ converges to the solution:

1. Obtain a direction $p^{(k)}$ by solving $p^{(k)} = -H^{(k)^{-1}} \nabla f(x^{(k)})$ (exactly the same as the Newton method).

2. Perform a one-dimensional optimization (line search) to find an acceptable step size $\alpha^{(k)}$ in the direction found in the first step. If an exact line search is performed, then $\alpha^{(k)} = \arg \min f(x^{(k)} + \alpha p^{(k)})$. In practice, an inexact line search usually suffices, with an acceptable $\alpha^{(k)}$ satisfying Wolfe conditions.

3. $s^{(k)} = \alpha^{(k)} p^{(k)}$ and update $x_{k+1} = x^{(k)} + s^{(k)}$.

4. $y^{(k)} = \nabla f(x^{(k+1)}) - \nabla f(x^{(k)})$.

5. $H^{(k+1)} = H^{(k)} + \frac{y^{(k)}y^{(k)^T}}{y^{(k)^T}s^{(k)}} - \frac{H^{(k)}s^{(k)}s^{(k)^T}H^{(k)^T}}{s^{(k)^T}H^{(k)}s^{(k)}}$.

4.4 Autoencoder

**Definition 4.16.** Autoencoder is a type of artificial neural network used to learn efficient data codings in an unsupervised (do not require labeled inputs to enable learning) manner. It is constituted of two main parts: an encoder that maps the input into the code (latent variable), and a decoder that maps the
code (latent variable) to a reconstruction of the input.

\[
\phi : \mathcal{X} \rightarrow \mathcal{F} \\
\psi : \mathcal{F} \rightarrow \mathcal{X}
\]

\[
\phi, \psi = \arg \min_{\phi, \psi} \|x - (\psi \circ \phi)(x)\|^2 \quad \triangleright \text{loss function}
\]

Figure 32: From left to right: 2D original distribution, distribution recovered by PCA, and distribution recovered by autoencoder. Since the original feature space is 2D, namely the covariance matrix of the distribution is \(2 \times 2\), which means the distribution would be 1D after dimension reduction, and this is the reason why the distribution recovered by PCA is totally linear.

Figure 33: From left to right: 3D original distribution, distribution recovered by PCA, and distribution recovered by autoencoder. Since the original feature space is 3D, which means the distribution would be 2D after dimension reduction, and this is the reason why the distribution recovered by PCA looks like a plane.

**Remark 4.16.** Relationship between **PCA** and **autoencoder**

- PCA is essentially a linear transform method while autoencoders are capable of modeling non-linear complex functions.

- Features in PCA are linearly uncorrelated with each other, as they are orthogonal. While the autoencoded features may have correlations since they are just trained for accurate reconstruction.

- PCA is faster and computationally cheap compared to autoencoder.

- Autoencoder is prone to overfitting due to a large amount of parameters.

**Example 35.** PyTorch implementation of autoencoder:

```python
class AutoEncoder(nn.Module):
    def __init__(self):
```

super(AutoEncoder, self).__init__()

self.encoder = nn.Sequential(
    nn.Linear(28*28, 128),
    nn.Tanh(),
    nn.Linear(128, 64),
    nn.Tanh(),
    nn.Linear(64, 12),
    nn.Tanh(),
    nn.Linear(12, 3),  # compress to 3 features which can be visualized in plt
)

self.decoder = nn.Sequential(
    nn.Linear(3, 12),
    nn.Tanh(),
    nn.Linear(12, 64),
    nn.Tanh(),
    nn.Linear(64, 128),
    nn.Tanh(),
    nn.Linear(128, 28*28),
    nn.Sigmoid(),  # compress to a range (0, 1)
)

def forward(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return encoded, decoded

Example 36. Different types of autoencoders:

- **Sparse autoencoder**\(^{21}\) constraints most of the hidden units inactive (close to 0) by adding a regularization term to the loss function

\[
L(x, \hat{x}) + \lambda \sum_i \|a_i^{(h)}\| \quad \triangleright \text{L}_1 \text{ regularization}
\]

\[
L(x, \hat{x}) + \sum_j D_{\text{KL}}(\rho, \rho_j) \quad \triangleright \text{KL divergence regularization}
\]

where \(a_i^{(h)}\) is the activation output of \(h\)-th layer's hidden unit \(i\), \(\rho\) is the sparsity parameter which is typically preset as a small value close to 0, and \(\rho_j\) denotes the average activation of hidden unit \(j\) (averaging over all training data).

- **Denoising autoencoder**'s input is the noised original data, and its output should be the noise-free original data.

\(^{21}\)https://web.stanford.edu/class/cs294a/sparseAutoencoder.pdf

Definition 4.17. **Variational autoencoder**\(^{8}\) is the artificial neural network architecture that is meant to compress the input information into a constrained multivariate latent distribution (encoding) to reconstruct it as accurately as possible (decoding).
Example 37. The workflow of variational autoencoder:

1. We feed a real data sample $x^{(i)}$ into the encoder, the output of which is the parameters $\log \sigma_i^2, \mu_i$ of $q_\phi(z|x)$;

2. We sample a $z^{(i)}$ from the Gaussian distribution $q_\phi(z|x) \sim \mathcal{N}(\mu, \sigma)$;

3. We feed the sampled $z^{(i)}$ into the decoder, the output of which is the parameters $\sigma'_i, \mu'_i$ of $p_\theta(x|z)$;

4. We sample a $x'$ from the Gaussian distribution $p_\theta(x'|z) \sim \mathcal{N}(\mu', \sigma')$.

![Variational autoencoder diagram]

Figure 34: Variational autoencoder

Example 38. Pseudocode\textsuperscript{23} of VAE:

```python
class VAE(nn.Module):
    def __init__(self, x_dim, h_dim1, h_dim2, z_dim):
        super(VAE, self).__init__()
        # encoder part
        self.fc1 = nn.Linear(x_dim, h_dim1)
        self.fc2 = nn.Linear(h_dim1, h_dim2)
        self.fc31 = nn.Linear(h_dim2, z_dim)
        self.fc32 = nn.Linear(h_dim2, z_dim)
        # decoder part
        self.fc4 = nn.Linear(z_dim, h_dim2)
        self.fc5 = nn.Linear(h_dim2, h_dim1)
        self.fc6 = nn.Linear(h_dim1, x_dim)

    def sampling(self, mu, log_var):
        std = torch.exp(0.5*log_var)
        eps = torch.randn_like(std)
        return eps.mul(std).add_(mu) # return z sample

    def forward(self, x):
        # In practice, we do not sample $z^{(i)}$ from $q_\phi(z|x)$, as this would fail the backpropagation. Instead, we derive it by $z^{(i)} = \mu_i + \sigma_i \odot \epsilon_i$, where $\epsilon_i$ is sampled from $\mathcal{N}(0,1)$ and $\odot$ stands for element-wise multiplication, which guarantees $z^{(i)}$ follow the normal distribution. This is so-called reparameterization trick.
```

\textsuperscript{23}https://github.com/lyeoni/pytorch-mnist-GAN/blob/master/pytorch-mnist-VAE.ipynb

56
# encoder part
h = F.relu(self.fc1(x))
h = F.relu(self.fc2(h))
mu, log_var = self.fc31(h), self.fc32(h)

# sampling
z = self.sampling(mu, log_var)

# decoder part
h = F.relu(self.fc4(z))
h = F.relu(self.fc5(h))
reconstructed_x = F.sigmoid(self.fc6(h))
return reconstructed_x, mu, log_var

# build model
vae = VAE(x_dim=784, h_dim1=512, h_dim2=256, z_dim=2)
for epoch in range(n_epochs):
    for x_real in enumerate(train_loader):
        reconstructed_x, mu, log_var = vae(x_real)
        BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), reduction='sum')
        KLD = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())
        loss = BCE + KLD
        loss.backward()

Remark 4.17. The ultimate goal of VAE is to find the potential probabilistic model behind the $p_θ(x)$, the distribution of real-world samples $x$. Once we know how the real-world samples are distributed, sampling a high-likelihood sample (in other words, a sample looks real) cannot be difficult. But how do we know we have found a good probabilistic model? By maximum likelihood estimation! $\sum \log p_θ(x_i)$ should be as large as possible if we send real-world samples into the model. Once we have the model achieves a high score on real-world samples, we can confidently say that it’s a good measure of truthfulness and that the new sampling on this model is reliable. Below is the deviation of the loss function of VAE:

$$
\log p_θ(x) = 1 \cdot \log p_θ(x) \\
= \int q_φ(z|x)dz \cdot \log p_θ(x) \\
= \int q_φ(z|x) \log \frac{p_θ(x,z)}{p_θ(z|x)} dz \\
= \int q_φ(z|x) \log \left( \frac{p_θ(x,z)}{q_φ(z|x)} \frac{q_φ(z|x)}{p_θ(z|x)} \right) dz \\
= \underbrace{\int q_φ(z|x) \log \frac{p_θ(x,z)}{q_φ(z|x)} dz}_{\text{ELBO}} + \int q_φ(z|x) \log \frac{q_φ(z|x)}{p_θ(z|x)} dz \\
\leq D_{KL}(q_φ(z|x)||p_θ(z|x))$$

We want to maximize the $\log p_θ(x)$ but also minimize $D_{KL}(q_φ(z|x)||p_θ(z|x))$, namely making the two
distribution $q_\phi(z|x)$ and $p_\theta(z|x)$ as close as possible, which is equivalent to maximize ELBO:

$$\log p_\theta(x) = \frac{\int q_\phi(z|x) \log \frac{p_\theta(x,z)}{q_\phi(z|x)} dz + D_{KL}(q_\phi(z|x)||p_\theta(z|x))}{\text{ELBO}}$$

$$\log p_\theta(x) \uparrow - D_{KL}(q_\phi(z|x)||p_\theta(z|x)) = \int q_\phi(z|x) \log \frac{p_\theta(x,z)}{q_\phi(z|x)} dz, \uparrow$$

where

$$\text{ELBO} = \int q_\phi(z|x) \log \frac{p_\theta(x,z)}{q_\phi(z|x)} dz$$

$$= \int q_\phi(z|x) \log \frac{p_\theta(x,z) \cdot p_\theta(z)}{q_\phi(z|x)} dz$$

$$= \int q_\phi(z|x) \log \frac{p_\theta(x,z)}{q_\phi(z|x)} dz + \int q_\phi(z|x) \log \frac{p_\theta(z)}{q_\phi(z|x)} dz$$

$$= E_{q_\phi}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x)||p_\theta(z)) \quad \triangleright D_{KL}(q_\phi(z|x)||p_\theta(z)) = \int q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(z)} dz$$

$$= E_{q_\phi}[\log p_\theta(x|z)] + \frac{1}{2} \sum_{i=1}^{d} (1 - \mu_i^2 - \sigma_i^2 + \log(\sigma_i^2)) \quad \triangleright \text{See deviation below}$$

$$\Rightarrow - \sum_{i=1}^{d} (x_i - \mu_i)^2 + \frac{1}{2} \sum_{i=1}^{d} (1 - \mu_i^2 - \sigma_i^2 + \log(\sigma_i^2)) \quad \triangleright \text{See deviation below}$$

Since $p_\theta(x|z) \sim N(\mu', \sigma'^2)$, the maximization of $= E_{q_\phi}[\log p_\theta(x|z)](\text{MLE})$ is equivalent to minimizing mean squared error (MSE), see Remark 3.21. Therefore, the loss function of VAE can be written as

$$L = \sum_{i=1}^{d} (x_i - \mu_i')^2 - \frac{1}{2} \sum_{i=1}^{d} (1 - \mu_i'^2 - \sigma_i'^2 + \log(\sigma_i'^2)),$$

(1)

where $d$ is the dimension of the random variable and assuming that $q_\phi(z|x) \sim N(\mu, \sigma^2)$, $p_\theta(x|z) \sim N(\mu', \sigma'^2)$ and $p_\theta(z) \sim N(0, 1)$. In other words, the ideal model of VAE should output a $x'$ as close as $x$, whereas the absolute value of mean and variance of $q_\phi(z|x)$ should be as small as possible.

$$- D_{KL}(q_\phi(z|x)||p(z)) = - \int q_\phi(z|x) \cdot \log \frac{q_\phi(z|x)}{p(z)} dz$$

$$= - \int q_\phi(z|x) \cdot \log q_\phi(z|x) dz + \int q_\phi(z|x) \cdot \log p(z) dz$$

$$= \frac{1}{2} + \frac{1}{2} \log 2\pi\sigma^2 - \frac{1}{2} (\mu^2 + \sigma^2) - \frac{1}{2} \log 2\pi$$

$$= \frac{1}{2} (1 - \mu^2 - \sigma^2 + \log(\sigma^2)) \quad \triangleright \text{See deviation below}$$
For one-dimensional distribution, recalling $q_\phi(z|x) \sim \mathcal{N}(\mu, \sigma^2)$ and $p_\theta(z) \sim \mathcal{N}(0, 1)$, we have

$$
\int q_\phi(z|x) \cdot \log q_\phi(z|x) \, dz = \int q_\phi(z|x) \cdot \log \left( \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp \left( -\frac{(z-\mu)^2}{2\sigma^2} \right) \right) \, dz \\
= \int q_\phi(z|x) \cdot \left( -\frac{(z-\mu)^2}{2\sigma^2} \right) \, dz - \int q_\phi(z|x) \log \sqrt{2\pi} \sigma^2 \, dz \\
= -\frac{1}{2\sigma^2} \int q_\phi(z|x) \cdot (z-\mu)^2 \, dz - \log \sqrt{2\pi} \sigma^2 \int q_\phi(z|x) \, dz \\
= -\frac{1}{2} \left( \int q_\phi(z|x) \cdot (z-\mu)^2 \, dz + \int q_\phi(z|x) \cdot 2\mu \, dz - \int q_\phi(z|x) \cdot \mu^2 \, dz \right) - \log \sqrt{2\pi} \int q_\phi(z|x) \, dz \\
= -\frac{1}{2} \left( \sigma^2 + 2\mu \int q_\phi(z|x) \cdot z \, dz - \mu^2 \int q_\phi(z|x) \, dz \right) - \log \sqrt{2\pi} \int q_\phi(z|x) \, dz \\
= -\frac{1}{2} \left( \sigma^2 + 2\mu \cdot \mu - \mu^2 \cdot 1 \right) - \log \sqrt{2\pi} \int q_\phi(z|x) \, dz \\
= -\frac{1}{2} (\mu^2 + \sigma^2) - \frac{1}{2} \log 2\pi
$$

Remark 4.18. Relationship between **autoencoder** and **variational autoencoder**:

- **Autoencoders** try to map a sample to a deterministic latent representation. It is a way to compress the data.

- **Variational autoencoders** try to map a sample to a distribution, which is represented by a mean vector and a variance vector, rather than a deterministic representation. It is a probabilistic model to generate new samples that follow a certain distribution.

4.5 Transformer

**Definition 4.18.** Self-attention function is formulated as

$$
\text{multiHead}(Q, K, V) = \text{concat} (\text{head}_1, ..., \text{head}_h) W^O \\
\text{where } \text{head}_h = \text{attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d}} \right) V,
$$

where $d$ is the dimension of embedding vectors.
Remark 4.19. Multi-head attention is concatenated by a single-head attention tensor along the feature vector dimensions.

Example 39. In practice, $Q, K, V \in \mathbb{R}^{n \times d}$ are identical as a concatenated sequence of embedding vectors together, where the dimension of the embedding vector is $d$ and the number of vectors is $n$. Consequently, $\text{softmax}(QK^T/\sqrt{d}) \in \mathbb{R}^{n \times n}$ represents the attention score inside of the sequence.

Example 40. Below shows the structure of a vanilla transformer encoder layer:

```
TransformerEncoderLayer(
    (self_attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=512, out_features=512, bias=True)
    )
    (linear1): Linear(in_features=512, out_features=2048, bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
    (linear2): Linear(in_features=2048, out_features=512, bias=True)
    (norm1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
    (norm2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
    (dropout1): Dropout(p=0.1, inplace=False)
    (dropout2): Dropout(p=0.1, inplace=False)
)
```
4.6 Contrastive Learning

Definition 4.19. SimCLR\cite{balntas2021simclr} is a simple framework for contrastive learning of visual representations. The loss function for a positive pair of examples \((i, j)\) is defined as

\[
\text{loss} = -\log \frac{\exp(\text{sim}(z^{(i)}, z^{(j)})/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{k \neq i} \exp(\text{sim}(z^{(i)}, z^{(k)})/\tau)}
\]

where \(\text{sim}(z^{(i)}, z^{(j)}) = \frac{\langle z^{(i)}, z^{(j)} \rangle}{\|z^{(i)}\| \cdot \|z^{(j)}\|}\), \(\mathbb{1}_{k \neq i} \in \{0, 1\}\) is an indicator function evaluating to 1 iff \(k \neq i\) and \(\tau\) denotes a temperature parameter.

Figure 37: Two separate data augmentation operators are sampled from the same family of augmentations \((t \sim T\) and \(t' \sim T\)\) and applied to each data example to obtain two correlated views. A base encoder network \(f(\cdot)\) and a projection head \(g(\cdot)\) are trained to maximize agreement using a contrastive loss. After training is completed, we throw away the projection head \(g(\cdot)\) and use encoder \(f(\cdot)\) and representation \(h\) for downstream tasks.

Definition 4.20. CLIP (Contrastive Language–Image Pre-training)\cite{radford2021clip} is a pre-training framework to generate a representation for both image and text.

Figure 38: Summary of our approach. While standard image models jointly train an image feature extractor and a linear classifier to predict some label, CLIP jointly trains an image encoder and a text encoder to predict the correct pairings of a batch of (image, text) training examples. At test time the learned text encoder synthesizes a zero-shot linear classifier by embedding the names or descriptions of the target dataset’s classes.

Example 41. Example snippet of the usage of CLIP:
image, class_id = cifar100[3637]
image_input = preprocess(image).unsqueeze(0).to(device)
text_inputs = torch.cat([clip.tokenize(f"a photo of a {c}")) for c in cifar100.classes])

# image_features of shape [1, 512] ([sample #, feature #])
image_features = model.encode_image(image_input)

# text_features of shape [100, 512] ([sample #, feature #])
text_features = model.encode_text(text_inputs)

# Normalization within the representation
image_features /= image_features.norm(dim=-1, keepdim=True)
text_features /= text_features.norm(dim=-1, keepdim=True)

# cosine similarity calculation: 1x512x512x100 -> 1x100
similarity = (100.0 * image_features @ text_features.T).softmax(dim=-1)

4.7 Generative Adversarial Networks

Definition 4.21. Generative Adversarial Networks \[4\] generative part generates candidates while the discriminative part evaluates them.

Example 42. The workflow of GAN:

1. Optimize discriminator \(D\) with generator \(G\) initialized and fixed:
   
   (a) Sample \(x^{(1)}, \ldots, x^{(n)}\) from \(p_{data}\), which are positive samples;
   
   (b) Sample \(z^{(1)}, \ldots, z^{(n)}\) from \(p_{prior}\), which give negative samples;
   
   (c) Generate \(x^{(1)}', \ldots, x^{(n)}'\) from \(G(z) = x^{(i)}'\);
   
   (d) Maximize \(V(G, D) = \frac{1}{n} \sum_{i=1}^{n} \log D(x^{(i)}) + \frac{1}{n} \sum_{i=1}^{n} \log(1 - D(G(z^{(i)})))\) by gradient descent \(\theta_D \leftarrow \theta_D - \eta \nabla V(D)\) for a few iterations.

2. Optimize generator \(G\) with discriminator \(D\) fixed:
   
   (a) With positive samples \(\{x^{(i)}\}\) given, sample \(z^{(1)}, \ldots, z^{(n)}\) from \(p_{prior}\), which give negative samples;
   
   (b) Minimize \(V(G, D) = \frac{1}{n} \sum_{i=1}^{n} \log D(x^{(i)}) + \frac{1}{n} \sum_{i=1}^{n} \log(1 - D(G(z^{(i)})))\) (equivalent to minimizing \(\frac{1}{n} \sum_{i=1}^{n} \log(1 - D(G(z^{(i)})))\)) by gradient descent \(\theta_G \leftarrow \theta_G - \eta \nabla V(G)\) for a few iterations.

3. Iterate from the beginning.
Example 43. Pseudocode\textsuperscript{24} of GAN:

```python
# Generator class Generator(nn.Module):
    def __init__(self, g_input_dim, g_output_dim):
        super(Generator, self).__init__()
        self.fc1 = nn.Linear(g_input_dim, 256)
        self.fc2 = nn.Linear(self.fc1.out_features, self.fc1.out_features*2)
        self.fc3 = nn.Linear(self.fc2.out_features, self.fc2.out_features*2)
        self.fc4 = nn.Linear(self.fc3.out_features, g_output_dim)

    # forward method
    def forward(self, x):
        x = F.leaky_relu(self.fc1(x), 0.2)
        x = F.leaky_relu(self.fc2(x), 0.2)
        x = F.leaky_relu(self.fc3(x), 0.2)
        return torch.tanh(self.fc4(x))

# Discriminator class Discriminator(nn.Module):
    def __init__(self, d_input_dim):
        super(Discriminator, self).__init__()
        self.fc1 = nn.Linear(d_input_dim, 1024)
        self.fc2 = nn.Linear(self.fc1.out_features, self.fc1.out_features//2)
        self.fc3 = nn.Linear(self.fc2.out_features, self.fc2.out_features//2)
        self.fc4 = nn.Linear(self.fc3.out_features, 1)

    # forward method
    def forward(self, x):
        x = F.leaky_relu(self.fc1(x), 0.2)
        x = F.dropout(x, 0.3)
        x = F.leaky_relu(self.fc2(x), 0.2)
        x = F.dropout(x, 0.3)
        x = F.leaky_relu(self.fc3(x), 0.2)
        x = F.dropout(x, 0.3)
        return torch.sigmoid(self.fc4(x))
```

\textsuperscript{24}https://github.com/lyeoni/pytorch-mnist-GAN/blob/master/pytorch-mnist-GAN.ipynb
def discriminator_train(x_real):
    y_real_pred, y_real = Discriminator(x_real), torch.ones()
    discriminator_real_loss = criterion(y_real_pred, y_real)

    x_fake = Generator(z)
    y_fake_pred, y_fake = Discriminator(x_fake), torch.zeros()
    discriminator_fake_loss = criterion(y_fake_pred, y_fake)

    discriminator_loss = discriminator_real_loss + discriminator_fake_loss
    discriminator_loss.backward()

def generator_train():
    x_fake = Generator(z)
    y_fake_pred, y_real = Discriminator(x_fake), torch.ones()
    generator_fake_loss = criterion(y_fake_pred, y_real)
    generator_fake_loss.backward()

for epoch in range(n_epochs):
    for x_real in enumerator(train_loader):
        discriminator_train(x_real)
        generator_train()

Remark 4.20. The workflow of GAN above can be interpreted as:

1. Find the JS divergence between the \( p_{\text{data}} \) and current \( p_G \);

2. Minimize the JS divergence (loss function: \( \frac{1}{n} \sum_{i=1}^{n} \log(1 - D(G(z^{(i)}))) \)), between the \( p_{\text{data}} \) and current \( p_G \) by updating \( G \). Minimizing \( \frac{1}{n} \sum_{i=1}^{n} \log(1 - D(G(z^{(i)}))) \) entails that we want the discriminator will classify our generated samples as true.

Remark 4.21. • The input and output of \( G \) are vector \( z \) (noise) and vector \( x \) (negative sample);

• The input of \( D \) is vector \( x_n \) (negative sample) and vector \( x_p \) (positive sample), while the output is simply a scalar between 0 and 1.

Remark 4.22. Both maximizing \( \frac{1}{n} \sum_{i=1}^{n} \log D(x^{(i)}) + \frac{1}{n} \sum_{i=1}^{n} \log(1-D(G(z^{(i)}))) \) and minimizing \( \frac{1}{n} \sum_{i=1}^{n} \log(1-D(G(z^{(i)}))) \) are intuitive to understand in this adversarial framework. We are not involving any probabilistic theory in the implementation, but however, this simple equation reveals the latent relationship between real data and generated data in the probabilistic space.

Our aim is to find a probabilistic model \( p \) s.t. maximize \( \prod_{i=1}^{n} p_{\text{data}}(x) \), where \( x \) is the real-world sample. However, the true \( p_{\text{data}} \) may be very hard to express explicitly. We alternatively use a neural
network $G$ and its corresponding $p_G$ to approximate the ground truth $p_{data}$.

$$
\theta = \arg \max_{\theta} \prod_{i=1}^{n} p_G(x^{(i)}; \theta)
= \arg \max_{\theta} \sum_{i=1}^{n} \log(p_G(x^{(i)}; \theta))
\approx \arg \max_{\theta} E_{p_{data}}[\log(p_G(x^{(i)}; \theta))]
= \arg \max_{\theta} \int p_{data}(x) \log(p_G(x^{(i)}; \theta)) \, dx
= \arg \max_{\theta} \left( \int p_{data}(x) \log(p_G(x; \theta)) \, dx - \int p_{data}(x) \log(p_{data}(x)) \, dx \right)
= \arg \min_{\theta} D_{KL}(p_{data}(x) \| p_G(x; \theta))
$$

**Remark 4.23.** The objective optimization equation reads as:

$$
\min_{G} \max_{D} V(G, D) = \min_{G} \max_{D} (E_{x \sim p_{data}(x)}[\log D(x)] + E_{z \sim p_{z(x)}}[\log(1 - D(G(z)))]
$$

This objective equation is exactly minimizing the distance of distribution $p_{data}$ and $p_G$.

**Proof.** First, we have

$$
V(G, D) = E_{x \sim p_{data}}[\log D(x)] + E_{z \sim p_{prior}}[\log(1 - D(G(z)))]
= E_{x \sim p_{data}}[\log D(x)] + E_{x \sim p_G(x)}[\log(1 - D(x))]
= \int p_{data}(x) \log D(x) \, dx + \int p_G(x) \log(1 - D(x)) \, dx
= \int [p_{data}(x) \log D(x) + p_G(x) \log(1 - D(x))] \, dx
$$

Assuming for now $x$ is fixed, we need to find a $D(x)$ s.t. maximize $V(G, D)$

$$
f(D) = p_{data}(x) \cdot \log(D) + p_G(x) \cdot \log(1 - D)
\frac{\partial f(D)}{\partial D} = \frac{p_{data}(x)}{D} + \frac{p_G(x)}{1 - D} \times (-1) = 0
D^* = \frac{p_{data}}{p_{data} + p_G}
$$

Therefore, $\max_{D} V(G, D)$ can be written as

$$
\max_{D} V(G, D) = V(G, D^*) = \int p_{data}(x) \cdot \log \left( \frac{p_{data}}{p_{data} + p_G} \right) + p_G(x) \cdot \log \left( \frac{p_G}{p_{data} + p_G} \right) \, dx
= \int p_{data}(x) \cdot \log \left( \frac{p_{data}}{(p_{data} + p_G)^{1/2}} \right) \, dx - \log 2 + \int p_G(x) \cdot \log \left( \frac{p_G}{(p_{data} + p_G)^{1/2}} \right) \, dx - \log 2
= D_{KL}(p_{data} \| p_{data} + p_G) + D_{KL}(p_G \| p_{data} + p_G) - 2 \log 2
= D_{JS}(p_{data} \| p_G) - 2 \log 2
$$

$$
\min_{D} \max_{G} V(G, D) \Leftrightarrow \min_{G} V(G, D^*) \Leftrightarrow \min_{D} D_{JS}(p_{data} \| p_G)
$$

**Remark 4.24.** An individual element of $z$ in a vanilla GAN does not necessarily correspond to semantic features of data, like age and gender etc.
4.8 Deep Learning for Computer Vision

Overview.

- Image Classification: Predict the type or class of an object in an image.
- Object Localization: Locate the presence of objects in an image and indicate their location with a bounding box.
- Object Detection: Locate the presence of objects with a bounding box and types or classes of the located objects in an image.

![Diagram of object recognition computer vision tasks](image)

Figure 40: Overview of object recognition computer vision tasks

4.8.1 Datasets

**Definition 4.22.** [CIFAR-10](https://www.cs.toronto.edu/~kriz/cifar.html) (Canadian Institute For Advanced Research) is a collection of images that are commonly used to train machine learning and computer vision algorithms. The CIFAR-10 dataset contains 60,000 $32 \times 32$ color images in 10 different classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 6,000 images of each class.

**Definition 4.23.** [ImageNet](http://image-net.org) project is a large visual database designed for use in visual object recognition software research. More than 14 million images have been hand-annotated by the project to indicate what objects are pictured and in at least one million of the images, bounding boxes are also provided. ImageNet contains more than 20,000 categories with a typical category.

4.8.2 LeNet

At a high level, LeNet (LeNet-5) consists of two parts:

1. a convolutional encoder consisting of two convolutional layers;
2. a dense block consisting of three fully-connected layers.

The architecture is summarized in the Figure below:
Figure 41: Data flow in LeNet. The input is a handwritten digit, the output a probability over 10 possible outcomes. Courtesy of [21], section 6.6.

**Remark 4.25.** The basic units in each convolutional block are a convolutional layer, a sigmoid activation function, and a subsequent average pooling operation. Note that while ReLUs and max-pooling work better, these discoveries had not yet been made in the 1990s.

A vanilla LeNet can be constructed by the following snippet:

```python
import torch
from torch import nn
from d2l import torch as d2l
net = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Flatten(),
    nn.Linear(16 * 5 * 5, 120), nn.Sigmoid(),
    nn.Linear(120, 84), nn.Sigmoid(),
    nn.Linear(84, 10))
```

### 4.8.3 AlexNet

The design philosophies of AlexNet[9] and LeNet are very similar, but there are also significant differences:

1. AlexNet is much deeper than the comparatively small LeNet-5, with more convolutional layers and a larger parameter space to fit the large-scale ImageNet dataset.

2. AlexNet used the ReLU instead of the sigmoid as its activation function.
Remark 4.26. Dropout, ReLU, and data augmentation were the other key steps of AlexNet in achieving excellent performance in computer vision tasks.

A vanilla AlexNet can be constructed by the following snippet:

```python
import torch
from torch import nn
from d2l import torch as d2l

net = nn.Sequential(
    nn.Conv2d(1, 96, kernel_size=11, stride=4, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nn.Conv2d(96, 256, kernel_size=5, padding=2), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nn.Conv2d(256, 384, kernel_size=3, padding=1), nn.ReLU(),
    nn.Conv2d(384, 384, kernel_size=3, padding=1), nn.ReLU(),
    nn.Conv2d(384, 256, kernel_size=3, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nn.Flatten(),
    nn.Linear(6400, 4096), nn.ReLU(),
    nn.Dropout(p=0.5),
    nn.Linear(4096, 4096), nn.ReLU(),
    nn.Dropout(p=0.5),
    nn.Linear(4096, 10)
)
```

4.8.4 VGG

While AlexNet offered empirical evidence that deep CNNs can achieve good results, it did not provide a general template to guide subsequent researchers in designing new networks. VGG\cite{15}, for the first time, proposed the idea of using blocks to build deep learning framework.
Remark 4.27. In their VGG paper, Simonyan and Zisserman found that several layers of deep and narrow convolutions were more effective than fewer layers of wider convolutions.

A vanilla VGG can be constructed by the following snippet:

```python
import torch
from torch import nn
from d2l import torch as d2l

conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))

def vgg_block(num_convs, in_channels, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
        in_channels = out_channels
    layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
    return nn.Sequential(*layers)

def vgg(conv_arch):
    conv_blks = []
    in_channels = 1
    for (num_convs, out_channels) in conv_arch:
        conv_blks.append(vgg_block(num_convs, in_channels, out_channels))
        in_channels = out_channels
    return nn.Sequential(*conv_blks)
```

Figure 43: From AlexNet to VGG that is designed from building blocks. Courtesy of [21], section 7.2.
*conv_blks, nn.Flatten(),
# The fully-connected part
nn.Linear(out_channels * 7 * 7, 4096), nn.ReLU(), nn.Dropout(0.5),
nn.Linear(4096, 4096), nn.ReLU(), nn.Dropout(0.5),
nn.Linear(4096, 10))

net = vgg(conv_arch)

**Remark 4.28.** Evolution path of LeNet, AlexNet, and VGG:

\[
\text{LeNet} \xrightarrow{\text{MaxPooling, ReLU}} \text{Dropout, Data Augmentation} \xrightarrow{\text{Block Reuse}} \text{AlexNet} \xrightarrow{\text{VGG}}
\]

### 4.8.5 DenseNet

![Diagram of DenseNet and Residual Block](image)

Figure 44: Comparison between Residual block and dense block, parenthesis reads as (input channel, output channel)

### 4.8.6 R-CNN series

**Example 44.** The workflow of R-CNN:

1. Region Proposal: Generate and extract around 2000 candidate bounding boxes by selective search and wrap them into squares;

2. Feature Extraction: Extract feature from each candidate region by a deep convolutional neural network. For each candidate region, the CNN would output a vector \( v \in \mathbb{R}^{4096} \) to present it;

3. Classification: Classify features as one of the known classes by linear SVM classifier model.
Remark 4.29. A downside of R-CNN is that it is slow, requiring a CNN-based feature extraction pass on each of the candidate regions generated by the region proposal algorithm.

Definition 4.24. **Selective Search** is a region proposal algorithm used in object detection.

Example 45. Selective Search takes these over-segments as initial input and performs the following steps:

1. Add all bounding boxes corresponding to segmented parts to the list of regional proposals;
2. Group adjacent segments based on similarity;
3. Go to step1.

Example 46. The workflow of **Fast R-CNN**:

1. Feature Extraction: Extract feature map of a whole image by a deep convolutional neural network;
2. Region Proposal: Identify the region of proposals from feature maps and warp them into squares by using an ROI pooling layer. We reshape them into a fixed size so that they can be fed into a fully connected layer;
3. Classification: Using the ROI feature vector, we use a softmax layer to predict the class of the proposed region and also the offset values for the bounding box.
Remark 4.30. The reason “Fast R-CNN” is faster than R-CNN is that you don’t have to feed 2000 region proposals to the convolutional neural network every time. Instead, the convolution operation is done only once per image and a feature map is generated from it. Region proposals become bottlenecks in Fast R-CNN algorithm affecting its performance.

Example 47. The testing workflow of Faster R-CNN:

1. Feature Extraction: Extract feature map of a whole image by a deep convolutional neural network;

2. Region Proposal: Using region proposal network (RPN) to generate a region of proposals and corresponding class score from feature maps and warp them into squares by using an ROI pooling layer. We reshape them into a fixed size so that they can be fed into a fully connected layer;

3. Classification: After ROI pooling, we can have an ROI feature vector of the same size, then we use a softmax layer to predict the class of the proposed region and also the offset values for the bounding box.

Definition 4.25. Region proposal network (RPN) is a region proposal generator used in object detection.

---

25ROI pooling divides the region proposal into $pooled_h \times pooled_w$ grids, among each grid, it will perform the max pooling. Consequently, we can have the region proposal of the same size without distorting the original image which can compromise the shape information.
Example 48. The workflow of RPN:

1. Feed the 256D feature map generated from VGG into PRN;
2. The feature vector is then fed into two sibling fully connected layers - a box-regression layer (reg) and a box-classification layer (cls);
3. RPN generates \( k \) anchor boxes for each pixel in the feature map, for each anchor box, there are 2 scores and 4 coordinates \( x, y, h, w \). And only 128 positive anchors and 128 negative anchors would be selected for the following training.

![Diagram of RPN workflow](image)

Figure 49: RPN: for each anchor box, it features negative and positive scores (after softmax), and four coordinates of \( x, y, h, w \) (after regression). For a convolutional feature map of a size \( W \times H \) (typically \( \sim 2,400 \)), there are \( WHk \) anchors in total. \[14\]

4.8.7 YOLO series

4.9 Deep Learning for Science

Overview. When we are talking about a deep neural network, the first thing that comes to my mind is that the input/output of the network can be an image. However, in this chapter, we will learn that the neural network can also be a “image”, in view of the function/mapping nature of a network, whose input and output are the coordinates and pixel value, respectively.

Definition 4.26. **Ordinary differential equation** is a differential equation containing one or more functions of one independent variable and the derivatives of those functions.

\[
a_0(x)y + a_1(x)y' + a_2(x)y'' + \cdots + a_n(x)y^{(n)} + b(x) = 0
\]

where \( a_0(x), \ldots, a_n(x) \) and \( b(x) \) are arbitrary differentiable functions that do not need to be linear, and \( y', \ldots, y^{(n)} \) are the successive derivatives of the unknown function \( y \) of the variable \( x \).

Example 49. Integrating a curve from a vector field is actually solving the very naive ODE: \( \frac{dy}{dt} = v(y,t) \), typically we solve this by Euler method. Solving ODE is not always as simple as integrating.

Definition 4.27. **Partial differential equation** is an equation that imposes relations between the various partial derivatives of a multivariable function. For example:

\[
A u_{xx} + 2Bu_{xy} + Cu_{yy} + \cdots \text{(lower order terms)} = 0,
\]

where the coefficients \( A, B, C \cdots \) may depend upon \( x \) and \( y \).
Remark 4.31. Relationship between ODE and PDE:

- The differential operation in ODE is only with respect to one variable (not necessarily means the function only has one variable).
- The differential operation in PDE would be related to multiple variables.

Example 50. When the equation involves a gradient $\nabla$ or Laplacian $\Delta$, it is a partial differential equation.

4.9.1 PINN

In one space dimension, Burger's equation along with Dirichlet boundary conditions reads as

$$u_t + uu_x - \frac{0.01}{\pi} u_{xx} = 0 \quad x \in [-1, 1], t \in [0, 1]$$

$$u(0, x) = -\sin(\pi x)$$

$$u(t, -1) = u(t, 1) = 0$$

where $u(t, x) : \mathbb{R}^2 \rightarrow \mathbb{R}$. Let us define $f(t, x) : \mathbb{R}^2 \rightarrow \mathbb{R}$ to be given by

$$f(t, x) = u_t + uu_x - \frac{0.01}{\pi} u_{xx}$$

followed by approximating $u(t, x)$ by a deep neural network. Following is a TensorFlow snippet that highlights the simplicity of this idea:

```python
def u(t, x):
    u = neural_net(tf.concat([t, x], 1), weights, biases)
    return u

def f(t, x):
    u = u(t, x)
    u_t = tf.gradients(u, t)[0]
    u_x = tf.gradients(u, x)[0]
    u_xx = tf.gradients(u_x, x)[0]
    f = u_t + u*u_x - (0.01/tf.pi)*u_xx
    return f
```

The shared parameters between the neural networks $u(t, x)$ and $f(t, x)$ can be learned by minimizing the mean squared error loss

$$\mathcal{L}(\theta) = \text{MSE}_u + \text{MSE}_f$$

$$\text{MSE}_u = \frac{1}{N_u} \sum_{i=1}^{N_u} |u(t^i_u, x^i_u) - u^i|^2$$

$$\text{MSE}_f = \frac{1}{N_u} \sum_{i=1}^{N_u} |f(t^i_u, x^i_u) - 0|^2$$

74
Here, \( \{t_i^u, x_i^u, u_i^u\}_{i=1}^{N_u} \) denotes the initial and boundary (cross mark in Figure 50) training data on \( u(t, x) \) and \( \{t_j^f, x_j^f\}_{j=1}^{N_f} \) specify the collocations points for \( f(t, x) \).

![Figure 50: Solved two variables function: \( u(t, x) \), courtesy of [13].](image)

**Remark 4.32.** The neural network presents function \( u \) in a map \( \mathbb{R}^2 \rightarrow \mathbb{R} \) manner, which requires an input and an output, while Figure 50 presents \( u \) in a image manner. Thereby, the neural network can get myriad samples for the training process, as the cross marks in the figure above show. For the multi-variable functions \( \mathbb{R}^n \rightarrow \mathbb{R}, n \leq 3 \), we can regard the deep neural network for solving PDE as a \( n \) dimensional “image”.

**Example 51.** The 2D Poisson’s equation reads as

\[
\Delta u = f
\]

Consequently, we can have the code snippet below:

```python
def u(x, y):
    u = neural_net(tf.concat([x,y],1), weights, biases)
    return u

def f(x, y):
    u = u(x, y)
    u_x = tf.gradients(u, x)[0]
    u_xx = tf.gradients(u_x, x)[0]
    u_y = tf.gradients(u, y)[0]
    u_yy = tf.gradients(u_y, y)[0]
    f = u_xx + u_yy
    return f
```

You can regard function \( u(x, y), f(x, y) \) happen to be meaningful images, while the neural network is another way to present this function or image.
4.10 Meta Learning

Definition 4.28. Support set is a very small set of labeled images.

Example 52. A support set is called $k$-way $n$-shot support set if there are $k$ classes of samples in the set and $n$ samples for each class. If there is only one sample for each class, then it is called one-shot learning.

Definition 4.29. Few-shot learning is the problem of making predictions based on a support set.

Remark 4.33. Few-shot learning is different from standard supervised learning. The goal of few-shot learning is not to let the model recognize the images in the training set and then generalize to the test set. Instead, the goal is “learn to learn”. The goal of training is not to know what an elephant is and what a tiger is. Instead, the goal is to know the similarity and differences between objects.

Example 53. One-shot learning is one of the extreme forms of transfer learning and meta-learning: only one labeled example of the transfer task is given for one-shot learning.

Example 54. Zero-shot learning is another of the extreme forms of transfer learning and meta-learning: no labeled example of the transfer task is given. Consider the problem of having the learner read a large collection of text and then solve object recognition problems. It may be possible to recognize a specific object class even without having seen an image of that object if the text describes the object well enough.

5 PyTorch

5.1 torch.nn

Function 5.1. L1Loss creates a criterion that measures the mean absolute error (MAE) between each element in the prediction and target.

Example 55. The unreduced (i.e. with reduction set to none) loss can be described as:

$$l(x, y) = [l_1, \cdots, l_N]^T = L,$$

$$l_n(x[n, 0, h, w], y[n, 0, h, w]) = |x[n, 0, h, w] - y[n, 0, h, w]|.$$
where $N$ is the batch size. If reduction is not ‘none’ (default ‘mean’), then

$$l(x, y) = \begin{cases} 
\text{mean}(L) & \text{if reduction= ‘mean’} \\
\text{sum}(L) & \text{if reduction= ‘sum’}
\end{cases}$$

Therefore L1Loss() is a point-wise operation (w.r.t. shape dimension), summed by batch.

**Function 5.2.** MSELoss() creates a criterion that measures the mean squared error (MSE) between each element in the prediction and target.

**Example 56.** The unreduced (i.e. with reduction set to none) loss can be described as:

$$l(x, y) = [l_1, \ldots, l_N]^T = L,$$

$$l_n(x[n, 0, h, w], y[n, 0, h, w]) = |x[n, 0, h, w] - y[n, 0, h, w]|^2,$$

where $N$ is the batch size. If reduction is not ‘none’ (default ‘mean’), then

$$l(x, y) = \begin{cases} 
\text{mean}(L) & \text{if reduction= ‘mean’} \\
\text{sum}(L) & \text{if reduction= ‘sum’}
\end{cases}$$

Therefore MSELoss() is a point-wise operation (w.r.t. shape dimension), summed by batch.

**Function 5.3.** BCELoss() creates a criterion that measures the Binary Cross Entropy between the prediction and the target.

**Example 57.** The unreduced (i.e. with reduction set to none) loss can be described as:

$$l(x, y) = [l_1, \ldots, l_N]^T,$$

$$l_n(x[n, 0, h, w], y[n, 0, h, w]) = -w_n(y[n, 0, h, w] \log(x[n, 0, h, w]) + (1 - y[n, 0, h, w]) \log(1 - x[n, 0, h, w])),$$

where $N$ is the batch size, $x, y$ are prediction and ground truth tensor, $w_n, x[n, 0, h, w]$ and $y[n, 0, h, w]$ are scalars. The channel dimension should always be 1 here, $x[n, 0, h, w] \in \{0, 1\}$, $y[n, 0, h, w] \in \{0, 1\}$. Therefore BCEwithLogitsLoss() is a point-wise operation (w.r.t. shape dimension), summed by batch. If reduction is not ‘none’ (default ‘mean’), then

$$l(x, y) = \begin{cases} 
\text{mean}(L) & \text{if reduction= ‘mean’} \\
\text{sum}(L) & \text{if reduction= ‘sum’}
\end{cases}$$

**Remark 5.1.** A Sigmoid layer after the network is necessary for both the training and testing stage, as the Sigmoid layer is not incorporated in the function.

**Remark 5.2.** For the case of $x = 0$, if either $y = 0$ or $1 - y = 0$, then we would be multiplying 0 with infinity as $\lim_{x\to 0} \log(x) = -\infty$. Secondly, if we have an infinite loss value, then we would also have an infinite term in our gradient, since $\lim_{x\to 0} \frac{d}{dx} \log(x) = -\infty$. This would make BCELoss()’s backward method nonlinear with respect to $x$, and using it for things like linear regression would not be straightforward.

Our solution is that BCELoss() clamps its log function outputs to be greater than or equal to -100. This way, we can always have a finite loss value and a linear backward method.
**Function 5.4. BCEwithLogitsLoss()** combines a Sigmoid layer and the BCELoss in one single class. This version is more numerically stable than using a plain Sigmoid followed by a BCELoss as, by combining the operations into one layer, we take advantage of the log-sum-exp trick for numerical stability.

**Example 58.** The unreduced (i.e. with reduction set to none) loss can be described as:

\[
l(x, y) = [l_1, \cdots, l_N]^T,
\]

\[
l_n(x[n, 0, h, w], y[n, 0, h, w]) = -w_n(y[n, 0, h, w] \log(\sigma(x[n, 0, h, w])) + (1 - y[n, 0, h, w]) \log(1 - \sigma(x[n, 0, h, w])))
\]

where \(N\) is the batch size, \(\sigma\) is the sigmoid function, \(x, y\) are prediction and ground truth tensor, \(w_n, x[n, 0, h, w]\) and \(y[n, 0, h, w]\) are scalars. The channel dimension should always be 1 here, \(x[n, 0, h, w] \in (-\infty, +\infty), y[n, 0, h, w] \in \{0, 1\}\). Therefore BCEwithLogitsLoss() is a point-wise operation (w.r.t. shape dimension), summed by batch. If reduction is not ‘none’ (default ‘mean’), then

\[
l(x, y) \begin{cases} 
\text{mean}(L) & \text{if reduction= ‘mean’} \\
\text{sum}(L) & \text{if reduction= ‘sum’}
\end{cases}
\]

**Remark 5.3.** When we are training the network with BCEwithLogitsLoss(), don’t forget to put a Sigmoid layer after the network in the testing stage, as the Sigmoid layer is already incorporated in BCEwithLogitsLoss().

**Function 5.5. LogSoftmax()** can be simplified as:

\[
\text{LogSoftmax}(x^{(i)}) = \log \left( \frac{\exp(x^{(i)})}{\sum_j \exp(x^{(j)})} \right) \in (-\infty, 0]
\]

**Function 5.6. NLLLoss()** is negative log likelihood loss. It is useful when training a classification problem with \(C\) classes. If provided, the optional argument weight should be a 1D Tensor assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set. The prediction is expected to contain raw, unnormalized scores for each class.

**Example 59.** The loss can be described as:

\[
l(x, y) = [l_1, \cdots, l_N]^T,
\]

\[
\text{loss}(x[n,:, h, w], \text{class}) = -w[\text{class}] \cdot \log \left( \frac{\exp(x[n, \text{class}, h, w])}{\sum_j \exp(x[n, j, h, w])} \right),
\]

where \(N\) is the batch size, \(x\) is prediction tensor, \(w[\text{class}], x[n, \text{class}, h, w]\) are scalars. The channel dimension of prediction tensor \(x\) should be \(C\), whereas there is no channel dimension in target \(y\) and \(x[n, \text{class}, h, w] \in (-\infty, 0], y[n, h, w] \in [0, C - 1]\).

**Function 5.7. CrossEntropyLoss()** combines LogSoftmax and NLLLoss in one single class. It is useful when training a classification problem with \(C\) classes. If provided, the optional argument weight should be a 1D Tensor assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set. The prediction is expected to contain raw, unnormalized scores for each class.

**Example 60.** The loss can be described as:

\[
l(x, y) = [l_1, \cdots, l_N]^T,
\]

\[
\text{loss}(x[n,:, h, w], \text{class}) = -w[\text{class}] \log \left( \frac{\exp(x[n, \text{class}, h, w])}{\sum_j \exp(x[n, j, h, w])} \right).
\]
where \( N \) is the batch size, \( x \) is prediction tensor, \( w[\text{class}], x[n, \text{class}, h, w] \) are scalars. The channel dimension of prediction tensor \( x \) should be \( C \), whereas there is no channel dimension in target \( y \) and \( x[n, \text{class}, h, w] \in (-\infty, +\infty), y[n, h, w] \in [0, C - 1] \).

**Remark 5.4.** The second dimension of \( x \) should always be the class dimension.

**Function 5.8.** `KLDivLoss()` is a useful distance measure for continuous distributions and is often useful when performing direct regression over the space of (discretely sampled) continuous output distributions. As with `NLLLoss`, the input given is expected to contain log probabilities and is not restricted to a 2D Tensor. The targets are interpreted as probabilities by default but could be considered as log probabilities with \( \log_{\text{target set to True}} \). This criterion expects a target Tensor of the same size as the input Tensor.

**Example 61.** The unreduced (i.e. with reduction set to `none`) loss can be described as:

\[
l(x, y) = [l_1, \ldots, l_N]^T,
l_n(x[n, 0, h, w], y[n, 0, h, w]) = y[n, i, h, w](\log(y[n, i, h, w]) - x[n, i, h, w])
\]

where \( N \) is the batch size, \( x, y \) are prediction and ground truth tensor, \( x[n, i, h, w] \) and \( y[n, i, h, w] \) are scalars. The channel dimension should always be 1 here, \( x[n, 0, h, w] \in \{0, 1\}, y[n, 0, h, w] \in \{0, 1\} \). Therefore `BCEWithLogitsLoss()` is a point-wise operation (w.r.t. shape dimension), summed by batch. If reduction is not `none` (default `mean`), then

\[
l(x, y) \begin{cases} 
  \text{mean}(L) & \text{if reduction= `mean`} \\
  \text{sum}(L) & \text{if reduction= `sum`} 
\end{cases}
\]

**References**


